Dynamic Adaptation of Aspect-Oriented Components

Cristóbal Costa[1], Jennifer Pérez[2] and José Ángel Carsí[3]

[1,3]Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n, 46022 Valencia, Spain

[2]Department of Organization and Information Structure
Polytechnic University of Madrid
Ctra. Valencia, km7, 28051 Madrid, Spain

ccosta@dsic.upv.es, jeperez@eui.upm.es, pcarsi@dsic.upv.es

Abstract. Current works address self-adaptability of software architectures to build more autonomous and flexible systems. However, most of these works only perform adaptations at configuration-level: a component is adapted by being replaced with a new one. The state of the replaced component is lost and related components can undergo undesirable changes. This paper presents a generic solution to design components that are capable of supporting runtime adaptation, taking into account that *component type* changes must be propagated to its instances. The adaptation is performed in a decentralized and autonomous way, in order to cope with the increasing need for building heterogeneous and autonomous systems. As a result, each component type manages its instances and each instance applies autonomously the changes. Moreover, our proposal uses aspect-oriented components to benefit from their reuse and maintenance, and it is based on *MOF* and *Reflection* concepts to benefit from the high abstraction level they provide.

Keywords: runtime adaptation, dynamic evolution, component adaptability, reflection, CBSD, software architectures, AOSD

1 Introduction

Complex software systems frequently undergo changes during their lifetime. This is due to the fact that they are exposed to many sources of variability and also have a dynamic nature. It is very common for unforeseen bugs to appear during system execution, and they will have to be corrected at run-time. In order to address this software adaptability, most approaches modify the subsystem that must be updated offline, and once the modification has been completed, they restart the entire system to reflect the new changes. However, this solution has several disadvantages: (i) the state of the system that is running is lost, unless it has been previously saved; (ii) the shutdown and restart processes of the system could increase the performance cost; (iii) a lot of complex systems cannot stop their activity (such as servers or real-time systems). As a result, a solution that overcomes these disadvantages must be provided.

This work focuses specifically on two approaches of software development that improve the reuse, maintenance and adaptability of software. They are the

Costa, C., Pérez, J., Carsí, J.A.: Dynamic Adaptation of Aspect-Oriented Components.
In: Schmidt, H.W., Crnkovic, I., Heineman, G.T., Stafford, J.A. (eds.) Component-Based Software Engineering 2007.
Lecture Notes in Computer Science, vol. 4608, pp. 49-65. Springer, Heidelberg (2007)
ISSN 0302-9743 - ISBN 978-3-540-73550-2

Component-Based Software Development (CBSD) approach [10, 33] and the Aspect-Oriented Software Development (AOSD) approach [15, 16]. CBSD reduces the complexity of software development and improves its maintenance by increasing software reuse and independence. CBSD decomposes the system into reusable and independent entities called components. By extension, these advantages are provided by software architectures [26, 30] since architectural models are constructed in terms of components and their interactions.

Both software architectures and AOSD facilitate software adaptation. On the one hand, software architectures allow us to focus on changes at the component level instead of changes at the implementation level. Software architecture adaptability is managed in terms of the creation and destruction of component instances and their links. This kind of adaptation is called *Dynamic Reconfiguration* or *Structural Dynamism* [8], which has been explored by a lot of research works [3]. However, most of these works only address dynamic adaptation at the architectural level. The internal adaptation of running components is not considered: a component is adapted offline and then the old component (which is running) is replaced with the adapted one. In several cases, runtime component replacement is not enough if the preservation of the component state is mandatory; for instance, when a component needs to be extended with new properties (such as security, persistence, etc.) and its functional state and properties are not modified. In order to support self-adaptability of software architectures, a solution that allows us to both internally update a component at runtime and to preserve the old state not subject to change must be provided.

On the other hand, Aspect-Oriented Models propose the separation of the crosscutting concerns of software systems into separate entities called *aspects*. This separation avoids the tangled concerns of software and allows the reuse of the same aspect in different entities of the software system (objects, components, modules, etc.). This separation of concerns also improves the isolated maintenance of the different concerns of the software systems. Aspect-Oriented Software Development (AOSD) [15] extends the advantages that aspects provide to every stage of the software life cycle. For this reason, several proposals for the integration of the aspects in software architectures have emerged [5, 6]. AOSD allows us to manage changes from the different concerns in an independent way. This facilitates evolution and provides flexibility to adapt components by adding or removing aspects to software architectures. In addition, interaction policies between different concerns of software architectures can be easily changed by adding or removing aspect synchronizations.

This paper proposes a solution to support the dynamic component adaptation of aspect-oriented software architectures. Specifically, the solution is based on the PRISMA approach, which combines AOSD with software architectures. The dynamic adaptation proposal that is presented in this paper provides mechanisms for dynamically changing the internal structure of PRISMA components. This internal change preserves the component state and minimizes the impact of the change on other components that are connected to the updated component. In addition, this proposal takes into account that *component type* changes must be propagated to each one of its running instances, so the needed mechanisms are also described.

This paper is structured as follows. The PRISMA approach is introduced in section 2. Section 3 presents the two concepts on which our proposal is based: *MOF* and

Computational Reflection. In section 4, our proposal to support dynamic adaptation of *component types* is presented in detail. Related works that address runtime component adaptation are discussed in section 5. Finally, conclusions and further works are presented in section 6.

2 PRISMA

PRISMA is an approach to develop technology-independent, aspect-oriented software architectures [22]. It integrates software architecture and AOSD in order to take advantage of the two approaches. The PRISMA approach is based on its model [24] and its formal Aspect-Oriented Architecture Description Language (AOADL) [23].

