Journal of Universal Computer Science, vol. 19, no. 10 (2013), 1433-1473
submitted: 19/10/12, accepted: 27/5/13, appeared: 28/5/13 © J.UCS

Model-Driven Development of Aspect-Oriented Software Architectures

Jennifer Pérez
(Technical University of Madrid (UPM), Madrid, Spain
jenifer.perez@eui.upm.es)

Isidro Ramos, Jose A. Carsí
(Universidad Politécnica de Valencia, Valencia, Spain
iramos@dsic.upv.es, pcarsi@dsic.upv.es)

Cristóbal Costa-Soria
(Global Metanoia S.L., Paterna Technological Science Park (Valencia), Spain
ccosta@globalmetanoia.com)

Abstract: The Model-Driven Development (MDD) paradigm has become widely spread in the last few years due to being based on models instead of source code, and using automatic generation techniques to obtain the final software product. Until now, the most mature methodologies that have been proposed to develop software following MDD are mainly based on functional requirements by following the Object-Oriented Paradigm. Therefore, mature MDD methodologies are required for supporting the code generation from models that specify non-functional requirements. The Aspect-Oriented Software Development (AOSD) approach was created to provide explicit mechanisms for developing non-functional requirements through reusable elements called aspects. Aspect-Oriented Software Architectures (AOSA) emerged to deal with the design of both, functional requirements and non-functional requirements, which opened an important challenge in the software engineering field: the definition of a methodology for supporting the development of AOSAs following the MDD paradigm. This new methodology should allow the code generation from models which specify functional and non-functional requirements. This paper presents a mature approach, called PRISMA, which deals with this challenge. Therefore, this contribution takes a step forward in the area presenting in detail the PRISMA MDD process, which has been applied to generate the code of several real applications of the tele-operated robotics domain. PRISMA MDD approach provides complete support for the development of technology-independent AOSAs, which can be compiled from high-level, aspect-oriented architectural models into different technology platforms and languages following an MDD process. This contribution illustrates how to apply the PRISMA MDD approach through the modelling framework that has been developed to support it, and a case study of a tele-operated robot that has been completely developed using this approach. Finally, the results obtained from the application of PRISMA MDD process to develop applications of the tele-operation domain are analyzed in terms of code generation.

Keywords: Model-Driven Development MDD), Software Architecture, Aspect-Oriented Software Development (AOSD), Aspect-Oriented Software Architectures, Code generation
Categories: D.2.2, D.2.10, D.2.11, D.2.13

1 Introduction

The complexity of current software systems has increased the time and cost required in the development and maintenance processes of software. As a result, there is great interest in the software engineering area to fix this problem. Thus, to achieve the quality goals of software products and ensure market competitiveness, the Model-Driven Development (MDD) paradigm emerged.

MDD is a software development paradigm related to Model-Driven Engineering (MDE) [Schmidt, 06], which has become more and more widespread in the last few years [Beydeda, 05]. This is due to the fact that it improves the different stages of the software life cycle (requirements, analysis, software architectures, implementation, etc.) by automating their activities. It is mainly based on models for software development, techniques to improve reusability, and processes to support automation, traceability, and maintainability. It promotes a high level of abstraction of software artifacts (models instead of code) and the automation of most of programming tasks. The use of models to develop software provides solutions that are independent of technology, and whose source code can be obtained by means of automatic code generation techniques for different technologies and programming languages.

Until now, the most mature methodologies that have been proposed to develop software following MDD are based on the Object-Oriented Paradigm (OOP) [Meyer, 98]. So, only OO software systems can take advantage of the reduction of time to market and cost that MDD provides. OO modelling approaches are mainly focused on the specification of functional requirements. However, non-functional requirements are acquiring much relevance in the current software systems: such as safety, security or distribution in embedded systems, or persistence, graphical-user interface or privacy in information systems or social network services, etc. These non-functional requirements increment the complexity of software, and their implementation requires software modules being decoupled and modularized in order to be maintainable and reusable. Several software development approaches have been defined to deal with this need, such as Aspect-Oriented Software Development (AOSD) [Chitchyan, 05] and Software Architectures [Perry, 92]. Therefore, to provide the advantages of MDD to those software systems where non-functional requirements are critical, MDD needs to deal with the modelling and the code generation of these software development approaches, such as AOSD and Software Architectures. Aspect-Oriented Software Architectures (AOSAs) emerged to take advantage of both, the Software Architectures and AOSD approaches [Cuesta, 05], [Chitchyan, 05], etc. PRISMA is a model that follows the AOSA approach by integrating the Software Architectures and AOSD approaches [Pérez, 06a]. It is described by its meta-model, provides a formal Aspect-Oriented Architecture Description Language (AOADL) [Pérez, 06b] and defines a methodology for specifying its AOSAs [Pérez, 08a].

An important challenge in the software engineering field is the integration of the AOSA approach into the MDD paradigm to support the development and maintenance of complex software systems in an efficient way. In this paper, we deal with this challenge by the definition of the PRISMA MDD process for supporting the MDD of AOSAs.

From the PRISMA previous work, in this paper we take a step forward and we define a consolidated MDD process for AOSAs that wraps up all previous PRISMA

contributions. The construction of this AOSA MDD process is based on the results of the study that we have performed about "*The main set of desirable properties that any AOSA MDD approach should fulfil"*. These profitable results for the AOSA community are presented in this paper as the founding premises of the PRISMA MDD process.

This paper presents a novel contribution for PRISMA: its MDD process. The main goal of this PRISMA MDD process is to provide complete support for the development of technology-independent AOSAs, which could be compiled to different technology platforms and languages using automatic code generation techniques. To make feasible the PRISMA MDD process, the development of a tool for supporting the complete code generation for AOSAs was necessary. The PRISMA CASE framework has been developed to cope with this need. The paper illustrates how the PRISMA CASE framework supports each one of the steps of the PRISMA MDD process.

This PRISMA MDD process has been consolidated by means of its application to academic examples (such as: banking systems and auctions), and also by means of the generation of real applications from the robotic domain (such as: tele-operated robots and agriculture robots). From these applications of the PRISMA MDD process, we have obtained a set of code generation results, which analysis reveals that the complete automatic-code generation from AOSA models is feasible.

In summary, the novel contributions of this paper are: (i) the results of the study of the MDD support of AOSA approaches, (ii) the PRISMA MDD process, (iii) the complete PRISMA CASE framework, and (iv) the results of the application of the PRISMA MDD process. These contributions are structured in the paper as follows. The background about MDD, AOAS, and PRISMA, and the study about properties of AOSA MDD approaches are presented in Section 2. The robot *TeachMover* and the piece of its architecture that is used as an example throughout the paper are introduced in Section 3. The MDD process of PRISMA is explained in detail in Section 4. Section 5 discusses this contribution by analyzing related work. Section 6 describes and compares the experimental results obtained from the development of two applications using the PRISMA MDD process. Finally, conclusions and further work are presented in Section 6.

2 Background

2.1 Model-Driven Development (MDD)

Most mistakes that are made during software development come from the first stages of the software life cycle. Since these mistakes grow in an exponential way as the life-cycle progresses, it is necessary to focus on improving these first stages instead of postponing the solution for later stages. Most proposals that try to solve these problems improve software development by automating its first stages following MDD. Our contribution also fits these ideas and challenges focusing at the architectural level.

MDD is a software development paradigm of Model-Driven Engineering (MDE) [Beydeda, 05]. It is mainly based on models for software development, techniques to improve reusability, and processes to support automation, traceability, and maintainability. It promotes using models instead of code to automate software

development, guaranteeing solutions that are independent of the technology-platform, consistent and reusable. It is based on code generation techniques for generating the code of different technologies and languages from models (either models or metamodels (models of models)).

The OMG Meta-Object Facility (MOF) 2.0. [MOF, 12] specification defines an architecture to support meta-modeling, MDD and model management (definition of transformations, traceability links, mappings, etc., between models). It is a four-level architecture, whose main purpose is the management of model descriptions at different levels of abstraction. The upper layer (M3) defines the abstract language used to describe the entities of the M2 layer (metamodels) (see layer M3, Figure 1). The metamodel layer (M2) defines the structure and semantics of the models defined at the M1 layer (see layer M2, Figure 1). The M1 layer comprises the models that describe data, i.e. the application of the M0 layer (see layer M1, Figure 1). These models are described using the primitives and relationships described in the metamodel layer (M2). The lowest level is the information layer (M0), which comprises the final application to be described (see layer M0, Figure 1), i.e. the instances of the models that are defined at the M1 layer.

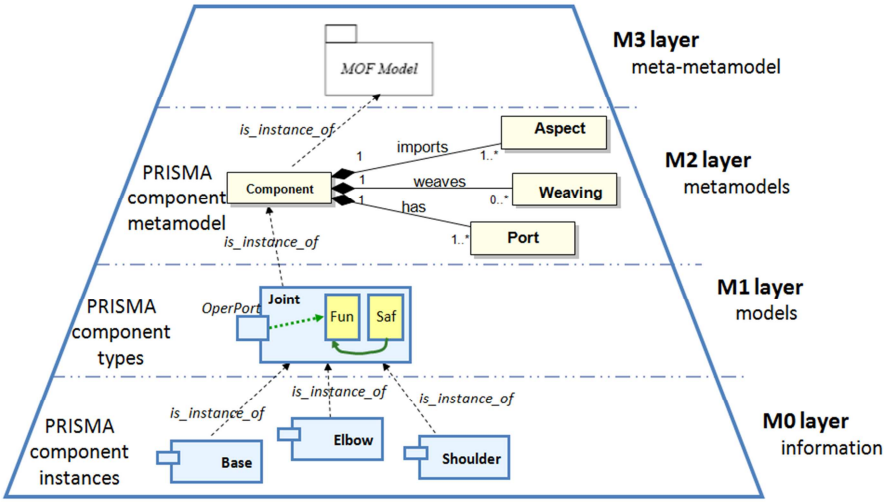


Figure 1: Meta-Object Facility (MOF) layers exemplified by PRISMA models

2.2 Aspect-Oriented Software Architectures in MDD

Software architectures and AOSD are the foundation of the AOSA approach. In this section, we provide an overview of AOSA by explaining the integration of Software Architectures and AOSD and their role in MDD, and the main properties that any AOSA MDD process should take into account for its construction.

2.2.1 Software Architectures

Software Architectures make software systems simpler and more understandable by hiding the low-level details. The works of Garlan [Garlan, 95] and Perry & Wolf

[Perry, 92] clearly define the role of software architectures in the software life cycle. The Software Architecture discipline bridges the gap between requirements and implementation stages of the software life cycle. The traceability between the resulting requirements of the requirements stage and developed code in the implementation stage is a key issue to prevent the failure of the software development process.

Software architectures comprise a wide-scope of tasks [Perry, 92]. This work is focused on the design of the structural viewpoint of Software Architectures, i.e. the description of Software Architectures. This structure is usually represented in terms of computational elements and the interactions among them using Architecture Description Languages (ADLs).

Software Architecture Descriptions are critical to trace the requirements to code in software development in general, and in MDD in particular. However, despite the fact that a wide variety of formal ADLs [Cuesta, 02], [Medvidovic, 00] have been proposed, there is a lot of work to be done to integrate formal ADLs into MDD.

2.2.2 Aspect-Oriented Software Development (AOSD)

Non-functional requirements have acquired an important relevance in current software systems. Non-functional requirements are usually common concerns of a domain system that are scattered in the software units that a system is composed of (classes/objects [Meyer, 98], components [Szyperski, 98], modules modules [Meyer, 03], etc.). These common concerns are crosscutting most software units of the system (*crosscutting concerns)*. These *crosscutting concerns* are tangled with the other concerns that also modify the same software unit affecting most functionalities of the system. Hence, the support of software modularity and *crosscutting-concerns* are essential challenges to be faced in software development. The *Separation of Concerns (SoC)* helps to address these challenges [Parnas, 72]. The *SoC* principle promotes dealing with the different concerns of a software system separately and context-unaware when possible. *Aspect-Oriented Programming (AOP)* [Kiczales, 97] has emerged as an innovative way of applying *SoC* in software development.

AOP proposes the separation of the *crosscutting-concerns* of software systems into separate entities, which are called *aspects.* This separation avoids having tangled concerns scattered among the software units and allows the reuse of the same aspect in different software units of the software system. AOSD has emerged not only to apply AO to the implementation stage, but also to apply it to every stage of the software life cycle. For this reason, the integration of AOSD development with the MDD process has to be taken into account.

There are some approaches that combine MDD and AOSD mechanisms such as [Aksit, 05], [Amaya, 05],[Kulkarni, 03], and [Simmonds, 05]. However, none of these AO approaches takes into account software architectures, and none of the original ADLs previously presented (see Section 2.2.1) explicitly distinguishes the conventional architectural elements from concerns, which crosscut multiple architectural elements of software architectures.

