A Reflective Approach for Supporting the Dynamic Evolution of Component Types

Cristóbal Costa-Soria[1], David Hervás-Muñoz[1], Jennifer Pérez[2], José Ángel Carsí[1]
[1]*ISSI, Dept. of Information Systems and Computation,*
Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain
[2]*Escuela Universitaria de Informática,*
Technical University of Madrid (UPM), Ctra. Valencia km. 7, 28051 Madrid, Spain
ccosta@dsic.upv.es, dahermuo@fiv.upv.es, jenifer.perez@eui.upm.es, pcarsi@dsic.upv.es

Abstract

The increasing complexity of software systems requires a continuous revisions process in order to correct errors or to add new functionalities. However, the nature of some systems makes unfeasible their stopping to integrate changes. Dynamic evolution of types is a feature that provides support for changing completely at runtime the types that a system is composed of. Thus, a system is able to integrate new types, or to modify/remove existing ones, while it is running. In software architecture, these types are component specifications, and its instantiations, component instances. This paper presents a reflective approach for providing dynamic evolution of component types and instances in a decentralized way. Each type can be evolved separately from others, and each one of its instances evolves asynchronously, only after finishing their running transactions. The approach is reflective since it dynamically provides editable specifications of the type to evolve, and reflects changes on both types and instances while they are running.

1. Introduction

Nowadays, software systems are more and more complex. This entails that such systems must frequently undergo subsequent revisions in order to correct errors or to add new unforeseen functionalities. However, the intrinsic nature of some systems makes unfeasible their stopping to integrate changes. Examples of such systems are those that undergo critical missions and run continuously and uninterruptedly: they cannot be stopped to be evolved. It is while using (and maintaining) such kind of systems when the need for dynamic evolution emerges.

Software architecture [28][29] describes complex systems in terms of architectural elements (components and connectors) and their interrelations (attachments). There are some proposals for the description and specification of software architectures that, with the aim of providing more flexibility to the systems, provide support for dynamic evolution up to a point [3]. Dynamic evolution can be of two kinds, depending on what is changed: the architecture configuration, or the types that compose this architecture. The first kind of evolution, called dynamic reconfiguration (also called structural dynamism [11]), enables a software architecture to change its configuration (i.e. structure) at runtime, by creating or destroying architectural element instances (i.e. components and connectors) and its links dynamically. The second kind of evolution, called dynamic evolution of architectural types (also called architectural dynamism [11]), allows either a software architecture or an architectural element to change completely its type (i.e. its specification) at runtime. This kind of dynamism supports the introduction of new architectural element types and connections, the removal of existing element types, or the modification of the way that the different types interact. That is, it supports changing both the composition and behaviour of the software architecture while it is running.

Our work supports the latter degree of dynamism: the dynamic evolution of architectural types. The dynamic evolution of an architectural type (e.g. a component) does not only involve the change of its specification, but also the migration or evolution of all its running instances to the structure defined by the new specification. This paper describes how the internal structure of architectural types is evolved. The evolution support is provided to each component type in an independent way: each component type can be evolved independently from the others. Moreover, to

978-0-7695-3702-3/09 $25.00 © 2009 IEEE
DOI 10.1109/ICECCS.2009.35

reduce system disruption, each one of its instances evolves asynchronously, after the successful finalization of its running transactions. This approach is presented from a platform-independent view, by describing the different concerns of the dynamic evolution process and how they interrelate which each other. With the aim of illustrating our proposal, we describe how it has been applied for a concrete Architecture Description Language (ADL) –PRISMA [24][25]-, although it can be easily applied to any other ADL. This paper describes in detail the dynamic evolution infrastructure outlined in previous works [8]. We advance in the definition of the reification mechanisms and in the reflection process.

This paper is structured as follows. The PRISMA model where the approach has been applied is briefly introduced in section 2. In section 3, our approach to support dynamic evolution of types is presented in detail. Related works are discussed in section 4. Finally, conclusions and further works are presented.

2. PRISMA

PRISMA [24][25] is a symmetrical aspect-oriented model [15] because it does not consider functionality as a kernel entity that is different to aspects, and it does not constrain aspects to specify only non-functional requirements. In PRISMA, functionality is also specified as an aspect. As a result, PRISMA provides a homogeneous treatment for functional and non-functional requirements. Aspects have been introduced in the PRISMA model as a new concept of software architectures rather than simulating the aspect using other existing architectural terms (components, connectors, views, etc). In PRISMA, aspects are first-order citizens of software architectures and represent a specific behaviour of a *concern* (safety, coordination, 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 the behaviour of the concern that this aspect defines. As a result, an architectural element is defined by a set of aspects that describe it from different *concerns* of the architecture.

