

Model-Driven Development of Aspect-Oriented Software Architectures

Jenifer Pérez Benedí

Department of Information Systems and Computation
Polytechnic University of Valencia



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

A thesis submitted in partial fulfilment of the requirements for the
degree of Master in Software Engineering, Formal Methods and
Information Systems

Supervisors: Prof. Dr. Isidro Ramos Salavert
Dr. Jose Ángel Carsí Cubel

September 2007

This thesis has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, META (Models, Environments, Transformations and Applications) project, MOMENT (a technological framework for the model management in the Model Engineering) subproject, TIN2006-15175-C05-01.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION	15
1.1. MOTIVATION	16
1.2. OBJECTIVES OF THE THESIS	18
1.3. RESEARCH METHODOLOGY OF THE THESIS.....	19
1.4. STRUCTURE OF THE THESIS	20
CHAPTER 2 PRELIMINARIES	23
2.1. SOFTWARE ARCHITECTURES	23
2.1.1. Component	27
2.1.2. Connector	28
2.1.3. Port	29
2.1.4. Connection.....	30
2.1.5. System	30
2.1.6. Composition Relationship	30
2.2. ASPECT-ORIENTED SOFTWARE DEVELOPMENT	31
2.2.1. Base Code.....	33
2.2.2. Join Point.....	33
2.2.3. Pointcut.....	34
2.2.4. Advice	34
2.2.5. Aspect.....	35
2.3. TELE-OPERATED SYSTEMS: THE TEACHMOVER ROBOT ...	35
2.3.1. The Tele-operation Domain.....	36
2.3.2. The TeachMover Robot.....	37
2.3.2.1. The morphology of the TeachMover Robot	37
2.3.2.2. The Software Architecture of the TeachMover Robot.....	39
2.4. CONCLUSIONS.....	41
CHAPTER 3 STATE OF THE ART	43
3.1. ASPECT-ORIENTED APPROACHES AT THE ARCHITECTURAL LEVEL	44
3.1.1. PCS: The Perspectival Concern-Space Framework.....	45
3.1.2. CAM/DAOP: Component-Aspect Model/Dynamic Aspect-Oriented Platform	46
3.1.3. Superimposition.....	48
3.1.4. TRANSAT.....	49

3.1.5.	ASAAM: Aspectual Software Architecture Analysis Method	51
3.1.6.	AVA: Architectural Views of Aspects	52
3.1.7.	AspectLEDA	53
3.1.8.	AOCE: Aspect-Oriented Component Engineering	54
3.1.9.	Component Views	55
3.1.10.	Aspectual Components	55
3.1.11.	Caesar	56
3.1.12.	JASCO	57
3.1.13.	FUSEJ	57
3.1.14.	JAC	58
3.1.15.	JIAZZI	59
3.2.	COMPARISON OF ASPECT-ORIENTED SOFTWARE ARCHITECTURES	60
3.3.	CONCLUSIONS.....	66
CHAPTER 4	PRISMA BACKGROUND	69
4.1.	THE PRISMA MODEL.....	69
4.2.	THE PRISMA METAMODEL	75
4.2.1.	THE PACKAGE “TYPES”	77
4.2.1.1.	The Package “Interfaces”	77
4.2.1.2.	The package “Aspects”	78
-	The package “Attributes”	81
-	The package “Services”	83
-	The package “Constraints”	85
-	The package “Preconditions”	85
-	The package “Valuations”	86
-	The package “PlayedRoles”	87
-	The package “Protocols”	89
4.2.1.3.	The package “ArchitecturalElements”	90
4.2.1.4.	The package “Weaver”	92
4.2.1.5.	The package “Components”	94
4.2.1.6.	The package “Connectors”	95
4.2.1.7.	The package “Attachments”	96
4.2.1.8.	The package “Systems”	97
4.2.1.9.	The package “Bindings”	98
4.2.1.10.	The package “Ports”	99
4.2.2.	THE PACKAGE “ARCHITECTURE SPECIFICATION”	101
4.3.	CONCLUSIONS.....	102
CHAPTER 5	COORDINATION	111
5.1.	INTRODUCTION	112
5.2.	ASPECT ORIENTED CONNECTORS	113
5.2.1.	Non-Aspect-Oriented, connector less ADLs	113

5.2.2.	Non-Aspect-Oriented ADLs with connectors.....	114
5.2.3.	Aspect-Oriented, connector-less ADLs	115
5.2.4.	Aspect-Oriented ADLs with connectors.....	116
5.3.	CONNECTORS IN PRISMA.....	117
5.3.1.	Architectural Element.....	117
	- Formalization: Architectural Element	117
5.3.2.	Ports.....	119
	- Formalization: Ports	119
5.3.3.	Aspect.....	120
	- Formalization: Aspects	120
5.3.4.	Weavings.....	121
	- Formalization: Weavings.....	122
5.4.	ANALYSIS OF THE PROPOSAL	127
5.5.	CONCLUSIONS.....	130
CHAPTER 6	MODEL-DRIVEN DEVELOPMENT.....	133
6.1.	INTRODUCTION	133
6.2.	THE MDD SUPPORT OF PRISMA	135
6.2.1.	PRISMA in MOF	136
6.2.2.	PRISMA transformations	137
6.3.	FOLLOWING MDD WITH PRISMA CASE	139
6.3.1.	PRISMA CASE development: Domain Specific Language Tools (DSL Tools).....	140
6.3.2.	The PRISMA Type Modelling Tool: A) From the PRISMA Metamodel to the PRISMA Type Models.....	141
6.3.3.	The PRISMA Model Compiler for Types: 1)Transformation: Code and AOADL generation patterns for types	145
6.3.4.	The PRISMA Configuration Modelling Tool: B)From PRISMA Type Models to PRISMA Configuration Models	150
6.3.5.	PRISMA Model Compiler Instances: 2) Transformation: Code and AOADL generation patterns for instances	153
6.4.	CONCLUSIONS.....	154
CHAPTER 7	VERIFICATION.....	157
7.1.	INTRODUCTION	157
7.2.	VERIFICATION IN PRISMA.....	159
7.2.1.	Verification from the PRISMA metamodel.....	159
7.2.2.	Kinds of constraints.....	160
7.2.2.1.	Hardconstraints.....	160
7.2.2.2.	Weakconstraints.....	160
7.2.3.	Kinds of verification.....	161
7.2.3.1.	Partial Verification.....	161

7.2.3.2.	Complete Verification	161
7.3.	VERIFICATION IN PRISMA CASE	161
7.3.1.	Hardconstraints in PRISMA CASE.....	162
7.3.2.	Weakconstraints in PRISMA CASE	166
7.3.3.	Partial and complete verification in PRISMA CASE.....	167
7.4.	RELATED WORKS.....	168
7.5.	CONCLUSIONS.....	169
CHAPTER 8	<i>COTS: Commercial Off-The-Shelf.....</i>	171
8.1.	INTRODUCTION	171
8.2.	THE SOFTWARE ARCHITECTURE OF A TEACHMOVER'S JOINT USING COTS	173
8.3.	INTEGRATING COTS INTO THE PRISMA MODEL	174
8.3.1.	COTS as components	175
8.3.2.	COTS as aspects	175
8.4.	USING COTS DURING THE MDD PROCESS	177
8.4.1.	The use of COTS in the PRISMA CASE modeling tool.....	177
8.4.2.	The use of COTS in the PRISMA CASE model compiler	180
8.5.	RELATED WORKS.....	183
8.6.	CONCLUSIONS.....	184
CHAPTER 9	<i>THE PRISMA MDD METHODOLOGY.....</i>	187
9.1.	1ST STAGE: DETECTION OF ARCHITECTURAL ELEMENTS AND ASPECTS.....	188
9.1.1.	Identification of Architectural Elements	189
9.1.2.	Identification of Crosscutting-Concerns.....	190
9.2.	2ND STAGE: TYPE ARCHITECTURAL MODELLING	190
9.2.1.	STEP 1: Interfaces.....	191
9.2.2.	STEP 2: Aspects.....	192
9.2.2.1.	The safety aspect.....	193
9.2.2.2.	The coodination aspect	195
9.2.3.	STEP 3: Simple Architectural Elements.....	196
9.2.4.	STEP 4: Complex Architectural Elements	200
9.3.	3RD STAGE: TYPE CODE GENERATION	203
9.4.	4TH STAGE: CONFIGURATION MODELLING.....	204
9.5.	5TH STAGE: CONFIGURATION CODE GENERATION	205
9.6.	6TH STAGE: CODE EXECUTION	205

9.7.	DISCUSSION.....	206
9.8.	CONCLUSIONS.....	210
CHAPTER 10 CONCLUSIONS AND FURTHER RESEARCH.....		211
10.1.	CONCLUSIONS.....	211
10.2.	FURTHER RESEARCH	222
BIBLIOGRAPHY.....		227
APPENDIX A PRISMA CODE GENERATION PATTERNS.....		249
A.1.	INTERFACES	249
A.2.	ASPECTS	252
A.2.1.	Attributes	254
A.2.2.	Protocol	257
A.2.3.	Services.....	259
A.2.3.1.	Begin	259
A.2.3.2.	Public Services	263
A.2.3.3.	Private Services	267
A.2.3.4.	Transactions.....	270
A.2.3.5.	Checking The Service Execution	276
A.2.4.	Preconditions	278
A.2.5.	Valuations	280
A.2.6.	Constraints.....	283
A.2.7.	State of the Protocol	285
A.2.8.	Processing of a service sequence	288
A.3.	SIMPLE ARCHITECTURAL ELEMENTS: COMPONENTS AND CONNECTORS	292
A.4.	COMPLEX ARCHITECTURAL ELEMENTS: SYSTEMS	295
A.5.	IMPORTATION OF ASPECTS FROM AN ARCHITECTURAL ELEMENT	298
A.6.	WEAVINGS.....	300
A.7.	PORTS.....	303

INDEX OF FIGURES

Figure 1.	<i>The TeachMover Robot</i>	37
Figure 2.	<i>Joints of the TeachMover robot</i>	38
Figure 3.	<i>Architectural Elements of the TeachMover Software Architecture</i>	40
Figure 4.	<i>A Perspectival Concern-Space in Overview [Kan03]</i>	45
Figure 5.	<i>Superimposition [Sih03]</i>	49
Figure 6.	<i>Concern Diagram of AVA [Kat03]</i>	53
Figure 7.	<i>Unified Component Architecture [Suv05b]</i>	58
Figure 8.	<i>Crosscutting-concerns in PRISMA architectures</i>	70
Figure 9.	<i>Black box view of an architectural element</i>	71
Figure 10.	<i>White box view of an architectural element</i>	71
Figure 11.	<i>Communication between the white box and the black box views</i>	73
Figure 12.	<i>Attachments</i>	73
Figure 13.	<i>Systems</i>	74
Figure 14.	<i>Main packages of the PRISMA metamodel</i>	76
Figure 15.	<i>The package Types of the PRISMA metamodel</i>	76
Figure 16.	<i>The package Interfaces of the PRISMA metamodel</i>	77
Figure 17.	<i>The package SignatureOfService of the PRISMA metamodel</i>	78
Figure 18.	<i>The sub-packages of the package Aspects of the PRISMA metamodel</i>	79
Figure 19.	<i>The metaclass Aspect of the package Aspects of the PRISMA metamodel</i>	79
Figure 20.	<i>Constraints of the metaclass Aspect</i>	81
Figure 21.	<i>The package Attributes of the PRISMA metamodel</i>	81
Figure 22.	<i>The package KindsOfAttributes of the PRISMA metamodel</i>	82
Figure 23.	<i>The package Derivations of the PRISMA metamodel</i>	83
Figure 24.	<i>The package Services of the PRISMA metamodel</i>	84
Figure 25.	<i>The package KindsOfServices of the PRISMA metamodel</i>	84
Figure 26.	<i>The package Constraints of the PRISMA metamodel</i>	85
Figure 27.	<i>The package Preconditions of the PRISMA metamodel</i>	86
Figure 28.	<i>The package Valuations of the PRISMA metamodel</i>	87
Figure 29.	<i>The package PlayedRoles of the PRISMA metamodel</i>	88
Figure 30.	<i>The package Protocols of the PRISMA metamodel</i>	89
Figure 31.	<i>The subpackages of the package ArchitecturalElements of the PRISMA metamodel</i>	91
Figure 32.	<i>The package ArchitecturalElements of the PRISMA metamodel</i>	92
Figure 33.	<i>The package KindsOfArchitecturalElements of the PRISMA metamodel</i>	92
Figure 34.	<i>The package Weaver of the PRISMA metamodel</i>	93
Figure 35.	<i>Constraints of the metaclass Weaving</i>	94
Figure 36.	<i>The package Components of the PRISMA metamodel</i>	94
Figure 37.	<i>The package Connectors of the PRISMA metamodel</i>	95
Figure 38.	<i>The package Attachments of the PRISMA metamodel</i>	96
Figure 39.	<i>The package Systems of the PRISMA metamodel</i>	98
Figure 40.	<i>The package Bindings of the PRISMA metamodel</i>	99
Figure 41.	<i>The package Ports of the PRISMA metamodel</i>	100
Figure 42.	<i>Constraints of the metaclass Port</i>	100

Figure 43.	The package Architecture Specification of the PRISMA metamodel.....	102
Figure 44.	Sensor-Actuator coordination by using a Connector-less ADL	114
Figure 45.	Sensor-Actuator coordination by using an ADL with Connectors	115
Figure 46.	Sensor-Actuator coordination by using a connector-less Aspect-Oriented ADL (AOADL)	115
Figure 47.	Sensor-Actuator coordination by using the PRISMA ADL.....	116
Fig. 49 (a).	The black box representation	118
Fig. 49 (b).	PRISMA specification	118
Figure 48.	The RobotConnector Connector.....	118
Figure 49.	Formalization of a Service	122
Figure 50.	Formalization of a service controlled by a weaving.....	124
Figure 51.	Translation for beforeif weaving patterns	125
Figure 52.	Translation for the weaving in the RobotConnector example	127
Figure 53.	Meta-Object Facility (MOF) layers and PRISMA models	137
Figure 54.	MDD from the PRISMA Metamodel to Applications.....	138
Figure 55.	PRISMA CASE	140
Figure 56.	Toolbox of DSL Tools.....	141
Figure 57.	Definition of Architectural Elements and Aspects in the DomainModel of DSL	142
Figure 58.	PRISMA ToolBox.....	143
Figure 59.	The Visual Studio Project of PRISMA.....	144
Figure 60.	PRISMA Type Modelling Tool.....	144
Figure 61.	PRISMA Code Generation Templates	146
Figure 62.	The generated AOADL and C# code of the component Actuator.....	149
Figure 63.	Model Persistence and Configuration Language Information	151
Figure 64.	Generation and Execution of the PRISMA Modelling Configuration Tool ..	152
Figure 65.	XML document for storing instances.....	153
Figure 66.	Generic GUI of PRISMA Applications.....	154
Figure 67.	Graphical representation of an aspect in PRISMA CASE.....	163
Figure 68.	Relationship verification using graphical modelling primitives.....	163
Figure 69.	The partial C# class of the relationship ArchitecturalElementHasAspect ...	165
Figure 70.	The partial C# class of the metaclass Component.....	166
Figure 71.	Error List.....	167
Figure 72.	Verification Menu.....	167
Figure 73.	Contextual menu of an interface	168
Figure 74.	COTS as components.....	175
Figure 75.	COTS as aspects.....	176
Figure 76.	Integration of the TeachMover.dll into a PRISMA architectural model	178
Figure 77.	PRISMA architectural model of a Joint.....	179
Figure 78.	COTS Execution Process in PRISMACASE	181
Figure 79.	The C# code that is automatically generated from the IACOT aspect	182
Figure 80.	The methodology of the PRISMA approach following the MDD paradigm ..	188
Figure 81.	The ISUC interface.....	191
Figure 82.	The safety aspect SMotion.....	194
Figure 83.	The coordination aspect CProcessSUC.....	195
Figure 84.	The component Actuator.....	197
Figure 85.	The component Sensor.....	198

Figure 86.	The connector <i>SUCConnector</i>	198
Figure 87.	The system <i>SUC</i> (Simple Unit Controller).....	201
Figure 88.	The system <i>MUC</i> (Mechanism Unit Controller).....	202
Figure 89.	The system <i>RUC</i> (Robot Unit Controller)	203
Figure 90.	The architectural model of the <i>TeachMover</i>	203
Figure 91.	The configuration <i>MUC</i> for the <i>TeachMover</i>	205
Figure 92.	Reusability of aspects	208
Figure 93.	Resusability of architectural elements.....	208

INDEX OF TABLES

<i>Table 1.</i>	<i>First comparison of aspect-oriented software architecture approaches.....</i>	<i>64</i>
<i>Table 2.</i>	<i>Second comparison of aspect-oriented software architecture approaches</i>	<i>66</i>
<i>Table 3.</i>	<i>Translation set of π-processes for beforeif weaving pattern</i>	<i>125</i>

CHAPTER 1

INTRODUCTION

The work presented in this thesis of master is an approach that takes advantage of the Model-Driven Development approach for developing complex software systems. This approach improves the software quality and reduces the time and cost invested in its development and maintenance processes. It is supported by the results obtained in the thesis of Pérez [Per06c]: a model, a language, a methodology, and a Computer-Aided Software Engineering (CASE) tool prototype. The model and the tool defined in this work are called PRISMA and PRISMA CASE, respectively. The PRISMA model combines two approaches to define software architectures: the Component-Based Software Development (CBSD) and the Aspect-Oriented Software Development (AOSD). The main contributions of the model are the way that it integrates both approaches to take their advantages as well as the definition of a formal Aspect-Oriented Architecture Description Language (AOADL). The AOADL is independent of technology and is based on a formal language and formalisms that preserve non-ambiguity for applying code generation techniques.

In this thesis of master, a step forward on the work of [Per06c] is done. A complete MDD support for the PRISMA approach is defined. It follows the Paradigm of Automatic Programming [Bal85] by applying the Model-Driven Development (MDD) approach. In addition to the code generation, the MDD approach defined in this thesis of master defines

verification and reusability properties associated to the MDD process of aspect-oriented software architecture.

The structure of this chapter is as follows: Section 1 introduces the motivation of this work. Section 2 explains the main goals of this thesis of master, section 3 presents the research methodology that has been followed during the development of the thesis, and section 4 summarizes the structure of the thesis.

1.1. MOTIVATION

Complex structures, non-functional requirements, heterogeneity, scalability, traceability, reusability and maintainability are leading properties that current software systems need to deal with. In the last few years, these properties have increased the time and the staff invested in the development and maintenance processes of software. As a result, there is greater interest in research areas to reduce the time and the cost invested in these software system processes. In order to achieve the milestones of software products and to overcome the competitiveness of the market, models for the software development, techniques to improve reusability and processes to support automation, traceability and maintainability of software have been proposed.

Some new approaches have recently emerged in order to improve software development. They try to improve the early stages of the software life cycle by automating their activities as much as possible by following Model-Driven Development (MDD) [Bey05], [Am04]. MDD is a software development paradigm that is based on models that use automatic generation techniques in order to obtain the software product. MDD is included within Model-Driven Engineering (MDE) [Sch06], which increases the variety of software artefacts that can be represented as models (ontologies, UML models, relational schemas, XML schemas, etc). The use of models to develop software provides solutions that are independent of technology, whose source code can be obtained by means of automatic code generation techniques for different technologies and programming languages. The high level of abstraction that models provide permits working with metamodels in the same way as with specific models or domain-specific models.

The complexity, heterogeneity, scalability and reusability properties of current software systems have led to considering the analysis of the software structure as an important phase of the software life cycle. As a result, in the last two decades, a new research area called Software Architectures has emerged. Software architectures are presented as a solution for the design and development of complex software systems.

The Component-Based Software Development (CBSD) approach is used in the field of software architectures. This approach decomposes the software system into reusable entities called components. Components provide services to the rest of the system by encapsulating their functionality (black boxes). As a result, software architectures can be described preserving the reusability of their components.

The reusability of software allows the same software artefact to be used in different places of the same application or in different applications. The artefact is only programmed one time and can be used more than once. This reusability reduces the development time of software systems. Also, reused software artefacts guarantee their quality and suitable functionality because they have been tested and used before. As a consequence, the COTS (*Commercial Off-The-Shelf*) importation has acquired relevance, because tools that allow the reuse of their components and the COTS importation achieve the highest reuse and quality code.

Another approach that has emerged to improve reusability is the Aspect-Oriented Software Development (AOSD) approach. This approach allows for the separation of concerns by modularizing crosscutting concerns into a separate entity called aspect. As a result, the same aspect can be reused by different software artefacts, which are usually, objects.

The automatic code generation from models reduces the cost and time of the development process as well. Nowadays, there are many CASE tools that are able to generate applications following the Automatic Programming Paradigm proposed by Balzer [Bal85]. These tools are widely-known as *model compilers*. They automatically generate the application code and the database schema from the conceptual schema of a software system. The automatic generation can be complete as in Oblog Case [Ser94], OlivaNova® (OO-Method/CASE [Pas97]), or it can be partial, as in Rational Rose [RAT07], System Architect [SYS07], Together [TOG07]

and others. However, since these model compilers follow the Object-Oriented Paradigm, the need for developing model compilers that follow the CBSD and/or AOSD approaches has emerged. The combination of the CBSD and AOSD reusability and the automatic code generation achieves higher reduction in the time and cost of the development process than using only one of these approaches.

In the software life cycle, the maintenance process is as important as the development process due to the fact that the requirements of software systems are continuously evolving. The sources of these changes can be caused by several factors. First of all, the requirements specifications are inaccurate and ambiguous and these deficiencies promote misunderstandings from the very beginning of the software life cycle. An incorrect requirements specification can be produced by an inexperienced analyst, by a lack of accuracy in the presentation of the customer's needs or by a misunderstanding between the analyst and the customer because of the semantic gap in their vocabularies. This means that the software product will require continuous changes until the software that the customer really wanted is finally produced. The traceability among the different stages of the software life cycle must be preserved in order to ensure quality maintenance of software products.

An important challenge in the software engineering area is the integration of software architectures, CBSD, AOSD and MDD in a unique approach in order to support the development and maintenance of complex software systems in an efficient way.

1.2.OBJECTIVES OF THE THESIS

The main goal of this thesis is to provide a complete support for the development of PRISMA models following the MDD approach, i.e., the support for the development of technology-independent aspect-oriented software architectures. In addition, the PRISMA CASE must make this software development support feasible.

The main goal of the thesis can be divided into several specific objectives:

- To extend the related works presented in [Per06c] of the proposals that integrate the aspect-orientation approach and ADLs in order to take into account in this comparison the MDD support that these proposals provide.

- To analyze, define and formalize a coordination model that improves the reusability, maintainability and traceability of aspect-oriented software architectures.
- To define the MDD process of the PRISMA approach
- To define a verification process associated to the proposed MDD process. This verification process must be flexible and intuitive in order to be an important help for the user during the modelling stage of the MDD process.
- To introduce the use of COTS in the PRISMA MDD process. This introduction must preserve the PRISMA properties to be compliant with the PRISMA model and to maintain the PRISMA advantages.
- To define a methodology to follow the proposed MDD and verification processes and to provide support for the use of COTS.
- To modify and extend the PRISMA CASE Tool in order to support the PRISMA MDD process, its methodology, verification and the use of COTS.

1.3.RESEARCH METHODOLOGY OF THE THESIS

The research methodology that has been applied in order to fulfil the objectives proposed in this thesis follows a classical methodological strategy often called the “feasibility research strategy”. This methodology departs from a generic and conceptual hypothesis that is presented as a contribution in the area in which the thesis is developed. This hypothesis is based on a previous analysis of the state of art where the contribution of the thesis is justified. This thesis departs from the following hypothesis: Is it possible to define and implement software aspect-oriented software architectures following a Model-Driven Development Process?. In addition, the thesis departs from the results previously obtained from the thesis [Per06c] and the set of objectives that have been established in order to answer this question. From this starting point, the main goal of this thesis is to reach to a software engineering solution that copes with the set of specific objectives that have been established in section 1.2 .

1.4.STRUCTURE OF THE THESIS

The remainder of this thesis is organized in the following chapters:

➤ Chapter 2: Preliminaries

This chapter provides an introduction to the role of software architectures in the software life cycle and their main concepts. It also establishes a conceptual base for the aspect-oriented paradigm. Finally, it is introduced the case study that has been chosen to illustrate the contributions of this thesis.

➤ Chapter 3: State of the Art

This chapter extends the analysis of [Per06c] of the most relevant approaches that integrate aspects in software architectures. It includes the MDD support as part of the set of desirable properties that aspect-oriented software architecture approaches should fulfil.

➤ Chapter 4: PRISMA Background

This chapter introduce the PRISMA model and metamodel as the basis of the rest of the thesis of master. Therefore, this thesis is self-content.

➤ Chapter 5: Coordination

This chapter discuss the interest of using aspect-oriented connectors in detail, justifying the relevance of the PRISMA model and its merits with regard to other proposals, especially to provide a complete MDD support. It is described the concrete structure of connectors in PRISMA, and it is also defined the formalization of the relevant PRISMA concepts for coordination.

➤ Chapter 6: Model-Driven Development

This chapter presents the MDD proposal for the PRISMA approach and how to be supported by PRISMA CASE Tool.

➤ Chapter 7: Verification

This chapter presents how the PRISMA approach provides a complete support for the verification of aspect-oriented architectural models following the MDD approach. The verification proposal and how PRISMA CASE makes feasible this verification are presented in detail.

➤ Chapter 8: Commercial Off-The Self

This chapter presents a proposal for integrating COTS into aspect-oriented architectural models that are developed and maintained following the Model-Driven Development (MDD) approach. In addition, this chapter presents how PRISMA CASE supports the use of COTS in the PRISMA model.

➤ Chapter 9: The PRISMA MDD Methodology

This chapter presents the PRISMA methodology in order to develop aspect-oriented software architectures following the PRISMA MDD process. This methodology takes advantage of the PRISMA reusability properties (coordination model, modelling and reusability facilities, the use of COTS), the graphical specification of PRISMA models, and the verification process proposed by the PRISMA approach.

➤ Chapter 10: Conclusions and Further Research

This chapter presents the main contributions of the thesis and future research work.

➤ Appendix A: PRISMA CODE-GENERATION PATTERNS

This appendix presents the patterns that allow the transformation from models to C# code.

CHAPTER 2

PRELIMINARIES

Nowadays, software systems are becoming more and more difficult to develop due to their complex structures, non-functional requirements and distributed and dynamic nature. Two approaches of software development have emerged to overcome these needs: software architectures and Aspect-Oriented Software Development. This thesis of master is focused in the development of applications that combine these two approaches by following the MDD approach.

In this chapter, the basis that is necessary to understand the rest of the chapters is provided. As a result, an introduction about software architectures and AOSD is presented; as well as the explanation of the main concepts of both approaches. In addition, the case study that is used to illustrate the contributions of this thesis is presented.

2.1.SOFTWARE ARCHITECTURES

The complexity of current software systems has led computer community to recognize the analysis of software structure as an important phase of the software life cycle. As a result in the last decades, a new research area called *Software Architecture* has emerged to deal specifically with this phase. The software architecture discipline has emerged due to the natural increase in size and complexity of current software systems. An inaccurate architectural design leads to the failure of large software systems. For this reason, the design, specification, and analysis of the

structure of these software systems have become critical issues in software development [Gar01].

Software architectures are presented as a solution for the design and development of large, complex software systems. They allow us to describe the structure of a software system by hiding the low-level details and abstracting the high level important features [Per92]. This structure is usually represented in terms of computational elements and their interactions. As a result, software architectures make software systems simpler and more understandable [Gar95a].

The software architecture discipline is capable of performing the following functions: analyze and describe the properties of systems at a high level of abstraction; validate software requirements; estimate the cost of the development and maintenance processes; reuse software, and establish the bases and guides for the design of large complex software systems [Per92]. At the same time, software architectures should be adaptable and should provide support for the reuse of architectural elements and of partial or complete software architecture descriptions in the new software architecture specifications. Thus, new designs are not started from scratch and only the specific features of the new systems are created from the beginning [Per92].

However, despite the attempt of the IEEE to standardize the software architecture discipline [IEE00], there is no consensus about the definition of software architecture and the different concepts and approaches to be used in this field. The main drawback of this deficiency is the fact that the concept of software architecture is used in different ways and sometimes, it is really difficult to know what the exact meaning is. As a result, it is common to refer to several definitions in order to provide a complete notion of the concept of software architecture.

There are some definitions of software architecture that are general and non-exclusive; however at the same time, they are incomplete, non-explicit, and imprecise definitions. An early definition that was defined by Perry and Wolf is:

<< A software architecture is a set of architectural (or, if you will, design) elements that have a particular form.>>

Dewayne Perry and Alex Wolf [Per92]

Another modern definition that it is typically used to define software architecture is the definition presented by Bass, Clements and Kazman.

<<The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.>>

Len Bass, Paul Clements and Rick Kazman [Bass03]

However, this definition is also imprecise and incomplete because of a lot of questions emerge from this definition: For example, What is a structure?, What is an external visible property?, and so forth.

The definition of Garlan and Perry is also used to define software architecture; in fact, this is the definition proposed by the Software Engineering Institute (SEI).

<<The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.>>

David Garlan and Dewayne Perry [Gar95a]

The definition that is recommended by the ANSI/IEEE Std 1471-2000 is in essence a small variation of Garlan and Perry's definition.

<<Architecture is defined by the recommended practice as 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.>>

ANSI/IEEE Std 1471-2000 [IEE00]

There are other proposals that are not exactly definitions of software architecture but which are more specific and address the issues of the software architecture discipline. Some of them are the following:

<< Beyond algorithms and data structures of the computation; designing and specifying the overall system structure emerge as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.>>

David Garlan and Mary Shaw [Gar93]

<< An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems, and the architectural style that guides this organization--these elements and their interfaces, their collaborations, and their composition>>

Jan Booch, Rumbaugh and Jacobson [Boo99]

From all these different definitions, it is possible to conclude that there are two principle kinds of definitions: those that define the concept of software architecture and those that define the specification of software architectures. The former are characterized by being general and non-specific and the latter are characterized by being an enumeration of issues related to the description of software architectures. However, there is a common concept in both kinds of definitions; they are both concerned with the notion of structure and how to organize software.

Software architecture descriptions are specified in a formal way using ADLs. Despite the diversity of the different ADLs that have been proposed to date, all of them share a common conceptual basis. They have a common set of elements to design the structure of software systems. The elements that provide a common foundation for software architecture descriptions are introduced in this section.

2.1.1. *Component*

The concept of *component* is the basis of software architecture and the concept that ADLs share par excellence. 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. Most ADLs permit the definition of more than one interface for each component. The *interface* or multiple interfaces of a component define the functionality that the component requires and provides. In this sense, components are considered as black boxes.

The concept of component is not only used in the field of software architecture. For this reason, it is sometimes difficult to know the exact meaning of the concept, and there is no consensus about the definition of component and how to identify the components that make up a software system. There are two tendencies: one is implementation-oriented and the other is more generic. The first one covers definitions that are related to the fact that a component is a package of code [DSO99]; whereas the second one defines a component as an artefact that has been developed to be reused. This second definition is abstract and generic, and a component could be a use case, a class or another element that emerges during the development process. The definition of component in the software architecture field is also in this category. There are a lot of definitions for component; the most widely used definition in the software architecture field is the one proposed by Szyperski.

<< A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties>>

Clemens Szyperski [Szy98]

Another well-known definition is the one of Meyer based on the “seven criteria”:

<<A component is a software element that:
1. May be used by other software elements

2. *May be used by clients without the intervention of the component's developer*
3. *Include a specification of all dependencies*
4. *Include a specification of the functionality it offers*
5. *Is usable on the sole basis of its specifications*
6. *Is composable with other components*
7. *Can be integrated into a system quickly and smoothly*>>

Bertran Meyer[MEY07] [Szy00]

This “seven criteria” definition of component has been refined over time by the following one:

<<A component is a software element (modular unit) satisfying the following three conditions:

1. *It can be used by other software elements, its “clients”.*
2. *It possesses an official usage description, which is sufficient for a client author to use it.*
3. *It is not tied to any fixed set of clients.* >>

Bertran Meyer [Mey03]

Despite the fact that D’Souza advocates the module of code notion for the definition of component, he also provides a generic definition for component:

<<A component is a coherent package of software artefacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger>>

Desmond D’Souza[DSo99]

2.1.2. Connector

The concept of *connector* emerges from the need to separate the interaction from the computation in order to obtain more reusable and modularized components and to improve the level of abstraction of software architecture descriptions.

Connectors represent the interactions of software systems. They define the coordination process among components, that is, the rules that govern the interaction of components. Interfaces are the way to interact with them and these interfaces represent the roles that each one of the components plays in the coordination process.

Mary Shaw [Sha94] defines the notion of connector as follows:

<<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.>>

Mary Shaw [Sha94]

2.1.3. Port

The concept of *port* is related to architectural elements, components, and connectors. Ports are the points through which architectural elements can interact with the rest 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.

Different names have been used to refer to this concept. Some of them use the name of *port* to refer to ports of architectural elements, while other approaches use the name of *port* to refer to the ports of components and the name of *role* for ports of connectors. Other less frequently used names are *players* or *name of interfaces*.

In this thesis, it will be used the generic name of *port* to refer to both component and connectors ports.

2.1.4. Connection

Connections are used to constrain the “placement” of architectural elements; that is, they constrain how the different elements may interact and how they are organized with respect to each other in the architecture [Per92].

They establish the communication channels among architectural elements. They connect a component port with a connector port or with a port of another component [Luc95a], depending on whether the connectors are considered first-order citizens or not, respectively. In this thesis, since connectors are considered first-order citizens, a connection is established between a component port and a connector port. These connections are usually called *attachments*.

2.1.5. System

Most architectural approaches need to provide abstraction mechanisms. These mechanisms permit definition of elements of higher granularity and increase the modularity, composition, and reuse of software systems. Software composition provides flexible support and a reduction in complexity for the development process of software systems [Nie95].

These needs and advantages have led to a wide variety of architectural models and their ADLs to provide the concept of complex component. A complex component is a component that is composed by other architectural elements. 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 [And03].

2.1.6. Composition Relationship

Compositional relationships emerge with systems due to the fact that it is necessary for systems to communicate with their 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 [Gar01]. As a result, bindings establish a connection between a system port and a port of one of its architectural elements.

2.2.ASPECT-ORIENTED SOFTWARE DEVELOPMENT

The nature of current software systems has led to software being more complex, its modularity is an essential feature in being more understandable, reusable and maintainable. In addition, non-functional requirements of software systems are acquiring as much relevance as functional requirements. As a result, the support of software modularity and non-functional requirements are essential challenges to be faced in software development. The application of Software Engineering principles is necessary in order to cope with these challenges. A consolidated principle of Software Engineering is *Separation of Concerns (SoC)*, which was introduced in [Par72].

The SoC principle promotes dealing with the different concerns of a software system individually. [Dij76] demonstrated that this division provides better results and offers many advantages. A suitable application of SoC provides a reduction in software complexity and an improvement in the modularity, reuse and maintenance of software artefacts.

In the last decade, *Aspect-Oriented Programming (AOP)* has emerged as an innovative way of applying SoC in software development [Kiz97], [Elr01]. Its proposal is different from previous ones (packages, modules, classes, interfaces, patterns, etc). The concerns that are dealt with individually in AOP are those that crosscut a software system, instead of those that can be perfectly located as software units of a system. As a result, AOP introduces a new notion of *concern*. The IEEE defines concerns as:

<<...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>>

ANSI/IEEE Std 1471-2000 [IEE00]

Tarr et al. define concern as a predicate over software units [Tar99]. The *crosscutting-concerns* concept comes from this notion of concern. Software systems are usually crosscut by common concerns of a domain system. 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. The repetition of crosscutting-concerns throughout software systems increases the volume of code and complicates the maintenance that preserves the consistency of changes. Furthermore, tangled concerns make the maintenance of a specific concern more costly because it is so difficult to locate the correct place to introduce the changes. As a result, AOP proposes the separation of the crosscutting-concerns of software systems into separate entities, which are called *aspects*. This separation avoids the tangled code of software and allows the reuse of the same aspect in different software units (objects, components, modules, etc.).

AOP applies the notion of aspect to cleanly structure software systems in order to easily develop, understand, customize, evolve, and maintain software systems. It was introduced to the research community by the works of Gregor Kiczales [Kiz97], [Kiz01].

AOP separates the crosscutting concerns into aspects. It introduces the existence of two kinds of software units: components and aspects. These are clearly defined in [Kiz97] from the point of view of a non-aspect-oriented programming language:

<< A component, if it can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API). By cleanly, we mean well localized, and easily accessed and composed as necessary. Components tend to be units of the system's functional decomposition, such as image filters, bank accounts and GUI widgets.

An aspect, if it can not be cleanly encapsulated in a generalized procedure. Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects and so forth. >>

The origin of AOP is the programming language AspectJ [ASP07a]. AspectJ is currently the most widely used language for aspect-oriented programming. But, crosscutting-concerns arise throughout the software life cycle. For this reason, despite the fact that AOP emerged from the implementation level, its use is being extended to all the stages of the software life cycle. As a result, *Aspect-Oriented Software Development* (AOSD) has emerged to gain the advantages that aspects provide in every stage of software development.

AOP introduces a set of new concepts that are essential for correctly understanding this new paradigm. These concepts are introduced in this section to establish a conceptual basis for aspect-oriented programming.

2.2.1. Base Code

AOP introduces a clear differentiation between the base and aspect codes. The base code is composed of the software units (modules, objects, components) of an application, which have been obtained as a result of a functional decomposition. However, the aspect code is composed of the aspects that have been implemented to encapsulate the crosscutting-concerns of the same application.

2.2.2. Join Point

Join points are situated in the base code of an application. A *join point* is a semantic concept that defines a well-defined point of the execution of a base code. This point can extend the base code with the aspect code, thereby altering the execution flow of the original application. As a result, join points allow us to suitably coordinate the base code with the aspect code.

In the AOP taxonomy defined by [Dou05], two approaches for specifying join points are detected: one approach marks the join points using labels [Wal03], [Dan04], and the other one uses the language constructors [Col00], [Dou01], [Dou2a], [Dou02b], [Dou04a], [Dou04b]. The former introduces a pre-processing procedure that slows the code injection process down at run-time. The latter is the most widely used way of defining join points. Consequently, this is the approach that has been adopted for defining join point in this thesis. This approach usually

matches join points with method calls. The different kinds of join points that this approach uses are presented and classified in [Kiz01].

As [Kiz97] defines, a join point is not an explicit language constructor, it is the semantics of the language constructor. In other words, the join point is associated to a language constructor; however, the different instantiations of this constructor will be different join points at run-time. A clear example is a joint point that is associated to a method call. The different invocations of this method are different join points with different semantics. The semantics is different because it depends on the object that has invoked the method, on the instantiation of its arguments, and so forth.

2.2.3. *Pointcut*

From the previous section, it can be deduced that an application has a large quantity of join points in its base code. However, not all join points of the application are interesting or relevant for injecting aspect code. As a result, the relevant join points for this injection of aspect code must be selected. The mechanism that permits this selection is the *pointcut*.

A pointcut is a set of join points, which are candidates for injecting aspect code into base code at run-time, and their multiple instantiations. Pointcuts perform the weaving between the base code and the aspect code by capturing joinpoints.

The most widely used pointcut model is the AspectJ model [Dan04], [Jag06a], [Jag06b], [Läm02], [Wal03], [Wan04]. There are also other pointcut models that use more general execution patterns, such as stack of events, tree of events, sequence of events, etc [Col00], [Dou2a], [Dou04a], [Mas03].

2.2.4. *Advice*

An *advice* defines the code that should be executed at the join points of a specific pointcut. Advices define additional code for the join points that have been selected by pointcuts. As [Bru04] cites, the advice code is the profiling code of an aspect-oriented program and the compiler profiles the join points by executing the advice code at run-time. This process by the compiler consists of inserting or replacing the base program code is called *weaving*. The execution of code depends on the kind of advice. There are three main kinds of advice:

- **Before:** The before advice adds code to the base program before the join point. As a result, the code of the advice is executed before the code of the join point.
- **Around:** The around advice substitutes the code of the join point. As a result, the code of the advice is executed instead of the code of the join point.
- **After:** The after advice adds code to the base program after the join point. As a result, the code of the advice is executed after the code of the join point.
 - **After returning:** This kind of after advice is executed when the execution of the join point finishes correctly.
 - **After throwing:** This kind of after advice is executed when the join point throws an exception. As a result, the code of the advice is executed after the throwing of the join point.

As [Wan04] concludes, an aspect-oriented program is composed of a base program and some advices.

2.2.5. *Aspect*

An *aspect* is a language constructor that encapsulates a *crosscutting-concern*. An aspect is linked to one or more methods of the base code by means of pointcuts. For this reason, an aspect is composed of pointcuts and an advice. As a result, aspects specify whether their execution will be before, after or around a method of the base code by means of the advice that is associated to pointcuts. Despite the fact that aspects are usually pairs of pointcuts and advices, they can have their own state [Dou05]. The definition of aspect proposed by Kizcales is the following:

<< Aspects are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations. >>

Gregor Kizcales et al. [Kiz01]

2.3. TELE-OPERATED SYSTEMS: THE TEACHMOVER ROBOT

There is a wide variety of domains that can take advantage of the PRISMA approach for developing software systems. PRISMA has been put into practice in the tele-operation domain,

which unlike academic examples, provides real problems that must be solved in real industrial systems. This domain has been chosen because it offers a framework for applying software engineering techniques.

The main purpose of this section is to provide an introduction to the tele-operation domain as well as to present the suitability of tele-operation systems to apply an aspect-oriented software architecture approach following the MDD paradigm. In addition, the specific robot that has been completely developed using PRISMA MDD proposal is presented. This robot is used throughout the thesis in order to illustrate the PRISMA MDD approach and its main concepts.

2.3.1. The Tele-operation Domain

Tele-operation systems are control systems that depend on software to perform their operations. Designing these systems is a difficult task that must integrate mechanical and electrical elements with software components in the same system. They are used for tele-operating mechanisms (robots, vehicles, and tools) that handle inspection and maintenance tasks. This thesis focuses specifically on robotic tele-operated systems. Tele-operated robots are software intensive systems that are used to perform tasks that human operators cannot carry out due to the dangerous nature of the tasks or the hostile nature of the working environment.

The importance of considering the software architecture in robotic tele-operated systems is well known [Cos00]. However, despite the fact that robotic tele-operated systems usually have many common requirements in their definition and many common components in their implementation, it is impossible for a single architecture to be flexible enough to cope with all the variability of the domain. Therefore, a further step is needed to provide a flexible and extensible architectural framework to develop systems with different requirements and commonalities. There have been numerous efforts to provide developers with frameworks such as [Bru02], [Sch01] and [Vol01]. All of them make very valuable contributions that simplify the development of systems. However, the way that the component-oriented approach has been applied may reduce some of its benefits. These frameworks are object-oriented or component-oriented frameworks that rely on object-oriented technologies and that highly depend on a

given infrastructure (Linux O.S. and the C++ language). As a result, a technology-independent framework that will follow MDD is necessary. This framework should provide mechanisms to define abstract software architectures that can be mapped into specific software architectures as well as mechanisms to dynamically evolve the interaction patterns among components. In addition, tele-operated systems have a wide range of common concerns in their domain. These concerns can be modelled as aspects in order to take advantage of AOSD. Some of these candidate aspects of the tele-operation domain are distribution, safety, mobility, security, coordination, etc.

Most robotic tele-operated system has strong requirements in terms of adaptability to different devices, operator safety, response time, dynamic reconfiguration, etc. As a result, PRISMA has been applied to these kinds of systems in order to cope with these requirements. A complete development of a small-scale robot has been done. This robot is called *TeachMover* [TEA07].

2.3.2. *The TeachMover Robot*

The *TeachMover* robot is a robotic arm that is frequently used to teach the fundamentals of robotics (see Figure 1). This robot was specially designed for the purpose of simulating the behaviour of large and heavy industrial robots.



Figure 1. The TeachMover Robot

2.3.2.1. *The morphology of the TeachMover Robot*

The *TeachMover* is formed by a set of joints that permit the movement of the robot. These joints are: *Base*, *Shoulder*, *Elbow* and *Wrist*. In addition, it has a *Tool* to perform different tasks

(see Figure 2). The movements that the robot is able to perform are: the rotation of the robot using the base, the articulation of the elbow and shoulder joints, and the rotation of the wrist. In this case, the *Tool* is a gripper, whose open and close actions allow the robot to pick up and deposit objects.

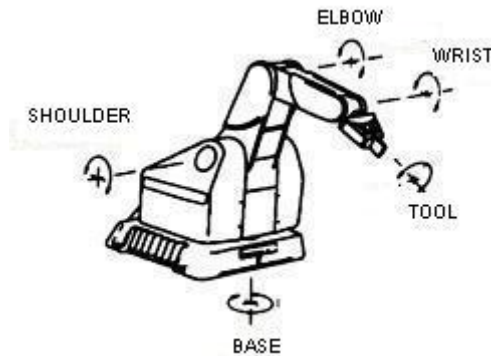


Figure 2. Joints of the *TeachMover* robot

The robot has six electric step motors for driving the direction of the movements of each joint. These motors perform the movements through gears that are joined by a cable system. The *TeachMover* can be moved at a specific speed by means of *half-steps* or inverse cinematics. A half-step movement moves the robot using the number of teeth that a gear of a joint must be moved as a measure. And, an inverse cinematic movement moves the robot using a specific point in the space as a measure. These features, together with the features of the gripper, allow the robot to move objects from an initial position to a final one.

In addition, safety directives of the robot require its movements to be checked to make sure that they are safe for the robot and the environment that surrounds it. The internal safety of the robot is preserved by establishing a set of constraints that forbid certain movements that will break the gears of the robot due to the position of the gears inside the robot. These constraints are defined by establishing minimum and maximum values for the movements of each joint. These values are specified in degrees as follows:

- Base: $\pm 90^\circ$
- Shoulder: $+ 144^\circ, - 35^\circ$
- Elbow: $+ 0^\circ, -149^\circ$
- Gradient of the Wrist: $\pm 90^\circ$
- Rotation of the Wrist: $\pm 180^\circ$
- Opening of the gripper: 0 inches, + 3 inches (7,62 cm.)

The robot has a sensor to pick up objects without breaking them. The weight of the objects that the robot is able to carry when its arm is stretched out is 450 grammes. Furthermore, the gripper presses the objects with a maximum pressure of 14 Newtons. Finally, the speed of movements fluctuates between 0 or 7 inches per second (178 mm/s) depending on the load that the robot carries when the movement is performed.

It is important to mention that the movements of the robot are commanded by an operator from a computer. This communication between the computer and the robot is possible by means of the serial/RS232C port. In order to stop the robot in situations of emergency, the robot has an interruption mechanism for disconnecting the power of the robot by means of software. This is possible because this interruption mechanism is connected to the parallel port of the computer.

All these features allow the TeachMover robot to simulate the movements of most of the industrial tele-operated robots that are currently in use. This robot allows the testing and verification of new solutions to be applied to more complex robotic systems in the future.

2.3.2.2. *The Software Architecture of the TeachMover Robot*

The *TeachMover* architecture has different levels of abstraction for its components, connectors and the interactions among them. The lowest abstraction level of the robot architecture has sensors and actuators as basic components, which are communicated with the hardware joints of the robot. The functionality of the actuators and sensors are the following:

- **Actuator:** An actuator sends commands to a joint of the robot. These commands are performed by the joint or the tool.

- **Sensor:** A sensor reads the results of the commands in order to know whether or not they have been performed successfully.

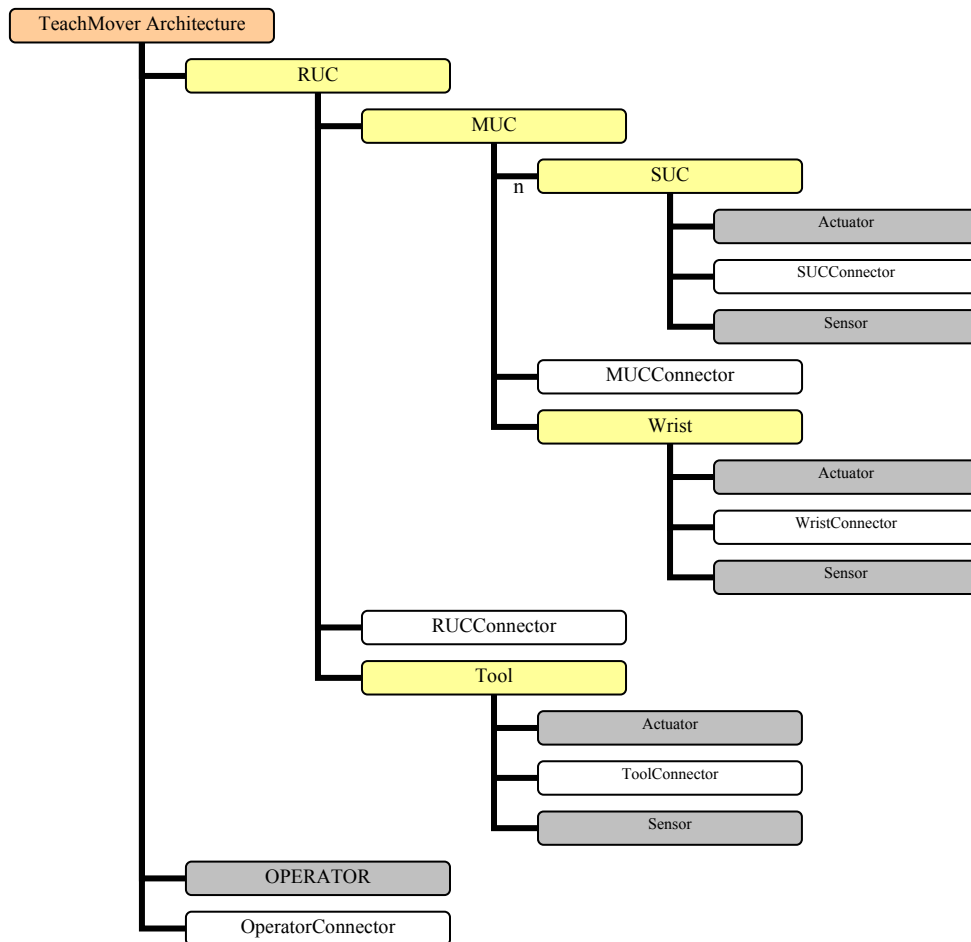


Figure 3. Architectural Elements of the TeachMover Software Architecture

An actuator and a sensor are coordinated by means of a connector. These three architectural elements (*actuator*, *sensor*, and *SUCconnector*) are encapsulated inside a complex component called the *Simple Unit Controller (SUC)* (see Figure 3). However, two special SUCs have been identified in order to take into account the peculiarities of the wrist joint and the tool. The *SUC*, the *Wrist SUC* and the *Tool SUC* must be composed and coordinated in order to form the

complete structure and functionality of a tele-operated robot. This composition generates different levels of granularity (see Figure 3):

- **Mechanism Unit Controllers (MUCs):** This architectural element type represents the arm of the robot, which is composed of the SUC and Wrist SUC coordinated by means of a connector.
- **Robot Unit Controllers (RUCs):** This architectural element represents the robot, which is composed of MUCs and Tool SUCs coordinated by means of a connector.
- **The Architectural Model:** This level represents the interactions between operators and robots through a connector.

2.4.CONCLUSIONS

This chapter has detailed the preliminary notions of this thesis of master. The origins of AOP and software architectures and their main concepts have been presented. This introduction is necessary in order to understand the PRISMA approach, and as a consequence, the MDD process of PRISMA that is proposed in this thesis.

In addition, the robotic tele-operated systems are presented. Robotic tele-operated systems have been chosen as application domain for the PRISMA MDD approach, since these provide real systems that need real solutions for their development and maintenance processes. Specifically, PRISMA has been applied to the *TeachMover* tele-operated robot, which is also used in this thesis to illustrate the PRISMA MDD approach as well as its methodology.

CHAPTER 3

STATE OF THE ART

The relevance that non-functional requirements have acquired in current software systems has led to the emergence of crosscutting concerns in software architectures. These crosscutting-concerns are spread throughout software architectures.

There is a wide variety of ADLs that have been proposed in order to specify software architectures, such as ACME [Gar00], Aesop [Gar94], [Gar95b], C2 [Med96], [Med99], Darwin [Mag95], [Mag96], MetaH [Bin96], [Ves96][Bin96], Rapide [Luc95b], [Luc95a], SADL [Mor95], [Mor97], UniCon [Sha95] [Sha96], Weaves [Gor91], [Gor94] and Wright [All97a],[All97b]. An interesting comparison with respect to these ADLs is presented in [Med00]. In addition to this work by Medvidovic and Taylor, there are other interesting surveys on ADLs such as the ones presented in the PhD. Thesis by Cuesta [Cue02], which covers the analysis of other ADLs such as Conic [Kram85], [Mag89], DURRA [Bar01], AML [Wyd01], and Armani [Mon98]. It is also important to mention other approaches that are especially prepared to support evolution in software architectures, such as CommUnity [And03], [Fia04], LEDA [Can00], Pilar [Cue02], GUARANA [Oli98] or R-RIO [Loq00].

These ADLs do not explicitly distinguish the conventional architectural elements from concerns that crosscut multiple architectural elements of software architectures. One of the few approaches that deals with the separation of concerns is the work by Jose Fiadeiro. This work addresses the separation of distribution, mobility [Fia04], contex-awareness [Lop05] and coordination [And03] in software architectures. However, none of these original ADLs supports the separation of concerns by means of the aspect-orientated approach at the

architectural level. For this reason, several approaches have emerged to cover this need either by extending original ADLs or by creating new ADLs from scratch.

There are some approaches that extend MDD by providing AOSD mechanisms during their different stages [Aks05], [Kul03]. Other works that are related to the development of aspect-oriented applications following MDD are [Sim05] and [Ama05]. However, none of these aspect-oriented approaches take into account software architectures.

The combination of AOSD and software architectures has created two new challenges: how to define the concept of aspect at the architectural level and how to integrate aspects and architectural elements in a suitable way. A few of these approaches that combine AOSD and software architectures give support for a complete development of software. In fact, they do not provide a complete MDD support for developing aspect-oriented software architectures.

In this chapter, the analysis presented in [Per06c] is extended by taking into account the MDD support that provide the most relevant approaches that deal with AOSD and Software Architectures. In this case, the analysis is also made by starting from the premise that an aspect-oriented software architecture approach should completely support the development and maintenance processes of software following the MDD approach. As a result, the set of desirable properties that aspect-oriented software architecture approaches should fulfil are also extended by introducing the MDD property. Finally, the comparison of these approaches using these new properties is presented and discussed.

3.1.ASPECT-ORIENTED APPROACHES AT THE ARCHITECTURAL LEVEL

The incorporation of aspects at the architectural level implies considering what an aspect is at this level. An aspect is a new entity for modularizing and encapsulating specifications in software architectures. As a result, it is necessary to define how aspects are related to the rest of the main concepts of software architectures, especially to components and connectors. It is also necessary to define the kind of relationships (reference, connection, composition, etc) that they have with these elements.

In this section, the most important works of the area are analyzed paying special attention to the way that they introduce the notion of aspect in software architectures, how they coordinate aspects and architectural elements, their MDD support and their main properties. Due to the fact that there has not been much work done at the architectural level, not only are ADL extensions analyzed, but also aspect-oriented component models that could be applied at the architectural level.

3.1.1. PCS: The Perspectival Concern-Space Framework

The Perspectival Concern-Space (PCS) [Kan03] approach is based on the MDSOC model [Har03] and IEEE-Std-1471. It uses UML for modelling concerns at the architectural level. PCS describes concerns by means of architectural views. These views consist of one or more models and one or more diagrams. A perspective in PCS is defined as “a way of looking” at a multidimensional space of concerns from a specific viewpoint. As a result, this approach defines a perspectival concern-space as a projection of a concern-space that involves a set of related concerns, their reifications into models, and the realization of these models (see Figure 4).

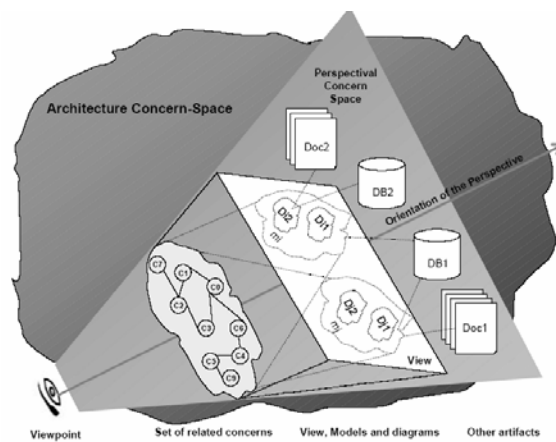


Figure 4. A Perspectival Concern-Space in Overview [Kan03]

The PCS approach uses UML to specify aspect-oriented software architectures, and it extends UML by defining a profile that supports the modelling of aspects and components. The

profile simulates aspects by means of architectural connectors based on the idea that aspects act as coordinators among components to intercept their interactions and then replace or add behaviour either before or after them [Kan02b]. As a result, PCS is based on an original ADL without connectors, whose aspect-oriented behaviour is introduced by means of connectors. Components and aspects are modelled by means of UML classes that have been profiled in order to have ports through publishing services. In addition, aspect classes are distinguished by component classes by means of the `<<aspect>>` stereotype [Kan02a]. A disadvantage of this combination of aspects and software architectures is the loss of the advantages that connectors provide to ADLS and the opportunity to specify how concerns crosscut the coordination rules of connectors.

The PCS approach is supported by the ConcernBase tool [Kan03]. This tool provides mechanisms for modelling software systems, and it also allows the translation from UML models to the SADL language [Mor97].

Technological independence is a clear advantage that this approach offers. Yet, at the same time, it is a drawback of PCS because it does not provide support to translate its models to a programming language or to trace from models to implementation. As a result, PCS supports MDD in a partial way. It provides mechanisms for modelling software systems, and it also allows the translation from UML models to the SADL language, but it does not translate its models to a programming language in order to be executed on a technological platform.

3.1.2. CAM/DAOP: Component-Aspect Model/Dynamic Aspect-Oriented Platform

CAM/DAOP is an approach that supports the separation of concerns from the design to the implementation stages of the software life cycle. It is composed of the CAM model, the DAOP-ADL [Pin03], and the DAOP platform [Pin05].

The CAM model extends UML in order to specify the components, the aspects and the mechanisms that compose components and aspects. Components are the core functionality that is crosscut by non-functional concerns, which are specified as aspects. In CAM, aspects are presented as special components, which are differentiated from the original ones by means of the `<<aspect>>` stereotype. Specifically, since its component model does not have the notion of

connector (see section 2.1.2), CAM introduces aspects as special connectors among components. The coordination of these connectors is performed by intercepting the services that arrive to or depart from components and by adding behaviour before, after or instead of their services. As a result, CAM does not introduce a new concept in software architectures for modelling aspects; it uses a refined version of the connector concept in order to simulate the behaviour of aspects and the composition of aspects and components. One of the advantages of this model is the fact that the weaving process between aspects and components is defined by means of interfaces. As a result, the encapsulation and reusability of components and aspects are preserved. In addition, this allows CAM to define the weaving process using the interfaces and also to execute it dynamically [Fue05]. However, the model does not provide original connectors to specify the architecture of the system. As a result, the model loses the advantages that connectors provide to ADLS and the opportunity to specify how concerns crosscut the coordination rules of connectors. Finally, it is important to emphasize the local or remote instantiation of components and the four kinds of instantiation that the CAM model provides for aspects: a single instance for each aspect, one aspect instance for each user of the system, one aspect instance for all the components that play the same role, and one aspect instance for each instance of a component.

CAM specifies aspect-oriented software architectures using its DAOP-ADL. This ADL uses XML to describe components, aspects, and their interactions. On one hand, this is an advantage because it is a standard of data exchange between tools, it is widely extended and there are other languages that support query and management mechanisms for XML documents. On the other hand, this is a disadvantage because XML is not a formal language, and it can involve problems of correctness, accuracy, inconsistency, etc. In addition, it introduces limitations such as mechanisms to validate properties, to automatically generate code without ambiguity, etc. Finally, with regard to the DAOP-ADL, it is important to emphasize that it is independent of technology. As a result, their specifications do not introduce expressions or syntaxes of specific programming languages or technologies.

The DAOP platform has been implemented in Java, and it provides a middleware in order to support the execution of aspects, components, and the dynamic weaving between them over

the Java technology. The platform and the DAOP-ADL specifications are integrated because the input of the DAOP platform is the XML document that contains the specification of the architectural model in XML. As a result, the middleware can perform the dynamic weaving since it knows all the information about the architectural model and knows the weavings that can be executed by each one of the aspects. The middleware performs the weaving by intercepting service requests and determining which aspect must be executed. In addition, the XML document contains the information needed to instantiate components and aspects. For this reason, when the document is loaded by the DAOP platform, the instantiation of components and aspects starts taking into account the instantiation information defined in the document.

The work [Fue03] of CAM/DAOP is a first step to support MDD in the DAOP platform, however a complete support using code generation techniques for the development is not provided. They use MDA to show the different views of the models that are specified in the platform.

3.1.3. Superimposition

The work of Sihman and Katz proposes the use of superimposition for incorporating aspects into object-oriented programs. The generic operation of superimposition consists of applying a concept on top of another one. In this approach, aspects are superimposed on top of base applications. This approach creates the SuperJ constructor in order to pre-process aspect-oriented superimpositions over AspectJ.

A superimposition consists of a set of aspects and new classes that represent the extension of an application. A SuperJ implements an algorithm to apply a superimposition to a base application. Base applications do not reference superimpositions, and superimpositions can also be defined and compiled independently of base applications. However, when a superimposition is connected to a base application using the SuperJ constructor, the code of the superimposition makes reference to the state of the base application and it is not independent (see Figure 5). In fact, the needed advices and pointcuts are defined inside the aspects of a superimposition. As a result, the behaviour of the aspect and its connections to the base

program are not defined separately. Despite the fact that the specification of an aspect is done at a high abstraction level, the specification of advices and pointcuts inside aspects reduces the capabilities of aspect reuse of the approach. In addition, superimposition allows the specification of conditions of applicability or the definition of desired results for the process of applying a superimposition to a base application.

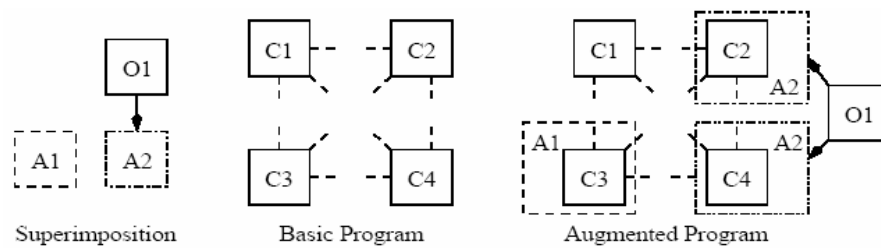


Figure 5. Superimposition [Sih03]

This approach allows us to combine superimpositions and to check the constraints that have been defined in a superimposition. It has been implemented using Java, and it uses AspectJ in order to apply this technique over Java base applications. As a result, the Java implementation of the SuperJ makes this model dependent on technology and it is closer to object-oriented programming languages than ADLs.

3.1.4. TRANSAT

Transat [Bar04b] is an approach for managing the evolution of software architecture specifications using aspect-oriented programming principles. The approach starts from a core architectural model that either needs to be extended during its development process or needs to be evolved during its maintenance process. The mechanisms of extension and evolution are provided using AOP techniques. As a result, this approach incrementally obtains a complete software architecture with business and technical concerns from a business software architecture.

This approach is supported by a framework that allows the evolution of software architectures by integrating new technical concerns. The framework guides the separated definition of technical and business concerns. Business concerns are the core architectural

model, and technical concerns are the aspects that extend the basic functionality of the system. The framework provides aspect-oriented mechanisms to weave both.

The core architectural model is defined using its component model, SafArchie [Bar03]. This model is a hierarchical component model that defines software architecture by means of composition relationships. The new technical concerns such as persistence, security, or transaction management are modelled as components. Finally, the weavings between business components and aspect components are defined by means of adapters and weavers. Adapters define the integration rules between technical components and business components, and weavers define the coordination rules between them. In other words, adapters and weavers materialize the integration of the core architecture and their extensions by identifying the join points in the core architecture and by defining the pointcuts at adapters and weavers.

The Transat framework consists of a tool called SafArchie Studio [Bar04a]. This tool is an extension of ArgoUML, which offers several views of the evolution process depending on the kind of user.

One of the main advantages of this approach is that it is based on ADL for defining the core architectural model. As a result, the formal definition of software architecture and its independence from a technological platform are guaranteed.

Another advantage is the way that evolution is supported. The integration of new requirements does not break the consistency of the original software architecture. In addition, the application of AOP principles to this integration ensures both the separation of concerns in the software architecture extension and better management if new requirements for these concerns arise. Finally, this extension mechanism allows analysts to easily identify where the original software architecture has been modified; they only need to find the adapters and weavers of the complete architecture.

A great limitation of this approach is the constraint of starting the development from a core architectural model without considering concerns from the beginning. Also, the fact that concerns are only technical and not more generic is another drawback. As a result, aspects in this approach are not introduced as a new concept for modelling software architectures.

Software architectures are defined using a pure compositional ADL, and aspects only appear as an extension or evolution mechanism of software architectures.

Since Transat is only focused on the evolution and maintenance stages of software, it does not provide a complete MDD support. Its tool only allows to analyze the evolution of the software architecture, and not to develop the aspect-oriented software architecture application following MDD.