2.2.3 The most relevant properties for MDD support of Aspect-Oriented Software Architectures

The combination of AOSD and software architectures permits the separation of concerns of architectural components. So, software architectures can take advantage of AOSD benefits. This section analyzes the main set of desirable properties that any aspect-oriented architecture MDD approach should fulfil. These properties are mainly founded on the MDD principles [Schmidt, 06] and on the key guidelines about software development the literature sets out: high-level languages for programming [Parnas, 85], software partitioning by dividing into views [Perry, 92], [Kruchten, 95] or the size of code ("divide and conquer") [Dijkstra, 76] and software composition and decomposition [Parnas, 72] [Harrison, 02]. Hence, the premise, in which the study is based on, is the following: *"An aspect-oriented software architecture approach should completely support the development and maintenance processes of software following the MDD approach"*. And, the properties and the reasons for considering them as classification criteria of MDD aspect-oriented architectural approaches are described below:

- **Aspect-Oriented Model (AOM)**: This feature defines the kind of AOM that is integrated with the software architectural model, which may be mainly classified into asymmetric and symmetric [Harrison, 02]. This characteristic is important because the facilities of reuse and code generation can vary depending on which aspect-oriented model is used. An asymmetric model is based on a dominant decomposition, which is usually an OO-like functional decomposition. Models of this kind assume that aspects are non-functional concerns that crosscut the functional units of software (called *base code)*. However, in symmetric models, everything is considered as a concern, and there is no dominant decomposition. As a result, functionality is considered to be another concern, and concerns crosscut each other. Therefore, symmetric models are considered to be more flexible and abstract. However, since the most extended AOM is the asymmetric model of AspectJ [Kiczales, 01], asymmetric models are more widely used because they are easier to integrate in current software development approaches.
- **Architectural model**: This feature determines whether or not an architectural model provides connectors for modelling software architectures. The ones that have connectors provide features that improve the structure and maintenance of software architecture [Shaw, 94] . So, those architectural models that provide connectors make architectural elements more independent and facilitate their transformation in an MDD process, as well as their reusability and integration in other architectural models.
- **Formal ADL**: Another feature that is necessary to take into account when comparing architectural models is whether or not their corresponding ADLs are formal languages. 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, and to evolve software architectures preserving the consistency of the system.
- **Graphical support**: The graphical specification of AOSAs greatly helps to avoid the complexity of using formal and technology independent ADLs. Graphical

support is achieved by defining the graphical metaphor for ADLs by means of a new language or by extending a well-known graphical language.

- **Definition of Aspects**: The most distinguishable feature of AO architectural models is how they integrate aspects and software architectures. There are two ways of doing this: (i) by simulating the notion of aspect by means of another architectural concept, or (ii) by defining a new concept for aspects in software architectures. The first way refines the architectural concepts by varying their original semantics; and the second one requires understanding a new concept to model software architectures. Therefore, those proposals that define aspects as a new concept with its proper semantics facilitate their transformation and unambiguous integration with any architectural model.
- **Definition of Weavings**: Weaving is a process that, in asymmetric AOM, consists of coordinating the *base code* and the *aspect code*. In symmetric AOM, weavings coordinate the different aspects. This is an important feature of AOMs and AO architectural models. The definition of weavings specifies where the weaving process between aspects and architectural elements is defined. If it is defined inside the aspect, the aspect is dependent on the context that the aspect is connected to. However, if it is defined outside the aspect, the behaviour of the aspect can be reused independently of where the aspects are connected to, and so, evolution and reuse are facilitated.
- **Aspect-Oriented Evolution**: The support of aspect evolution is an important feature that can improve the design-time and run-time evolution of software architectures. Thus, an approach that provides mechanisms for easily adding or removing aspects is a great advantage to perform transformations between models for their modification.
- **Purpose**: Since our premise of this analysis is the following: "*An aspect-oriented software architecture approach should completely support the development and maintenance processes of software following the MDD approach"*. The purpose of an approach is an essential feature to be able to compare the different AOSA approaches. There are AO 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 software architectural approaches that exist. If the purpose is to take advantage of MDD, AOSAs should be specified in an abstract way by means of formal and technology independent ADLs. As a result, a single specification can be transformed to different platforms and different programming languages. However, if the AOSA specification depends on a specific platform and/or programming language, its application and flexibility are considerably reduced.
- **Tool support**: A significant feature of a MDD AOSA approach is its support by means of a framework, which will guide 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, etc.
- **MDD support**: The previous criteria are essential to achieve the MDD support, which is essential for this contribution. The support that the MDD approach offers is significant in the development and maintenance processes. A complete MDD

support should consist in: (i) automatically generating the code from models, (ii) guiding the analyst through all the stages, and (iii) providing mechanisms to facilitate the tasks (some of these mechanisms are: verification techniques, reusability mechanisms, integration facilities, code generation mechanisms, etc.).

2.3 An Overview of PRISMA

2.3.1 The PRISMA Model

The PRISMA approach is based on the PRISMA model, its AOADL, its methodology, and its tool [Pérez, 06a], [Pérez, 06b], [Pérez, 08a]. PRISMA provides a model for the description of software architectures of complex and large systems. A PRISMA architectural element (components and connectors of an architecture design) can be seen from two different views: internal and external. In the external view (black box view), architectural elements encapsulate their functionality as black boxes and publish a set of services (*interface*) that they offer/ask to/from other architectural elements through their ports (see Figure 2.A). In the internal view (white box view), an architectural element is shown as a prism. Each side of the prism is an aspect that the architectural element imports (see Figure 2.B). In this way, architectural elements are represented as a set of aspects and the relationships among them (called *weavings*).

The notion of aspect arises to deal with crosscutting-concerns of software systems. This idea of crosscutting can vary depending on the nature of the model. The PRISMA model is a symmetrical AOM [Harrison, 02] because it does not consider functionality as a base code (i.e. different to aspects), and it does not constrain aspects to specify non-functional requirements (see property Aspect-Oriented Model (AOM), Section 2.2.3). In PRISMA, functionality is also specified as an aspect by providing a homogeneous treatment to functional and non-functional requirements. Aspects have been introduced in the PRISMA AOADL as a new concept rather than simulating the aspect using other architectural terms (components, connectors, views, etc.). This is due to the fact that a component can specify state and behaviour about different concerns, whereas an aspect is focused in a single concern. As a result, PRISMA preserves the meaning of the concepts of component and aspect, keeping them as first-order citizens.

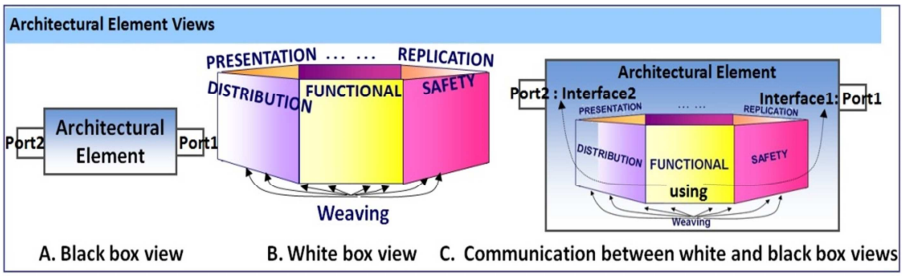


Figure 2: Views of an architectural element

With regard to AOSD, PRISMA aspects are artefacts of software architectures and represent a specific behaviour of a concern (safety, coordination, persistence, security, etc.) that crosscuts the software architecture. The same aspect can be imported by each one of the architectural elements (components and connectors) that need to take into account this behaviour. As a result, an architectural element is defined by a set of aspects that describe it from different concerns of the architecture. The communications between the white box and black box views is possible by means of interfaces, which are associated with ports and are used by aspects (see Figure 2.C). Consequently, a request for a service that arrives at a port of an architectural element is processed by an aspect that implements the same service that is provided by this port.

In PRISMA, a weaving indicates that the execution of an aspect service can trigger the execution of services in other aspects. In PRISMA, to preserve the independence of the aspect specification from other aspects and weavings, weavings are specified outside aspects and inside architectural elements (see Figure 2.C). They are specified using the weaving operators *after, before, instead, afterif(boolean_condition), beforeif(boolean_condition) and insteadif(boolean_condition),* and following the pattern: *aspect1.service1* ***weaving_operator*** *aspect2.service1*. These patterns means that the service2 of the aspect2 will be executed *after, before, instead* the *service1* of the *aspect1*, and in the case of the conditional weaving operators, the *service2* of the *aspect2* will be executed only *if the boolean_condition* is true. 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. Figure 3 illustrates an example of how PRISMA architectural elements import aspects, i.e. it shows the reusability facilities of PRISMA. This simple example shows how the same aspect, for example the aspect *FUNCTIONAL*, is imported by two different architectural elements, *Component1* and *Component3*, and how an architectural element, for example the connector *Connector1*, imports several aspects: *COORDINATION* and *SAFETY.*

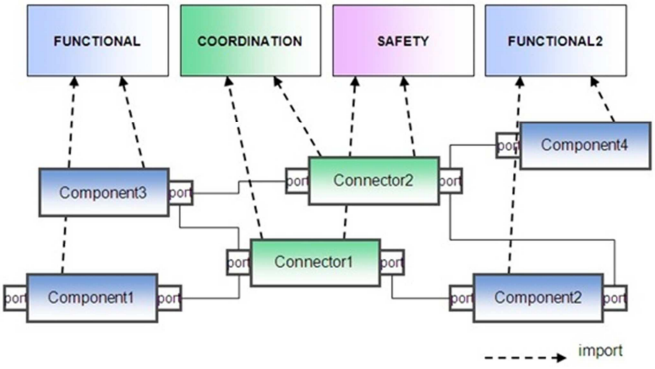


Figure 3: Crosscutting-concerns in PRISMA architectures

With regard to software architectures, PRISMA has three kinds of architectural elements: components, connectors, and Systems. 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 coordinates the interactions among other architectural elements. Connectors provide the separation of component interactions. 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. *Attachments* are the channels that enable the communication between components and connectors through their ports. *Systems* are complex PRISMA components, which include a set of architectural elements (connectors, components and other Systems) that are correctly attached. *Bindings* establish the connection among the ports of a System (i.e. a complex component) and the ports of the architectural elements that this System contains.

The PRISMA model has an AOADL to support it. The PRISMA AODL is a formal and technology-independent language (see property Formal ADL Section 2.2.3). It is important to emphasize that most ADLs only permit the specification of the skeleton of architectures and the services that are sent/received among their different architectural elements. The PRISMA AOADL has greater expressive power and can specify more features and requirements using aspects [Pérez, 06b]: it can specify state and behaviour for each aspect.

2.3.2 PRISMA in MOF

The PRISMA MDD approach fits the four-level architecture of MOF. In this way, MOF allows us to clearly present the differences between PRISMA *types* and *instances* and their corresponding *models*. The PRISMA metamodel is defined at the layer M2 (see Section 2.1), and the **PRISMA type models** are specified at the layer M1 e.g. the model of a joint of the tele-operated robot software architecture (see layer M1Figure 1). So, the PRISMA model of the joint of the tele-operated robot is compliant with the PRISMA metamodel. Finally, the specific instances of an architecture configuration are placed in the layer M0. They are called in PRISMA, **PRISMA configuration models**. For example, Figure 1 illustrates how the *Elbow* is an instance of the component *Joint* of the layer M1, which means that the *Elbow* behaves as the *Joint* describes. PRISMA reusable types and patterns are specified at layer M1 and architectural configurations are defined at the layer M0 by instantiating the types and patterns of the layer M1.

In this work, we present an MDD approach by making this MOF architecture of PRISMA models feasible (see Figure 1). To implement the approach, it is important to keep in mind that MDD and automatic code generation avoids the mistakes of correspondence between the semantics of the model and the application code, but it does not prevent the modelling mistakes that the user makes. To prevent them, help must be provided to guide the user during the modelling process. This guidance mechanism is provided using verification techniques. Verification can be performed in the modelling stage to detect modelling mistakes. During the modelling stage, the verification techniques allows us to know whether or not a model satisfies the constraints that its metamodel defines (the model conforms to its metamodel). If a

model satisfies every constraint of its metamodel, then the model is conformed and verified.

3 The Tele-Operation Domain

The PRISMA MDD approach has been validated by designing academic examples such as, banking systems, auctions, etc., and also, the generation of real applications from the robotic domain such as: tele-operated robots, agriculture robots, etc. In this section, we describe one of these real applications, which will be used throughout the rest of the paper to illustrate our contribution. It is a robotic tele-operated system, which is a family of robots for hull maintenance operations called EFTCoR (Environmental Friendly and Cost-effective Technology for Coating Removal) [EFTCoR, 02], [Fernández, 05]. This work was feasible thanks to the close collaboration[1] with the DSIE (System Division and Electronic Engineering) Group of the Polytechnic University of Cartagena of Spain, who was involved in the design of this EFTCoR Robot to be run in the cleaning tasks of the shipyard of Cartagena City.

Figure 4: Tele-Operated Robots: A) EFTCoR Primary Positioning System (arm joint and joint of tracks), B) manual cleaning by an operator, C) The TeachMover Robot

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 (see Figure 4. B). The EFTCoR is a robotic platform with strong non-functional requirements that cleans the hulls of ships in a way that reduces the environmental pollution. Since the EFTCoR is a family of robots that are very large (big dimensions) and very heavy (high tonnage) (see Figure 4. A), a complete development of a small-scale robot was done before developing the software architecture of EFTCoR. Specifically, we validated our proposal by

[1] This close collaboration was through the national project DYNAMICA.

developing the *TeachMover* robot [TeachMover, 12] (see Figure 4. C). This robot was specially designed for the purpose of simulating the behaviour of large and heavy industrial robots that cannot be manipulated in laboratories. The *TeachMover* is simpler than EFTCoR, but it has the same architectural features at a small scale to test the software of industrial robots before its deployment. All these features allow the *TeachMover* robot to simulate the movements of most of the industrial tele-operated robots.