PRISMA has three kinds of architectural elements: simple (components and connectors) and composite (systems). Each architectural element encapsulates its functionality as a black box and publishes a set of services that they offer to other architectural elements. However, the internal view of these architectural elements differs among simple and composite ones.

On the one hand, the internal view of **simple** architectural elements is an invasive composition of aspects, which can be shown as a prism (see Figure 1, left). Each side of the prism is an aspect that the architectural element imports. An aspect defines the state and behaviour of a specific *concern* (e.g. functionality, coordination, distribution, persistence, etc). The state of an aspect at any given moment is determined by the value of its attributes. The behaviour of an aspect is defined by the semantics of the services the aspect provides. More details about the semantics can be found on [25]. Aspects are synchronised among them by means of weavings. A weaving indicates that the execution of an aspect service can trigger the execution of services in other aspects. Thus, the behaviour of a simple architectural element emerges from the set of aspects it is invasively composed of.

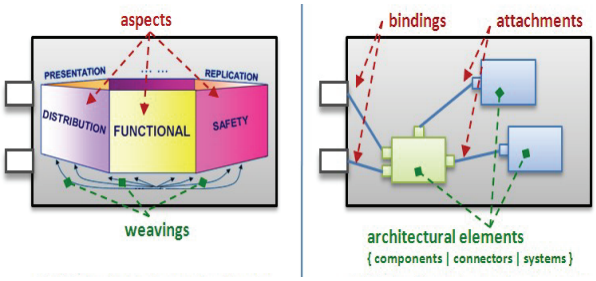

Figure 1. Internal views of PRISMA architectural elements

The difference between a component and a connector is that a component captures the functionality of software systems, whereas a connector acts as a coordinator among other architectural elements. As such, components have a functional aspect, whereas connectors have a coordination aspect.

On the other hand, the internal view of **composite** architectural elements includes a set of architectural elements (components, connectors and other systems) and the connections among them (see Figure 1, right). There are two kinds of connections: attachments and bindings. An attachment establishes a connection between a component and a connector. A binding establishes a connection between a complex component (the system) and one of the architectural elements that it contains.

3. Dynamic Evolution of Component Types

Thus, two kinds of architectural types can be evolved in PRISMA: simple and composite architectural types. The aim of this work is to provide, for each architectural type of the software system, the ability to support the dynamic change of both its specification (i.e. its type) and its running instances in an autonomous way. Thus, each architectural type will

be able to evolve independently of other types (i.e. without the need of a centralized evolution manager), in order to allow the building of heterogeneous, distributed and autonomous software systems. In order to do this, each architectural type must be provided with an infrastructure to support its evolution at runtime. In our approach, this infrastructure is integrated in each architectural type, and distributed among each one of its instances. There are two main reasons to distribute the evolution infrastructure among the instances. These reasons are related to the fact that, like the type, each instance must be capable of evolve autonomously with respect to the other instances. The first reason is that each architectural instance (i.e. component instance, connector instance or system instance) is the only one which can determine when it is ready to evolve. At runtime, each instance has a different state and a different set of running transactions from the other instances of the same architectural type. That is, the evolution instant will be different for each instance. The second reason is the support for an incremental evolution approach: each instance is decomposed in the structural parts it is composed of, and by a set of atomic evolution operations, the instance structure is changed. Since these structural parts are only accessible inside the context of the instance they belong to, its modification will only be possible if it is performed by means of evolution mechanisms provided by the instance.

The evolution infrastructure is distributed in the following way. On the one hand, an architectural type has mechanisms tailored to: (1) provide (or generate) an editable specification of itself; (2) update its internal specification with the desired changes (i.e. the code in disk), in order to allow the creation of new instances according to the new specification; and (3) supervise the migration (or evolution) process of each one of its instances. On the other hand, each instance provides a set of mechanisms to: (4) reach a quiescent state [18], in order to finish running transactions consistently; (5) modify its structure dynamically (in memory), according to the changes provided by the type, and (6) if possible, migrate the old state to the new structure introduced by the new type specification. State migration will only be possible when the replaced part provides a function to transform the state of the previous version to the new one.