3.1.5. ASAAM: Aspectual Software Architecture Analysis Method

ASAAM [Tek04] is the approach that introduces aspect-orientation techniques to the SAAM approach, which introduces three perspectives to analyze software architecture specifications: functionality, structure and allocation. As a result, ASAAM is an extension and refinement of SAAM. The steps of ASAAM are the following:

1. **Develop a candidate architecture:** A candidate architecture is generated taking into account quality attributes and potential aspects.
2. **Develop scenarios:** The scenarios that define the business rules of the system and possible future changes are created.
3. **Perform scenario evaluations:** The scenarios are evaluated and categorized, and potential aspects are identified for each scenario.
4. **Assess scenario interaction and classify components:** The separation of concerns is assessed for both crosscutting-concerns and non-crosscutting concerns.
5. **Refactor the architecture:** A refactorization of the architecture is proposed using conventional techniques and aspect-oriented techniques.

This evaluation method of aspect-oriented software architectures has been implemented as an Eclipse add-in called ASAAM-T [Tek05]. The main difference between this approach and the others is the fact that the purpose of ASAAM is to assess an aspect-oriented software architecture instead of specifying and implementing a software architecture. As a result, this approach is a valuable contribution to the field for evaluating if an aspect-oriented software architecture has considered the correct aspects, and if the aspects are factorized in a proper way.

3.1.6. *AVA: Architectural Views of Aspects*

AVA is an approach where aspects are introduced in software architectures as views [Kat03]. The notion of aspect in software architectures is simulated by the architectural view concept. This facilitates the comprehensibility of the model for the software architecture community. However, an aspect is not semantically a view because an aspect has its own behaviour independently of the architectural elements that it affects. Furthermore, this approach constrains the notion of architectural view to an aspect, losing other viewpoints for defining views such as kinds of users, models, level of abstraction, features, etc.

In AVA, an aspect is a module that encapsulates a set of components and their connections that are crosscut by this aspect. As a component can be crosscut by more than one aspect, the dependencies between different aspects must be explicitly specified in order not to lose the consistency of the software architecture. Since it is possible to define several aspects for the same concern in AVA, aspect modules (S,O) can be composed to form a single concern module (C) (see Figure 6). This aspect composition is performed using superimposition, which is an asymmetric operation in which one aspect is applied on top of another one [Kat02]. As a result, the software architecture is completely remodularized in different modules that represent aspects or concerns, depending on the level of abstraction. This modularization distributes components into different modules taking into account the aspects that affect them. The main disadvantage of this remodularization is the loss of the complete software architecture view. However, this concern and remodularization structure of modules allows analysts to easily locate where to introduce new changes, taking into account the concerns that should be modified.

The AVA model has been created by defining a UML profile. As a result, the definition of AVA software architectures is really intuitive because these architectures use the OMG standard. In addition, the use of UML allows analysts to specify software architectures independently of technology and in a graphical way. In the AVA profile, the aspect is a stereotype of the package UML metaclass, and the concern diagram is an extension of the component diagram.

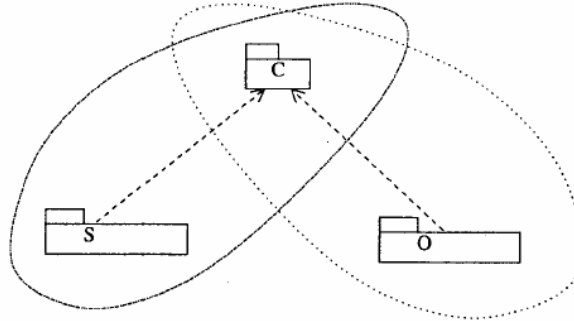


Figure 6. Concern Diagram of AVA [Kat03]

The AVA approach has also been applied to the definition and documentation of pattern systems [Ham05], and the MADE tool has been developed to support it [Ham04]. This tool shows the different views of the architecture, but it does not provide a complete MDD support.

3.1.7. *AspectLEDA*

AspectLeda is an approach that extends the LEDA ADL [Can00][Can99] with aspect-oriented concepts [Nav05]. This approach consists of two steps: the definition of an initial architectural model and the addition of aspects. The initial architectural model is defined using the LEDA ADL in order to have the advantages of a formal basis, to validate the software architecture by executing a prototype, and to be independent of technology. Once the initial architecture has been defined, the new requirements that emerge during the development and maintenance processes are incorporated in the architecture such as aspects. This is a clear drawback of this approach because it does not give the analyst the chance to introduce aspects at the beginning of the software development process. Aspects are only used at the maintenance stage or at refinement processes of the development stage. In addition, it is important to take into account that not all new requirements of a system are aspects. However, AspectLEDA forces the analyst to introduce new requirements into the model as new aspects without taking into account whether they are aspects or not.

In AspectLEDA, aspects are specified in the way as components because LEDA is an ADL without connectors. However, aspect components and components of the initial software

architecture are defined in different levels. Since AspectLEDA does not have architectural connectors, it cannot specify the concerns that crosscut connectors, and it loses the advantages that connectors provide to ADLs. However, AspectLEDA introduces the notion of coordinator to define the weaving process that synchronizes aspects and components and coordinates both levels. This coordinator preserves the reusability and encapsulation of aspects because the coordination of aspects and components is specified outside aspects.

Finally, it is important to emphasize that this approach is still only a proposal. It does not have a tool to support for its methodology, and it is not able to compile its aspect-oriented software architecture into any technological platform.

3.1.8. AOCE: *Aspect-Oriented Component Engineering*

The Aspect-Oriented Component Engineering approach (AOCE) is based on AOREC (Aspect-Oriented Requirements for Component-Based Systems). AOREC uses the notion of aspect in order to suitably define and categorize the requirements of components in terms of what they provide or require through their services. In AOREC, an aspect is a characteristic of a system for which components provide and/or require services. This approach takes into account some aspects such as user interface, collaboration, persistence, distribution, and configuration. As a result, AOREC uses aspects in order to attain multiple perspectives of the components in order to better understand and reason about the behaviour and semantics of these components.

Since AOCE is the step after AOREC in the development process, AOCE defines aspects in the same way that AOREC does; aspects are specified as components. The definition of an aspect component is done separately from the component specification in order to be independent and reusable. However, an AspectManager must be introduced in order to coordinate aspect components and components. Despite the fact that this approach does not have connectors (because is a component-based approach), it still must introduce a connector called AspectManager to weave aspects and components at run-time.

Apart from not having the notion of connector and not classifying the interfaces of connectors in terms of aspects, the main disadvantage of this approach is the fact that the design

language of AOCE is based on a specific component-based platform, the JViews [Grun98] AOP implementation platform, which is not independent of technology.

Finally, it is important to mention that AOREC and AOCE have a tool to support their methodology. They have extended the tool of JViews to support aspects. This tool is called JComposer [Grun98]. It is an implementation framework used by developers; it only provides support for programming. As a result, it does not support for MDD.

3.1.9. *Component Views*

The component views approach [Sto02] is not really an approach to specify aspects of software architectures. The component views approach is an extension of component-based models to define views using concerns as viewpoints. As a result, this approach decomposes the architecture taking into account which components and connections among them are affected by a specific concern. The result of this decomposition is that each view of a concern contains the components and relationships affected by this concern. This approach defines a UML profile to support the definition of these views for software architectures. However, this approach does not introduce new concepts or simulates the notion of aspect because it does not support this notion. It only works with concerns which are only used to analyze component-based models. Their specification is made using a UML profile. The purpose is not to execute an aspect-oriented component-based model, it is simply to analyze component models.

3.1.10. *Aspectual Components*

Aspectual Components are proposed as a new kind of component by the work of Lieberherr [Lie99]. They are defined using a generic data model called a participant graph. This graph is then refined to deploy aspects as normal components.

This approach proposes adding a new dimension to aspects over the organization of an object-oriented application. As a result, the first task of the software development process is to decompose software into aspects. The second one is to decompose each aspect into classes following the object-oriented approach. The result of this process is an aspectual component composed of object-oriented classes. However, these aspectual components should be composed with the application base. In other words, the new dimension of aspects must be

communicated to the bottom dimension of the application. This communication is achieved by means of connectors that coordinate both dimensions.

Aspectual components can be programmed using Java programming language because this approach does not introduce a new programming constructor; instead, aspectual components are implemented as normal components. However, this approach does not offer a tool to support work with this model.

3.1.11. Caesar

Caesar is a model for aspect-oriented programming [Mez03] with its own programming language. This model is characterized by being technology-dependent and by developing a higher-level module to develop aspects independently of the mechanisms for join point interceptions.

Caesar specifies the implementation of aspects and their weaving relationships in a separate way in order to reuse the aspect independently of what the aspect is related to. The main feature that distinguishes Caesar from other aspect-programming models is the concept of Aspect Collaboration Interface (ACI). An aspect is specified by means of an ACI. An ACI decouples the implementation and weavings of aspects. An ACI is composed of two different interrelated modules: an implementation aspect module and a binding aspect module. The former implements the methods that the aspect provides, independently of the context. The latter implements the required methods from a specific context by means of pointcuts and advices. In addition, an ACI can be composed of other ACIs, which provide a complete level of composition in order to define aspects over the base code.

Caesar defines an instantiation mechanism for its ACIs. To instantiate ACIs, the implementation and a specific binding for the aspects must be composed in the same unit; this unit is called *weavelet*. A weavelet is a class that is composed of the interface that provides and requires. Once, the weavelets are defined; they can be instantiated in a static or dynamic way. This mechanism of instantiation allows several instances of the same weavelet to be defined. It also provides a choice of different weavelets using aspectual polymorphism.

3.1.12. JASCO

JAsCo is originally an aspect-oriented programming language for the Java Beans component model [Suv03]; however, a prototype for .NET platform is currently being developed [Ver03]. As a result, JAsCo is a programming language that is dependent on technology. It introduces three new concepts to extend Java to support aspect-oriented programming, which include aspects, hooks, and connectors.

In JAsCo, aspects are composed of a set of hooks that define how to link an aspect to a specific context. A hook consists of two parts: the pointcut (when the hook is activated) and the advice (what is going to be executed as a result of the activation). Finally, connectors allow the definition of the mappings between a hook and one or more elements of the base code (joinpoint and pointcut correspondence).

The JAsCo execution model is very flexible and provides many advantages. It supports aspectual polymorphism, which is the weaving between aspects and code. This is dynamic because aspects can be added and removed at run-time. However, the referential nature of the dynamic weaving requires an execution platform to intercept the application and insert it into the aspects at execution time. In addition, aspects can be combined to form complex structures by means of inheritance and aggregation relationships.

Finally, it is important to mention that there are a pair of tools that support the JAsCo approach. One of them transforms a Java bean into a JAsCo bean, and the other one is the integration of JAsCo into the PacoSuite [Van01], [Wyd01], which allows modelling component models at a high abstraction level and also allows generating one or more JAsCo connectors from its models.

These tools are implementation frameworks used by developers. As a result, they provide support for programming and not for MDD.

3.1.13. FUSEJ

FuseJ is a programming language for component-based software architectures onto the Java Beans component model [Suv05b]. This language asserts that there are no aspects and these services can be implemented as a component. It is a platform dependent language that does not make distinctions between normal components and aspect components. It is based on the

component architecture presented in Figure 7. Each concern and component is programmed as a component of the Component Layer. The provided and required services of components are sent through the gates of the Gate Layer by preserving the encapsulation of components (black box view). Finally, the coordination among the gates is programmed using connectors of the Connector Layer. However, this connector must be implemented in different ways depending on whether two normal components are being coordinated or a component and an aspect component are being coordinated. In this last case, the coordination is performed using the aspect-oriented primitives to define the pointcuts and advices (to specifying the weaving process).

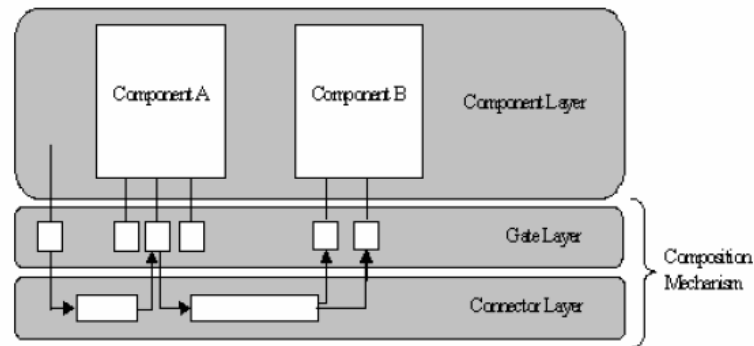


Figure 7. Unified Component Architecture [Suv05b]

With regard to instantiation, aspects are instantiated as regular components; each aspect can have more than one instance. Finally, it is important to emphasize that the tool support for FuseJ is currently being developed.

3.1.14. JAC

JAC is a framework to develop aspect-oriented distributed applications in Java programming language [Paw04]. The main contribution of the programming model of JAC is the fact that aspects can be distributed. They also have a dynamic nature, which means they can be added and removed at run-time.

JAC provides a set of classes and methods that are extended when a new JAC application is developed. JAC provides two different levels of aspect-oriented programming: the programming level and the configuration level. The former is used when new aspects are programmed from scratch. The latter is used when existing aspects are customized for new requirements.

Since JAC packages normal components inside containers, it also defines aspects as components that are inside containers. The components that define aspects are called aspect components. These containers are remote servers that can represent normal or aspect components. JAC aspect components crosscut normal components that are not necessarily in the same location as the aspect components. As a result, JAC gives support to distributed weaving processes. This need emerges because aspects are treated as components and the weaving process (the pointcuts and advices) are defined inside the aspect. If the weaving process were defined in a different entity of the aspect such as a connector, this distribution need would not arise, because the distributed communication is supported by components and connectors. The main drawback is not the effort needed to support distributed communication in pointcuts, it is the fact that the behaviour of the aspect cannot be reused. JAC loses the reusability of aspects because the relationships for applying the aspect to a specific context are defined inside it.

3.1.15. JIAZZI

Jiazzi is an aspect-programming model that extends Java by means of encapsulated code modules called units [McD03]. They are separately compiled and are externally linked code modules that are introduced into a Java program. These units were originally created to obtain higher modularity of code. However, they are currently being used to add aspects to non-aspect-oriented Java programs in a non-invasive way. This is possible because JIAZZI also provides linking units like connectors to specify the connections of units and a base Java program.

Jiazzi units are composed of Java classes that implement the behaviour of a specific concern. They are compiled independently of linking units and base code; as well as the type

checking is also performed internally. In addition, an external compilation is needed to perform the connection between base code and units by means of linking units. In this compilation process, the types of connections must also be checked.

Jiazzi does not extend the syntax of the Java programming language because it introduces aspects as externally linked Java modules. Jiazzi and its interaction with Java are implemented using Java, and it runs perfectly on this technology. Its main drawback is the fact that it is a technology-dependent model and the pre-compilation and encapsulation of its modules before its integration reduces the flexibility to evolve aspects at run-time.

3.2.COMPARISON OF ASPECT-ORIENTED SOFTWARE ARCHITECTURES

There are several features that are essential for analyzing and to comparing the different approaches that have been presented in the previous section. The features that have been used as comparison criteria have been selected starting from the premise that an aspect-oriented software architecture approach should completely support the development and maintenance processes of software. It is important to mention that there are important features, such as the instantiation mechanisms and the types checking, that have not been used to compare the different approaches because the proposal does not usually give very much information about these features. The analysis of these features is included in their descriptions above for those approaches that provide information about them.

Next, it is detailed the features of comparison and the reasons because they have been considered as a classification criteria of aspect-oriented architectural models.

- **Aspect-oriented model:** This feature defines the kind of aspect-oriented model that is integrated with the software architectural model. The four kinds are: asymmetric or symmetric [Har02], multidimensional [Oss01, [Oss00], and composition filters [Ber01], [Ber94]. This characteristic is important because the integration is completely different depending on the aspect-oriented model.

- **Architectural model:** This feature determines whether an architectural model provides connectors for modelling software architectures or not. Those that have connectors provide features that improve the structure and maintenance of software architecture (see section 2.1.2 for details).
- **Definition of Aspects:** The most distinguishable feature of aspect-oriented architectural models is how they integrate aspects and software architectures. There are two ways of doing this: by simulating the notion of aspect by means of another architectural concept or by defining a new concept in software architecture for aspects. The first way refines the architectural concepts varying their original semantics; and the second one requires understanding a new concept to model software architectures.
- **Definition of Weavings:** This is an important feature of aspect-oriented models, and of aspect-oriented architectural models. The definition of weavings feature specifies where the weaving process between aspects and architectural elements is defined. If the pointcuts and advices are defined inside the aspect, the aspect is dependent on the context that the aspect is connected to. However, if they are defined outside the aspect, the behaviour of the aspect can be reused independently of where they will be connected.
- **ADL:** Another feature that is necessary to take into account when comparing architectural models is whether the ADL is a formal language or not. The formal nature of an ADL is an indispensable property of architectural models if the purpose of the approach is to generate code without ambiguity, to verify properties, to validate behaviour, to trace the different levels of abstraction in a suitable way, to evolve software architectures preserving the consistency of the system and so forth.
- **Aspect-Oriented Evolution:** The evolution of aspect support is an important feature that can improve the evolution and run-time evolution of software architectures. As a result, an approach that provides mechanisms for adding or removing aspects is a great advantage.
- **Purpose:** The purpose of the approach is an essential feature to be able to compare models. There are aspect-oriented architectural models that give complete support during the development process, others that analyze or evolve models, and still others that fulfil several purposes.

- **Technology:** This is an important feature that distinguishes the wide variety of aspect-oriented architectural models that exists. An aspect-oriented architectural model should be specified in an abstract way by means of an ADL. As a result, the same specification can be applied to different platforms and different programming languages. However, if the model depends on a specific platform and/or programming language, its application and flexibility are considerably reduced.
- **Graphical support:** The graphical specification of aspect-oriented software architectures is a necessary feature to avoid the complexity of using ADLs. The graphical support is achieved by defining the graphical metaphor of ADLs by means of a new language or by extending a well-known graphical language.
- **Tool support:** A significant feature of the aspect-oriented architectural models is its support by means of a framework that guides the analyst during the development and maintenance processes. A framework can provide a wide variety of facilities such as modelling support, ADL generation, code generation, code execution, validation, verification, evolution, run-time evolution, etc.
- **MDD support:** A significant feature of an approach that should completely support the development and maintenance processes of software following the MDD approach is the MDD support that it offers. The complete MDD support should consist of automatically generating the code from models and to guide the analyst during all the stages by providing mechanisms to facilitate the tasks. Some of these mechanisms are: verification techniques, reusability mechanisms, integration facilities, code generation mechanism, etc.

A comparison table has been developed from the features and the approaches analyzed in the above section. This table is divided into two separate tables due to the limitation of the page dimensions. Blank cells indicate that no information was available.

	Aspect-oriented model	Architectural model	Definition of Aspects	Definition of Weavings	ADL
PCS	Multidimensional and symmetric	Without connectors	Aspects like connectors	Inside aspects	SADL: Formal compositional ADL
CAM/DAOP	Asymmetric	Without connectors	Aspects like connectors	Outside aspects using communication between interfaces	DAOP – ADL: Not formal, based on XML
Superimposition	Asymmetric: Two levels: aspects and architectures		Java Classes inside a superimposition layer	Inside aspects	
TRANSAT	Asymmetric. Only technical aspects	Without connectors	Aspects like components. Aspect components	Outside aspects. Using adapters or weavers \equiv connectors	SafArchie component model
ASAAM	Asymmetric	Not fixed	Scenarios	Outside Aspects	Not fixed
AVA	Asymmetric	Not fixed	Aspects as views	Outside Aspects	Not fixed
AspectLEDA	Asymmetric: Two levels: Aspects and architectures	Without connectors	Aspects as components	Outside aspects using coordinators \equiv connectors	Leda: Formal Compositional ADL
AOCE	Asymmetric	Without connectors	Aspects as components	Outside aspects using aspect managers \equiv connectors	
Component Views	Asymmetric		Not aspects. Concerns as viewpoints for defining architectural views		
Aspectual Components	Asymmetric: Two levels: Aspects and object-oriented applications		Aspects as components: Aspectual components	Outside aspects with connectors	

Caesar	Asymmetric		Aspect Collaboration Interface (ACI)	Separation of ACI modules into implementation and interaction of aspects	
JASCO	Asymmetric		Aspects	Hooks and connectors	
FUSEJ		With Connectors	Without Aspects: Components	Connectors	
JAC	Asymmetric		Aspects as components: aspectcomponents	Inside aspects	
JIAZZI	Asymmetric: Two levels: Aspects and object-oriented applications		Units	Linking units	

Table 1. First comparison of aspect-oriented software architecture approaches

	Aspect-Oriented Evolution	Purpose	Technology	Graphical support	Tool support	MDD support
PCS		Development of AO Software Architecture	Independent	UML profile: Aspect is a stereotype of a UML class	ConcernBase tool: modelling support, ADL generation from UML, no code generation, no execution	Partial: Modelling techniques
CAM/DAOP	Dynamic weaving but not adding and removing aspects at run-time	Development of AO Software Architecture	Independent	UML profile	DAOP platform: Java Technology, modelling support, DAOP middleware for code execution	Partial: Multiple views of analysis

Superimposition		Programming aspect-oriented Java applications and verifying properties of aspect-oriented superimposition	Dependent on Java technology			
TRANSAT		Only evolution support, the initial aspect-oriented specification is not supported.	Independent	UML profile	SafArchie Studio. Extension of ArgoUML	Partial: Models to analyze evolution and maintenance
ASAAM		Analysis of Software Architectures	Independent	UML profile: scenarios	ASAAM-T	
AVA		Development of AO Software Architecture	Independent	UML profile: aspect is an stereotype of a UML package that contains an extension of component diagram	MADE tool: modelling support	Partial: Multiple views of analysis
AspectLEDA		Development of AO Software Architecture	Independent			
AOCE	Dynamic weaving	Development of AO Software Architecture	Dependent on JViews		JComposer: An extension of the JViews tool	Not supported: development framework
Component Views		Analysis of software architectures	Independent	UML profile		

Aspectual Components		Programming aspect-oriented Java applications	Dependent on Java technology			
Caesar		Programming aspect-oriented Caesar applications	Dependent on Caesar programming language		Programming framework	Not supported: development framework
JASCO	Dynamic weaving and support for adding and removing aspects at run-time	Programming aspect-oriented application	Dependent on Java or .Net technology		Programming framework	Not supported: development framework
FUSEJ		Programming aspect-oriented applications onto Java Beans	Dependent on the Java Beans component model			
JAC		Programming aspect-oriented Java applications	Dependent on Java technology			
IAZZI		Programming aspect-oriented Java applications	Dependent on Java technology			

Table 2. Second comparison of aspect-oriented software architecture approaches

3.3.CONCLUSIONS

After the analysis and comparison of different approaches for aspect-oriented software architecture, it is possible to conclude that these proposals at the architectural level usually extend ADLs without connectors and mainly follow an asymmetric model by considering functionality as architectural components. Despite the fact that there has been a lot of work done, these proposals are only focused on a single specific purpose: the analysis, evolution or development of software architectures. They do not pursue several purposes simultaneously to provide a complete development and maintenance support. Furthermore, they always introduce the notion of aspect by using original architectural concepts, despite the fact that they do not

provide the suitable semantics for aspects. And finally, the most important conclusion for this thesis, they do not provide a complete support for MDD. As a result, it is necessary to provide an aspect-oriented model for symmetric aspect-oriented models and ADLs with connectors, whose development will follow the MDD paradigm. This model should include:

- A suitable semantics for the aspect concept
- A graphical modelling metaphor
- Analysis and evolution capabilities
- Technological support in order to execute the aspect-oriented architectural models that have been defined independently of technology
- A guided support during the development and maintenance processes of software following MDD: Reusability, Verification, Code generation, Maintenance, Evolution, etc.

The PRISMA approach has been defined to fulfil these needs. In particular, in this thesis the main properties to facilitate and provide mechanism to give a complete MDD support in PRISMA are defined. As a result, this thesis is a step forward in the previous PRISMA work.

CHAPTER 4

PRISMA BACKGROUND

PRISMA provides a model for the definition of complex software systems [Per05a]. 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 MDD paradigm 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 for different technological platforms and languages using automatic code generation techniques.

The purpose of this chapter is to present the main properties of the PRISMA model and to explain in detail the PRISMA metamodel. The PRISMA metamodel is the starting point to apply the MDD to the development of PRISMA applications.

4.1. THE PRISMA MODEL

PRISMA provides a model for the description of software architectures of complex and large systems. It introduces aspects as first-order citizens of software architectures. This means that,

in PRISMA, aspects are not simulated through other architectural concepts such as connectors, views or similar mechanisms as in other approaches. PRISMA creates a new concept for modelling concerns called aspects. As a result, PRISMA specifies different characteristics (distribution, safety, context-awareness, coordination, etc.) of an architectural element (component, connector) using aspects. As a result, PRISMA preserves the meaning of the component and aspect concepts.

From the aspect-oriented point of view, PRISMA is a symmetrical model that does not distinguish a kernel or core entity to encapsulate functionality; functionality is also defined as an aspect. One concern can be specified by several aspects of a software architecture, whereas a PRISMA aspect represents a concern that crosscuts the software architecture. This crosscutting is due to the fact that the same aspect can be imported by more than one architectural element of a software architecture. In this sense, aspects crosscut those elements of the architecture that import their behaviour (see Figure 8).

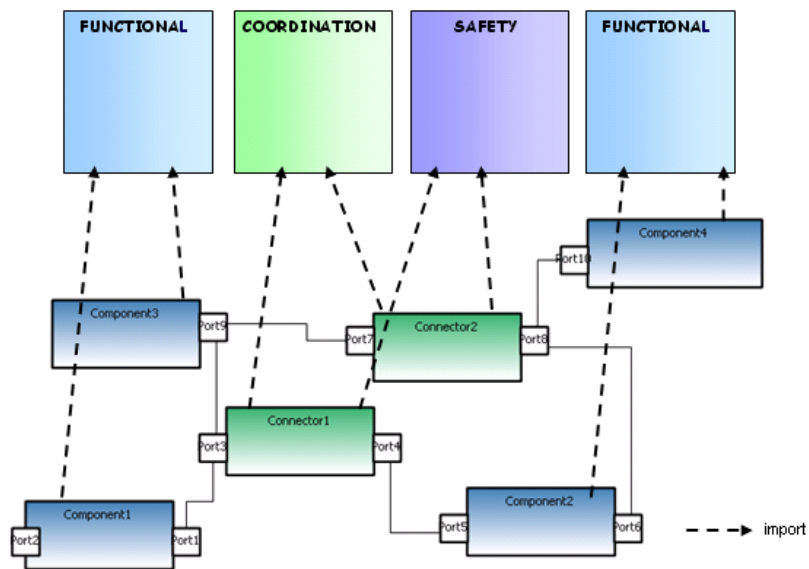


Figure 8. Crosscutting-concerns in PRISMA architectures

The fact that PRISMA is a symmetrical model is an advantage. This facilitates the construction of software architectures since the model does not manage two different concepts

(class or component, and aspect) in different ways. In addition, the reusability of functional properties is independent of the architectural element that imports it because the functionality is specified as an aspect. However, if this functionality were implemented as a kernel class of the architectural element, the reuse of the functionality would only be achieved by reusing the full architectural element. Consequently, a more uniform model is obtained because of the homogeneity of the concepts that build an architectural element.

A PRISMA architectural element can be seen from two different views: internal and external. In the external view, architectural elements encapsulate their functionality as black boxes and publish a set of services that they offer to other architectural elements (see Figure 9). These services are grouped into interfaces to be published through the ports of architectural elements. Each port has an associated interface that contains the services that are provided and requested through the port. As a result, ports are the interaction points of architectural elements.



Figure 9. Black box view of an architectural element

The internal view shows an architectural element as a prism (white box view). Each side of the prism is an aspect that the architectural element imports. In this way, architectural elements are represented as a set of aspects (see Figure 10) and the weaving relationships among aspects.

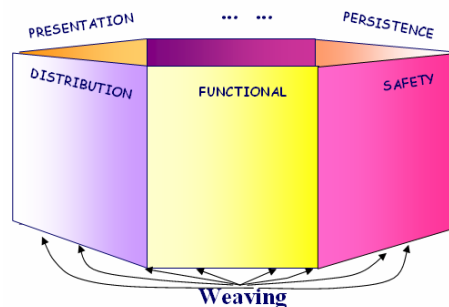


Figure 10. White box view of an architectural element

Since PRISMA is a symmetrical aspect-oriented model that it is applied at the architectural level, the weaving process does not define the pointcuts between the base code and the aspect code and their corresponding advices. In PRISMA, there is no base code; all behaviour of the system 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 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. In PRISMA, in order 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. A weaving is defined by means of operators that describe the order in which services are executed. A weaving that relates service *s1* of aspect *A1* and service *s2* of aspect *A2* can be specified using the following operators: *after*, *before*, *instead*, *afterif* (Boolean condition), *beforeif*(Boolean condition, and *insteadif*(Boolean condition).

The communications between the white box and black box views is possible by means of interfaces; which are associated to ports and are used by aspects (see Figure 11). Consequently, a request for a service that arrives to a port of an architectural element is processed by an aspect that uses the same interface that is used by this port.

PRISMA has three kinds of architectural elements: components, connectors, and systems. Components and connectors are simple, but systems are complex components. A component is an architectural element that captures the functionality of software systems and does not act as a coordinator among other architectural elements; whereas, a connector is an architectural element that acts as a coordinator among other architectural elements.

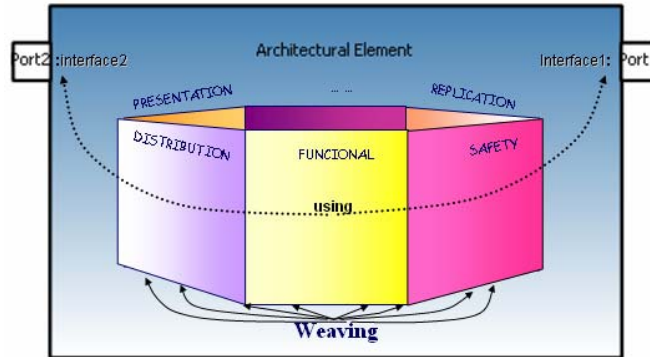


Figure 11. Communication between the white box and the black box views

Connectors provide the separation of component interactions thereby achieving a higher level of abstraction, modularity, and a greater architectural view of the system [Sha94]. For this reason, PRISMA connectors are first-class citizens of the ADL. Connectors do not have the references of the components that they connect and vice versa. Thus, architectural elements are reusable and unaware of each other. This is possible due to the fact that the channels defined between components and connectors have their references (*attachments*) instead of architectural elements. Attachments are the channels that enable the communication between components and connectors. Each attachment is defined by attaching a component port with a connector port.

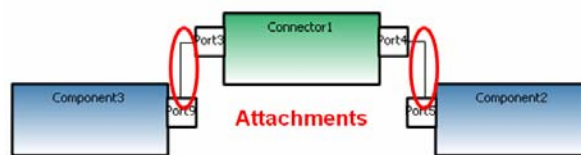


Figure 12. Attachments

PRISMA components can be simple or complex. The complex ones are called systems. A PRISMA system is a component that includes a set of architectural elements (connectors, components and other systems) that are correctly attached. In addition, a system can have its own aspects and weavings as components and connectors. Since a system is composed by

other architectural elements, the composition relationships among them must be defined. These composition relationships are called bindings. Bindings establish the connection among the ports of the complex component (the system) and the ports of the architectural elements that a system contains (see Figure 13).

In PRISMA, the dynamics of aspect-oriented architectures are treated at the meta-level. The meta-level contains the elements that define the PRISMA concepts as data. They can be created, modified and destroyed through the execution of the services of the meta-level. In this way, the execution of services is reflected in the architecture by updating this data (the concept of reflection). As a result, the PRISMA meta-level allows for the creation, destruction and evolution of architectural elements and aspects as well as the dynamic reconfiguration of software architectures. The PRISMA meta-level is represented by means of a metamodel that contains one metaclass for each PRISMA concept. These metaclasses define a set of properties and services for each concept considered in the model (see section 4.2).

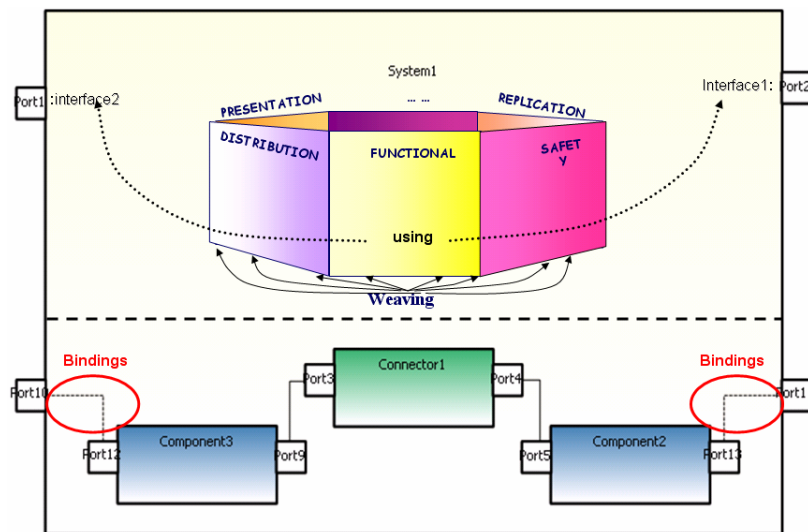


Figure 13. Systems

In PRISMA, the complete view of the software architecture is not lost because of the use of aspect, that is, the use of the view notion is not required to define aspects in the PRISMA

software architectures. But, it is possible to define an aspect-oriented view of the software architecture by considering all the architectural elements that import a specific aspect, and also another view by considering all the architectural elements that import a kind of aspect concern. For example, this allows the definition of the view of all the architectural elements of the *TeachMover* robot that import a safety concern.

The PRISMA model takes advantage of the notion of aspect from the beginning of the system definition by specifying the aspects that are found in the requirements specifications. These aspects are reusable and can be used during the rest of the development process as well as in the maintenance process of the PRISMA software architectures. Thus, PRISMA does not require an initial architectural specification of the system in order to introduce aspects. Moreover, the change of a property only requires the change of the aspect that defines it, and then, each architectural element that imports the changed aspect is also updated

4.2. THE PRISMA METAMODEL

Metamodels define models and establish their properties in a precise way. In addition, metamodels facilitate the automation and maintenance of software development thanks to the support that modelling tools currently offer for these tasks and the MDD paradigm. In order to take advantage of these properties, the PRISMA meta-level is represented by means of a metamodel that contains a set of metaclasses that are related to each other. These metaclasses define a set of properties and services for each concept considered in the model. The metaclasses and their relationships define the structure and the information that is necessary to describe PRISMA architectural models. In addition, the PRISMA metamodel defines the constraints that must be satisfied by a PRISMA architectural model. These constraints guide the methodology for modelling PRISMA architectural models. At the end of the modelling process, all of them must be satisfied in order to ensure that an architectural model is correct.

The PRISMA metamodel has been specified using the class diagram of the Unified Modelling Language (UML) 1.5. and the Object Constraint Language (OCL) 2.0 [UML07]. UML has been used to model the metaclasses and their relationships, attributes and services. OCL has been used to specify the constraints of the metamodel. The choice of these languages

over others is based on the fact that they are standards and are widely extended languages. As a result, they facilitate the comprehension of the model by new users.

The PRISMA metamodel is composed of three main packages: *Types*, *Architecture Specification*, and *Common* (see Figure 14).

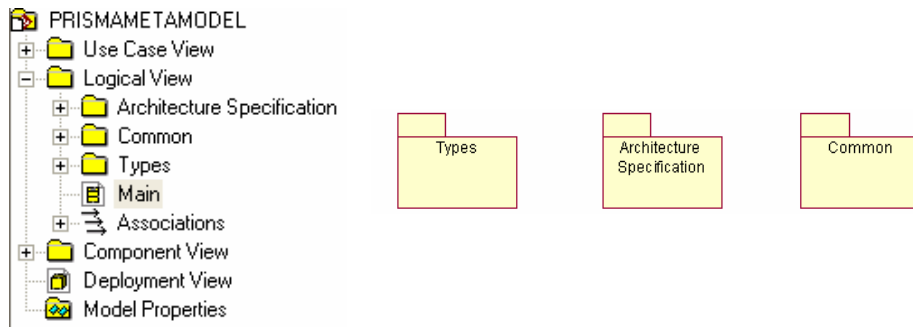


Figure 14. Main packages of the PRISMA metamodel

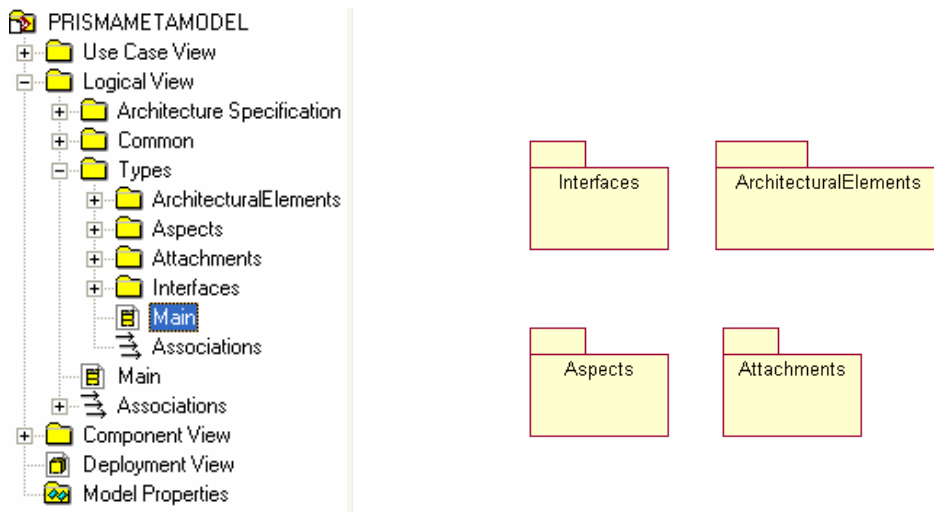


Figure 15. The package *Types* of the PRISMA metamodel

The package *Types* contains the packages *Interfaces*, *Aspects*, *Architectural Elements* and *Attachments* of the PRISMA model (see Figure 15). These packages define the properties of PRISMA types.

The package *Architecture Specification* defines the elements that form an architectural model using the types that are defined in the package *Types*. This package provides the mechanisms to build an architectural model.

The package *Common* defines the utilities that are necessary to develop any kind of model. It provides mechanisms to define data types, parameters, constant values, formulae of different kinds and complex process [Per06c].

4.2.1. THE PACKAGE “TYPES”

4.2.1.1. The Package “Interfaces”

An interface publishes a set of services. This set of services is composed of at least one service, and there is no limit to the number of services that can be specified (see Figure 16). The services that make up an interface are called *InterfaceServices*. These services are defined in an abstract way, without specifying whether they are going to be provided (*in*), requested (*out*), or provided and requested (*in/out*) by architectural elements. An *InterfaceService* only specifies its signature.

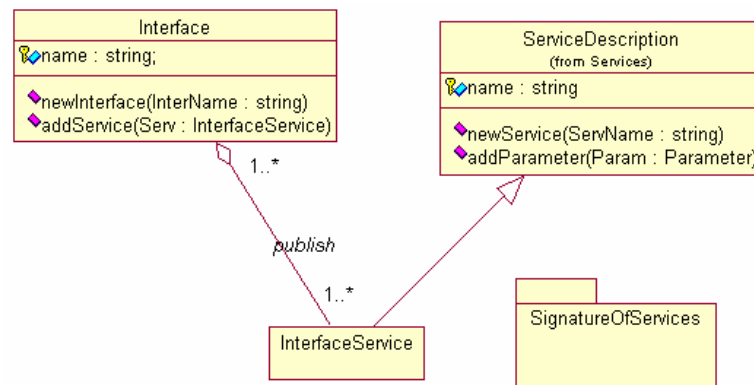


Figure 16. The package *Interfaces* of the PRISMA metamodel

The signature of a service specifies its name and parameters. The parameters are defined in a specific order. The data type and kind (*input/output*) of parameters are also defined (see Figure 17).

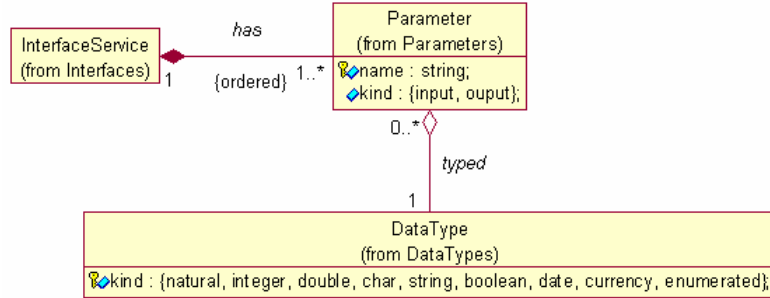


Figure 17. The package *SignatureOfService* of the PRISMA metamodel

The metaclass *InterfaceService* inherits its properties from the metaclass *ServiceDescription*. This class allows the creation of services using the service *newService*, whose parameter defines the name of the service that is created. The attribute *name* stores the value of the *ServName* parameter of *newService*. The service *addParameter* adds parameters to services.

The metaclass *Interface* creates interfaces. The service *newInterface* creates a new interface, whose parameter defines the name of the interface that is created. The service *addService* adds a service to the interface, whose parameter provides the *InterfaceService* that is added to the interface.

4.2.1.2. The package “Aspects”

An *aspect* defines structure and behaviour of a specific *concern* of the software system. The *Aspects* package includes all the metaclasses that are necessary to specify an aspect. The structure and the behaviour of aspects are defined by attributes, services, preconditions, valuations, constraints, played_roles and protocols. These concepts are sub-packages of the *Aspects* package (see Figure 18).

The metaclass *Aspect* (see Figure 19) has two attributes, *name* and *concern*. These attributes store the name of the aspect and the concern that the aspect belongs to, respectively. The service *newAspect* creates a new aspect, whose parameters define the name, the concern and the protocol of the aspect.

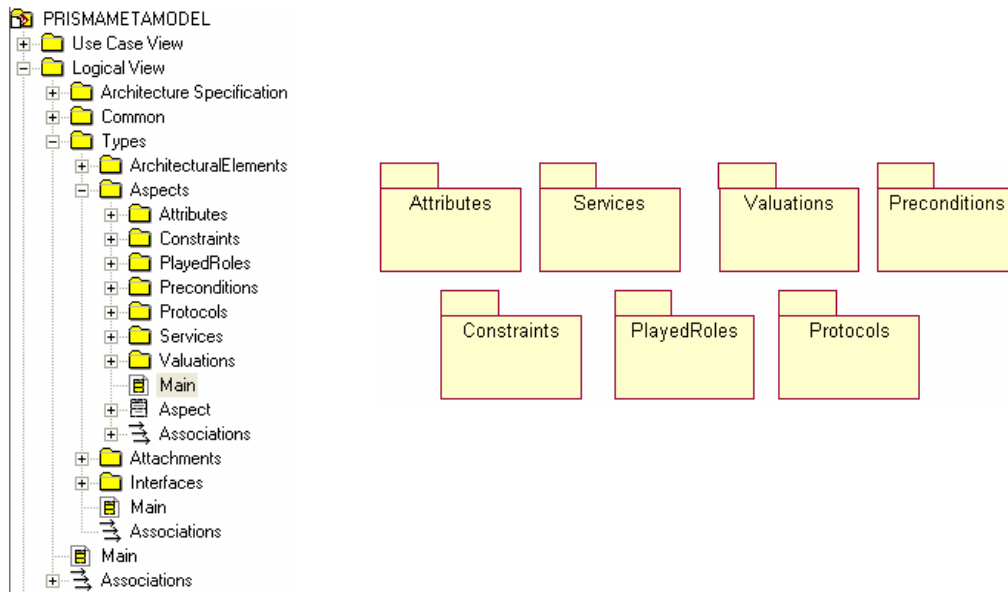


Figure 18. The sub-packages of the package *Aspects* of the PRISMA metamodel

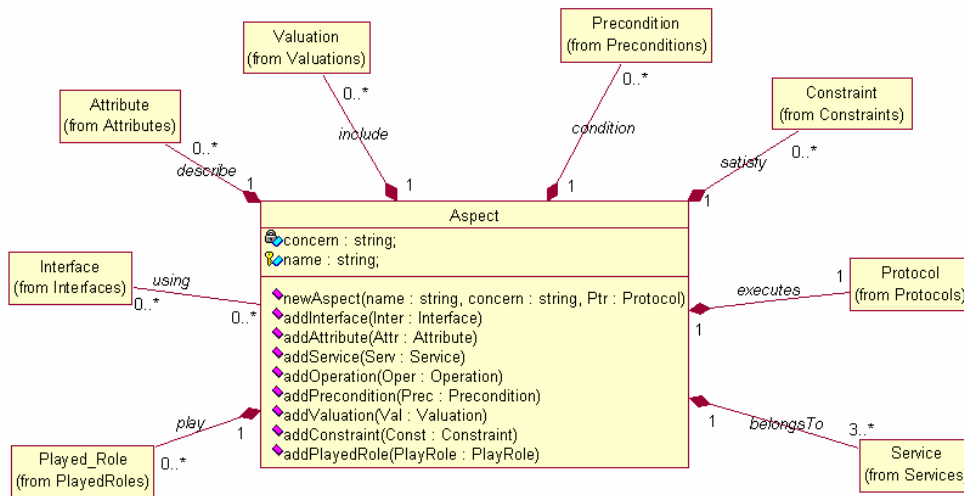


Figure 19. The metaclass *Aspect* of the package *Aspects* of the PRISMA metamodel

Aspects may need to store information to successfully perform their computation. For this reason, the *Aspect* metaclass has an aggregation relationship with the *Attribute* metaclass . This

relationship aggregates the attributes that an aspect is composed of (see the *describe* aggregation relationship in Figure 19). This aggregation is established by invoking the *addAttribute* service. This service adds attributes to an aspect through its *Attr* parameter by providing the attribute that is added to the aspect. Aspects may need to constrain the value of attributes. For this reason, an aspect can be composed of constraints that determine the value of aspect attributes (see the *satisfy* aggregation relationship in Figure 19).

Aspects must be composed of three or more services. The three services that are required are the following: The services *begin* and *end* to start and finish the execution of the aspect, and at least one service to perform the necessary computations of an aspect (see the *belongsTo* aggregation relationship in Figure 19). Aspect services can be private or public. Public services of an aspect are those that are published by an interface whose semantics is defined by the aspect. As a result, aspects import the interfaces whose semantics they define (see the *using* association relationship in Figure 18). In order to associate interfaces and services to aspects, the *aspect* metaclass provides the *addInterface* and *addService* services, whose *Inter* and *Serv* parameters provide the interface and the service that are added to aspects.

In order to define the semantics of aspect services, aspects are composed of preconditions, valuations, played_roles and a protocol (see the aggregation relationships *condition*, *include*, *play* and *executes* in Figure 19). For this reason, the *Aspect* metaclass has three services to associate preconditions, valuations, and played_roles to aspects. These services are *addPrecondition*, *addValuation*, and *addPlayedRole*, respectively.

In addition to these attributes, services, and relationships, the metaclass *Aspect* has an associated set of constraints to completely model its properties (see Figure 20).

These constraints correspond to the OCL rules shown in Figure 20. They specify the following:

<<Every aspect has a “begin” service>>

<<Every aspect service must participate in the protocol of the aspect>>

<<Every aspect has an “end” service>>

<<For each interface that an aspect imports, the aspect must define at least a played_role associated to this interface>>

<<Every service of an interface that is imported by an aspect must be a service of the aspect>>

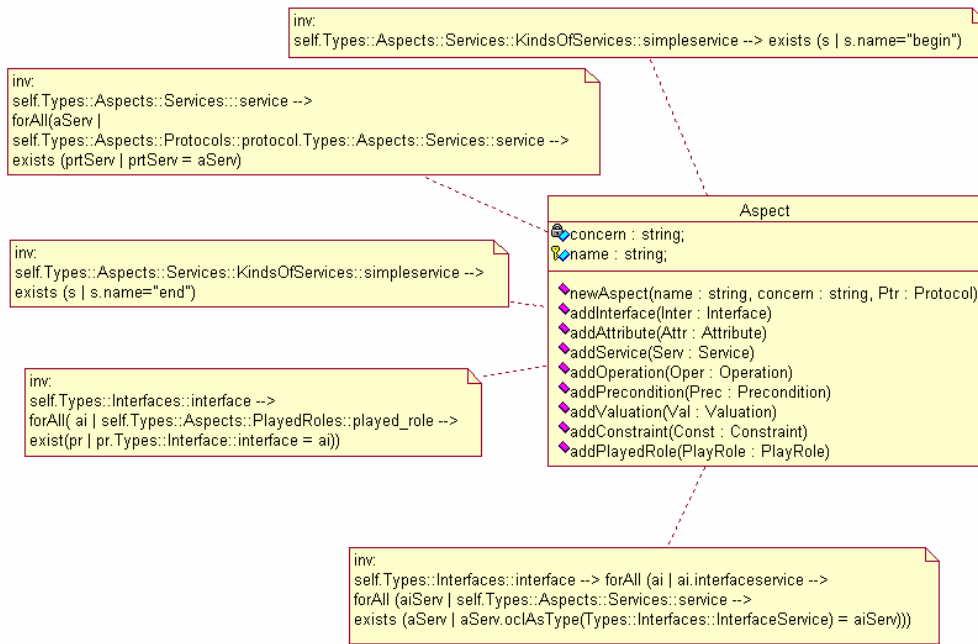


Figure 20. Constraints of the metaclass *Aspect*

- The package “Attributes”

Attributes store a value of a specific data type. Therefore, each aspect attribute must be associated to a data type (see Figure 21).

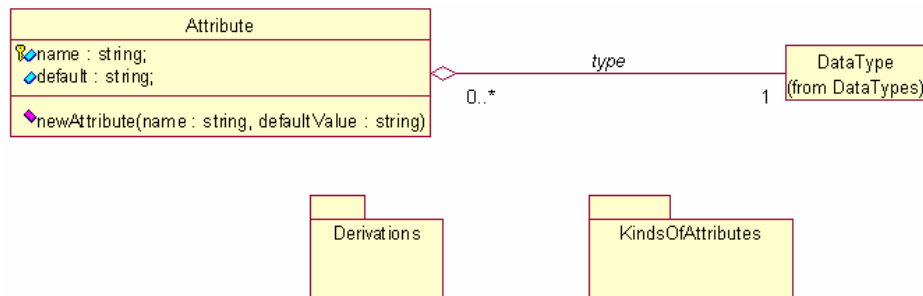


Figure 21. The package *Attributes* of the PRISMA metamodel

The metaclass *Attribute* has two attributes, *name* and *default*. The attribute *name* stores the name of the aspect and the attribute *default* stores the default value of the attribute when necessary. The service *newAttribute* creates a new attribute, whose parameters define the name and the default value of the attribute.

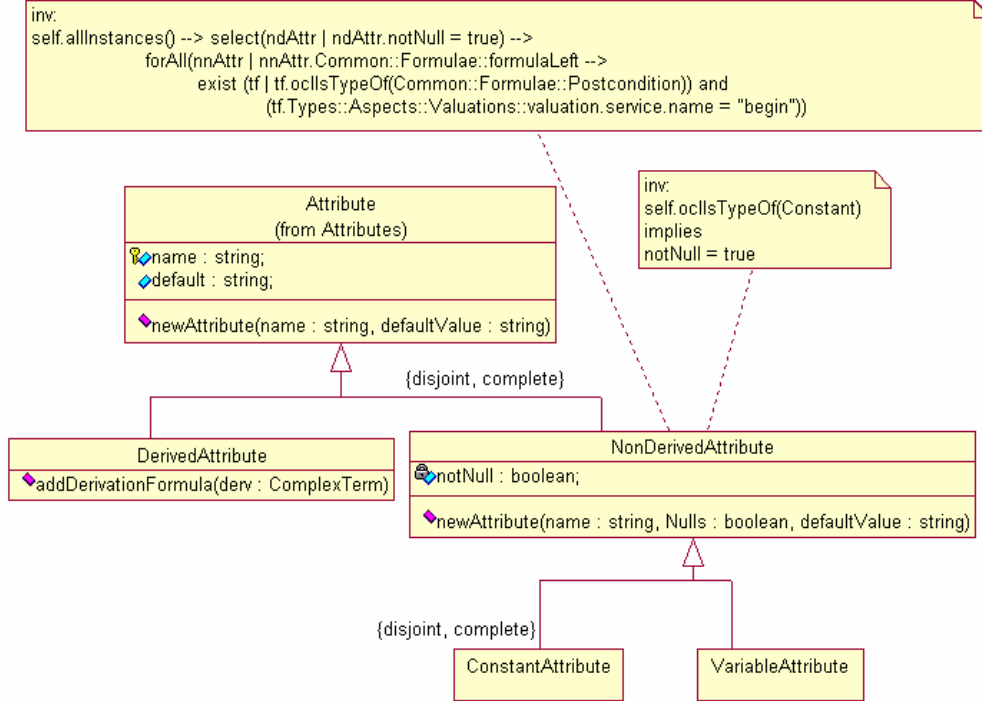


Figure 22. The package *KindsOfAttributes* of the PRISMA metamodel

In PRISMA, it is possible to define different kinds of attributes. The semantics of each kind is defined in the *kindsOfAttributes* sub-package of the *Attributes* package. Attributes can be classified into derived and non-derived attributes (see Figure 22). Derived attributes calculate their values on demand by applying a derivation rule. The derivation rule is associated to the derived attribute (see Figure 23). Non-derived attributes store their values, and it is possible to constrain the fact that they must contain a value by means of the *notNull* attribute (see the *NonDerivedAttribute* metaclass in Figure 22). If *notNull* is true, the attributes must contain a value; if *notNull* is false, no value is required. In order to correctly define the semantics of non-

derived attributes, there are two constraints associated to the *NonDerivedAttribute* metaclass. These constraints correspond to the OCL rules shown in Figure 22. They specify the following:

<<For each non-derived attribute that cannot contain a null value, there is a postcondition of the begin service valuation that must provide a value to the attribute >>

<<The "notNull" attribute of a constant attribute is always true>>

Non-derived attributes can be *constant* or *variable*. Constant attributes store values that cannot change; i.e., they cannot be modified during the execution of the aspect. Also, variable attributes store values that can be modified during the execution of the aspect.

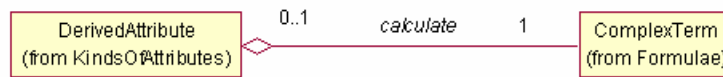


Figure 23. The package *Derivations* of the PRISMA metamodel

- The package “Services”

The metaclass *Service* is a specialization of the metaclass *InterfaceService* (see Figure 24). As a result, it inherits all its properties and services. The metaclass *Service* defines that every service of an aspect must be characterized by the behaviour that it offers in the context of the aspect. This means that the service can either be provided, requested or both by the aspect. This characteristic is specified by the *type* attribute of the metaclass. The *type* values are *in*, *out* and *in/out* to define the behaviour of a server (provide), a client (request), or both a server and a client (provide and request), respectively. A service of an aspect can also have an alias. An alias permits changing the name of an *InterfaceService* inside the aspect. This metaclass stores the alias name and provides the *newAlias* service to change the name. The parameters of *newAlias* are the service whose name is going to be changed and the new alias.

There are two kinds of services: simple services and transactions (see Figure 25). A transaction is a complex service that it is composed of more than one service and is executed in a transactional way (all or nothing) (see the *composedService* aggregation in Figure 25). A transaction describes a process, which models how and when the different services that

compose the transaction are executed. As a result, a transaction is a specialization of the *Process* metaclass.

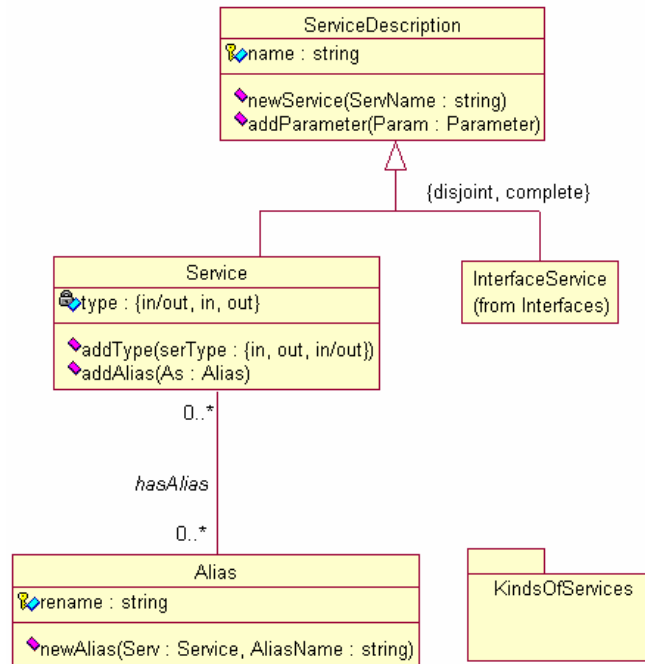


Figure 24. The package *Services* of the PRISMA metamodel

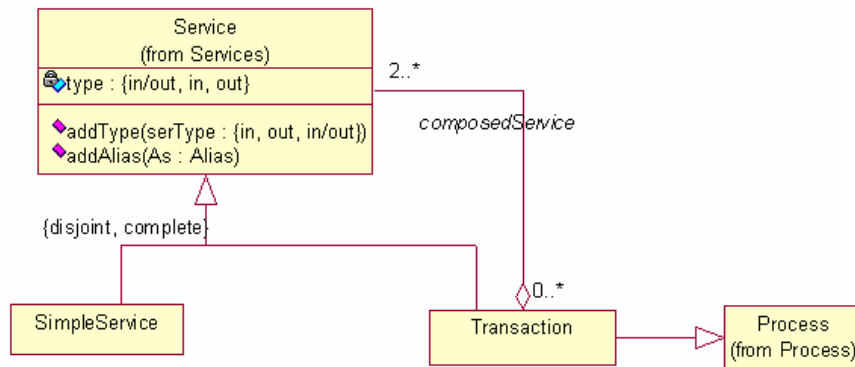


Figure 25. The package *KindsOfServices* of the PRISMA metamodel

- The package “Constraints”

Constraints are formulae that establish conditions on the state of the aspect that they belong to. As a result, each time that a service execution is finished, the value of each attribute must satisfy the aspect constraints. There are two kinds of constraints: static and dynamic (see Figure 26).

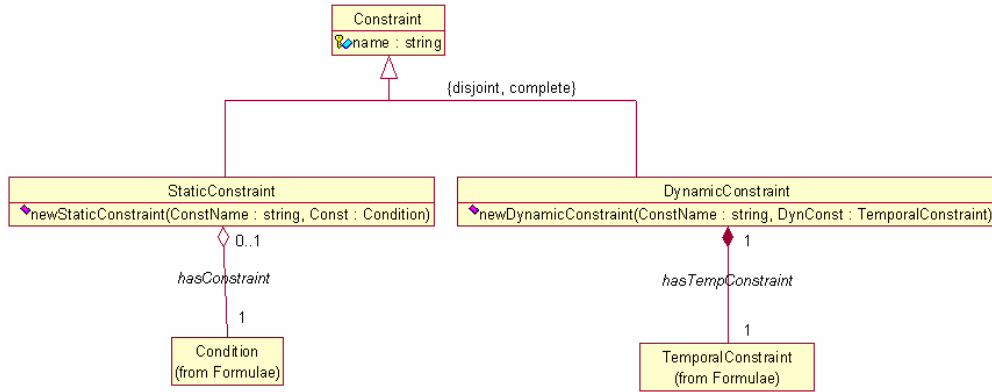


Figure 26. The package *Constraints* of the PRISMA metamodel

The metaclass *Constraint* is an abstract class that has only one attribute, the *name* of the constraint. This metaclass is specialized into two metaclasses: *StaticConstraint* and *DynamicConstraint*. Each one of them provides a constructor service to create instances of static and dynamic constraints, respectively. The service *newStaticConstraint* creates a new static constraint giving the name and the condition of the constraint as parameters. The service *newDynamicConstraint* creates a new dynamic constraint, whose parameters define the name of the constraint and a condition that uses a temporal.

- The package “Preconditions”

Preconditions establish the condition that must be satisfied to execute an aspect service. Therefore, the metaclass *Precondition* has the aggregation relationship *establishCondition* and aggregation relationship *constrains* with the metaclasses *Condition* and *Service*, respectively (see Figure 27). The first aggregation establishes that a precondition must define the service that it affects. The second aggregation establishes the condition that must be satisfied to execute the service.

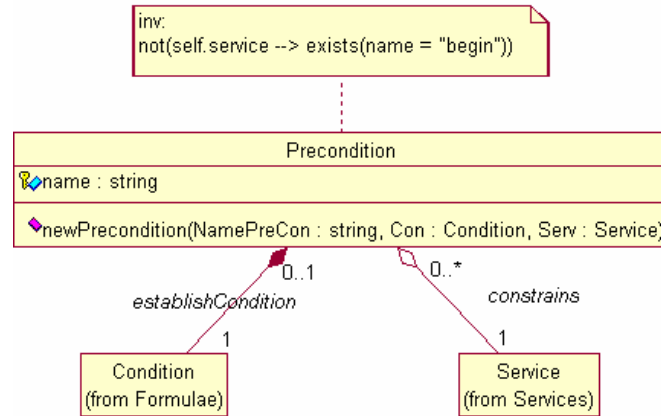


Figure 27. The package *Preconditions* of the PRISMA metamodel

The metaclass *Precondition* only has one attribute, the *name* of the precondition. The service *newPrecondition* creates a new precondition, whose parameters define the name, the condition that must be satisfied, and the service that will only be executed if the condition is satisfied. In addition, the metaclass *Precondition* has an associated constraint in order to ensure that the aspect execution is not conditioned by a precondition. This constraint corresponds to the OCL rule shown in Figure 27. It specifies the following:

<<A service begin does not have preconditions associated to it since the start of the aspect execution cannot be conditioned by the aspect itself>>

This constraint is necessary because preconditions are used to define the business logic of the software system and not to define the mechanisms of creating, destroying or executing instances.

- The package “*Valuations*”

Valuations establish how the service executions affect the aspect state. This semantics is specified by means of two conditions: one that must be satisfied before the service execution and another that must be satisfied after the service execution. For this reason, the metaclass *Valuation* has three aggregation relationships with the metaclasses *Condition*, *Service* and *Postcondition* (see Figure 28).

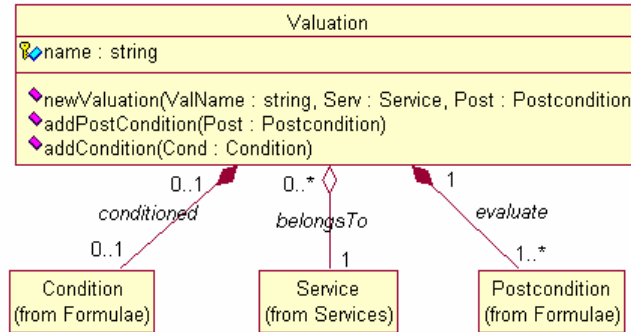


Figure 28. The package *Valuations* of the PRISMA metamodel

The metaclass *Condition* defines the condition that specifies the state of the aspect before the service execution. This condition is optional, this means it does not have to be specified when the state before the execution is not relevant to the state change (see the *conditioned* aggregation in Figure 28). However, the specification of the condition after the service execution is mandatory. The postcondition must be satisfied after the service execution. Since the metaclass *Postcondition* defines the change in one attribute or parameter and a valuation can affect several attributes or parameters, a valuation can have more than one postcondition associated to it in order to model the service changes in several attributes and/or parameters (see the *evaluate* aggregation in Figure 28).

The metaclass *Valuation* has only one attribute, the *name* of the valuation. The service *newValuation* creates a new valuation, whose parameters define the name, the service that produces the change of state, and the condition that must be satisfied after the service execution. Moreover, it has two services *addCondition* and *addPostCondition*. The *addCondition* adds a condition to the valuation. This condition must be satisfied before the service execution. The *addPostCondition* adds more than one postcondition when the valuation affects several attributes or parameters.

- The package “*PlayedRoles*”

PlayedRoles establish how the services of an interface can be executed. As a result, a *played_role* defines a process that orchestrates the service execution of a specific interface.

Since the metaclass *Played_Role* defines a process, it inherits the properties of the metaclass *Process*.