In order to define our software adaptation proposal, we chose PRISMA from the different approaches that combine AOSD and software architectures because of the advantages that it offers. Its main advantages are the following: (1) components and aspects are independent of each other, so they provide good properties for dynamic evolution; (2) the PRISMA model is completely formalised and its AOADL is a formal language, so the evolution requirements of our proposal can be easily formalized; (3) PRISMA software architectures can be automatically compiled for a technological platform and a programming language using code generation techniques; and (4) a PRISMA tool has been developed to cope with the challenge of developing aspect-oriented software architectures following the Model-Driven Development (MDD) paradigm [12, 19].

The PRISMA model introduces aspects as first-order citizens of software architectures. As a result, PRISMA specifies different crosscutting concerns (distribution, safety, context-awareness, coordination, etc.) of the software architecture using aspects. From the aspect-oriented point of view, PRISMA is a symmetrical model [13] that does not distinguish a kernel or core entity to encapsulate functionality; functionality is also defined as an aspect. A concern can be specified by several aspects of a software architecture, whereas a PRISMA aspect represents a concern that crosscuts the software architecture. This crosscutting is due to the fact that the same aspect can be imported by more than one architectural element (i.e. a component or a connector) of a software architecture. In this sense, aspects crosscut those elements of the architecture that import their behaviour (see Figure 1).

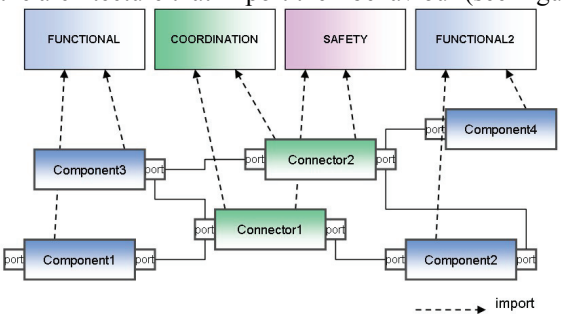

Figure 1. Crosscutting-concerns in PRISMA architectures

The PRISMA approach takes advantage of the notion of aspect from the beginning of the system definition. The change of a property only requires the change of the

aspect that defines it, and then, each architectural element that imports the changed aspect is also updated. A PRISMA architectural element can be seen from two different views: internal and external. In the external view, architectural elements encapsulate their functionality as black boxes and publish a set of services that they offer to other architectural elements (see Figure 2.A). These services are grouped into interfaces to be published through the ports of architectural elements. As a result, ports are the interaction points of architectural elements.

The internal view shows an architectural element as a prism (white box view). Each side of the prism is an aspect that the architectural element imports. In this way, architectural elements are represented as a set of aspects (see Figure 2.B) and their synchronization relationships,which are called weavings. A weaving indicates that the execution of an aspect service can trigger the execution of services in other aspects. The weaving process of an architectural element is composed of a set of weavings.

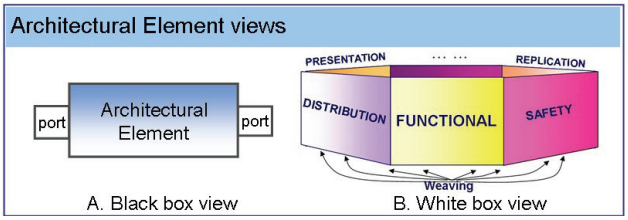

Figure 2. Views of an architectural element

In PRISMA, in order to preserve the independence of the aspect specification from other aspects and weavings, weavings are specified outside aspects and inside architectural elements. Weavings weave the different aspects that form an architectural element. As a result, aspects are reusable and independent of the context of application, which facilitates their maintenance.

3 MOF+Computational Reflection

In order to illustrate our proposal, we use a simple case study throughout the paper. This case study consists of a robotic arm whose movements are controlled by different joints: Base, Shoulder, Elbow, Wrist and Gripper. These joints are modeled as instances of a Joint *component type*. The specific initialization values of each Joint instance are provided at its instantiation. The behaviour of the Joint is defined by two aspects: (i) a functional aspect, Fun, which defines how the movements are sent to each hardware robotic joint, and (ii) a safety aspect, Saf, which checks that the Joint movements are between the maximum and minimum values that are allowed to ensure the safety of the robotic arm and its environment. Services are exported to other components through the port OperPort.

One of the most important non-functional requirements of the Joint component is that its instances are going to be executed for long periods of time in an high availability software system. Thus, if an update is needed, it will have to be applied at runtime without disturbing the system execution. For this reason, the Joint component provides an evolution infrastructure to support its runtime adaptation without forcing the system to be restarted. In our case study new requirements emerged after the

execution of the Joint component, specifically, the inclusion of an emergency service to instantly stop all the robotic movements. This requirement involves the adaptation of the safety aspect. A new *component type*, called Updater, was developed to replace the old safety aspect (Saf) by a new one (Saf2) at runtime.

The evolution infrastructure proposed in this work is based on several key concepts. First, to be able to evolve components, we must distinguish between *component instances* and *component types*, since they are placed at different abstraction levels. The OMG Meta-Object Facility (MOF) specification [20] allows us to clearly distinguish between *types* and *instances* in a proper and elegant way. MOF defines a four-level "architecture" that is focused on Model-Driven Development [19]. Its main purpose is the management of model descriptions at different levels of abstraction and their static modification. The upper layer (M3) is the most abstract one (see M3 layer, Figure 3). This layer defines the abstract language used to describe the entities of the lower layer (metamodels). The MOF specification proposes the *MOF Model* as the abstract language for defining all kind of metamodels, such as UML or PRISMA.