The set of runtime changes allowed are those that can be applied to the specification of a PRISMA architectural element (see Figure 2). For instance, a simple architectural element can be evolved by adding, replacing, or removing aspects, ports and weavings. We have not addressed the evolution at smaller granularity levels (i.e. aspect methods and attributes) because the balance between the flexibility obtained and the performance costs introduced is negative. In our approach, aspect behaviour (i.e. methods and attributes) is modified offline. Then, the aspect is dynamically weaved to a simple architectural type. In addition, we have not covered how the evolution impacts on the interactions with the adjacent architectural elements, since this is an issue that has been already addressed by other authors. For instance, Cámara [7] addresses the adaptation of connections among architectural elements by means of the dynamic generation of adaptors that act as mediators among the existing instances and the replaced (or evolved) ones.

```
Component ImageProcessingCard
  Functional Aspect import ImgProcessingCardCtrl;
  Presentation Aspect import ImageProcessingCardGUI;
  Ports
    VideoInputPort : I_VideoServices,
          Played_Role ImgProcessingCardCtrl.VIDEOCARD;
    ImageOutputPort : I_ImageProcessingServices,
          Played_Role ImgProcessingCardCtrl.IMAGEANALYZER;
  End_Ports
  Weavings
          ImageProcessingCardGUI.showImage(image)
              after
          ImgProcessingCardCtrl.newProcessedImage(image);
  End_Weavings
  new() {
    ImgProcessingCardCtrl.begin();
  }
  destroy() {
    ImgProcessingCardCtrl.end();
  }
End_Component ImageProcessingCard;
```

Figure 2. ImageProcessingCard component specification

3.1. Type-level Evolution

In order to illustrate our approach we use the component *ImageProcessingCard* described in Figure 2. This component is weaved with two aspects: a functional aspect and a presentation aspect. The functional aspect: (i) receives images through the *VideoInputPort* component port, (ii) processes the images, and (iii) outputs the images through the *ImageOutputPort* port. The presentation aspect is synchronised (by means of a weaving) with the functional aspect to show each image that it is being processed.

Our approach is characterised by providing each architectural type (e.g. a component type) with a real presence in the software system. This presence is provided by means of an entity, M_C, that represents an architectural type, C, and which is executed together

the rest of instances of such type: C_1, C_2, C_3,... Using the example described above (the architectural type *ImageProcessingCard*), we will have $M_{ImageProcessingCard}$ This entity, M_C, can be viewed from two different viewpoints. On the one hand, M_C behaves as a class (i.e. an instance factory), since it: (i) contains a type specification, (ii) creates instances of such type, and (3) maintains the population of instances of such type. On the other hand, M_C behaves as an object, since it has a state and a set of services which change this state. The state of M_C is an editable description of the type that it represents (C), and the services M_C provides are actually evolution services: they change the editable description of the type. According to the concepts of computational reflection [19], the entity M_C is actually a meta-instance (or meta-component): it contains a reification of the type it belongs to (that is, an editable description), and this reification is causally connected to the type. All the changes performed on this reification will be reflected on the type and its respective instances.

The internal structure of such meta-instance is composed of four modules or functional areas: (1) *Builder*, responsible of the creation and destruction of instances of the architectural type; (2) *TypeDescription*, which encapsulates the reification of the architectural type and the population of instances; (3) *TypeEvolution*, which provides the evolution services; and (4) *EvolutionMonitoring*, which supervises the instance migration process from the old type specification to the new one. Meta-instances have been integrated into PRISMA by using the same concepts of the PRISMA model: a meta-instance is a simple PRISMA architectural element composed of a set of aspects. Such aspects are each one of the functional areas described above, since each functional area identifies a different concern of the evolution process and is shared among the meta-instances that provide type evolution mechanisms. By shared we refer to the fact that the specification of each aspect is common for all the meta-instances, and they only differ in the state that they acquire when they are instantiated in a particular meta-instance. The only aspect which is completely different for each meta-instance is the *Builder* aspect, since it defines the instantiation process of an architectural type (see Figure 3). The relationships among aspects are defined by means of weavings, although we are not going to describe them here due to space limitations. We describe below each one of these aspects in detail.

(1) The *Builder* aspect provides services to create and destroy instances of the type represented by the meta-instance. Its services are published through a port of the meta-instance. These services are blocked when an evolution process starts and until the evolution process finishes. This is due to the fact that the creation and destruction services can be also modified by the evolution process, and new instantiations must be made according to the new type specification.

```
void BuildComponent(string name, IComponent comp, object[] params)
{
      comp.AddAspect(new ImageProcessingCardController());
      comp.AddAspect(new ImageProcessingCardGUI());
      comp.AddWeaving("ImageProcessingCardGUI",
            "showImage", "image",   WeavingType.AFTER,
            "ImageProcessingCardController",
            "newProcessedImage","image");
      comp.AddPort("ImageOutputPort", "I_ImageProcessingServices",
            "IMAGEANALYZER");
      comp.AddPort("VideoInputPort", "I_VideoServices", "VIDEOCARD");
}
```