The *TeachMover* is formed by a set of joints that permit the movement of the robot, which are commanded by an operator from a computer. These joints are: *Base, Shoulder, Elbow, and Wrist*. In addition, it has a Tool to perform different tasks: catch, drop, push, pull, etc. (see Figure 4.C). In this case, the *Tool* is a gripper (in other cases it can be a brush to paint, a hose to clean, etc.), whose open and close actions allow the robot to pick up and deposit objects. Therefore, it allows the robot to move objects from an initial position to a final one. There are required safety constraints of the robot movements to be checked and to make sure that its movements are safe for itself and the environment that surrounds it.

From the different components that a tele-operated robot is composed of, and the *TeachMover* in particular, we are going to focus on those that implements the joints of the robot [Pérez, 08a]. The architecture of a tele-operated robot joint is defined by a System (complex component) called *Joint*, which is composed of architectural elements that interact with the hardware joints of the robot. Specifically, the *Joint* is a System composed of two components and a connector and their corresponding connections (see Figure 5). The component *Actuator* is in charge of (i) communicating with the hardware joint, when commands are sent to the hardware joint of the robot to be performed and (ii) notifying the joint System when the commands have been performed successfully. The component *WrapAspSys* encapsulates the behaviour and the state related to the software joint, such as the position of the joint and its movements. And the connector *CnctJoint* coordinates the interaction between the *Actuator* and the *WrapAspSys*.

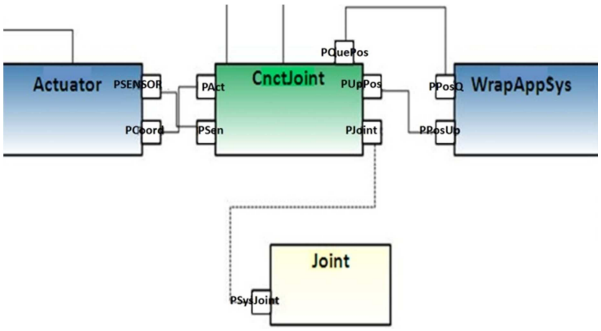


Figure 5: The joint architecture of a tele-operated robot

4 PRISMA: MDD Support for Developing Aspect-Oriented Software Architectures

This section describes how PRISMA provides complete support for the development of technology-independent, aspect-oriented software architectures following MDD.

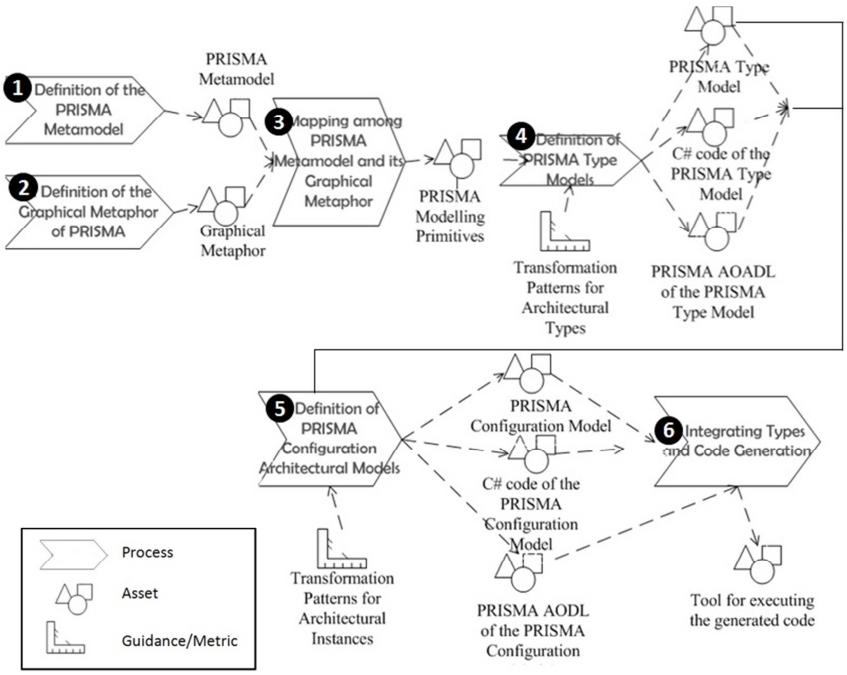


Figure 6: MDD Process from the PRISMA Metamodel to Applications

The PRISMA MDD process is not constrained to the definition of a specific number of levels of abstraction or techniques, because it can vary depending on the needs of each software system. PRISMA follows the MDD approach by enabling software architects to define AOSA models, which allow the complete generation of the final code of AOSAs. The tasks of the software architect are facilitated thanks to the fact that: (i) the level of abstraction provided by models is higher than the provided by programming languages, and (ii) the code is automatically generated from models. PRISMA CASE is the tool that makes the MDD software development of PRISMA feasible. It has been constructed using the Domain-Specific Languages Tools (DSL Tools) of the Microsoft Visual Studio framework [Cook, 07]. From PRISMA AOSA models, PRISMA CASE supports the modelling of PRISMA AOSA models and the generation of both PRISMA AOADL specifications and C# code, which is executable on .NET technology [NET,12]. The PRISMA CASE is composed of the following parts: the specification of the PRISMA metamodel, a graphical modelling tool, a model compiler, the PRISMANET middleware and a generic Graphical User Interface (GUI) to execute the generated code [Pérez, 06a]. The

PRISMA model compiler implements the C# code and AOADL generation patterns of the MDD process of PRISMA. When the model compiler is executed, the patterns are applied to transform the PRISMA architectural models to code, which is executed in the PRISMANET middleware. Finally and PRISMA CASE provides a generic GUI to assist the user in checking the behaviour of the architecture. Thereby, the C# PRISMA software architectures can be executed. In the paper, from now on, to illustrate the contribution, we are going to focus on the C# code generation, i.e. the execution of PRISMA models on .NET platform. And, we are going to use the PRISMA CASE tool and the example of a TeachMover joint to illustrate the PRISMA MDD approach. Figure 6 presents the complete PRISMA MDD process based on MOF, i.e. the different levels of refinement that models undergo during the development process. This process is explained in detail in the following sections.

4.1 From the PRISMA Metamodel to PRISMA Type Models

The PRISMA metamodel defines the PRISMA model and establishes its properties in a precise way. So, PRISMA type models are defined conforming to the PRISMA Metamodel. This definition is described in the following subsections.

4.1.1 Definition of the PRISMA Metamodel

The PRISMA metamodel is defined by a set of inter-related metaclasses (a class of classes) and constraints (see process 1, Figure 6). These metaclasses contain a set of properties and services for each concept considered in the model. Metaclasses, their properties and their relationships define the structure and the information that is necessary to describe PRISMA AOSAs. In addition, the PRISMA metamodel defines the constraints that cannot be specified using the structure or the information of the metamodel [Pérez, 06a]. The structure, information and constraints of the PRISMA metamodel must be satisfied by PRISMA type models in order to ensure that they are correct. One of the most representative packages, that the PRISMA Metamodel is composed of, is the package *Architectural Element* (see Figure 7). This package is going to be used for illustrating the contribution of this paper.

A PRISMA architectural element is defined by the metaclass *ArchitecturalElement* (which is modelled as a UML class). It is an abstract metaclass that specifies the commonalities of the three kinds of PRISMA architectural elements (Components, Connectors and Systems (see Section 2.3.1). To define that an architectural element has ports and weavings, the metaclass *ArchitecturalElement* has two aggregation relationships (the UML Class Diagram aggregation), *has* and *weaves,* with the metaclassess *Port* and *Weaving*, respectively. In addition, the metaclass *ArchitecturalElementen* has one association relationship *imports* with the metaclass *Aspect* to denote that an architectural element imports a set of aspects. Next, these three relationships (*has, weaves* and *imports*) are explained in detail taking into account the semantics that the cardinalities define: (i) An architectural element has at least one port and a port can only be defined as part of an architectural element (see the aggregation *has* in Figure 7); (ii) 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 association *imports* in Figure 7); and (iii) An architectural element can include a set of weavings to synchronize its aspects. These Weavings are

related to the architectural element; in fact they can only be defined as part of an architectural element (see the aggregation *weaves* in Figure 7). Also, the metaclass *ArchitecturalElement* has attributes, services and constraints to completely define its properties.

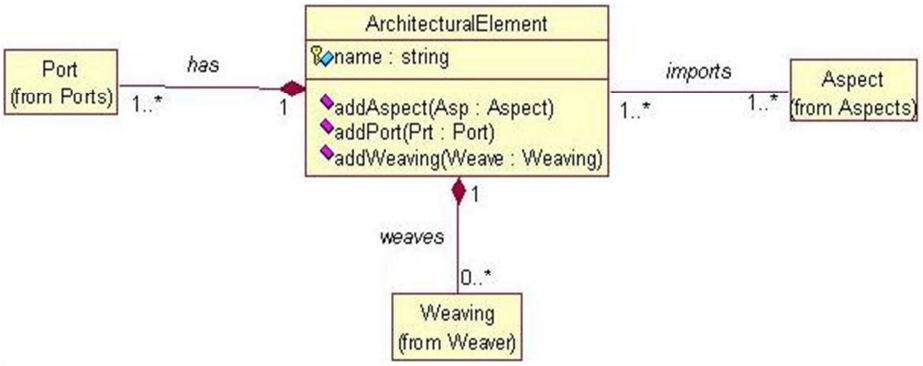


Figure 7: The package ArchitecturalElements of the PRISMA metamodel

With regard to the verification of the PRISMA metamodel, we have distinguished between two kinds of verification: verification rules that must always be satisfied (*hard constraints*), and verification rules that must be satisfied once the model has been completely finished (*weak constraints*).

The hard constraints and weak constraints of the MDD process of PRISMA have been included during the development of PRISMA CASE thanks to the partial C# classes that DSL Tools provide. In DSL Tools, each metaclass of the metamodel is implemented as a partial C# class, which can be extended and customized to add additional behaviour that cannot be included in the metamodel (such as verification rules) [Cook, 07]. In PRISMA CASE, the extension facilities of the partial C# classes have been used for including the needed verification constraints. Thus, since each time that a metaclass is used for modelling a concept in a PRISMA model, its Partial C# class is executed, the constraints of the PRISMA metamodel are verified during the modelling process.

4.1.2 Graphical representation and mapping with PRISMA metamodel concepts

The PRISMA metamodel provides the primitives to model AOSAs. However, to use these primitives in a modeling context, it is necessary to define a graphical representation for using them. PRISMA CASE provides a graphical language to model PRISMA software architectures in an intuitive and friendly way to facilitate the architect tasks (see process 2, Figure 6). The graphical representation that supports the main modelling concepts of PRISMA have been already presented in the paper (see Figures 2, 3, 5 and 10).

Once the graphical representations are defined, each one is associated with its corresponding metamodel concept (see process 3, Figure 6). The PRISMA CASE is generated from the PRISMA metamodel, its graphical representations, and its partial

C# classes. It is composed of a toolbox, a drawing sheet, a model explorer, a window of properties, and a PRISMA menu (see Figure 8).