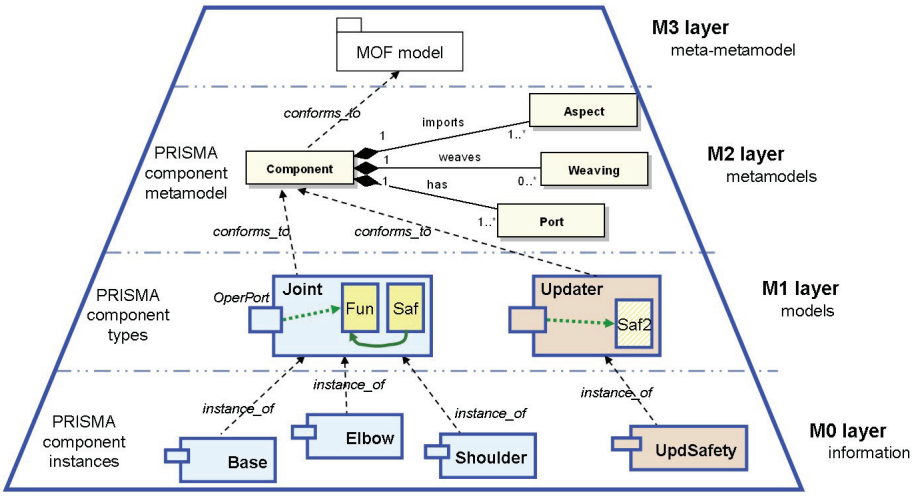

Figure 3. Meta-Object Facility (MOF) layers and PRISMA Components

The metamodel layer (M2) defines the structure and semantics of the models defined at the lower layer. The PRISMA metamodel is defined at this level: PRISMA components are an aggregation of Aspects, Ports, and Weavings (see M2 layer, Figure 3). Component behaviour is defined by importing aspects and by synchronizing them through the use of weavings. Published services are defined through the use of ports. The M1 layer comprises the models that describe data. These models are described using the primitives and relationships described in the metamodel layer (M2). PRISMA *component types* (i.e.: Joint and Updater) are placed in the M1 layer (see M1 layer, Figure 3). For instance, the Joint component imports two aspects: a Functional aspect and a Safety aspect (see Fun and Saf aspects, M1 layer, Figure 3), which are synchronized through a weaving. In addition, services from the Functional aspect are provided to other components through a port.

The semantics of the PRISMA metamodel (described at the M2 layer) defines that a *component type* (described at the M1 layer) can be instantiated. These instances (i.e. PRISMA *component instances*: Base, Elbow, UpdSafety, etc.) are placed in the lowest level, the M0 layer (see M0 layer, Figure 3) and behave as described in the *component type*.

However, the MOF specification was designed to specify and manage technology-neutral metamodels from a static point of view. MOF does not describe how to address the dynamic adaptation of its elements at run-time. For this reason, the *Computational Reflection* concept [17, 31] is used to provide dynamic adaptation to models (in this case, *component types*). *Computational Reflection* is the capability of a software system to reason about itself and act upon itself. In order to do so, a system must have a representation of itself that is editable and that is *causally connected* to itself. Thus, the changes that are made in this representation (which is managed as data) will be *reflected* on the system, and vice versa. Therefore, a system has two different views of itself: the *base-view* and the *meta-view* (see Figure 4.A). The *base-view* "executes" the system business logic behaviour and modifies a set of values that define the process state. The *meta-view* defines how the system behaves; it is a "description" of the system. This view allows the system to change its behaviour by modifying its representation. The process of obtaining an editable representation of the system (the *meta-view*) is called *reification*, and the opposite process is called *reflection* (see Figure 4.A). The main advantage of *computational reflection* is the fact that it describes system self-adaptation in a simple and natural way.

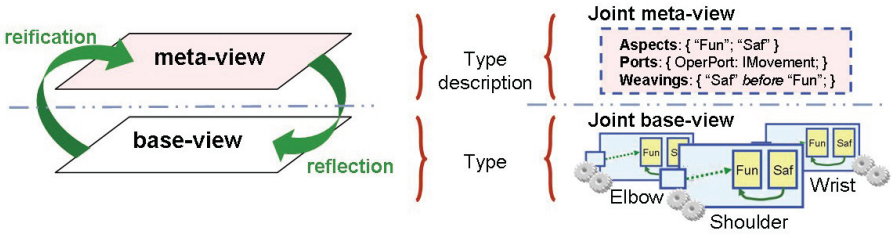

Figure 4. Dual views of a reflective system

The PRISMA metamodel describes the component structure and behaviour by means of aspects, weavings, and ports (see M2 layer, Figure 3). For this reason, the *meta-view* of a PRISMA *component type* is an editable data structure (composed of aspects, ports, and weavings) that "describes" the *component type* behaviour. For instance, the *meta-view* of the Joint *component type* describes a component that is made of: (i) the aspects Fun and Saf, (ii) a port OperPort, and (iii) a weaving between the aspects Saf and Fun (see Type description, Figure 4.B). The *base-view* of a PRISMA *component type* is the "execution" of these aspects, ports, and weavings "described" in the *meta-view*. For instance, the *base-view* of the Joint *component type* is the set of all its instances: Base, Elbow, Shoulder, etc. (see Type, Figure 4.B)

The abstraction layers provided by MOF and the capability to describe self-adaptation provided by Computational Reflection allow us to define the necessary infrastructure to dynamically adapt the internal component structure. In this work, we focus only on *component types* (M1 layer) and *component instances* (M0 layer). Each

component type has a dual view: the *base-view* and the *meta-view*. However, each view is in a different MOF layer (see Figure 5) as described below.

Each *component instance* (i.e.: Elbow, Wrist, …) is a running process which has its own state and behaves as the *component type* (i.e.: Joint) specifies. Thus, the behaviour of the instance is "provided" by the *component type base-view* (i.e. the *running* type, see *base-view*$_1$ at M0, Figure 5 below). This behaviour is "described" by the *component type meta-view* at the upper layer (see the *meta-view*$_1$ at the M1 layer, Figure 5).