Figure 3. Fragment of the automatically generated Builder aspect for the ImageProcessingCard component

(2) The *TypeDescription* aspect contains the state of the meta-instance: the population of instances and the type description (i.e. the specification containing both the structure and behaviour of the type). On the one hand, population is updated whenever a new instance is created and/or destroyed, by adding or removing respectively a reference to the instance. On the other hand, the type description describes the structural parts the architectural type is composed of and their interrelations. For instance, in the case of simple PRISMA components, this data structure stores: the aspect types a component consists of, the set of weavings (i.e. relations) among these aspects, and the set of ports to provide/require services from outside the component (see Figure 4). This aspect encodes the relations among platform-independent concepts (i.e. the PRISMA metamodel) and the technology dependent concepts (i.e. the implementation of the aspect-oriented component model in .NET [27]). Thus, this aspect provides also the code-generation patterns which must be used to regenerate the type.

```
typeSpec = ComponentSpec {
    Type : ImageProcessingCard;
    ArchitecturalElementType : Component;
    Aspects :  {  ImageProcessingCardController,
                  ImageProcessingCardGUI  };
    Weavings : {
        (“ImageProcessingCardGUI”, ”showImage”, WeavingType.AFTER,
         "ImageProcessingCardController", "newProcessedImage”)  };
    Ports : { ("ImageOutputPort", "I_ImageProcessingServices",
                                                    "IMAGEANALYZER"),
            ("VideoInputPort", "I_VideoServices", "VIDEOCARD")  };
}
```

Figure 4. Specification maintained by the TypeDescription aspect of the ImageProcessingCard meta-instance

(3) The *TypeEvolution* aspect provides services to dynamically evolve the architectural type. It only provides two services to carry out the evolution process: *reify*, which returns an object <Type>Spec, and *reflect*, which requires as input parameter an object <Type>Spec. <Type>Spec is an object whose state is the editable specification of the type represented by the meta-instance. However, this specification (i.e. its state) can only be modified by means of a set of evolution services this object provides, in order to allow only consistent modifications. <Type>Spec is a generic way of naming the type of the object, since it will depend of what kind of type the meta-instance represents. This is because the editable specification and the services to change this specification that this object provides are different for each represented type. <Type> here is the meta-type of the type the meta-instance represents. For instance, in PRISMA there are three kind of architectural types (i.e. meta-types): simple (Components and Connectors), and composite (Systems). Each meta-type is specified differently and has different evolution services (see [26]). The previously introduced meta-instance M_C, which represents a simple PRISMA component type *C*, returns a *ComponentSpec* object as a result of the execution of the *Reify* service. Thus, the ComponentSpec object has an editable specification which consists of aspects, weavings and ports; it provides the evolution services defined in the PRISMA metamodel to change simple components: *addAspect, addPort, addWeaving, removeAspect, removePort, removeWeaving,* etc.

The actor of the evolution process –which can be either an actor from outside the software system (a human), or an actor from inside (another architectural element)- will evolve an existing component type this way (see Figure 5): (1) the actor obtains an editable specification of the architectural type (i.e. a <Type>Spec object) by means of the *reify* service, (2) the editable specification is modified by means of the evolution services the <Type>Spec object provides, and (3) the actor returns such object through the *reflect* service, which starts the dynamic evolution process over the component type and its instances.

The *TypeEvolution* aspect coordinates the evolution process. The *reify* service builds and returns the object <Type>Spec from the type specification that is stored in the *TypeDescription* aspect. However, the evolution process does not start until the *reflect* service is called. The evolution process consists of several stages, which are performed in a distributed way. The evolution process starts in the *TypeEvolution* aspect, is propagated to each one of the component instances, and is supervised by the *EvolutionMonitoring* aspect.

The evolution tasks performed by the *TypeEvolution* aspect are the following. First, the *Builder* aspect is blocked, in order to avoid the creation/destruction of instances while the type is being updated. This is performed by blocking the port of the meta-instance that exports the services from the *Builder* aspect. Second, the type specification contained in the *TypeDescription* aspect is updated, by using the information contained in the object <Type>Spec provided to the *reflect* service. Third, the *Builder* aspect is completely regenerated according to the type specification of the *TypeDescription* aspect. The data structures contained in the *TypeDescription* aspect are used in code generation patterns to dynamically produce new source code. This source code is dynamically compiled (by using .NET CodeDom) to create a new *Builder* aspect, which is dynamically instantiated and weaved to the meta-instance. Thus, since the new *Builder* aspect can create instances of the new type, it is unblocked and the creation of instances is allowed. Next, for each instance (whose reference is stored in the *TypeDescription* aspect), the service called *reflectToInstance* is called. This service requests each instance to start the evolution process of its structure. Finally, the control is transferred to the *EvolutionMonitoring* aspect, which will supervise the instance-level evolution process. Meanwhile, the *TypeEvolution* aspect is available for accepting new evolution requests.