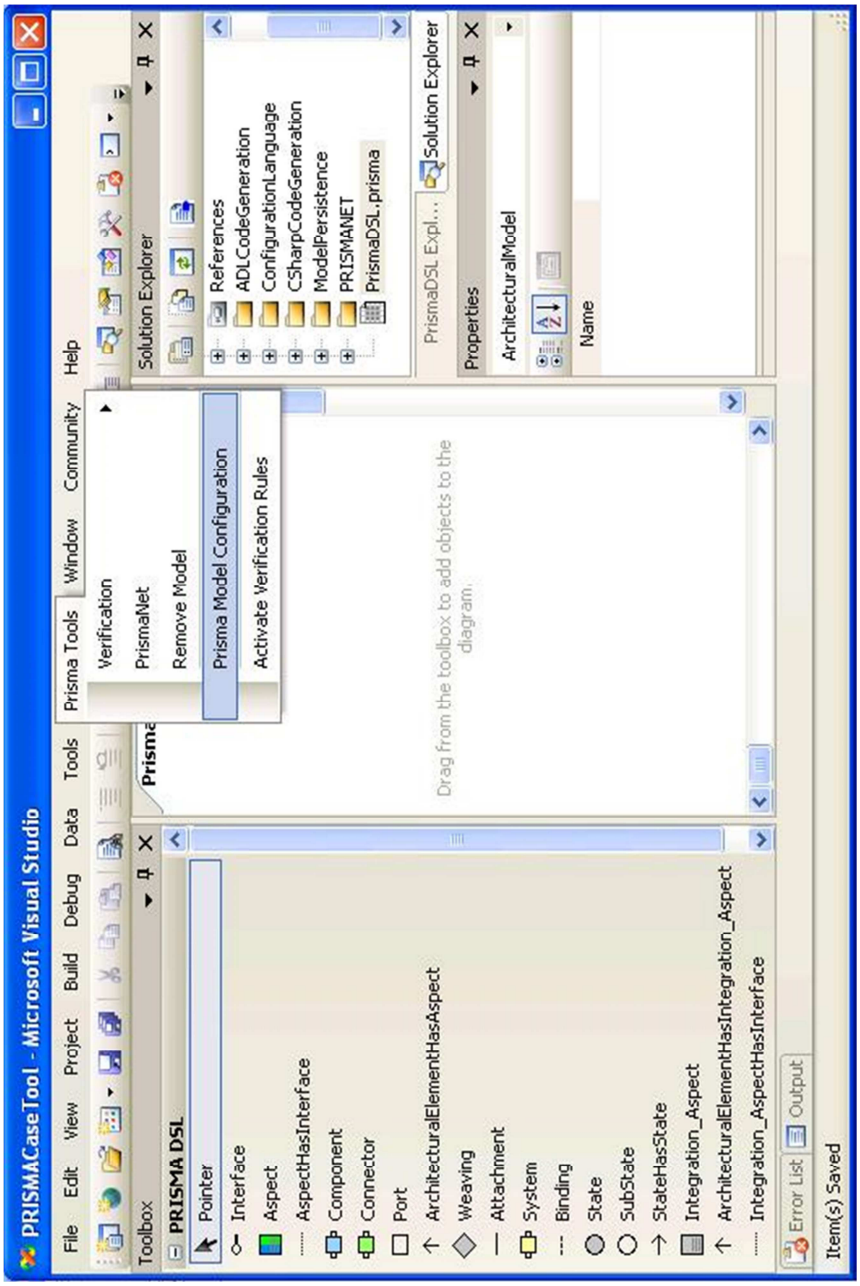


Figure 8: PRISMA CASE Tool: Toolbox, Drawing Sheet, PRISMA Tools Menu, and Solution Explorer and Properties Windows.

4.1.3 Definition of PRISMA type models

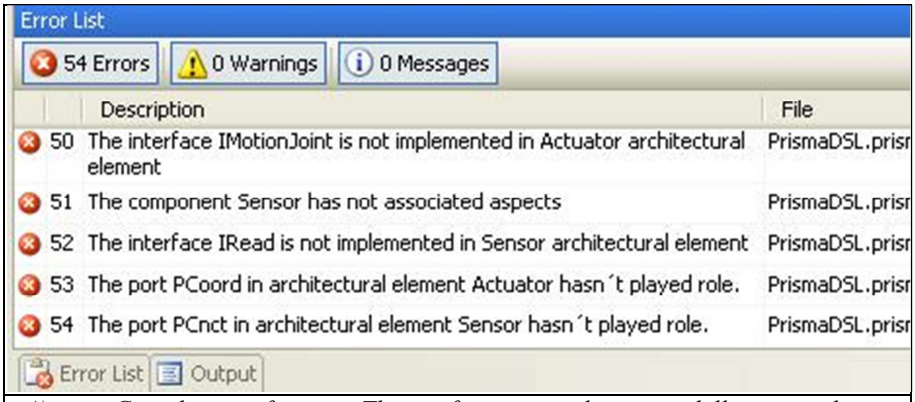
Every PRISMA type model must be defined conforming to the PRISMA metamodel (see process 4, Figure 6). A PRISMA type model is a generic system architecture (banking system, tele-operated robot, electronic auction, etc.) that can be reused for different specific systems. This step is easily developed thanks to the fact that PRISMA type models are described using the concepts that are defined in the metamodel as modelling primitives provided by PRISMA CASE. As a result, the user is able to model PRISMA architectural models and to make sure that these models satisfy the PRISMA metamodel.

The PRISMA MDD process assists the architect by providing mechanisms for the **verification of models**. The verification of models allows the detection of modelling mistakes, and keeps them from spreading throughout the rest of the stages. This is essential in the PRISMA MDD process in order to avoid code generation from incorrect models. Therefore, when a model reaches the last generation step (see Figure 6), it is guaranteed the correctness of the model and the proper performance of the code generation. The verification of architectural models consists of checking that PRISMA type architectural models satisfy the properties and constraints that are defined in the PRISMA metamodel. Specifically, the verification consists of checking that the models satisfy the following properties: (i) the types of a model contain all the information that their metaclasses establish, (ii) the relationships of the model only connect the types which connection is allowed, (iii) the number of types or the relationships between these types is correct, and (iv) the constraints of the metamodel are satisfied. This verification process must always be applied to the modelling process of PRISMA architectural models and must guide the software architect throughout the process. In the modelling tool, the verification of constrains is different depending on their kind. The **weak constraints** act as warnings during the modelling process of **PRISMA type models** (see Figure 9.A), whereas **hard constraints** are verified while the user is modelling.

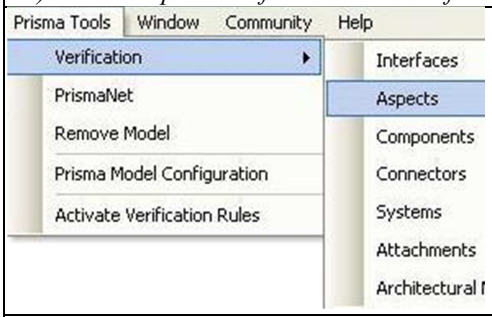
Weak constraints can be violated during the modelling process, but they must be rectified during the modelling process because all of them must be satisfied once the architectural model is finished. For example: in PRISMA, an architectural element must import at least one aspect, and must have at least one port (see Figure 7), but it is possible to define an architectural element without establishing its ports and/or aspect, and to establish them later.

Weak constraints provide more flexibility to the modelling process. The fact that there are weak constraints that are not satisfied means that the modelling process has not finished. However, there may be parts of the architectural model that are finished and the architect may want to verify them. As a result, there are two kinds of verifications that are supported by the PRISMA MDD process: **Partial Verification** and **Complete Verification**. The **Partial Verification** consists of applying only those constraints that affect the elements, concepts or parts of the model that have been selected by the architect for verification (see Figure 9. B and Figure 9.C). This kind of verification allows the architect to define a model, and then 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. **Complete Verification** is the verification that is applied to the complete architectural model (see Figure 9.A). As a result, complete verification consists of verifying all the constraints that must satisfy a

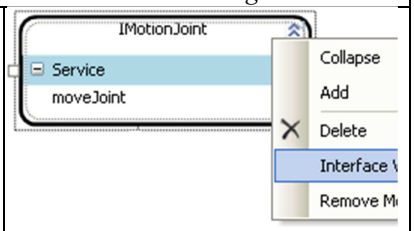
model. In PRISMA, this process implies that all the restrictions of the PRISMA metamodel are checked.



A) Complete verification: The verification result as a modelling error list



B) Partial verification: a specific type of element: Interfaces, Aspects, Components, etc.



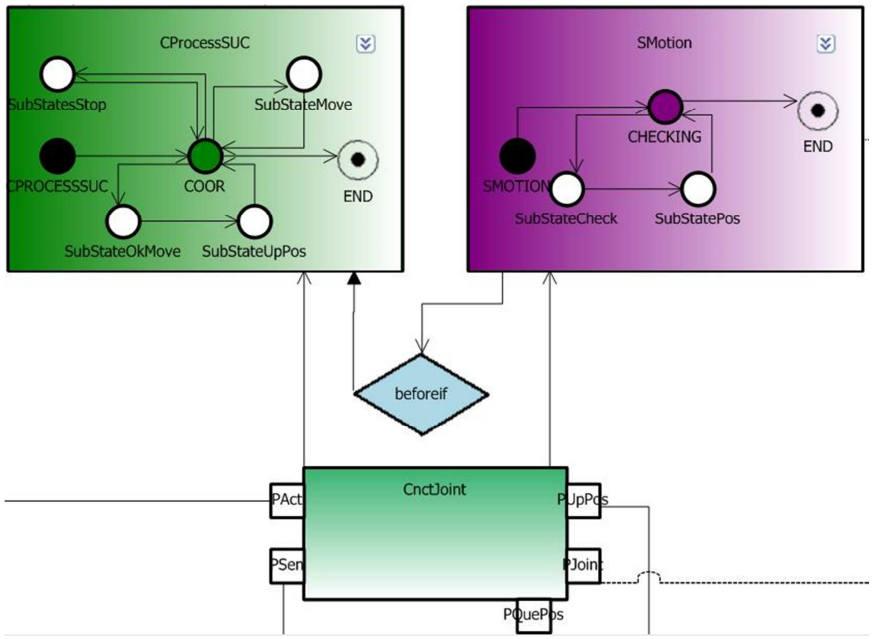
C) *Partial verification: only one element of the architectural model, the interface* IMotionJoint

Figure 9: Partial and Complete Verification of PRISMA Type Modelling Tool

Hard constraints are very close to the graphical metaphor and must always be satisfied without taking into account the modelling process situation. An example of hard constraint is the requirement that a component cannot import a coordination aspect. This is due to the definition of Component: "A component is an architectural element that captures the functionality of software systems and does not act as a coordinator among other architectural elements"(see Section 2.3.1). Since a component is not a coordinator, it never imports a coordination aspect. This hard constraint is materialized in the modelling tool as follows: If an architect associates a coordination aspect with a component, the resulting model would violate the PRISMA model. So, PRISMA CASE does not allow drawing this connection.

In our example, the result of this stage is a PRISMA type model for a generic joint of a tele-operated robot that can be reused for designing different joints of the same robot or joints of different robots. In this PRISMA type model, both components, *Actuator* and *WrapAspSyst,* are coordinated through a connector

CnctJoint (see Figure 5). Each component imports its functionality through aspects, and the connector *CnctJoint* imports its behaviour through a coordination and a safety aspects, *CoordJoint* and *SMotion* respectively (see the appendix A). These two aspects, *CoordJoint* and *SMotion,* are weaved (see Section 2.3.1) to ensure that a joint is moved only after the connector safety constraints are satisfied assuring that a movement is safe. Figure 10 illustrates this weaving, which specifies that the invocation of the *moveJoint* service of *CProcessSUC* implies that the *DANGEROUSCHECKING* service of *SMotion* will be executed *beforeif* the *moveJoint* service of *CProcessSUC*. The weaving condition also establishes that the execution of *moveJoint* must only be performed if the parameter *Secure* of *DANGEROUSCHECKING* returns *true* (see code in Figure 10).




```
Weavings
   Smotion.DANGEROUSCHECKING
     (FTransStepsToAngle(NewSteps), Speed, Secure)
   beforeif (Secure = true)
   CoordJoint.moveJoint(NewSteps, Speed);
End_Weavings;
```

Figure 10: Weaving Definition among Aspects

Once the architect has finished modelling a PRISMA type model, the architect can proceed to generate the C# code corresponding to this model (see process 4, Figure 6). **PRISMA type models** are inputs of the transformations that must be executed to automatically generate part of the C# code of the application. To execute C# PRISMA AOSAs, the .NET platform-specific model for PRISMA has been hard-

wired in its PRISMANET middleware. The PRISMANET middleware is a software layer that sits above the .NET platform and allows the execution of PRISMA applications by offering the aspect-oriented functionalities that .NET does not directly provide. Thereby, PRISMANET implements the PRISMA model by extending the .NET technology with the integration of aspects.

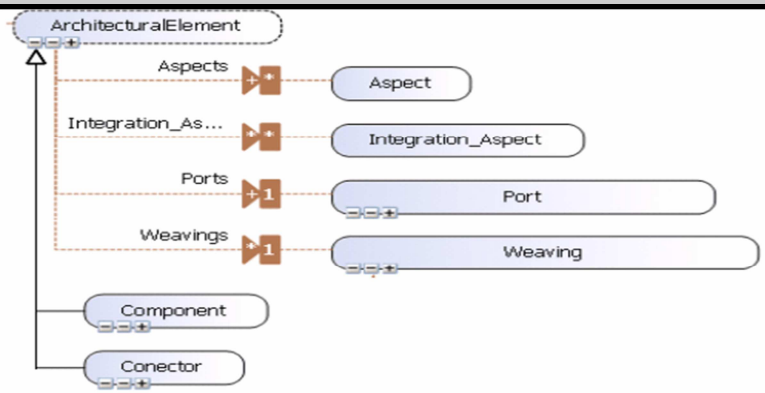
The transformation from models to code is performed using *the code generation patterns for types*. These patterns constitute a catalogue of thirty eight code generation patterns, structured as follows: the patterns PCS1- PCS19 are for generating C# and the patterns PADL1- PADL18 are for generating AOADL (see pattern PC15, Table 1). The patterns PADL1- PADL18 have the mappings between the PRISMA metamodel and AOADL. And the patterns PCS1- PCS19 have the mappings between the PRISMA metamodel and the .NET platform-specific model, i.e., the C# classes that allow instances to be executed on PRISMANET (see the example of the component *Actuator,* Table 1). Since each type that is defined in a PRISMA type model conforms to a metamodel concept, the execution of these mappings applied to the type generates its corresponding C# code. This transformation process (see Figure 6) also contains the mappings between the metamodel and the graphical metaphor because it has been defined in the process 3 of the MDD process (see Section 4.1.2). Hence, the patterns are applied to the graphical models by transitivity: if a concept *gx* of the graphical model corresponds with a concept of the metamodel *mx,* and the concept of the metamodel *mx* corresponds with a concept of the . NET platform-specific model *.NETx,* and a concept of the AOADL *lx,* then the concept *gx* corresponds with the concepts *.NETx* and *lx*. This transformation is possible thanks to the facility of DSL Tools "Transformation Templates", which supports the implementation of code generation patterns [Cook, 07]. The implementation of patterns consists in substituting the parameters of the patterns (see section *template*, Table 1) by the elements that the architect had modelled (see section *graphical representation,* Table 1). However, it is important to keep in mind that the result of this generation consists of reusable C# classes that still are not directly executable: they are architectural element types that must be instantiated to configure a specific system.