Moreover, the *component type* can be viewed as a running process that also has state and behaviour: (i) the state is the *component type meta-view* (an editable representation of itself), and (ii) the behaviour is the *base-view* of the PRISMA Component (see the *base-view*$_2$ at the M1 layer, Figure 5), which is made of a set of services to manage the state of the *component type* (see the *meta-view*$_1$, Figure 5). These set of services are called evolution services, because they change the *component type* description. In PRISMA, these evolution services are: *addAspect, removeAspect*, *addPort*, etc. and are "described" in the PRISMA Component *meta-view* at the M2 layer (see *meta-view*$_2$ at M2, Figure 5).

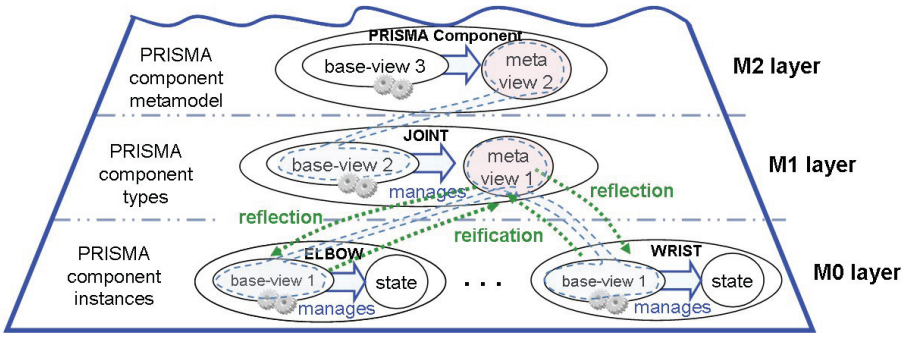

Figure 5. Dual view of reflective component types

The evolution services are provided and executed by each *component type* (in our example, the Joint component), because the PRISMA Component *base-view* is part of each *component type* (see Joint *base-view*$_2$, at the M1 layer, Figure 5). As an evolution service changes the *component type* internal representation (the *meta-view*), those changes are also *reflected* in the *component type base-view*. This means that each *component instance* would have its structure and behaviour updated according to changes made on the *component type meta-view*. For instance, as a consequence of the reflection relationship (see reflection, Figure 5), the execution of the evolution service `addAspect("Saf2")` on the Joint *component type meta-view* will trigger the addition of the aspect Saf2 on each instance of Joint (i.e.: Base, Shoulder, etc.).

The model described here allows the description of the dynamic adaptation process from a high abstraction level. However, there are some issues in the reification and reflection processes that have to be addressed in each specific implementation. The reification process must take into account how to get the type and its internal structure from a running *component instance*. The reflection process must take into account: (i) how to spread the changes made to a *component type meta-view* to its *component instances*, and (ii) how to change the internal structure of each *component instance*

without affecting those parts of the structure that have not been modified. For this reason, in the next section we describe the mechanisms that our evolution infrastructure provides to address these issues.

4 Dynamic Adaptation of Component Types

Once the concepts of *Computational Reflection* and *MOF* have been introduced, we present in detail our dynamic adaptation proposal for aspect-oriented components. The dynamic adaptation of the internal structure of components is triggered when any evolution service provided by a *component type* is invoked. Then, the evolution service modifies the *component type* description (the *component type meta-view*), and the reflection relationship performs the internal adaptation of its instances.

4.1 Evolution of component types

Heterogeneous and autonomous systems require that each one of their components implements its own adaptation mechanisms in a decentralized way. For this reason, the main objective of our proposal is to provide internal component adaptation at runtime in a decentralized and autonomous way. Decentralized adaptation is achieved because there is no a centralized evolution manager that maintains and evolves *all* the *component types* of a software architecture. Each *type* is the only entity that is responsible for its instances and is the only one capable of evolving them. Autonomous adaptation is provided in the sense that *instances* provide themselves with the infrastructure necessary to be dynamically evolved in a safe way. Each *instance* mantains its own state and is the only one capable of deciding the best moment to apply the adaptations.

4.1.1 Evolution services provided by component types

The evolution services that a *component type* provides depend on its internal implementation technology (in an imperative style, or in any declarative or formal language). However, it is possible to identify those parts of the component that are independent of technology. The main technology-independent parts of a component are: (i) behaviour and state; (ii) ports, and (iii) internal interactions between the different processes of the component. In PRISMA, behaviour and state are provided by aspect composition, ports are provided by component ports, and internal interactions are provided by weavings (they synchronize the execution of aspects). Thus, the evolution services that a *component type* should provide are those that modify the main parts of a component. We can distinguish two kinds of evolution services: Type Evolution Services and Introspection Services.

Type Evolution Services are those related to type modification, such as additions and removals of component parts. In PRISMA, some of the Type Evolution Services provided are: `AddAspect()`, `RemoveAspect()`, `AddPort()`, `RemovePort()`, `AddWeaving()`, and `RemoveWeaving()`.

Introspection Services are those evolution services that allow the structure of a component to be known. In PRISMA, some of the Introspection Services provided are: `GetAspects()`, `GetWeavings()`, and `GetPorts()`.

4.1.2 Component type reflective structure

The evolution process can be triggered by the business logic or by a user of the system. Both the business logic of the system and the user are represented in the architecture by means of components. Thus, the need for evolving a specific component emerges from another component that dynamically invokes the evolution services of the component to be updated. In our case study, the UpdSafety instance will invoke the evolution services of the Joint type in order to introduce the new safety requirements of the system.