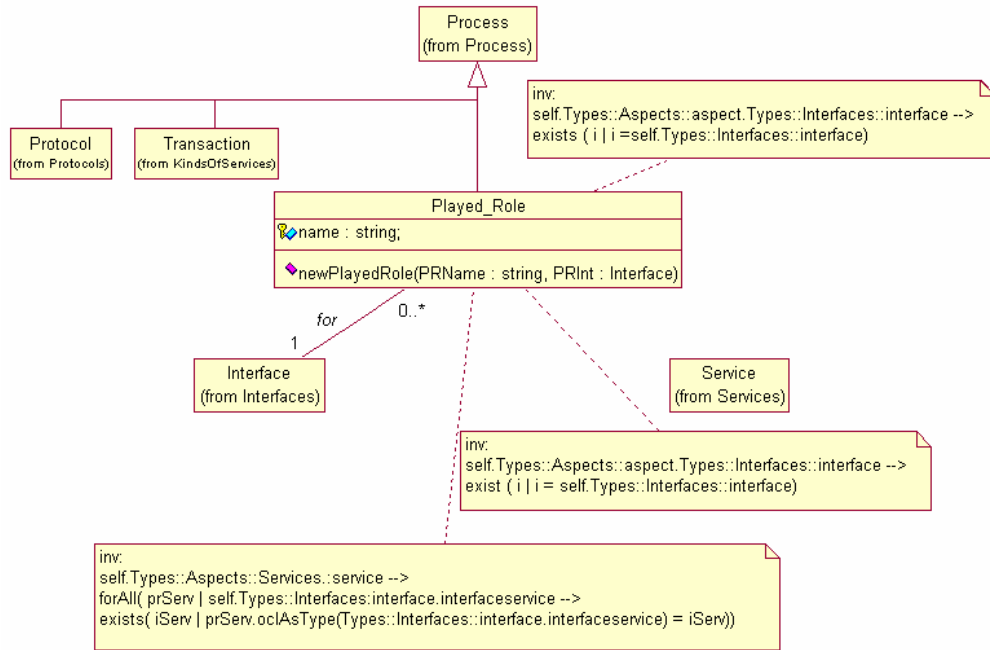


Figure 29. The package *PlayedRoles* of the PRISMA metamodel

The metaclass *Played_Role* has two association relationships with the metaclasses *Interface* and *Service* (see Figure 29). A played_role defines the behaviour of only one interface (see the the *for* association in Figure 29) and describes the execution process of more than one service (see the *order* association in Figure 29). However, the played_role cannot be related to any interface or service of the software system. As a result, these relationships are constrained by three constraints. These constraints correspond to the OCL rules shown in Figure 29. They specify the following:

<<Every interface that an aspect imports must have associated a played_role>>

<<The interface of a played_role is one of the interfaces that imports the aspect that the played_role belongs to>>

<<Every service that participate in a played_role must be a service of the played_role interface>>

The metaclass *Played_Role* has one attribute, the *name* of the played_role. The service *newPlayedRole* creates a new played_role, whose parameters define the name and the interface. The behaviour of this interface is defined by the played_role.

- The package “*Protocols*”

A protocol establishes how the services of an aspect can be executed. As a result, a protocol defines a process that coordinates the private and public services of an aspect. Since the metaclass *Protocol* defines a process, it inherits the properties of the metaclass *Process*.

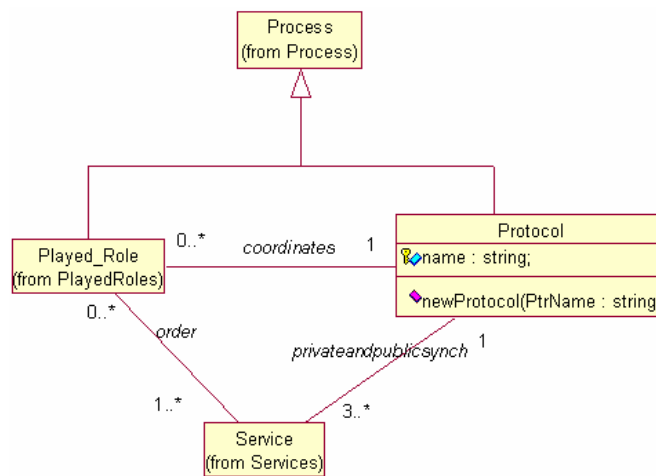


Figure 30. The package *Protocols* of the PRISMA metamodel

A *service* can be private or public and only belongs to one aspect. For this reason, a *service* can only participate in one protocol: the protocol of the aspect that it belongs to (see the *privateandpublicsynch* association Figure 30). In addition, each service of the aspect must participate in its protocol (see the constraint in the *Aspect* package in Figure 20). These services can be either private or public services, but there must be at least three: the *begin* and *end* services of an aspect, and one service to perform the computation of the aspect (see the *privateandpublicsynch* association in Figure 30).

A protocol is the glue of the played_roles and the private services of the aspect. As a result, the protocol coordinates the many different played_roles that have been defined in the aspect that it belongs to. However, a played_role is only coordinated one protocol, its aspect protocol

(see the *coordinates* association in Figure 30). Played_roles are specified using the public services of an aspect. Since aspect services can be private or public, those that are private are not related to played_roles (see the *order* association in Figure 30).

The metaclass *Protocol* has one attribute, the *name* of the protocol. The service *newProtocol* creates a new protocol by providing the name of the protocol as a parameter.

4.2.1.3. The package “*ArchitecturalElements*”

In PRISMA, there are three kinds of architectural elements: components, connectors, and systems (see Figure 31). The package *ArchitecturalElements* defines the metaclass *ArchitecturalElement*. It is an abstract metaclass that specifies the commonalities of the three kinds of PRISMA architectural elements. In addition, it includes all subpackages that define the concepts required to specify PRISMA architectural elements.

The metaclass *ArchitecturalElement* has two aggregation relationships with the metaclassess *Port* and *Weaving*, and one association relationship with the metaclass *Aspect* (see Figure 32). An architectural element has at least one port; the port is part of the architectural element and does not have its own entity without the architectural element. In other words, the aggregation between the port and the architectural element is inclusive (see the *has* aggregation in Figure 32). An architectural element imports at least one aspect and an aspect can be imported by one or more architectural elements of the software system (see the *imports* association in Figure 32). In addition, an architectural element can include a set of weavings to synchronize its aspects. These *Weavings* are related to the architectural element by means of an inclusive aggregation (see the *weaves* aggregation in Figure 32).

The metaclass *ArchitecturalElement* has one attribute, the *name* of the architectural element. In addition, it has three services *addAspect*, *addPort*, *addWeaving* to associate aspects, ports, and weavings to the architectural element, respectively (see Figure 32). It is important to emphasize that this metaclass does not have a constructor (new service) because it is an abstract class that cannot be instantiated.

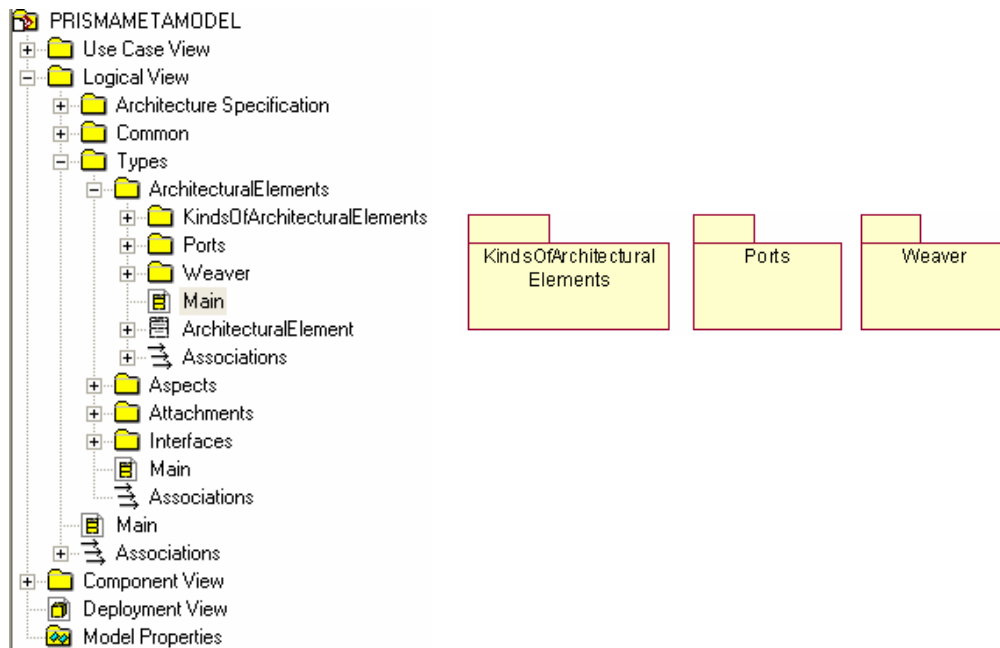


Figure 31. The subpackages of the package *ArchitecturalElements* of the PRISMA metamodel

The metaclass *ArchitecturalElement* has two constraints associated to it in order to completely define its properties. These constraints correspond to the OCL rules shown in Figure 32. They specify the following:

<<There are no two ports of an architectural model that have the same interface and the same played_role associated >>

<<An architectural element cannot import more than one aspect of the same concern>>

The package *KindsOfArchitecturalElements* is a subpackage of the package *ArchitecturalElements*, and it classifies architectural elements into components and connectors. As a result, this package specifies that components and connectors inherit the properties of the *ArchitecturalElement* metaclass, and it also contains the packages that define components and connectors.

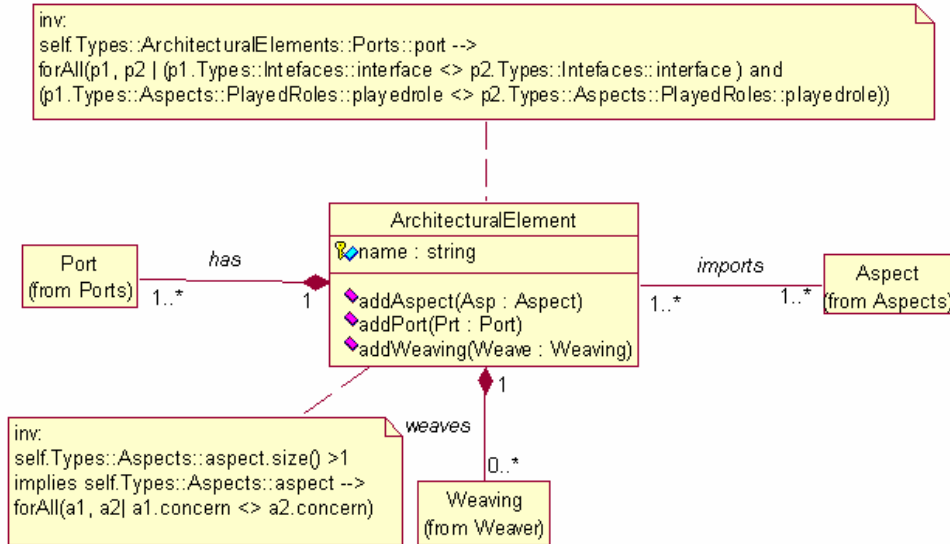


Figure 32. The package *ArchitecturalElements* of the PRISMA metamodel

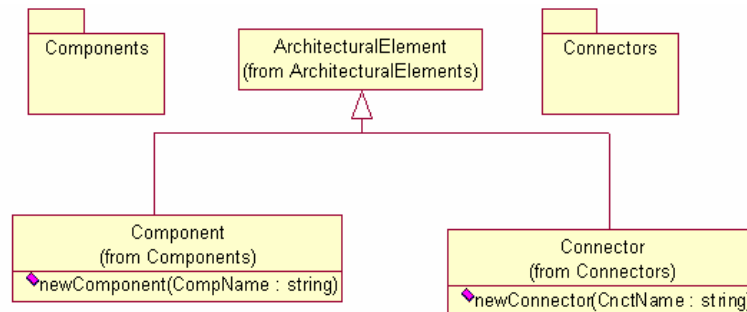


Figure 33. The package *KindsOfArchitecturalElements* of the PRISMA metamodel

4.2.1.4. The package “Weaver”

The package *Weaver* defines the weavings of architectural elements. It contains the metaclass *weaving* which is formed by two aspect services. One of the services, the pointcut service, triggers the execution of the weaving; the other service, the advice service, is executed as a consequence of the weaving. The relationships between the weaving and these two services are modelled in the metamodel by means of two aggregations (see Figure 34). In addition, if the weaving is conditional, it has a condition associated to it.

There are certain constraints that must be satisfied in order to associate the appropriate services to a weaving definition. As a result, the metaclass *Weaving* has constraints associated to it. These constraints correspond to the OCL rules shown in Figure 35. They specify the following:

<<The services that participate in the weaving must belong to aspects that are imported by the architectural element in which the weaving is defined>>

<<If the weaving uses a conditional operator, it must have a condition associated to it. However, if the weaving does not use a conditional operator, it cannot have a condition associated to it>>

<<The services that participate in a weaving must belong to different aspects>>

<<The aspects of the services that participate in a weaving must define different concerns>>

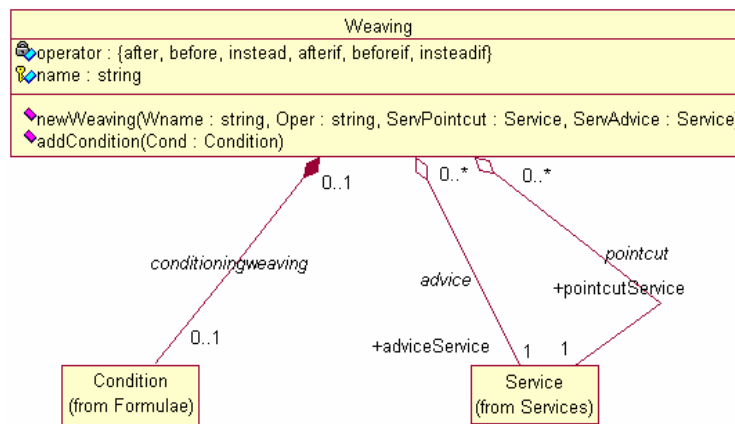


Figure 34. The package *Weaver* of the PRISMA metamodel

The metaclass *Weaving* has two attributes, *name* and *operator* (see Figure 34). These store the name of the aspect and the operator that the weaving applies to the service execution, respectively. The service *newWeaving* creates a new aspect; whose parameters define the name, the operator of the weaving, and the two services that participate in the weaving. In addition, the metaclass provides a service for adding a condition to a weaving when it uses a conditional operator. This service is called *addCondition*.

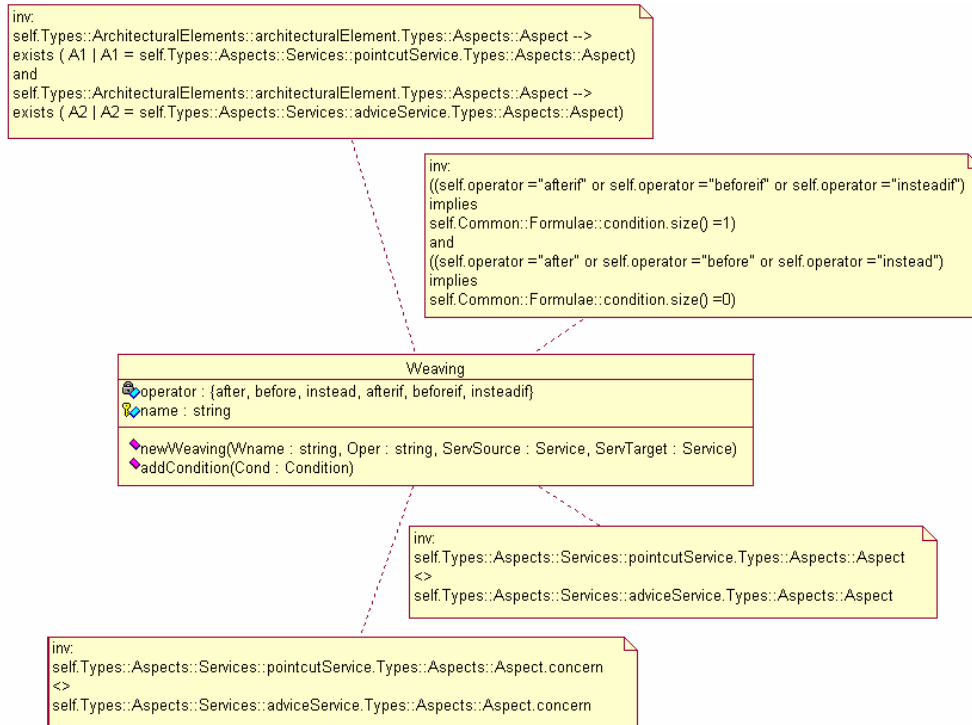


Figure 35. Constraints of the metaclass *Weaving*

4.2.1.5. The package “Components”

The package *Components* defines simple and complex components (see Figure 36). Since components cannot be coordinators of the software system, there is a constraint that specifies that a component cannot import an aspect whose concern is coordination.

The metaclass *Component* provides a service to create components. This service is called *newComponent*, and its parameter is the name of the component that is created as a result of the service execution.

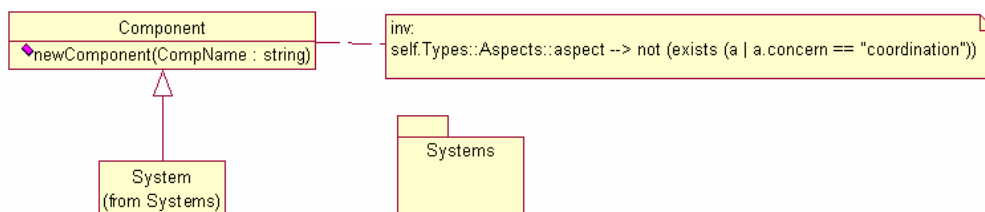


Figure 36. The package *Components* of the PRISMA metamodel

Since systems are complex components, they inherit all the properties of components. For this reason, the package that defines a system is a subpackage of the *Component* metaclass.

4.2.1.6. The package “Connectors”

The package *Connectors* defines the connector architectural element (see Figure 37). Since connectors act as coordinators of components, the metaclass *Connector* has an associated constraint that specifies that a connector must import an aspect whose concern is coordination (see the first constraint that appears in Figure 37).

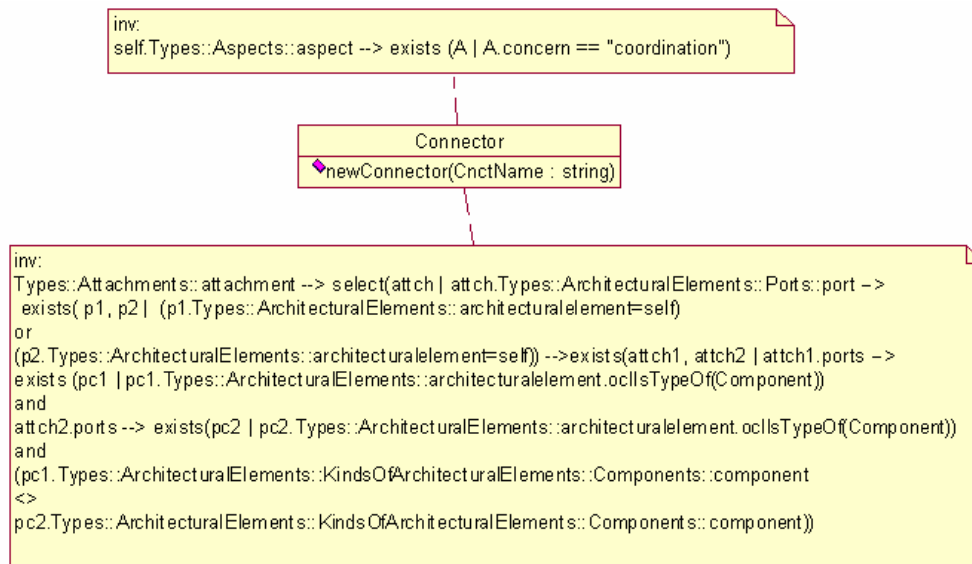


Figure 37. The package *Connectors* of the PRISMA metamodel

Moreover, the metaclass *Connector* has another constraint associated to it that specifies the following:

<<A connector must have at least two attachments associated to it, and each attachment must connect the connector to two different components>>>

The metaclass *Connector* provides a service to create connectors. This service is called *newConnector*, and its parameter is the name of the connector that is created as a result of the service execution

4.2.1.7. The package “Attachments”

Attachments define types of communication channels between the ports of a component and the port of a connector. As a result, the metaclass *Attachment* is related to the metaclass *Port* by means of an association relationship (see Figure 42). This relationship establishes that the attachment must be related to two ports. However, it is necessary to constrain this association with a constraint in order to establish that one of the ports must belong to a component and that the other one must belong to a connector.

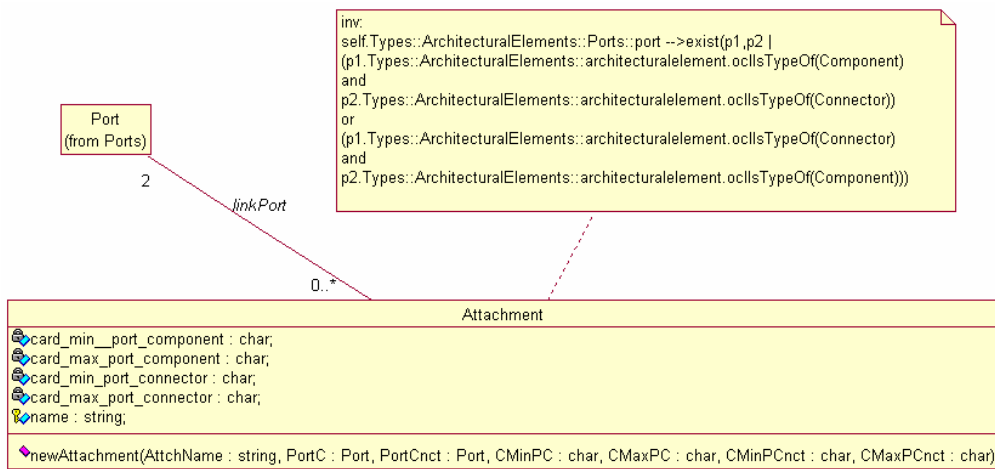


Figure 38. The package *Attachments* of the PRISMA metamodel

In addition to the attachment name for storing the name of the attachment, the metaclass *Attachment* has four more attributes to specify the attachment communication pattern, i.e., the instantiation pattern of the attachment. It is necessary to constrain how many instances of the attachment can be attached to the port of the component instance and the port of the connector instance. The attribute *card_min_port_component* specifies the minimum number of attachment instances that must be connected to one instance of this *component* through the *port*. The attribute *card_max_port_component* specifies the maximum number of attachment instances that must be connected to one instance of this *component* through the *port*. The attribute *card_min_port_connector* specifies the minimum number of attachment instances that must be connected to one instance of this *connector* through the *port*. The attribute

card_max_port_connector specifies the maximum number of attachment instances that must be connected to one instance of this *connector* through the *port*.

Moreover, the metaclass *Attachment* has the service *newAttachment* to create a new attachment. Its parameters are the name of the attachment that is created as a result of the service execution, the component port and the connector port that it connects, and the minimum and maximum cardinalities for each one of the ports.

4.2.1.8. The package “Systems”

The package *Systems* defines complex components (see Figure 39). Systems are complex components that are composed of a set of architectural elements and their attachments. For this reason, the metaclass *System* has an aggregation with each one of the metaclasses *Component*, *Connector* and *Attachment*.

The architectural elements that compose a system can be directly related to other elements or their access can only be possible through the system. These two kinds of composition are referential and inclusive, respectively. The analyst can model any of these compositions for the architectural elements of the system, depending on the requirements of the system. However, the definition of an inclusive composition between a system and an architectural element requires the definition of a channel between a system port and a port of the architectural element. This channel is required to resend the provided and requested services of the architectural element through the system port. These channels are called *bindings*. Therefore, the metaclass *System* has an aggregation relationship with the metaclass *Binding*. Bindings are defined in the *Bindings* subpackage of the *System* package.

The metaclass *System* must be related to at least one component in order to be complex (see the *containsComp* aggregation in Figure 39). In addition, it has an associated constraint that ensures a correct composition. It specifies the following:

<<If a system does not import any aspect, the system must have at least one binding associated to it >>

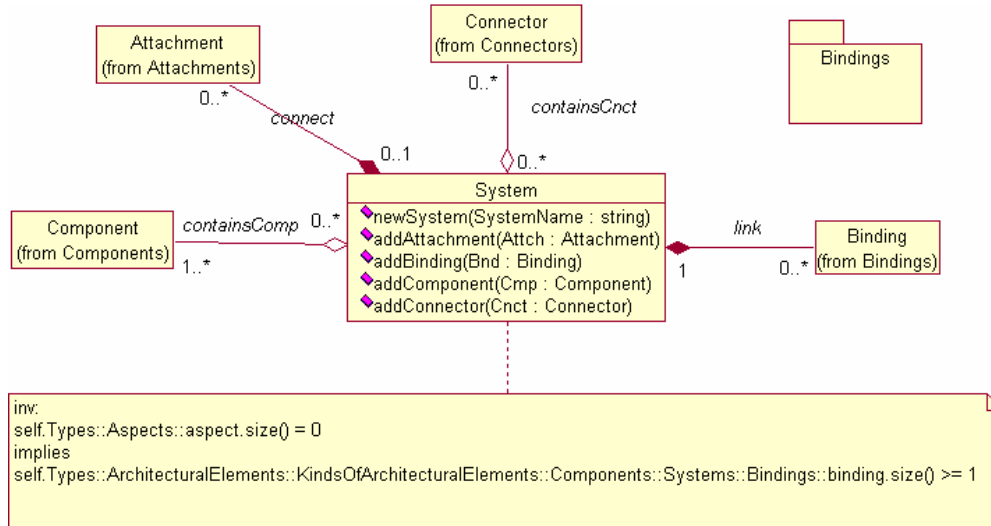


Figure 39. The package *Systems* of the PRISMA metamodel

The metaclass *System* has four services. The service *newSystem* creates a new system by providing the name of the system as a parameter. In addition, the services *addComponent*, *addConnector*, *addAttachment* and *addBinding* add components, connectors, attachments and bindings, respectively, to the system.

4.2.1.9. The package “*Bindings*”

The metaclass *Binding* is related to the metaclass *Port* by means of two association relationships (see Figure 42). These associations establish that the binding must be related to a system port and an architectural element port. However, the architectural element that the port belongs to must be one of the architectural elements of the system. This constraint is applied not only at the types level, but also at the configuration level (instances). For this reason, the metaclass *Binding* has an associated constraint that specifies this requirement.

In addition to the attribute *name* for storing the name of the binding, the metaclass *Binding* has four more attributes to specify the communication pattern of the binding. As a result, the attribute *card_min_port_AR* specifies the minimum number of binding instances that must be connected to one instance of this architectural element through the *port*. The attribute *card_max_port_AR* specifies the maximum number of binding instances that must be

connected to one instance of this architectural element through the *port*. The attribute *card_min_port_Sys* specifies the minimum number of binding instances that must be connected to one instance of this *system* through the *port*. The attribute *card_max_port_Sys* specifies the maximum number of binding instances that must be connected to one instance of this system through the *port*.

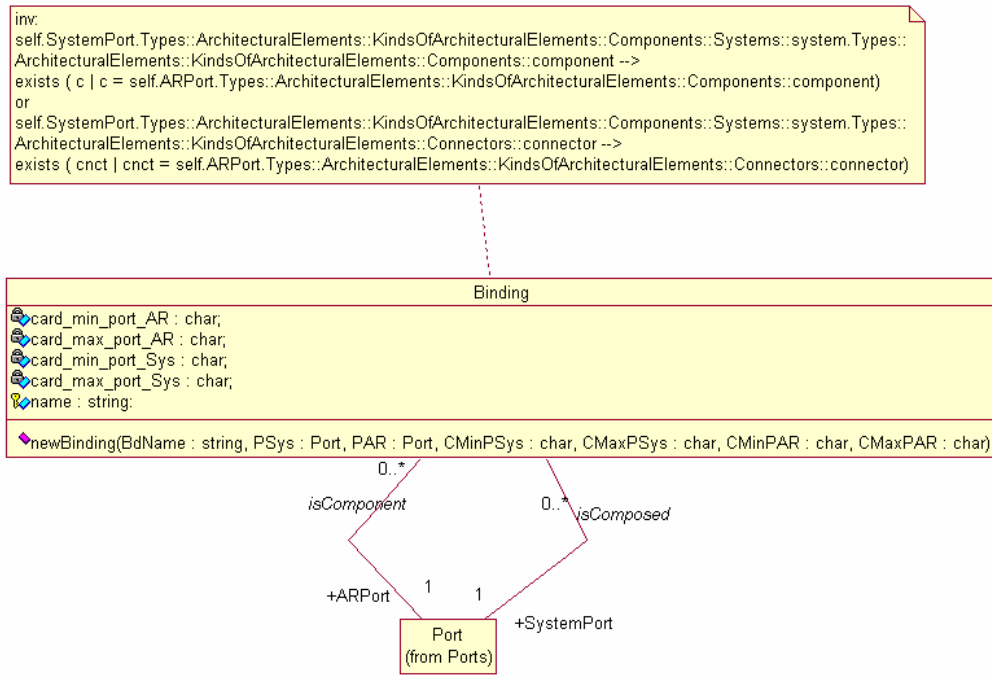


Figure 40. The package *Bindings* of the PRISMA metamodel

Moreover, the metaclass *Binding* has the service *newBinding* to create a new binding. Its parameters are the name of the binding that is created as a result of the service execution, the system port and architectural element port that it connects, and the minimum and maximum cardinalities for each one of the ports.

4.2.1.10. The package “Ports”

Ports publish the services of an interface and constrain how these services can be provided or requested by means of a played_role. For this reason, the metaclass *Port* has two aggregation relationships with the metaclasses *Interface* and *Played_Role* (see Figure 41).

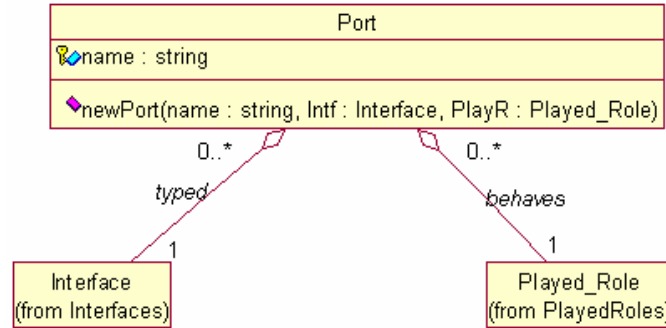


Figure 41. The package *Ports* of the PRISMA metamodel

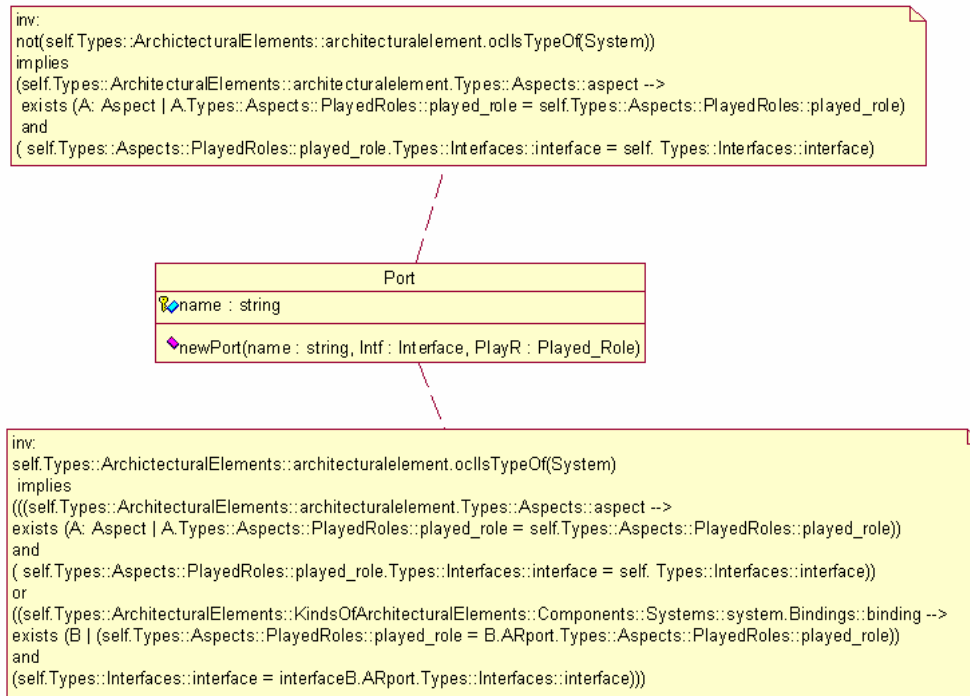


Figure 42. Constraints of the metaclass *Port*

However, ports cannot be related to any interface or played_role of the software system. As a result, these relationships are constrained by the following two constraints that correspond to the two that appear in Figure 42:

<<If the architectural element that the port belongs to is not a system, the played_role of the port must be defined for one of the aspects that the architectural element imports. In addition, the interface of the played_role must be the same one as the interface of the port>>

<<If the architectural element that the port belongs to is a system, there are two possible options: 1) either the played_role of the port is defined for one of the aspects that the system imports, and the interface of the played_role is the same as the interface of the port; 2) or the port has the same interface and played_role as one port of the architectural element of the system>>

The metaclass *Port* has one attribute, the *name* of the port. The service *newPort* creates a new port by providing the name of the port, the interface and the *played_role* as parameters.

4.2.2. THE PACKAGE “ARCHITECTURE SPECIFICATION”

The package *Architecture Specification* defines how a PRISMA architecture can be defined using the types defined in the package *Types*. The metaclass *PRISMAArchitecture* has five aggregation relationships with each one of the first-order citizens of the PRISMA model. They are components, connectors, aspects, interfaces, and attachments. Since components, connectors, interfaces, and aspects are reusable, they can be used by more than one architectural element (see Figure 43).

The metaclass *PRISMAArchitecture* has one attribute, the *name* of the architectural model. The service *newArchitecture* creates a new architectural model by providing its name as a parameter. In addition, the metaclass provides five services to add attachments, components, connectors, interfaces and aspects to the architectural model. In order to ensure that a model is correctly defined, the metaclass *PRISMAArchitecture* has a set of constraints associated to it (see Figure 43). Their meaning is the following:

<<An architectural model must include every aspect that is imported by its components and/or connectors>>

<<An architectural model must include every interface that is used by its aspects >>

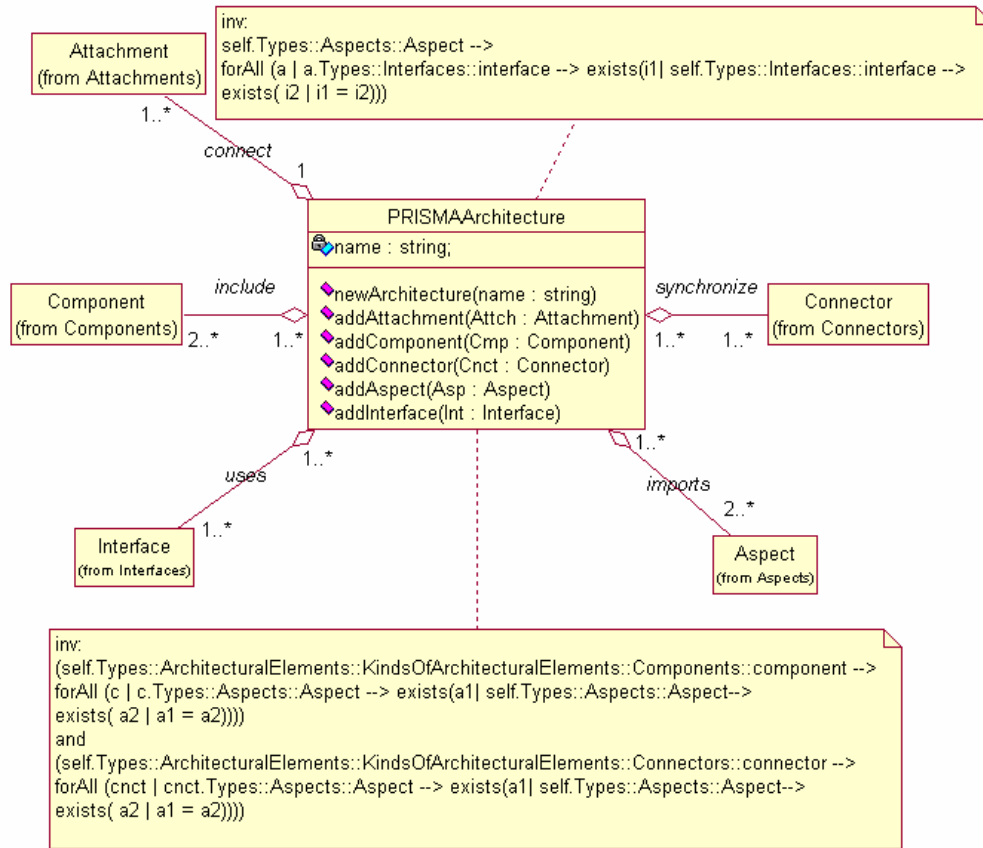


Figure 43. The package *Architecture Specification* of the PRISMA metamodel

4.3.CONCLUSIONS

The PRISMA model has been presented in this chapter. This model allows us to describe software architectures of complex and large systems and to improve their reusability and maintainability. This is possible because the application of aspect-oriented software development to software architectures provides different levels of reusability and maintenance: the concern level (aspects) and the functional level (architectural elements).

- The concern level places the properties of a concern inside an aspect. As a result, the modification of a concern is easily found in the aspects of this concern, and the aspects of a concern can be reused by any architectural element that needs their properties.

- The functional level places functional or coordination processes of the business rules of the software system in components and connectors, respectively. These can be easily found in order to be reused or modified.

Another important property of this model is the fact that the weavings between aspects and the relationships among architectural elements are defined outside aspects, which improves their reusability and maintenance.

Instead of using a kernel or core entity to encapsulate functionality, aspects to define non-functional requirements and their weavings, this model only uses aspects and weavings to define architectural elements. The symmetrical way in which aspects are introduced in PRISMA software architectures provides homogeneity to the model and a clean and novel way of modelling software architectures. In PRISMA, architectural elements and aspects are used as if they were pieces of a puzzle that fit together to form a software architecture. This way of specifying PRISMA software architectures is presented in detail in chapter.

This chapter also has presented the PRISMA metamodel. This metamodel permits the creation of PRISMA architectural models in a correct way and the accurate definition of their properties following the MDD paradigm. In addition, it facilitates the integration of the PRISMA model into modelling tools that support the incorporation of new metamodels.

The PRISMA metamodel defines the required metaclasses, their properties and services, and their relationships with each other. In addition, the metamodel specifies the constraints to ensure that the definition of an architectural model is correct.

The metamodel is the repository structure that stores PRISMA architectural models, preserving the reusability of interfaces, architectural elements and aspects. In addition, the metamodel introduces a methodology to follow during the MDD process when a PRISMA architectural model is specified by means of constraints. This is supported by verification rules that are associated to the MDD process.

The PRISMA metamodel has been defined to be able to support evolution at run-time in the future. This evolution could be supported at different levels of granularity by adding the evolution services to the different metaclasses of the metamodel. This would consist of adding or removing architectural elements of the model, or adding or removing properties of an aspect

(attributes, services, etc). In addition to adding evolution services to the metamodel, a mechanism to invoke these services at run-time should be provided to support run-time evolution.

The work related to PRISMA model has produced a set of results that are published in the following publications:

- Nour Ali, **Jennifer Pérez**, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, *Replicación Distribuida en Arquitecturas Software Orientadas a Aspectos utilizando Ambientes*, Journal IEEE América Latina, Vol. 5, Issue 4, July 2007.
- **Jennifer Pérez**, Nour Ali, Jose Ángel Carsí, Isidro Ramos, *Designing Software Architectures with an Aspect-Oriented Architecture Description Language*, 9th Symposium on the Component Based Software Engineering (CBSE), Springer Verlag LNCS 4063 ,pp. 123-138, ISSN: 0302-9743, ISBN: 3-540-35628-2, Vasteras, Suecia, June 29th-July 1st, 2006.
- Nour Ali, **Jennifer Pérez**, Cristóbal Costa, Isidro Ramos, José Ángel Carsí, *Mobile Ambients in Aspect-Oriented Software Architectures*, IFIP Working Conference on Software Engineering Techniques: Design for Quality- SET 2006, Springer, Volume 227 pp. 37-48, ISSN: 1571-5736, ISBN: 0-387-39387-0, Warsaw, Poland, October, 17-20, 2006.
- **Jennifer Pérez**, Elena Navarro, Patricio Letelier, Isidro Ramos, *A Modelling Proposal for Aspect-Oriented Software Architectures*, 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), IEEE Computer Society , pp.32-41, ISBN: 0-7695-2546-6, Potsdam, Germany (Berlin metropolitan area), March 27th-30th, 2006.
- **Jennifer Pérez**, Elena Navarro, Patricio Letelier, Isidro Ramos, *Graphical Modelling for Aspect Oriented SA*, 21st Annual ACM Symposium on Applied Computing, ACM, pp. 1597-1598, ISBN: 1-59593-108-2, Dijon, France, April 23 -27, 2006. (short paper)

- **Jennifer Pérez**, Manuel Llavador, Jose A. Carsí, Jose H. Canós, Isidro Ramos, *Coordination in Software Architectures: an Aspect-Oriented Approach*, Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society Press, pp. 219-220, ISBN: 0-7695-2548-2, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper)
- Nour Ali, **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *Integrating Ambient Calculus in Mobile Aspect-Oriented Software Architectures*, Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society Press, pp. 233-234, ISBN: 0-7695-2548-2, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper)
- **Jennifer Pérez**, Nour Ali, Cristobal Costa, José Á. Carsí, Isidro Ramos, *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*, 3rd International Conference on .NET Technologies, pp. 97-108, Pilsen, Czech Republic, May-June 2005.
- Nour Ali, **Jennifer Pérez**, Isidro Ramos, *Aspect High Level Specification of Distributed and Mobile Information Systems*, Second International Symposium on Innovation in Information & Communication Technology ISSICT, pp. 14, Amman, Jordania, 21-22, April, 2004.
- Nour Ali, **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *Aspect Reusability in Software Architectures*, 8th International conference of Software Reuse (ICSR), July, 2004 (poster)
- Elena Navarro, Isidro Ramos, **Jennifer Pérez**, *Goals Model-Driving Software Architecture*, 2nd International Conference on Software Engineering Research, Management and Applications (SERA), pp. 205-212, ISBN: 0-9700776-9-6, May 5-8, 2004, Los Angeles, CA, USA.
- Nour Hussein, Josep Silva, Javier Jaen, Isidro Ramos, Jose Ángel Carsí, **Jennifer Pérez**, *Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures*, 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS), ACTA Press, ISBN: 0-88986-392-X, ISSN: 1027-2658, pp. 820-826, Marina del Rey, California, USA, 3-5, November 2003,

- **Jennifer Pérez** , Isidro Ramos , Javier Jaén, Patricio Letelier, Elena Navarro , *PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*, 3rd IEEE International Conference on Quality Software (QSIC 2003), IEEE Computer Society Press, pp.59-66, ISBN: 0-7695-2015-4, Dallas, Texas, USA, November 6 - 7, 2003.
- Elena Navarro, Isidro Ramos, **Jennifer Pérez**, *Software Requirements for Architected Systems*, 11th IEEE International Requirements Engineering Conference (RE'03), IEEE Computer Society Press, pp. 365-366, ISSN: 1090-705X, ISBN: 0-7695-1980-6, Monterey, California, 8-12 September 2003 (Poster)
- **Jennifer Pérez**, Nour Ali, Jose A. Carsí, Isidro Ramos, *Dynamic Evolution in Aspect-Oriented Architectural Models*, Second European Workshop on Software Architecture, Springer LNCS 3527, pp.59-16, ISSN: 0302-9743, ISBN: 3-540-26275-X , Pisa, Italy, June 2005.
- **Jennifer Pérez** , Isidro Ramos , Javier Jaén, Patricio Letelier, *PRISMA: Development of Software Architectures with an Aspect Oriented, Reflexive and Dynamic Approach*, Dagstuhl Seminar N° 03081, Report N° 36 "Objects, Agents and Features", Copyright (c) IBFI gem. GmbH, Schloss Dagstuhl, D-66687 Wadern, Germany . Eds.H.-D. Ehrich (Univ. Braunschweig, D), J.-J. Meyer (Utrecht, NL), M. Ryan (Univ. of Birmingham, GB), pp. 16, Germany, January, 2003.
- Jorge Ferrer, Ángeles Lorenzo, Isidro Ramos, José Ángel Carsí, **Jennifer Pérez**, *Modeling Dynamic Aspects in Architectures and Multiagent Systems*, Logic Programming and Software Engineering (CLPSE), pp. 1-13, Copenhagen, Denmark, affiliated with ICLP, july 2002.
- Nour Ali, **Jennifer Pérez**, Cristóbal Costa, Isidro Ramos, José Ángel Carsí, *Distributed Replication in Aspect-Oriented Software Architectures using Ambients*, XI Conference on Software Engineering and Databases (JISBD), pp. 379-388, ISBN: 84-95999-99-4, Sitges, Barcelona, October 2006. (In Spanish)

- Cristóbal Costa, **Jennifer Pérez**, Nour Ali, Jose Angel Carsí, Isidro Ramos, *PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures*, X Conference on Software Engineering and Databases (JISBD), pp. 27-34, ISBN: 84-9732-434-X, Granada, September, 2005. (In Spanish)
- **Jennifer Pérez**, Nour H. Ali, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, *Development of a Tele-Operation System using the PRISMA Approach*, VIII Conference on Software Engineering and Databases (JISBD), pp. 411-420, ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)
- **Jennifer Pérez**, Isidro Ramos, Ángeles Lorenzo, Patricio Letelier, Javier Jaén, *PRISMA: OASIS PlatfoRm for Architectural Models*, VII Conference on Software Engineering and Databases (JISBD), pp. 349-360, ISBN: 84-688-0206-9, El Escorial (Madrid), November, 2002. (In Spanish)
- Cristóbal Costa, **Jennifer Pérez**, Jose Angel Carsí, *Towards the Dynamic Configuration of Aspect-Oriented Software Architectures*, IV Workshop on Aspect-Oriented Software Development (DSOA), XI Conference on Software Engineering and Databases (JISBD), Technical Report TR-24/06 of the Polytechnic School of the University of Extremadura, pp.35-40, Sitges, Barcelona, Octubre, 2006. (In Spanish)
- Rafael Cabedo, **Jennifer Pérez**, Nour Ali, Isidro Ramos, Jose A. Carsí, *Aspect-Oriented C# Implementation of a Tele-Operated Robotic System*, III Workshop on Aspect-Oriented Software Development (DSOA), X Conference on Software Engineering and Databases (JISBD), pp. 53-59, ISBN: 84-7723-670-4, Granada, September, 2005. (In Spanish)
- **Jennifer Pérez**, Nour H. Ali, Isidro Ramos, Jose A. Carsí, *PRISMA: Aspect-Oriented and Component-Based Software Architectures*, Workshop on Aspect-Oriented Software Development (DSOA), Conference on Software Engineering and Databases (JISBD), Technical Report TR-20/2003 of the Polytechnic School of the University of Extremadura, pp. 27-36, Alicante, November, 2003. (In Spanish)

- M^a Eugenia, Nour Ali, **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *DIAGMED: An Architectural model for a Medical Diagnosis*, IV workshop DYNAMICA – DYNAmic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 1-7, Archena, Murcia, November, 2005. (In Spanish)
- Rafael Cabedo, **Jennifer Pérez**, Jose A. Carsí, Isidro Ramos, *Generation and Modelling of PRISMA Architecture using DSL Tools*, IV Workshop DYNAMICA – DYNAmic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp.79-86, Archena, Murcia, November, 2005. (In Spanish)
- Nour Ali, **Jennifer Pérez**, Jose A. Carsí, Isidro Ramos, *Mobility of Objects in the PRISMA Approach*, II workshop DYNAMICA – DYNAmic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 111-118, Almagro, Ciudad Real, April, 2005.
- **Jennifer Pérez**, Rafael Cabedo, Pedro Sánchez, Jose A. Carsí, Juan A. Pastor, Isidro Ramos, Bárbara Álvarez, *PRISMA Architecture of the Case Study: an Arm Robot*, II workshop DYNAMICA – DYNAmic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November 2004. (In Spanish)
- Nour Ali **Jennifer Pérez**, Cristobal Costa, Jose A. Carsí, Isidro Ramos, *Implementation of the PRISMA Model in the .Net Platform*, II workshop DYNAMICA – DYNAmic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- Nour H. Ali, Josep F. Silva, Javier Jaen, Isidro Ramos, Jose Á. Carsí, **Jennifer Pérez**, *Distribution Patterns in Aspect-Oriented Component-Based Software Architectures*, IV Workshop Distributed Objects, Languages, Methods and Environments (DOLMEN), pp.74-80, Alicante, November, 2003.
- Rafael Cabedo, **Jennifer Pérez**, Isidro Ramos, *The application of the PRISMA Architecture Description Language to an Industrial Robotic System*, Technical Report, DSIC-II/11/05, pp.180, Polytechnic University of Valencia, September 2005. (In Spanish)

- Cristobal Costa, **Jennifer Pérez**, Jose Ángel Carsí, *Study and Implementation of an Aspect-Oriented Component-Based Model in .NET technology*, Technical Report, DSIC-II/12/05, pp. 198, Polytechnic University of Valencia, September, 2005. (In Spanish)
- **Jennifer Pérez**, Nour Ali , Jose A. Carsí, Isidro Ramos, *PRISMA Architecture of the Robot 4U4 Case Study*, Technical Report DSIC-II/13/04, pp. 72, Polytechnic University of Valencia, 2004. (In Spanish)
- **Jennifer Pérez**, Isidro Ramos, *OASIS as a Formal Support for the Dynamic, Distributed and Evolutive Hypermedia Models*, Technical Report DSIC-II/22/03, pp. 144, Polytechnic University of Valencia, October 2003. (In Spanish)
- **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *A Compiler to Automatically Generate the Metalevel of Specifications using Properties of the Base Level*, Technical Report, DSIC-II/23/03, pp. 107, Polytechnic University of Valencia, October, 2003.(In Spanish)

CHAPTER 5

COORDINATION

Coordination has become a key concept in the modelling process of industrial systems as it leads to a better understanding of the interactions that take place in complex and distributed systems. In the last few years, coordination has been introduced in two important fields of Software Engineering: Software Architectures, through the notion of connector, and Aspect-Oriented Software Development, through the notion of weaving and by considering coordination as an aspect. The separation of coordination from functionality is a key concept in order to provide a better reusability and maintenance during the MDD process.

In this chapter the interest of using aspect-oriented connectors is discussed in detail, justifying the relevance of the PRISMA model and its merits with regard to other proposals, especially to provide a complete MDD support. Once this interest has been stated, the concrete structure of connectors in PRISMA is described in detail, and the formalization of the relevant PRISMA concepts is provided and explained. The notion of weavings is described with special care. In addition, a discussion of the proposal is done by comparing it with other works and highlighting the advantages of the PRISMA proposal. Finally, the chapter concludes by summarizing the results and the directions of further work about PRISMA coordination processes.

5.1. INTRODUCTION

Currently, there is a great interest in coordination. Coordination orchestrates processes in order to achieve the correct functionality of software products. Good coordination management is essential and is a risk factor for the synchronization of difficult tasks that industrial systems must perform. As a result, several software development approaches have taken coordination into account. Two widely used are Component-Based Software Development (CBSD) [Szy98] and Aspect-Oriented Software Development (AOSD) [Kiz97].

On the one hand, coordination is an important topic in CBSD and, by extension, in Software Architectures since it can be used to synchronize the components that form a specific architecture. In fact, as it is presented in chapter 3, Architecture Description Languages (ADLs) [Med00] could be classified according to the importance they give to coordination. Some of these ADLs have introduced the notion of connector, which is an architectural element that acts as a coordinator among other architectural elements (either connectors or components) [All97a], [Cue05], [Med00]. However, other ADLs do not include connectors [Can99], [Mag95]. Those that use the notion of connector give more relevance to coordination because they provide a specific architectural element to define it. In addition, they offer an architectural view of systems; whereas, an ADL without connectors has a more compositional view, as in object-oriented models [Luc95b], [Mag95]. As a result, an ADL should provide connectors in order to separate coordination from computation and to provide an architectural view instead of a compositional view.

On the other hand, AOSD allows the separation of cross-cutting concerns of software systems in a modular entity called aspect. Among the different crosscutting concerns that can be identified in software systems, coordination is perhaps one of the most common. But in addition to this characterization as a concern, coordination has also emerged as an important feature within AOSD itself, because the different aspects of a software system must also be synchronized. The need for aspect coordination has been identified as a key feature in this approach [Kiz01].

The main contribution of this chapter is the formalization and definition of aspect-oriented connectors in order to define their coordination process in a formal way and thus, to avoid

ambiguity in the code generation process. In addition, its definition must provide reusability and avoid the replication of specification to facilitate the development and maintenance of software following the MDD paradigm.

Since PRISMA connectors are observable processes that have state and behaviour, the formalisms which are used to formalize the PRISMA model are a variant of a Modal Logic of Actions [Sti92], and an extension of the π -calculus [Mil93] which provides priorities. The π -calculus is a process algebra which is used to specify and formalize the processes of the PRISMA model, and the Modal Logic of Actions is used to formalize the way in which the execution of these processes affects the state of architectural elements. More detail about these formalisms can be found in [Per06c].

5.2.ASPECT ORIENTED CONNECTORS

It is important to keep in mind that current software systems perform complex coordination processes that have to take into account not only the coordination concern, but also other concerns such as: safety, distribution, security, etc. These other concerns are necessary in order to provide a correct coordination process. For example: The connectors that coordinate the actuators and sensors of tele-operated robots need to check that the movement is safe for the robot before sending the movement to the actuator. PRISMA aspect-oriented connectors are presented as a solution for the specification of these complex coordination processes by improving the reusability and maintenance of software during the MDD process. This improvement has been achieved by overcoming the disadvantages of the rest of ADLs. Current ADLs can be classified into three different kinds: non-aspect-oriented ADLs without connectors, non-aspect-oriented ADLs with connectors, and aspect-oriented ADLs. Next, it is presented how they specify complex coordination processes that have to take into account several concerns.

5.2.1. *Non-Aspect-Oriented, connector less ADLs*

There are ADLs that prefer the absence of connectors because they distort the compositional nature of software architectures. Some ADLs, such as Darwin [Mag95], [Mag96]Leda [Can01], [Can00], and Rapide [Luc95b], [Luc95a], [Ken95] do not consider connectors as

first-class citizens. However, these ADLs make difficult the reusability of components because they have the coordination process tangled with the computation inside them, and they are aware of the coordination process that has to happen in order to communicate with the rest. The notion of connector emerges from the need to separate the interaction from the computation in order to obtain more reusable and modularized components and to improve the level of abstraction of software architecture descriptions. Mary Shaw [Sha94] 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-order citizens of ADLs. Figure 44 illustrates how two components (actuator and sensor) are communicated using an ADL without connectors. The coordination process is encapsulated in the components and tangled with the computation and other concerns.

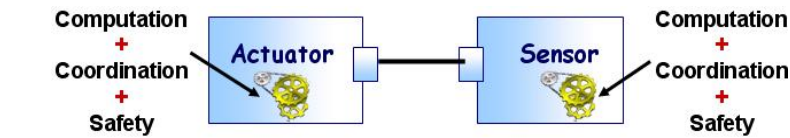


Figure 44. Sensor-Actuator coordination by using a Connector-less ADL

5.2.2. *Non-Aspect-Oriented ADLs with connectors*

Most ADLs provide connectors as a first order citizens of the language such as: ACME [Gar00], Aesop [Gar94], [Gar95b], C2 [Med96], [Med99], SADL [Mor95], [Mor97], UniCon [Sha95], [Sha96], Wright [All97a], [All97b], CommUnity [Lop05], [And03], [Fia04][9], Pilar [Cue02], [Cue04], ArchWare π -ADL [Oqu04a], [Oqu04b], etc. All of these languages go a step forward with regard to the previous kind of ADLs. They improve the reusability of components and connectors by separating computation from coordination. However, their connectors are non-aspect-oriented and they specify their coordination processes by tangling the code inside them. For example: the coordination process between an actuator and a sensor

of a robot will imply the specification of a connector with tangled concerns of coordination and safety (see Figure 45).

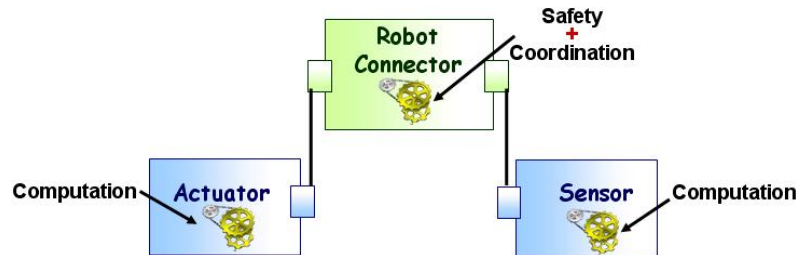


Figure 45. Sensor-Actuator coordination by using an ADL with Connectors

5.2.3. Aspect-Oriented, connector-less ADLs

Most aspect-oriented approaches applied to software architectures and their ADLs are based on an original ADL without connectors such as: PCS [Kan02b][Kan02a][Kan03], DAOP-ADL [Pin03][Pin05], AspectLEDA [Nav05], AOCE [Gru00], etc. These ADLs introduce the aspect-oriented behaviour by means of connectors, i.e., aspects are connectors among components. However, when there are two components that are coordinated by several connectors (aspects), the connectors cannot be synchronized among them (weavings among aspects). And in those ADLs that could try to solve this problem by connecting both connectors they will lose the reusability of the concerns of those connectors, because they will be dependent to the connector (aspect) that are connected to. Figure 46 illustrates how two components are communicated using an aspect oriented ADL without connectors.

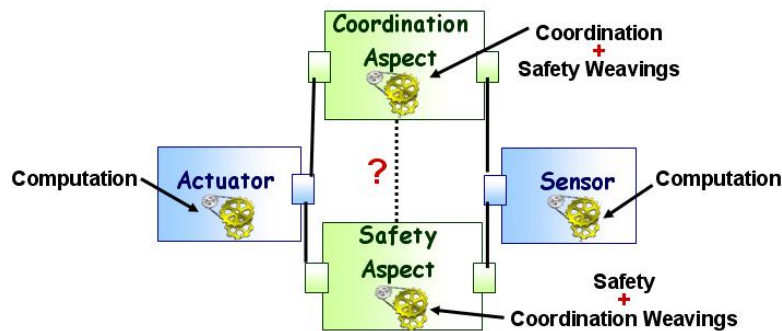


Figure 46. Sensor-Actuator coordination by using a connector-less Aspect-Oriented ADL (AOADL)

5.2.4. *Aspect-Oriented ADLs with connectors*

However, in PRISMA a new kind of ADLs is introduced, namely aspect-oriented ADLs with connectors. PRISMA is based on an ADL with connectors, and aspects are introduced as a new concept in software architectures for concerns called aspects. As a result, each concern is specified in its aspect and the coordination rules among the different aspects are inside the connector being aspects reusable and independent one to each other. Figure 47 presents how PRISMA coordinates the sensor and the actuator by separating the concerns or computation, safety and coordination. As a result, they are not scattered through the architecture and they are not repeated. These properties are the base to improve the reusability and maintenance of software during the Model-Driven Development of aspect-oriented software architectures.

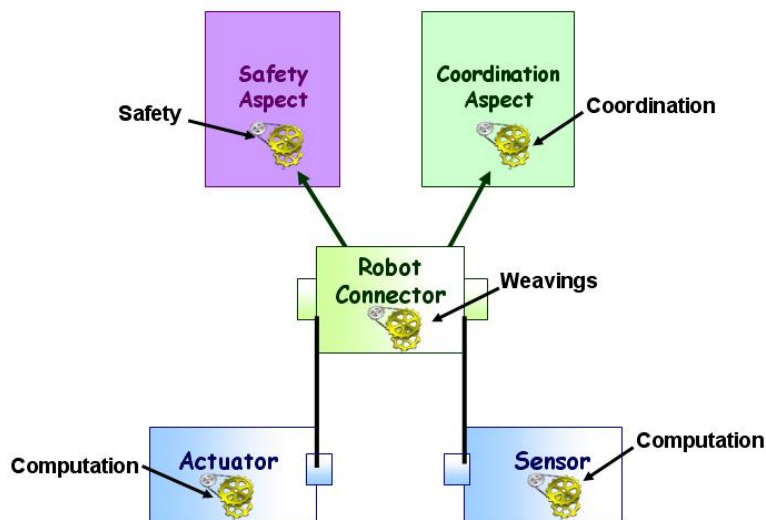


Figure 47. Sensor-Actuator coordination by using the PRISMA ADL

In addition, Figure 47 shows that the coordination process among components, connectors and aspect is very complex. For this reason, this coordination process must be very well defined and formalized in order to guarantee that it coordinates all the pieces of software successfully. The formalization of this coordination process is presented in detail along the next section.

5.3.CONNECTORS IN PRISMA

A connector is an architectural element that acts as a coordinator between other architectural elements. As such, connectors have a coordination aspect. An example is the connector that synchronizes the Actuator and the Sensor of a robot joint. This connector imports a safety aspect and a coordination aspect to coordinate the movements of the robot in a safe way for the joint, the robot and the environment that surrounds it.

5.3.1. *Architectural Element*

Since a connector is an architectural element, a connector is formalized as an architectural element. An architectural element is formed by a set of aspects, their weaving relationships, and one or more ports. These ports represent interaction points among architectural elements.

- *Formalization: Architectural Element*

An architectural element AE is built by composing a set of aspects A_1, A_2, \dots, A_n , which are conceived as the smallest modules in our approach, and will be defined in section 5.3.3. The resulting element AE is in turn defined itself by the 4-tuple (A, X, Φ, Π) , as follows:

- A: the set of attributes in aspects A_1, A_2, \dots, A_n
- X: the set of the services in aspects A_1, A_2, \dots, A_n (see Definition 1 in section 5.3.4)
- Φ : the set of formulae (in Modal Logic of Actions) providing constraints for aspects $A_1 \dots A_n$
- Π : the process P_{AE} defined as follows:

$$P_{AE} ::= P_{P1} || \dots || P_{Pm} || P_{A1} || \dots || P_{An} || P_W$$

This means that the processes of the ports, weavings and aspects of the architectural element are executed concurrently. For this reason, PAR is defined as their parallel composition, and therefore their dependencies are expressed and solved just as concurrency conflicts.



Fig. 49 (a). The black box representation

```

Connector CnctJoint

Coordination Aspect Import CProcessSuc;
Safety Aspect Import SMotion;

Weavings
    SMotion. DANGEROUSCHECKING(NewSteps, Speed, Secure)
    beforeif (Safe = true)
        CProcessSuc.movejoint(NewSteps, Speed);

End_Weavings;

Ports
    PAct : IMotionJoint,
        Played_Role CProcessSuc.ACT;
    PSen : IRead,
        Played_Role CProcessSuc.SEN;
    PJoint : IJoint,
        Played_Role CProcessSuc.JOINT;
    PPos : IPosition,
        Played_Role CProcessSuc.POS;
End_Ports

... ..
End Connector CnctJoint;
    
```

Fig. 49 (b). PRISMA specification

Figure 48. The *RobotConnector* Connector

A brief comment about the role of the Modal Logic of Actions in PRISMA is relevant here. Basically, the formulae in Φ are used for implementing obligations, prohibitions, and permissions, providing the concurrent equivalent of a deontic logic. As a result, it permits the

analysis and formulation of assertions about processes that change the execution environment. A formula of this Modal Logic of Actions is written following the structure $\psi [a] \phi$, where ψ and ϕ are well-formed formulae (wff) in conventional first-order logic, which characterize the state before or after the execution of the action a , respectively. As usual in modal logics, the construct \Box represents the necessity operator, and a represents an action. As a result, the meaning of formulae which are constructed following this pattern ($\psi [a] \phi$) is the following: “if ψ is satisfied before the execution of a , ϕ must be satisfied after the execution of a ”. To conclude, an example for an architectural element (and particularly of a connector) is provided, namely the *RobotConnector* in charge of synchronizing the Actuator and the Sensor of a robot (see Figure 47). This connector imports the *SMotion* safety aspect and the *CoordJoint* coordination aspect as mentioned above and is formed by the follow set of ports and weavings (see Figure 48).

The formalization of this connector is therefore given by the following composite π -process:

$$P_{AE} ::= P_{P1} \parallel \dots \parallel P_{Pm} \parallel P_{A1} \parallel \dots \parallel P_{An} \parallel P_W$$

5.3.2. Ports

Ports are the interaction points of architectural elements (components and connectors). Every port has associated a process, which establishes the services that publishes, and how and when they can be executed.

- Formalization: Ports

Let P be a port of an architectural element, such that its behaviour is specified by a process PR_P . Then its semantics are given by the process P_P , defined simply as follows:

$$P_P ::= PR_P$$

An example is the port P_{Act} in the *RobotConnector* example (see Figure 48), which has its behaviour specified as a process $P_{P_{Act}}$, which in turn refers to the generic definition of another process ACT .

$$P_{P_{Act}} ::= ACT$$

5.3.3. *Aspect*

An aspect defines the structure and the behaviour of a specific concern of the software system. Examples of concerns are functionality, coordination, safety, distribution, among others.

Structure is defined by a set of attributes, each of which has a value in every state. The state of the aspect at any given moment is determined by the value of its attributes. An aspect defines a semantics for its services. This semantics captures when the services cannot be executed, how the execution of services changes the state of the aspect, and the order in which they can be executed. The behaviour of an aspect is defined by means of a protocol. The protocol describes how the different services of the aspect are coordinated.

- *Formalization: Aspects*

An aspect is defined by the tuple (A, X, Φ, Π) :

- A : a set of attributes
- X : a set of services (see section 5.3.4)
- Φ : a set of formulae in modal logic of actions
- Π : a set of terms in π -calculus; this is, a set of concurrent terms describing partial processes in the π -calculus.

The contents of the set Π are therefore a set of π -calculus processes. For instance, let α be an aspect whose behaviour is specified by the PRT_1 protocol. Then its semantics is the process P_α defined as follows:

$$P_\alpha ::= PRT_1$$

Again, in the *RobotConnector* example of Figure 48, the *SMotion* aspect is similarly defined as:

$$P_{SMotion} ::= SMotionProtocol$$

The dialect that is used to describe terms in the Π set is a syntactic variant of the polyadic π -calculus. It also includes an extension to include priorities, which are not describe nor use here. But apart from this extension, the language is largely standard, even in the choice of derived

operators (such as **if . . . then**). The main syntactic differences are the use of the arrow (\rightarrow) as the prefix operator to define a sequence of actions, instead of the dot (\cdot), which is used here with its usual meaning at the programming level, to indicate scope nesting. Finally, the dialect provides also support for vector-like tuples of channels, which are simply indicated as \vec{v} . It is assumed an implicit indexing operator in this kind of vectors, so the name vI will refer to the first channel in the vector \vec{v} . This should be considered just as syntactic sugar.