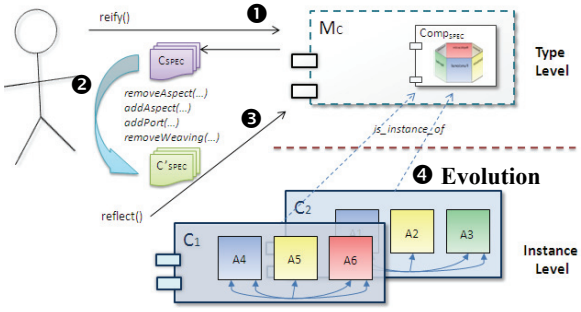

Figure 5. Black-box view of the dynamic evolution process

(4) The *EvolutionMonitoring* aspect supervises whether the instances evolve after a certain time or not. Otherwise, this aspect will take corrective measures, according to previously defined instance migration policies. We support three policies: (i) only new instances must be created according to the new type, (ii) all running instances must be evolved to the new type, and (iii) only a subset of instances *is not* evolved

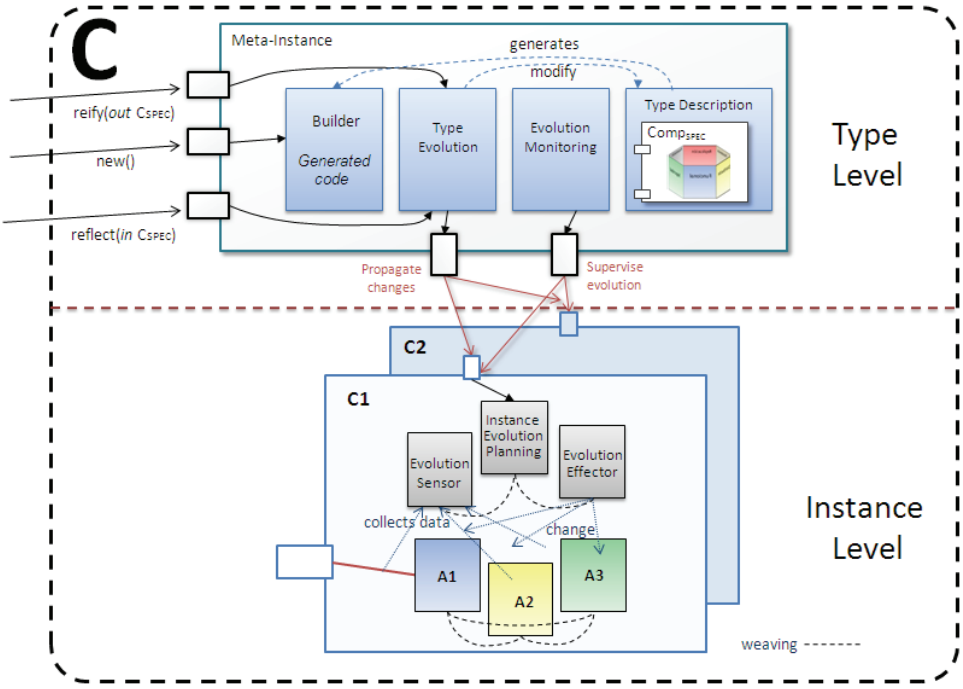

Figure 6. White-box view of the dynamic evolution process

to the new type. These policies allow specifying the available time that instances will have to evolve, and the action to perform in case an instance does not evolve in the provided timeout: (i) to force evolution and lose the current execution state, (ii) to extend the timeout *k* times, and (iii) to cancel the evolution of this instance.

Next, it is described how the evolution process is performed at the instance-level, for each one of the instances.

3.2. Instance-level Evolution

Most of the approaches that address the dynamic type evolution perform the evolution of instances by means of state migration [30]: an instance of the new type is created and then the state of the old instance is transferred to the new one. In order to do that, the new type must provide functions to transform the data structures from the old type to the data structures of the new type. However, this requires that the meta-instance (or the type) drives the entire evolution process of its instances.