Pattern PCS15 presents the transformation template for architectural elements and shows an example of the pattern execution, the component *Actuator* (see section *result of the pattern execution*, Table 1). The pattern generates a component implemented as a serializable C# class (see Table 1, Pattern PCS15, Section 4.3, line 4) as the transformation template establishes (see Table 1, Pattern PCS15, Section 3.2, line 7). This class is *serializable* to enable its mobility in distributed versions of PRISMA CASE. Since the architectural element is a component and it is named *Actuator,* a public class called *Actuator* is generated as a subclass of the *ComponentBase* class of PRISMANET (see Table 1, Pattern PCS15, Section 4.3, line 5), as the template establishes for the generation of components (see Table 1, Pattern PCS15, Section 3.2, lines 6 and 8). This .NET class *ComponentBase* implements the .NET specific component behaviour of the PRISMA model. Then, the constructor of the class is created (see Table 1, Pattern PCS15, Section 4.3, line 6), as the template establishes (see Table 1, Pattern PCS15, Section 3.2, lines 12-13). Finally, the set of ports and aspects that make up a component are included by invoking the constructors of the *port* and *aspect* PRISMANET classes (see Table 1, Pattern PCS15, Section 3.2,

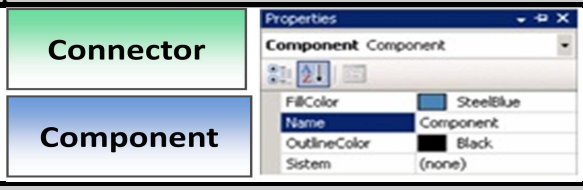
line 14). Both classes implement the port and aspect elements of the PRISMA model generating the ports and required aspects (see Table 1, Pattern PCS15, Section 4.3, lines 7-11).

Pattern PCS15: Simple Architectural Elements

1. PRISMA metamodel in DSL Tools



2.Graphical Representation



3.Transformation

3.1.Description

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 weavings are generated by other patterns related to this pattern.

3.2.Template

```
...
1 using System;  using System.Reflection; using PRISMA; using PRISMA.Aspects; using PRISMA.Aspects.Types;
2 using PRISMA.Components; using PRISMA.Middleware;
3 namespace <#=this.Model.Name#>
4 {
5 <#foreach (ArchitecturalElement architecturalElement in this.Model.ArchitecturalElements){
6  if (architecturalElement is Component || architecturalElement is Connector)  {#>
7  [Serializable]
8  public class <#=architecturalElement.Name#> : ComponentBase
9  <#    if (architecturalElement is Connector) {#>
10    , IConnector
11 <# } #>
12  {  public <#=architecturalElement.Name#>
13    (string name<#=ArchitecturalElementArguments(architecturalElement)#> ) : base(name) {
14<#/* Aspects */ /* Weavings */ /* Ports */ #>
15   }  }
16 <#
17 }/* endif (architecturalElement is Component || architecturalElement is Connector)*/...
```

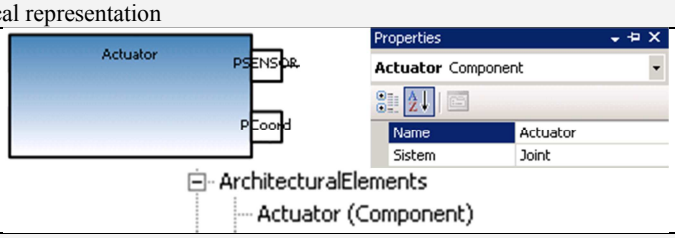
Pattern PCS15: Simple Architectural Elements

4. Case Study

4.1.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 as following.

4.2.Graphical representation



4.3. Result of the pattern execution

```
...
1 using System.Reflection; using PRISMA; using PRISMA.Aspects;
2 using PRISMA.Aspects.Types;using PRISMA.Components;using PRISMA.Middleware;
3 namespace RobotJoint
4 {[Serializable]
5   public class Actuator : ComponentBase{
6    public Actuator(string name ) : base(name){
7      /* Aspects */  AddAspect(new RS232 ()); /* Weavings */
8      /* Ports */InPorts.Add ("Pcoord", "IMotionJoint", INTMOVE);
9                OutPorts.Add ("Pcoord", "IMotionJoint", INTMOVE);
10                InPorts.Add ("PSENSOR", "IMotionJoint", OUTMOVE);
11                OutPorts.Add ("PSENSOR", "IMotionJoint", OUTMOVE);
12              }}}...
```

Related Patterns

Pattern PCS16: Aspects importation, **Pattern PCS17:** Weavings and **Pattern PCS18**: Ports.

Table 1: PRISMA Code Generation pattern PCS15: Simple Architectural Elements

4.2 From PRISMA Type Models to PRISMA Configuration Models

The definition of PRISMA configuration models from PRISMA type models comprises processes 5 and 6, illustrated in Figure 6. Next they are described.

4.2.1 Definition of PRISMA Configuration Models

Every **configuration model** must conform to a **PRISMA type model**. A specific configuration instantiates a generic architecture to specify a particular system. An example is the *TeachMover* robot or a specific joint of a robot. This step is easily developed thanks to the fact that PRISMA configurations are defined using the concepts that are defined in **its PRISMA type model** as modelling primitives. They are provided by PRISMA CASE, which automatically generates a domain-specific graphical modelling tool to configure specific software architectures from the type models. This is possible by generating a new DSL Tools project from the output of the PRISMA type model (see Figure 11). The generation of the domain-specific graphical modelling tool is launched from the PRISMA Type Modelling Tool (see Figure 8, menu PRISMA Tools).

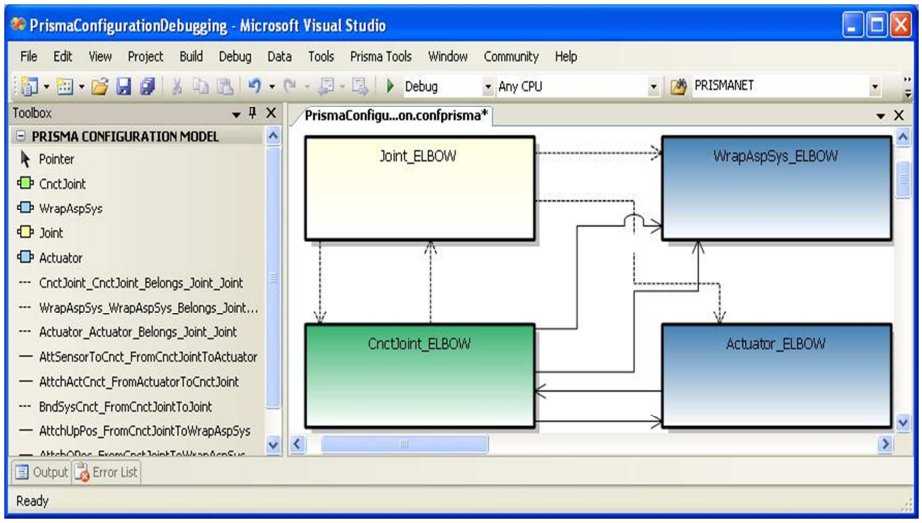


Figure 11: The architecture configuration of the joint Elbow

Figure 11 illustrates a domain-specific graphical modelling tool generated from the Type model of the *TeachMover*. Note that the elements of the toolbox are the components *Actuator* and *WrapAspSyst,* the connector *CnctJoint* and the connections (attachments and bindings) that were previously defined in the PRISMA type model of a Joint for Tele-Operatated Robots (see Figure 5 and Appendix A). In our example we have configured **the configuration model** of the *Elbow* of the *TeachMover* robot by dragging and dropping the primitives on the drawing sheet.

It is important to keep in mind that the **verification of architecture configurations** is also really important. It consists of checking that a configuration of instances satisfies the architectural model that it is an instance of (i.e., interconnections and compositions among instances are compliant with the interaction and composition patterns of the architectural model). In these verifications it is also considered the difference between **weak constraints** and **hard constraints** to support partial and complete verification as it is provided in PRISMA type models (see Section 4.1.3).

4.2.2 Integrating Types and Instances by means of Code Generation

Once the architect has finished the PRISMA configuration model, can proceed to generate the C# application corresponding to the **configuration model** (see process 5, Figure 6). The configuration model, together with the type model the code generated for these type model, are the inputs of the transformations that must be executed to automatically generate the final code of the application.

The transformation from models to code is performed using other patterns for architectural instances. In the case of C#, these patterns implement the mappings between a PRISMA type model (in our example, the Joint of a tele-operated robot) and the .NET platform-specific model (i.e., the C# primitives that allow PRISMA

AOSAs to be executed on PRISMANET). Since each instance that is defined in a PRISMA configuration model is an instance of a type of its PRISMA mode (e.g. the System instance *Elbow* is a System type *Joint* (see Figure 11), the mappings of the PRISMA model concept with PRISMANET and with the AOADL are executed generating its corresponding C# code. In this transformation step, the code obtained from process 5 of Figure 6 is completed by generating the C# instances. These instances can be launched and executed on PRISMANET. In order to do this, the PRISMA CASE executes the middleware PRISMANET and instantiates the defined configuration.

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 process 6, Figure 6). 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. 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 GUIs, they can do so.

5 Related Work

The combination of AOSD and software architectures has created two new challenges: (i) how to define the concept of aspect at the architectural level? and (ii) how to integrate aspects and architectural elements in a suitable way? This section analyzes the most relevant approaches that deal with these two questions and how they satisfy the main set of desirable properties that any aspect-oriented architecture MDD approach should fulfil (see Section 2.2.3). This section also discusses our contribution taken into account the state of the art of AOSAs, paying special attention to their role in the software development process and their MDD support.

5.1 Analysis of the main Aspect-Oriented Software Architectural approaches

Several approaches have emerged to integrate aspects and software architectures, as [Fiadeiro, 04] promoted, either by extending original ADLs with aspects or by creating new ADLs from scratch. AOSA approaches can introduce aspects in their ADLs in different ways: as a component (AspectLEDA [Navasa, 05], JAC [Pawlak, 04], Jiazzi [McDirmid, 03]); as a connector among components (CAM/DAOP [Pinto, 03]); as a view of the architecture (AVA [Katara, 03]), etc. On the other hand, there are approaches such as FuseJ [Suvée, 06], [Suvée, 05] that promote the idea that there are no aspects because aspects can be modelled using components such as FuseJ, AspectLEDA, JAC, Jiazzi and others. However, as Kizcales explains in his work [Kiczales, 97], the semantics of these concepts is different; i.e., components and aspects are not the same thing. This is due to the fact that aspects can be modelled as components, but components cannot usually be modelled as aspects since a component generally implements properties of more than one concern. Thanks to the introduction of AO techniques to all the stages of the software life cycle, it is possible to take advantage of the notion of aspect from the beginning of the system definition by specifying the aspects that are found in the requirements specifications. As a result, these aspects can be used throughout the rest of the development process as well as in

the maintenance process. However, approaches such as AspectLEDA and TranSAT [Barais, 03], [Barais, 04] require having an initial architectural specification of the system to introduce aspects. In these two approaches, aspects can only emerge as new requirements of the system that are hooked to the base architecture, i.e. aspects are only used for evolution during the maintenance process and not during the development process. TranSAT is an approach for managing the evolution of software architecture specifications using AOP principles. The fact that TranSAT concerns are only technical and not more generic is another drawback. Thus, TranSAT software architectures are defined using a pure compositional ADL, and aspects only appear as an extension or evolution mechanism of software architectures.

In the AVA approach, the complete view of the software architecture is lost because the use of the view notion is required to define aspects in its software architectures. AVA uses concerns as viewpoints to obtain those views that are the aspects of the software architecture. However, the use of architectural views from an AO point of view should be considered as an additional mechanism for analyzing software architectural features, instead of limiting the use of aspects to solely defining views. In fact, in software development not only is it possible to define views using aspects, but it is also possible to use common criteria such as roles, different stakeholders, etc. As a result, this approach loses flexibility and expressiveness to define views.

As Shaw presents in her work [Shaw, 94] , the specification of software systems with complex coordination protocols is too difficult without the connector architectural element. This is because the connector provides separation of component interactions, thereby achieving a higher level of abstraction, modularity, and a greater architectural view of the system. For this reason, connectors should be first-class citizens of an original ADL. However, there are many ADLs that do not provide the notion of connector. From these ADLs, many AOADLs such as CAM/DAOP, AspectLeda, Jiazzi, AOCE [Grundy, 99] or JAC have been defined. CAM/DAOP introduces aspects as connectors, JAC models aspects as components, and the approaches of AspectLEDA, Jiazzi and AOCE implement weavings as connectors in order to coordinate their "component aspects" (aspects are modelled as components and are called component aspects). Therefore, they use the connector as their new primitive to model aspects or weavings. It also introduces the weaving process inside their "component aspects" losing the reusability feature of aspects, since component aspects are dependent on the component aspects that they are connected to. In addition, ADLs should be formal languages because it is the only way to reason about the obtained software architectures, validate properties and generate code. However, we lose the advantages of using an extended language such as XML or Java, as in the CAM/DAOP and FuseJ proposals, respectively.