However, in order to invoke the evolution services, there must be a link from the instance layer (M0) to the model layer (M1), that is, the link between the UpdSafety *base-view* and the Joint *meta-view* (see base and meta views in Figure 5). We call it *reification link* because (i) it is an upward link between layers and (ii) it allows instances to invoke type modification services, which are only available at an upper layer. The *reification link* should be provided by any ADL on which dynamic evolution of *component types* must be supported. There are several ways that a *reification link* could be syntactically expressed. In PRISMA, it is described by specifying the name of the *component type* that has to be evolved, followed by the dot operator "." and the evolution service to be invoked. For instance, the UpdSafety instance adds the aspect "Saf2" to the Joint component this way:

```
Joint.AddAspect("Saf2")
```

"Joint" is the type to be evolved, and "AddAspect" is the evolution service to be executed. The *reification link* is syntactically expressed by means of the "." operator. We have chosen this syntax because it is self-descriptive: *component types* provide their own adaptation services.

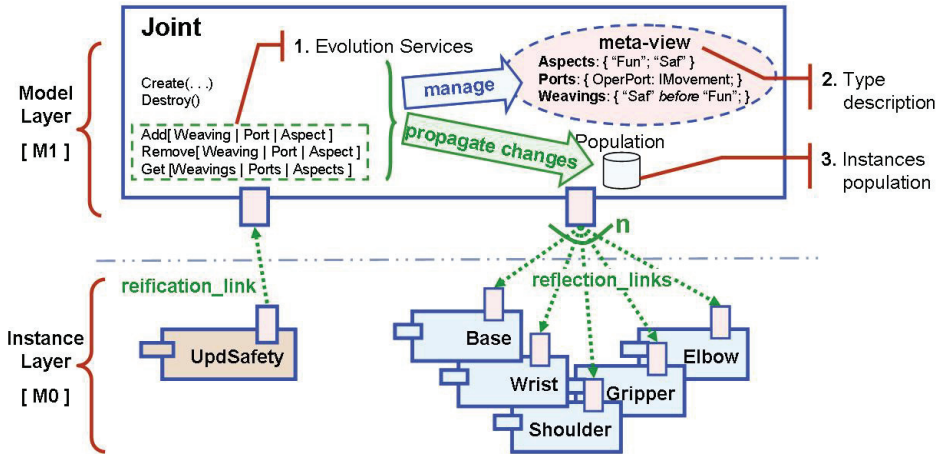

Figure 6. Reflective infrastructure for component adaptation

A *component type* is an instance factory: it is responsible for the creation and the destruction of its instances. For this reason, a *component type* manages the population of its instances, that is, a reference to each instance that is running (see note 3, Figure 6). The instance population is usually mantained by the execution platform (i.e. the garbage collector in Java or .NET). However, access to this information is necessary to be able to propagate changes later. In our case study, the population of the Joint

component is composed of the instances that have been created in the initial configuration: Base, Elbow, Shoulder, etc.

In addition, a *component type* provides its own evolution services to change itself (see note 1, Figure 6). These services modify (or get information from) the *meta-view* of the *component type* (see note 2, Figure 6). Each *component type* is responsible for keeping the reification of its *meta-view* updated while it is running. Depending on the implementation, the *meta-view* can be represented in an ADL or directly in platform-dependent code (like C#). In our prototype, the reified structure is represented in C# to make its programmed manipulation easy.

Thus, when the UpdSafety instance invokes the `AddAspect("Saf2")` evolution service (by using a reification link, see Figure 6), the Joint *meta-view* is updated. As a consequence of modifying the *meta-view*, changes are reflected, that is, they are made available to the running system. This is done in two steps. The first step is to store the changes made to the *meta-view* (i.e. the Joint representation) in order to make it persistent, by generating the resulting ADL or platform-dependent code. The immediate effect is that the aspect list `{"Fun"; "Saf"}` from the *meta-view* is updated by `{"Fun"; "Saf2"}`. Thus, new instances of Joint will be created (by calling `Joint.Create(...)`) using the updated type.

The following step is to propagate the changes to the type instances. As with reification links, there must be a link from the model layer (M1) to the instance layer (M0) in order to *reflect* the changes into each running instance. We call these links *reflection links* (see the reflection links in Figure 6). A *reflection link* is created for each *component instance* reference (the population).

4.2 Evolution of Component instances

Component types do not directly perform evolution changes inside their instances. These changes have to be internally executed by each *instance* for two reasons.

On the one hand, each instance has to decide when and how to execute its changes in order to perform the modification in a safe way. This safe modification is necessary: (i) to ensure that those parts that have not undergone changes are not affected by the modification process; (ii) to guarantee that the modification is executed once the running transactions have finished and all internal processes reach a safe state. For this reason, each component instance (i.e. Base, Elbow, Shoulder, etc.) is provided with an *Evolution Planner* aspect, whose goal is to supervise the update process of the *component instance* to which it belongs.

On the other hand, the adaptation process only makes sense at the instance-level, because state-preserving runtime adaptations are very technology-dependent operations (i.e. stopping threads, modifying memory areas, etc.). For this reason, each instance is composed of two technology-dependent aspects: the *Actuator*, which is the aspect that actually performs the changes on the internal component instance structure, and the *Sensor*, which gives information about what is going on in the *component instance*.

4.2.1 The Evolution Planner

The key aspect to achieving the dynamic adaptation of the internal structure of component instances is to maximize the independence among the internal parts of a

component which may undergo changes. In this way, replacing a component internal part has only a minimal impact on the other running parts. Those parts that are dependent to some degree on the part being changed will only need to stop temporarily while changes are being made.