5.3.4. *Weavings*

A weaving specification defines how the execution of a service of an aspect can trigger the execution of a service of another aspect. Of course, the same service can be involved in several weavings. In order to preserve the independence of the aspect specification from other aspects and weavings, weavings in PRISMA are specified outside aspects and inside architectural elements, including connectors. As a result, weavings specified inside connectors are the ones which coordinate the different aspects that a connector imports.

A weaving is defined by means of operators that describe the order in which services are executed. A weaving that relates service $s1$ of aspect $A1$ and service $s2$ of aspect $A2$ can be specified using the following operators. Note the use of the dot (\cdot) operator to indicate scope nesting, as indicated above.

- $A2.s2$ **after** $A1.s1$. $A2.s2$ is executed after $A1.s1$.
- $A2.s2$ **before** $A1.s1$. $A2.s2$ is executed before $A1.s1$
- $A2.s2$ **instead** $A1.s1$. $A2.s2$ is executed instead of $A1.s1$
- $A2.s2$ **afterif** (Boolean condition) $A1.s1$. $A2.s2$ is executed after $A1.s1$ if the condition is satisfied.
- $A2.s2$ **beforeif** (Boolean condition) $A1.s1$. If the condition is satisfied, $A2.s2$ is executed followed by $A1.s1$; otherwise, only $A2.s2$ is executed.
- $A2.s2$ **insteadif** (Boolean condition) $A1.s1$. $A2.s2$ is executed instead of $A1.s1$ if the condition is satisfied.

The invocation of $A1.s1$, the second argument of the weaving, triggers the execution of weaving (*pointcut*). When a weaving is specified, the operator is chosen from the trigger service point of view; depending on whether the trigger service needs the execution of a service before, after, or instead of it (*advice*). Therefore the before and after weaving modifiers are not directly interchangeable.

- **Formalization: Weavings**

The semantics of a weaving is a coordination process that intercepts the invocation of a service $A1.s1$ and either replaces it with, or executes it in relation to, another service $A2.s2$. $A1.s1$ and $A2.s2$ belong to different aspects.

The weaving must be executed each time that $A1.s1$ is invoked, upon which it executes either $A2.s2$ instead of $A1.s1$ or $A1.s1$ and $A2.s2$ in the correct order. This means that the invocation of a service does not automatically trigger the execution of its associated process. Taking into account that the formalization of a service in PRISMA is the following:

- **DEFINITION 1. (Service)** *A service is a process that executes a set of actions to produce a result.*

Let S be a service. The semantics of S is a process in the polyadic π -calculus called P_S . This process has a channel C_S through which it is able to interact; or, conversely, it can be invoked for execution (see Figure 49). It is possible to see immediately that services are not invoked directly by other processes, but only through weavings that coordinate execution of services within architectural elements.



Figure 49. Formalization of a Service

Let's start by defining a service invocation. This will make much easier to understand later the way in which it is defined the internal behaviour of a service.

- **DEFINITION 2. (Service Invocation)** Let $\vec{x} = x_1, \dots, x_n$ be the input parameters for a service S , and $y = y_1 : : y_m$ be its output parameters. The invocation of S is formalized by means of a message sent through channel C_S . Moreover, each output parameter y_i must have a return channel r_{y_i} , which is dynamically created for each invocation using the π -calculus restriction operator (ν) . These channels are used to send the results of S and to indicate and acknowledge termination of the execution of S . All this considered, a service invocation is described as the following process.

$$(\nu \vec{r}_y) (C_S !(\vec{x}, \vec{r}_y) \rightarrow r_{y1}?(y_1) \dots r_{ym}?(y_m))$$

The structure of this process defines the different ways in which a service is able to interact; so, it is now possible to define the behaviour of a service as a set of π -calculus processes, as indicated by the following definition.

- **DEFINITION 3. (Service Process)** The behaviour of a process P_S of a service S can be divided in three kinds of actions:

- *Request Reception.* The first action of P_S must be the reception of the messages that come through C_S . This reception is specified as follows.

$$C_S ? (\vec{x}, \vec{r}_y)$$

- *Service Execution.* The execution of the service internal behaviour consists of processing a set of internal actions. The output parameters $(\vec{y} = y_1, \dots, y_m)$ are created, and it is assumed that internal actions bind them with some useful value. Then this internal execution is specified as follows.

$$(\nu \vec{y}) (\tau)$$

- *Termination.* The last action in P_S is always the sending of the output parameters $(\vec{y} = y_1, \dots, y_m)$ through return channels $(\vec{r}_y = r_{y1} \dots r_{ym})$. This

way, the invoker is confirmed that execution of S has ended. This termination is therefore specified as follows.

$$r_{y1}!(y_1) \dots r_{ym}!(y_m)$$

As a result, the complete formalization of P_S is the replicated sequence of these three actions.

$$P_S ::= * (\overset{->}{C_S} ? (\overset{->}{x} , \overset{->}{r_y}) \rightarrow (v \overset{->}{y}) (\tau \rightarrow r_{y1}!(y_1) \dots r_{ym}!(y_m)))$$

This replication allows us to execute the service as many times as it is necessary. In terms of our formalization in the π -calculus, and given a service S which is being controlled by the weaving, this means that the weaving process P_W interacts with P_S via the channel C_S defined in DEFINITION 1. To do so, it must provide a channel C_{WS} which other processes can use to invoke S (see Figure 50).



Figure 50. Formalization of a service controlled by a weaving

Considering these two channels, the invocation of S by other processes is defined as the following π -term:

$$(v \overset{->}{r_y}) (\overset{->}{C_{WS}} ! (\overset{->}{x} , \overset{->}{r_y}) \rightarrow r_{y1}?(y_1) \dots r_{ym}?(y_m))$$

And then the invocation of S by the weaving process is therefore as follows:

$$(v \overset{->}{r_y}) (\overset{->}{C_S} ! (\overset{->}{x} , \overset{->}{r_y}) \rightarrow r_{y1}?(y_1) \dots r_{ym}?(y_m))$$

After that, each weaving operator defines a different process with a specific behaviour, to provide the required semantics for each one of them. As an example, let's consider the process for the beforeif weaving operator, which involves two services belonging to two different aspects.

$$P_{1..n} ::= (v \overset{->}{r_y}) (\overset{->}{C_{WA1_S1}} ! (\overset{->}{x} , \overset{->}{r_y}) \rightarrow r_{y1}?(y_1) \dots r_{ym}?(y_m))$$

$$\begin{aligned}
P_{BWIF} ::= & * (C_{WA1_s1} ? (x, r_y) \rightarrow (v \ r_{s2}) (C_{A2_s2} ! (x, r_{s2}) \rightarrow \\
& r_{s21}?(s2_1) \dots r_{s2m}?(s2_m)) \rightarrow \\
& \text{if (boolean_condition = true) then} \\
& \quad (v \ r_{s1}) (C_{A1_s1} ! (x, r_{s1}) \rightarrow r_{s11}?(s1_1) \dots r_{s1m}?(s1_m)) \rightarrow \\
& \quad r_{y1}!(s1_1) \dots r_{ym}!(s1_m)) \\
& \text{else} \\
& \quad r_{y1}!(s2_1) \dots r_{ym}!(s2_m)) \\
P_{A1_s1} ::= & * (C_{A1_s1} ? (x, r_{s1}) \rightarrow (v \ s1) ((\tau) \rightarrow r_{s11}!(s1_1) \dots r_{s1m}!(s1_m))) \\
P_{A2_s2} ::= & * (C_{A2_s2} ? (x, r_{s2}) \rightarrow (v \ s2) ((\tau) \rightarrow r_{s21}!(s2_1) \dots r_{s2m}!(s2_m)))
\end{aligned}$$

Table 3. Translation set of π -processes for beforeif weaving pattern

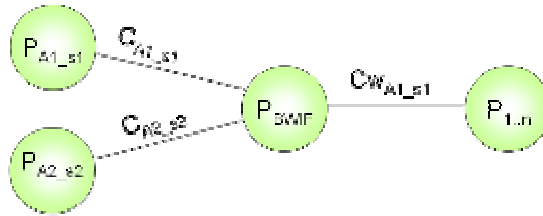


Figure 51. Translation for beforeif weaving patterns

A2:s2 **beforeif** (Boolean condition) A1:s1

According to PRISMA formal semantics, this weaving pattern will be translated to the λ -calculus as a compound process P_{BWIF} , which has the context depicted in Figure 51. This means that P_{BWIF} receives the invocation of A1.s1 from another process ($P_{1..n}$) through C_{WA1_S1} . As BWIF is a “before” weaving, P_{BWIF} starts by invoking A2.s2 using C_{A2_s2} . Then, P_{A2_s2} receives the invocation, executes a set of internal actions, sends the results, and notifies the

weaving that execution has finished. Next, if the Boolean condition in $BWIF$ is true, the first service of the weaving is executed; otherwise P_{BWIF} sends the results of A2.s2 to the process that invoked A1.s1. In the first case, when the condition is satisfied, P_{BWIF} invokes A1.s1 using C_{A1_s1} and P_{A1_s1} receives the invocation upon which it executes a set of internal actions, sends the results, and notifies the weaving that the execution has finished. Finally, P_{BWIF} sends the results of A1.s1 to the process that invoked A1.s1.

The semantics of the set of weavings defined inside a connector is therefore translated as the P_W process, the parallel composition of every individual weaving process.

$$\begin{aligned} P_W ::= & P_{AW1} \parallel \dots \parallel P_{AWn} \parallel P_{BW1} \parallel \dots \parallel P_{BWn} \parallel P_{IW1} \parallel \dots \parallel \\ & P_{IWn} \parallel P_{AWIF1} \parallel \dots \parallel P_{AWIFn} \parallel P_{BWIF1} \parallel \dots \parallel \\ & P_{BWIFn} \parallel P_{IWIF1} \parallel \dots \parallel P_{IWIFn} \end{aligned}$$

This means that the weavings are executed concurrently, interacting as specified. In addition, the same service can be involved in several weavings of the same architectural element and there is an order for processing the different weavings that a service triggers. This ordering establishes that weavings are executed from more restrictive to less restrictive. The precedence is as follows: *InsteadIf*, *Instead*, *BeforeIf*, *Before*, *After*, *AfterIf*. Deadlocks and infinite loops that could appear when using these operators are avoided at the specification time.

An example of a weaving appears in the *RobotConnector* case study. This connector imports the *SMotion* safety aspect and the *CoordJoint* coordination aspect. The need for a weaving emerges due to the fact that the robot is moved only after the connector is sure that a movement is safe. The invocation of the *moveJoint* service (the second argument of the weaving) of the *CoordJoint* triggers the execution of the weaving (see the process $P_{BWIFSMotionCoordJoint}$ in Figure 12). Specifically, the weaving of the connector receives the

invocation of the *moveJoint* service (the term $C_{WCoordJoint_moveJoint} \overset{\rightarrow}{?}(\text{newsteps}, \text{speed}, r_y)$ in the process) and afterwards it specifies that the DANGEROUSCHECK service of *SMotion* has to be executed, and it must answer before the *moveJoint* service of *CoordJoint* is even invoked,

hence the term $(\nu \overset{->}{r}_{s2}) (C_{SMotion_DANGER} ! (\text{newsteps}, r_{s21}) \rightarrow r_{s21}?(safe))$ in the process. Then the condition guarantees that the execution of the moveJoint service is only performed if the *safe* return parameter of the *DangerousCheck* service is set to true (hence the *if/then/else* construct in Figure 12, which encloses the invocation of the moveJoint service through the $C_{CoordJoint_moveJoint}$ channel). On the other hand, the processes defining the behaviour of each one of the services, which are in turn defined within the aspects, are ready to be invoked by the weaving at any time (see the definition for both $P_{CoordJoint_moveJoint}$ and $P_{SMotion_DANGER}$ as replicated, hence permanent, processes in the Figure).

$$\begin{aligned}
P_W &::= P_{BWFSMotionCoordJoint} \\
P_{BWFSMotionCoordJoint} &::= * (C_{WCoordJoint_movejoint} ? (\text{newsteps}, \text{speed}, \overset{->}{r}_y) \rightarrow \\
&\quad (\nu \overset{->}{r}_{s2}) (C_{SMotion_DANGER} ! (\text{newsteps}, r_{s21}) \rightarrow r_{s21}?(safe)) \rightarrow \\
&\quad \text{if (safe = true) then} \\
&\quad \quad (\nu \overset{->}{r}_{s1}) (C_{CoordJoint_movejoint} ! (\text{newsteps}, \text{speed}, \overset{->}{r}_{s1}) \rightarrow \\
&\quad \quad r_{s11}?(s11)) \rightarrow r_{y1}!(s11)) \\
&\quad \text{else} \\
&\quad \quad r_{y1}!(s21)) \\
P_{CoordJoint_movejoint} &::= * (C_{CoordJoint_movejoint} ? (\text{newsteps}, \text{speed}, \overset{->}{r}_{s1}) \rightarrow (\nu \overset{->}{s1}) ((\tau) \rightarrow \\
&\quad r_{s11}!(s11))) \\
P_{SMotion_DANGER} &::= * (C_{SMotion_DANGER} ? (\text{newsteps}, r_{s21}) \rightarrow ((\tau) \rightarrow r_{s21}!(safe)))
\end{aligned}$$

Figure 52. Translation for the weaving in the *RobotConnector* example

5.4. ANALYSIS OF THE PROPOSAL

Both coordination and architecture are generic high-level abstractions of a software system; they provide different approaches to close concerns, and both have long and separate research traditions. At the same time, there is an obvious relationship between them. Both notions try to identify highlevel patterns in the system, though their perspectives are slightly different. Architecture identifies structural patterns defined by inner interaction within a (mostly)

compositional configuration, while coordination defines high-level interaction patterns shown by the resulting structure.

However when the relationship between them is considered, even their relative ordering has not always been clear. Different authors have considered their relationship in different ways, and this is the best proof of their intertwining and the intrinsic difficulty of their separation. For instance, Andrade et al. [And02a], [And02b] consider that configurations are built on top of a coordination layer which guarantees a shared behaviour. On the contrary, Eisenbach and Radestock [Eis98] conceive coordination as the higher level abstraction, which is built on top of a configuration layer, which guarantees a substrate for shared interaction. At the same time, many authors present these two abstractions at the same level and provide a common support for it; in particular, many coordination languages have also been presented as ADLs, provided that their particular abstractions are equally good for describing both [Mag95], [Pap01]. In particular, connectors and special-purpose components bear many similarities to some constructs in several control-driven coordination proposals.

Probably among the most important reasons for the success of the architectural approach is the implicit separation of concerns it provides; the designer is just concerned with the functionality of components (and possibly some relevant non-functional requirements), but he is now relieved of describing compositional and coordination issues, which have become the architect's responsibility. Though connectors are not the only way in which an ADL can describe interaction and coordination abstractions, their existence and the emphasis on them is probably the reason why these languages are so apt in specifying these issues. And once they have been separated, the relevant high-level linguistic constructs in different approaches are similar.

It is possible to conclude that coordination is an emergent property of some architectures; an architecture-level description has the means for providing the coordination concern, but of course it can also describe non-coordinated systems. In summary, architectures describe interaction structures; and coordination can be described as a higher-level abstraction on interaction, therefore supported by architecture [Cue06].

Connectors alone do not provide a global coordination policy, but only local coordination groups; therefore the use of connectors (as discussed in section 5.2 and above) eases the description of a coordinated system, but it is not a sufficient condition. Shaw's original identification of connectors [Sha94] tried not to provide a coordination, but an interaction abstraction. However, subsequent work has defined ever more complex connectors, which were grouped in types and categories, tending towards the definition of much more complex abstractions, even higher-order connectors [Lop03]. Mehta provided an initial taxonomy for connectors [Meh00], which could have provided a basis for later developments in this direction, but this thread has not had continuity.

The locality of the connector approach justifies still the definition of generic coordination language proposals, which provide the means to describe general policies. However an aspect-oriented alternative is also possible. Instead of providing a complete language from scratch, it is also possible to define an aspect-oriented extension of some existing language. More than that, this would ease the integration of this "coordination aspect" with other concerns in the architecture. Consider also that earlier proposals for aspect-orientation [Kiz97] defined specific-purpose languages to deal with aspects, rather than aspectual extensions, so this evolution towards an architectural extension is also within the tradition in the field.

Therefore it is possible to provide coordination by means of pure compositional ADLs, but connectors make it easier. Then, a specific coordination language provides general policies, but an aspect-oriented extension makes integration easier. Thus, providing an aspect-oriented, connector-based ADL would gather the benefits of different proposals. The reader is again referred to section 5.2 for a detailed discussion of the different approaches for providing coordination in ADLs, including connector-based and aspect-oriented alternatives.

The reflective ADL PiLar has explored the way in which a very general architecture language is able to describe coordination as a separate concern, i.e. as an architectural aspect. In [Cue04] this was made by exploiting the reflective capabilities, thus proving that this is indeed possible, but very complex. Later research has explored also an aspect-oriented approach which makes a non-explicit use of these reflective capabilities in PiLar, providing an aspectual layer and showing how coordination can be independently managed as an aspect [Cue06]; but

the relationship between this aspect and others, though possible, was complex, and is not explored in detail.

And this is the main benefit of the PRISMA approach, as highlighted in previous discussion. The aspect-oriented structure of the language itself, and the symmetry of its model, provide the basis to be able to relate coordination to other concerns, such as safety. The notion of weaving, which is required by the aspect model, provides also the means to reconcile the conflicts between aspects, whenever they appear. This, combined to the benefits of both connectors and aspects themselves for coordination, defines PRISMA as one of the most complete proposals in the field, gathering all the benefits provided by other approaches in a single, consistent and rigorous conceptual model. Thus, PRISMA coordination model proposal provides a suitable framework to develop aspect-oriented software architectures following the MDD proposal.

5.5.CONCLUSIONS

In this chapter, the advantages of combining software architectures and AOSD to define coordination have been presented. In addition, a detailed analysis about how to take more advantage of this combination has been done. From this analysis, this chapter defines and formalizes PRISMA aspect-oriented connectors. They are specified in an elegant and novel way through the combination of AOSD and Software Architectures. As a result, PRISMA presents a coordination process that provides the following advantages:

1. **Connectors to coordinate components:** Reusability and maintenance of components and connector is improved by separating coordination from computation. Components and connectors can be used during the MDD process as building blocks of the modelling process and they can be reused throughout all their stages.
2. **Aspects to specify the coordination process of connectors:** Reusability and maintenance of different concerns is improved by separating coordination from other concerns that are necessary for the coordination process (safety, security, distribution, mobility, etc.). There are no tangled concerns inside complex connectors. Aspects can be

used during the MDD process as building blocks of the modelling process and they can be reused throughout all their stages.

3. **Weavings are inside connectors to coordinate their aspects:** Reusability and maintenance of different aspect is improved by not specifying weavings inside aspects.

4. **Formalization of the coordination processes among aspects (weavings) and architectural elements.** Thus, non-ambiguity and proper execution of the different coordination processes is guaranteed. The code generation of coordination models is improved during the MDD process.

The work presented in this chapter has been published in the following publication:

- **Jennifer Pérez,** Carlos E. Cuesta, *Aspect-Oriented Connectors for Coordination*, International Workshop on Synthesis and Analysis of Component Connectors (SYANCO 2007), Joint to The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC-FSE, **ACM Digital Library**, Dubroknik, Croacia, September 3-4, 2007.

CHAPTER 6

MODEL-DRIVEN DEVELOPMENT

This chapter presents how PRISMA gives a complete support for the development of technology-independent aspect-oriented software architectures following the MDD approach, and the PRISMA CASE, which is the tool that makes this software development support feasible. PRISMA has been applied to several case studies: banking systems, electronic auctions, robotic tele-operated systems such as the *TeachMover* robot, etc. The MDD support of PRISMA CASE is going to be illustrated using these case studies.

6.1. INTRODUCTION

Some new approaches have recently emerged in order to improve software development. They try to improve the early stages of the software life cycle by automating their activities as much as possible by following Model-Driven Development (MDD) [Bey05], [Am04]. MDD is a software development paradigm that is based on models that use automatic generation techniques in order to obtain the software product. MDD is included within Model-Driven Engineering (MDE) [Sch06], which increases the variety of software artefacts that can be represented as models (ontologies, UML models, relational schemas, XML schemas, etc). The use of models to develop software provides solutions that are independent of technology, whose source code can be obtained by means of automatic code generation techniques for different technologies and programming languages. The high level of abstraction that models

provide permits working with metamodels in the same way as with specific models or domain-specific models.

Aspect-Oriented Models propose the separation of the crosscutting concerns of software systems into separate entities called *aspects* [Kiz97]. Despite the fact that the aspect-oriented paradigm emerged from the implementation level, its use is being extended to all stages of the software life cycle. As a result, Aspect-Oriented Software Development (AOSD) has emerged in order to extend the advantages that aspects provide to every stage of the software life cycle [Kiz01]. One interesting stage where AOSD is being introduced is the software architecture stage.

Software architectures [Per92] make software systems simpler and more understandable. Some proposals for the integration of software architecture and AOSD have emerged to take advantage of both approaches [Chi05], [Cue05], [Nav05], [Pin05], [Pin03], [Kat03], etc.

The automatic code generation from models reduces the cost and time of the development process. Nowadays, there are many CASE tools that are able to generate applications following the Automatic Programming Paradigm proposed by Balzer [Bal85]. These tools are widely known as model compilers. They automatically generate the application code from the conceptual schema of a software system. The automatic generation can be complete as in Oblog Case [Ser94], or it can be partial, as in Rational Rose [RAT07], Together [TOG07] and others. However, since these model compilers follow the Object-Oriented Paradigm, the need for developing model compilers that follow the Software Architectures and/or AOSD approaches has emerged. The combination of the Software Architectures and AOSD reusability and the automatic code generation achieves higher reduction in the time and cost of the development process than using only one of these approaches. As a result, an important challenge in the software engineering area is the integration of software architectures and AOSD approaches, and automatic code generation and traceability techniques in a unique approach in order to support the development and maintenance of complex software systems in an efficient way.

PRISMA is an approach that integrates software architecture and AOSD in order to take advantage of both. The PRISMA approach is based on its meta-model [Per05a] and its formal

Aspect-Oriented Architecture Description Language (AOADL) [Per06d]. Since the PRISMA model is a technology-independent model, the PRISMA approach also follows the MDD paradigm 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 for different technological platforms and languages using automatic code generation techniques. A PRISMA CASE has been developed in order to cope with the challenge of developing aspect-oriented software architectures following the MDD paradigm.

The PRISMA Aspect-Oriented Architecture Description Language is a formal language that is based on a Modal Logic of Actions [Sti92] and a dialect of polyadic π -calculus [Mil93][Mil99]. It is important to emphasize that most ADLs only permit the specification of the skeleton of architectures and the services that are interchanged among their different architectural elements. The PRISMA AOADL has greater expressive power and can specify more features and requirements using aspects. This complete specification of the system requirements, and the fact that the PRISMA AOADL is a formal technology independent language, facilitates the automatic code generation and the validation of architectural and aspect features of the system.

6.2. THE MDD SUPPORT OF PRISMA

The PRISMA approach follows the MDD paradigm. There are two main approaches that apply this paradigm. They are the Model-Driven Architecture (MDA) approach proposed by the OMG [MDA07], and the Software Factories approach proposed by Microsoft [Gre04]. 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. In this way, this provides us the opportunity of extending the MDD support of PRISMA in future works. PRISMA follows the MDD approach by providing the software architect 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 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.

6.2.1. *PRISMA in MOF*

In order to present PRISMA model specifications and how they follow the MDD approach, the OMG Meta-Object Facility (MOF) specification is going to be used [MOF02]. MOF allows us to clearly present in this thesis 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 is the most abstract one (see the M3 layer, Figure 53). This layer defines the abstract language used to describe the next lower layer, which contains metamodels (the M2 layer). The MOF specification proposes the MOF Model as the abstract language for defining all kinds of metamodels, such as UML or PRISMA.

The metamodel layer defines the structure and semantics of the models defined at the next lower layer (the M1 layer). The PRISMA metamodel is defined at the M2 level. It defines the properties that the interface, aspect, architectural element, and connection primitives have (see the system package of the PRISMA metamodel in the M2 layer, Figure 53).

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

VirtualBank type that has been defined using the PRISMA system primitive (see M1 layer, Figure 53). 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 myVirtualBank, M0 layer, Figure 53), which behave as described in the system type.

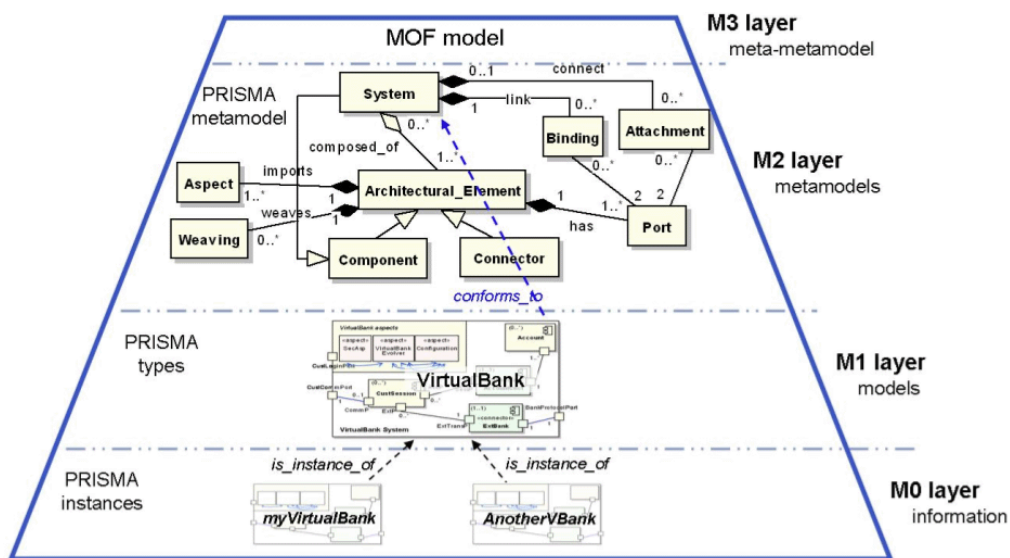


Figure 53. Meta-Object Facility (MOF) layers and PRISMA models

6.2.2. PRISMA transformations

The PRISMA AOADL [Per06d] defines the architectural elements at different levels of abstraction: the type definition level and the configuration level. The type definition level defines architectural types with a high abstraction level in order to be reused by other types or specific architectures. The configuration level designs the architecture of software systems by creating and interconnecting instances of the defined architectural elements in the type definition level. In other words, it specifies the topology of a specific architectural model. These

two levels of abstraction also appear in PRISMA models: PRISMA type models and PRISMA configuration models.

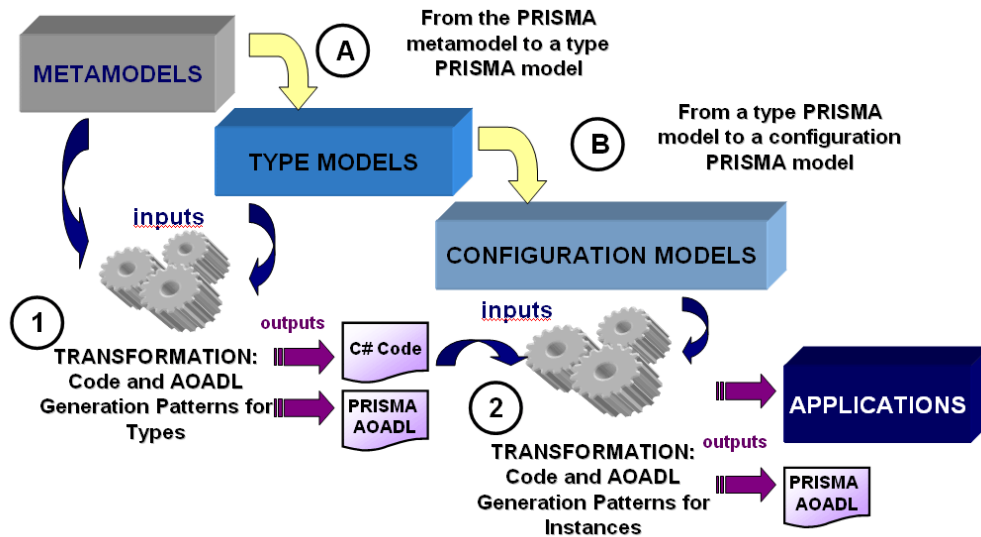


Figure 54. MDD from the PRISMA Metamodel to Applications

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 54), and to define configuration models from type models (see step B, Figure 54). In addition, PRISMA approach has created a set of transformation patterns to transform PRISMA models into its AOADL specifications and into C# code (see steps 1 and 2, Figure 54). 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 PRISMA CASE. This CASE tool supports the PRISMA approach and it is presented in detail in the following section.

6.3. FOLLOWING MDD WITH PRISMA CASE

PRISMA CASE currently supports the generation 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 55).

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 AOADL is a formal language [Per06d]. 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 [Per06a], [Per06b] and a graphical modelling tool to model PRISMA software architectures using an intuitive and friendly graphical AOADL. 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 [Per05b]. PRISMANET extends the .NET technology through the execution of aspects on the .NET platform in accordance with the PRISMA model.

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.

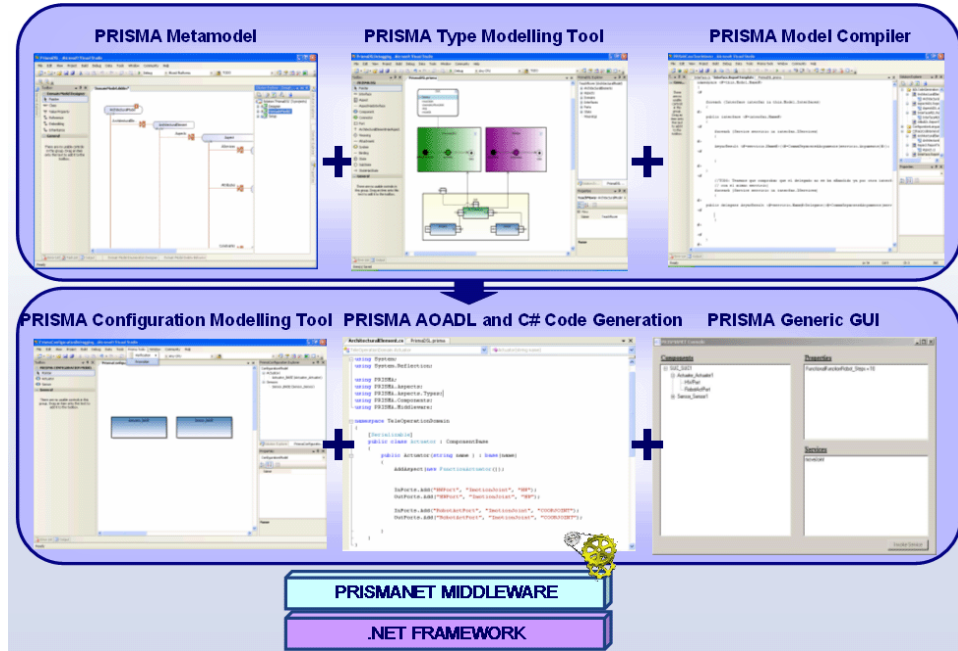


Figure 55. PRISMA CASE

6.3.1. PRISMA CASE development: Domain Specific Language Tools (DSL Tools)

Since there are a lot of tools in the market that provides mechanisms to follow the MDD, PRISMA CASE has been developed using one of them instead of developing a tool from scratch. This decision is taken in order to reduce the time and cost invested in the implementation of this MDD support. Following this same criteria, since the PRISMANET middleware was previously developed using the Visual Studio framework, the Domain Specific Languages Tools (DSL Tools) [DSL07] was chosen among the different tools of the market to develop PRISMA CASE.

DSL Tools is a set of tools for creating, editing, visualizing, and using domain-specific models to automate and improve the software development process. This set of tools is integrated into the Visual Studio 2005 framework to define domain models with their customized graphical representations.

DSL tools have been created to model specific models such as the model of a web page, a banking system, a tele-operated system, etc. DSL Tools allows for the definition of domain specific models and their customized graphical representations. From these two projects, DSL tools is able to generate domain-specific tools for these specific models. These specific and customized tools are then used to define specific applications of web pages, banks systems, tele-operated systems with domain-specific tool boxes and concepts.

The generated tool not only provides a customized modelling tool, it also provides *Code Generation Templates*, which automatically generate code using a set of code generators. These templates help users to define a model compiler in an easy way by browsing through the concepts that have been modelled and stored in the domain specific model. The DSL code generators take the templates, the domain specific model definition and its XML document as inputs of the code generation process. The output of this process is generated by the code generators following the defined templates and substituting the parameters for the concepts stored in the model.

6.3.2. *The PRISMA Type Modelling Tool: A) From the PRISMA Metamodel to the PRISMA Type Models*

In order to develop PRISMA CASE, every metaclass and relationship of the PRISMA metamodel have been introduced in the DSL Tools using the primitives that DSL provides (see Figure 56). For example, Figure 57 shows the definition of the architectural element and aspect metaclasses in DSL Tools.

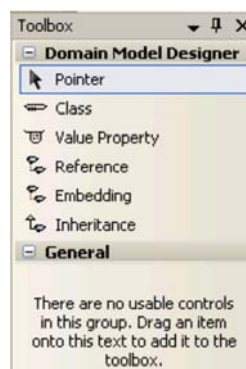


Figure 56. Toolbox of DSL Tools

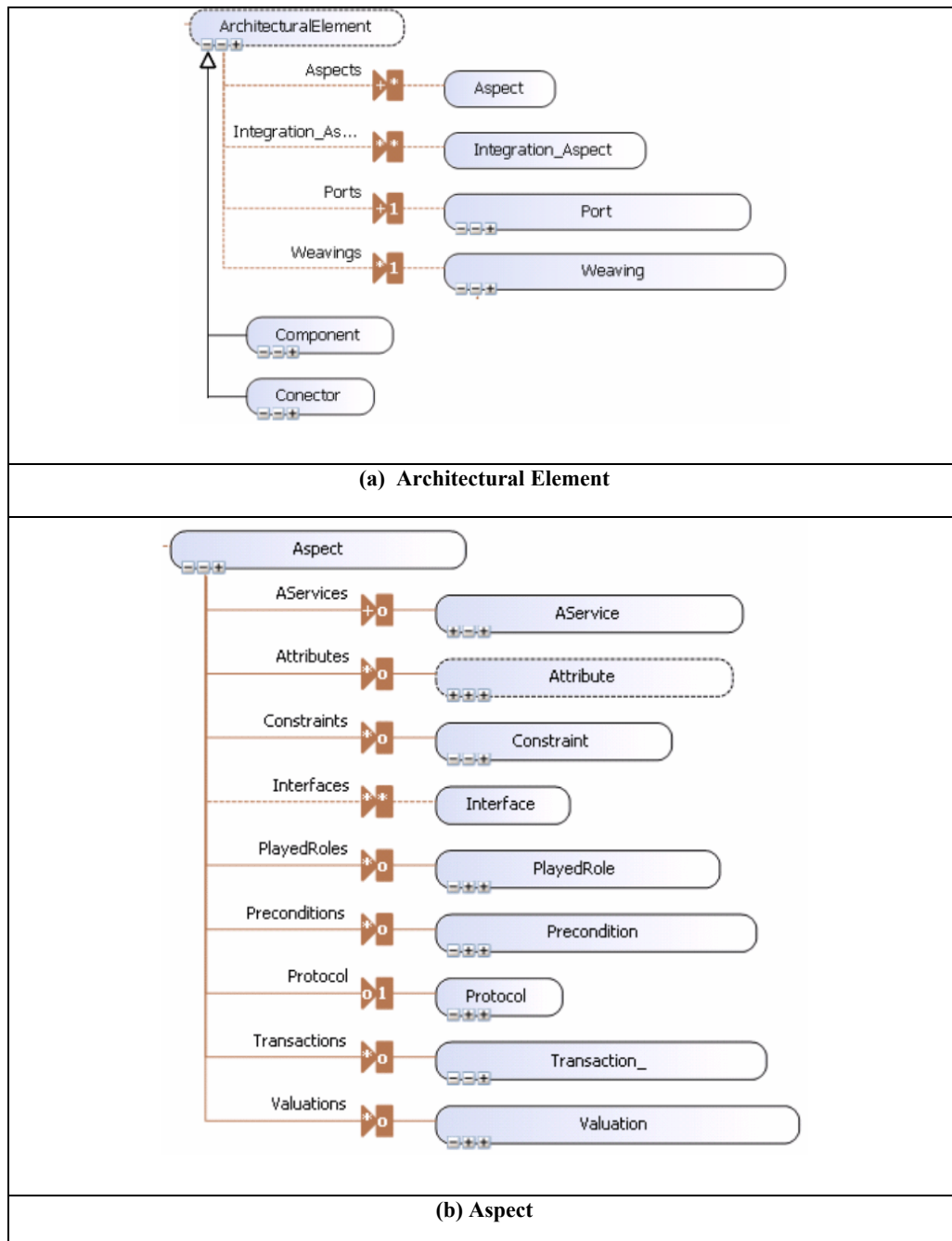


Figure 57. Definition of Architectural Elements and Aspects in the DomainModel of DSL

All the classes of the PRISMA metamodel are translated by DSL to partial C# classes in order to access and update value properties, to navigate across relationships, and to enable an object to participate in a relationship. In addition, they can be used to add new behaviour to the model, such as to include verification rules (see section 7.3).

In addition, the graphical metaphor to each metaclass of the PRISMA metamodel has been defined. DSL tools stores the graphical representations selected for the PRISMA concepts. As a result of this definition, the PRISMA graphical modelling tool provides a tool box that permits the graphical modelling of PRISMA models by dragging and dropping the shapes to the drawing sheet (see Figure 6). In PRISMA, only the main concepts and their relationships are graphically represented; the rest of the concepts are specified using the AOADL and are included in the definition of the corresponding shapes.

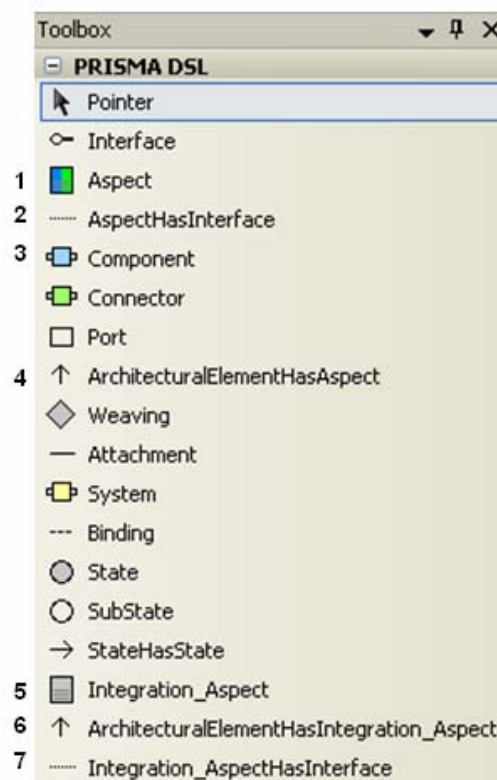


Figure 58. PRISMA ToolBox

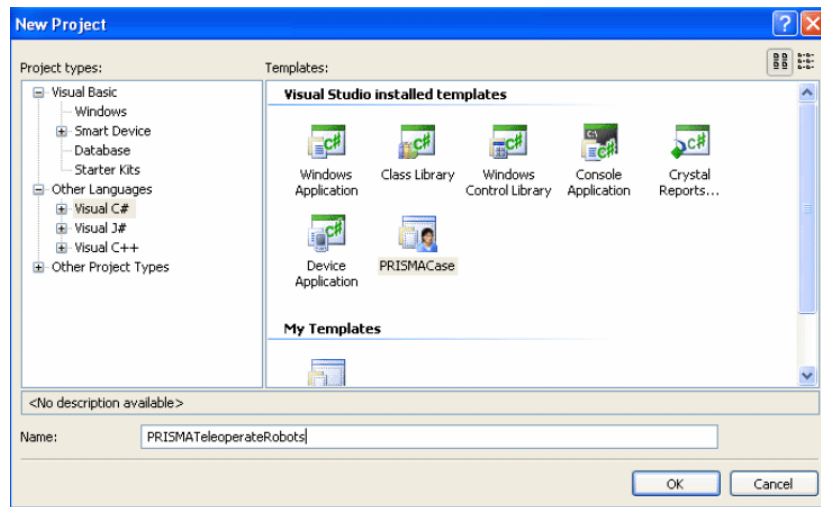


Figure 59. The Visual Studio Project of PRISMA

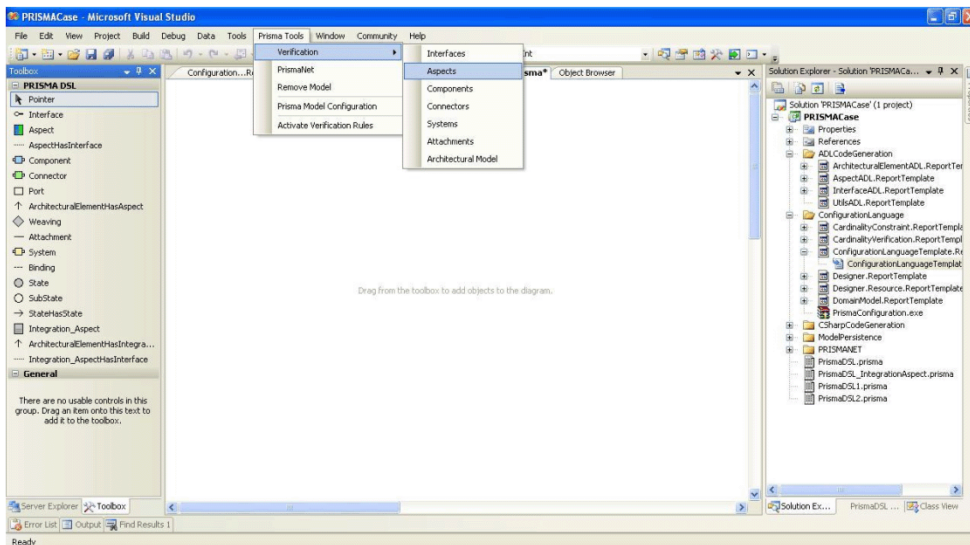


Figure 60. PRISMA Type Modelling Tool

Finally, it is important to mention that a setup for the PRISMA CASE has been defined in DSL Tools. As a result, PRISMA CASE can be provided to the software architect as an independent project of the PRISMA CASE implementation. When the software architect executes the setup, it creates a new kind of project for Visual Studio 2005 called PRISMA Case

(see Figure 59). The creation of a PRISMA CASE project consists of launching PRISMA CASE and starting the development process.

The *PRISMA Type Modelling Tool* is generated from the PRISMA metamodel, its graphical representations and its partial C# classes (see step A, Figure 54). The modelling tool is composed of a toolbox, a drawing sheet, a *model explorer*, a *window of properties* and a PRISMA menu (see Figure 60).

6.3.3. The PRISMA Model Compiler for Types: 1)Transformation: Code and AOADL generation patterns for types

The *PRISMA Type Modelling Tool* also provides a set of templates to automatically generate the code from the models that have been graphically modelled. The templates for generating the AOADL specification and the C# code are already available (see step 1 in Figure 54). They can be extended to generate the code for other languages. The templates and the command to execute the generators that produce the result are provided by the window *Solution Explorer* of PRISMA CASE. This command is called *Transform All Templates*. Its execution calls the code generators that execute the code generation templates of PRISMA by substituting the parameters for the elements that have been modelled. The PRISMA templates are stored in two different folders: *ADLCodeGeneration* and *CSharpCodeGeneration*. These folders contain the templates to generate each PRISMA type and the file that contains the result of the last *Transform All Templates* execution (see Figure 61).

The *ADLCodeGeneration* folder contains the formal specification of the software architecture that has been modelled following the PRISMA AOADL [Per06d]. Despite the fact that the specifications are introduced in the graphical shapes using the PRISMA AOADL, the AOADL generation permits the user to see the complete textual specification of the model.

The *CSharpCodeGeneration* folder contains the C# code generation, which allows the execution of the specified software architecture on the PRISMANET. The implementation of a specific PRISMA software architecture is performed by extending the classes provided by PRISMANET. In order to develop these code generation templates, a set of patterns has been identified and defined to generate the C# code for each one the PRISMA concepts to be executed over PRISMANET (see appendix A). Next, a simplified example of a component

pattern is presented by using the *Actuator* component of the *TeachMover* Robot case study(Pattern 15 of the catalogue).

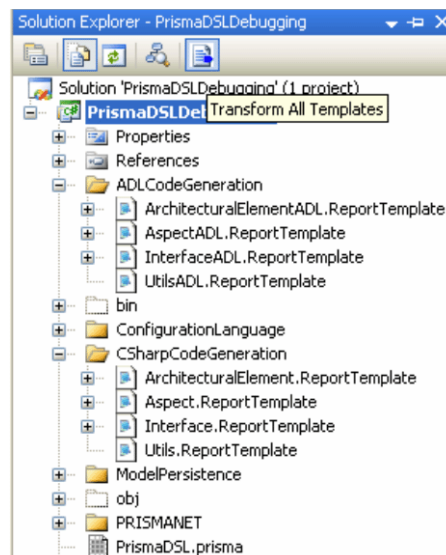
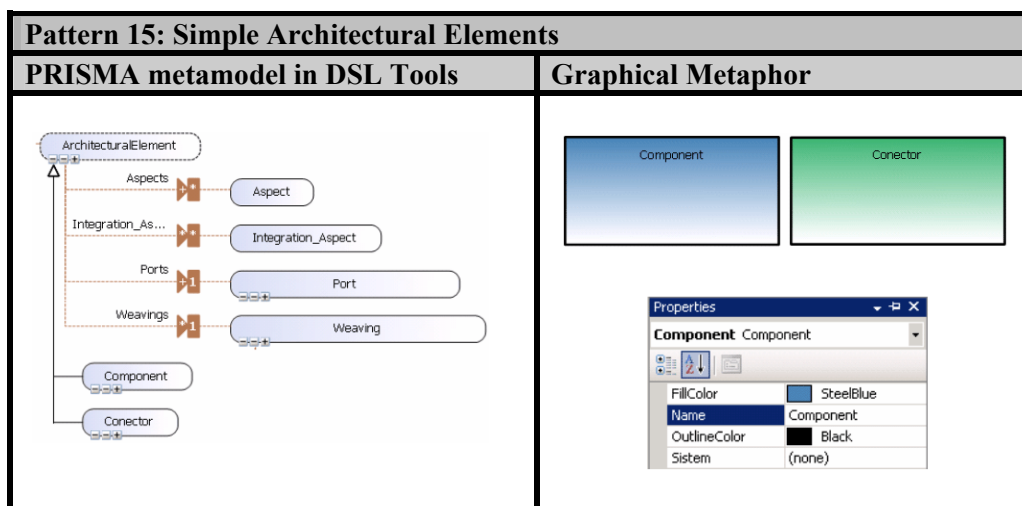


Figure 61. PRISMA Code Generation Templates



<div> <div>ArchitecturalElement (Class)</div> <ul style="list-style-type: none"> ArchitecturalModel (Reference property) Aspects (Reference property) Ports (Reference property) Weavings (Reference property) Sistem (Reference property) Integration_Aspects (Reference property) </div>	
Transformation	
Description	
<p>This pattern details how to generate the C# code from a simple architectural element. Specifically, it only generates the structure of the architectural model, the internal code of this structure, that is, ports, aspects and weaving, is generated by other patterns related to it.</p>	
Template	
<pre> ... using System; using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types; using PRISMA.Components; using PRISMA.Middleware; namespace <#=this.Model.Name#> { <# foreach (ArchitecturalElement architecturalElement in this.Model.ArchitecturalElements) { if (architecturalElement is Component architecturalElement is Connector) { #> [Serializable] public class <#=architecturalElement.Name#> : ComponentBase <# if (architecturalElement is Connector) { #> , IConnector <# } #> { public <#=architecturalElement.Name#> (string name<#=ArchitecturalElementArguments(architecturalElement)#>) : base(name) { <# /* Aspects */ /* Weavings */ /* Ports */ #> } } } } } </pre>	

```

}
<#
}/* endif (architecturalElement is Component || architecturalElement is
Connector)*/
...

```

Case Study

Description

This pattern is illustrated using the component *Actuator* of the TeachMover case study. The representation of the *Actuator* in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.

Graphical representation

The diagram shows a UML component diagram for the *Actuator* component. The component is represented by a blue rectangle labeled *Actuator*. It has two provided interfaces: *PSensor* and *PCoord*. Below the diagram is a screenshot of the PRISMA tool's Properties window. The window shows the *Actuator* component selected. The Properties window has a table with the following data:

Name	Actuator
Sistem	Joint

To the right of the Properties window, there is a tree view showing the *ArchitecturalElements* and *Actuator (Component)*.

Result of the pattern execution

```

...
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
    [Serializable]
    public class Actuator : ComponentBase
    {
        public Actuator(string name) : base(name)
        {
            /* Aspects */
            /* Weavings */
            /* Ports */
        }
    }
}
...

```

Related Patterns

Pattern 16, Pattern 17 and Pattern 18.

```

Component Actuator
  Integration Aspect Import RS232;

  Ports
    PCoord : IMotionJoint,
      Played_Role RS232. INTMOVE;
    PSENSOT : IMotionJoint,
      Played_Role RS232. OUTMOVE;

  End_Ports;
  new() {RS232.begin();}
  destroy() {RS232.end();}
End_Component Actuator;

```

```

using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
    [Serializable]
    public class Actuator : ComponentBase
    {
        public Actuator(string name) : base(name)
        {
            /* Aspects */
            AddAspect(new RS232 ());
            /* Weavings */
            /* Ports */
            InPorts.Add ("Pcoord", "IMotionJoint", INTMOVE);
            OutPorts.Add ("Pcoord", "IMotionJoint", INTMOVE);
            InPorts.Add ("PSENSOR", "IMotionJoint", OUTMOVE);
            OutPorts.Add ("PSENSOR", "IMotionJoint", OUTMOVE);
        }
    }
}

```

Figure 62. The generated AOADL and C# code of the component Actuator

All the patterns are not presented due to space limitations. But, the results of the complete C# and AOADL transformations of the component pattern for the *Actuator* are presented in Figure 13. As it can be seen in the C# code presented in Figure 13, a component is implemented as a serializable C# class. This class is serializable in order to enable mobility in future versions of PRISMA CASE [Ali06]. This class inherits from the *ComponentBase* class of PRISMANET, which implements the component of the PRISMA model. The component name is the same as the one in the PRISMA specification. The set of ports and aspects that make up a component are included by invoking the constructors of the *port* and *aspect* PRISMANET classes. Both classes implement the port and aspect elements of the PRISMA model.

6.3.4. The PRISMA Configuration Modelling Tool: B) From PRISMA Type Models to PRISMA Configuration Models

The PRISMA metamodel that has been introduced in DSL, and the modelling tool that has been developed from this model provide us mechanisms to specify PRISMA aspect-oriented software architectures. However, it is necessary to instantiate and to configure these architectures into specific ones and provide the software architecture mechanisms to do so. In order to cope with these needs, PRISMA CASE automatically generates a domain specific graphical modelling tool to configure the software architectures that have been defined using *PRISMA Type modelling tool*. A *PRISMA Configuration Modelling Tool* is generated for each PRISMA software architecture that is modelled using the *PRISMA Type Modelling Tool*. The *PRISMA Configuration Modelling Tool* is used to develop specific software architectures using the PRISMA types defined in the *PRISMA Type Modelling Tool* as modelling primitives.

The capacity to store all the information needed to automatically generate a domain-specific tool permits the use of PRISMA architecture as a domain specific language in other model specifications. It also permits the generation of the specification of the configuration language. This information is generated using the code generators of DSL and is stored in the *persistence* and *configuration language* folders of PRISMA type modelling tool (see Figure 63). The

information of these folders is the input for creating the new modelling tool for the domain-specific PRISMA software architecture that has been defined.

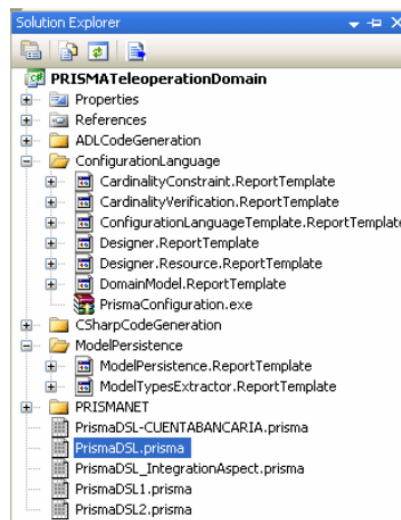


Figure 63. Model Persistence and Configuration Language Information

The automatic generation of a tool for modelling configurations of a PRISMA software architectures is performed by executing the *PRISMA Model Configuration* option of the menu PRISMA after the transformation of all templates has been done (see step 1, Figure 64). Next, a new project is automatically created and can be used as a *Configuration Modelling Tool*. Step 2 of Figure 64 shows how the *Actuator* and *Sensor* component types of the *TeachMover* robot defined in step 1, Figure 64 appear in the tool box of the Configuration Modelling Tool as shapes for modelling. It shows how these types have been dragged and dropped on the drawing sheet generating two instances. In Figure 64 example, the *base* joint of the *TeachMover* robot is modelled by defining its *actuator* and *sensor* component instances: *Actuator_BASE* and *Sensor_BASE*.

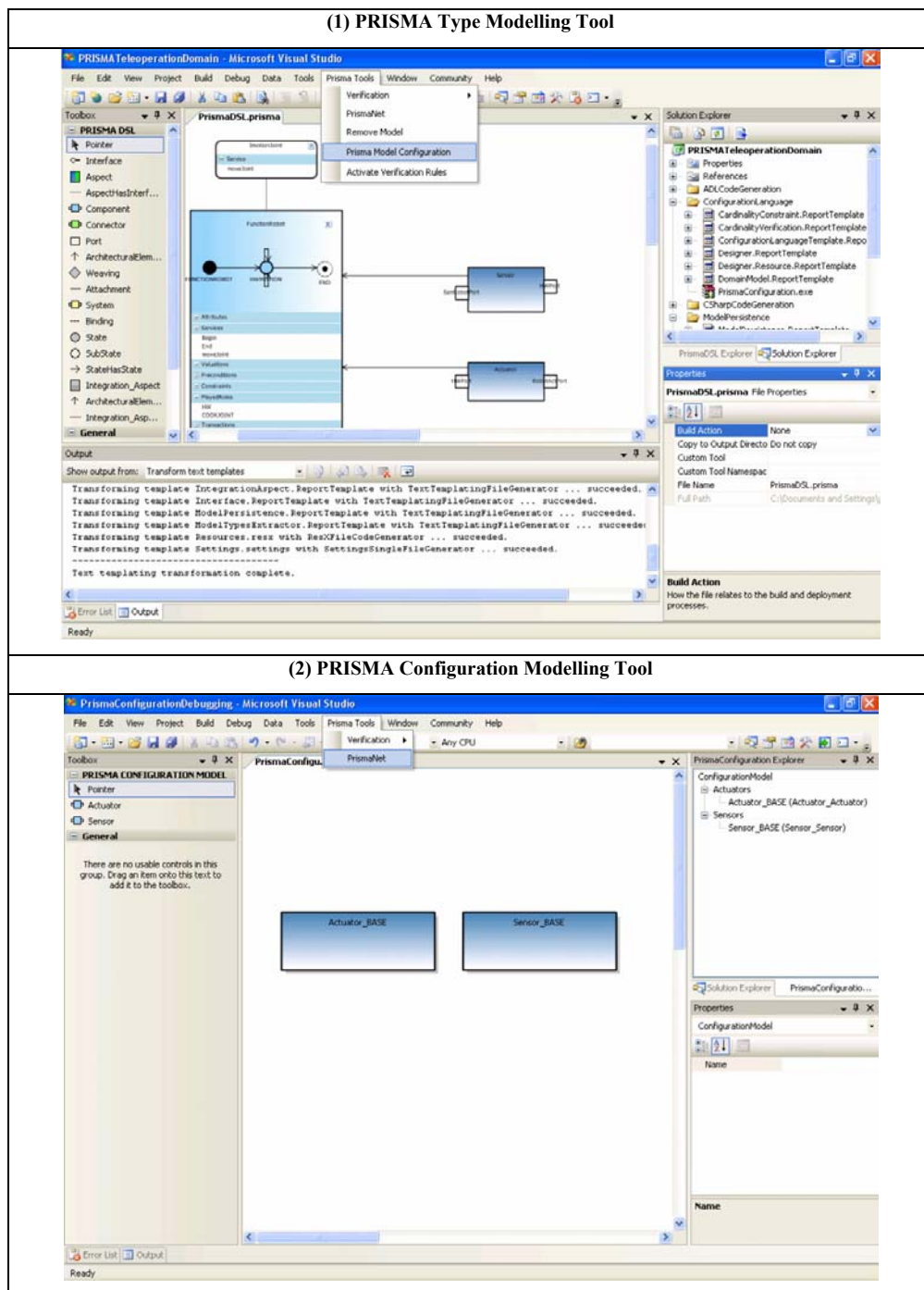


Figure 64. Generation and Execution of the PRISMA Modelling Configuration Tool

6.3.5. *PRISMA Model Compiler Instances: 2) Transformation: Code and AOADL generation patterns for instances*

In addition, the configuration modelling tool provides a command to transform its templates to obtain the AOADL specification that corresponds to the configuration that has been defined using the tool, and to instantiate the code, that has been generated in step 1, with the instances that are in the configuration model. In order to allow this transformation process, the information of the configuration model is stored in a XML document. Figure 65 shows the structure of this XML store simple components:

Finally, this tool permits the execution of the generated code by launching the defined instances. In order to do this, the PRISMA menu offers the option PRISMANET, which executes the middleware PRISMANET and instantiates the defined configuration (see the menu *PRISMA Tools*, step 2, Figure 64). As a result of this execution, a generic GUI is launched to interact with the architecture by invoking its services and checking the value of its attributes (see Figure 66). The main purpose of the generic GUI is to assist the user in checking the behaviour of the architecture without having to worry about aesthetic details and without forcing the user to define a GUI in order to obtain a result.

```
<?xml version="1.0" encoding="utf-8"?>
<ConfigurationModel name = "_operation">
  .....
  <Components>
    <Component name = "" type = "">
      <Properties>
        <Property name = ""
          type = "" value = "">
        </Property>
      </Properties>
      <SystemRef name = "">
        </SystemRef>
    </Component>
  </Components>
  ... ..
</ConfigurationModel>
```

Figure 65. XML document for storing instances

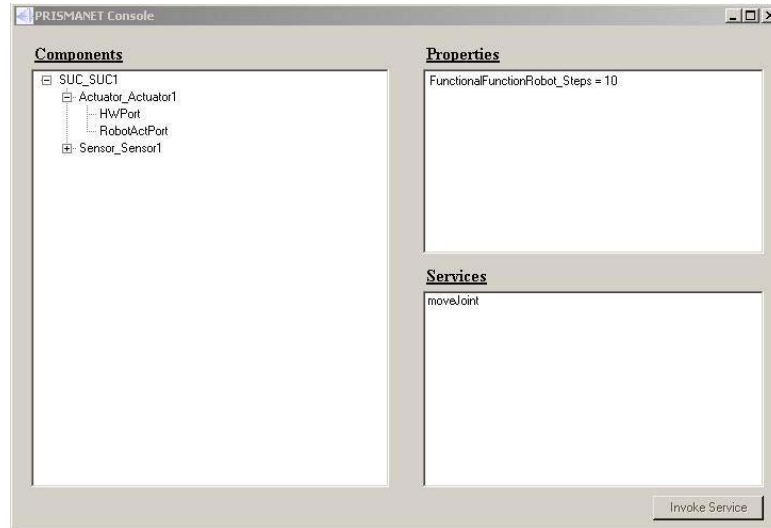


Figure 66. Generic GUI of PRISMA Applications

However, it is important to mention that the use of this interface is not mandatory. In other words, if the users prefer to define their own specialized forms, they can do so.

6.4.CONCLUSIONS

In this chapter, the PRISMA approach is presented as an important advance in the combination of the aspect-oriented paradigm and software architectures due to the fact that it completely supports the development process of these systems by applying the MDD paradigm. In addition, the PRISMA CASE that supports this approach has been presented in this chapter.

PRISMA CASE is a framework that provides complete support for the PRISMA approach. It is composed of a set of tools that is suitably integrated to provide a unique framework that gives support for the user throughout the software life cycle. This integration also provides top-down traceability during the different stages of the software life cycle and facilitates the maintenance of the developed software products.

This set of tools includes the *PRISMA Type Modelling Tool* with its code generation patterns, the *PRISMA Configuration Modelling Tool* with its code generation patterns, the generic Graphical User Interface for PRISMA applications, and the middleware PRISMANET.

The PRISMA *Types and Configuration Modelling Tools* give support for the development of PRISMA software architectures following the MDD approach and using the PRISMA AOADL [Per06d] in a graphical way [Per06a], [Per06b]. As a result, PRISMA offers mechanisms to develop software architectures in a more intuitive and friendly way and mechanisms to verify their models. In addition, the code generation patterns that PRISMA modelling tools offer allow automatically generate executable C# code on PRISMANET from the specified graphical models. Thus, PRISMA CASE deals with the traceability between software architectures and implementation and reduces the time and cost invested in the development and maintenance processes.

PRISMA CASE provides a generic Graphical User Interface to execute software architectures. This is an important advantage because it is a simple way of validating that software architectures provide the behaviour expected by the user without having to develop a customized graphical user interface.

This chapter demonstrates that all the tools and mechanisms that PRISMA CASE provides make PRISMA a well-supported approach for developing aspect-oriented software architectures following the MDD approach. The demonstrations of the PRISMA CASE and its download are available in the [PRI07].

The work presented in this chapter has been submitted to the following publication:

- **Jennifer Pérez**, Jose A. Carsí, Isidro Ramos, *Model-Driven Development of Aspect-Oriented Software Architectures*, The Computer Journal, Oxford Journal, (**JCR 2006: 0.593**) (submitted, status: first review)

The work presented in this chapter has been published in the following publication:

- **Jennifer Pérez**, Cristóbal Costa, Jose A. Carsí, Isidro Ramos, *PRISMA CASE*, XII Conference on Software Engineering and Databases (JISBD), Zaragoza, Spain, 12-14 September. (Demonstration, In Spanish)

CHAPTER 7

VERIFICATION

Most mistakes that are made during the software production come from the first stages of the software life cycle. For this reason, there is an increase in the number of proposals that try to deal with the problems that appear in these first stages. They usually improve the user help during the modelling process. A fundamental mechanism that must be provided to guide the user during the modelling process is the verification of models. The verification of models allows the detection of modelling mistakes and avoids that these mistakes will be spread throughout the rest of stages.

This chapter presents how the PRISMA approach provides a complete support for the verification of aspect-oriented architectural models following the MDD approach. The verification proposal and how PRISMA CASE makes feasible this verification are presented in detail.

7.1.INTRODUCTION

Nowadays, to provide software production with properties such as reliability, quality and easy maintenance is one of the challenges of software engineering. Most mistakes that are made during the software production come from the first stages of the software life cycle. Since these mistakes grow in an exponential way as projects progress, it is necessary to focus on the improvement of these first stages instead of postponing the solution for late stages. Thus, most proposals that try to solve these problems improve the software development by automatizing

the first stages of the software life cycle following the Model Driven Development Paradigm (MDD) [Am04], [Bey05].

The use of models to develop software provides important advantages such as the high level of abstraction and the technology independence. The high level of abstraction that models provide permits working with metamodels in the same way as with specific models or domain-specific models. Whereas the solutions that are independent of technology can generate the application code by means of automatic code generation techniques for different technologies and programming languages.

The automatic code generation avoids the mistakes of correspondence between the semantics of the model and the application code. However, it does not prevent the modelling mistakes that the user makes. In order to prevent these modelling mistakes, it is necessary to provide user help in order to guide the user during the modelling process. This guidance mechanism must be well-supported by the approach and its corresponding tool.

The verification and the validation of software are not only possible to be performed in the testing stage of the classic software life cycle by means of black-box and white-box testing for the application code. The verification and validation can be also performed in the modelling stage in order to detect modelling mistakes. These modelling mistakes can be related to the structure and/or the behaviour of the model. As a result, the verification and validation of models should be part of the software development process, and they should be integrated into those processes that try to improve the development of the first stages of the software life cycle.

On the one hand, the verification of models allows us to know if a model satisfies or not the constraints that its metamodel defines. If a model satisfies every constraint of its metamodel, it is possible to state that the model is correct. On the other hand, the validation of models allows us to know if the behaviour of a model is the one that the user expected and if a model satisfies certain quality properties. The validation of properties and model is performed by using prototyping techniques and model checking.

This chapter is focused in the verification of PRISMA models in order to avoid the code generation from incorrect models. In addition, this verification of models allows us to propagate these modelling mistakes to the following stage, i.e., the code generation stage. The

PRISMA metamodel defines a set of constraints that every PRISMA architectural model must satisfy (see section 4.2). This is made feasible by the verification process of the PRISMA CASE tool, which assists the user the whole time.