It is important to mention that a few of these approaches that combine AOSD and software architectures provide support for complete development of software; i.e. generating code from AOSA models. Therefore, they do not provide complete MDD support for developing AOSAs through a tool. The PCS approach is supported by the ConcernBase tool [Kande, 03]. This tool supports MDD in a partial way. It provides mechanisms for modelling software systems, and it also allows the translation of UML models to the SADL language [Moriconi, 97]. Technology independence is a clear advantage of this approach. However, at the same time, it is a drawback of PCS

because it does not provide support to translate its models to a programming language and they cannot be executed on a specific technological platform. The CAM/DAOP approach provides the DAOP platform [Pinto, 05]. It has been implemented in Java, and it provides a middleware 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. 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 of Fuentes et al. [Fuentes, 03] of CAM/DAOP is a first step to support MDD in the DAOP platform; however, complete support using code generation techniques for the development is not provided. It uses MDA to show the different views of the models that are specified in the platform. The TranSAT framework consists of a tool called SafArchie Studio [Barais, 03]. This tool offers several views of the evolution process depending on the kind of user. It only supports a tool to analyze the evolution of the software architecture and does not develop the aspect-oriented software architecture application. The MADE tool has been developed to support the AVA approach [Hammouda, 04]. It also shows the different views of the architecture, but it does not provide complete MDD support. AOCE has a tool to support its methodology, which extends the JViews [Grundy, 00] tool to support aspects. This tool is called JComposer [Grundy, 98]. JASCO provides two different tools. One of the JASCO tools transforms a Java bean into a JAsCo bean, and the other one integrates JAsCo with the PacoSuite [Wydaeghe, 01], which allows component models to be modelled at a high abstraction level and also allows one or more JAsCo connectors to be generated from its models. However, the JComposer and the pair of tools that support the JAsCo approach are dependent on technology. They are, in fact, implementation frameworks.

5.2 Comparison of the main Aspect-Oriented Software Architectural approaches

Two comparison tables[2] have been defined from the criteria established in Section 2.2.3 and the approaches studied in Section 5.1 to analyze and compare them (see Table 2 and Table 3). After the analysis and comparison of different approaches of AOSA, it may be concluded that, at the architectural level, these proposals usually extend ADLs without connectors and mainly follow an asymmetric model by considering functionality as architectural components. They always introduce the notion of aspect by using original architectural concepts instead of providing the suitable semantics for aspects. Furthermore, despite the fact that there has been a lot of work done, these proposals are only focused on a specific purpose: the analysis, evolution or development of software architectures. And, they do not pursue several purposes simultaneously to provide complete development and maintenance support.

[2] 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 ≅ connectors	SafArchie component model
ASAAM	Asymmetric	Not fixed	Scenarios	Outside Aspects	Not fixed
AVA	Asymmetric	Not fixed	Aspects as views	Outside Aspects	Not fixed
Aspect LEDA	Asymmetric: Two levels: Aspects and architectures	Without connectors	Aspects as components	Outside aspects using coordinators ≅ connectors	Leda: Formal Compositional ADL
AOCE	Asymmetric	Without connectors	Aspects as components	Outside aspects using aspect managers ≅ 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 2: First comparison of aspect-oriented software architecture approaches

	Aspect-Oriented Evolution	Purpose	Technology	Graphical support	Tool support
PCS		Development of AO Software Architecture	Independent	UML profile: Aspect is a stereotype of a UML class	ConcemBase tool: modelling support, ADL generation from UML, no code generation, no execution
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
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
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
Aspect LEDA		Development of AO Software Architecture	Independent		
AOCE	Dynamic weaving	Development of AO Software Architecture	Dependent on JViews		JComposer: An extension of the JViews tool
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

	Aspect-Oriented Evolution	Purpose	Technology	Graphical support	Tool support
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
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		
JIAZZI		Programming aspect-oriented Java applications	Dependent on Java technology		

Table 3: First comparison of aspect-oriented software architecture approaches

Therefore, it may be concluded that an approach of AOSAs for symmetric AOMs and ADLs with connectors that follows the MDD paradigm should be defined in such a way that the starting premise would be fulfilled. This approach should include: (i) a suitable semantics for the aspect concept; (ii) a graphical modelling metaphor, (iii) technological support in order to execute the aspect-oriented architectural models that have been defined independently of technology, (iv) guided support throughout the development and maintenance processes of software following MDD: Reusability, Verification, Code generation, Maintenance, Evolution, etc. The PRISMA MDD approach was defined to fulfil these needs by completing the Tables 2 and 3 as follows:

	Aspect-oriented model	Architectural model	Definition of Aspects	Definition of Weavings	ADL
PRISMA	Symmetric	With Connectors	Aspects	Outside aspects and inside architectural elements	Formal AOADL

	Aspect-Oriented Evolution	Purpose	Technology	Graphical support	Tool support	MDD support
PRISMA	Dynamic weaving and support for adding and removing aspects	Development and Maintenance of aspect-oriented software architectures	Technology independent	Graphical AOADL	PRISMA CASE	Complete MDD support

6 Experimental Results: PRISMA as an step forward in the MDD process of Aspect-Oriented Software Architectures

This PRISMA MDD process has been applied to different kind of applications to validate its feasibility. It has been used for the design and code generation of academic examples such as, banking systems, auctions, etc. But also, it has been put into practice for the generation of real applications that are deployed in the industry. Specifically, PRISMA has been put into practice in the robotic domain. In particular, the PRISMA MDD process has been applied to the development of the robots

TeachMover [Pérez,08a] and *Agrobot* [Costa-Soria, 11], as a proof-of-concept for this MDD process.

6.1 Validation Goal and Variables of measurement

The validation of our approach has been focused on the code generation capability of the PRISMA MDD process. The analysis of this capability has been driven by following question: *Is the PRISMA MDD process able to automatically generate the executable C# code of applications from Aspect-Oriented Software Architectural models?*.This main question has been refined into two more concrete questions:

Q1) *Is the code generation complete or partial?*, and

Q2) *Is the code generation ready to be executed or it requires any update for its execution?*.

These two questions aims to find out the degree of automation of the process and allow us to define the response variables for quantitatively measuring the PRISMA MDD process. These variables are the following: Percentage of Generation (%G), Percentage of Manually Programmed Code(%MPC), Percentage of Updated Code (%UC), Number of Generated Lines of Code (NGLC), Number of Manually Programmed Lines of Code (NMPLC), Number of Updated Lines of Code (NULC), Number of Generated Classes (NGC), Number of Manually Programmed Classes (NMPC), Number of Updated Classes (NUC), Number of Generated Interfaces (NGI), Number of Manually Programmed Interfaces (NMPI), and Number of Updated Interfaces (NUI) . In addition, other variables in terms of aspect-oriented software architectures concepts has been collected: Number of Lines of Code for Interfaces (NLCI), Number of Lines of Code for Aspects (NLCA), Number of Lines of Code for Architectural Elements (NLCAE), Number of Lines of Code for Components (NLCC). Number of Lines of Code for Conectors (NLCCN), and Number of Lines of Code for Systems (NLCS).

6.2 Description of the Robotic Applications.

In this paper, we have used part of the architecture of the tele-operated robot *TeachMover* to illustrate the PRISMA MDD process. The detailed description of the functionality of this tele-operated robot is described in Section 3. The complete architecture of the robot TeachMover consists of 16 architectural elements (5 Systems, 6 components and 5 connectors), and 16 aspects (6 functional aspects, 5 safety aspects and 5 coordination aspects) [Pérez, 08a].

In addition, the PRISMA MDD approach has been applied to the code generation of the agriculture robot *Agrobot* [Costa-Soria, 11]. The Agrobot is conceived as an autonomous, small-sized robot, which objective is to patrol –at periodical intervals- a small field or delimited area, looking for pests or disease attacks over a set of growing crops. When a threat is detected, a pesticide is applied to the field, as a first counter-attack measure, and an alarm is sent to the manager to take further specialized actions. The use of small agricultural robots (small-sized, light-weight, and autonomous) [Blackmore, 06] is being encouraged in the Spanish agriculture sector to reduce the high labour involved and the production costs of plague control. The *Agrobot* architecture is hierarchically defined as a composition of Systems. The top-level architecture consists of 12 Systems and 10 connectors. From these systems, the MDD

process has been applied to one of them, the System which allowed us to validate its behaviour without being geographically in an agriculture field: the Vision System [Costa-Soria, 2011]. The Vision System captures and pre-filters images from the environment, which are used by other Systems of the Agrobot to look for crop diseases and/or guide the movement. This System has been completely modelled using the PRISMA CASE tool, and its code has been automatically generated. This System consists of 8 architectural elements (1System, 4 components, and 3 connectors), 8 aspects (4 functional aspects, 3 coordination aspects, and 1 presentation aspect) and 3 interfaces.

6.3 Analysis of the Experimental Results

The PRISMA MDD process has been applied to the development of both robots. The results obtained from the measurement of the variables identified in Section 6.1 are described in Tables 4 and 5.

	%G	%MPC	%UC	NGC	NMPC	NUC	NGI	NMPI	NUI
TeachMover	94,3	5,7	0	32	2	0	19	0	0
Agrobot	68,1	20,09	10,90	16	2	4	3	0	0

Table 4: Result of the robot TeachMover and Agrobot in percentages, nº classes and nº interfaces

	NGLC	NMPLC	NULC	NLCI	NLCA	NLCAE	NLCC	NLCCN	NLCS
TeachMover	5686	345	0	146	5215	325	99	150	65
Agrobot	1125	345	180	28	1388	234	109	84	41

Table 5: Results of the robot TeachMover and Agrobot in Lines of Code

The results presented in Tables 2 and 3 reveals a high percentage of automatic-code generation from PRISMA AOSA models: 94,3% for the *TeachMover* and 68,1% for the *Agrobot*, which correspond with 5686 and 1125 Generated Lines of Code (NGLC), respectively (see Tables 4 and 5). The percentage of lines of code manually programmed (NMPLC) of both applications correspond to the 345 lines of code that implement the Generic Graphical User Interface (GUI) of PRISMA (see Tables 4 and 5). This GUI allows the execution of any PRISMA application, and both robots are executed using this GUI. As a result, both robots have the same number of lines manually programmed, and the same Manually Programmed Classes (NMPC) (see Tables 2 and 3). Since this number is constant, the percentage of generation varies depending on the extension of the system. In this case, as the TeachMover is more extensive than Agrobot, the percentage of automatic generation is higher.

On the other hand, the number of classes (see Tables 2 and 3) illustrates that the PRISMA MDD process generates one class for each concept modelled, e.g. the 32 generated classes (NGC) of the TeachMover (see **Fehler! Verweisquelle konnte nicht gefunden werden.**) correspond to the 16 architectural elements and 16 aspects of its PRISMA aspect-oriented architectural model.

With regard the number of lines of code that have been generated, most of them are related to the body of aspects, which define state (attributes), services (methods), and protocols (the valid transitions among services).

As a result of the modelling and code generation process of the Vision System of the Agrobot, the experiment showed that some behaviour could not be modelled and must be directly introduced on the generated code. This was the case for low-level behaviour: the code responsible for interacting with the image capturing device, and the algorithms for processing and filtering the images captured. In this case, the approach was to replace the body of the services automatically generated with the specific, low-level code. This resulted in 180 lines of code updated (see NULC in Table 5), which in total represented only the 10,9% of the total amount of code (see %UC in Table 3). This is significantly better that developing all the code from the beginning.

These code generation results reveal that the complete automatic-code generation from AOSA models is feasible. In this contribution we provide an approach that takes a step further in the automatic code generation from aspect-oriented software architectures. The PRISMA MDD process is able to generate the code of the business logic, but is not able to automatically generate its GUI. This establishes the next step to improve this MDD process and its modelling framework.

6.4 Evaluation of validity and limitation

This approach has been validated by automatically generating the code of two real applications from their aspect-oriented architectural models. However, it is difficult to generalize the results because, as any experiment, there is a set of factors that affects the results of the response variables. In this case, the main factors that have been identified are: 1) the project size, 2) the architecture complexity, and 3) the concerns modelled.

The experimentation field recommends to intentionally vary these factors to mitigate the variations that could impact the results and facilitate the generalization of these results [Juristo, 01]. Following this guideline, we have applied the PRISMA MDD process to two applications with different size and complexity. However, to completely mitigate these results, it is necessary to apply the process to other domains, in which other concerns should be required, such as persistence, graphical-user interface or privacy in information systems or social network services.