This can be optimized if the specification of the type being evolved is known: the type can be decomposed in a set of smaller entities and its interrelations. Then, only the entities (or parts) that are going to be evolved (i.e. changed) are isolated, by means of the temporal stopping of its relations with other entities. For instance, PRISMA simple components are decomposed into a set of aspects, the weavings among them, and a set of ports to interact with other components. The evolution process consists in providing each instance with mechanisms to: (i) isolate its structural parts (i.e. the entities and relations that compose the type); (ii) replace these parts; and (iii) reassembly again these parts to the instance. The advantage of such instance decomposition, as opposite to instance migration, is remarkably when the types evolved are composed of concurrent entities which are highly independent among them, such as software architecture specifications (i.e. composite components) and aspect-oriented components (i.e. PRISMA simple components).

The mechanisms to evolve instances are provided by three modules or functional areas: *InstanceEvolutionPlanning*, *EvolutionSensor*, and *EvolutionEffector*. Since these modules identify different concerns of the instance evolution process, they have also been encapsulated into aspects, which are described below.

(1) The *EvolutionSensor* aspect provides services to obtain the references (i.e. memory pointers) to each one of the structural parts that the instance consists of. In addition, this aspect provides services to monitorise the status of each structural part. The status describes the execution state of structural parts, in order to know when a structural part is ready to be evolved: it has no pending transactions that can change its internal attributes, i.e. it is quiescent [18] or tranquil [31].

(2) The *EvolutionEffector* aspect provides the services that actually perform the modification of the instance structure, in terms of the instance meta-type. For instance, the provided services in simple PRISMA components are: *addAspect*, *replaceAspect*, *removeAspect*, *addPort*, *removePort, addWeaving*, etc. These services are the same that are provided by the ComponentSpec object. However, the main difference is that the *EvolutionEffector* aspect applies changes to memory structures (i.e. executing code), while the ComponentSpec object updates only a type specification (i.e. data). In addition, the *EvolutionEffector* aspect provides the services to stop and restart each one of the structural parts, that is, a set of services to drive each part to a quiescent state or to abandon it, respectively. These services in PRISMA are: *StartAspect*, *StartPort*, *StopAspect*, *StopPort*, etc.

The main challenges faced with the evolution of instances are how to manage the running processes that are concurrently executing, and how to decompose the instance structure. On the one hand, the management of running threads has been managed by the development of an executing model that allows the asynchronous execution of services. Thus, when a stop is requested, all the incoming service requests are queued and postponed. The instance will be ready to evolve when the services that are being processed finish their computations. On the other hand, the decomposition of the instance structure has been performed by means of dynamic linking strategies. The reference to each structural part is available to the evolution mechanisms. When a structural part has been stopped, it can be safely removed or replaced from memory by unlinking it from other structural parts and by linking the new part to the other structural parts. However, the *EvolutionEffector* aspect does not take into account neither the dependencies among the structural parts when applying changes, nor if they are ready to be evolved. It performs changes on the instance structure. If the instance has not been safely stopped before, then it loses its state.

(4) A planning mechanism is needed to safely stop the dependent parts and to decide when it is safe to execute the evolution actions. This is carried out by the aspect *InstanceEvolutionPlanning*, which coordinates the evolution process at instance-level. This aspect receives from the meta-instance (see Figure 6) the set of evolution changes to apply in the instance structure. These changes are provided by means of the <Type>Spec object. Internally, this object stores the set of changes performed to the type specification as a set of differences with respect to the original type specification. Thus, the type evolution process is performed as an incremental evolution process, by means of atomic operations that modify the original instance structure, either by introducing new elements or removing existing ones. Each evolution operation implies that the structural part that is going to be modified reaches previously a quiescent state (that is, it must finish first its running transactions in a consistent way). However, since the technical details of how the quiescent status is achieved are out the scope of this paper, this is not described here. The reader can refer to an abstract description in [18]. In order to carry out this process, the *InstanceEvolutionPlanning* coordinates the different services provided by both the *EvolutionSensor* and *EvolutionEffector* aspects. The services of the former aspect are used to obtain the references and the status of the structural parts to stop, while the services of the latter are used to apply the changes in an incremental way.

3.3. Dynamic Type Evolution as a Crosscutting Concern

The dynamic evolution of types is a concern that should be taken into account to the design of evolvable systems [21]. Aspect-Oriented Software Development (AOSD [17]) proposes the separation of the crosscutting concerns of software systems into separate entities called aspects. Aspects can help in separating the evolution logic from the business logic.