7.2. VERIFICATION IN PRISMA

The MDD process of PRISMA is based on the hierarchy of levels that MOF (Meta-Object Facility) [MOF02] proposes (see section 6.2.1). This hierarchy of PRISMA models implies that there are two kinds of model verification in PRISMA: verification of architectural models and verification of architectural configurations.

The verification of architectural models consists in checking that PRISMA type architectural models satisfy the properties and constraints that are defined in the PRISMA metamodel. Whereas, the verification of architectural configurations consists in checking that a configuration of instances satisfies the architectural model that it is instance of, i.e., interconnections and compositions among instances are compliant with the interaction and composition patterns of the architectural model. This chapter is focused on the verification of architectural models from the PRISMA metamodel.

7.2.1. *Verification from the PRISMA metamodel*

The PRISMA metamodel defines the properties of PRISMA models in a precise way by means of metaclasses, relationships among metaclasses and constraints. These metaclasses define a set of properties and services for each concept considered in the model. The metaclasses and their relationships define the structure and the information that is necessary to describe PRISMA architectural models. In addition, the PRISMA metamodel defines the constraints that cannot be specified using the structure or the information of the metamodel. These constraints are associated to a metaclass of the metamodel, specifically the metaclass that is affected by the constraint. The structure, information and constraints of the PRISMA metamodel must be satisfied by PRISMA architectural models in order to ensure that an architectural model is correct.

The verification process of the PRISMA architectural models exactly consist in checking that the models satisfy the following properties: (1) the types of model contain all the

information that their metaclasses establishes, (2) the relationships of the model connect the types in suitable way, (3) the number or relationships between types is correct, and (4) the constraints of the metamodel are satisfied. This verification process must be always applied to the modelling process of PRISMA architectural models and must guide the software architect the whole moment.

7.2.2. *Kinds of constraints*

Any metamodel have two kinds of constraints: hardconstraints and weakconstraints. These two kinds of constraints also appear in the PRISMA metamodel (see section 4.2).

7.2.2.1. *Hardconstraints*

Hardconstraints are those that must always be satisfied without taking into account the modelling process situation. An example of hardconstraint is the relationship between aspects and attributes. An attribute must always be associated to the aspect concept, it will never be associated to another concept (see section 4.2.1.2). This is a hardconstraint of the PRISMA metamodel because if an attribute would be associated to another concept of the architectural model, it would violate the PRISMA model. Another example is the fact that a component cannot import a coordination aspect (see section 4.2.1.5). If an architect associates a coordination aspect to a component, the resulting model would violate the PRISMA model.

7.2.2.2. *Weakconstraints*

Weakconstraints are those that can be violate during the modelling process, but once the architectural model will be finished, all of them must be satisfied.

Architectural models have a lot of weakconstraints associated to them. For example: an architectural element must import one aspect at least and must have one port at least (see section 4.2.1.3). This is an example of a weakconstraint of the PRISMA metamodel because it is possible to define an architectural element without establishing its ports and/or aspect, and to establish them later.

Weakconstraints are necessary in any modelling process due to the fact that the definition of hardconstraint dependencies among two concepts implies that none of them can be modelled. For example, if a constraint establishes that any port must be associated to an architectural

element and another constraint establishes that any architectural element must have a port, none of them could be created. The creation of an architectural element will require the previous existence of a port and vice versa.

7.2.3. *Kinds of verification*

Weakconstraints provides more flexibility to modelling process. The fact that there are weakconstraints that are not satisfied means that the modelling process has not finished. However, it is possible that there can be parts of the architectural model that are finished and the architect want to verify them.

7.2.3.1. *Partial Verification*

The partial verification consists in only applying those constraints that affect the elements, concepts or parts of the model that have been selected by the architect for their verification. This kind of verification allows the architect to verify the model in an incremental way, as well as to verify elements of the model for their later storage in repositories and/or reuse in other models. For example, in PRISMA, the verification of a specific component would consist in verifying that it has all the needed relationships that the metaclasses *ArchitecturalElement* and *Component* establish, and it satisfies all the OCL rules of both metaclasses.

7.2.3.2. *Complete Verification*

The complete verification is the verification that is applied to the complete architectural model. As a result, the complete verification consists in verifying all the constraints that must satisfy a model. In PRISMA, this process implies that all the restrictions of the PRISMA metamodel are checked.

7.3. VERIFICATION IN PRISMA CASE

The PRISMA approach is supported by PRISMA CASE, a development framework for aspect-oriented software architectures that follows the MDD proposal. This CASE tool guides the user during the development process and facilitates his/her task by providing: (1) The use of graphical modelling [Per06a], [Per06b] to specify the models instead of using a formal ADL [Per06d], (2) Support for the partial and complete verification of architectural models, (3) C#

code and formal AOADL automatic generation, and (4) the behaviour validation by means of the execution of the generated code and the interaction with the final application. This section presents in detail how PRISMA CASE supports the hard and weak constraints of the PRISMA metamodel, and the partial and complete verification of PRISMA architectural models. In this way, PRISMA CASE guides the user throughout the MDD process.

The verification process defined for PRISMA has been included in PRISMA CASE. It is based on the PRISMA metamodel, which is also part of the PRISMA CASE tool (see 6.3.2).

7.3.1. Hardconstraints in PRISMA CASE

Hardconstraints are part of the graphical modelling behaviour of the tool. As a result, those relationships that violate hardconstraints cannot be drawn. The hardconstraints have been defined by means of: (1) the specification of the PRISMA metamodel in DSL Tools and the corresponding graphical representations of its concepts, and (2) the development of the graphical constraints for each metaclass of the PRISMA metamodel.

Since the PRISMA CASE tool is generated from the PRISMA metamodel and its graphical metaphor that have been introduced in DSL Tools, the dependency and association relationships among the metamodel metaclasses are inherently satisfied by the PRISMA CASE tool. As a result, the graphical metaphor of each metaclass embodies the inclusion relationships by defining graphical representations that group a set of concepts. For example, the metaclass *Aspect* has an associated graphical metaphor that is a container of other concepts, the *Aspect* has an inclusion relationship with them. Figure 67 shows the graphical representation of the coordination aspect *CProcessSUC* of the *TeachMover* architectural model, and how the attribute *TempHalfSteps*, which stores the current position of the robot's joint, is included as part of the graphical representation of the aspect. As a result, the itself graphical representation satisfies the constraint: <<*An attribute does not have its own entity, it can only be defined inside an aspect.*>>. The rest of concepts that are necessary to specify an aspect have the same constraint (see section 4.2.1.2).

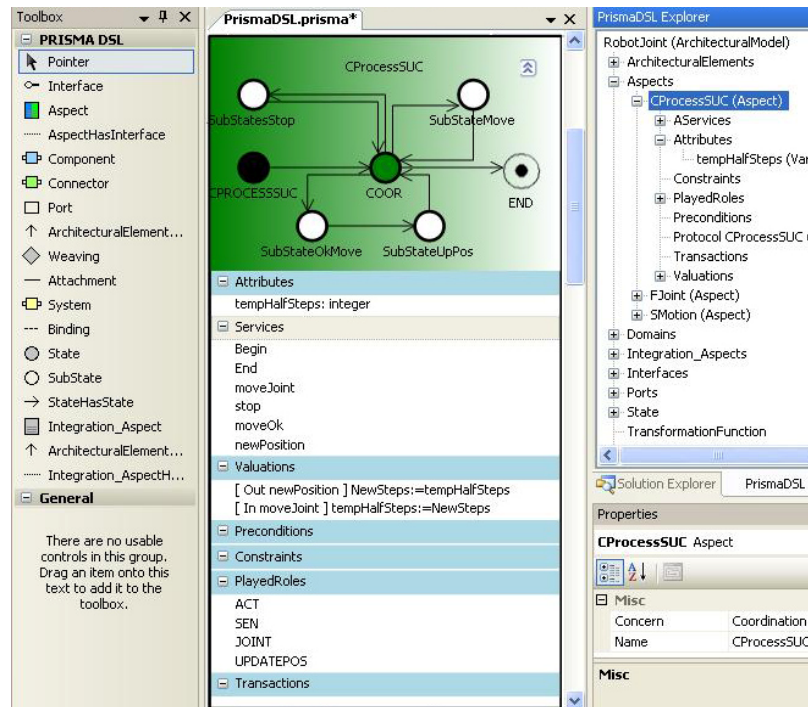


Figure 67. Graphical representation of an aspect in PRISMA CASE

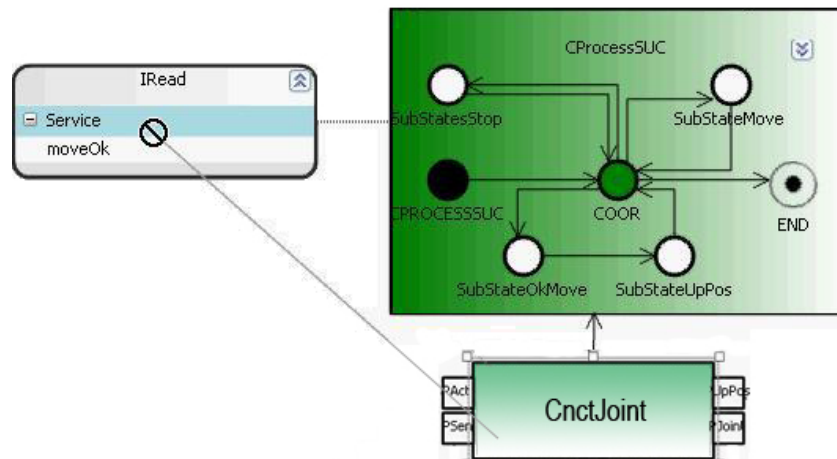


Figure 68. Relationship verification using graphical modelling primitives

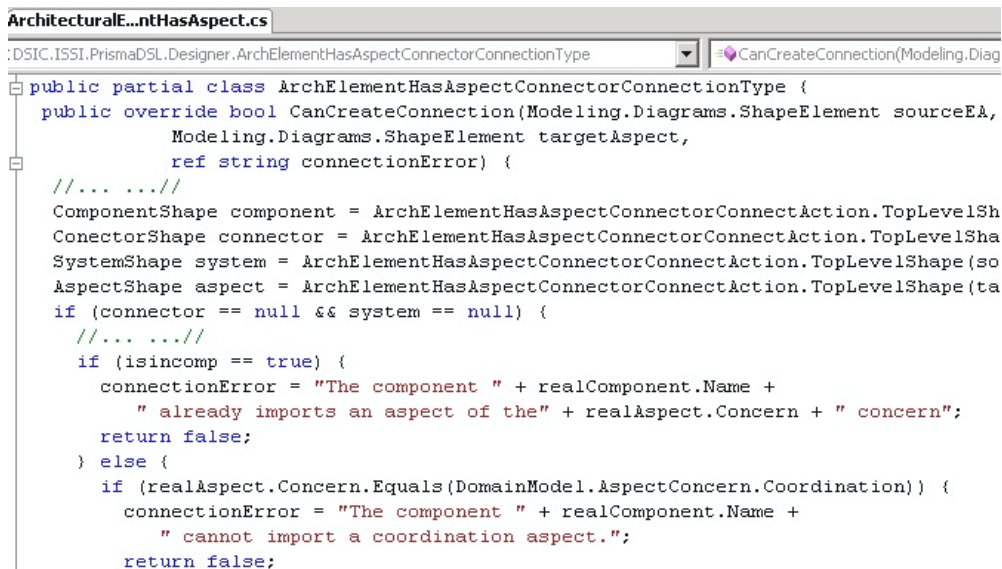
In addition, the referential relationships are also satisfied in an inherent way. Therefore, the modelling tool guides the architect by only allowing the connections of those elements that have a referential relationship between them in the PRISMA metamodel. Each connection has an associated graphical metaphor. For example, the relationship between the concept architectural element and the concept aspect corresponds with the modelling primitive *ArchitecturalElementHasAspect*, whose graphical representation is an arrow. It allows the connection between an architectural elements and an aspect. If the architect used this primitive to connect an architectural element with, e.g. an interface, a forbidden sign would appear to denote that it is impossible to link these two concepts using this graphical primitive (see Figure 68).

As it has been explained in section 7.2.1, the PRISMA metamodel defines the constraints that cannot be specified using the structure or the information of the metamodel. They are OCL rules that are associated to the metaclasses. These constraints have been introduced in PRISMA CASE by extending the partial classes of PRISMA metaclasses that have constraints associated (see section 6.3.2). Thus, the constraints of the PRISMA metamodel are verified during the modelling process. DSL tools also distinguishes between two kinds of verification: verification rules that must always be satisfied (*hardconstraints*), and verification rules that must be satisfied once the model has been completely finished (*verification rules*). They are PRISMA *hardconstraints* and PRISMA *weakconstraints*, respectively.

On the one hand, *Verification rules* are not verified while the user is modelling. They are related to the concept of the metamodel, and are verified when it is explicitly requested by the user or when the model is saved. These *Verification rules* act as warnings during the modelling process. These warnings must be rectified before the model is finished so that, it is compliant with the PRISMA metamodel.

On the other hand, *Hardconstraints* are verified while the user is modelling. They are related to the graphical metaphor and they do not permit links between certain graphical entities, compositions of certain entities, changes of name, etc.

Figure 69 show the partial C# class of the relationship between the metaclasses *ArchitecturalElement* and *Aspect*. This partial C# class is associated to the graphical metaphor of the relationship, and it has been extended to check the *hardconstraints* that are related to the fact that architectural elements import aspects. Specifically, this partial class checks the following OCL rules: <<An architectural element cannot import more than one aspect of the same concern>> (see section 4.2.1.3), and <<A component cannot import an aspect whose concern is coordination>> (see section 4.2.1.5). These constraints are checked by means of the method *CanCreateConnection* of the partial class *ArchitecturalElementHasAspect*.



```

ArchitecturalElementHasAspect.cs
:DSIC.ISSI.PrismaDSL.Designer.ArchElementHasAspectConnectorConnectionType
CanCreateConnection(Modeling.Diag
public partial class ArchElementHasAspectConnectorConnectionType {
    public override bool CanCreateConnection(Modeling.Diagrams.ShapeElement sourceEA,
        Modeling.Diagrams.ShapeElement targetAspect,
        ref string connectionError) {
        //...
        ComponentShape component = ArchElementHasAspectConnectorConnectAction.TopLevelSh
        ConnectorShape connector = ArchElementHasAspectConnectorConnectAction.TopLevelSha
        SystemShape system = ArchElementHasAspectConnectorConnectAction.TopLevelShape(so
        AspectShape aspect = ArchElementHasAspectConnectorConnectAction.TopLevelShape(ta
        if (connector == null && system == null) {
            //...
            if (isincomp == true) {
                connectionError = "The component " + realComponent.Name +
                    " already imports an aspect of the" + realAspect.Concern + " concern";
                return false;
            } else {
                if (realAspect.Concern.Equals(DomainModel.AspectConcern.Coordination)) {
                    connectionError = "The component " + realComponent.Name +
                        " cannot import a coordination aspect.";
                    return false;
                }
            }
        }
    }
}

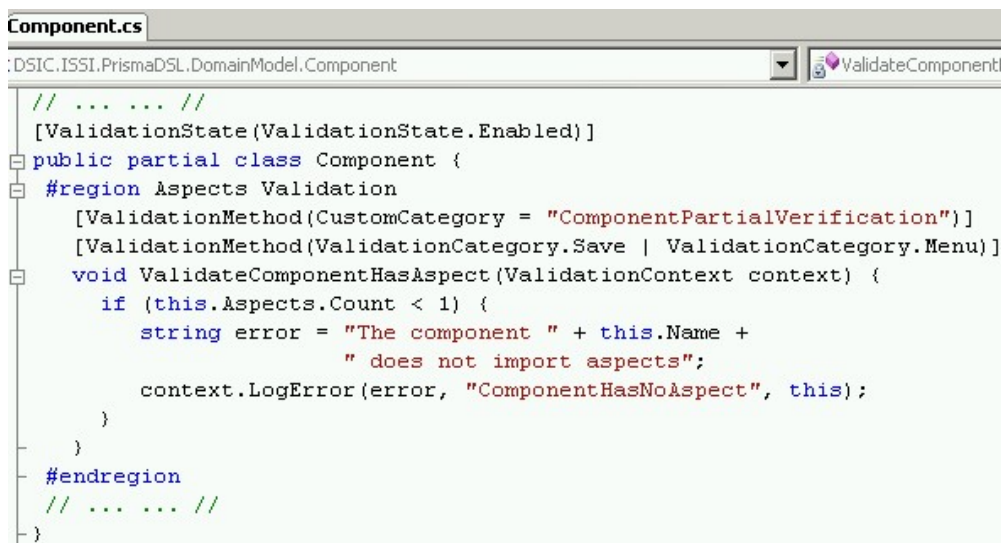
```

**Figure 69. The partial C# class of the relationship
*ArchitecturalElementHasAspect***

As a result, each time that an architect imports an aspect from an architectural element, the method *CanCreateConnection* is executed and both constraints are checked. The source code shows that if the constraints are not satisfied, a graphical connection error is launched. This error has an associated text message that is shown to the user together with a forbidden sign. From this example, it is possible to conclude that *hardconstraints* are part of the graphical metaphor and they are checked without being requested by the architect.

7.3.2. Weakconstraints in PRISMA CASE

Since weakconstraints are implemented using *Verification rules*, that are associated to the concept instead of the graphical metaphor. They are only checked when the architect requests their execution, the architectural model is saved, or once the architectural model has been finished and the C# code generation is launched.



```

Component.cs
:DSIC.ISSI.PrismaDSL.DomainModel.Component
ValidateComponentHasAspect

// ... .. //
[ValidationState(ValidationState.Enabled)]
public partial class Component {
    #region Aspects Validation
        [ValidationMethod(CustomCategory = "ComponentPartialVerification")]
        [ValidationMethod(ValidationCategory.Save | ValidationCategory.Menu)]
        void ValidateComponentHasAspect(ValidationContext context) {
            if (this.Aspects.Count < 1) {
                string error = "The component " + this.Name +
                    " does not import aspects";
                context.LogError(error, "ComponentHasNoAspect", this);
            }
        }
    #endregion
    // ... .. //
}

```

Figure 70. The partial C# class of the metaclass *Component*

Figure 70 shows the partial C# class of the metaclass *Component*, whose method *ValidateComponentHasAspect* checks if a component imports one aspect at least. The source code of the method shows that if the constraint is not satisfied, the error is saved in a Log file together the rest of errors that have been identified in the verification process instead of being immediately shown as hardconstraints do. Then, when the verification process is requested by the user, the list of errors that have been previously saved in the Log file is displayed in the *Error List* window (see Figure 71).

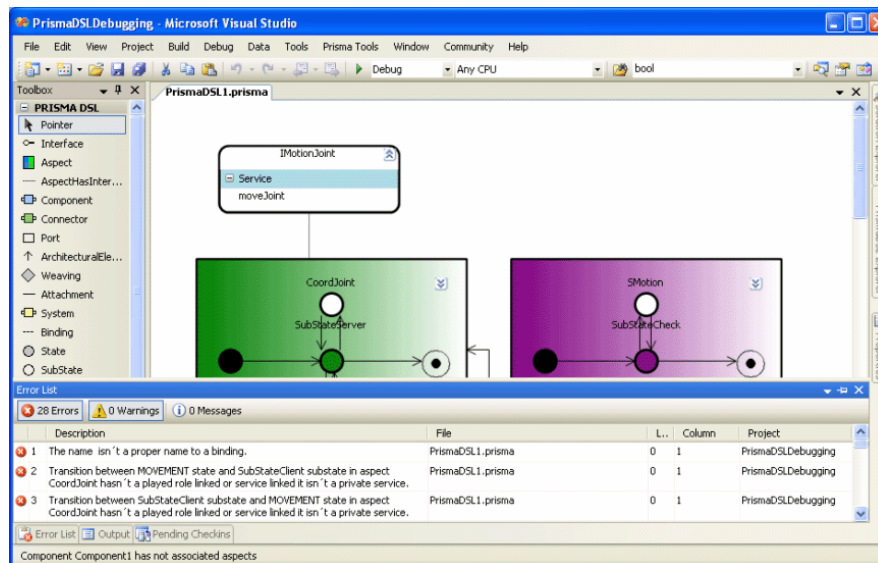


Figure 71. Error List

7.3.3. Partial and complete verification in PRISMA CASE

Verification rules can also be checked whenever the user requires it. The PRISMA menu offers the option of checking these rules in a complete or partial way (see Figure 72). The complete way checks all the verification rules, while the partial way allows you to only check one kind of PRISMA type. The options that are provided to check an architectural model in a partial way are the following: interfaces, aspects, components, connectors, systems and attachments. For example, if the user requests the *Interface Verification*, only the rules associated to interfaces are checked. The advantage of this partial verification is that the user can incrementally check the models and focus on the problems of a specific type of the model.

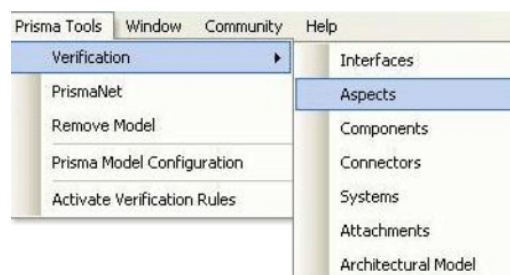


Figure 72. Verification Menu

In addition, the modelling tool offers the mechanism of checking only one element of the architectural model. This is possible by executing the option that appears in the contextual menu that is associated to the element that the user wants to check. For example, Figure 73 shows the contextual menu that only verifies the interface *IMotionJoint*.

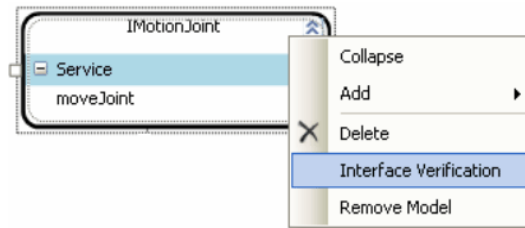


Figure 73. Contextual menu of an interface

7.4.RELATED WORKS

In the MDD field there are three kinds of strategies to perform the verification of models [Mod06]: analysis, construction and monitoring. Analysis strategies define the constraints together with the metamodel using OCL rules. Then, a verification tool is in charge of checking that models satisfy these constraints. These strategies perform a complete verification of the model. There is a wide range of tools that offer OCL checkers [Tov07].

The constructive strategies construct models by applying QVT transformation rules [QVT05] or ATL [Bez06]. The monitoring strategies define models by means of graphical tools that prevent the creation of inconsistent models. Both strategies have in common that they construct models in an incremental way by defining implicit modelling constraints. However, the constraints are spread through the transformation rules and the monitoring logia. Therefore, this is an inconvenient because the maintenance of constraints is really difficult. There are mixed approaches, such as the Bézivin & Jouault approach [Jou05], which use the engine of ATL transformation rules to verify that the OCL rules of the metamodel are satisfied. But the main inconvenient of these approaches is the fact that OCL rules are separated from the metamodel.

The proposals that have been acquired more relevant are those that combine analysis and monitoring strategies. They take advantage of the flexibility that provide the analysis strategies and the security that offers the monitoring strategies. For example: Microsoft DSL-Tools [DSL07], AMMA [Kur06], XMF-Mosaic [Tov07], EMF [EMF07] and GME [GME07]. These tools provide support for all the stages of the MDD process, although each one provides this support using a different meta-metamodel, different strategies of consistence verification, and a different code generation process.

The choice of one tool or another depends on different factors. One of the reasons to choose DSL Tools for developing PRISMA CASE was that it offers mechanisms for monitoring and analyzing the verification of the consistence. The monitoring is offered in an implicit way by its graphical modelling tool, once the metamodel has been previously defined in the DSL Tools. The analysis of constraints is offered by customizing the metaclasses using the partial class facility.

It is important to mention that despite the fact that the area of software architectures is making an important effort to give support for validation, there are not solid proposal for the verification of architectures. As a result, a solution for this lack is presented in this chapter.

7.5. CONCLUSIONS

In this chapter the need to support model verification throughout the MDD process is set out. An important contribution in the area is the classification of constraints that is presented in the chapter: hardconstraints and weakconstraints. From this classification the contribution takes a step forward and the partial and incremental verification has been proposed instead of only taking into account the complete verification of the model. Therefore, the flexibility of the modelling process is increased by adapting the help to user needs. In addition, a methodology to support the verification of models during the MDD process is defined by establishing the needed mechanisms to offer all kinds of verification.

This methodology has been applied to the PRISMA approach. The PRISMA model defines through its metamodel, the properties and constraints that any PRISMA architectural model must satisfy. These constraints have been analyzed in order to be classified into

hardconstraints and weakconstraints. It has been only defined as hardconstraints, those that are strictly necessary. As a result, the PRISMA modelling process has more flexibility. PRISMA CASE has made the application of this methodology to the PRISMA approach feasible.

PRISMA CASE provides support for : (1) the partial and complete verification process, (2) the difference between hardconstraints and weakconstraints, and (3) the incremental verification of aspect-oriented architectural models by means of the partial verification of types (aspects, components, etc) or the partial verification of specific elements of the model. As a result, PRISMA is presented as an approach that facilitates the modelling process of aspect-oriented architectural models by means of a well-defined verification process that follows the MDD approach.

The work presented in this chapter has been published in the following publication:

- **Jennifer Pérez**, Cristóbal Costa, Jose A. Carsí, Isidro Ramos, *Verification of Aspect-Oriented Architectural Models*, XII Conference on Software Engineering and Databases (JISBD), Zaragoza, Spain, 12-14 September. (In Spanish)

CHAPTER 8

COTS: Commercial Off-The-Shelf

Reusability reduces the development time of software systems because artefacts are only programmed one time and can be used more than once. Reused software artefacts guarantee their quality and suitable functionality because they have been tested and used before. As a consequence, Commercial Off-The-Shelf (COTS) importation has acquired relevance in the last few years. This chapter presents a proposal for integrating COTS into aspect-oriented architectural models that are developed and maintained following the Model-Driven Development (MDD) approach. The proposal is based on the PRISMA approach, which gives a complete support to the development of technology-independent, aspect-oriented software architectures. PRISMA improves the reusability of software by combining COTS, components, and aspects. In addition, PRISMA integrates COTS into its MDD process to automatically obtain the complete application code.

8.1. INTRODUCTION

In the last few years, the high complexity of software has increased the time and the staff that are invested in the development and maintenance processes of software. As a result, there is greater interest in research areas to reduce this time and cost. In order to achieve these goals, software community is making a big effort to provide techniques that improve the reusability of software.

Reusability of software allows the same software artefact to be used in different places of the same application or in different applications. The artefact is only programmed once and can be used many times. This reusability reduces the development time of software systems, and their quality and suitable functionality are guaranteed because they have been tested and used before.

The Component-Based Software Development (CBSD) approach [DSo99][Szy98] is used in the field of software architectures [Per92], [Sha96]. This approach decomposes the software system into reusable entities (black boxes) called components. As a result, software architectures can be described preserving the reusability of their components and are presented as a solution for the design and development of complex software systems.

Another approach that has emerged to improve reusability is the Aspect-Oriented Software Development (AOSD) approach [Kiz01], [Kiz97]. This approach allows for the separation of concerns by modularizing crosscutting concerns into a separate entity called aspect. As a result, the same aspect can be reused by different software artefacts, which are usually objects.

In addition, there is another approach to develop software that improves its reusability. It consists of buying components (black boxes that offer a set of services that are properly documented for use) from third-party developers and then, integrating them into the system. These components that are for sale commercially are known as Commercial Off-The-Shelf (COTS) [Obe97], [Car00]. The use of COTS during the development process has increased in the last few years due to market competitiveness. This increase has led developers to try to reduce the time required to develop a software product. Since developers are using the software components of other companies more and more, COTS importation has acquired greater relevance. This is because tools that allow the reuse of their components combined with COTS importation achieve the highest reuse and quality code. PRISMA is an approach whose aim is to provide these two benefits.

This chapter takes a step forward with regard to previous work of the PRISMA approach. It presents a new version of the PRISMA approach, which is able to integrate COTS into its MDD process. The chapter describes how PRISMA integrates COTS into its aspect-oriented software architectures without violating the properties of the PRISMA model. As a result,

PRISMA improves the reusability of software by combining COTS, components and aspects. In addition, the chapter explains the process to obtain a complete application code, which is composed of the generated code from its architectural models and the code from COTS. It is then ready to be executed on the .NET platform [Per05b].

8.2. THE SOFTWARE ARCHITECTURE OF A TEACHMOVER'S JOINT USING COTS

In this section present the evolved version of the *TeachMover* case study that has been used as an example in the rest of the thesis. This new version uses COTS to permit the communication between hardware and software, i.e., the hardware robot and software components of the architecture. This example is going to be used in this chapter to illustrate how COTS are introduced in PRISMA.

The TeachMover architecture has different levels of abstraction for its components, connectors, and the interactions with each other. The lowest abstraction level of the robot architecture is defined by a system called *Joint* or *SUC*, which is composed of components that interact with the hardware joints of the robot (see section 2.3.2.2). The communication between the robot and the computer is performed through the serial port of the computer because the computer is connected to it. The *Joint* system defines a joint of the robot. It is composed of two components and a connector and their corresponding connections:

- **ActSen:** This software component is in charge of communicating with the hardware joint of the robot. It communicates with the actuator of the hardware joint when commands are sent to the hardware joint of the robot. These commands are performed by the hardware joints or the tool. This software component also notifies the joint system when the commands have been performed successfully.
- **WrapAspSys:** This software component encapsulates the behaviour and the state related to the software joint, such as the position of the joint and its movements.
- **CnctJoint:** This software connector coordinates the *ActSen* component and the *WrapAspSys* component.

Since the *ActSen* component is responsible for interacting with the hardware pieces of the robot, it is necessary to use a COTS that provides the services to move the joint and to listen to the results through the RS232 serial port. The COTS that have been used in this case study is a dynamic linking library called *RS232* (RS232.dll). This COTS provides a set of services. To facilitate the readers' comprehension of the example in this chapter, it is illustrated the integration of the COTS using the most representative service, the service *Send*. Its definition is the following:

```
Send (int joint, int halfsteps, int speed): int
```

The first parameter specifies the joint to which the movement is sent. Each joint of the TeachMover has a predefined number (1. Base, 2. Shoulder, 3. Elbow, 4. Wrist (right rotation), 5. Wrist (left rotation), and 6. Tool). The second parameter is the number of halfsteps that the robot is going to be moved when the service is executed. And finally, the third parameter corresponds to the speed of the processing movement.

8.3. INTEGRATING COTS INTO THE PRISMA MODEL

COTS integration is usually presented as a handicap for developers because there are many incompatibilities with programming languages, frameworks, platforms, communications, etc. Therefore, COTS must be adapted so that it can be reused in a software system. There are three well-known techniques for integrating COTS in software systems: wrappers, gluewares, and proxies. Wrappers wrap COTS in a kind of software artefact of the software system; gluewares are intermediaries between the COTS and the software components of the software system; and proxies are adapters that hide the incompatibilities between COTS and the components of the software system. These three techniques are black box techniques to integrate COTS into software systems.

The wrappers technique was chosen over gluewares and proxies, since it integrates COTS into a model using its own concepts, which is the goal of PRISMA. There are two PRISMA candidates that can serve as wrappers: components and aspects.

8.3.1. *COTS as components*

A PRISMA component can act as a wrapper because it can wrap a COTS and publish its interface through the component port (see Figure 74). As a result, the services of the COTS can be requested through the port.

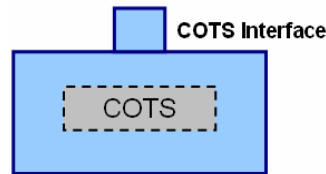


Figure 74. COTS as components

This proposal of integrating COTS into software architectures does not violate the concepts of any architectural model. In addition, the proposal allows the rest of components of the software architecture to interact with COTS since it is a component that has been specified from scratch using the model constructors. However, one of the drawbacks of this proposal is that it is really difficult to extend the behaviour of the COTS. There is also another drawback that specifically affects the PRISMA model: the COTS behaviour is directly provided by the component. As a result, this proposal of wrapping COTS using components does not fit the PRISMA model properties since PRISMA component behaviour is always defined inside aspects and never inside components. Component behaviour cannot be specified inside components because they do not provide mechanisms to do so, therefore, it is not possible to specify the properties and processes that are needed to integrate the services of the software system with the services of the COTS. However, aspects do provide properties, services and protocols that can be used to specify the integration behaviour of the COTS into the software system.

8.3.2. *COTS as aspects*

An aspect defines the structure and the behaviour of a specific concern of the software system. An aspect declares a number of interfaces and defines a semantics for the services that these interfaces publish. As a result, an aspect can act as a wrapper because it can wrap a COTS and declare the interface of the COTS. In order to the COTS integration to be compliant with the

PRISMA model, the aspect must be imported by an architectural element of the model so that it can publish its interfaces through its ports and communicate with other architectural elements through the channels (see Figure 75). This wrapper proposal provides an easy way of extending the behaviour of the COTS. The architectural element that imports the wrapper aspect can import other aspects and weave with each other using weavings. As a result, the architectural element provides an extended version of the COTS through its ports, which is compliant with the PRISMA model and is also more flexible than the proposal presented in section 8.3.1. For this reason, aspects are the PRISMA element that has been selected to wrap COTS. In fact, a new kind of aspect called Integration Aspect has been defined to do this.

An integration aspect allows the integration of COTS components in PRISMA software architectures in an abstract way. It contains a reference to the COTS that it is related to and uses a PRISMA interface that defines all the services that the COTS provides. The integration aspect also uses the interfaces of the architectural model that define services that must be transformed into requests to the COTS. As a result, the protocol of the PRISMA integration aspect consists of redirecting each request service to its corresponding COTS service.

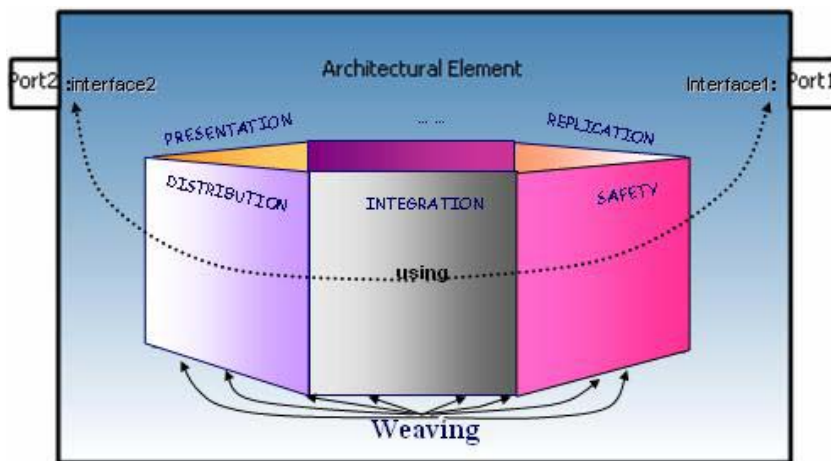


Figure 75. COTS as aspects

All the architectural elements that import the integration aspect are crosscut by the concern that it represents. For example, in the case study, all the components that import the integration

aspect are crosscut by the *hardware interaction* concern. This is due to the fact that the COTS *RS232.dll* allows communication with the serial port that the robot is connected to.

8.4. USING COTS DURING THE MDD PROCESS

The PRISMA approach follows the MDD process to define its application. As it has been presented in section 6.2, PRISMA supports the MDD process through its PRISMA CASE Tool. The addition of COTS to this process has been also made feasible thanks to PRISMA CASE. Next, how COTS have been introduced in the PRISMA CASE modelling tool and in the PRISMA CASE model compiler.

8.4.1. *The use of COTS in the PRISMA CASE modelling tool*

The aspect is one of the modelling primitives of the PRISMA Modelling Tool of PRISMA CASE (see Figure 58). Every concern is modelled using the aspect primitive, which is represented as a rectangle (see number 1, Figure 58). It includes the definition of attributes, services, valuations, preconditions, constraints, transactions, played_roles and a protocol. The concern of the aspect is determined by its properties. In order to visually identify the concern of an aspect, each concern has a colour associated to it; depending on the value of the aspect concern, the aspect is painted in one colour or another.

However, the integration aspect is not modelled using the aspect modelling primitive as other concerns do. There is a specialized integration aspect modelling primitive (see number 5, Figure 58) because it only provides the properties that are strictly necessary for integrating COTS into software architectures. The integration aspect only defines services, played_roles, and a protocol (see the IACOT aspect specification in Figure 76). The integration aspect is graphically represented by a rectangle. Integration aspects are painted grey to denote their wrapper semantics, which are viewed as a black box. In addition to the name of the aspect, the aspect has a COTS property to specify the name of the COTS that it wraps (see the Properties Window in Figure 76 (lower right)).

The importation of interfaces by an aspect is specified using the link *AspectHasInterface*, which is provided by the tool box (see number 2, Figure 58). However, integration aspects use

a different link called *Integration_AspectHasInterface*. It is graphically represented by a line (see number 7, Figure 58).

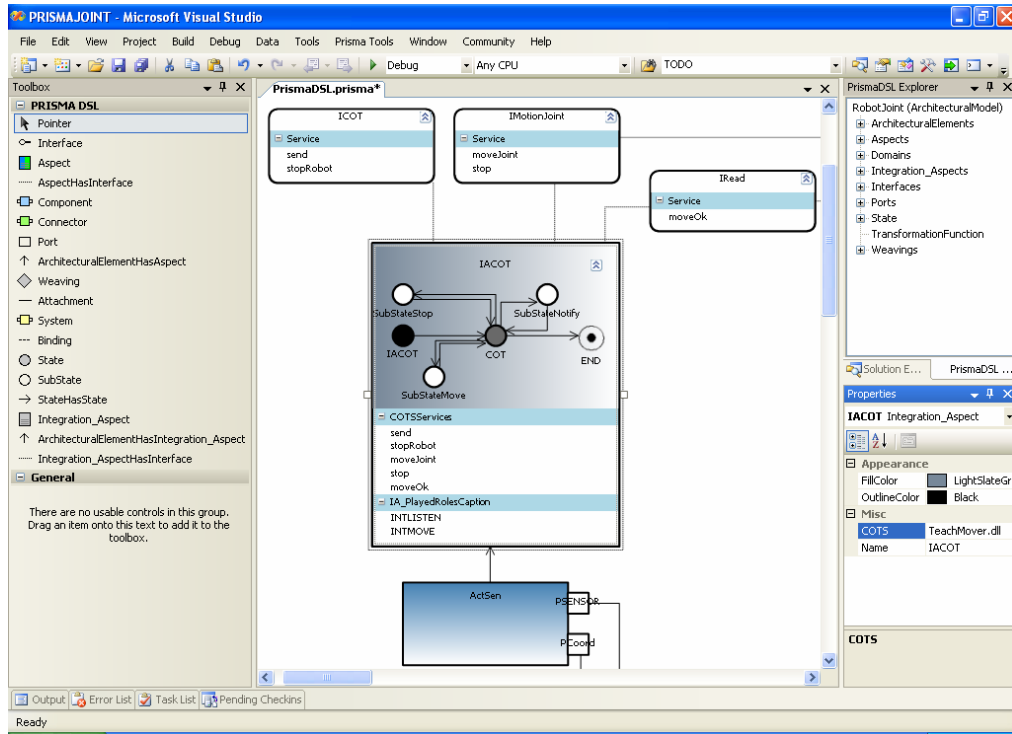


Figure 76. Integration of the TeachMover.dll into a PRISMA architectural model

Finally, architectural elements are represented by rectangles that have one pin for each one of the ports that are associated to them (see number 3, Figure 58 (components)). The importation of aspects by architectural elements is specified using the link *ArchitecturalElementHasAspect* (see number 4, Figure 58). In addition, there is another specialized link for importing the integration aspect called *ArchitecturalElementHasIntegrationAspect* (see number 6, Figure 58).

The IACOT integration aspect that is illustrated in Figure 76 integrates a TeachMover.dll into the architectural model of the TeachMover robot. For this reason, it has as a value of the COTS property the TeachMover DLL. In addition, there is an *ActSen* component that imports the IACOT integration aspect and interacts with the rest of the software architecture.

IACOT imports the interfaces *IRead* and *IMotionJoint* of the TeachMover architectural model in order to synchronize their services with the COTS services. In addition, *IACOT* imports another interface, called *ICOT*, which defines the services of the COTS. As a consequence, the *IACOT* aspect has all the services that define these three interfaces and can specify the protocol in order to synchronize them (see the COTSServices section inside the *IACOT* aspect in Figure 76). The protocol of *IACOT* consists of redirecting the requests of the services from the *IACOT* to the COTS. These services (for example: the *moveJoint* service) arrives to the *IACOT* from the ports of the *ActSen* component. To do this the protocol establishes that each time that the *movejoint* service is requested, the COTS service send is called with the same values of the parameters of the *moveJoint* service. The protocol is specified by modelling a State Transition Diagram (STD) (see the *IACOT* aspect in Figure 76).

In order to model the complete behaviour, the actuator is connected to a component called *WrappAspSys* through the connector *CnctJoint* (see section 8.2). Moreover, these three components belong to a system called *Joint* (see Figure 77).

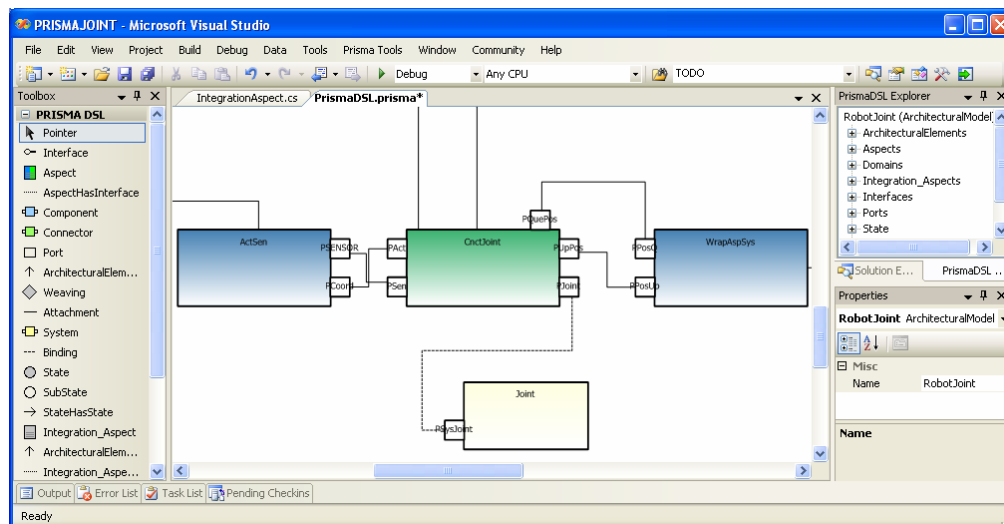


Figure 77. PRISMA architectural model of a Joint

8.4.2. *The use of COTS in the PRISMA CASE model compiler*

The PRISMA CASE also provides a model compiler, which is composed of a set of templates that automatically generate the code from the models that have been graphically modelled. A specific template has been developed for automatically generating the code of the integration aspect. The code generation template of integration aspects is different from other kinds of aspects. This is due to the fact that it must integrate the code of the COTS with the code of the aspect. However, the remaining aspects have a common template that generates the functionality of the system that each aspect describes.

There are many kinds of COTS: Web Services, COM components, ActiveX components, dynamic linking libraries, etc. This chapter is going to focus on dynamic linking libraries, specifically: native libraries and libraries that have been developed using the .NET framework.

Native libraries can be used as provided by the supplier. As a result, the template should generate the C# code that imports the set of services of the COTS by means of the C# attribute `DLLImport`. This attribute, which is provided by the C# programming language, allows us to include the service of a COTS inside its wrapper (see numbers 1 and 2 in the code presented below). In addition, it is necessary to define the COTS service as a private method that returns the result of the invocation (see number 4 in the code presented below) and the invocation of the COTS service by the corresponding service of the software architecture to make the integration (see number 3 in the code presented below). The C# code that the model compiler should generate for integrating the RS232.dll of the case study into the IACOT aspect is the following:

```
...
//1. To use the attribute DLLImport
using System.Runtime.InteropServices;
...
namespace RobotJoint
{ public class IACOT : IntegrationAspect , IMotionJoint, ICOT, IRead
  {
    //2. To define the entry point to the DLL through the send service
    [DllImport("RS232.dll", EntryPoint = "Send")]
    enum protocolStates
    {
        SubStateNotify, IACOT, COT, END
    }
    protocolStates state;
    private protocolStates State{...}
    public IACOT(string name) : base(name)    {...}
  }
}
```

//3. The request of the *moveJoint* service called the *send* service of the *ICOT* interface

```
public AsyncResult moveJoint (int NewSteps,
                              int Speed)
{
    send(NewSteps, Speed, this.aspectName);
    CallOutService("IRead", "INTLISTEN", "moveOk",
        this.aspectStateCareTaker.ActiveTransaction,
        null);
    return null;
}
```

public AsyncResult stop () {...}

//4. The request of the *send* service called the *Send* service of the *COTS*

```
private static extern int send (int Speed,
                                int HalfSteps, string Joint)
{
    int response = Send(NewSteps, Speed,
                        this.aspectName);
    return null; }
public AsyncResult stopRobot () {...}
public AsyncResult moveOk () {...} }
```

However, the code presented above is not the code generated by the PRISMA CASE model compiler because the libraries that are developed using the .NET platform cannot be used as provided by the supplier, and cannot be executed with this code. The .NET libraries need to create another wrapper that encapsulates the services that the COTS provides. This wrapper consists of a class that permits the invocation of the COTS in a transparent way and prepares the parameters required to call the COTS services. This class must be contained in a project that is a class library (see Figure 78).

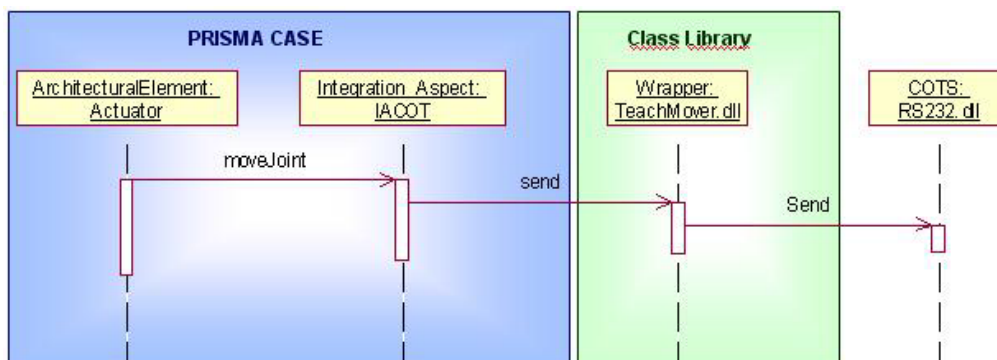
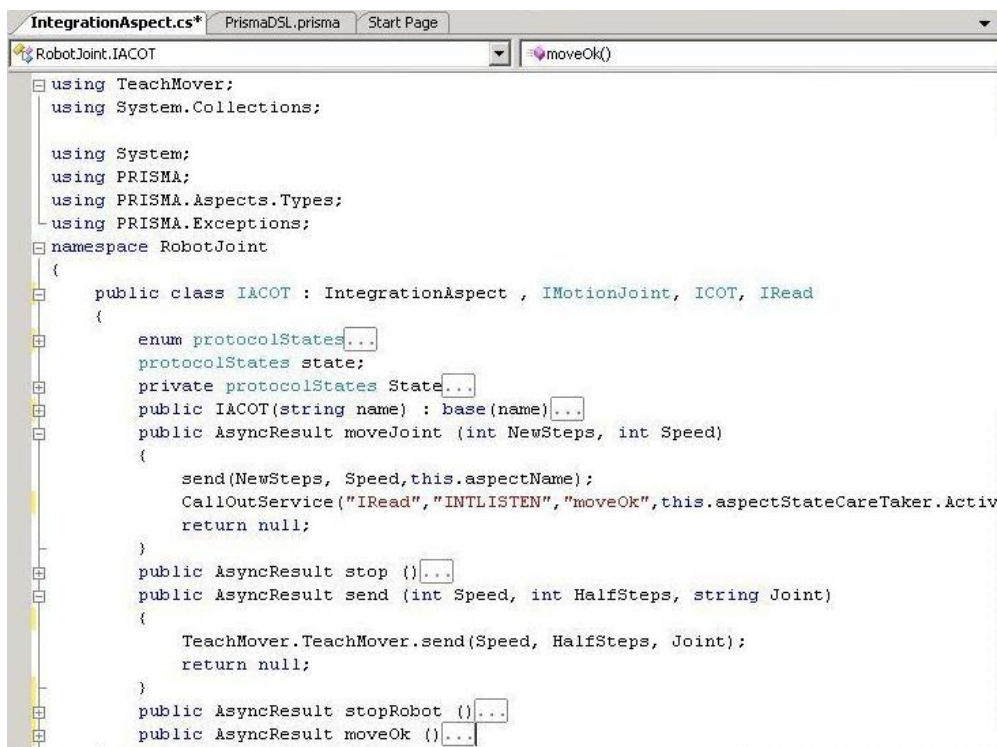


Figure 78. COTS Execution Process in PRISMACE

Since PRISMA model compiler automatically generates the code for the integration aspect, the generated code must be valid for both the native and the .NET libraries. For this reason, the solution for the .NET libraries has been chosen.

In consequence, the PRISMA integration aspect interacts with a wrapper that wraps the COTS in a library that contains the class that manages the COTS services (see Figure 78). In our example, the class that wraps the COTS is called *TeachMover.dll*. Thus, the COTS property of IACOT aspect contains the value *TeachMover.dll* instead of *RS232.dll* (see the Properties Window in Figure 77).



```

IntegrationAspect.cs* PrismaDSL.prisma Start Page
RobotJoint.IACOT moveOk()

using TeachMover;
using System.Collections;

using System;
using PRISMA;
using PRISMA.Aspects.Types;
using PRISMA.Exceptions;

namespace RobotJoint
{
    public class IACOT : IntegrationAspect , IMotionJoint, ICOT, IRead
    {
        enum protocolStates...
        protocolStates state;
        private protocolStates State...
        public IACOT(string name) : base(name)...
        public AsyncResult moveJoint (int NewSteps, int Speed)
        {
            send(NewSteps, Speed, this.aspectName);
            CallOutService("IRead", "INTLISTEN", "moveOk", this.aspectStateCareTaker.Activ
            return null;
        }
        public AsyncResult stop {}...
        public AsyncResult send (int Speed, int HalfSteps, string Joint)
        {
            TeachMover.TeachMover.send(Speed, HalfSteps, Joint);
            return null;
        }
        public AsyncResult stopRobot {}...
        public AsyncResult moveOk {}...
    }
}

```

Figure 79. The C# code that is automatically generated from the *IACOT* aspect

Figure 79 shows the code generated for *IACOT* by the PRISMA model compiler: the *IACOT* aspect imports the *TeachMover* library, the *moveJoint* service addresses its request to

the COTS by requesting the *send* service, and the *send* service calls the *send* service of the *TeachMover.dll* in a transparent way. Once the code has been generated, the COTS assemblies and project of libraries must be stored in the same folder as the generated assemblies by the PRISMA project. This is done so that COTS assemblies can be added to the compilation process and can be executed in the same way as other PRISMA software architectural elements. In our case, the generated code the *TeachMover.dll* and the *RS232.dll* are stored together in the assembly folder.

8.5. RELATED WORKS

COTS are components that are developed and sold by third parties. Since they have been run on different software systems by different companies, their quality and correct behaviour are guaranteed. However, their proper integration into software systems is both difficult and time-consuming [Vig96].

COTS integration and use has not considered as part of the software life cycle and most proposals and initiatives related to COTS are based on the implementation stage of the software life cycle only. In this chapter, it is proposed introducing COTS at the software architecture stage of the software life cycle.

There are a wide variety of ADLs that have been proposed to specify software architectures (see chapter 3). However, all of them are based on the fact that the software architecture is completely specified using their ADL constructors and none of the components are imported from other developers or architects. In addition, there are other approaches that combine software architectures with aspect-orientation (see chapter 3). Even though these approaches are closer to PRISMA approach, they do not propose mechanisms to introduce COTS in their software systems.

The work of Yakimovich et al [Yak99] creates a method that supports estimation of the cost of integration of COTS products in software architectures. The work of Guerra et al [Gue02], [Gue03a], [Gue03b] outlines a proposal that takes advantage of COTS in software architecture by introducing its specific COTS in the C2 architectural style. However, they only focus on its components for supporting fault tolerance, and it is not a proposal for a complete

software architecture of a real software system. This proposal is platform-dependent and does not use a formal ADL as recommended for the definition of software architectures.

The work of Kvale et al [Kva05] compares the advantages and disadvantages of using COTS in Aspect-Oriented Programming (AOP) or in Object-Oriented Programming (OOP). This work explains how to wrap COTS using aspects. It describes the advantages and disadvantages of this wrapping depending on the weavings process (which is where the synchronizations between the base code and the aspect code are localized). This work also states that the AOP development frameworks do not provide good mechanisms for importing COTS.

8.6. CONCLUSIONS

This chapter explains how to integrate COTS into software architectures in a novel way. This integration is feasible using aspects as COTS wrappers. Specifically, a new kind of aspect called `integration_aspect` has been defined to specify the COTS that it wraps and to specify the integration process with the rest of software architecture. In the same way as other aspects, integration aspects must be imported by an architectural element in order to publish their services through ports that enable their communication with other architectural elements. This is an advantageous way of introducing COTS into software architectures because the COTS services can be requested and received and can also be extended by using the aspect-oriented mechanisms that PRISMA offers. The integration aspect functionality can be extended by weaving it with the other aspects that the architectural element imports.

This chapter also describes how the use of COTS is supported by the PRISMA methodology thanks to the facilities that the PRISMA CASE provides. This is an important characteristic since COTS are widely used, and any development approach that needs to reduce development time must provide it.

In summary, the proposal presented here is a suitable development framework for reducing the time and cost invested in the software development process and for improving the quality of the code. This can be done since PRISMA provides mechanisms to use COTS, it reuses its aspects and components, and it uses code-generation techniques to automatically generate

code. This proposal has used in practice to develop an aspect-oriented software architecture for a robot. The robot code was automatically generated and the COTS was integrated in the code. The application code was executed through the PRISMANET middleware and it was possible to move a robot with an aspect-oriented software architecture that integrates COTS.

The work presented in this chapter has been submitted to the following publication:

- **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *Taking Advantage of COTS for Developing Aspect-Oriented Software Architectures*, Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society, Vancouver, BC, Canada, 18 – 21 February 2008. (submitted)

CHAPTER 9

THE PRISMA MDD METHODOLOGY

This chapter presents the PRISMA methodology in order to develop aspect-oriented software architectures following the PRISMA MDD process. This methodology takes advantage of the PRISMA reusability properties (coordination model, modelling and reusability facilities, the use of COTS), the graphical specification of PRISMA models [Per06a], [Per06b], and the verification process proposed by the PRISMA approach. In this chapter is illustrated how the PRISMA approach can improve the development and maintenance processes of complex software systems.

The methodology is divided into six stages: detection of architectural elements and aspects, type architectural modelling, type code generation, configuration modelling, configuration code generation and execution (see Figure 80). These six stages are applied by the analyst of the software system in an iterative and an incremental way depending on his/her needs. This thesis of master is focused on how the modelling and code generation stages in order to illustrate how the PRISMA combination of AOSD and software architectures can improve the development and maintenance processes of software.

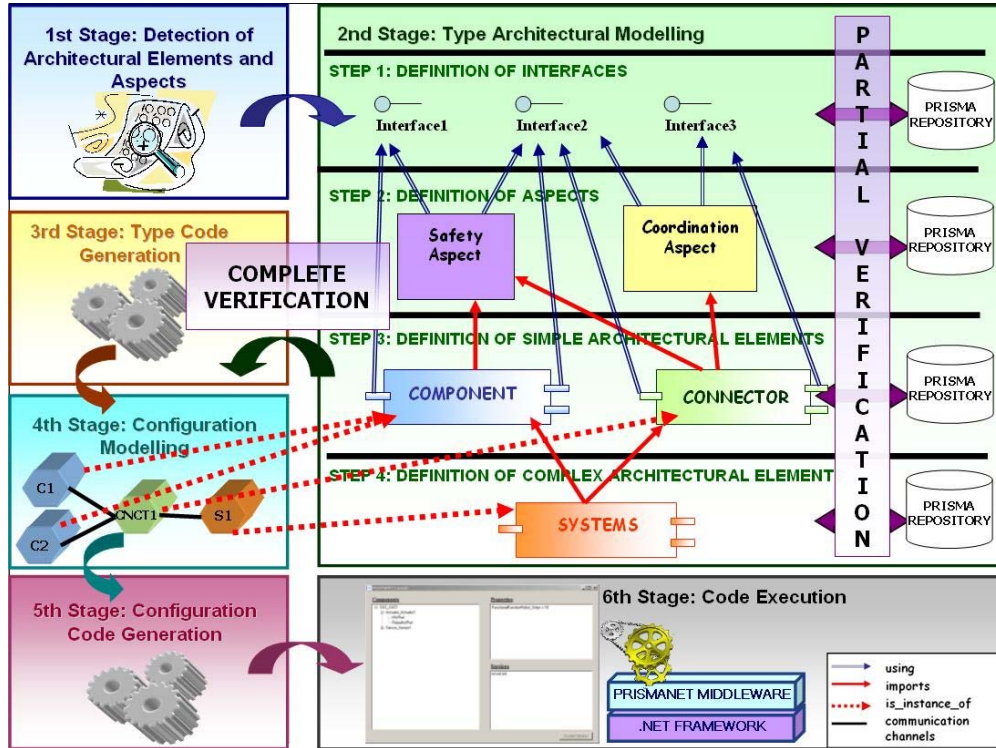


Figure 80. The methodology of the PRISMA approach following the MDD paradigm

9.1.1ST STAGE: DETECTION OF ARCHITECTURAL ELEMENTS AND ASPECTS

The first tasks for developing software architectures are to identify which architectural elements make up the architecture and to detect the aspects that crosscut the software architecture. In PRISMA, the detection of those architectural elements and aspects is performed from the requirements document in an intuitive way.

The running example that it is going to be used to illustrate our methodology is the TeachMover tele-operation system (see section 2.3.2 [TEA07]). The computational units (components) and the coordination units (connectors), which allow the architecture to synchronize components and permit the communication among them, can be identified from the manual of the TeachMover robot. In addition, the concerns that crosscut the software architecture can be detected. The mechanisms and criteria to detect concerns, components and

connectors are out of the scope of this thesis of master, so it is assumed that these elements have already been detected.

9.1.1. Identification of Architectural Elements

The manual of the TeachMover shows that the robot is composed of a set of components that represent a set of joints, a set of connectors that coordinate the joints, and a set of complex components that allows the composition of joints to form a robot. These joints permit the robot's movements. They are the following: Base, Shoulder, Elbow and Wrist. In addition, it has a gripper, whose open and close actions allow the robot to pick up and deposit objects (see [TEA06]).

From the manual of the robot the components of the robot and the different level of composition have been identified. The result of this identification process is presented in section 2.3.2.2, specifically in Figure 3 where they are illustrated in their corresponding level of granularity using a tree view. In addition, the connectors needed to coordinate these components as well as to provide a suitable behaviour of the architecture have been identified. Connectors are necessary because PRISMA introduces connectors as first order citizens of architectural models. As a result, PRISMA avoids the dependencies among components that ADLs without connectors encounter by specifying coordination rules inside components [Sha94].

Both architectural elements and the services that must be interchanged among them can be identified during this stage. These services are requested and/or provided by the architectural elements. Since a PRISMA interface is a set of services that is provided and/or requested by means of the ports of architectural elements, the identification of these services implies the identification of the interfaces. The ports of architectural elements and the interconnections among these ports can also be detected taking into account the identified interfaces.

The PRISMA methodology is put into practice using the PRISMA CASE, except for this 1st Stage that the identification of architectural elements and aspects is made by hand.

9.1.2. Identification of Crosscutting-Concerns

The concerns that crosscut the software system must be identified from the requirements specification in order to modularize them into reusable entities called aspects. In the case of the TeachMover, the crosscutting concerns that have been identified in the case study are the following:

- **Functional:** The purpose of the *TeachMover* software system is to move the robot. The robot has a motor to accurately perform movements by half-steps. A half-step is an angular advance that is produced by a stimulating impulse. In the case of the TeachMover, the movements can be requested using half-steps or inverse cinematics (moving to a specific point in space). This functionality, together with the gripper functionalities allows the robot to move objects from an initial position to a final one. The movements of the robot are ordered by an operator from a computer.
- **Safety:** Safety directives are necessary for monitoring the *TeachMover* movements in order to make sure that the movements are safe for the robot, the operator, and the environment that surrounds them.
- **Coordination:** The inner behaviour of joints (SUCs) and the movements in which more than one joint has to be moved must be coordinated. In addition, the requests of the operator and the performance of the movements must be synchronized.

9.2. 2ND STAGE: TYPE ARCHITECTURAL MODELLING

Once the interfaces, aspects, architectural elements and their ports have been identified, the skeleton of the architectural elements and the aspects can be defined. The analyst is then ready to start the type modelling process of the software architecture. This stage can be divided into four modelling steps: Interfaces, Aspects, Simple Architectural Elements, and Complex Architectural Elements (see Figure 80).

It is important to keep in mind that the enumeration of these steps is not a restrictive order. The enumeration simply indicates the dependencies between the different concepts that arise when the architectural model is being modelled. These dependencies are the following:

- To completely define a complex architectural element, the architectural elements that it consists of must have been previously defined
- To completely define an architectural element, the aspects that it imports and the interfaces that their ports use must have been previously defined
- To completely define an aspect that uses interfaces, the interfaces must have been previously defined

Even though the order of these steps can be different, it should be followed in order to completely define an architectural model. In other words, it does not mean that partial descriptions of the architectural elements, aspects or architectural models cannot be performed during the development process. The analyst can start the modelling process from either steps 1, 2, 3, or 4, obtaining partial solutions of the model, and can go backward or forward depending on his/her needs.

This 2nd Stage is developed using the PRISMA Type Modelling Tool. The STEPS 1 to 4 are developed using the PRISMA Type Modelling Tool, where all the reusable types (interfaces, aspects, simple architectural elements and complex architectural elements) are modelled in a graphical way by drawing and dropping the PRISMA modelling primitives.

9.2.1. STEP 1: Interfaces

Interfaces are specified in step 1 of the PRISMA architecture modelling stage. This is due to the fact that it is not necessary to previously define other elements of the model. Interfaces are stored in a PRISMA repository for reuse (see step 1, 2nd Stage, Figure 80).

```

Interface ISUC

    moveJoint(input NewHalfSteps: integer, input Speed: integer);
    cinematicsMoveJoint(input NewAngle: integer, input Speed: integer);
    stop();
    moveOk(output success: boolean);

End_Interface ISUC;

```

Figure 81. The ISUC interface

The specification of the interfaces identified in the first stage consists of describing the interface services and their signatures. The signature of a service specifies its name and parameters. The data type and the kind (input/output) of parameters are also declared. An example of an interface is shown in Figure 81.

Once an interface is specified is convenient to verify it in order to be sure that the specification is correct. In addition, it is recommended to verify all the interfaces the entire model once they have been defined. After the verification of an interface, it can be stored in the PRISMA repository to be reused by other architectural models.

9.2.2. STEP 2: Aspects

As it has been previously mentioned, a PRISMA aspect encapsulates a concern that crosscuts the architectural elements of software architectures. This aspect semantics is different from any of the architectural terms in the software architecture discipline that currently exist. The notion of aspect has an entity of its own, and aspects are first-order citizens of the PRISMA AOADL [Per06d].

Aspects are defined in step 2 of the second stage using the combination of two formalisms: a modal logic of actions [Sti92] and a dialect of the polyadic π -calculus [Mil93]. π -calculus is used to specify and formalize the processes of the PRISMA model, and the modal logic of actions is used to formalize how the execution of these processes affects the state of aspects. The kind of aspects and the number of each one depends on the software system. Aspects are reusable entities that define a specific behaviour of a crosscutting-concern and are, therefore, stored in a PRISMA repository (see step 2, 2nd stage, Figure 80). As a result, not only can aspects be used more than once in a software architecture description, they can also be reused in different software architectures. If there are going to use COTS in the model specification, the specification of the aspect that wraps the COTS must be done in this step. On the other hand, it is important to mention that before the storage of aspects in the PRISMA repository, it is recommended that each aspect will be verified.

There are aspects that specify the semantics of the services that are published by an interface (public services), there are aspects that specify the semantics of services that are not

published by any interface (private services), and there are aspects that specify the semantics of both, public services and private services. The aspects that specify the semantics of public services must be defined after the interface of their services has been defined. This is why interfaces are defined in step 1 and aspects are defined in the step 2 of the methodology. However, this does not constrain the specification order. Either the interface is defined before the aspect, or the services are initially defined as private services of the aspect and are then changed to publish services by means of an interface. Furthermore, when the needed interfaces are reused from the repository, step 1 is not necessary.

The aspects are defined by taking into account the crosscutting concerns identified in the first stage. The number of aspects for the same concern is decided by the analyst, taking into account criteria such as reusability and understanding. Depending on the analyst's criteria, he/she will define one aspect for a concern or several aspects for the same concern. For example, for the safety concern that crosscuts the software system, two safety aspects can be defined. Each of these aspects has different safety behaviour. This chapter is going to focus on the specification of a safety and a coordination aspect of the *TeachMover* case study.

9.2.2.1. *The safety aspect*

The concerns (safety, coordination, functionality, distribution, etc) that an aspect can specify in PRISMA is not constrained because the concerns vary depending on the system and the domain that is being modelled. A keyword is used to establish the concern of the aspect that is being specified. For example, in this case, the Safety word establishes that every property, service, or behaviour that is specified in the aspect is related to the safety concern (see Figure 82). This keyword is a property of a PRISMA aspect called concern, whose value is provided when a specific aspect is defined. There is no predefined list of keywords; the value is introduced by the analyst when he/she starts the aspect modelling task.