An evolution dependency is defined as a binary relation over the set of internal parts (P) of a component. An internal part $x \in P$ has an evolution dependency with other internal part $y \in P$, defined as $Ev_{DEP}(x,y)$, if any change in x causes a change in y. The total amount of possible evolution dependencies over the set of internal parts P is defined by the mathematical permutation with repetition of P: $|P|^2$. Since a PRISMA component is composed of three kinds of parts (P_{PRISMA} = *{ports, weavings, aspects}*), the set of potential evolution dependencies that a PRISMA component can have is nine (i.e.: Ev_{DEP}={ (port,aspect), (port,weaving), (port,port), …}). However, due to the high degree of PRISMA reusability, there are actually two evolution dependencies between the internal parts: Ev_{DEP}(*aspect,port*) and Ev_{DEP}(*aspect*, *weaving*). These evolution dependencies are taken into account by the Evolution Planner.

The Evolution Planner (see note 3, Figure 7) is an aspect that has the knowledge about how to adapt the internal structure of a *component instance* in a safe way. For this reason, it is aware of what kind of evolution dependency relationships between the component internal parts can occur when applying a dynamic change. The action to perform is different depending on the type of change to be made: deletions, modifications or additions. In aspect deletions, related ports and weavings need also to be deleted. In aspect replacements or modifications, related ports and weavings need only to be deleted if the dependency points between the aspect and the ports and weavings are modified. In other words: (i) a port will be removed if the interface it publishes is removed from the aspect being changed, and (ii) a weaving will be removed if the method it intercepts/triggers is modified in the aspect being changed. Finally, aspects, weaving and port additions can be made without compromising other running component parts or their communications because there is still no relationship among them.

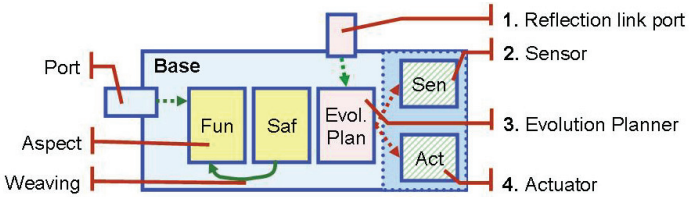

Figure 7. Internal structure of a component instance

The Evolution Planner *reflects* the changes that are made to the *component type meta-view* to the instance that the Evolution Planner belongs to. For this reason, the Evolution Planner aspect provides the same evolution services as the *component type*, although these services are only applied at instance-level. The adaptation changes to be applied are received through the *reflection links* (see note 1, Figure 7). These changes are the evolution services that the UpdSafety instance has applied on the Joint component *meta-view* and that have been propagated to the Base Evolution Planner through a *reflection link*:

```
RemoveAspect("Saf"); AddAspect("Saf2"); AddWeaving(...)
```

The Evolution Planner will apply these changes by coordinating the actions to be performed by the Actuator and Sensor aspects. This coordination is done in accordance with the correct adaptation protocols that it knows.

4.2.2 Actuator and Sensor

The *Actuator* (see note 4, Figure 7) is the aspect that performs the changes on the running instance: by generating and linking code, by creating dynamic elements, by invoking low-level adaptation mechanisms, etc. It provides additional services to prepare component elements so that changes can be safely made. These services are: `StopAspect()`, `StopPort()`, `StopWeaving()`, `StartAspect()`, `StartPort()`, and `StartWeaving()`. These services are necessary in case an evolution dependency is present, in order to avoid interactions between the part being changed and the dependent parts. The *Sensor* (see note 2, Figure 7) provides services to supervise what is going on: when an aspect has actually been added to the running component instance; when a component part (i.e.: port, aspect, weaving) has been started/stopped; etc. For these purposes, the Sensor provides additional services to get the running state of each element: `GetAspectState()`, `GetWeavingState()`, and `GetPortState()`. Both *Sensor* and *Actuator* are technology-dependent aspects because they perform tasks that rely on how component instances are implemented. The main advantage is that they allow us to abstract from platform-specific details.

The main services provided by the Actuator and the Sensor have been developed in a previous work on .NET technology [25]. A PRISMA component has been developed as a collection of different objects (aspects, ports and weavings) that: (i) can be dynamically added or removed; (ii) are highly independent on each other; (iii) interact with each other by using asynchronous mechanisms. Asynchronous mechanisms are very important because they allow internal component parts (i.e.: ports, weavings and aspects) to be stopped and restarted after changes have been made.

5 Related Works

In the last few years, there has been greater interest in evolution research in order to reduce the time and the cost of the maintenance process and to provide a solution for dynamic evolution. Thus, many approaches that provide mechanisms to support run-time adaptability have been proposed. Due to space limitations in this paper, we focus on those works related to AOSD and software architectures.

Adaptability works that are proposed by the Aspect-Oriented community usually provide mechanisms to dynamically weave aspects to running *base code*[1]. Some approaches are designed to support AOP for Object-Oriented Programming. These approaches are mainly developed in Java and .NET and are platform-dependent. SetPoint [2] allows for the dynamic addition and removal of aspects. Its weaving is based on the evaluation of logical predicates in which the base code is marked with meta-information that permits the evaluation of such predicates. Rapier-Loom.NET [29] and EOS [28] both allow for the dynamic addition and removal of aspects, but

[1] The base code is composed of the software units (modules, objects, components) of an application, which have been obtained as a result of a functional decomposition

weaving definitions are defined inside aspects, thereby losing their reusability. The work of Yang et al. [34] allows the definition of adaptation rules (i.e. weavings) separately from the code, and the dynamic addition and removal of new code (i.e. aspects) through the evaluation of such rules. However, these approaches (i) weave aspects at the instance-level instead of at the type-level, and (ii) cannot change the base code, they can only extend the base code with new behaviour.