7 Conclusions and Future Work

In this paper, the PRISMA MDD process of PRISMA is presented as an important advance in the automatic generation of code from AOSAs. This contribution describes this process and each one of their generation steps to serve as a first guidance for the MDD of AOSAs. This process also defines two steps of transformation that are supported by generation patterns that allow the generation of AOADL specification and C# code. Hence, this contribution provides a pattern template to describe the transformations of code generation. Moreover, the need to support model verification throughout the MDD process is established by defining not only the kind of verifications (**partial, incremental, complete)**, but also the kind of

constraints (**hard constraints** and **weak constraints)** to consider. It is important to emphasize that the refinement from the **types model** to the **configuration model** of PRISMA MDD process provides a **domain-specific model** to configure PRISMA software architectures. This domain-specific model is important because it reduces the gap between the user and his/her knowledge about modelling during the MDD process.

This PRISMA MDD process has been materialized into the PRISMA CASE framework, which supports each one of the steps that the process establishes. PRISMA CASE has allowed us to apply the process to the *TeachMover* and *Agrobot* robots. In this paper, we present the code generation results of both applications and the analysis reveals that the complete automatic-code generation from AOSA models is feasible with the presented MDD process,

PRISMA is a new approach that opens a perfect setting for further research. All the parts that the PRISMA approach is composed of can be extended in order to face new challenges. This MDD process can be enriched by introducing more layers of refinement or defining new transformations for other languages, models or platforms and/or developing abstract middleware that would hide the differences between the different platforms. Another extension to this MDD process is the incorporation of COTs throughout the process. 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. Finally, it is necessary to evaluate PRISMA using applications of other domains, which could allow us to generalize the obtained results..

Acknowledgements

The work reported here has been partially sponsored by the Spanish MEC projects (DSDM TIN2008-00889-E and MULTIPLE TIN2009-13838), and MICINN (INNOSEP TIN2009-13849)

References

[Aksit, 05] Aksit, M., Systematic analysis of crosscutting concerns in the model-driven architecture design approach. Symposium How Adapatable is MDA?, 2005.

[Amaya, 05] 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.

[Barais, 04] Barais, O., Cariou, E., Duchien, L., Pessemier, N., & Seinturier, L., Transat: A framework for the specification of software architecture evolution. The 1st Int. Workshop on Coordination and Adaptation Techniques for Software Entities, Oslo, 2004.

[Barais, 03] Barais, O., Duchien, L., & Pawlak, R, Separation of Concerns in Software Modeling: A Framework for Software Architecture Transformation. IASTED Int. Conf. on Software Engineering Applications, ACTA Press, pp. 663-668, Los Angeles, CA, USA, 2003

[Beydeda, 05] Beydeda, S., Book, M., & Gruhn V., Model-Driven Software Development, Springer, 2005.

[Blackmore, 06] B.S. Blackmore, H.W. Griepentrog, S. Fountas. Autonomous Systems for European Agriculture. In proc. of Automation Technology for Off-Road Equipment (ATOE). Bonn, Germany, 2006.

[Chitchyan, 05] Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Pinto, M., Bakker, J., et al.: Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. AOSD-Europe Deliverable D11, 2005.

[Cook, 07] Cook S., Jones G., Kent S., Cameron Wills A., Domain-Specific Development with Visual Studio DSL Tools, ISBN-10: 0-321-39820-3, Addison-Wesley, 2007.

[Costa-Soria, 11] Costa-Soria, Cristobal: Dynamic Evolution and Reconfiguration of Software Architectures through Aspects. PhD Thesis.Universidad Politécnica de Valencia, June 2011.

[Cuesta, 05] Cuesta, C., Romay, M.P., De La Fuente, P., & Barrio-Solórzano, M., Architectural Aspects of Architectural Aspects. 2nd European Workshop on Software Architecture, LNCS 3527, Pisa, 2005.

[Cuesta, 02] Cuesta C.E.. Dynamic Software Architecture based on Reflection. PhD. Thesis, Dpt. of Computer Science, University of Valladolid, 2002. (In Spanish)

[Dijkstra, 76] Dijkstra, E., *A Discipline of Programming*. Prentice-Hall, 1976.

[EFTCoR, 02] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794, 2002.

[Fernández, 05] 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.

[Fiadeiro, 04] Fiadeiro, J.L., & Lopes, A, CommUnity on the Move: Architectures for Distribution and Mobility. FMCO 2003, LNCS3188, pp. 177–196. Springer Heidelberg, 2004.

[Fuentes, 03] Fuentes. L., Pinto. M., & Vallecillo A. How MDA can help designing component- and aspect-based applications. EDOC, pp. 124-135, 2003.

[Garlan, 95] Garlan, D., Perry D., Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, vol. 21 no. 4, April 1995.

[Grundy, 00] Grundy J.. Multi-perspective specification, design and implementation of software components using aspects. Int. Journal of Software Engineering and Knowledge Engineering, vol. 20, 2000.

[Grundy, 99] Grundy, J. Aspect-Oriented Requirements Engineering for Component-based Software Systems. The 4th IEEE Int. Symp. on RE, 1999.

[Grundy, 98] Grundy, J.C., Mugridge, W.B., & Hosking, J.G.. Static and dynamic visualisation of component-based software architectures. The 10th Int. Conf. on Software Engineering and Knowledge Engineering, KSI Press, San Francisco, California, USA, 1998

[Hammouda, 04] Hammouda, I., Koskinen, J., Pussinen, M., Katara, M., & Mikkonen, T., Adaptable Concern-Based Framework Specialization in UML. Automated Software Engineering, pp. 78-87, Linz, Austria, 2004.

[Harrison, 02] Harrison, W., Ossher, H., & Tarr, P., Asymmetically vs Symmetrically Organized Paradigms for Software Composition. IBM T.R. RC22685 (W0212-147) Thomas J. Watson Research Center, IBM, 2002.

[Juristo, 01] Juristo N., Moreno A.M., Basics of software engineering experimentation, Kluwer, 978-0-7923-7990-4, pp.1-395, 2001.

[Kande, 03] Kande M., A concern-oriented approach to software architecture. PhD. Thesis, Lausanne, Switzerland: Swiss Federal Institute of Technology (EPFL), 2003.

[Katara, 03] Katara, M, & Katz, S., Architectural Views of Aspects. The Int. Conf. on AOSD, ACM Press, 2003.

[Kiczales, 01] Kiczales, G., Hilsdale, E., Huguin, J., Kersten, M., Palm, J., & Griswold W.G., An Overview of AspectJ. The 15th European Conf. on Object-Oriented Programming, LNCS 2072, Budapest, Hungary, 2001.

[Kiczales, 97] Kiczales, G., Lamping, J., Mendekar, A., & Maeda, C., Aspect-Oriented Programming. The 11th European Conf. on Object-Oriented Programming, LNCS-1241, Jyväskylä, Finland, 1997.

[Kruchten, 95] Kruchten P., The 4+1 View Model of Architecture. IEEE Software, Vol. 12, no. 6, November, 1995.

[Kulkarni, 03] Kulkarni, V., & Reddy S., Separation of concerns in model-driven development, IEEE software 20(5), 2003.

[McDirmid, 03] McDirmid, S., & Hsieh, W.C.. Aspect-Oriented Programming with Jiazzi. The 2nd Int,. Conf. on Aspect-Oriented Software Development (AOSD), Boston, Massachusetts, pp. 70-79, 2003.

[Medvidovic, 00] Medvidovic, N., & Taylor, R.N., A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering,Vol. 26, No. 1, 2000.

[Meyer, 03] Meyer B., The Grand Challenge of Trusted Components. International Conference on Software Engineering (ICSE), IEEE Computer Press, Portland, Oregon, May 2003.

[Meyer, 98] Meyer, Bertrand, Object-Oriented Software Construction, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[MOF, 12] Object Management Group (OMG), "Meta-Object Facility (MOF) Specification 2.0 TR formal-06-01-01," http://www.omg.org/spec/MOF/2.0/PDF/.

[Moriconi, 97] Moriconi, M., & Riemenschneider, R. A., Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI Int., 1997.

[Navasa, 05] Navasa, A., Pérez, M.A., & Murillo, J.M., Aspect Modelling at Architecture Design. Second European Workshop on Software Architecture, LNCS 3527, Springer, Pisa, 2005.

[NET,12] .NET Technology, Microsoft Corporation, http://www.microsoft.com/net

[Parnas, 85] Parnas, D. L. 1985. Software aspects of strategic defense systems. Commun. ACM 28, 12 (Dec. 1985), 1326-1335. DOI= http://doi.acm.org/10.1145/214956.214961

[Parnas, 72] 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.

[Pawlak, 04] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F. , & Martelli, L, JAC: an Aspect-Based Distributed Dynamic Framework. Software Practice and Experience, V. 34, pp.1119-1148, 2004.

[Pérez, 08a] Pérez, J., Ali, N., Carsí, J.A., Ramos, I., Álvarez, B., Sánchez, P., Pastor, J.A., Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. Information & Software Technology 50(9-10): 969-990, 2008.

[Pérez, 06a] Pérez, J. PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Polytechnic University of Valencia, 2006.

[Pérez, 06b] Pérez, J., Ali, N., Carsí, J.A., & Ramos, I., Designing Software Architectures with an Aspect-Oriented Architecture Description Language. The 9th Int. Symp. on Component-Based Software Engineering, LNCS 4063, Västerås, Sweden, 2006.

[Perry, 92] Perry, D., & Wolf, A., Foundations for the Study of Software Architecture. ACM Software Engineering Notes, Vol. 17, No 4, p 40-52, 1992.

[Pinto, 05] Pinto, M., Fuentes, L., Troya, J.M., A Dynamic Component and Aspect Platform, The Computer Journal Vol. 48, No. 4, pp. 401-420, 2005.

[Pinto, 03] Pinto, M., Fuentes, L, & Troya, J.M., DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. The Int.Conf. on Generative Programming and Component Engineering, LNCS 2830, Springer, 2003.

[Schmidt, 06] Schmidt D.C., Model-Driven Engineering, IEEE computer Society, 2006.

[Shaw, 94] Shaw, M.,B, Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. Workshop on Studies of Software Design, 1994.

[Simmonds, 05] Simmonds, D. et al, , An aspect oriented model driven framework. 9th IEEE Int. EDOC Enterprise Computing Conf. pp. 119-130, 2005

[Suvée, 06] Suvée, D., De Fraine B., & Vanderperren, W., A symmetric and Unified Approach Towards Combining Aspect-Oriented and Component Based Software Development. 9th Int. Symp. on Component-Based Software Engineering, LNCS 4063, Västerås, Sweden, 2006.

[Suvée, 05] Suvée, D., Vanderperren, W., Wagelaar, D., & Jonckers, V., There Are No Aspects. ENTCS, Special Issue on Software Composition, Vol. 114, pp. 153-174, 2005.

[Szyperski, 98] Szyperski C., Component software: beyond object-oriented programming. ACM Press and Addison Wesley, New York, USA (1998).

[TeachMover, 12] The TeachMover Robot, http://www.microbotzone.com/TeachMover/TeachMover/tabid/3648/Default.aspx

[Wydaeghe, 01] Wydaeghe, B., Vanderperren, W., Visual Component Composition Using Composition Pattems. Tools, Santa Barbara, California, 2001.

Appendix A: Model-Driven Development of Aspect-Oriented Software Architectures

In this appendix, we show in detail the complete architecture of the running example that has been used in the paper for illustrating the PRISMA MDD process. This is the PRISMA type model for a generic joint of a tele-operated robot that can be reused for designing different joints of the same robot or joints of different robots.

Figure 13 illustrates how the components, *Actuator* and *WrapAspSyst,* are coordinated through a connector *CnctJoint* (see Figure 12), how each component imports their functionality through aspects (see Figure 17, **Fehler! Verweisquelle konnte nicht gefunden werden.**, and **Fehler! Verweisquelle konnte nicht gefunden werden.**), and how the connector *CnctJoint* imports its behaviour through a coordination and a safety aspect, *CoordJoint* and *SMotion* respectively*,* to define its behaviour (see Figure 16). In addition, aspects import their corresponding interfaces, which are published through the architectural element ports that import these aspects (see Figure 14 and Figure 15). Finally, it is important to mention the need for a weaving emerges due to the fact that a joint is moved only after the connector safety constraints are satisfied assuring that a movement is safe (see Figure 16).