The concern of an aspect and its name are detailed at the head of the aspect. Several constant attributes are declared in the body in order to store the information of the minimum and maximum values that have to be taken into account to preserve the safety of the TeachMover (see section 1, Figure 82). *Begin* and *end* services start and end the execution of

an aspect, respectively (see sections 2 and 5, Figure 82.). Also, several services to preserve the safety of the system are defined in the *SMotion* aspect using dynamic logic. Some of them are specified in section 3 of Figure 82, but only the complete specification of the service *check* is presented to facilitate the understanding of the specification. This service ensures that the requested movement is safe by determining whether it is between the minimum and maximum degree (see section 4, Figure 82). In addition, transactional operations are defined to execute a set of services atomically (see section 6, Figure 82). Finally, the protocol defines the process of execution of the aspect services using a dialect of π -calculus (see section 7, Figure 82).

```

Safety Aspect SMotion
  Attributes
1  Constant
    minimum, maximum, minRoll, maxRoll, minPitch,
    maximumPitch: integer, NOT NULL; ... ..
  Services
2  begin( input InitialMinimum: integer,
    input InitialMaximum: integer, ... ..);
    Valuations
    [begin (InitialMinimum, InitialMaximum)]
    minimum := InitialMinimum,
    maximum := InitialMaximum;

3  in checkdistance(input NewX: integer, input NewY: integer,
    3    input NewZ: integer, output Secure: boolean); ... ..

    in check(input Degrees: integer, output Secure: boolean);
4  Valuations
    {(Degrees >= minimum) and (Degrees <= maximum)}
    [check(Degrees, Secure)]
    Secure := true;

    {(Degrees < minimum) or (Degrees > maximum)}
    [check(Degrees, Secure)]
    Secure := false;

    ... ..
5  end;
6  Transaction
    DANGEROUSCHECKING(input Degrees: integer,
    input CurrentSpeed: integer, output Secure: boolean): ... .. ;
    ... ..
7  Protocol
    SMOTION = begin.CHECKING;
    CHECKING = check (Degrees, Secure).CHECKING +
    checkDistance(NewX, NewY, NewZ, Secure). CHECKING
    + ... .. +
    DANGEROUSCHECKING(Degrees, Secure).CHECKING
    + end;
End_Aspect SMotion;

```

Figure 82. The safety aspect *SMotion*

9.2.2.2. The coordination aspect

The coordination aspect defines the interactions needed to coordinate the sending of movements to the robot and the robot's answers. These answers notify whether or not the movements have been satisfactorily performed.

```

Coordination Aspect CProcessSUC using IMotionJoint, IRead, ISUC
Attributes
Variable
1  halfSteps: integer, NOT NULL;
   tempHalfSteps: integer;

Derived
   angle: integer, Derivation FtransHalfstepsToAngle(halfSteps);

Services
begin (input InitialHalfSteps: integer);
   Valuations
   [begin (InitialHalfSteps)]
   halfSteps := InitialHalfSteps;

2  in/out movejoint(input NewHalfsteps: integer, input Speed: integer);
   Valuations
   [in movejoint (NewHalfsteps, Speed)]
   tempHalfSteps := NewHalfsteps;

3  in cinematicsmovejoint( input NewAngle: integer, input Speed:integer);
   Valuations
   [in cinematicsmovejoint(NewAngle, Speed)]
   tempHalfSteps := FtransAngleToHalfsteps(NewAngle);

4  in/out moveok(output Success: boolean);
   Valuations
   {Success=1}
   [in moveok(Success)]
   halfSteps:= halfSteps + tempHalSteps;

end;

5 Protocol
6 CPROCESSSUC = begin(InitialHalfStep).MOTION
7 MOTION =
   (ISUC.movejoint?(NewHalfsteps, Speed)
   → IMotionJoint.movejoint!(NewHalfsteps, Speed).ANSWER)
   +
   (ISUC.cinematicsmovejoint?(NewAngle, Speed)
   → IMotionJoint.movejoint!(FtransAngleToHalfSteps(NewAngle,
Speed).ANSWER)
   +
   end;
8 ANSWER= IRead.moveok?(Success) → ISUC.moveok!(Success).MOTION
End_Aspect CProcessSUC;

```

Figure 83. The coordination aspect *CProcessSUC*

In this case, the aspect uses the services of several interfaces. This is detailed at the head of the aspect (see Figure 83), that means, that these services are public. As a result, these interfaces have been previously defined. Since the request of a movement to the robot does not guarantee that it will be satisfactorily performed, the coordination aspect must coordinate the position of the joint that is synchronizing the movement request (see sections 2 and 3, Figure 83) with the movement notification of the robot (*moveOk service*, see section 4, Figure 83). At this point, the aspect changes the value of the robot position, i.e., the *halfstep* attribute (see section 1, Figure 83).

The protocol of the *CProcessSUC* aspect coordinates the requested movements and the notification movements of the robot. It is composed of three processes *CProcessSUC*, *MOTION*, and *ANSWER* (see section 5, Figure 83). The *CProcessSUC* process starts the execution of the aspect (*begin*), initializes the attributes that need a value (Not Null), and starts the *MOTION* process (see section 6, Figure 83). This process either receives a movement request or ends the aspect execution (see section 7, Figure 83). Finally, this *MOTION* process continues with the *ANSWER* process, in which the coordination process waits for the sensor's answer and notifies of the failure or success of the movement (see section 8, Figure 83).

9.2.3. STEP 3: Simple Architectural Elements

The definition of simple architectural elements is performed in step 3 of the second stage. The aspects that are defined in step 2 are used to completely define these architectural elements. An architectural element imports the aspects that define the concerns that it requires. For this reason, aspects must be defined before architectural elements.

The same aspect is imported by each architectural element that needs to take into account the same behaviour of this concern (crosscutting concerns). As a result, an aspect can be imported by one or more architectural elements (see steps 2 and 3, 2nd stage, Figure 80). It is important to note that the changes performed in an aspect also affect every architectural element that imports this aspect.

From the architectural element point of view, each architectural element is formed by a set of aspects. For example, the connector in Figure 80 is made up of a safety aspect and a

coordination aspect. It is important to emphasize that one of these aspect can be an integration aspects that wraps a COTS.

In PRISMA, weavings weave the different aspects that form an architectural element and they are encapsulated inside the architectural element. The temporal order of the weaving process is described by temporal operations called weaving operators. As a result, a PRISMA simple architectural element is specified by a set of aspects, the weavings of aspects, and a set of ports that have an associated interface. Ports represent the points of interaction of architectural elements. The architectural elements can be verified using the partial verification of a specific architectural element or all the architectural elements. The architectural elements that are defined in this step are types that are reusable by different software architectures because they are stored in the PRISMA repository. The storage of PRISMA architectural elements implies the storage of the aspects that they import. In addition, the reusability of the aspect can be due to the fact that it is reused in many architectural elements and also when the architectural element that imports the aspect is reused.

The simplest components that are found in the TeachMover system are actuators and sensors:

- *Actuator*: An actuator has two ports. To describe its functional behaviour, it imports the FActuator aspect that has been previously stored in the repository (see Figure 84).

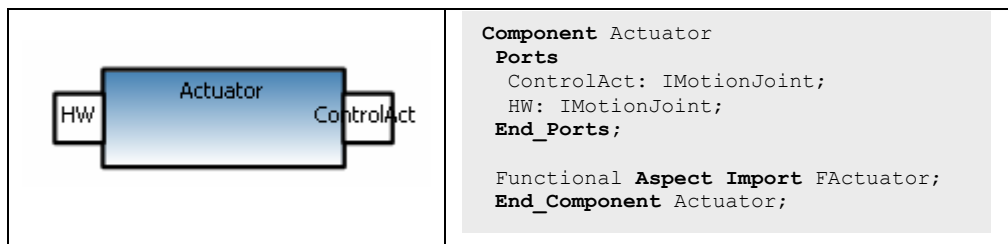
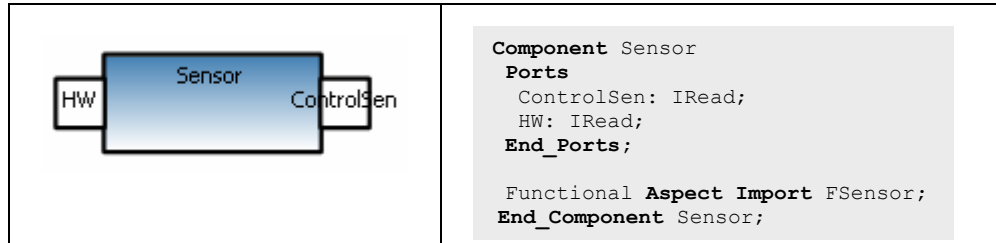


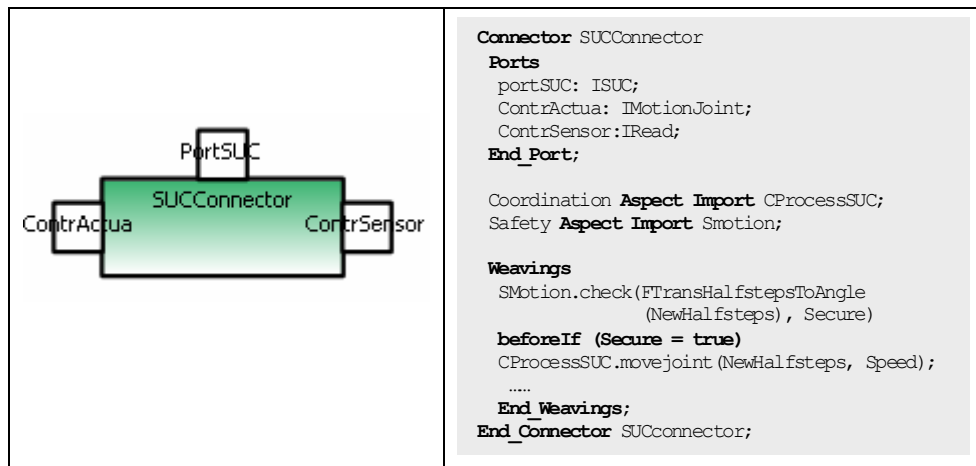
Figure 84. The component *Actuator*

- *Sensor*: A sensor has two ports. To describe its functional behaviour, it imports the FSensor aspect that has been previously stored in the repository (see Figure 85).

Figure 85. The component *Sensor*

In addition, the actuators and sensors for the tool and the wrist joints are defined as new architectural elements. They have the same functionality as the ones presented in Figure 84 and Figure 85. However, the signature of the services that they offer is different, that is, the number of parameters and their types is different.

Weavings are only specified when they are necessary to weave the execution of two services from different aspects. An example of weavings appears in the *SUCConnector* architectural element where the joint is moved only after the connector is sure that the movement is safe (see the weavings section of the *SUCConnector* in Figure 86).

Figure 86. The connector *SUCConnector*

The invocation of the *movejoint* service (the second service of the weaving) of the *CProcessSUC* aspect triggers the execution of a weaving. When a weaving is specified, the weaving operator is chosen from the point of view of the service that triggers the weaving; that

is, depending on whether the service needs the execution of a service before, after, or instead of it. As a result, the weaving of the *SUCConnector* (see Figure 86) means that the check service of the *SMotion* safety aspect will be executed before the *moveJoint* service of the *CProcessSUC* coordination aspect. The condition also establishes that the execution of the *moveJoint* service must only be performed if the *Secure* parameter of the *check* service returns true.

The connectors that coordinate the actuators and the sensors of the tool and the wrist joints are also defined as new architectural elements. *SUCconnector*, *WristSUCconnector*, and *ToolSUCConnector* are architectural elements stored in the PRISMA repository after their verification. These three architectural elements import the same coordination (*CProcessSUC*) and safety (*SMotion*) aspects (crosscutting concerns); however their behaviour is different because they have different weavings. This is a clear example of the reuse of aspects inside the same architectural model thanks to the fact that the definition of weavings outside aspects and inside architectural elements, and an example of how these aspects crosscut the behaviour of several architectural elements.

The *Actuator*, *Sensor*, and their connector *SUCConnector* are reusable architectural elements that can be used several times in the *TeachMover* architectural model or in another architecture specification of a bigger tele-operation domain such as the *EFTCoR* robot [Fer05], [EFT02]. In the case of the *TeachMover*, they have been specified in the simplest way. However, in the *EFTCoR*, they also have a distribution aspect in order to provide distributed behaviour to these components [Ali05a], [Ali03]. As a result, we have reused the *Actuator*, *Sensor* and *SUCConnector* architectural elements of the *TeachMover* in the architecture of the *EFTCoR*. We have modified them by adding a distribution aspect, a pair of weavings between the functional and distribution aspects of the *Actuator* and *Sensor*, and another pair of weavings between the coordination and distribution aspects of the *SUCConnector*. This is an example of reusability of defined components in other architectural models using the PRISMA repository. This reusability avoids having to start from scratch to build a new architectural model. The analyst can reuse the architectural elements in their original way or can reuse them and then

introduce the changes that the new software system requires, just as we did with the actuator and sensor components of the *EFTCoR*.

An actuator and a sensor must be coordinated through a connector in order to separate their computations from their interactions [Sha94]. The *SUCConnector* imports a coordination and a safety aspect in order to define its behaviour (see Figure 86).

9.2.4. STEP 4: Complex Architectural Elements

PRISMA complex components are called systems. Systems are defined in the 4 step of the second stage. A PRISMA system is a complex component that imports a set of connectors, components, and other systems that must be correctly attached. A system is defined by using two kinds of communication channels: attachments and bindings

To completely specify a system, the architectural elements that the system is composed of should be previously defined. In addition, the communication channels that permit the communication among them are defined. It is important to emphasize that the attachments are only defined if the system includes components and connectors that must be coordinated.

Systems are defined as patterns or architectural styles [Gar93] that can be reused in any software architecture whenever they are needed. For this reason they are stored in the PRISMA repository after their previous verification.

It is important to note that any changes that occur in aspects affect the architectural elements that import them and, consequently, affect the systems that import these architectural elements. Moreover, the changes that occur in architectural elements that are imported by a system also affect the system (see step 3, 2nd stage, Figure 80).

In the case of the TeachMover, there are several systems, at different levels of granularity. These systems are guided by the skeleton identified in the first stage (see Figure 3). The *SUC* (Simple Unit Controller) system is composed of an actuator, a sensor, and the connector that synchronizes them. The system specifies the architectural elements that it is composed of and the communication channels among them (see Figure 87).

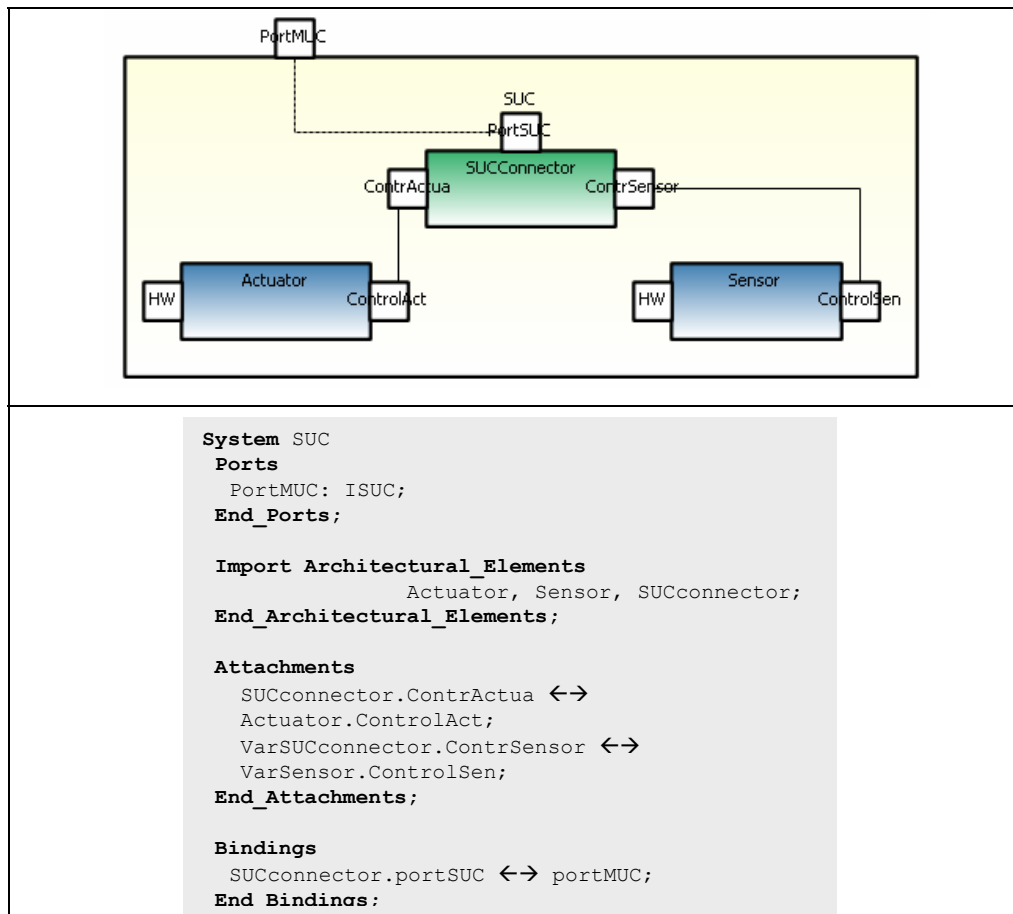


Figure 87. The system *SUC* (Simple Unit Controller)

The *SUC* system delegates the commands that it receives to the connector in order to perform the movements between the actuator and sensor in a synchronized way. For this reason, the *SUC* system and the *SUCConnector* have a binding between their *portMUC* and *portSUC* ports, respectively. In addition, two attachments haven been defined in order to establish the communication channels between the *Actuator* and the *SUCconnector*, and the *Sensor* and the *SUCconnector*, respectively.

A *ToolSUC* and a *WristSUC* are also defined with their corresponding actuators, sensors, connectors and their relationships. The *SUC* system is stored in the PRISMA repository so that it can be reused as an architectural pattern.

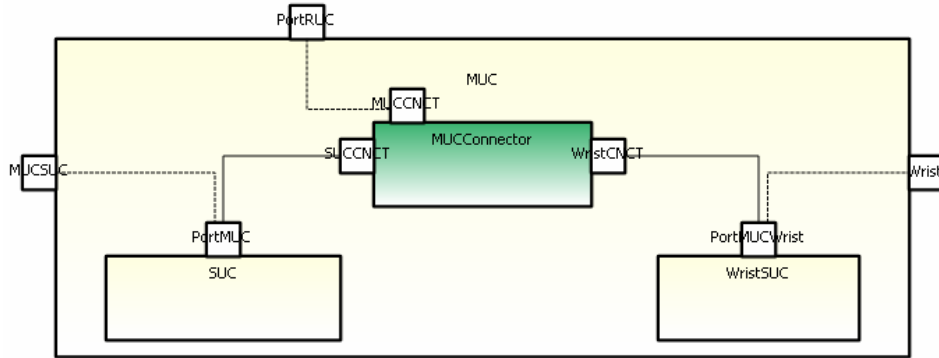


Figure 88. The system *MUC* (Mechanism Unit Controller)

The *MUC* (Mechanism Unit Controller) system is the third layer of granularity of the decomposition of the *TeachMover* system. It integrates a set of *SUC*s and a connector. The connector coordinates the *SUC*s in order to achieve a common goal. Specifically, the *MUC* of the *TeachMover* is composed of the generic *SUC*, the *SUC* of a wrist, and a connector that synchronizes them (see Figure 88). As a result, the *MUC* of the *TeachMover* is matched with the arm of the robot and specifies the behaviour required to accurately perform its movements. The *MUC* is also stored in the PRISMA repository.

The *RUC* (Robot Unit Controller) system coordinates every part of the robot and is composed of the *MUC* of the arm and the *SUC* of the tool, which are synchronized through a connector (see Figure 89). The *SUC* of the tool is not inside the *MUC* system in order to easily allow changes in the tool. For example, the tool of the *TeachMover* is a gripper, but it can be changed for a paintbrush, a hosepipe, etc., in order to perform other tasks. The *RUC* is matched with the *TeachMover*. As a result, we obtain a software architecture that is easy to evolve.

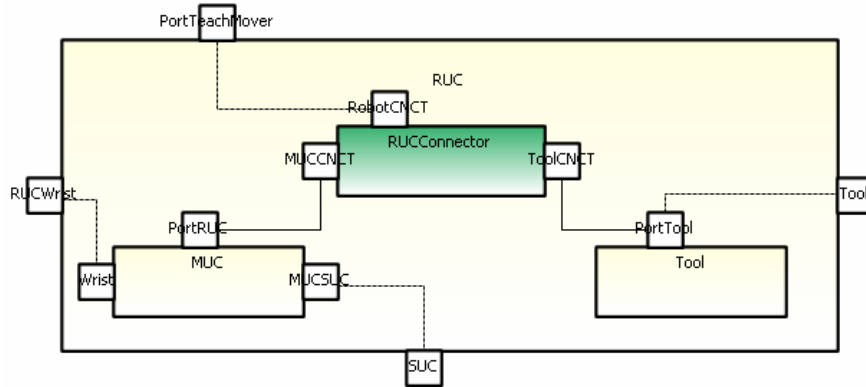


Figure 89. The system *RUC (Robot Unit Controller)*

Finally, the last layer of decomposition is composed of the operator, the robot (*RUC*), and the connector that coordinates them (see Figure 90). This last level provides us the most abstract view of the software architecture, which is called the *Architectural Model*. It is important to emphasize that since the architectural model does not define a system that encapsulates it, bindings are do not need to be defined.

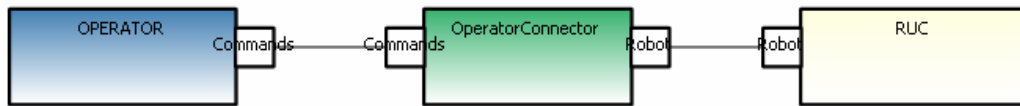


Figure 90. The architectural model of the *TeachMover*

9.3.3RD STAGE: TYPE CODE GENERATION

Once the interfaces, aspects, and simple and complex architectural elements have been completely specified, their code and specifications of PRISMA AOADL can be automatically generated. This code generation must be performed after the complete verification of the model and to check that there are no constraints that are not satisfied by the model (see 3rd stage, Figure 80).

This stage is performed by executing the code generation templates that the *PRISMA Type Modelling Tool* provides (see section 6.3.3). This generation is possible thanks to the code generation templates (model compiler), which isolate the specification from the source code

preserving their independence. Until now, PRISMA CASE generates PRISMA aspect-oriented C# code that is executable in .NET framework thanks to our PRISMANET middleware, which gives support to aspect execution over .NET technology [Per05b].

9.4.4TH STAGE: CONFIGURATION MODELLING

The architectural elements that have been defined in the previous stage and have been stored in the PRISMA repository are instantiated in this stage. In order to understand the trace of the approach, it is important to take into account that instances have all the properties and behaviours of their architectural elements, and as a consequence, instances have the properties and behaviours of the aspects that their architectural elements import. As a result, when an architectural element is instantiated, the aspects that it imports are also instantiated in order to have their specific state and behaviour.

This 4th Stage is developed using the PRISMA Configuration Modelling Tool, where the configuration of the architectural model of the software system is modelled in a graphical way by drawing and dropping the domain specific PRISMA modelling primitives (see section 6.3.4). The configuration of the initial architecture of a specific system is modelled by instantiating the types and the architectural model that has been defined in the previous stage.

A specific software architecture is defined by connecting a set of instances of components, systems, and connectors with each other (see *instance_of* relationships, Figure 80). For this reason, the instantiation of the architectural elements of a model and the definition of attachment and binding relationships among instances is necessary to obtain an executable architectural model.

To obtain the software architecture of the TeachMover, the *SUC* has been instantiated three times in order to obtain the *base*, *shoulder*, and *elbow* joints of the robot (see section 2.3.2). The gripper and the wrist are instances of the *ToolSUC* and *WristSUC*, respectively. As a result of these instantiations, the instantiation of the MUC generates the configuration that appears in Figure 91.

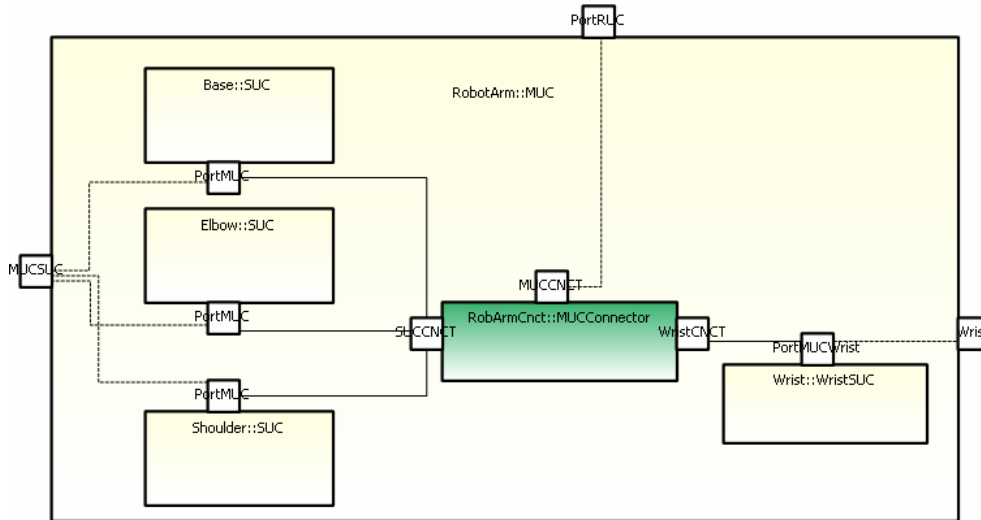


Figure 91. The configuration *MUC* for the *TeachMover*

The *RUC* and the architectural model of the *TeachMover* are instantiated generating one instance for each one of them. This instantiation constitutes the last step of the configuration stage. As a result of this step, the software architecture of the *TeachMover* is completely defined.

9.5TH STAGE: CONFIGURATION CODE GENERATION

Once the complete architectural model has been instantiated, their code and specifications of PRISMA AOADL can be automatically generated (see 5th stage, Figure 80). They are generated by executing the PRISMA Model Compiler from the PRISMA Configuration Modelling Tool (see section 6.3.5).

9.6.6TH STAGE: CODE EXECUTION

Next, the execution of the generated code joint the PRISMANET middleware can be launched from the PRISMA Configuration Modelling Tool (see 6th stage, Figure 80). Once the aspect-oriented software architecture is executed the user can interact with it using the PRISMA Generic GUI, which allows the user to execute services, query the value of attributes and

validate the correct behaviour of each of the architectural elements that compose the architecture (see section 6.3.5).

9.7. DISCUSSION

The TeachMover example illustrates that the PRISMA approach allows the development of aspect-oriented software architectures as if aspects and architectural elements were building blocks. You can work with them in different ways to obtain different results. This flexibility and facility for working is achieved thanks to the fact that aspects and components are independent entities that can be imported in different entities of the same software architecture or in different ones.

The introduction of aspects in a separate entity allows for the *TeachMover* software architecture to separate the safety concern in the *SMotion* aspect. This aspect is only defined once and it is imported by the *SUCconnector*, *WristSUCconnector*, *ToolSUCconnector*, *MUCconnector* and *RUCconnector* connectors by referencing this aspect. As a consequence, the aspect is used by the instances of these connectors. For example, the *TeachMover* instances of the *SUCconnector* are the *BaseSUCconnector*, *ShoulderSUCconnector* and *ElbowSUCconnector*. As a result of considering aspects to model the *TeachMover*, tangled code is avoided by separating coordination properties from safety properties. In addition, this separation of concerns can improve the maintenance process of the system due to the fact that the introduction of a change in the safety properties of the *TeachMover* only requires modifying the *SMotion* Aspect and, consequently, all the changes are propagated to the connectors and their instances.

The PRISMA repository also promotes the reuse of architectural elements and aspects in different software architectures. They can be used as they are stored in the repository or can be adapted to the features of the new architecture that is being modelled. For example, most of the architectural elements of the *TeachMover* were reused in the *EFTCoR* software architecture. The specific properties of the *EFTCoR* were easily adapted; for example, some of the robot pieces of the *EFTCoR* are distributed. This property was introduced by defining a distribution aspect and importing it from the reused architectural elements of the *TeachMover* that

represent these pieces [Ali05a], [Ali03]. It must be noted that reuse is limited for software architectures of different domains, but the PRISMA repository could be a great contribution for reuse in product families such as the tele-operation family. In addition, a large repository with good searching mechanisms could provide excellent support for the development and maintenance of software.

The PRISMA MDD methodology is supported by the PRISMA CASE framework for the construction of aspect-oriented software architectures that are independent of specific technologies. The first stage of the PRISMA methodology is the identification of the aspects and the architectural elements of a software architecture. Currently, this identification is performed in an intuitive way from the requirements document of the system. However, we are working in this stage in order to propose guidance for the user in this identification [Nav03], [Nav04a], [Nav04b], [Nav04c].

The modelling stage of the PRISMA methodology is based on the PRISMA metamodel. It has been defined by integrating aspects and software architectures. In this way, software architectures can be constructed gaining the advantages of two different paradigms: the aspect-oriented paradigm (AOP) and the component-based paradigm. One of these advantages is the reusability of software at different levels of granularity. As an example, the reusability level of the SUC system (see Figure 87) is illustrated in the following:

- *Analysis of the reusability of the SUC system*

The reusability of the SUC is analyzed by taking into account the system, its architectural elements and the aspects that these architectural elements import. In addition, the reusability is going to be analyzed at the type level and at the configuration level.

Figure 92 shows the number of times that an aspect is reused by different architectural elements and the names of the architectural elements that import the aspect. The aspects are enumerated in the first columns of the two tables. These aspects are the ones that have been used for defining the SUC system. The first table details the reusability at the type level, and the second table details the reusability at the configuration level, respectively. From these tables, it is possible to conclude that: the aspect *SMotion* is defined once and it is reused by five architectural elements at the type level and by seven architectural elements at the configuration

level. The aspects *FActuator*, *FSensor* and *CProcessSUC* have been defined once and they are reused by one architectural element at the type level and three architectural elements at the configuration level.

Reusability of Aspects (Type Level)		ARCHITECTURAL ELEMENTS	
		Number	Name
ASPECTS	SMotion	5	SUCConnector, WristSUCConnector, ToolSUCConnector, MUCCConnector, RUCConnector
	CProcessSUC	1	SUCConnector
	FActuator	1	Actuator
	FSensor	1	Sensor

Reusability of Aspects (Configuration Level)		ARCHITECTURAL ELEMENT INSTANCES	
		Number	Name
ASPECTS	SMotion	7	BaseCnct, ElbowCnct, ShoulderCnct, WristCnct, ToolCnct, RobotArmCnct, RobotCnct
	CProcessSUC	3	BaseCnct, ElbowCnct, ShoulderCnct
	FActuator	3	BaseAct, ElbowAct, ShoulderAct
	FSensor	3	BaseSen, ElbowSen, ShoulderSen

Figure 92. Reusability of aspects

On the other hand, Figure 93 shows the number of times that an architectural element type is reused at the configuration level of the architecture by means of its instantiation.

Reusability of Architectural Elements		ARCHITECTURAL ELEMENT INSTANCES	
		Number	Name
ARCHITECTURAL ELEMENT TYPES	SUC	3	BASE, ELBOW, SHOULDER
	SUCConnector	3	BaseCnct, ElbowCnct, ShoulderCnct
	Actuator	3	BaseAct, ElbowAct, ShoulderAct
	Sensor	3	BaseSen, ElbowSen, ShoulderSen

Figure 93. Resusability of architectural elements

In addition to the *SUC* rates of reusability inside the *TeachMover* software architecture, the reusability of aspects and architectural element types by other software architectures, such as the *EFTCoR* software architecture, must be taken into account. For example, in the case of the *EFTCoR* it has been imported every aspect and architectural elements of the *TeachMover* without having to start the modelling process from scratch. Some of the reused aspects have

been modified introducing new properties and others had not to be modified. Some of the reused architectural elements have been modified adding new aspects and/or weavings and others have been preserved in their original version.

Maintainability and evolution are also gained from integrating the aspect-oriented paradigm (AOP) and the component-based paradigm. This is due to the fact that aspects and architectural elements facilitate the task of locating where the changes must occur. For example, in the case of the *TeachMover*, safety properties are located in only one place, in the aspect *SMotion*. As a result, if a change in the safety properties occurs, the safety aspect is only modified instead of changing the five architectural elements that import it at the type level. But not only the number of changes is reduced, but also the time invested in locating where the change must be introduced is improved. In our case, the safety properties of the aspect have to be only searched; whereas in a non-aspect-oriented approach, the architectural elements with tangled concerns have to be searched and the location of the safety properties is more difficult. Finally, another important advantage is the reduction of code. As it was demonstrated by [Kiz97], the lines of code are reduced. Our proposal avoids the replication of code. For example, the lines of the safety aspect *SMotion* were avoided to be repeated five times at the type level.

The DSL tools development environment has been chosen in order to provide a framework for the PRISMA approach. DSL tools allow us to define PRISMA metamodel, associate a graphical notation to each metamodel concept and its instantiation, and implement the C# code generation templates of PRISMA. The PRISMA framework provides mechanisms to optimize the programming, reusability, and modularity of code. Developers use the graphical notation of the PRISMA AOADL to build their software architectures using the methodology presented in this thesis. After, the models are verified they can generate automatically the code. The *TeachMover* software architecture has been modelled, its code has been generated and it has been executed in the .NET platform. As a result, PRISMA has made it possible to move a robot using an aspect-oriented architecture that is executed in .NET technology (see demonstrations in [PRI07]).

9.8. CONCLUSIONS

This chapter defines and explains the MDD PRISMA methodology. This methodology allows the complete development of aspect-oriented architectural models following the MDD paradigm. This methodology is divided into six stages and allows the development and maintenance of software systems in a simple and flexible way.

The precise semantics of the PRISMA language gives us the chance to include the verification of model and its compile to code as parts of this methodology. As a result of this methodology application executable systems on the PRISMANET middleware [Per05b] are obtained.

The ideas developed in the case study have been applied to specific products of the EFTCoR and *TeachMover*. The benefits of applying PRISMA to the case study are mainly: 1) the definition of reusable concerns that crosscut the software architecture called aspects and the storage in a repository 2) the definition of reusable architectural elements by reusing aspects, 3) the definition of complex architectural elements reusing other architectural elements, 5) the verification of models supports, 5) the use of COTS support, 6) building the formal software architecture of the system using a friendly graphical notation independently of technology, 7) the reduction of the development time as code is automatically generated from the graphical notation to a specific technology.

It is important to emphasize that this chapter has applied a software engineering approach to the development of a robotic system. In this way, this development has taken advantage of the good properties provided by this software engineering approach, especially reuse of components, maintainability, evolution, etc.

The work presented in this chapter has been published in the following publication:

- **Jennifer Pérez**, Nour Ali, Jose A. Carsí, Isidro Ramos, Bárbara Álvarez, Pedro Sánchez, Juan A. Pastor, *Integrating Aspects in Software Architectures: PRISMA Applied to Robotic Tele-operated Systems*, Journal of Information and Software Technology, Elsevier, (**JCR 2006: 0.726**) (accepted, to be published)

CHAPTER 10

CONCLUSIONS AND FURTHER RESEARCH

This chapter presents and analyzes the main contributions of this thesis of master. It also presents future work that can be done to continue this research and to extend the results that have already been obtained.

10.1. CONCLUSIONS

The complexity of current software systems and the fact that their non-functional requirements have become very relevant to the user are challenges to be faced in all software development. Software Architectures and AOSD have emerged to overcome these needs. In order to take advantage of Software architectures and AOSD, several approaches have emerged to combine both approaches providing all their advantages together. However, these approaches usually extend architectural models without connectors and mainly follow an asymmetric model. They are only focused on a single specific purpose: analysis, evolution or development of software architectures without attempting to provide complete development and maintenance support following the MDD process. Furthermore, these approaches always introduce the notion of aspect by using original architectural concepts, despite the fact that they do not provide the suitable semantics for aspects. This thesis presents the MDD support for the PRISMA approach, a new software development approach that integrates Software Architectures and AOSD to fulfil these needs.

The PRISMA MDD provides complete support during the development and maintenance processes of software following the MDD paradigm. The PRISMA MDD approach is presented as an important advance in the combination of the aspect-oriented paradigm and software architectures. The PRISMA approach integrates an aspect-oriented symmetric model with an architectural model that has the notion of connector. The MDD advantages that this combination of software architectures and AOSD provides to the definition of coordination models have been presented. In addition, a detailed analysis about how to improve coordination models from this combination has been done.

In PRISMA, aspects and software architectures are smoothly integrated with a clear semantics, which has been formalized using a Modal Logic of Actions and π -calculus. In addition, the PRISMA metamodel allows the definition the PRISMA models and to establish its properties in a precise way. This metamodel and formalization have facilitated the automation and maintenance tasks of PRISMA software architectures since the main goal of this thesis was to define the MDD support of the PRISMA approach.

Another important characteristic of PRISMA that has facilitated the definition of the PRISMA MDD process is the PRISMA Aspect-Oriented Architecture Description Language (AOADL) [Per06d], which supports the PRISMA model [Per05a]. This AOADL allows the definition of PRISMA aspect-oriented software architectures, not only providing components and connectors as first-order citizens of the language, but also provides aspects and interfaces. The structure, design and maintainability of architectures specified in the PRISMA AOADL are improved by defining and reusing entities at different levels of granularity (interfaces, aspects, components, connectors and systems). This improvement is possible since (1) AOADL provides interfaces and aspects, (2) weavings are defined outside aspects, (3) aspects are defined independently of architectural elements, and (4) architectural elements are defined without being aware of the architectural elements that are connected to them. As a result, an interface can be used by several aspects, an aspect can be used by several architectural elements, and an architectural element can be used by several software architectures. In addition, the precise semantics of the PRISMA AOADL and its independence of technology

provide the opportunity to validate PRISMA software architecture properties and to compile models for different programming languages and platforms.

In order to improve this reusability and the development time of the PRISMA MDD approach. This thesis has defined a novel way of integrating COTS into software architectures. This integration is feasible using aspects as COTS wrappers. Specifically, a new kind of aspect called `integration_aspect` has been defined to specify the COTS that it wraps and to specify the integration process with the rest of software architecture. In the same way as other aspects, integration aspects must be imported by an architectural element in order to publish their services through ports that enable their communication with other architectural elements. Moreover, it is presented how the use of COTS is supported by the PRISMA methodology thanks to the facilities that the PRISMA CASE provides. This is an important characteristic since COTS are widely used, and any development approach that needs to reduce development time must provide it.

This independency and reusability properties of PRISMA elements has facilitated the definition of a flexible verification process. This thesis has defined a complete verification process to be supported by the PRISMA MDD approach. This verification process has classified PRISMA constraints into `hardconstraints` and `weakconstraints`. From this classification the PRISMA verification process takes a step forward and the partial and incremental verification has been proposed instead of only taking into account the complete verification of the model.

Since the use of a formal language is a hindrance for many users, PRISMA provides a graphical AOADL to describe formal software architectures using a friendly graphical notation. Finally, it is important to mention that the PRISMA AOADL has great expressive power to specify more features and requirements related to the software system by means of aspects. Therefore, PRISMA AOADL is not only able to specify simple architectural systems for academic projects such as pipelines, filters, blackboards, etc, but it is also able to completely specify complex software systems. In addition, this expressive power permits to have enough information for the code generation task of the PRISMA model compiler.

The MDD approach proposed in this thesis is based on the models hierarchy of PRIMA and the code generation techniques. 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, and to define configuration models from type models. In addition, PRISMA approach has created a set of transformation patterns to transform PRISMA models into its AOADL specifications and into C# code. 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.

PRISMA CASE is a framework that provides complete support for the PRISMA MDD approach. It is composed of a set of tools that is suitably integrated to provide a unique framework that gives support for the user throughout the software life cycle. This integration also provides top-down traceability during the different stages of the software life cycle and facilitates the maintenance of the developed software products. This set of tools includes the *PRISMA Type Modelling Tool* with its code generation patterns, the *PRISMA Configuration Modelling Tool* with its code generation patterns, the generic Graphical User Interface for PRISMA applications, and the middleware PRISMANET. The *PRISMA Types and Configuration Modelling Tools* give support for the development of PRISMA software architectures following the MDD approach and using the PRISMA AOADL in a graphical way. As a result, PRISMA offers mechanisms to develop software architectures in a more intuitive and friendly way and mechanisms to verify their models. In addition, the code generation patterns that PRISMA modelling tools offer allow automatically generate executable C# code on PRISMANET from the specified graphical models. Thus, PRISMA CASE deals with the traceability between software architectures and implementation and reduces the time and cost invested in the development and maintenance processes. PRISMA CASE provides a generic Graphical User Interface to execute software architectures. This is an important advantage because it is a simple way of validating that software architectures provide the behaviour expected by the user without having to develop a customized graphical user

interface. All the tools and mechanisms that PRISMA CASE provides make PRISMA a well-supported approach for developing aspect-oriented software architectures following the MDD approach.

Just as important is the methodology that has been defined to guide the user during the MDD process of PRISMA software architectures. This methodology is supported by the PRISMA CASE and consists of six stages that define how to specify software architectures from scratch or how to reuse software by importing PRISMA architectural elements and aspects and COTS, and how to obtain the final code from these specifications. These stages are the following: detection of architectural elements and aspects, type architectural modelling, type code generation, configuration modelling, configuration code generation and execution. These six stages are applied by the analyst of the software system in an iterative and an incremental way depending on his/her needs. As a result this methodology allows the development and maintenance of software systems in a simple and flexible way.

All the contributions of this thesis of master have been demonstrated using the *TeachMover* robot case study. The reuse capabilities of the PRISMA model have been presented by means of the *TeachMover* case study. The *TeachMover* architecture has also helped to present the capabilities of the PRISMA modelling tool and the verification process. The case study has been totally specified and its code has been generated and executed by PRISMACASE using COTS and without COTS.

The contributions of this thesis of master are based in the previous PRISMA contributions that have been published in the following publications:

JOURNALS

- Nour Ali, **Jennifer Pérez**, Cristóbal Costa, Isidro Ramos, Jose Ángel Carsí, *Distributed Replication in Aspect-Oriented Software Architectures using Ambients*, Journal IEEE América Latina, Vol. 5, Issue 4, July 2007. (In Spanish)

INTERNATIONAL CONFERENCES

- **Jennifer Pérez**, Nour Ali, Jose Ángel Carsí, Isidro Ramos, *Designing Software Architectures with an Aspect-Oriented Architecture Description Language*, 9th Symposium on the Component Based Software Engineering (CBSE), Springer Verlag LNCS 4063 ,pp. 123-138, ISSN: 0302-9743, ISBN: 3-540-35628-2, Vasteras, Sweden, June 29th-July 1st, 2006.
- Nour Ali, **Jennifer Pérez**, Cristóbal Costa, Isidro Ramos, José Ángel Carsí, *Mobile Ambients in Aspect-Oriented Software Architectures*, IFIP Working Conference on Software Engineering Techniques: Design for Quality- SET 2006, Springer, Volume 227 pp. 37-48, ISSN: 1571-5736, ISBN: 0-387-39387-0, Warsaw, Poland, October, 17-20, 2006.
- **Jennifer Pérez**, Elena Navarro, Patricio Letelier, Isidro Ramos, *A Modelling Proposal for Aspect-Oriented Software Architectures*, 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), IEEE Computer Society , pp.32-41, ISBN: 0-7695-2546-6, Potsdam, Germany (Berlin metropolitan area), March 27th-30th, 2006.
- **Jennifer Pérez**, Elena Navarro, Patricio Letelier, Isidro Ramos, *Graphical Modelling for Aspect Oriented SA*, 21st Annual ACM Symposium on Applied Computing, ACM, pp. 1597-1598, ISBN: 1-59593-108-2, Dijon, France, April 23 -27, 2006. (short paper)
- **Jennifer Pérez**, Manuel Llavador, Jose A. Carsí, Jose H. Canós, Isidro Ramos, *Coordination in Software Architectures: an Aspect-Oriented Approach*, Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society Press, pp. 219-220, ISBN: 0-7695-2548-2, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper)
- Nour Ali, **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *Integrating Ambient Calculus in Mobile Aspect-Oriented Software Architectures*, Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society Press, pp. 233-234, ISBN: 0-7695-2548-2, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper)

- **Jennifer Pérez**, Nour Ali, Cristobal Costa, José Á. Carsí, Isidro Ramos, *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*, 3rd International Conference on .NET Technologies, pp. 97-108, Pilsen, Czech Republic, May-June 2005.
- Nour Ali, **Jennifer Pérez**, Isidro Ramos, *Aspect High Level Specification of Distributed and Mobile Information Systems*, Second International Symposium on Innovation in Information & Communication Technology ISSICT, pp. 14, Amman, Jordania, 21-22, April, 2004.
- Nour Ali, **Jennifer Pérez** Isidro Ramos, Jose A. Carsí , *Aspect Reusability in Software Architectures*, 8th International conference of Software Reuse (ICSR), July, 2004 (poster)
- Elena Navarro, Isidro Ramos, **Jennifer Pérez**, *Goals Model-Driving Software Architecture*, 2nd International Conference on Software Engineering Research, Management and Applications (SERA), pp. 205-212, ISBN:0-9700776-9-6, May 5-8, 2004, Los Angeles, CA, USA.
- Nour Hussein, Josep Silva, Javier Jaen, Isidro Ramos, Jose Ángel Carsí ,**Jennifer Pérez** , *Mobility and Replicability Patterns in Aspect-Oriented Component- Based Software Architectures*, 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS), ACTA Press, ISBN: 0-88986-392-X, ISSN: 1027-2658, pp. 820-826, Marina del Rey, California, USA, 3-5, November 2003,
- **Jennifer Pérez** , Isidro Ramos , Javier Jaén, Patricio Letelier, Elena Navarro , *PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures*, 3rd IEEE International Conference on Quality Software (QSIC 2003), IEEE Computer Society Press, pp.59-66, ISBN: 0-7695-2015-4, Dallas, Texas, USA, November 6 - 7, 2003.
- Elena Navarro, Isidro Ramos, **Jennifer Pérez**, *Software Requirements for Architected Systems*, 11th IEEE International Requirements Engineering Conference (RE'03), IEEE Computer Society Press, pp. 365-366, ISSN: 1090-705X, ISBN: 0-7695-1980-6, Monterey, California, 8-12 September 2003 (Poster)

INTERNATIONAL WORKSHOPS

- **Jennifer Pérez**, Nour Ali, Jose A. Carsí, Isidro Ramos, *Dynamic Evolution in Aspect-Oriented Architectural Models*, Second European Workshop on Software Architecture, Springer LNCS 3527, pp.59-16, ISSN: 0302-9743, ISBN: 3-540-26275-X , Pisa, Italy, June 2005.
- **Jennifer Pérez** , Isidro Ramos , Javier Jaén, Patricio Letelier, *PRISMA: Development of Software Architectures with an Aspect Oriented, Reflexive and Dynamic Approach*, Dagstuhl Seminar N° 03081, Report N° 36 "Objects, Agents and Features", Copyright (c) IBFI gem. GmbH, Schloss Dagstuhl, D-66687 Wadern, Germany . Eds.H.-D. Ehrlich (Univ. Braunschweig, D), J.-J. Meyer (Utrecht, NL), M. Ryan (Univ. of Birmingham, GB), pp. 16, Germany, January, 2003.
- Jorge Ferrer, Ángeles Lorenzo, Isidro Ramos, José Ángel Carsí, **Jennifer Pérez**, *Modeling Dynamic Aspects in Architectures and Multiagent Systems*, Logic Programming and Software Engineering (CLPSE), pp. 1-13, Copenhagen, Denmark, affiliated with ICLP, july 2002.

NATIONAL CONFERENCES

- Nour Ali, **Jennifer Pérez**, Cristóbal Costa, Isidro Ramos, José Ángel Carsí, *Distributed Replication in Aspect-Oriented Software Architectures using Ambients*, XI Conference on Software Engineering and Databases (JISBD), pp. 379-388, ISBN: 84-95999-99-4, Sitges, Barcelona, October 2006. (In Spanish)
- Cristóbal Costa, **Jennifer Pérez**, Nour Ali, Jose Angel Carsí, Isidro Ramos, *PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures*, X Conference on Software Engineering and Databases (JISBD), pp. 27-34, ISBN: 84-9732-434-X, Granada, September, 2005. (In Spanish)

- **Jennifer Pérez**, Nour H. Ali, Isidro Ramos, Juan A. Pastor, Pedro Sánchez, Bárbara Álvarez, *Development of a Tele-Operation System using the PRISMA Approach*, VIII Conference on Software Engineering and Databases (JISBD), pp. 411-420, ISBN: 84-688-3836-5, Alicante, November, 2003. (In Spanish)
- **Jennifer Pérez**, Isidro Ramos, Ángeles Lorenzo, Patricio Letelier, Javier Jaén, *PRISMA: OASIS Platform for Architectural Models*, VII Conference on Software Engineering and Databases (JISBD), pp. 349-360, ISBN: 84-688-0206-9, El Escorial (Madrid), November, 2002. (In Spanish)

NATIONAL WORKSHOPS

- Cristóbal Costa, **Jennifer Pérez**, Jose Angel Carsí, *Towards the Dynamic Configuration of Aspect-Oriented Software Architectures*, IV Workshop on Aspect-Oriented Software Development (DSOA), XI Conference on Software Engineering and Databases (JISBD), Technical Report TR-24/06 of the Polytechnic School of the University of Extremadura, pp.35-40, Sitges, Barcelona, Octubre, 2006. (In Spanish)
- Rafael Cabedo, **Jennifer Pérez**, Nour Ali, Isidro Ramos, Jose A. Carsí, *Aspect-Oriented C# Implementation of a Tele-Operated Robotic System*, III Workshop on Aspect-Oriented Software Development (DSOA), X Conference on Software Engineering and Databases (JISBD), pp. 53-59, ISBN: 84-7723-670-4, Granada, September, 2005. (In Spanish)
- **Jennifer Pérez**, Nour H. Ali, Isidro Ramos, Jose A. Carsí, *PRISMA: Aspect-Oriented and Component-Based Software Architectures*, Workshop on Aspect-Oriented Software Development (DSOA), Conference on Software Engineering and Databases (JISBD), Technical Report TR-20/2003 of the Polytechnic School of the University of Extremadura, pp. 27-36, Alicante, November, 2003. (In Spanish)
- M^a Eugenia, Nour Ali, **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *DIAGMED: An Architectural model for a Medical Diagnosis*, IV workshop DYNAMICA – DYNAMIC and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 1-7, Archena, Murcia, November, 2005. (In Spanish)

- Rafael Cabedo, **Jennifer Pérez**, Jose A. Carsí, Isidro Ramos, *Generation and Modelling of PRISMA Architecture using DSL Tools*, IV Workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp.79-86, Archena, Murcia, November, 2005. (In Spanish)
- Nour Ali, **Jennifer Pérez**, Jose A. Carsí, Isidro Ramos, *Mobility of Objects in the PRISMA Approach*, II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, pp. 111-118, Almagro, Ciudad Real, April, 2005.
- **Jennifer Pérez**, Rafael Cabedo, Pedro Sánchez, Jose A. Carsí, Juan A. Pastor, Isidro Ramos, Bárbara Álvarez, *PRISMA Architecture of the Case Study: an Arm Robot*, II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November 2004. (In Spanish)
- Nour Ali **Jennifer Pérez**, Cristobal Costa, Jose A. Carsí, Isidro Ramos, *Implementation of the PRISMA Model in the .Net Platform*, II workshop DYNAMICA – DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, Conference on Software Engineering and Databases (JISBD), pp. 119-127, Málaga, November, 2004.
- Nour H. Ali, Josep F. Silva, Javier Jaen, Isidro Ramos, Jose Á. Carsí, **Jennifer Pérez**, *Distribution Patterns in Aspect-Oriented Component-Based Software Architectures*, IV Workshop Distributed Objects, Languages, Methods and Environments (DOLMEN), pp.74-80, Alicante, November, 2003.

TECHNICAL REPORTS

- Rafael Cabedo, **Jennifer Pérez**, Isidro Ramos, *The application of the PRISMA Architecture Description Language to an Industrial Robotic System*, Technical Report, DSIC-II/11/05, pp.180, Polytechnic University of Valencia, September 2005. (In Spanish)

- Cristobal Costa, **Jennifer Pérez**, Jose Ángel Carsí, *Study and Implementation of an Aspect-Oriented Component-Based Model in .NET technology*, Technical Report, DSIC-II/12/05, pp. 198, Polytechnic University of Valencia, September, 2005. (In Spanish)
- **Jennifer Pérez**, Nour Ali , Jose A. Carsí, Isidro Ramos, *PRISMA Architecture of the Robot 4U4 Case Study*, Technical Report DSIC-II/13/04, pp. 72, Polytechnic University of Valencia, 2004. (In Spanish)
- **Jennifer Pérez**, Isidro Ramos, *OASIS as a Formal Support for the Dynamic, Distributed and Evolutive Hypermedia Models*, Technical Report DSIC-II/22/03, pp. 144, Polytechnic University of Valencia, October 2003. (In Spanish)
- **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *A Compiler to Automatically Generate the Metalevel of Specifications using Properties of the Base Level*, Technical Report, DSIC-II/23/03, pp. 107, Polytechnic University of Valencia, October, 2003.(In Spanish)

In addition to be well supported the contributions of this thesis of master by the previous PRISMA publications. The contributions of this thesis that have been published or submitted in the following publications:

JOURNALS

- **Jennifer Pérez**, Jose A. Carsí, Isidro Ramos, *Model-Driven Development of Aspect-Oriented Software Architectures*, The Computer Journal, Oxford Journal, (**JCR 2006: 0.593**) (submitted)
- **Jennifer Pérez**, Nour Ali, Jose A. Carsí, Isidro Ramos, Bárbara Álvarez, Pedro Sánchez, Juan A. Pastor, *Integrating Aspects in Software Architectures: PRISMA Applied to Robotic Tele-operated Systems*, Journal of Information and Software Technology, Elsevier, (**JCR 2006: 0.726**) (accepted, to be published)

INTERNATIONAL CONFERENCES

- **Jennifer Pérez**, Isidro Ramos, Jose A. Carsí, *Taking Advantage of COTS for Developing Aspect-Oriented Software Architectures*, Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society, Vancouver, BC, Canada, 18 – 21 February 2008. (submitted)

INTERNATIONAL WORKSHOPS

- **Jennifer Pérez**, Carlos E. Cuesta, *Aspect-Oriented Connectors for Coordination*, International Workshop on Synthesis and Analysis of Component Connectors (SYANCO 2007), Joint to The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC-FSE, ACM Digital Library, Dubroknik, Croacia, September 3-4, 2007.

NATIONAL CONFERENCES

- **Jennifer Pérez**, Cristóbal Costa, Jose A. Carsí, Isidro Ramos, *Verification of Aspect-Oriented Architectural Models*, XII Conference on Software Engineering and Databases (JISBD), Zaragoza, Spain, 12-14 September. (In Spanish)
- **Jennifer Pérez**, Cristóbal Costa, Jose A. Carsí, Isidro Ramos, *PRISMA CASE*, XII Conference on Software Engineering and Databases (JISBD), Zaragoza, Spain, 12-14 September. (Demonstration, In Spanish)

10.2. FURTHER RESEARCH

The PRISMA MDD approach opens a perfect setting for further research. All the parts that the PRISMA MDD approach is composed of can be extended in order to face new challenges.

PRISMA has been applied to both: the tele-operation domain and the electronic bank domain. However, other domains can have other specific properties that are not taken into account in PRISMA. As a result, the application of the PRISMA model to other domains can assist us in defining new aspects that can introduce new properties of modelling and differences in aspect specifications. In fact, PRISMA only supports the definition of software architectures that are locally executed, despite the fact that most software architectures have a distributed nature. For this reason, we are currently working on introducing distribution and mobility properties in PRISMA using aspects [Ali05a], [Ali03], [Ali06], [Ali05b]. In addition, until now, PRISMA has been applied to software architectures that do not require persistence. However, information systems usually store their information in secondary memory. As a result, persistence is another important property that the model should support. There are other concepts from software architectures that PRISMA does not provide such as views and architectural patterns. In the long term, these concepts should also be defined in PRISMA.

The PRISMA model extensions imply modifications in the PRISMA AOADL at its different levels of abstraction (types and configuration) and at its different kinds of representation (textual and graphical). PRISMA AOADL supports cardinality constraints to define systems, but it should be extended to support other kinds of constraints as well.

Since PRISMA does not yet provide model checking mechanisms to check the properties of its architectural specifications such as reachability, deadlock detection and liveness, these model checking techniques should also be applied to PRISMA. With regard to the verification process associated to the PRISMA MDD process, the verification of the configuration models is not supported. As a result, it is necessary to provide the needed verification mechanisms for configuration models and to define the complete verification process.

Future work will exploit the results of the coordination model formalization to show the effects of combining several complex aspects, and will consider also the combination with the influence of assertions in the Modal Logic of Actions provided in architectural elements, as well as the possibility of extending this to a temporal logic such as the modal μ -calculus, which has already been made for recent work in PiLar [Cue04], [Cue02]. Also, a detailed comparison with the formalization and capabilities of some other π -calculus-based ADLs, such as Leda

[Can01], [Can00], PiLar [Cue04], [Cue02] or π -ADL [Oqu04a] will be carried out, studying the extent in which our results can be provided as extensions to non-symmetric, non-aspect-oriented existing ADLs.

Another important property of software systems is the continuous evolution that they undergo. Development frameworks must provide mechanisms to support evolution and to facilitate the software maintenance. As a result, PRISMA must be able to support the evolution of aspect-oriented software architectures. The division of the PRISMA architecture specifications into two levels of abstraction opens the opportunity to distinguish between the evolution of types and the evolution of a specific architecture. Despite the fact that the PRISMA evolution services have been identified and included in the PRISMA metamodel to modify software architectures, this only permits its modification at modelling time. However, since there are a lot of software systems that cannot stop their execution to be modified, run-time evolution must be provided. This run-time evolution is usually called dynamic configuration. Therefore, we are currently working on defining mechanisms to dynamically execute evolution services at run-time. Over the long term, our work with regard to software evolution will be related to the data evolution problem of software architectures, where we will apply our previous experience on data migration and data evolution of object-oriented conceptual schemas [Per02a], [Per02b], [Per02c].

For the application of PRISMANET, there are a lot of lacks that must be dealt with in the near future. The most important ones are the support of transactions and fault tolerance. In the long term, PRISMANET must also provide distributed and evolution mechanisms to the architectural elements. Automatic code generation of other programming languages and technologies is future work that could imply the implementation of other middlewares if required by the new technological platforms. Furthermore, an abstract middleware that would hide the differences between the different platforms could also be developed.

The PRISMA methodology does not support the identification of architectural elements and aspects from the requirements specification. As a result, one future work is to integrate ATRIUM [Nav03] with PRISMA to provide complete support to every stage of the software life-cycle (from requirements to implementation).

In addition, the extension of PRISMA methodology also offers opportunities for future work. This extension can consist of providing mechanisms that will take into account product family modelling as well as the variability that software architectures of this kind would introduce at the PRISMA modelling stage. Yet another task is to create a repository with a query language and metadata description of the architectural elements and aspects to improve reusability even more. The incorporation of COTS introduces the possibility of importing web services in PRISMA, making the study of PRISMA as a Service-Oriented Architecture (SOA) necessary. Therefore, another interesting task is to analyze what implications the SOA support will have for the PRISMA model and the PRISMANET implementation.

Finally, it is necessary to evaluate PRISMA using different applications and case studies in order to perform a quantified evaluation of the approach. A comparison of aspect-oriented and non-aspect oriented applications is necessary to be able to measure the advantages that PRISMA provides in comparison with other approaches. This measurement should be made taking into account different case studies and domains in order to have a wide sample that will allow us to get a set of well based conclusions.

Some of these research works have already started, especially those that improve the MDD support. For this reason, there are contributions about the maintenance and evolution support of the PRISMA MDD approach that have been published, but they are not included in this thesis of master. These publications are the following:

INTERNATIONAL CONFERENCES

- Cristóbal Costa, Nour Ali, **Jennifer Pérez**, Jose A. Carsí, Isidro Ramos, *Dynamic Reconfiguration of Software Architectures through Aspects*, In: Oquendo, F. (ed.) 1st European Conference on Software Architecture (ECSA'07). LNCS, vol. 4758, pp. 279-283 Springer, Aranjuez, Madrid Spain, 24-26 September 2007 (To appear)
- Cristóbal Costa, **Jennifer Pérez**, Jose A. Carsí, *Dynamic Adaptation of Aspect-Oriented Components*, 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'07), Springer Verlag, LNCS 4608, ISSN 0302-9743, ISBN 978-3-540-73550-2, Tufts University, Medford (Boston area), Massachusetts, USA, 9-11 July 2007.

BIBLIOGRAPHY

- [Aks05] Aksit, M., *Systematic analysis of crosscutting concerns in the model-driven architecture design approach*. Symposium How Adaptable is MDA ?, 2005.
- [Ali06] Ali N., Pérez J., Costa C., Ramos I., Carsí J.A., *Mobile Ambients in Aspect-Oriented Software Architectures*. IFIP Working Conference on Software Engineering Techniques, Warsaw, Poland, October, 2006.
- [Ali05b] Ali N., Pérez J., Ramos I., Carsí J. A., *Integrating Ambient Calculus in Mobile Aspect-Oriented Software Architectures*. Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society Press, pp. 233-234, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper)
- [Ali05a] Ali N., Ramos I., Carsí J.A., *A Conceptual Model for Distributed Aspect-Oriented Software Architectures*. International Symposium on Information Technology: Coding and Computing (ITCC), IEEE Computer Society, Vol. 2, Las Vegas, Nevada, USA, April, 2005.
- [Ali03] Ali N., Silva J.F., Jaen J., Ramos I., Carsí J. A., Pérez J., *Mobility and Replicability Patterns in Aspect-Oriented Component- Based Software Architectures*. 15th IASTED International Conference, Parallel and Distributed Computing and Systems (PDCS), Marina del Rey, California, USA, 3-5, November 2003, ACTA Press, ISBN: 0-88986-392-X, ISSN: 1027-2658 , pp. 820-826.
- [All97a] Allen R., Garlan D., *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3, pp. 213-249, July 1997.
- [All97b] Allen R., *A Formal Approach to Software Architecture*. Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMUCS-97-144, Pittsburgh, Pennsylvania, USA, May, 1997.

- [Ama05] Amaya, P. A., González, C. F., Murillo J. M., *MDA and separation of aspects: An approach based on multiple views and subject oriented design*. AOM, AOSD, Chicago, USA, 2005.
- [Am04] Ambler, S., *Agile Model-driven Development with UML 2.0*, Cambridge University Press, 2004.
- [And03] Andrade, L.F., Fiadeiro J. L., *Architecture Based Evolution of Software Systems*. Formal Methods for Software Architectures, Lecture Notes in Computer Science, Springer Verlag, Eds. Marco Bernardo and Paola Inverardi, LNCS 2804, September 2003.
- [And02a] Andrade, L.F., Fiadeiro J. L., Gouveia J., Koutsoukos F., Wermelinger M., *Coordination for Orchestration*, 5th International Conference on Coordination Models and Languages.. LNCS 2315, pp. 5-13, York, UK; 2002.
- [And02b] Andrade, L.F., Fiadeiro J. L., Gouveia J., Koutsoukos F., *Separating Computation, Coordination and Configuration*.. Journal of Software Maintenance, Vol. 14 No. 5, pp. 353-369, 2002.
- [ASP07a] The AspectJ Project Website, <http://eclipse.org/aspectj/>
- [Bal85] Balzer, R., *A 15 Year Perspective on Automatic Programming*. IEEE. Transactions on Software Engineering, Vol.11, No.11, pp. 1257-1268, November 1985.
- [Bar04a] Barais O., Duchien L., *Safarchie studio: Argouml extensions to build safe architectures*. Workshop on Architecture Description Languages, IFIP WCC World-Computer Congress, Toulouse, France, 2004.
- [Bar04b] Barais, O., Cariou, E., Duchien, L., Pessemier, N., Seinturier L., *Transat: A framework for the specification of software architecture evolution*. The First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04), ECOOP, Oslo, Norway, June, 2004. <http://wcat04.unex.es/bib>

- [Bar03] Barais O., Duchien L., Pawlak R., *Separation of Concerns in Software Modeling: A Framework for Software Architecture Transformation*. IASTED International Conference on Software Engineering Applications (SEA) ACTA Press, ISBN 0-88986-394-6, pp. 663-668, Los Angeles, California, USA, November 2003.
- [Bar01] Barbacci M.R., Doubleday D., Weinstock C. B., Lichota R.W., *DURRA: An Integrated Approach to Software Specification, Modeling and Rapid Prototyping*. Technical Report CMU/SEI-91-TR-21, Software Engineering Institute (SEI), September, 1991.
- [Bass03] Bass L., Clements P., Kazman R., *Software Architecture in Practice*. Addison Wesley, 2nd Edition, 2003.
- [Ber01] Bergmans L., Aksit M., *Composing Multiple Concerns Using Composition Filters*. Communications of the ACM, Vol. 44, No.10, pp. 51-57, October 2001.
- [Ber94] Bergmans L., *The Composition-Filters Object Model*. PhD Thesis, Department of Computer Science, University of Twente, 1994.
- [Bey05] Beydeda, S., Book, M., Gruhn V., *Model-Driven Software Development*, Springer, 2005.
- [Bez06] Bézivin, J., Jouault, F., *Using ATL for Checking Models*. International Workshop on Graph and Model Transformation (GraMoT), ENTCS 152, 2006.
- [Bin96] Binns P., Engelhart M., Jackson M., Vestal S., *Domain-Specific Software Architectures for Guidance, Navigation, and Control*. International Journal of Software Engineering and Knowledge Engineering, Vol. 6, No. 2, 1996.
- [Boo99] Booch, Rumbaugh and Jacobson, *The UML Modeling Language User Guide*. Addison-Wesley, 1999
- [Bru04] Bruns G., Jagadeesan R., Jeffrey A., Riely J., *μabc : A Minimal Aspect Calculus*. 15th International Conference on Concurrency Theory (CONCUR), Lecture Notes in

Computer Science (LNCS), Springer-Verlag, Vol. 3170, London, UK, August 31 - September 3, 2004.