Although there are also several software architecture works that address adaptability at runtime, they are mainly focused on dynamic reconfiguration [3, 7]. Dashofy et al. [9] describe an infrastructure to build self-healing, architecture-based software systems. Their approach consists of dynamically generating a repair plan in order to repair the system. This repair plan is executed by a global Architecture Evolution Manager (AEM). The AEM invokes the needed low-level evolution services that are provided by the runtime infrastructure. The runtime infrastructure performs the required changes in the whole system. The Rainbow Framework [4] also describes an architecture-based approach to provide the self-adaptation of running systems. The Architecture Layer is responsible for the adaptation process from the moment a change requirement is detected until the change is executed. However, these approaches use external and centralized adaptation mechanisms. These mechanisms are appropriate for small to medium-size systems. However, large systems need their adaptation to be managed in a decentralized way, i.e., each subsystem must provide its own adaptation mechanisms. In this sense, Georgiadis et al. [11] describe a decentralized infrastructure to support self-organization. However, since each component instance stores a copy of the global architecture, the infrastructure does not support scalability.

Hardly any of these approaches take into account the runtime internal adaptation of components: the old running components are replaced by the new ones, thereby losing their previous state. MARMOL [7, 8] is a formal, meta architectural model that provides ADLs with *Computational Reflection* concepts [17]. The main idea is to provide the system with an editable representation of itself. Thus, the changes made to this representation are reflected to the running system. However, this work only formalizes the required *Reflection* concepts but it does not describe the necessary infrastructure to support these concepts. The ArchWare project [18, 21] provides a prototype based on a formal language and offers a complete support for the dynamic evolution of software architectures. Runtime adaptability is performed by the Evolution Meta-Process Model [1]. Each ArchWare component is composed of a production process (which provides the component behaviour), and an evolution process. This evolution process evolves and controls the production process of the component. ArchWare supports *programmed evolution*, by providing the specification of the production process to the evolution process. It also supports *ad-hoc evolution* by using *hyper-code* abstraction. However, there is no evidence about how the ArchWare evolution process is able to evolve *component types*. All the examples are always based at the instance-level.

There are approaches that provide dynamic evolution and also combine AOP and software architectures. JAsCo [32] introduces the concept of connectors for the weaving between the aspects and the base code, which permits a high level of aspect reusability. In addition, it provides an expressive language that permits the definition of relationships among aspects. However, due to the fact that aspects are woven in a

referential way, this proposal requires an execution platform to intercept the target application and then insert the aspects at runtime. In a similar way, CAM/DAOP [27] is a component-based software architecture approach that introduces aspects as special connectors between components. It supports the separation of concerns from the design to the implementation stages of the software life cycle. However, even though it supports the dynamic weaving of aspects, it does not support the addition of new aspect types at runtime.

Kephart [14] describes the Autonomic Computing (AC) vision, where software systems are able to manage themselves, following a goal-driven approach. An autonomic element is an entity that provides functions to *monitor*, *analyze*, *plan* and *execute* control operations for a managed resource. AC is mainly focused on the IT-management of resources, performance concerns and security concerns. The main contribution of AC is that it establishes the need to build autonomous and heterogeneous software systems to address the software complexity problem.

6 Conclusions and further work

This paper has presented a novel approach to support the runtime adaptability of aspect-oriented components. This work takes the advantages of AOSD in software architecture to benefit from its reuse and maintenance, which are fundamental properties for developing complex systems. Dynamic adaptability is provided by using computational reflection concepts, since they provide a natural way to define self-modifying systems. In addition, this proposal describes the needed mechanisms to modify both *component types* and *component instances*. Thanks to the evolution infrastructure provided, running instances can trigger the modification of *component types*, so that their running instances are self-adapted dynamically according to the modifications required. Moreover, the self-adaptation process of *component instances* is possible because it only affects those parts of the instance that are undergoing the changes, thanks to the independence of PRISMA elements. The adaptation process acquires major relevance when it is applied to non-synchronized, multi-threaded components; for instance, two non-woven aspects of a PRISMA component are not aware of changes in each other.

This infrastructure provides the mechanisms needed to *plan* and *execute* adaptations. Thus, as soon as a *component type* is asked to make an adaptation, both the *component type* and its instances *plan* when the changes can be performed and then *execute* them. However, in order to provide the complete functionality of self-adaptable components, the mechanisms to *monitor* and *analyze* component adaptations should be provided by the infrastructure. These two mechanisms are future works that will be dealt with in the future. Some future works that we plan to complete in the short term are the following: (i) to define constraints for *component type* evolution; for example, it could be useful to limit the addition of new aspects or to limit the deletion of specific ports; and (ii) to ensure that a *component type* can only be dynamically evolved by authorized components (security). We are currently working on the self-adaptability capabilities of component instances in order to provide a complete framework for the self-adaptability of aspect-oriented software architectures. In addition, it is important to note that the main adaptation services

described in this paper are *additions* and *removals*. However, from a runtime perspective, *replace* operations should also be adressed.

Acknowledgements. This work is funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, META project TIN2006-15175-C05-01. This work is also supported by a FPI fellowship from Conselleria d'Educació i Ciència (Generalitat Valenciana) to C. Costa.