A concern can be represented by several aspects, like our approach does, where each aspect provides one part of the type evolution concern. Some aspects provide platform-independent functionality, whereas other aspects provide platform-dependent functionality. This avoids that changes (i.e. maintenance operations) on the platform-dependent evolution mechanisms could have an impact on the platform-independent evolution functionality, and vice versa. On the one hand, *InstanceEvolutionPlanning, TypeEvolution* and *EvolutionMonitoring* are platform-independent aspects: they describe the evolution process in terms of the metamodel used (PRISMA) and coordinate the actions to perform at a high abstraction level. On the other hand, *Builder*, *TypeDescription*, *EvolutionSensor* and *EvolutionEffector* are those aspects that bridge the platform-independent concepts (PRISMA) and the platform-dependent concepts (the implementation of the component model in a specific technology). The services provided by these latter aspects depends on the PRISMA metamodel, but its implementation is developed in the technology PRISMA architectures are executed (currently .NET [27]). Thus, this separation of aspects make easier the implementation of PRISMA in other platform: the

changes to apply in the evolution model are localised in the latter four aspects, while the rest of aspects only use concepts that are platform-independent.

On the other hand, another advantage of aspect oriented models is that one aspect can be weaved to more than one architectural element. In the case of the dynamic evolution concern, all the PRISMA architectural types that require dynamic type evolution will import this set of aspects, thus improving reuse and maintenance of the evolution code.

4. Discussion

Several works have addressed the dynamic evolution of software systems, as stated in [4] and [20]. PROSE [23] provides a low-level approach for software evolution by performing reversible changes to running Java applications. It works at the method level by replacing the old code with the new version of the code, by means of a modification of the Java Virtual Machine (JVM). Wang [32] inherits the default Java Class loader to support the dynamic evolution of (simple) Java components. It blocks the execution of new service requests and waits until the current service finishes its execution. Then, the old state is transferred to the new component by using the reflection mechanisms provided by Java. The work of Ayed [1] also uses the transference of the old state to replace the old component. It describes a policy-driven system to dynamically adapt CORBA component-based applications. It extends both the execution and deployment model of CORBA Component Model by introducing new entities and adaptation interfaces in the containers of components. This approach is similar to the Chisel framework described in [16]. Chisel is also a policy-driven context-aware system, but it has a smaller granularity: it is used to add non-functional behavior to Java classes. These works perform simple state transfers; complex ones have been well addressed by Vandewoude in [30].

These works are interesting, but all of them perform the evolution process in a centralized way: the proposed infrastructures (that is, the middlewares) extend the execution model (e.g. Java) with evolution mechanisms in order to support the evolution of all the component types, even though there are component types that do not need to evolve. For this reason, centralized approaches are not suitable, since: (i) they do not scale for large systems; (ii) the overhead introduced by evolution mechanisms is not needed by all component types (particularly those that are not evolvable), and (iii) they are only acceptable for homogeneous systems (all the elements of the system are implemented in the same technology). In our approach, type evolution is provided independently for each type of the system: a type can be provided with a meta-instance, thus providing dynamic evolution features, or not. Maintainability is provided by the aspects, since all the types that use the same technology (our approach) will import the same aspects, which are defined only once in the code. Scalability is supported because each architectural type is provided with its own evolution mechanisms.

In the area of software architecture, there some works that address runtime adaptability [3], although most of them only address dynamic reconfiguration. This is due to the fact that a lot of the authors have not established the distinction between dynamic reconfiguration and dynamic type evolution, as it is described in this work. In the literature, dynamic type evolution is used for evolving simple architectural elements (components). Dynamic reconfiguration is used for evolving the topology of a software architecture. Most of ADLs that provides support for architectural dynamism, such as PiLaR [11], Plastik [2] (based on OpenCOM, [10]) or SOFA [5], do not describe how to effectively support such dynamism, since they are focused only in the description of such dynamism. Our work follows an hybrid approach: the changes to be performed are described at a high abstraction level (in terms of the ADL chosen, PRISMA), and the different mechanisms that make possible the dynamic evolution have been identified and made available to the architecture, so that the architectural elements can interact with such mechanisms (i.e. a component can use services from the meta-instance in order to evolve another component type). The usage of reflection in our work is similar to the PiLaR ADL, where each architectural element can access its editable specification. SAFRAN [13] is an extension of the FRACTAL component model [6] which introduces adaptation aspects to decouple reconfiguration from the functional concerns. Dynamic type evolution is performed by means of component replacements.

Most of the works presented support component evolution by means of complete component replacements and state transference mechanisms. The main disadvantage is that it requires to rewire the connections from old components to the new one, so adjacent components are affected by the replacement. Since our approach performs an internal evolution, the adjacent components are unaware of the evolution process (except when evolving also public interfaces).