- [Bru02] Bruyninckx H., Konincks B., Soetens P., *A Software Framework for Advanced Motion Control*. Department of Mechanical Engineering, K.U. Leuven. OROCOS project inside EURON, Belgium, 2002.
- [Can01] Canal C, Pimentel E., Troya J.M., *Compatibility and Inheritance in Software Architectures*. Science of Computer Programming, vol. 41, no. 2. 2001.
- [Can00] Canal C., *A Language for the Specification and Validation of Software Architectures*. PhD. Thesis, The University of Malaga, 2000. (In Spanish)
- [Can99] Canal C, Pimentel E., Troya J.M., *Specification and Refinement of Dynamic Software Architectures*. The First Working IFIP Conference on Software Architecture (WICSA), Kluwer Academic Publishing, pp. 107-126, San Antonio, Texas, USA, February, 1999.
- [Car00] Carney, D., Long, F., *What Do You Mean by COTS?*. IEEE Software, March/April 2000, pp. 83-86.
- [Chi05] Chitchyan R., Rashid A., Sawyer P., Garcia A., Pinto M., Bakker J., Tekinerdogan B., Clarke S., Jackson A., *Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design*. Lancaster University, Lancaster, AOSD-Europe Deliverable D11, , pp. 1-259, AOSD-Europe-ULANC-9, 18 May 2005, <http://www.aosdeurope.net/>
- [Col00] Colcombet T., Fradet P., *Enforcing Trace Properties by Program Transformation*. The 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL), Communications of the ACM, pp. 54-66., 2000.
- [Cos00] Coste-Manière E., Simmons R., *Architecture, the Backbone of Robotic System*. IEEE International Conference on Robotics & Automation, pps. 505-513, San Francisco, USA, April, 2000.

- [Cue06] Cuesta C.E., Romay M.P., Fuente P., Barrio-Solorzano M., *Coordination as an Architectural Aspect*. Electronics Notes in Theoretical Computer Science, Vol. 154 No. 1, pp. 25-41, May 2006.
- [Cue05] Cuesta C.E., Romay M.P., Fuente P., Barrio-Solorzano M., *Architectural Aspects of Architectural Aspects*. 2nd European Workshop on Software Architecture (EWSA), Lecture Notes on Computer Science, Springer Verlag, LNCS 3527, pp. 247-262, May 2004.
- [Cue04] Cuesta C.E., Romay M.P., Fuente P., Barrio-Solorzano M., *Reflection-Based, Aspect-Oriented Software Architecture*. 1st European Workshop on Software Architecture (EWSA), Lecture Notes on Computer Science, Springer Verlag, LNCS 3024, pp. 43-26, Pisa, Italy, June, 2005.
- [Cue02] C.E. Cuesta Quintero, *Dynamic Software Architecture based on Reflection*. PhD. Thesis, Department of Computer Science, University of Valladolid, 2002. (In Spanish)
- [Dan04] Dantas D., Walker D., Washburn G., Weirich S., *Analyzing Polymorphic Advice*. Technical Report TR-717-04, Princeton University, December 2004.
- [Dij76] Dijkstra, E., *A Discipline of Programming*. Prentice-Hall, 1976.
- [Dou05] Douence R, Le Botlan D., *Report Towards a Taxonomy of AOP Semantics*. AOSD-Europe Deliverable D11, AOSD-Europe-INRIA-1, INRIA, CNRS, 7 July 2005, pp. 1-13 http://www.comp.lancs.ac.uk:8080/c/portal/layout?p_1_id=1.94
- [Dou04b] Douence R., Fradet P., Sudholt M., *Trace-Based Aspects*. Aspect-Oriented Software Development. Mehmet Aksit, Sioh' an Clarke, Tzilla Elrad, and Robert E. Filman, editors. Addison-Wesley, 2004.
- [Dou04a] Douence R., Teboul L., *A Crosscut Language for Control-Flow*. The 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE), Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 3286, Vancouver, Canada, October 2004.

- [Dou02b] Douence R., Sudholt M., *A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP)*. Technical Report 02/11/INFO, Ecole des mines de Nantes, 2nd edition, December 2002.
- [Dou2a] Douence R., Fradet P., Sudholt M., *A Framework for the Detection and Resolution of Aspect Interactions*. Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering GPCE 2002, Lecture Notes in Computer Science (LNCS), Springer-Verlag. In D. Batory, C. Consel, and W. Taha, editors, Vol. 2487, pp. 173–188, Pittsburgh, PA, USA, October 2002.
- [Dou01] Douence R., Motelet O., Sudholt M., *A Formal Definition of Crosscuts*. Proceedings of the 3rd International Conference on Reflection, Lecture Notes in Computer Science, Springer-Verlag, A. Yonezawa and S. Matsuoka eds., Vol. 2192, pp. 170–186, Kyoto, Japan, September 2001.
- [DSL07] Domain-Specific Language (DSL) Tools.
<http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx>
- [DS09] D’Souza D., Wills A., *Objects, Components and Frameworks with UML*. The Catalysis approach. Addison-Wesley 1.999.
- [EFT02] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794, 2002.
- [Eis98] Eisenbanch S., Radestock M., *Component Coordination in Middleware Systems*, IFIP International Conference on Distributed Systems Platforms and OpenDistributed Processing (Middleware98), septiembere 1998.
- [Elr01] Elrald T., Filman R. E., Bader A., *Aspect-Oriented Programming: An Introduction*. Communication of the ACM, Vol. 44, No. 10, October 2001.
- [EMF07] Eclipse Modeling Framework (EMF) <http://www.eclipse.org/modeling/emf>

- [Fer05] Fernández C., Pastor J.A., Sánchez P., Álvarez B., Iborra A., *Co-operative Robots for Hull Blasting in European Shiprepair Industry*. IEEE Robotics and Automation Magazine (RAM), September 2005.
- [Fia04] Fiadeiro J.L., Lopes A., *CommUnity on the Move: Architectures for Distribution and Mobility*. FMCO 2003, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 3188, pp. 177–196, F.S. de Boer et al. (Eds.), Heidelberg , Berlin, Germany, 2004.
- [Fue05] Fuentes L., Pinto M., Sánchez P., *Dynamic Weaving in CAM/DAOP: An Application Architecture Drive Approach*. Workshop on Dynamic Aspect, Aspect-Oriented Software Development, Chicago, Illinois, March, 14-18.
- [Fue03] Fuentes. L., Pinto. M., & Vallecillo A.. *How MDA can help designing component- and aspect-based applications*. EDOC, pp. 124-135, 2003.
- [Gar01] Garlan, D., *Software Architecture*, Wiley Encyclopedia of Software Engineering, J. Marciniak (Ed.), John Wiley & Sons, 2001
- [Gar00] Garlan D., Monroe R. T., Wile D., *Acme: Architectural Description of Component-Based Systems*. Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, pp. 47-68, 2000.
- [Gar95b] Garlan, D., *An Introduction to the Aesop System*. July 1995.
<http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesopoverview.ps>
- [Gar95a] Garlan, D., Perry D., *Introduction to the Special Issue on Software Architecture*. IEEE Transactions on Software Engineering, vol. 21 no. 4, April 1995.
- [Gar94] Garlan D., Allen R., Ockerbloom J., *Exploiting Style in Architectural Design Environments*. SIGSOFT'94: Foundations of Software Engineering, pp. 175–188, New Orleans, Louisiana, USA, December 1994.

- [Gar93] Garlan, D., Shaw M., *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering, Vol. I. Eds. V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jersey, 1993.
- [GME07] GME, The Generic Modeling Environment, <http://www.isis.vanderbilt.edu/Projects/gme/>
- [Gor94] Gorlick M., Quilici A., *Visual Programming in the Large versus Visual Programming in the Small*. The 1994 IEEE Symposium on Visual Languages, pp. 137-144, St. Louis, Missouri, USA, October, 1994.
- [Gor91] Gorlick M., Razouk R., *Using Weaves for Software Construction and Analysis*. The 13th International Conference on Software Engineering (ICSE13), pp. 23-34, Austin, Texas, USA, May, 1991.
- [Gre04] Greenfield J., Short K, Cook S., and Kent S. *Software Factories*. Wiley Publishing Inc., 2004.
- [Gru00] Grundy J., *Multi-perspective specification, design and implementation of software components using aspects*. International Journal of Software Engineering and Knowledge Engineering, vol. 20, 2000.
- [Grun98] Grundy J.C., Mugridge W.B., Hosking J.G., *Static and dynamic visualisation of component-based software architectures*. The 10th International Conference on Software Engineering and Knowledge Engineering, KSI Press, San Francisco, California, USA, 18-20 June, 1998.
- [Gue03a] Guerra P. A. C., Rubira C. M. F., Romanovsky A., de Lemos R., *Integrating COTS Software Components into Dependable Software Architectures*, The Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 2003.

- [Gue03b] Guerra P. A. C., Rubira C. M. F., Romanovsky A., de Lemos R., *A Dependable Architecture for COTS-Based Software Systems using Protective Wrappers*, ICSE, Workshop on Architecting Dependable Systems, Portland, Oregon, USA, 2003.
- [Gue02] Guerra P. A. C., Rubira C. M. F., de Lemos R., *An Idealized Fault-Tolerant Architectural Component*, ICSE, Workshop on Architecting Dependable Systems, Orlando, USA, 15-20, 2002
- [Ham05] Hammouda I., Hakala M., Pussinen M., Katara M., Mikkonen T., *Concerni-Based Development of Pattern Systems*. The 2nd European Workshop on Software Architecture (EWSA), Lecture Notes on Computer Science, Springer Verlag, LNCS 3527, pp. 113-129, Pisa, Italy, June, 2005.
- [Ham04] Hammouda I., koskinen J., Pussinen M., Katara M., Mikkonen T., *Adaptable Concerni-Based Framework Specialization in UML*, Automated Software Engineering, pp. 78-87, Linz, Austria, 2004.
- [Har02] Harrison W., Harold L., Ossher H., Peri T., *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. IBM Research Report RC22685 (W0212-147), Thomas J. Watson Research Center, IBM, December, 2002.
- [Har03] Harrison W., Ossher H., *Subject-oriented programming (a critique of pure objects)*. Conference on Object-Oriented Programming: Systems, Languages and Applications, Communications of the ACM, pp. 411-428, September 1993.
- [IEE00] *IEEE Recommended Practices for Architectural Description of Software-Intensive Systems*. IEEE Std 1471-2000, Software Engineering Standards Committee of the IEEE Computer Society, 21 September 2000.
- [Jag06a] Jagadeesan R., Jeffrey A., Riely J., *Typed Parametric Polymorphism for Aspects*. Science of Computer Programming, 2006.
- [Jag06b] Jagadeesan R., Jeffrey A., Riely J., *A Typed Calculus of Aspect-Oriented Programs*. <http://fpl.cs.depaul.edu/rjagadeesan/pubs.html>, 2003.

- [Jou05] Jouault, F., Kurtev, I. *Transforming Models with ATL*. Workshop on Model Transformations in Practice, collocated with MoDELS, Jamaica 2005.
- [Kan02b] Kande M. M., Kienzle J., Strohmeier A., *From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach*, Technical Reports in Computer and Communication Sciences, Faculté I&C, École Polytechnique Fédérale de Lausanne, 2002.
- [Kan02a] Kande M. M., Kienzle J., Strohmeier A., *From AOP to UML - A Bottom-Up Approach*. Workshop on Aspect-Oriented Modeling with UML, AOSD, Enschede, The Netherlands, 2002.
- [Kan03] Kande M. M., *A concern-oriented approach to software architecture*. PhD. Thesis, Lausanne, Switzerland: Swiss Federal Institute of Technology (EPFL), 2003.
- [Kat03] Katara M., Katz S., *Architectural Views of Aspects*. The 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), Boston, Massachusetts, USA, 2003.
- [Kat02] Katara M., *Superposing UML Class Diagram*. Workshop on Aspect-Oriented Modelling with UML, AOSD, Enschede, The Netherlands, 2002.
- [Ken95] Kenney J.J., *Executable Formal Models of Distributed Transaction Systems based on Even Processing*. PhD. Thesis, University of Stanford, CSL, December, 1995.
- [Kiz01] Kiczales G., Hilsdale E., Huguin J., Kersten M., Palm J., Griswold W.G., *An Overview of AspectJ*. The 15th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol.2072, Budapest, Hungary, June 18-22, 2001.
- [Kiz97] Kiczales G., Lamping J., Mendhekar A., Maeda C., *Aspect-Oriented Programming*. The 11th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 1241, Jyväskylä, Finland, June 9-13, 1997.

- [Kram85] Krammer J. Magee J., *Dynamic Configuration for Distributed Systems*. IEEE Transactions on Software Engineering, Vol. 11, No. 4, pp. 424-436, 1985.
- [Kul03] Kulkarni, V., Reddy S., *Separation of concerns in model-driven development*, IEEE software 20(5), 2003.
- [Kur06] Kurtev, I., et al. *Model-based DSL frameworks*. 21st Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Portland, Oregon, USA, 2006.
- [Kva05] Kvale A. A., Li J., Conradi R., *A case study on building COTS-based system using aspect-oriented programming*. ACM Sym. on Applied Computing, 1491-149, 2005
- [Läm02] Lämmel R., *A Semantical Approach to Method-Call Interception*. The 1st International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 41–55, Twente, The Netherlands, April 2002.
- [Lie99] Lieberherr K., Lorenz D., Mezini M., *Programming with Aspectual Components*. Technical Report NU-CCS-99-01, pp. 1-27, Northeastern University, Boston, Massachusetts, March 1999.
- [Lop05] Lopes A., Fiadeiro J. L., *Context-Awareness in Software Architectures*. The 2nd European Workshop on Software Architecture (EWSA), Lecture Notes on Computer Science, Springer Verlag, LNCS 3527, pp. 146-161, Pisa Italy, June, 2005.
- [Lop03] Lopes A., Wermelinger M., Fiadeiro J.L., *High-Order Architectural Connectors*. ACM Transaction on Software Engineering and Methodology. Vol. 12 No. 1, pp.64-104, 2003.
- [Loq00] Loques O., Sztajnberg A., Leite J., Lobosco, M., *On the Integration of Meta-Level Programming and Configuration Programming*, In Reflection and Software Engineering (special edition), Editors: Walter Cazzola, Robert J. Stroud, Francesco Tisato, V. 1826, Lecture Notes in Computer Science, pp.191-210, Springer-Verlag, Heidelberg, Germany, June, 2000.

- [Luc95b] Luckham D.C., Vera J., *An Event-Based Architecture Definition Language*. IEEE Transactions on Software Engineering, Vol. 21, No. 9, pp. 717-734, September, 1995.
- [Luc95a] Luckham D.C., Kenney J.J., Augustine L.M., Vera J., Bryan D., Mann W., *Specification and Analysis of Software Architecture using Rapide*. IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 336-355, April, 1995.
- [Mag96] Magee J., Kramer J., *Dynamic Structure in Software Architectures*. ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4), pp. 3-14, San Francisco, CA, October 1996.
- [Mag95] Magee J.N., Dulay N., Eisenbach S., Kramer J., *Specifying Distributed Software Architectures*. Fifth European Software Engineering Conference (ESEC), Barcelona, Spain, September, 1995.
- [Mag89] Magee J., Krammer J., Sloman M., *Constructing Distributed Systems in Conic*. IEEE Transactions on Software Engineering, Vol. 15, No. 6, 1989.
- [Mas03] Masuhara H., Kawauchi K., *Dataflow Pointcut in Aspect-Oriented Programming*. The First Asian Symposium on Programming Languages and Systems (APLAS'03), Lecture Notes in Computer Science, Springer-Verlag, vol. 2895, pp. 105–121, 2003.
- [McD03] McDirmid S., Hsieh W.C., *Aspect-Oriented Programming with Jiazzi*. The 2nd International Conference on Aspect-Oriented Software Development (AOSD), pp. 70-79, Boston, Massachusetts, USA, march, 2003.
- [MDA07] Object Management Group. Model Driven Architecture Guide, 2003
<http://www.omg.org/docs/omg/03-06-01.pdf>
- [Med00] Medvidovic N., Taylor R.N., *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, Vol. 26, No. 1, January, 2000.

- [Med99] Medvidovic N., Rosenblum D. S., Taylor R. N., *A Language and Environment for Architecture-Based Software Development and Evolution*. The 21st International Conference on Software Engineering (ICSE'99), Los Angeles, California, May 1999.
- [Med96] Medvidovic N., Oreizy P., Robbins J. E., Taylor R. N., *Using Object-Oriented Typing to Support Architectural Design in the C2 Style*. ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4), pp. 24-32, San Francisco, California, USA, October 1996.
- [Meh00] Mehta N., Medvidovic N., Phadke S., *Towards a Taxonomy of Software Connectors*. 22nd International Conference on Software Engineering (ICSE2000), Vol. 11, pp.178-187, Limerick, June 2000.
- [MEY07] Meyer B., *What to compose?*. Eiffel Software,
<http://archive.eiffel.com/doc/manuals/technology/bmarticles/sd/compose.html>
- [Mey03] Meyer B., *The Grand Challenge of Trusted Components*. International Conference on Software Engineering (ICSE), IEEE Computer Press, Portland, Oregon, May 2003.
- [Mez03] Mezini M., Ostermann K., *Conquering Aspects with Caesar*. International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 90-100, Boston, Massachusetts, USA, March, 2003.
- [Mil99] Milner R., *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, June, 1999.
- [Mil93] Milner R., *The Polyadic π -Calculus: A Tutorial*. Laboratory for Foundations of Computer Science Department, University of Edinburgh, October 1993.
- [MOF02] Meta-Object Facility, *Object Management Group (OMG): MOF 1.4 Specification*,
 TR formal/2002-04-03, 2002 from
<http://www.omg.org/technology/documents/formal/mof.htm>

- [Mod06] Modelware project. D1.5 Model Composition: Development of consistency rules. TR, WP1 Modelling technologies, 2006 <http://www.modelware-ist.org>
- [Mon98] Monroe R.T., *Capturing Software Architecture Design Expertise With Armani*. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, October 1998.
- [Mor97] Moriconi M., Riemenschneider R. A., *Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies*. Technical Report SRI-CSL-97-01, SRI International, March, 1997.
- [Mor95] Moriconi M., Qian X., Riemenschneider R. A., *Correct Architecture Refinement*. IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 356-372, April 1995.
- [Nav05] Navasa A., Pérez M. A., Murillo J. M., *Aspect Modelling at Architecture Design*. 2nd European Workshop on Software Architecture (EWSA), Lecture Notes on Computer Science, Springer Verlag, LNCS 3527, pp. 41-58, Pisa, Italy, June, 2005.
- [Nav04a] Navarro E., Letelier P., Ramos I., *Goals and Quality Characteristics: Separating Concerns*, Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, OOPSLA, Monday, October 25, Vancouver, Canada, 2004.
- [Nav04b] Navarro E., Letelier P., Ramos I., *UML Visualization for an Aspect and Goal-Oriented Approach*, The 5th Aspect-Oriented Modeling Workshop (AOM'04), UML 2004, Monday, October 11, 2004, Lisbon, Portugal
- [Nav04c] Navarro E., Ramos I., Letelier P., Pérez J., *Goals Model-Driving Software Architectur*. The 2nd International Conference on Software Engineering Research, Management and Applications (SERA), May 5-8, 2004, Los Angeles, CA.
- [Nav03] Navarro E., Ramos I., Pérez J., *Software Requirements for Architected Systems*. The 11th IEEE International Requirements Engineering Conference (RE'03), IEEE Computer Society, September 8-12, Monterey, California , 2003.

- [Nie95] Nierstrasz O., Meijler T.D., *Research Directions of Software Composition*. ACM Computing Surveys (CSUR), Vol. 27, no. 2, June, 1995.
- [Obe97] Oberndorf T., *COTS and Open Systems - An Overview*, 1997,
<http://www.sei.cmu.edu/str/descriptions/cots.html#ndi>
- [Oli98] Oliva A., Garcia I.C., Buzato L.E., *The Reflective Architecture of Guaraná*, Technical Report IC-98-14. Instituto de computación, Universidad de Campiñas, April, 1998.
- [Oqu04a] Oquendo F., *π -ADL: An Architecture Description Language based on the Higher-Order Typed π -Calculus for Specifying Dynamic and Mobile Software Architectures*. ACM Software Engineering Notes, Vol. 29, No. 3, May, 2004.
- [Oqu04b] Oquendo F., Warboys B., Morrison R., Dindeleux R., Gallo F., Garavel H., Occhipinti C., *ArchWARE: Architecting Evolvable Software*. The 1st European Workshop in Software Architecture (EWSA 2004), Lecture Notes in Computer Science LNCS 3047, St Andrews, UK, pp. 257-271. Springer, ISBN 3-540-22000-3. 2004
- [Oss01] Ossher H., Tarr P., *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. The Symposium on Software Architectures and Component Technology: The state of the Art in Software Development, Kluwer, 2001.
- [Oss00] Ossher H., Tarr P., *Hyper/JTM: Multi-Dimensional Separation of Concerns for JavaTM*. The International Conference on Software Engineering (ICSE). Communications of the ACM, pp. 734-737, Limerick, Ireland (2000).
- [Pap01] Papadopoulos G. A., Arbab F., *Configuration and Dynamic Reconfiguration of Components using the Coordination Paradigm*, Future Generation Computer Systems, Vol. 17 No. 8, pp. 1023-1038, June 2001.
- [Par72] Parnas D. L., *On the Criteria to be used in Decomposing Systems into Modules*. Communications of the ACM, Vol 15, No.12, pp. 1053–1058, December 1972.

- [Pas97] Pastor O. Et al, *OO-METHOD: A Software Production Environment Combining Conventional and Formal Methods*. The 9th International Conference, CaiSE97, Barcelona, 1997.
- [Paw04] Pawlak R., Seinturier L., Duchien L., Florin G., Legond-Aubry F., Martelli L., *JAC: an Aspect-Based Distributed Dynamic Framework*. Software – Practice and Experience, Vol. 34, pp. 1119-1148, 2004.
- [Per92] Perry, D., Wolf A., *Foundations for the Study of Software Architecture*. ACM Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.
- [Per06a] Pérez J., Navarro E., Letelier P., Ramos I., *Graphical Modelling Proposal For Aspect-Oriented SA*. The 21st Annual ACM Symposium on Applied Computing (SAC), ACM, Dijon, France, April 23-27, 2006. (short paper)
- [Per06b] Pérez J., Navarro E., Letelier P., Ramos I., *A Modelling Proposal For Aspect-Oriented Software Architectures*. The 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), IEEE Computer Society Press, Hasso-Plattner-Institute For Software Systems Engineering at the University Of Potsdam, Potsdam, Germany (Berlin Metropolitan Area), 27-30 March.
- [Per06c] Pérez J., *PRISMA: Aspect-Oriented Software Architectures*. Tesis Doctoral, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.
- [Per06d] Pérez, J., Ali, N., Carsí, J.A., Ramos, I., *Designing Software Architectures with an Aspect-Oriented Architecture Description Language*. The 9th International Symposium on Component-Based Software Engineering (CBSE), LNCS 4063, Västerås, Sweden, 2006.
- [Per05a] Pérez, J., Ali, N., Carsí, J.A., Ramos, I., *Dynamic Evolution in Aspect-Oriented Architectural Models*. Second European Workshop on Software Architecture, LNCS 3527, Springer Verlag, Pisa, 2005.

- [Per05b] Pérez J, Ali N, Costa C, Carsí JA., Ramos I., *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. The International Conference of .NET Technologies, Vol.3, No.1-3, ISSN 1801-2108, Pilsen, Czech Republic, 2005.
- [Per02a] Pérez J., Carsí J A. and Ramos I., *On the implication of application's requirements changes in the persistence layer: an automatic approach*. Workshop on the Database Maintenance and Reengineering (DBMR'2002), IEEE International Conference of Software Maintenance, Montreal (Canada), October 1st, 2002, pp. 3-16, ISBN: 84-699-8920-0.
- [Per02b] Pérez J., Carsí J. A. and Ramos I., *ADML: A Language for Automatic Generation of Migration Plans*. The First Eurasian Conference on Advances in Information and Communication Technology, Tehran, Iran, October 2002 <http://www.eurasia-ict.org/> © Springer LNCS vol n.2510
- [Per02c] Pérez J., Anaya V., Cubel J M., Domínguez F., Boronat A., Ramos I. and Carsí J A., *Data Reverse Engineering of Legacy Databases to Object Oriented Conceptual Schemas*. Software Evolution Through Transformations: Towards Uniform Support throughout the Software Life-Cycle Workshop (SET'02), First International Conference on Graph Transformation(ICGT2002), Barcelona (Spain), October, 2002 © ENTCS vol n. 72.4
- [Pin05] Pinto M., Fuentes L., Troya J.M., *A Dynamic Component and Aspect Platform*, The Computer Journal Vol. 48, No. 4, pp. 401-420, 2005.
- [Pin03] Pinto M., Fuentes L., Troya J. M., *DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development*. Generative Programming and Component Engineering: Second International Conference, GPCE Springer Verlag, Lecture Notes Computer Science, ISSN: 0302-9743, Erfurt, Germany, September 22-25.
- [PRI07] PRISMA, <http://prisma.dsic.upv.es>

- [QVT05] Object Management Group. MOF QVT final adopted specification. Tech. Report formal/2005-11-01, 2005
- [RAT07] Rational Software, Rational Rose, <http://www-306.ibm.com/software/rational/>
- [Sch06] Schmidt D.C., *Model-Driven Engineering*, IEEE computer Society, 2006.
- [Sch01] Scholl K.U., Albiez J., Gassmann B., *MCA: An Expandable Modular Controller Architecture*. The 3rd Real-Time Linux Workshop, Karlsruhe University, Milano, Italy, 2001.
- [Ser94] Sernadas, A., Costa J.F., Sernadas C., Object Specifications Through Diagrams: OBLOG Approach. INESC Lisbon 1994.
- [Sha96] Shaw M., DeLine R., Zelesnik G., *Abstractions and Implementations for Architectural Connections*. The Third International Conference on Configurable Distributed Systems, May, 1996.
- [Sha95] Shaw M., DeLine R., Klein D. V., Ross T. L., Young D. M., Zelesnik G., *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 314-335, April 1995.
- [Sha94] Shaw M., *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status*. Workshop on Studies of Software Design, January, 1994.
- [Sih03] Sihman M., Katz S., *Superimpositions and Aspect-Oriented Programming*. The Computer Journal, Vol 46, No. 5, pp. 529-541, September, 2003.
- [Sim05] Simmonds, D., Reddy, R., France, R., Ghosh, S., Solberg, A., An aspect oriented model driven framework. Ninth IEEE International EDOC Enterprise Computing Conference pp. 119-130, 2005.
- [Sti92] Stirling C., *Modal and Temporal Logics*. Handbook of Logic in Computer Science, vol II, Clarendon Press, Oxford, 1992.

- [Sto02] Stojanovic Z., Dahanayak A., *Components and Viewpoints as Integrated Separations of Concerns in System Designing*, Workshop on Identifying, Separating and Verifying Concerns in the Design, AOSD, Enschede, The Netherlands, 2002.
- [Suv05b] Suvée D., De Fraine B., Vanderperren W., *FuseJ: An architectural description language for unifying aspects and components*. Workshop on Software-engineering Properties of Languages and Aspect Technologies, 2005.
- [Suv03] Suvée D., Vanderperren W., Jonckers V., *JAsCo: an Aspect-Oriented Approach Tailored for Component-Based Software Development*. The 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 21-29, ISBN 1-58113-660-9, Boston, Massachusetts, March, 2003.
- [SYS07] Telelogic System Architect.

http://www.telelogic.com/products/system_architect/index.cfm
- [Szy00] Szyperski C., Booch G., Meyer B., *Beyond Objects*. Software Development Magazine, March, 2000 (originally BrucePowel Douglass).
- [Szy98] Szyperski C., *Component software: beyond object-oriented programming*. ACM Press and Addison Wesley, New York, USA (1998).
- [Tar99] Tarr P., Ossher, H., Harrison, W., Sutton Jr., S. M., *N Degrees of Separation: Multidimensional Separation of Concerns*. The 21st International Conference on Software Engineering (ICSE), Communication of the ACM, pp. 107-119, New York, 1999.
- [TEA07] The *TeachMover* Robot,

<http://www.questechzone.com/microbot/teachmover.htm>
- [Tek05] Tekinerdogan B., Scholten F., *ASAAM-T: A tool environment for identifying architectural aspects*. Aspect-Oriented Software Development Conference (AOSD), March 18-19, Chicago, 2005.

- [Tek04] Tekinerdogan B., *ASAAM: Aspectual software architecture analysis method*. The 4th Working IEEE/IFIP Conference on Software Architecture (WICSA), Oslo, Norway, 2004.
- [TOG07] Together, Borland Software Corporation.
<http://www.borland.com/us/products/together/index.html>
- [Tov07] Toval, A., Requena, V., Alemán, J.L. OCL Tools, <http://www.um.es/giisw/ocltools/>
- [UML07] The Unified Modeling Language Website, Object Management Group (OMG), <http://www.uml.org/>
- [Van01] Vanderperren, W. and Wydaeghe, B. *Towards a New Component Composition Process*. The 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), Washington DC, April 2001.
- [Ver03] Verspecht D., Vanderperren W., Suvee D., Jonckers V., *JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services*. The Second Nordic Conference on Web Services NCWS'03, Vaxjo, Sweden. In "Mathematical modelling in Physics, Engineering and Cognitive Sciences", Vol. 8, November 2003.
- [Ves96] Vestal S., *MetaH Programmer's Manual, Version 1.09..* Technical Report, Honeywell Technology Center, April 1996.
- [Vig96] Vigder M., Gentleman W.M., Dean J., *COTS Software Integration: State of the Art*, Inst. for Information Technology, NRC Nat. Rsch., Council Canada, 1996.
- [Voa98] Voas J., *Maintaining Component-Based Systems*. IEEE Software, July/August, pp. 22-27, 1998.
- [Vol01] Volpe R., Nesnas I., Estlin T., Mutz D., Petras R., Das H., *The CLARAty architecture for robotic autonomy*. IEEE Aerospace Conference. Vol. 1, pp. 121-132, Montana, USA, 2001.

Bibliography

- [Wal03] Walker D., Zdancewic S., Ligatti J. *A Theory of Aspects*. International Conference on Functional Programming (ICFP), Communications of the ACM, pp. 127–139, 2003.

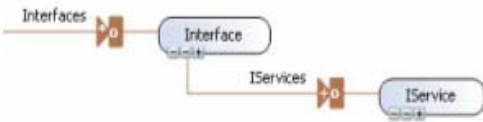
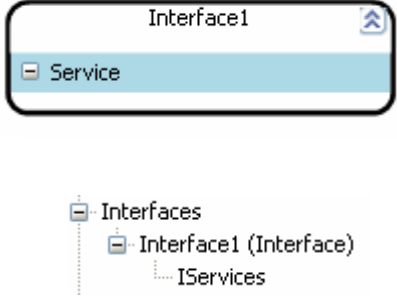
- [Wan04] Wand M., Kiczales G., Dutchyn C., *A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming*. TOPLAS, Vol. 26, No. 5, pp. 890–910, September 2004. Earlier versions of this paper were presented at the 9th International Workshop on Foundations of Object-Oriented Languages, January 19, 2002, and at the Workshop on Foundations of Aspect-Oriented Languages (FOAL), April 22, 2002.
- [Wyd01] Wydaeghe B., Vanderperren W., *Visual Component Composition Using Composition Patterns*. Tools, Santa Barbara, California, July 2001.
- [Yak99] Yakimovich, D., Bieman, J. M., and Basili, V. R. *Software architecture classification for estimating the cost of cots integration*. the 21st Int. Conf. on Software engineering, IEEE Computer Society Press, 296-302, 1999.

APPENDIX A

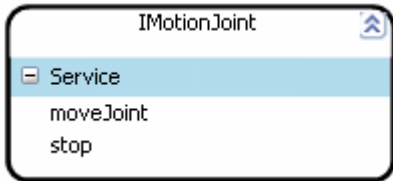
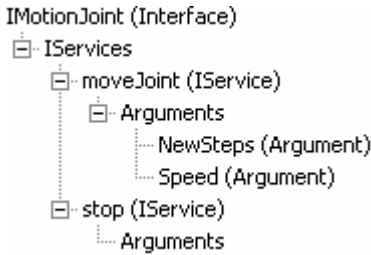
PRISMA CODE GENERATION PATTERNS

This appendix presents the code generation patterns to generate the C# from PRISMA types models.

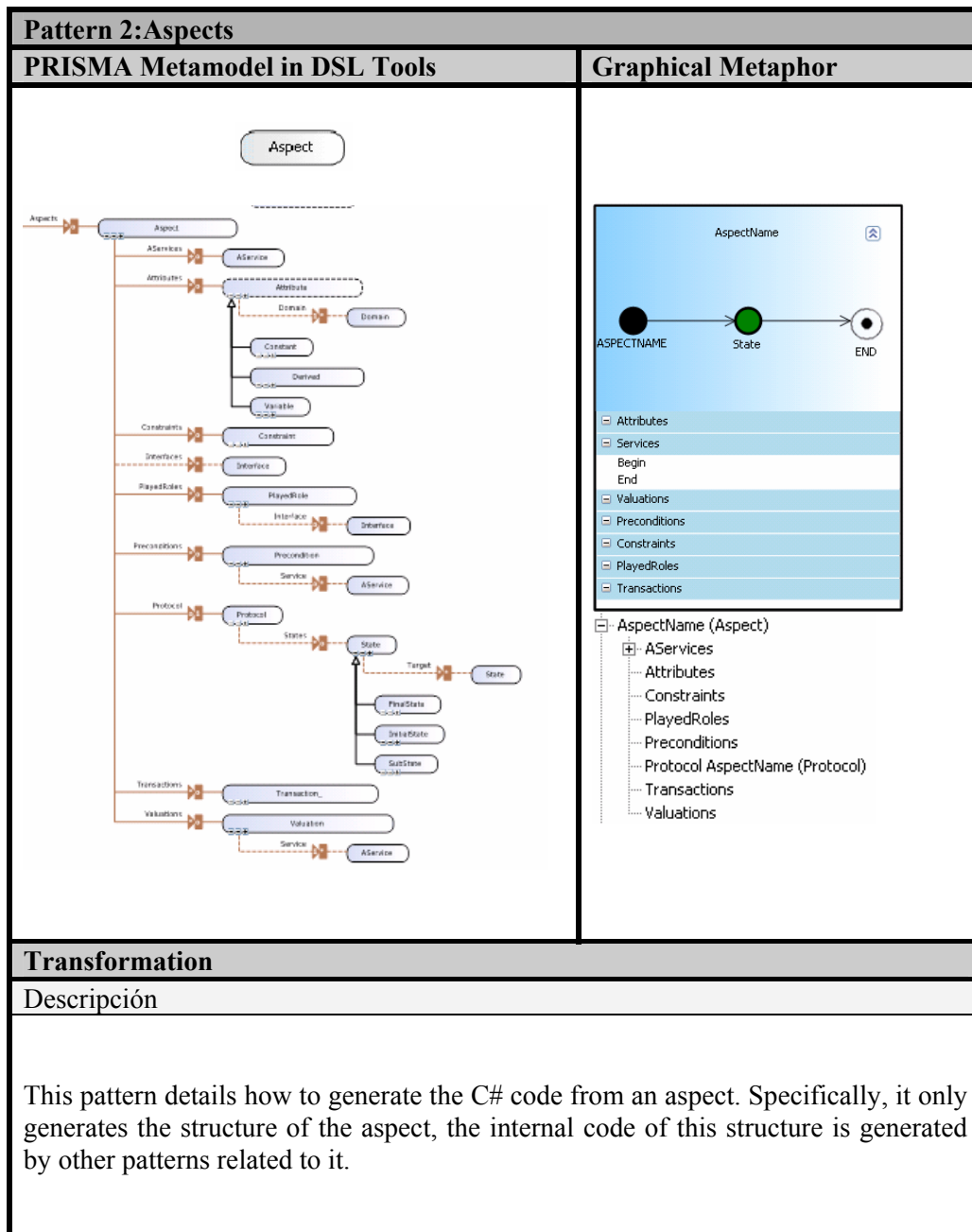
A.1. INTERFACES

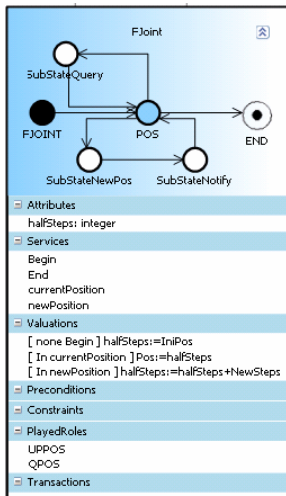
Pattern 1: Interfaces	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
	
Transformation	
Description	
This pattern details how to generate the C# code from an interface.	

Template
<pre> using System; using PRISMA; namespace <#=this.Model.Name#> { <# SortedList serviceList = new SortedList(); foreach (Interface interfaz in this.Model.Interfaces) { #> public interface <#=interfaz.Name#> { <# foreach (Service servicio in interfaz.IServices) { #> AsyncResult <#=servicio.Name#>(<#=CommaSeparatedArguments(servicio.Arguments) #>); <# if(!serviceList.Contains(servicio.Name)) serviceList.Add(servicio.Name,servicio); } #> } <# } foreach(Service servicio in serviceList.Values) { #> public delegate AsyncResult <#=servicio.Name#>Delegate (<#=CommaSeparatedArguments(servicio.Arguments) #>); <# } #> } } </pre>
Case Study
Description
<p>This pattern is illustrated using the interface <i>ImotionJoint</i> of the <i>TeachMover</i> case study. This interface specifies the services that are required to move and stop the <i>TeachMover</i> robot.</p> <p>The representation of the <i>ImotionJoint</i> in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>

Graphical representation	
	
Result of the pattern execution	
<pre>using System; using PRISMA; namespace RobotJoint { public interface IMotionJoint { AsyncResult moveJoint(int NewSteps, int Speed); AsyncResult stop(); } public delegate AsyncResult moveJointDelegate(int NewSteps, int Speed); public delegate AsyncResult stopDelegate(); }</pre>	
Related Patterns	
There are no related patterns	

A.2. ASPECTS



Template
<pre> using System; using PRISMA; using PRISMA.Aspects.Types; using PRISMA.Exceptions; using System.Collections; namespace <#=this.Model.Name#> { <# foreach (Aspect aspect in this.Model.Aspects) { #> [Serializable] public class <#=aspect.Name#> : <#=aspect.Concern#>Aspect<#=CommaSeparatedNames3 (aspect.Interfaces) #> { <# /* Internal code generation of the aspect ... */ }/* endforeach (Aspect aspect in this.Model.Aspects) */ #> } </pre>
Case Study
Description
<p>This pattern is illustrated using the aspect <i>FJoint</i> of the TeachMover architectural model. The representation of the <i>FJoint</i> in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>
Representación
 <p>The diagram shows the UML representation of the <i>FJoint</i> aspect and the corresponding C# code generated from it. The UML diagram includes states: <i>SubStateQuery</i>, <i>FJOINT</i> (initial state), <i>POS</i>, <i>SubStateNewPos</i>, <i>SubStateNotify</i>, and <i>END</i> (final state). Transitions are labeled with events and actions. The C# code below the diagram lists the following elements:</p> <ul style="list-style-type: none"> Attributes: <code>halfSteps: integer</code> Services: <code>Begin</code>, <code>End</code>, <code>currentPosition</code>, <code>newPosition</code> Valuations: <ul style="list-style-type: none"> <code>[none Begin] halfSteps:=IniPos</code> <code>[In currentPosition] Pos:=halfSteps</code> <code>[In newPosition] halfSteps:=halfSteps+NewSteps</code> Preconditions: Constraints: PlayedRoles: <code>UPPOS</code>, <code>QPOS</code> Transactions:

FJoint (Aspect)

AServices

Attributes

Constraints

PlayedRoles

Preconditions

Protocol FJoint (Protocol)

Transactions

Valuations

Properties

FJoint Aspect

Concern

FillColor

Name

OutlineColor

Functional

LightSkyBlue

FJoint

Black

Result of the pattern execution

```
using System;
using PRISMA;
using PRISMA.Aspects.Types;
using PRISMA.Exceptions;
using System.Collections;

namespace RobotJoint
{
    [Serializable]
    public class FJoint : FunctionalAspect, IQueryPos, IUpdatePos
    {
        ...
    }
}
```

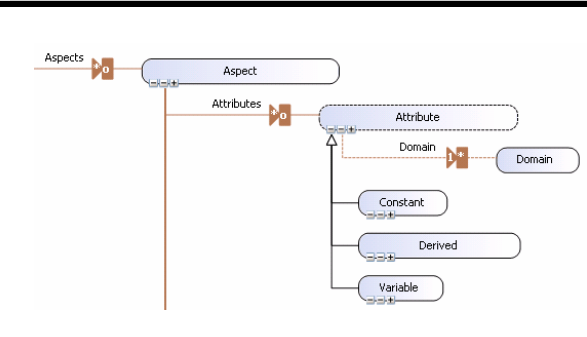
Related patterns

Pattern 3, Pattern 4, Pattern 5, Pattern 6, Pattern 7, Pattern 8, Pattern 9, Pattern 10, Pattern 11, Pattern 12, and Pattern 13.

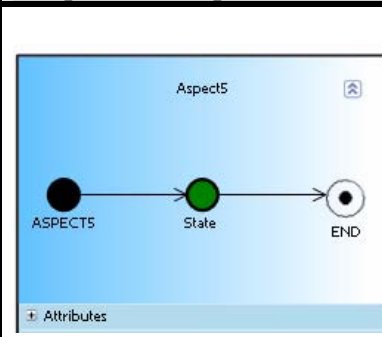
A.2.1.Attributes

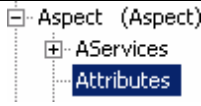
Pattern 3:Attributes

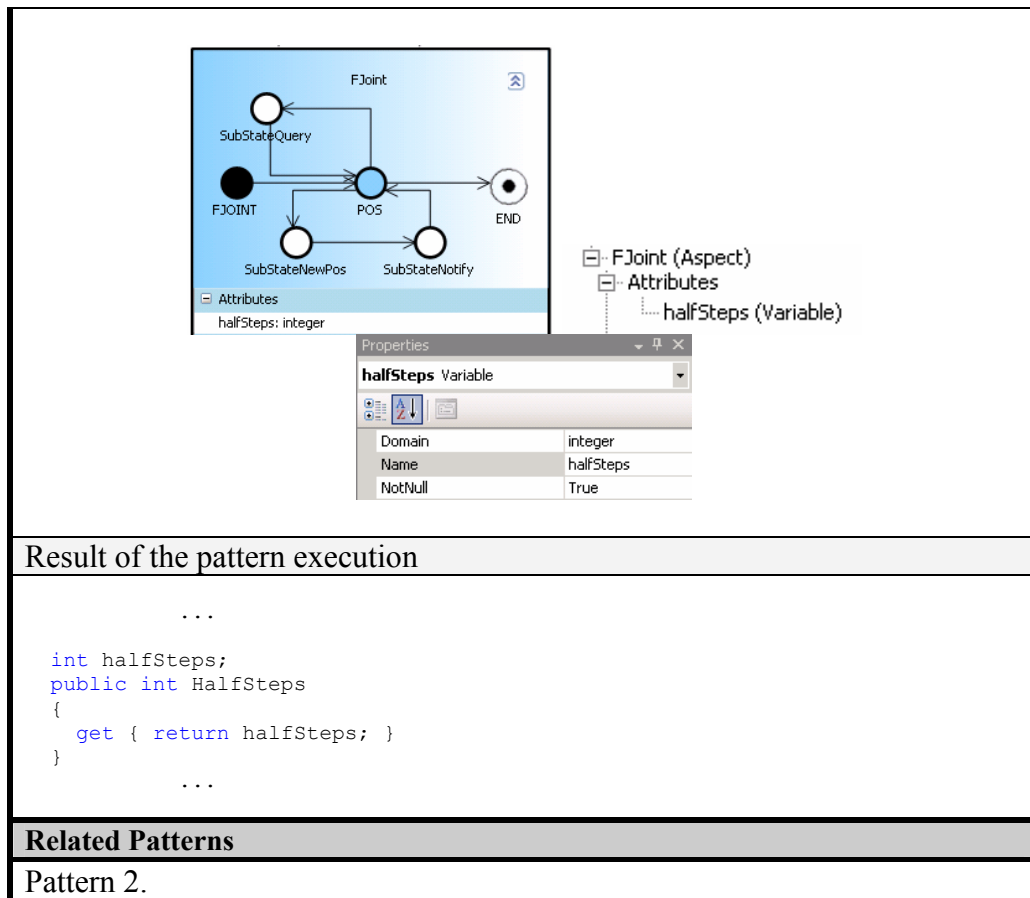
PRISMA Metamodel in DSL Tools



Graphical Metaphor



	
Transformation	
Description	
<p>This pattern details how to generate the C# code from an attribute. There are three kinds of attributes: Constant, Variable and Derived. The code generation for derived attributes is still not supported. Constant and Variable attributes follow the same pattern because there are no differences in their code generation.</p>	
Template	
<pre> ... <# SortedList attributes=new SortedList(); foreach (DSIC.ISSI.PrismaDSL.DomainModel.Attribute attribute in aspect.Attributes) { attributes.Add(attribute.Name,null); } #> <#=DomainToType(attribute.Domain) #> <#=attribute.Name #>; public <#=DomainToType(attribute.Domain) #> <#=attribute.Name.Substring(0,1).ToUpper() #><#=attribute.Name.Substring(1) #> { get { return <#=attribute.Name #>; } } <# } #> ... </pre>	
Case Study	
Description	
<p>This pattern is illustrated using the attribute <i>halfSteps</i> of the aspect <i>FJoint</i> of the TeachMover architectural model. The <i>halfSteps</i> is an Integer attribute to store the position of the Joint. The representation of the attribute <i>halfSteps</i> in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>	
Representation	



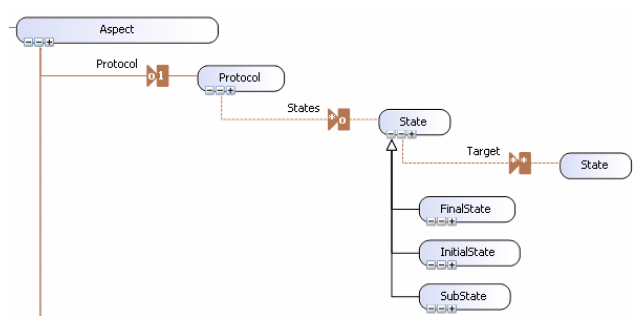
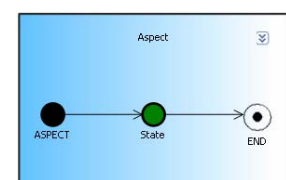
Result of the pattern execution

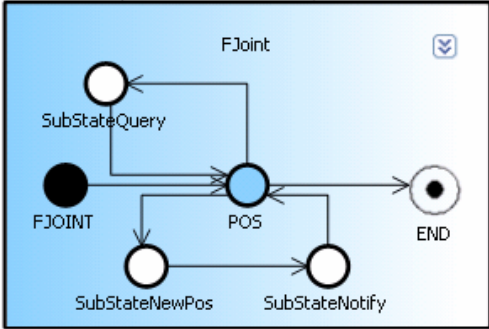
```
...
int halfSteps;
public int HalfSteps
{
    get { return halfSteps; }
}
...
```

Related Patterns

Pattern 2.

A.2.2. Protocol

Pattern 4: Protocols																																			
PRISMA Metamodel in DSL Tools	Graphical Metaphor																																		
 <p>The diagram illustrates the PRISMA Metamodel in DSL Tools. It shows an 'Aspect' entity containing a 'Protocol' entity. The 'Protocol' entity contains a collection of 'States'. A 'State' entity is shown with a 'Target' relationship to another 'State' entity. The 'State' entity has three sub-entities: 'FinalState', 'InitialState', and 'SubState'.</p>	 <p>The graphical metaphor shows a flow from 'Aspect' to 'State' to 'END'. The 'Aspect' is represented by a black circle, 'State' by a green circle, and 'END' by a white circle with a black border. The flow is indicated by arrows.</p> <table border="1"> <thead> <tr> <th colspan="2">Properties</th></tr> </thead> <tbody> <tr> <td colspan="2">StateHasState</td></tr> <tr> <td>AspectPlayedRole</td><td></td></tr> <tr> <td>AspectService</td><td>Begin</td></tr> <tr> <td>AspectTransaction</td><td></td></tr> <tr> <td>Color</td><td>Black</td></tr> <tr> <td>Condition</td><td></td></tr> <tr> <td>DashStyle</td><td>Solid</td></tr> <tr> <td>IA_PlayedRole</td><td>(none)</td></tr> <tr> <td>IntAsp_PlayedRole</td><td>(none)</td></tr> <tr> <td>Integration_AService</td><td>(none)</td></tr> <tr> <td>Modifier</td><td>none</td></tr> <tr> <td>PlayedRole</td><td>(none)</td></tr> <tr> <td>Priority</td><td>1</td></tr> <tr> <td>Service</td><td>Begin</td></tr> <tr> <td>Transaction</td><td>(none)</td></tr> <tr> <td>TransactionModifier</td><td>In</td></tr> </tbody> </table>	Properties		StateHasState		AspectPlayedRole		AspectService	Begin	AspectTransaction		Color	Black	Condition		DashStyle	Solid	IA_PlayedRole	(none)	IntAsp_PlayedRole	(none)	Integration_AService	(none)	Modifier	none	PlayedRole	(none)	Priority	1	Service	Begin	Transaction	(none)	TransactionModifier	In
Properties																																			
StateHasState																																			
AspectPlayedRole																																			
AspectService	Begin																																		
AspectTransaction																																			
Color	Black																																		
Condition																																			
DashStyle	Solid																																		
IA_PlayedRole	(none)																																		
IntAsp_PlayedRole	(none)																																		
Integration_AService	(none)																																		
Modifier	none																																		
PlayedRole	(none)																																		
Priority	1																																		
Service	Begin																																		
Transaction	(none)																																		
TransactionModifier	In																																		
Transformation																																			
Description																																			
<p>This pattern details how to generate the C# code from the protocol of an aspect. Specifically, it only generates the set of states that compose the protocol.</p>																																			
Template																																			
<pre> ... #> enum protocolStates { <#CommaSeparatedNames (aspect.Protocol.States) #> } protocolStates state; private protocolStates State{ get { return state;} set { state=value; this.StateName=state.ToString(); } } </pre>																																			

<#	...
Case Study	
Description	
<p>This pattern is illustrated using the protocol of the aspect <i>FJoint</i> of the TeachMover architectural model. The representation of the states of the protocol in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>	
Graphical representation	
	
Result of the pattern execution	
<pre>... enum protocolStates { FJOINT, POS, END, SubStateNewPos, SubStateNotify, SubStateQuery } protocolStates state; private protocolStates State{ get { return state;} set { state=value; this.StateName=state.ToString(); } } ...</pre>	
Related Patterns	
Pattern 2.	

A.2.3. Services

A.2.3.1. *Begin*

Pattern 5: Begin Services	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
Pattern	
Description	
<p>This pattern details how to generate the C# code from the service <i>Begin</i> of an aspect. This service acts as the constructor of the aspect. This generation follows four steps: 1) To establish the initial state of the aspect, 2) To generate the code of the valuations associated to the <i>Begin</i> service, 3) To generate the information of the played_roles of the aspect adding the priority to each service that compose them., 4) to indicate which is the state that is reached after the <i>Begin</i> execution. This pattern does not generate the code for the second step because the valuations of all services are generated by other pattern.</p>	
Template	
<pre> ... <# foreach (AService service in aspect.AServices) { if (service is Begin) { #> public <#=aspect.Name#> (<#=CommaSeparatedArguments(service.Arguments) #>) : base("<#=aspect.Name#>") { </pre>	

```

    State = protocolStates.<#=service.StateHasState[0].Source.Name#>;

<#
    /* Valuations */
    /* PlayedRoles */

    foreach (PlayedRole playedRole in aspect.PlayedRoles)
    {
#>
        PlayedRoleClass <#=playedRole.Name#> = new
        PlayedRoleClass("<#=playedRole.Name#>");

<#
        bool ServiceIn;
        foreach (IService servic in playedRole.Interface.IServices)
        {
            ServiceIn=false;
            foreach (StateHasState stateHasState in playedRole.StateHasState)
            {
                if(stateHasState.Service.Name == servic.Name)
                {
                    if(stateHasState.Modifier == TransitionModifier.In)
                    {
                        ServiceIn=true;
                        break;
                    }
                }
            }
            /* End if(stateHasState.Modifier == TransitionModifier.In)*/
        } /* End foreach (StateHasState stateHasState in
        playedRole.StateHasState) */
        /* End foreach (IService servic in playedRole.Interface.IServices)*/
#>

        <#=playedRole.Name#>.AddMethod("<#=servic.Name#>",<#=ServiceIn.ToString().ToLower()#>);

<#
    }
#>
    this.playedRoleList.Add(<#=playedRole.Name#>);

<#
    } /* End foreach (PlayedRole playedRole in aspect.PlayedRoles)*/
#>
    this.stateList=new ArrayList();

<#
    foreach(Microsoft.VisualStudio.Modeling.NamedElement element in
    aspect.Protocol.States)
    {
#>
        this.stateList.Add("<#=element.Name#>");

<#
    }
    foreach (PlayedRole playedRole in aspect.PlayedRoles)
    {
        foreach(StateHasState stateHasState in playedRole.StateHasState)

```

```

{
#>
AddPriorityService(protocolStates.<#=#stateHasState.Source.Name#>.ToString(),
<#=#playedRole.Name#>.PlayedRoleName,<#=#stateHasState.Service.Name#>,
<#=#stateHasState.Priority#>);

<#
}
#>
State = protocolStates.<#=#service.StateHasState[0].Target.Name#>;
<#
}/* endif (service is Begin) */
...

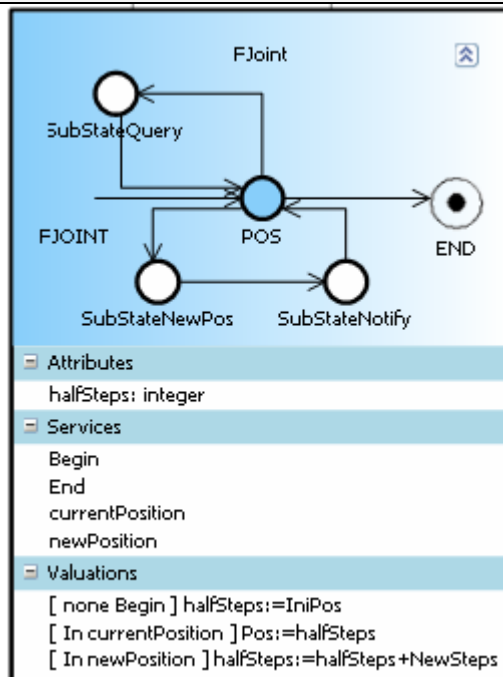
```

Case Study

Description

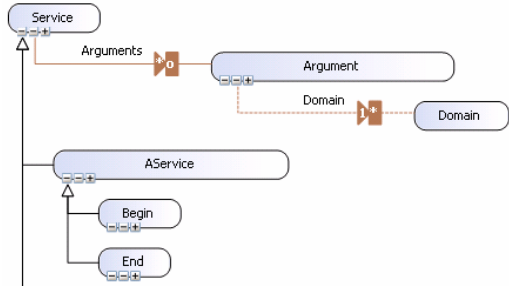
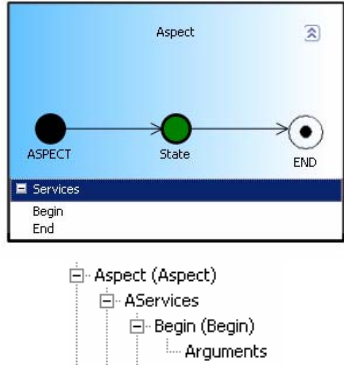
This pattern is illustrated using the service *Begin* of the aspect *FJoint* of the TeachMover architectural model. The representation of the *Begin* of the *FJoint* in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.

Graphical representation



Result of the pattern execution
<pre> ... public FJoint(int IniPos) : base("FJoint") { State = protocolStates.FJOINT; halfSteps=IniPos; PlayedRoleClass UPPOS = new PlayedRoleClass("UPPOS"); UPPOS.AddMethod("newPosition", true); this.playedRoleList.Add(UPPOS); PlayedRoleClass QPOS = new PlayedRoleClass("QPOS"); QPOS.AddMethod("currentPosition", true); this.playedRoleList.Add(QPOS); this.stateList=new ArrayList(); this.stateList.Add("FJOINT"); this.stateList.Add("POS"); this.stateList.Add("END"); this.stateList.Add("SubStateNewPos"); this.stateList.Add("SubStateNotify"); this.stateList.Add("SubStateQuery"); AddPriorityService(protocolStates.POS.ToString(), UPPOS.PlayedRoleName, "newPosition", 1); AddPriorityService(protocolStates.SubStateNewPos.ToString(), QPOS.PlayedRoleName, "currentPosition", 1); AddPriorityService(protocolStates.SubStateNotify.ToString(), QPOS.PlayedRoleName, "currentPosition", 1); AddPriorityService(protocolStates.POS.ToString(), QPOS.PlayedRoleName, "currentPosition", 1); AddPriorityService(protocolStates.SubStateQuery.ToString(), QPOS.PlayedRoleName, "currentPosition", 1); State = protocolStates.POS; } ... </pre>
Related Patterns
Pattern 2, Pattern 9, and Pattern 10.

A.2.3.2. *Public Services*

Pattern 6:Public Services	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
	
Transformation	
Description	
<p>This pattern details how to generate the C# code from a service of an aspect that is published by one of the interfaces that the aspect imports. Specifically, it only generates the head of the service and if the services is IN or OUT.</p>	
Template	
<pre> ... <# foreach (AService service in aspect.AServices) { if (!(service is Begin)) { ... }/* endif (service is Begin) */ else { if (!(service is End)) { ... } } } #> public AsyncResult <#=service.Name#>(<#=CommaSeparatedArguments(service.Arguments)#>) { <# </pre>	

```

        SortedList parameters=new SortedList();
        foreach (Argument element in service.Arguments)
        {
            parameters.Add(element.Name,null);
        }

        if(service.Modifier != AServiceModifier.none)
        {
#>
            // Modo In
            if(ServiceIn)
            {
<#
            }

            /* Checking if the state of the aspect is correct for the service
               execution */

            /* Preconditions*/

            /* Valuations */

            /* Constraints*/

            /* Execution of a service sequence of the protocol */

            /* Update of the state of the protocol*/

            if(service.Modifier != AServiceModifier.none)
            {
#>
            } //End modo IN
            // Modo Out
            else
            {
<#
            }

            if(service.Modifier == AServiceModifier.Out ||
               service.Modifier == AServiceModifier.InOut )
            {

                /* Valuations */

#>
                return
                CallOutService(this.interfaceName_ServiceOut,this.playedRoleName_ServiceOut,
                "<#=service.Name#>",this.aspectStateCareTaker.ActiveTransaction,
                <#=CommaSeparatedNames(service.Arguments)#>);

<#
            }
            else if(service.Modifier == AServiceModifier.In)
            {
#>
                throw new Exception("This method doesn't have Service mode Out");

```



```

<#
}

    /* end if (!(service is End)) */
    /* endelse (service is Begin) */
    /*endforeach (AService service in aspect.AServices) */
#>
...

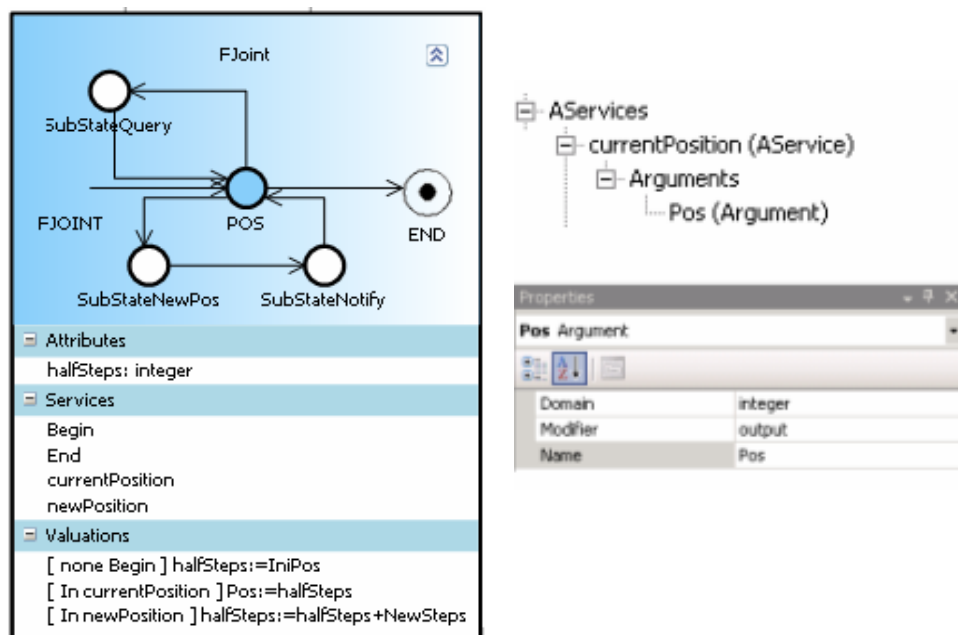
```

Case Study

Description

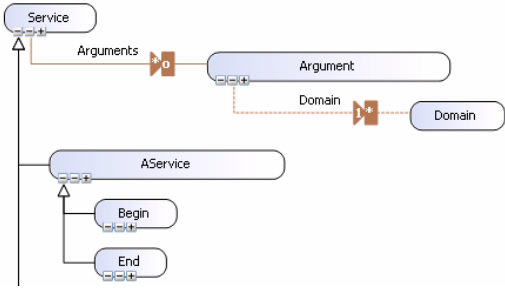
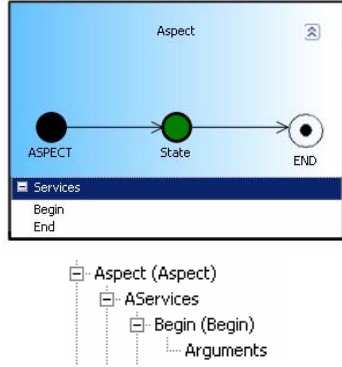
This pattern is illustrated using the public service *currentPosition* of the aspect *FJoint* of the TeachMover architectural model. The representation of the *currentPosition* in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.