References

1. Balasubramaniam, D., Morrison, R., Kirby, G. et al.: A Software Architecture Approach for Structuring Autonomic Systems. In proc. of Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005). St. Louis, Missouri, USA (2005) 1-7
2. Braberman, V.: The SetPoint! project, http://setpoint.codehaus.org (2006)
3. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. In proc. of 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04). Newport Beach, California (2004) 28-33
4. Cheng, S., Garlan, D., Schmerl, B.: Making Self-Adaptation an Engineering Reality. In: O. Babaoglu, M. Jelasity, A. Montresor et al. (eds.): Self-Star Properties in Complex Information Systems. LNCS, vol. 3460. Springer, Bertinoro, Italy (2005) 158-173
5. Chitchyan, R., Rashid, A., Sawyer, P. et al.: Report Synthesizing State-of-the-Art in Aspect-Oriented Requirements Engineering, Architectures and Design. Technical Report AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9. Lancaster Univ., UK (2005)
6. Cuesta, C.E., Romay, M.d.P., Fuente, P.d.l., Barrio-Solárzano, M.: Architectural aspects of architectural aspects. In: R. Morrison, & F. Oquendo (eds.): 2nd European Workshop on Software Architecture (EWSA'05). LNCS, vol. 3527. Springer, Pisa, Italy (2005) 247-262
7. Cuesta, C.E. Dynamic Software Architecture Based on Reflection. PhD Thesis, Department of Computer Science, University of Valladolid (2002). (In Spanish).
8. Cuesta, C.E., Fuente, P.d.l., Barrio-Solárzano, M.: Dynamic Coordination Architecture through the use of Reflection. In proc. of 2001 ACM Symposium on Applied Computing. Las Vegas, Nevada, United States (2001) 134-140
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards Architecture-Based Self-Healing Systems. In proc. of First Workshop on Self-Healing Systems (WOSS'02). Charleston, South Carolina (november 18-19 2002) 21-26
10. D'Souza, D. F., & Wills, A. C.: Objects, Components, and Frameworks with UML: the Catalysis Approach. Object Technology Series edn. Addison-Wesley, USA (1998)
11. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In proc. of First Workshop on Self-Healing Systems (WOSS'02). Charleston, South Carolina (November 18-19 2002) 33-38
12. Greenfield, J., Short, K., Cook, S. et al.: Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley (2004)
13. Harrison, W. H., Ossher, H. L., Tarr, P. L.: Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. Technical Report RC22685 (W0212-147), Thomas J. Watson Research Center, IBM (2002)
14. Kephart, J. O., & Chess, D. M.: The Vision of Autonomic Computing. In: Computer, Vol. 36, No. 1. IEEE Computer (2003) 41-50
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M. et al.: An overview of AspectJ. In: 15th European Conference on Object-Oriented Programming (ECOOP'01). Lecture Notes in Computer Science, Vol. 2072. Springer, London, UK (2001) 327-353

16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C. et al.: Aspect-Oriented Programming. In: 11th European Conference on Object-Oriented Programming (ECOOP'97). Lecture Notes in Computer Science, Vol. 1241. Springer, Jyväskylä, Finland (1997) 220-242
17. Maes, P.: Concepts and Experiments in Computational Reflection. In: SIGPLAN Not., Vol. 22, No. 12. ACM Press, New York, NY, USA (1987) 147-155
18. Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K. et al.: Support for Evolving Software Architectures in the ArchWare ADL. In proc. of 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04). Oslo, Norway (june 12-15 2004) 69-78
19. Object Management Group (OMG): Model Driven Architecture Guide, http://www.omg.org/docs/omg/03-06-01.pdf (2003)
20. Object Management Group (OMG): Meta-Object Facility (MOF) 1.4 Specification. TR formal/2002-04-03. http://www.omg.org/technology/documents/formal/mof.htm (2002)
21. Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R. et al.: ArchWare: Architecting evolvable software. In: F. Oquendo, B. Warboys, R. Morrison (eds.): First European Workshop on Softw. Architecture (EWSA'04). LNCS, Vol. 3047. Springer (2004) 257-271
22. Pérez, J. PRISMA: Aspect-Oriented Software Architectures. PhD Thesis, Department of Information Systems and Computation, Polytechnic University of Valencia (2006).
23. Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In proc. of the 9th International Symposium on Component-Based Software Engineering (CBSE06). Lecture Notes in Computer Science, Vol. 4063. Springer, Västerås, Sweden (2006) 123-138
24. Pérez, J., Ali, N., Carsí, J.A, Ramos, I.: Dynamic Evolution in Aspect-Oriented Architectural Models. In: R. Morrison, & F. Oquendo (eds.): 2nd European Workshop on Software Architecture (EWSA'05). LNCS, vol. 3527. Springer, Pisa, Italy (2005) 59-76
25. Pérez, J., Ali, N., Costa, C., Carsí, J.A, Ramos, I.: Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. In proc. of 3rd International Conference on .NET Technologies. Pilsen, Czech Republic (june 2005) 97-108
26. Perry, D. E., & Wolf, A. L.: Foundations for the Study of Software Architecture. In: ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4. (1992) 40-52
27. Pinto, M., Fuentes, L., Troya, J. M.: A Dynamic Component and Aspect-Oriented Platform. In: The Computer Journal, Vol. 48, No. 4, Oxford University Press (2005) 401-420
28. Rajan, H., & Sullivan, K.: Eos: Instance-Level Aspects for Integrated System Design. In proc. of 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Helsinki, Finland (september 2003) 297-306
29. Schult, W., & Polze, A.: Speed Vs. Memory Usage-an Approach to Deal with Contrary Aspects. In proc. of 2nd Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), International Conference on Aspect-Oriented Software Development (AOSD). Boston, Massachusetts (2003)
30. Shaw, M., & Garlan, D.: Software Architecture: Perspectives On An Emerging Discipline. Prentice-Hall, NJ, USA (1996)
31. Smith, B.C. Reflections and Semantics in a Procedural Language. PhD Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (1982).
32. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. In proc. of 2nd International Conference on Aspect-Oriented Software Development (AOSD). Boston, Massachusetts (2003) 21-29
33. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1998)
34. Yang, Z., Cheng, B.H.C., Stirewalt, R.E.K. et al.: An Aspect-Oriented Approach to Dynamic Adaptation. In proc. of First Workshop on Self-Healing Systems (WOSS'02). Charleston, South Carolina (November 18-19 2002) 85-92