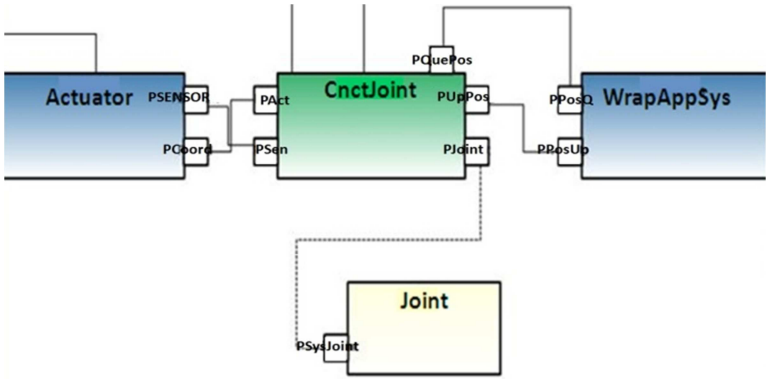


Figure 12: The joint architecture of a tele-operated robot

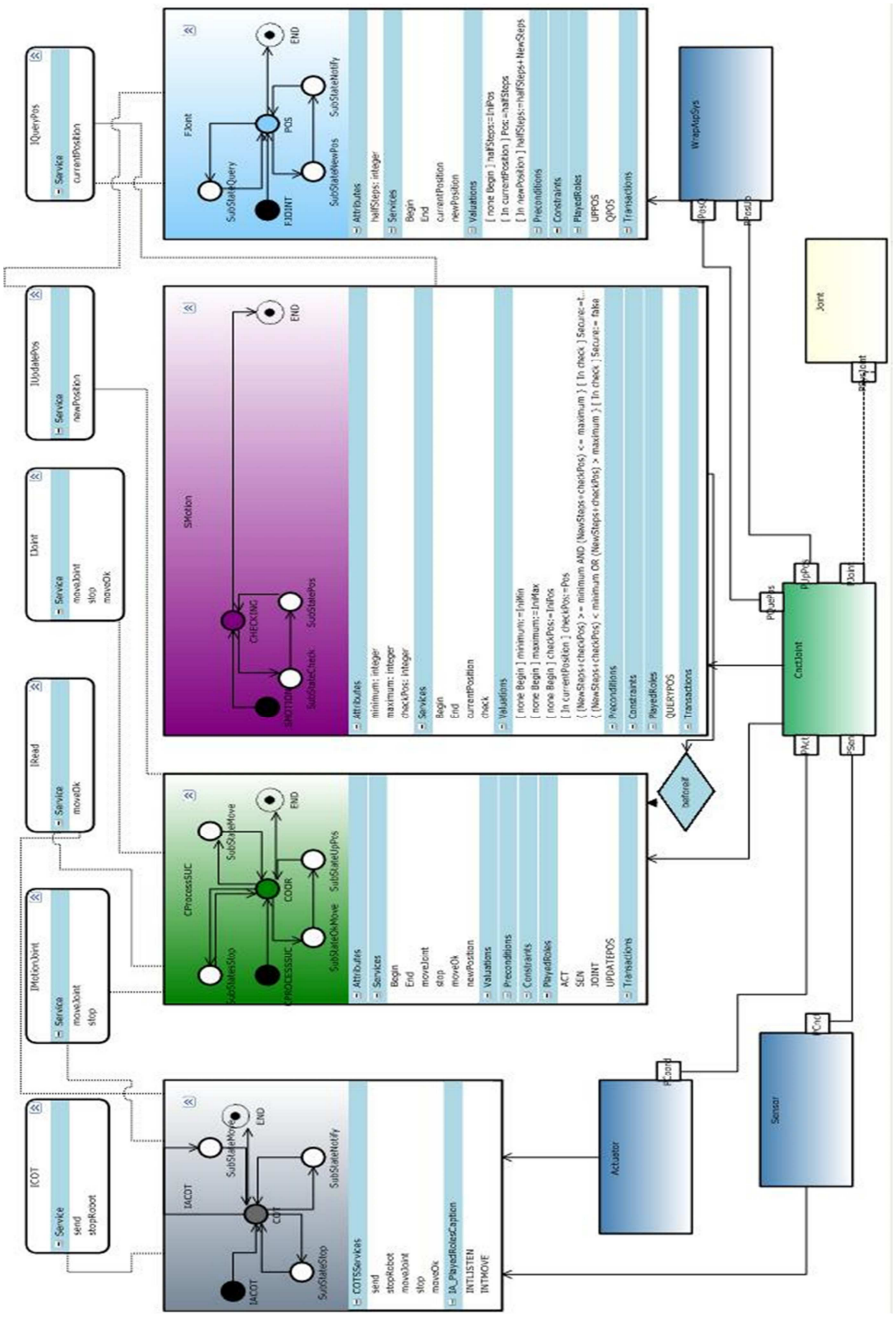


Figure 13: The PRISMA System Aspect-Oriented Architecture of a Joint for Tele-Operated Robots

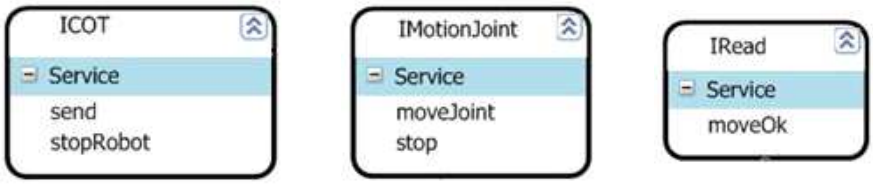


Figure 14: Interfaces ICOT, IMotionJoint and IRead

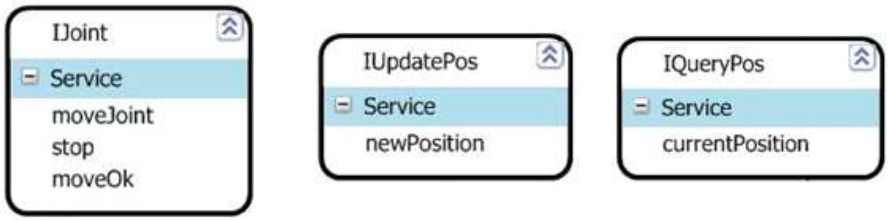


Figure 15: Interfaces IJoint, IUpdatePos and IQueryPos

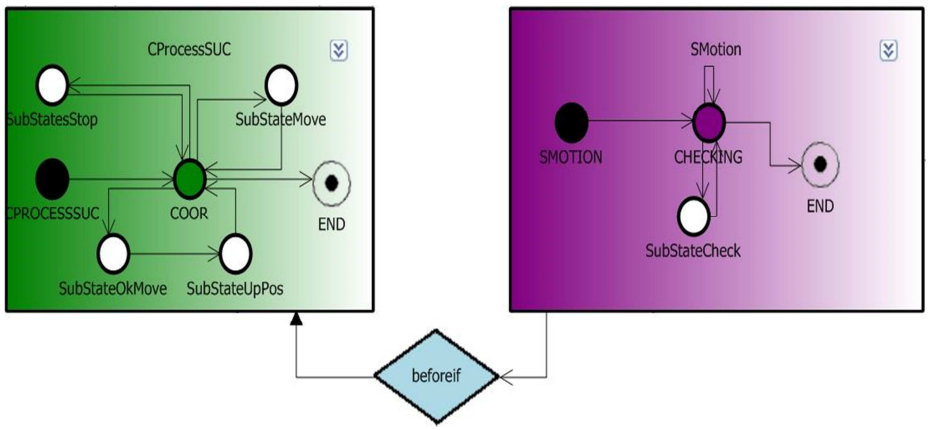


Figure 16: Weaving definition between the Aspects CProcessSuc and SMotion (shown in collapsed format)

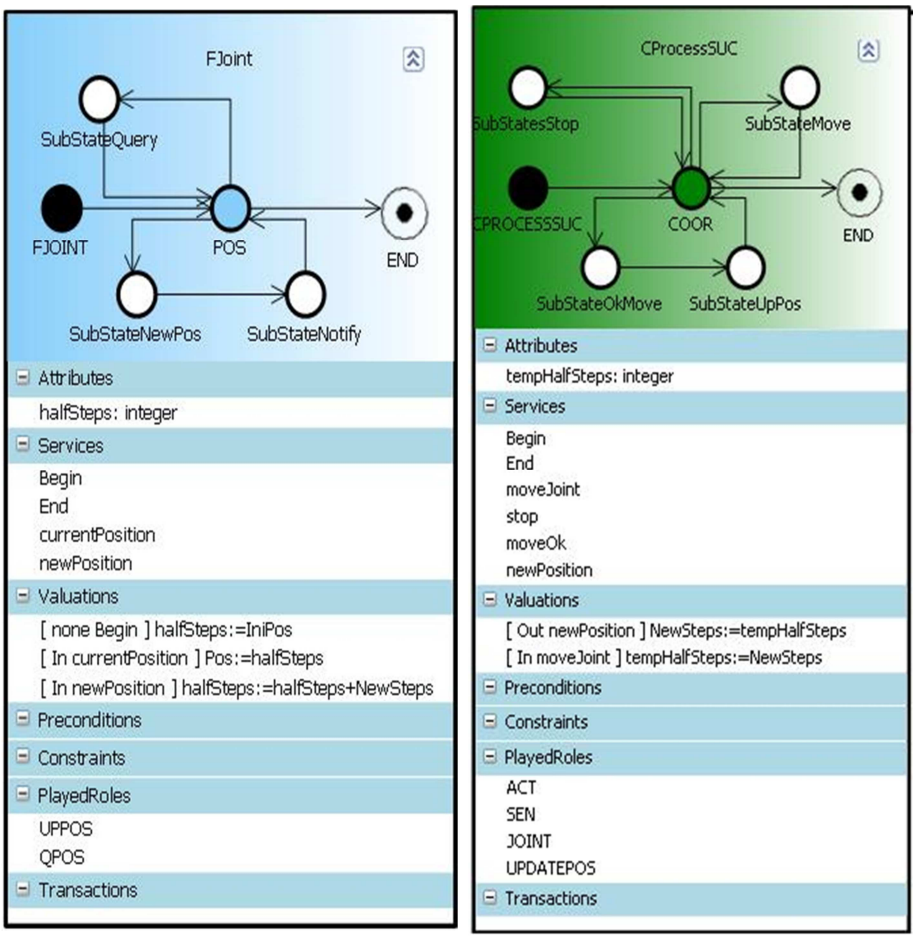


*Figure 17: Left: Functional Aspect **Fjoint** (shown in expanded format)*
*Right: Coordination Aspect **CProcessSUC** (shown in expanded format)*

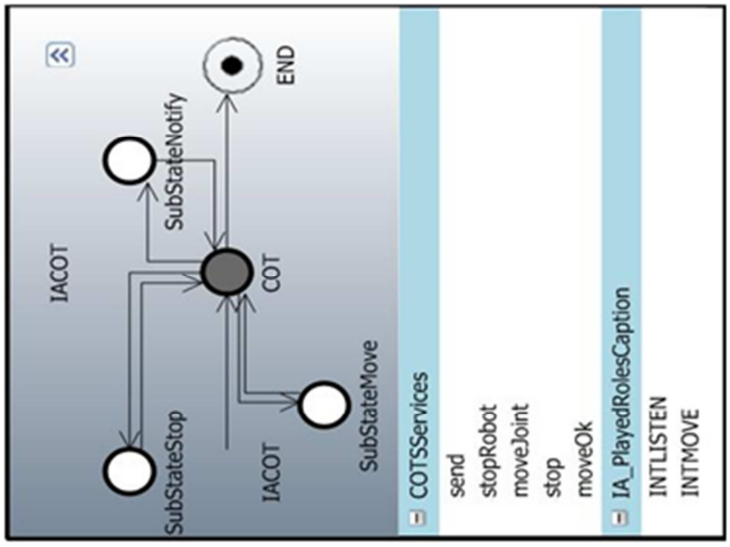


Figure 18: Functional Aspect IACOT (shown in expanded format)

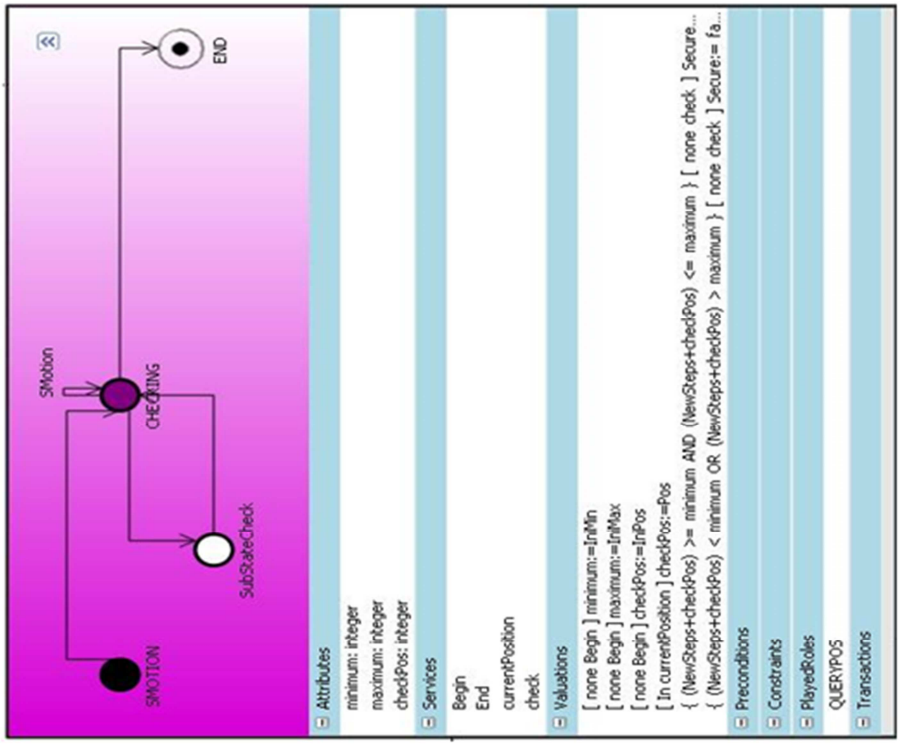


Figure 19: Safety Aspect SMotion (shown in expanded format)