Representación



Result of the pattern execution
<pre> ... public AsyncResult currentPosition(ref int Pos) { // Modo In if(ServiceIn) { /* Checking if the state of the aspect is correct for the service execution*/ /* Preconditions*/ /* Valuations */ /* Constraints*/ /* Execution of a service sequence of the protocol*/ /* Update of the state of the protocol */ } // Modo Out else { return CallOutService(this.interfaceName_ServiceOut,this.playedRoleName_ServiceOut, "currentPosition",this.aspectStateCareTaker.ActiveTransaction,Pos); } } ... </pre>
Related Patterns
<p>Pattern 2, Pattern 8, Pattern 9, Pattern 10, Pattern 11, Pattern 12,and Pattern 13.</p>

A.2.3.3. *Private Services*

Pattern 7: Private Services	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
	
Transformation	
Description	
This pattern details how to generate the C# code from a private service of an aspect.	
Template	
<pre> ... <# foreach (AService service in aspect.AServices) { if (!(service is Begin)) { ... }/* endif (service is Begin) */ else { if (!(service is End)) { if(service.Modifier == AServiceModifier.none) { ... } } } } #> public delegate AsyncResult <#=service.Name#>Delegate (<#=CommaSeparatedArguments(service.Arguments) #>); <# }//End if(service.Modifier != AServiceModifier.none) </pre>	

```
#>

public AsyncResult
<#=service.Name#>(<#=CommaSeparatedArguments(service.Arguments)#>)
{
    <#
        SortedList parameters=new SortedList();
        foreach (Argument element in service.Arguments)
        {
            parameters.Add(element.Name,null);
        }

        /* Checking if the state of the aspect is correct for the service
           execution */

        /* Preconditions*/

        /* Valuations */

        /* Constraints*/

        /* Execution of a service sequence of the protocol */

        /* Update of the state of the protocol*/

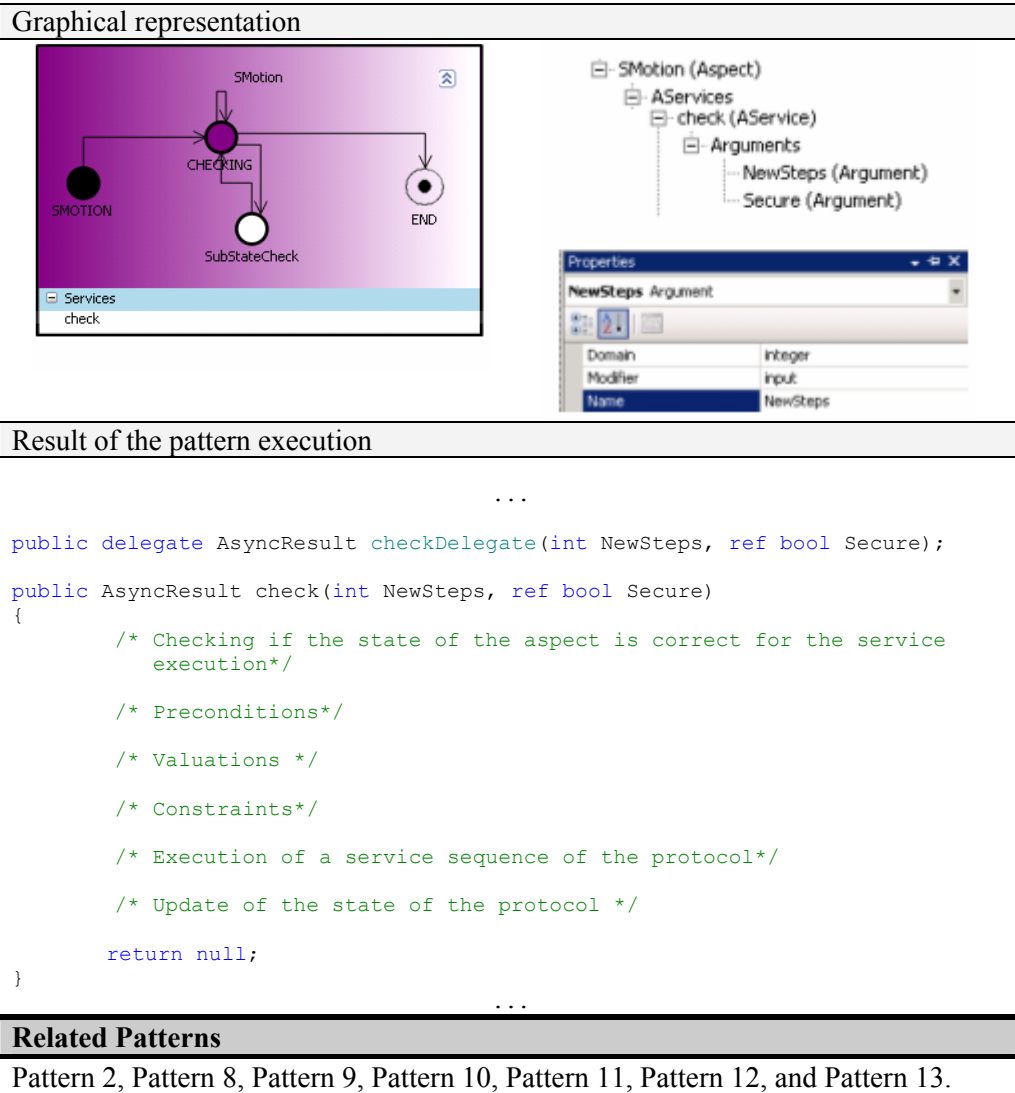
        if(service.Modifier == AServiceModifier.none)
        {
            #>
                return null;
            <#
        }
    }/* end if (!(service is End))*/
}/* endelse (service is Begin) */
}/*endforeach (AService service in aspect.AServices)*/
#>

...
```

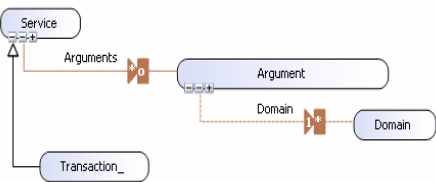
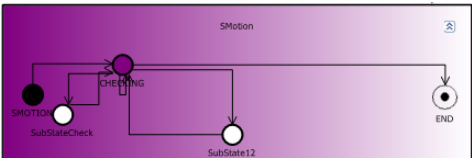
Case Study

Description

This pattern is illustrated using the private service *check* of the aspect *SMotion* of the TeachMover architectural model. The representation of the *check private service* in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.



A.2.3.4. Transactions

Pattern 8:Transactions	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
	 <div><div>Attributes</div><div>minimum: integer maximum: integer checkPos: integer</div><div>Services</div><div>Begin End currentPosition check controlSpeed</div><div>Valuations</div><div>[none Begin] minimum:= InitMin [none Begin] maximum:= InitMax [none Begin] checkPos:= InitPos [In currentPosition] checkPos:= Pos { (NewSteps+checkPos) >= minimum AND (NewSteps+checkPos) <= maximum } [none check] Secure... { (NewSteps+checkPos) < minimum OR (NewSteps+checkPos) > maximum } [none check] Secure:= fa...</div><div>Preconditions</div><div>Constraints</div><div>PlayedRoles</div><div>QUERYPOS</div><div>Transactions</div><div>dangerousChecking</div></div>
Transformation	
Description	
<p>This pattern details how to generate the C# code from a transaction. This generation is composed of three steps: 1) To generate the transaction definition, 2) To add the services that the transaction that it is composed of, 3) To obtain the sequence of execution.</p>	
Plantilla	
<pre>... <# foreach (DSIC.ISSI.PrismaDSL.DomainModel.Transaction_ transaction in aspect.Transactions) { #> public AsyncResult <#:=transaction.Name#> (<#:=CommaSeparatedArguments(transaction.Arguments) #>) { // Modo In if (ServiceIn) { <# /* Checking if the state of the aspect is correct for the service</pre>	

```

        execution */

        /* Preconditions*/

        /* Valuations */

        /* Constraints*/

        /* Execution of a service sequence of the protocol */

        /* Update of the state of the protocol*/

/* invocation of the transaction sequence */

foreach (AService serviceTrans in aspect.AServices)
{
    foreach(StateHasState stateHasState in serviceTrans.StateHasState)
    {
        if(stateHasState.Transaction != null && stateHasState.Transaction.Name
== transaction.Name)
        {
            if (!(stateHasState.Source is SubState))
            {
                if (stateHasState.Target is SubState)
                {
                    #>
                        if (state == protocolStates.<#stateHasState.Source.Name#>)
                        {
                            aspectStateCareTaker.StartTransaction();
                            try
                            {
                                <#
                                    /* Processing of the first service of the transaction */
                                    if(stateHasState.Condition != String.Empty)
                                    {
                                        #>
                                            if(<#stateHasState.Condition#>)
                                            {
                                                <#
                                                    }
                                                    /* Invocation of an OUT service */
                                                    if(stateHasState.Modifier == TransitionModifier.Out)
                                                    {
                                                        #>
                                                            InvokeOutService("<#stateHasState.PlayedRole.Interface.Name#>", "<#stateHasState.PlayedRole.Name#>",
                                                            "<#stateHasState.Service.Name#>", this.aspectStateCareTaker.ActiveTransaction
                                                            ,
                                                            <#CommaSeparatedNames(stateHasState.Service.Arguments)#>);
                                                            <#
                                                                }
                                                                /* Invocation of a private service of the aspect */
                                                                if(stateHasState.PlayedRole==null)
                                                                {
                                                                    #>

```

```

<#=stateHasState.Service.Name#> (<#=CommaSeparatedNames (stateHasState.Service.
Arguments) #>);
<#
    }
    /* Execution of a service IN because as a private
       service because it is associate to a transition of
       the protocol */
    if (stateHasState.Modifier == TransitionModifier.In)
    {
#>

<#=stateHasState.Service.Name#> (<#=CommaSeparatedNames (stateHasState.Service.
Arguments) #>);

<#
    }
#>
    aspectStateCareTaker.CheckConsistence;
<#
    if (stateHasState.Condition!="")
    {
#>
    }
<#
    }
#>

    /* Update of the state */
    state = protocolStates.<#=stateHasState.Target.Name#>;
<#
    /* Processing of the rest of the services that the transaction is
       composed of*/
    SubState subState = stateHasState.Target as SubState;
    System.Collections.IList substateLinks=
subState.GetElementLinks(subState.Source.TargetRole.Id);
    StateHasState subStateHasState=null;
    int i=0;
    while( i<substateLinks.Count)
    {
        if (substateLinks[i] is StateHasState)
        {
            subStateHasState=(StateHasState)substateLinks[i];
            if (subStateHasState.Condition != "")
            {
#>
                if (<#=subStateHasState.Condition#>)
                {
<#
                    }
                    /* Invocation of an OUT service */
                    if (subStateHasState.Modifier == TransitionModifier.Out)
                    {
#>
                        InvokeOutService("<#=subStateHasState.PlayedRole.Interface.Name#>",
"<#=subStateHasState.PlayedRole.Name#>", "<#=subStateHasState.Service.Name#>",
this.aspectStateCareTaker.ActiveTransaction,
<#=CommaSeparatedNames (subStateHasState.Service.Arguments) #>);
<#
                    }

```



```

        /* Invocation of a private service of the aspect */
        if(subStateHasState.PlayedRole==null)
        {
#>

<#=#subStateHasState.Service.Name#>(<#=#CommaSeparatedNames(subStateHasState.Service.Arguments)#>);

<#
        }
        /* Execution of a service IN because as a private service
        because it is associate to a transition of the protocol */

        if(subStateHasState.Modifier == TransitionModifier.In)
        {
#>

<#=#subStateHasState.Service.Name#>(<#=#CommaSeparatedNames(subStateHasState.Service.Arguments)#>);

<#
        }
#>
        aspectStateCareTaker.CheckConsistence;
<#
        if(subStateHasState.Condition!="")
        {
#>
        }

<#
        }
#>
        state = protocolStates.<#=#subStateHasState.Target.Name#>;

<#
        i++;

        if(subStateHasState.Target is SubState)
        {
            subState = subStateHasState.Target as SubState;
            substateLinks=
            subState.GetElementLinks(subState.Source.TargetRole.Id);
            i=0;
        }
        } /*End if(substateLinks[i] is StateHasState)*/
    } /*End While(i<substateLinks.Count) */
    } /* End stateHasState.Target is SubState */
#>
    }
    catch
    {
        aspectStateCareTaker.SetConsistecy(false);
    }
    finally
    {
        aspectStateCareTaker.EndTransaction();
    }
}
}

```

```

<#
    }
    }
    }
    }
    availableStates.Clear();
    }/*endforeach transaction*/
#>
return null;
}
...

```

Case Study

Descripción

This pattern is illustrated using the transaction *dangerousChecking* of the TeachMover architectural model. The representation of the *dangerousChecking* in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.

Graphical representation

Transactions

dangerousChecking

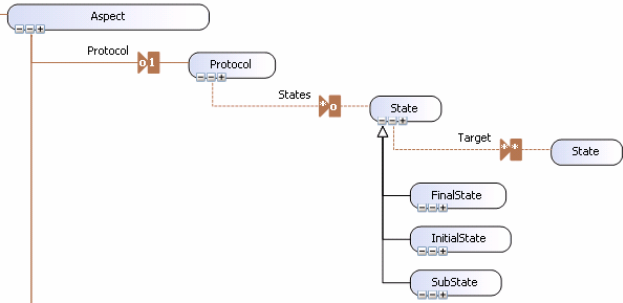
AspectPlayedRole	QUERYPOS
AspectService	controlSpeed
AspectTransaction	dangerousChecking
Color	Black
Condition	
DashStyle	Solid
IA_PlayedRole	(none)
IntAsp_PlayedRole	(none)
Integration_AServic	(none)
Modifier	In
PlayedRole	QUERYPOS
Priority	1
Service	controlSpeed
Transaction	dangerousChecking
TransactionModifier	In

Result of the pattern execution
<pre> public AsyncResult dangerousChecking() { // Modo In if(ServiceIn) { if(state != protocolStates.CHECKING) throw new InvalidProtocolStateException("SMotion","dangerousChecking"); if (state == protocolStates.CHECKING) { aspectStateCareTaker.StartTransaction(); try { controlSpeed(Steps, CurrentSpeed, Secure); aspectStateCareTaker.CheckConsistence; state = protocolStates.SubState12; InvokeOutService("IQueryPos","QUERYPOS","controlSpeed",this.aspectStat eCareTaker.ActiveTransaction,Steps, CurrentSpeed, Secure); aspectStateCareTaker.CheckConsistence; state = protocolStates.CHECKING; } catch { aspectStateCareTaker.SetConsistecy(false); } finally { aspectStateCareTaker.EndTransaction(); } } } } </pre>
Related Patterns
Pattern 2, Pattern 8, Pattern 9, Pattern 10, Pattern 11, Pattern 12, and Pattern 13.

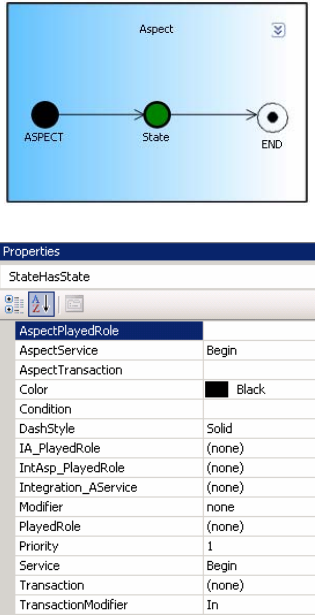
A.2.3.5. *Checking The Service Execution*

Pattern 9: Checking The Service Execution

PRISMA Metamodel in DSL Tools



Graphical Metaphor



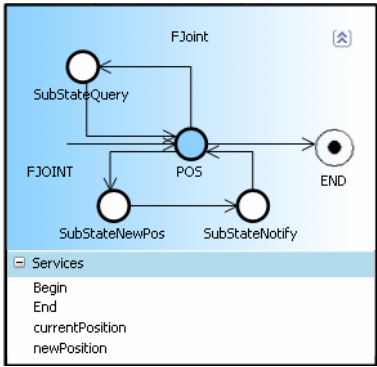
Transformation

Description

This pattern details how to generate the C# code to check if the state of the protocol allows the execution of a service.

Template

```
<#
bool isFirst = true;
System.Collections.SortedList availableStates=new
System.Collections.SortedList();
/* Tratamiento de estado correcto para los servicios*/
foreach(StateHasState stateHasState in service.StateHasState)
{
    if (isFirst)
    {
        isFirst = false;
        availableStates.Add(stateHasState.Source.Name,null);
    }
}
```

<pre> #> if(state != protocolStates.<#-stateHasState.Source.Name#> <# } else if(!availableStates.Contains(stateHasState.Source.Name)) { #> && state != protocolStates.<#-stateHasState.Source.Name#> <# availableStates.Add(stateHasState.Source.Name,null); } #>) throw new InvalidProtocolStateException("<#-aspect.Name#>","<#-service.Name#>"); <# ... </pre>
Case Study
Description
<p>This pattern is illustrated using the service <i>currentPosition</i> of the aspect <i>FJoint</i> of the TeachMover architectural model.</p>
Graphical representation

Result of the pattern execution
<pre> ... public AsyncResult currentPosition(ref int Pos) { ... if(state != protocolStates.SubStateNewPos && state != protocolStates.SubStateNotify </pre>

<pre> && state != protocolStates.POS && state != protocolStates.SubStateQuery) throw new InvalidProtocolStateException("FJoint", "currentPosition"); </pre>
Related Patterns
Pattern 6, Pattern 7, Pattern 8, and Pattern 14.

A.2.4. Preconditions

Pattern 10: Preconditions									
PRISMA Metamodel in DSL Tools	Graphical Metaphor								
<pre> classDiagram class Aspect { +List<Precondition> Preconditions } class Precondition { +Service Service } class AService Aspect "1" -- "0..*" Precondition : Preconditions Precondition "1" -- "0..*" AService : Service </pre> <p>Precondition (Class)</p> <ul style="list-style-type: none"> Condition (Value property) ConditionAndService (Value property) AspectService (Value property) Aspect (Reference property) Service (Reference property) 	<p>Properties</p> <p>Precondition1 Precondition</p> <table border="1"> <tr> <td>AspectService</td> <td></td> </tr> <tr> <td>Condition</td> <td></td> </tr> <tr> <td>Name</td> <td></td> </tr> <tr> <td>Service</td> <td>(none)</td> </tr> </table>	AspectService		Condition		Name		Service	(none)
AspectService									
Condition									
Name									
Service	(none)								
Transformation									
Description									
<p>This pattern details how to generate the C# code from a precondition.</p>									

Template

```

...

/* Comprobación de las Precondiciones*/
foreach(Precondition precondition in service.Precondition)
{
    #>
    if (!(<#precondition.Condition.Replace("=", "==").Replace("<>", "!=").
        Replace("<=", "<=").Replace(">=", ">=").Replace("and", "&&").
        Replace("or", "||").Replace("AND", "&&").Replace("OR", "||") #>))
    throw new
    InvalidPreconditionException("<#aspect.Name#>", "<#service.Name#>");
    <#
}

...

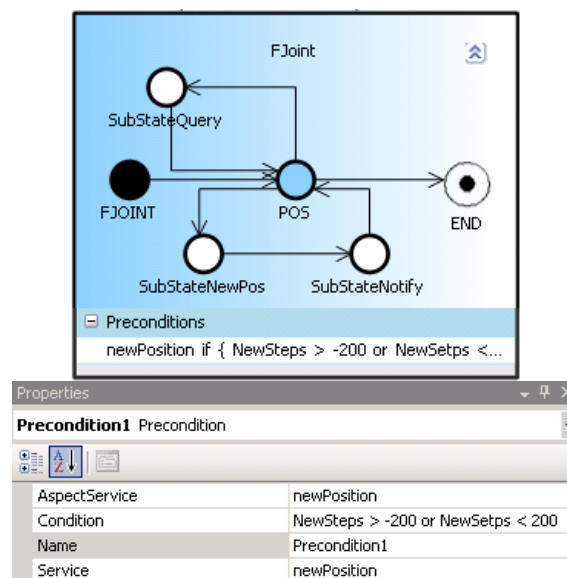
```

Case Study

Description

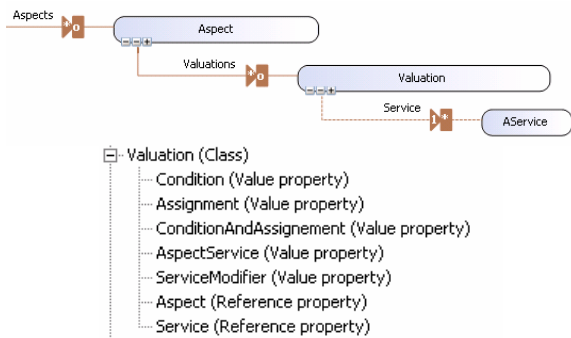
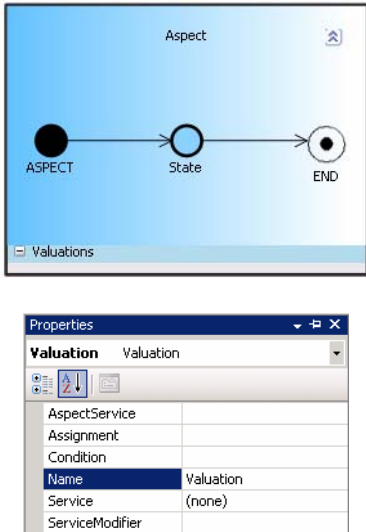
This pattern is illustrated using a precondition associated to the service *newPosition* of the aspect *FJoint* of the TeachMover architectural model. The representation of the precondition in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.

Graphical representation



Result of the pattern execution
<pre> ... public AsyncResult newPosition(int NewSteps) { ... if (!(NewSteps > -200 NewSteps < 200)) throw new InvalidPreconditionException("FJoint", "newPosition"); ... } ... </pre>
Related Patterns
Pattern 2, Pattern 5, Pattern 6, and Pattern 7.

A.2.5. Valuations

Pattern 11: Valuations	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
 <pre> classDiagram class Aspects class Aspect class Valuations class AService class Valuation { Condition Value property Assignment Value property ConditionAndAssignment Value property AspectService Value property ServiceModifier Value property Aspect Reference property Service Reference property } Aspects --> Aspect Aspect --> Valuations Valuations --> AService Valuation --> AService </pre>	
Transformation	
Description	
This pattern details how to generate the C# code from a valuation.	


```

Template
...

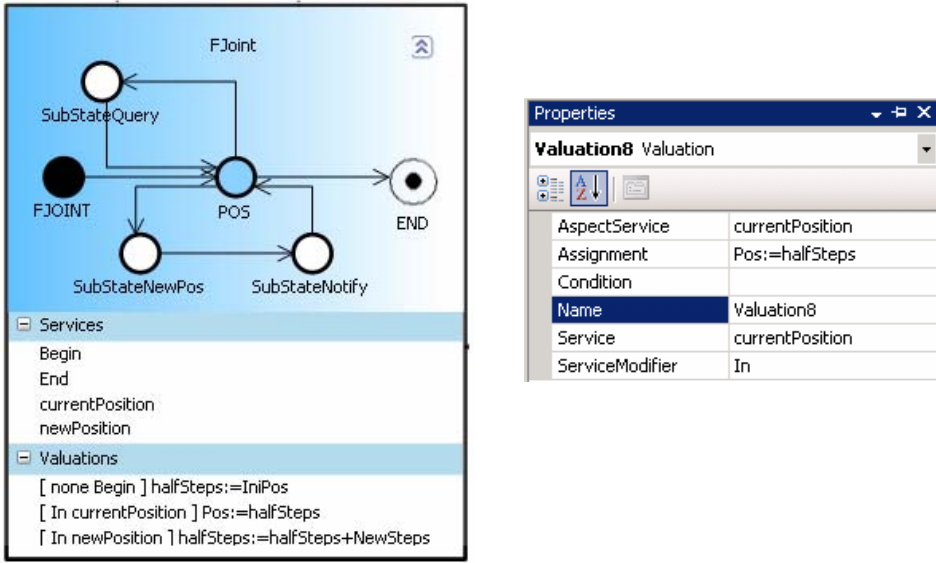
/* Valuations */

foreach(Valuation valuation in service.Valuations)
{
    if(valuation.ServiceModifier != AServiceModifier.Out)
    {
        if (valuation.Condition == String.Empty)
        {
            foreach(string valuationItem in
valuation.Assignment.Replace(":", "=").Split(', '))
            {
                string Item=valuationItem;
                if(Item.StartsWith(" "))
                    Item=Item.Substring(1);

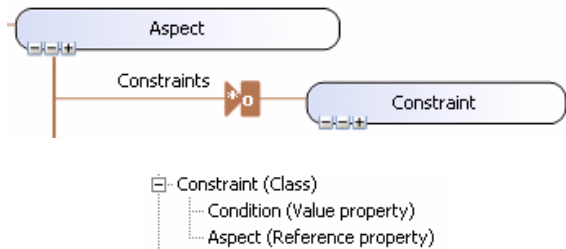
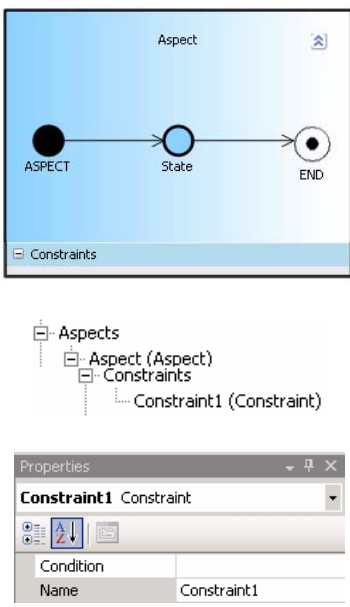
                <#Item#>;
            }
        }
        else
        {
            if
            (<#=valuation.Condition.Replace("=", "==").Replace("<>", "!=").
Replace("<==", "<=").Replace(">=", ">=").Replace("and", "&&").
Replace("or", "||").Replace("AND", "&&").Replace("OR", "||")#>)
            {
                foreach(string valuationItem in
valuation.Assignment.Replace(":", "=").Split(', '))
                {
                    string Item=valuationItem;
                    if(Item.StartsWith(" "))
                        Item=Item.Substring(1);

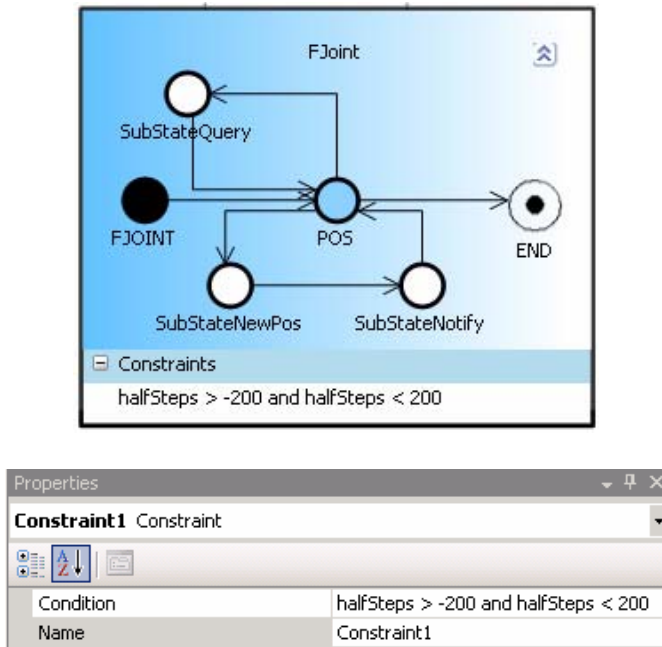
                    <#Item#>;
                }
            }
        }
    }
}
...

```

Case Study													
Description													
<p>This pattern is illustrated using a valuation of the service <i>currentPosition</i> of the aspect <i>FJoint</i> of the TeachMover architectural model. The representation of this valuation in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>													
Graphical representation													
 <p>The diagram shows a state machine for <i>FJoint</i> with states: <i>FJOINT</i> (initial), <i>SubStateQuery</i>, <i>SubStateNewPos</i>, <i>SubStateNotify</i>, and <i>END</i>. Transitions are labeled with events and actions. Below the diagram is a list of services and valuations.</p> <p>Services:</p> <ul style="list-style-type: none"> Begin End currentPosition newPosition <p>Valuations:</p> <ul style="list-style-type: none"> [none Begin] halfSteps:=IniPos [In currentPosition] Pos:=halfSteps [In newPosition] halfSteps:=halfSteps+NewSteps <p>The Properties window for Valuation8 shows:</p> <table border="1"> <tr> <td>AspectService</td><td>currentPosition</td></tr> <tr> <td>Assignment</td><td>Pos:=halfSteps</td></tr> <tr> <td>Condition</td><td></td></tr> <tr> <td>Name</td><td>Valuation8</td></tr> <tr> <td>Service</td><td>currentPosition</td></tr> <tr> <td>ServiceModifier</td><td>In</td></tr> </table>		AspectService	currentPosition	Assignment	Pos:=halfSteps	Condition		Name	Valuation8	Service	currentPosition	ServiceModifier	In
AspectService	currentPosition												
Assignment	Pos:=halfSteps												
Condition													
Name	Valuation8												
Service	currentPosition												
ServiceModifier	In												
Result of the pattern execution													
<pre> ... public AsyncResult currentPosition(ref int Pos) { ... Pos=halfSteps; ... } ... </pre>													
Related Patterns													
Pattern 2, Pattern 5, Pattern 6, and Pattern 7.													

A.2.6. Constraints

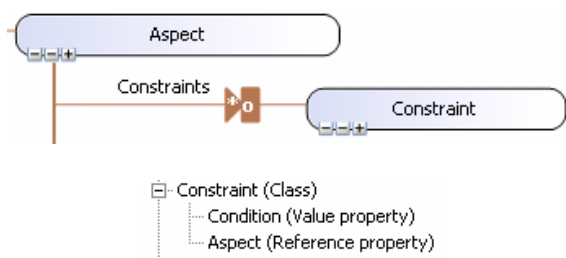
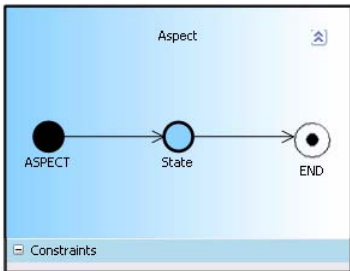
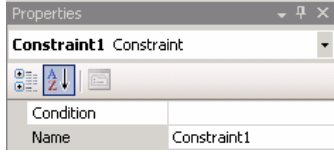
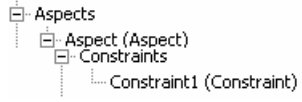
Pattern 12: Constraints	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
	
Transformation	
Description	
This pattern details how to generate the C# code from a constraint.	
Template	
<pre> ... /* Comprobación de las Restricciones de Integridad*/ foreach (Constraint constraint in aspect.Constraints) { #> if (!(<#=constraint.Condition.Replace("=", "==").Replace("<>", "!="). Replace("<==", "<=").Replace(">=", ">=").Replace("and", "&&"). Replace("or", " ").Replace("AND", "&&").Replace("OR", " ") #>)) throw new InvalidIntegrityConstraintException("<#=aspect.Name#>", "<#=service.Name#>"); <# } ... </pre>	

Case Study				
<p>Description</p> <p>This pattern is illustrated using a constraint of the aspect <i>FJoint</i> of the TeachMover architectural model. The representation of this constraint in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>				
<p>Graphical representation</p>  <p>The diagram shows a state machine for the <i>FJoint</i> aspect. It includes states: <i>FJOINT</i> (initial), <i>SubStateQuery</i>, <i>SubStateNewPos</i>, <i>SubStateNotify</i>, <i>POS</i>, and <i>END</i> (final). Transitions are: <i>FJOINT</i> to <i>SubStateQuery</i>, <i>SubStateQuery</i> to <i>POS</i>, <i>FJOINT</i> to <i>SubStateNewPos</i>, <i>SubStateNewPos</i> to <i>POS</i>, <i>SubStateNotify</i> to <i>POS</i>, and <i>POS</i> to <i>END</i>. A constraint is applied to the <i>POS</i> state.</p> <p>Properties</p> <p>Constraint1 Constraint</p> <table border="1"> <tr> <td>Condition</td> <td>halfSteps > -200 and halfSteps < 200</td> </tr> <tr> <td>Name</td> <td>Constraint1</td> </tr> </table>	Condition	halfSteps > -200 and halfSteps < 200	Name	Constraint1
Condition	halfSteps > -200 and halfSteps < 200			
Name	Constraint1			
<p>Result of the pattern execution</p> <pre> ... public AsyncResult newPosition(int NewSteps) { ... if (!halfSteps > -200 && halfSteps < 200) throw new InvalidIntegrityConstraintException("FJoint", "newPosition"); ... } ... </pre>				

Related Patterns

Pattern 2, Pattern 6, Pattern 7, and Pattern 8.

A.2.7.State of the Protocol

Pattern 13:State of the Protocol	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
 <pre> classDiagram class Aspect class Constraint { Condition Value property Aspect Reference property } Aspect "1" -- "0..*" Constraint : Constraints </pre>	 <pre> graph LR ASPECT((ASPECT)) --> State((State)) State --> END(((END))) </pre>
	
	
Transformation	
Description	
This pattern details how to generate the C# code that permits the change of state after a service execution.	
Template	
<pre> ... /* Update of the state of the protocol */ foreach(StateHasState stateHasState in service.StateHasState) { if (!(stateHasState.Source is SubState) && (stateHasState.Transaction == null && (IsSequence > 0))) { if (stateHasState.Target is SubState) </pre>	

```

    {
        SubState subState = stateHasState.Target as SubState;
        System.Collections.IList substateLinks=
            subState.GetElementLinks(subState.Source.TargetRole.Id);
        StateHasState subStateHasStateProtocol=null;
        int i=0;
        while( i<substateLinks.Count)
        {
            if(substateLinks[i] is StateHasState)
            {
                subStateHasStateProtocol=(StateHasState)substateLinks[i];
                i++;

                if(subStateHasStateProtocol.Target is SubState)
                {
                    subState = subStateHasStateProtocol.Target as SubState;
                    substateLinks=
subState.GetElementLinks(subState.Source.TargetRole.Id);
                    i=0;
                }
            } /*End if(substateLinks[i] is StateHasState)*/
        } /*End While(i<substateLinks.Count) */
    }
}

if (state == protocolStates.<#==stateHasState.Source.Name#>

<#
    if (stateHasState.Condition != "")
    {
        && <#==stateHasState.Condition#>

    }

)
state = protocolStates.<#==subStateHasStateProtocol.Target.Name#>;

}
else {
    if (state == protocolStates.<#==stateHasState.Source.Name#>

    <#
        if (stateHasState.Condition != "")
        {
            && <#==stateHasState.Condition#>

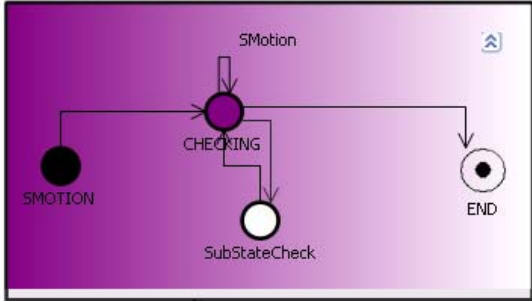
        }

    )
state = protocolStates.<#==stateHasState.Target.Name#>;

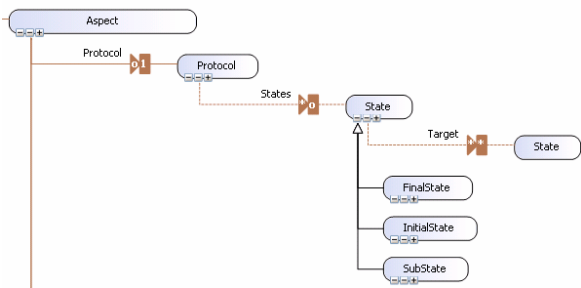
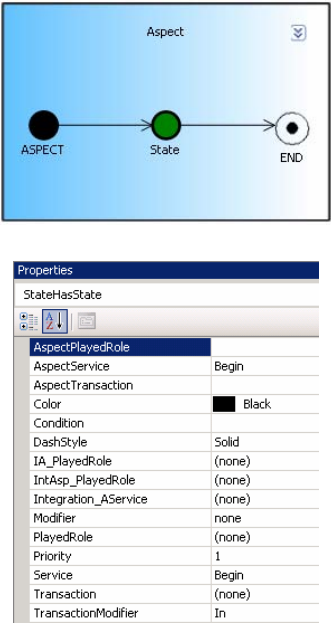
}
}
}

...

```

Case Study
<p>Description</p> <p>This pattern is illustrated using the change of state that generates the service <i>currentPosition</i> of the aspect <i>SMotion</i> of the TeachMover architectural model. The representation of this change of state in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p> <p>Como ejemplo de código generado para actualizar el estado de un aspecto tras ejecutar un servicio, se puede observar el resultado obtenido en el servicio “<i>currentPosition</i>” del aspecto “<i>SMotion</i>”.</p>
<p>Graphical representation</p>  <pre> stateDiagram-v2 [*] --> SMOTION SMOTION --> CHECKING CHECKING --> SubStateCheck SubStateCheck --> CHECKING CHECKING --> END </pre> <p>The diagram shows a state machine for the <i>SMotion</i> aspect. It starts at an initial state (black circle) and transitions to the <i>SMOTION</i> state (black circle). From <i>SMOTION</i>, it transitions to the <i>CHECKING</i> state (purple circle). The <i>CHECKING</i> state has a self-loop labeled <i>SubStateCheck</i> and a transition to the <i>END</i> state (white circle with a black dot). The <i>END</i> state is the final state of the machine.</p>
<p>Result of the pattern execution</p> <pre> ... public AsyncResult currentPosition(ref int Pos) { ... if (state == protocolStates.CHECKING) state = protocolStates.CHECKING; ... } ... </pre>
Related Patterns
<p>Pattern 2, Pattern 6, Pattern 7, and Pattern 8.</p>

A.2.8. Processing of a service sequence

Pattern 14: Processing of a service sequence	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
	
Transformation	
Description	
This pattern details how to generate the C# code from a service sequence of the protocol.	
Template	
<pre>... /* Processing of the executions of services that belong to a service sequence of the protocol */ int IsSequence=0; foreach(StateHasState stateHasState in service.StateHasState) { if(stateHasState.Transaction == null) { if (!(stateHasState.Source is SubState)) {</pre>	


```

        IsSequence++;
        if (stateHasState.Target is SubState)
        {
            IsSequence--;
            SubState subState = stateHasState.Target as SubState;
#>
            if (state == protocolStates.<#stateHasState.Source.Name#>)
            {
                state = protocolStates.<#stateHasState.Target.Name#>;
<#
                if (stateHasState.Condition != String.Empty)
                {
#>
                    if (<#stateHasState.Condition#>)
                    {
<#
                        }
                        System.Collections.IList substateLinks=
subState.GetElementLinks(subState.Source.TargetRole.Id);
                        StateHasState subStateHasState=null;
                        int i=0;
                        while( i<substateLinks.Count)
                        {
                            if(substateLinks[i] is StateHasState)
                            {
                                subStateHasState=(StateHasState) substateLinks[i];
                                if(subStateHasState.Condition != String.Empty)
                                {
#>
                                    if (<#subStateHasState.Condition#>)
                                    {
<#
                                        }
                                        foreach (Argument element in
subStateHasState.Service.Arguments)
                                        {
                                            if (!attributes.Contains(element.Name) &&
!parameters.Contains(element.Name))
                                            {
                                                parameters.Add(element.Name, null);
#>
                                                <#DomainToType(element.Domain)#> <#element.Name#>=0;
<#
                                            }
                                        }
                                        /* A service OUT is invoked*/
                                        if(subStateHasState.Modifier == TransitionModifier.Out)
                                        {
#>
                                            InvokeOutService("<#subStateHasState.PlayedRole.Interface.Name#>",
                                                "<#subStateHasState.PlayedRole.Name#>",
                                                "<#subStateHasState.Service.Name#>",
                                                this.aspectStateCareTaker.ActiveTransaction,
<#CommaSeparatedNames(subStateHasState.Service.Arguments)#>);
<#
                                            }
                                            /* A private service of the aspect is invoked */
                                            if(subStateHasState.PlayedRole==null)
                                            {

```

```

#>
    <#=<subStateHasState.Service.Name#>(
        <#=<CommaSeparatedNames4(subStateHasState.Service.Arguments)#>;

<#
    }
    /* Executes a service IN as a private service of the aspect
       because it is in a transition of the protocol */
    if(subStateHasState.Modifier == TransitionModifier.In)
    {
#>
        <#=<subStateHasState.Service.Name#>(
            <#=<CommaSeparatedNames4(subStateHasState.Service.Arguments)#>;
        <#
            }

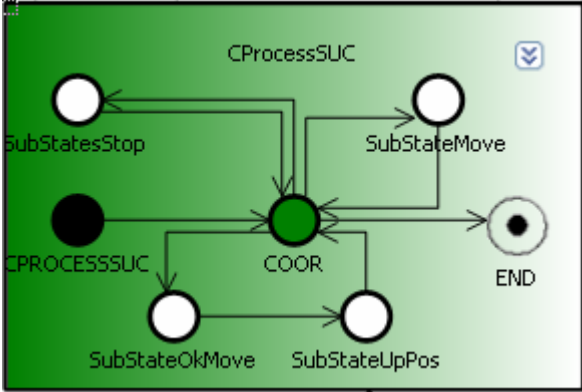
            if(subStateHasState.Condition != String.Empty)
            {
#>
            }
        <#
        }
    #>

    state=protocolStates.<#=<subStateHasState.Target.Name#>;
    <#
        i++;
        if(subStateHasState.Target is SubState)
        {
            subState = subStateHasState.Target as SubState;
            substateLinks=

subState.GetElementLinks(subState.Source.TargetRole.Id);
            i=0;
        }
    } /*End if(substateLinks[i] is StateHasState)*/
    } /*End While(i<substateLinks.Count) */

    if(stateHasState.Condition != String.Empty)
    {
#>
    }
    <#
    }
#>
    }
    <#
    }
    }
    }
    }
    }
    }
    ...

```

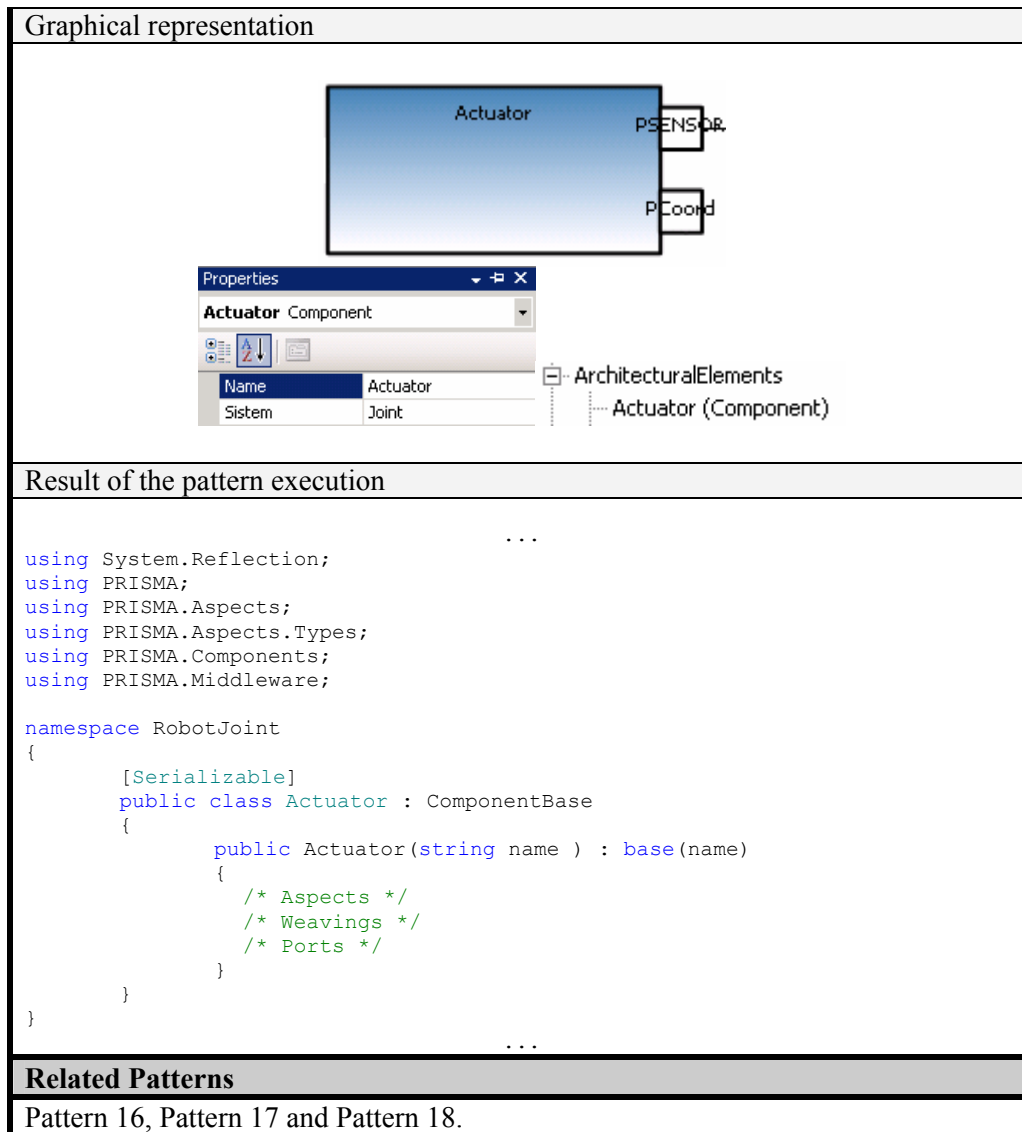
Case Study
<p>Description</p> <p>This pattern is illustrated using the sequence of services after the service <i>MoveOk</i> of the aspect <i>CProcessSUC</i> of the TeachMover architectural model. The representation of the sequence of services in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>
<p>Graphical representation</p>  <pre> stateDiagram-v2 [*] --> COOR COOR --> SubStatesStop COOR --> SubStateMove COOR --> SubStateOkMove COOR --> SubStateUpPos SubStatesStop --> COOR SubStateMove --> COOR SubStateOkMove --> COOR SubStateUpPos --> COOR COOR --> END </pre> <p>The diagram shows a state machine for <i>CProcessSUC</i>. It starts at an initial state (black dot) and transitions to a state labeled <i>COOR</i>. From <i>COOR</i>, there are four outgoing transitions labeled <i>SubStatesStop</i>, <i>SubStateMove</i>, <i>SubStateOkMove</i>, and <i>SubStateUpPos</i>, all of which lead back to the <i>COOR</i> state. Finally, there is a transition from <i>COOR</i> to an end state labeled <i>END</i>.</p>
<p>Result of the pattern execution</p> <pre> ... public AsyncResult moveOk() { // Modo In if(ServiceIn) { if(state != protocolStates.COOR && state != protocolStates.SubStateUpPos) throw new InvalidProtocolStateException("CProcessSUC", "moveOk"); if (state == protocolStates.COOR) { state = protocolStates.SubStateOkMove; int NewSteps=0; InvokeOutService("IUpdatePos", "UPDATEPOS", "newPosition", this.aspectStateCareTaker.ActiveTransaction, NewSteps); state=protocolStates.SubStateUpPos; InvokeOutService("IJoint", "JOINT", "moveOk", this.aspectStateCareTaker.ActiveTransaction, null); state=protocolStates.COOR; } } } </pre>

<pre> return null; } // Modo Out else { return CallOutService(this.interfaceName_ServiceOut, this.playedRoleName_ServiceOut, " moveOk", this.aspectStateCareTaker.ActiveTransaction, null); } } ... </pre>
Related Patterns
Pattern 2, Pattern 5, Pattern 6, and Pattern 7.

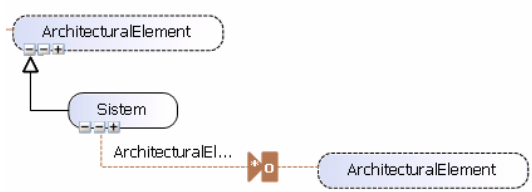
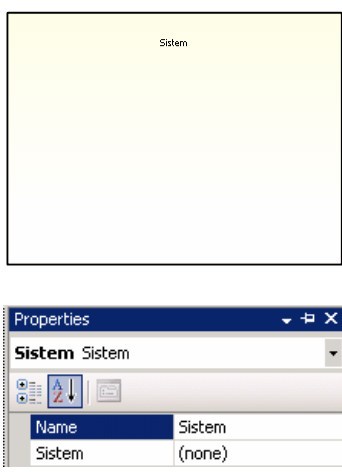
A.3. SIMPLE ARCHITECTURAL ELEMENTS: COMPONENTS AND CONNECTORS

Pattern 15: Simple Architectural Elements	
PRISMA metamodel in DSL Tools	Graphical Metaphor
Transformation	
Description	
<p>This pattern details how to generate the C# code from a simple architectural element. Specifically, it only generates the structure of the architectural model, the internal code of this structure, that is, ports, aspects and weaving, is generated by other patterns related to it.</p>	

Template
<pre> ... using System; using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types; using PRISMA.Components; using PRISMA.Middleware; namespace <#=this.Model.Name#> { <# foreach (ArchitecturalElement architecturalElement in this.Model.ArchitecturalElements) { if (architecturalElement is Component architecturalElement is Connector) { <# [Serializable] public class <#=architecturalElement.Name#> : ComponentBase <# if (architecturalElement is Connector) { <# , IConnector <# } <# { public <#=architecturalElement.Name#> (string name<#=ArchitecturalElementArguments(architecturalElement)#>) : base(name) { <# /* Aspects */ /* Weavings */ /* Ports */ <# } } <# }/* endif (architecturalElement is Component architecturalElement is Connector)*/ ... </pre>
Case Study
Description
<p>This pattern is illustrated using the component <i>Actuator</i> of the TeachMover case study. The representation of the <i>Actuator</i> in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>



A.4. COMPLEX ARCHITECTURAL ELEMENTS: SYSTEMS

Pattern 16: Systems	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
	
Transformation	
Description	
This pattern details how to generate the C# code from a system.	
Template	
<pre> using System; using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types; using PRISMA.Components; using PRISMA.Middleware; namespace <#=this.Model.Name#> { <# foreach (ArchitecturalElement architecturalElement in this.Model.ArchitecturalElements) { if (architecturalElement is Component architecturalElement is Conector) { ... } } ># } </pre>	

```

...
}
else
{
    if (architecturalElement is Sistem)
    {
        Sistem system = architecturalElement as Sistem;
    }
}
#>
[Serializable]
public class <#=architecturalElement.Name#> : SystemBase
{
    public <#=architecturalElement.Name#>(string
name<#=ArchitecturalElementArguments(architecturalElement)#>) : base(name)
{
    <#
        /* Aspects */
        /* Weavings */
        /* Ports */
    #>
}
}
<#
    }/*if (architecturalElement is Sistem)*/
}/* endforeach (ArchitecturalElement architecturalElement in
this.Model.ArchitecturalElements) */
#>
}
}
...

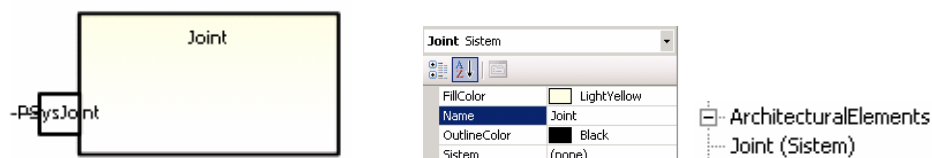
```

Case Study

Description

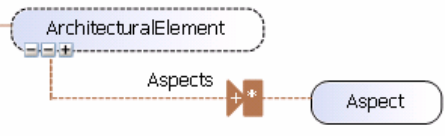
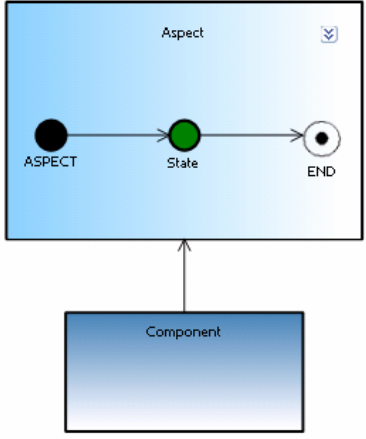
This pattern is illustrated using the system *Joint* of the TeachMover architectural model. The representation of the *Joint* in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.

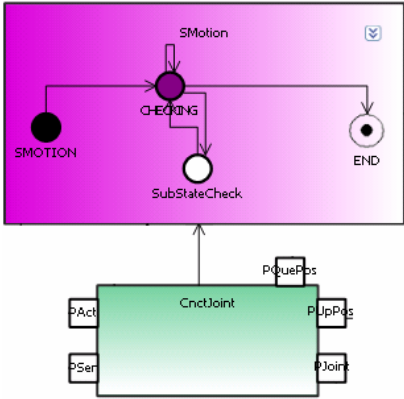
Graphical representation



Result of the pattern execution
<pre>using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types; using PRISMA.Components; using PRISMA.Middleware; namespace RobotJoint { [Serializable] public class Joint : SystemBase { public Joint(string name) : base(name) { /* Aspects */ /* Weavings */ /* Ports */ } } }</pre>
Related Patterns
Pattern 17, Pattern 18, and Pattern 19.

A.5. IMPORTATION OF ASPECTS FROM AN ARCHITECTURAL ELEMENT

Pattern 17: Aspects Importation	
PRISMA Metamodel in DSL Tools	Graphical Metaphor
 <p>The diagram shows a dashed box labeled 'ArchitecturalElement' containing a collection of 'Aspects' (represented by a red dashed line with a '+' sign). An arrow points from this collection to an 'Aspect' element (represented by a blue rounded rectangle).</p>	 <p>The diagram shows a state transition for an 'Aspect'. It starts with a black circle labeled 'ASPECT', followed by a green circle labeled 'State', and ends with a white circle labeled 'END'. An arrow points from a 'Component' box (blue rectangle) up to the 'State' circle. The entire transition is enclosed in a blue box labeled 'Aspect'.</p>
Transformation	
Description	
<p>This pattern details how to generate the C# code from the importation of aspects that makes an architectural element.</p>	
Template	
<pre> ... <# foreach (Aspect aspect in architecturalElement.Aspects) { AddAspect(new <#=aspect.Name#>(<#=AspectArguments(aspect)#>)); } /* endforeach (Aspect aspect in this.Model.Aspects) */ #> ... </pre>	

Case Study
<p>Description</p> <p>This pattern is illustrated using the importation of the aspect <i>SMotion</i> from the connector <i>CnctJoint</i> of the TeachMover architectural model. The representation of this aspect importation in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>
<p>Graphical representation</p> 
<p>Result of the pattern execution</p> <pre> ... using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types; using PRISMA.Components; using PRISMA.Middleware; namespace RobotJoint { [Serializable] public class CnctJoint : ComponentBase , IConnector { public CnctJoint(string name, int IniMin, int IniMax, int IniPos) : base(name) { AddAspect(new CProcessSUC()); AddAspect(new SMotion(IniMin, IniMax, IniPos)); ... } } } ... </pre>

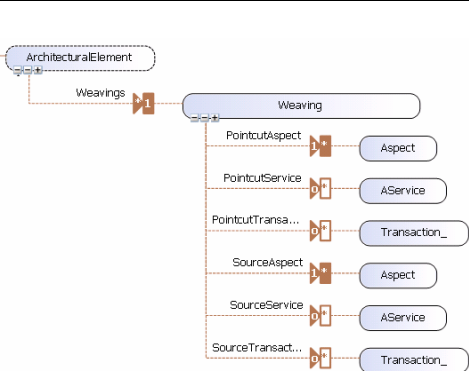
Related Patterns

Pattern 14, and Pattern 15.

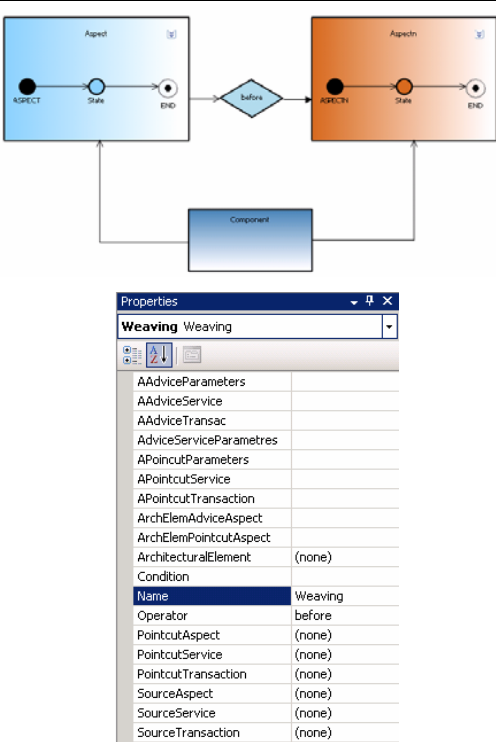
A.6. WEAVINGS

Pattern 18: Weavings

PRISMA Metamodel in DSL Tools



Graphical Metaphor



Transformation

Description

This pattern details how to generate the C# code from weaving.

Template

```
<#
foreach (Weaving weaving in architecturalElement.Weavings)
{
    if ( weaving.Operator.Equals(WeavingOperator.before) ||
```

300

```

        weaving.Operator.Equals(WeavingOperator.after) ||
        weaving.Operator.Equals(WeavingOperator.insteadof) )
    {
#>
        AddWeaving(GetAspect(typeof(<#weaving.SourceAspect.Concern#>Aspect)),
" <#weaving.SourceService.Name#>", "<#weaving.APoincutParameters#>",
        WeavingType.<#weaving.Operator.ToString().ToUpper()#>,
GetAspect(typeof(<#weaving.PointcutAspect.Concern#>Aspect)),
        "<#weaving.PointcutService.Name#>",
        "<#weaving.AAdviceParameters#>");

<#
    }
    else
    {
        //El separador dentro de la condición debe ser un espacio.
        string parametro_condicion=weaving.Condition.Split(' ')[0];
        string operador=weaving.Condition.Split(' ')[1];
        string valor_condicion=weaving.Condition.Split(' ')[2];
#>
        WeavingType weavingType =
            WeavingType.<#=ChangeWeavingType(weaving.Operator.ToString())#>
                ("<#=parametro_condicion#>",
WeavingType.OperatorType.<#=ChangeOperator(operador)#>,
                <#=valor_condicion#>);

        AddWeaving(GetAspect(typeof(<#weaving.SourceAspect.Concern#>Aspect)),
" <#weaving.SourceService.Name#>", "<#weaving.APoincutParameters#>",
        weavingType,
GetAspect(typeof(<#weaving.PointcutAspect.Concern#>Aspect)),
        "<#weaving.PointcutService.Name#>",
        "<#weaving.AAdviceParameters#>");

<#
    }
}/* endforeach (Weaving in architecturalElement.Weavings) */
#>
...

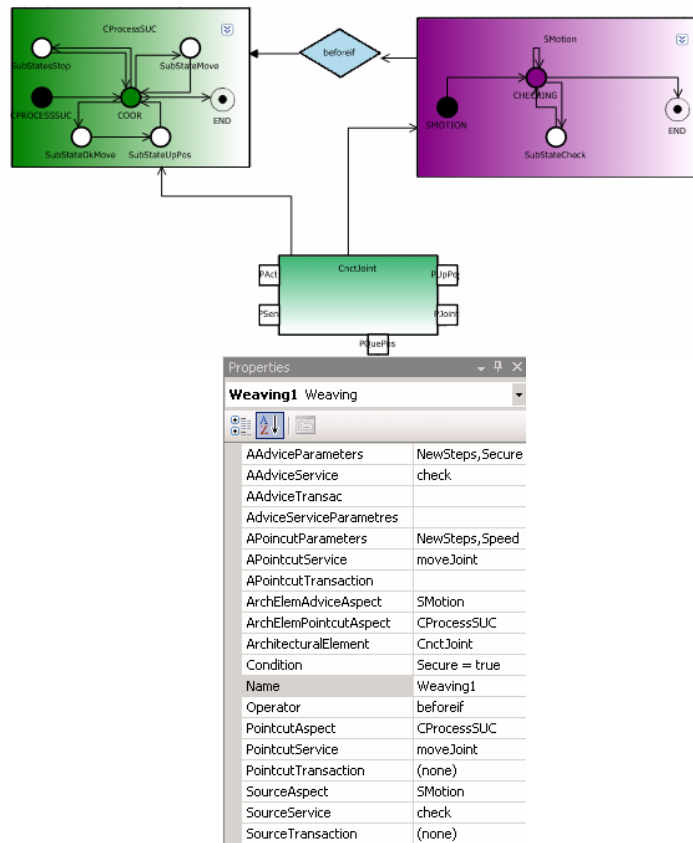
```

Case Study

Description

This pattern is illustrated using the *BeforeIf* weaving between the aspects *CProcessSUC* and *CnctJoint* of the TeachMover architectural model. The representation of this weaving in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.

Graphical representation



Result of the pattern execution

```

...
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace RobotJoint
{
    [Serializable]
    public class CnctJoint : ComponentBase , IConnector
    {
        public CnctJoint(string name, int IniMin, int IniMax, int IniPos ) :
        base(name)
        {
            ...

            WeavingType weavingType=
            WeavingType.BEFOREIF VALUE("Secure",WeavingType.OperatorType.Equality,true);

```

```

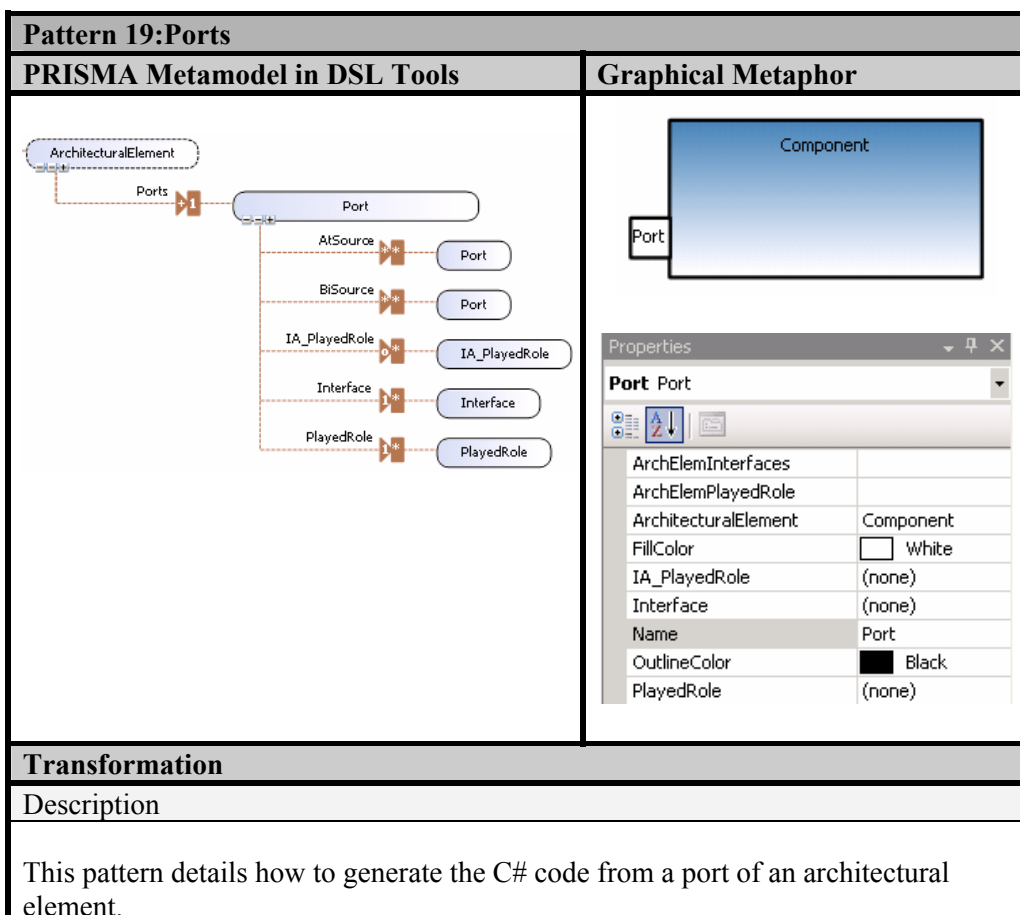
AddWeaving(GetAspect(typeof(SafetyAspect)), "check", "NewSteps, Secure",
weavingType,
        GetAspect(typeof(CoordinationAspect)), "moveJoint",
        "NewSteps, Speed");
    ...
}
}
}

```

Related Patterns

Pattern 14, and Pattern 15.

A.7. PORTS



Template																					
<pre> ... <# foreach (Port port in architecturalElement.Ports) { if (port.PlayedRole != null) { #> InPorts.Add("<#port.Name#>", "<#port.Interface.Name#>", "<#port.PlayedRole.Name#>"); OutPorts.Add("<#port.Name#>", "<#port.Interface.Name#>", "<#port.PlayedRole.Name#>"); <# } else if (port.IA_PlayedRole != null) { #> InPorts.Add("<#port.Name#>", "<#port.Interface.Name#>", "<#port.IA_PlayedRole.Name#>"); OutPorts.Add("<#port.Name#>", "<#port.Interface.Name#>", "<#port.IA_PlayedRole.Name#>"); <# } }/* endforeach (Port port in architecturalElement.Ports) */ #> ... </pre>																					
Case Study																					
Description																					
<p>This pattern is illustrated using the port <i>PUpPos</i> of the <i>CnctJoint</i> connector of the TeachMover architectural model. The representation of the <i>PUpPos</i> in the PRISMA model and the C# code generated from this model by applying this pattern are presented following.</p>																					
Graphical representation																					
<p>The diagram shows a central green rectangle labeled 'CnctJoint'. It has four ports: 'PAct' on the left, 'PSen' on the bottom-left, 'PUpPos' on the right, and 'PJoint' on the bottom-right. A small box labeled 'PQuePos' is located below the 'PUpPos' port.</p>	<p>The Properties window for 'PUpPos Port' displays the following attributes:</p> <table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>ArchElemInterfaces</td> <td>IUpdatePos</td> </tr> <tr> <td>ArchElemPlayedRole</td> <td>UPDATEPOS</td> </tr> <tr> <td>ArchitecturalElement</td> <td>CnctJoint</td> </tr> <tr> <td>FillColor</td> <td>White</td> </tr> <tr> <td>IA_PlayedRole</td> <td>(none)</td> </tr> <tr> <td>Interface</td> <td>IUpdatePos</td> </tr> <tr> <td>Name</td> <td>PUpPos</td> </tr> <tr> <td>OutlineColor</td> <td>Black</td> </tr> <tr> <td>PlayedRole</td> <td>UPDATEPOS</td> </tr> </tbody> </table>	Property	Value	ArchElemInterfaces	IUpdatePos	ArchElemPlayedRole	UPDATEPOS	ArchitecturalElement	CnctJoint	FillColor	White	IA_PlayedRole	(none)	Interface	IUpdatePos	Name	PUpPos	OutlineColor	Black	PlayedRole	UPDATEPOS
Property	Value																				
ArchElemInterfaces	IUpdatePos																				
ArchElemPlayedRole	UPDATEPOS																				
ArchitecturalElement	CnctJoint																				
FillColor	White																				
IA_PlayedRole	(none)																				
Interface	IUpdatePos																				
Name	PUpPos																				
OutlineColor	Black																				
PlayedRole	UPDATEPOS																				

Result of the pattern execution
<pre>using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types; using PRISMA.Components; using PRISMA.Middleware; namespace RobotJoint { [Serializable] public class CnctJoint : ComponentBase , IConnector { public CnctJoint(string name, int IniMin, int IniMax, int IniPos) : base(name) { InPorts.Add("PUpPos", "IUpdatePos", "UPDATEPOS"); OutPorts.Add("PUpPos", "IUpdatePos", "UPDATEPOS"); } } }</pre>
Related Patterns
Pattern 14, and Pattern 15.