The works of Dashofy [12] and Garlan [14] describe the required infrastructure for describing self-adaptable system, by using models that describe the

valid architecture of the system. However, the main disadvantage is that all the evolution mechanisms are centralized, under the assumption that all the subsystems must be reconfigurable and accessible. Morrison et al. [22] describes evolvable systems as structured in two functional processes: a Producer, which provides the system behaviour (i.e. an architecture), and an Evolver, which is able to evolve this behaviour (i.e. to change the architecture). The Evolver process decides when to evolve the Producer process taking into account the feedback received both from the Producer or the environment. This approach is closely related to ours, as it provides localised change to each complex component instance and it separates specifically functionality from evolution. In contrast, we have separated the evolution concern by using aspects, in order to benefit the reuse and easy maintenance they provide.

5. Conclusions and further works

This paper has described a novel approach for supporting dynamic evolution of types, being applied to the field of software architectures, and in particular to the PRISMA approach. This approach describes an infrastructure to provide each architectural type (either simple or composite) with the ability to be evolved at runtime in an independent way, without the need of a centralized entity in charge of evolving the overall system. In this way, the types built can be integrated in heterogeneous and distributed systems. Moreover, encapsulation is preserved as well: an architectural type is a black box, and as such, its evolution can only be performed by the internal mechanisms provided by this box (the type), which are those which know the internal structure of the type and how to change such structure. From an external point of view, an evolvable type provides reflective capabilities to obtain its reification and a set of evolution services in order to modify its specification in a consistent way. From an internal point of view, the type evolution process is divided into different concerns, which are distributed among the type reification and its running instances. These concerns have been encapsulated as aspects, in order to improve reuse and maintenance. The aspects from the type level are in charge of evolving the specification of the type and the instance creation/destruction process. On the other hand, the aspects from the instance level are in charge of evolving the internal structure of each instance. Another contribution of this work is that the evolution at the instance level is performed by means of the decomposition of the internal instance structure and by means of an incremental development process, by adding/removing the entities or part that have been added/removed from the type specification.

Nowadays, these concepts are being implemented in the PRISMANET middleware [27], which supports the execution of PRISMA software architectures and its dynamic reconfiguration [9]. Once the implementation finishes, a study will be carried out in order to evaluate the response times of the evolution process, in order to compare with other approaches. Another work to perform in the near future is the definition of constraints for the evolution process of types: for instance, to limit which parts of the types can be evolved or not.

Acknowledgements. This work is funded by the Spanish Department of Science and Technology under the National Program for R+I+D META project TIN2006-15175-C05-01, by the Universidad Politécnica de Valencia under the project "Quality-Driven Model Transformations", and by the Comunidad de Madrid and the Rey Juan Carlos University under the IASOMM project URJC-CM-2007-CET-1555.

6. References

[1] D. Ayed, Y. Berbers. Dynamic Adaptation of CORBA Component-Based Applications. In proc. of *ACM Symposium on Applied Computing* (SAC'07). Seoul, Korea, March 2007.

[2] T. Batista, A. Joolia, G. Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In proc of *2nd European Workshop on Software Architectures (EWSA'05)*. LNCS, vol. 3527, pp. 1-17. Springer, 2005.

[3] J.S. Bradbury, J.R. Cordy, J. Dingel, M. Wermelinger. 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.

[4] J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniesel. Towards a taxonomy of software change. *Software Maintenance and Evolution*, 17(5). Wiley, 2005.

[5] T. Bures, P. Hnetynka, F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *4th Int. Conference on Software Engineering Research, Management and Applications (SERA'06)*, pp. 40-48. Seattle, Washington, USA, 2006.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, J.B. Stefani. An open component model and its support in java. In proc. of the *7th Int. Symposium on Component-Based Software Engineering (CBSE'04)*. Vol. 3054 of LNCS., Edinburgh, Scotland, Springer-Verlag, 2004.

[7] J. Cámara, C. Canal, J. Cubo and J.M. Murillo. An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. *Electron. Notes Theor. Comput. Sci.* 189, pp. 21-34. Elsevier, 2007.

[8] C. Costa, J. Pérez, J.A. Carsí. Dynamic Adaptation of Aspect-Oriented Components. *10th Int. ACM SIGSOFT*

Symp. on Component-Based Software Engineering (CBSE'07). LNCS, vol. 4608. Springer, 2007.

[9] C. Costa, N. Ali, J. Pérez, J.A. Carsí, I. Ramos. Dynamic Reconfiguration of Software Architectures through Aspects. In *First European Conference on Software Architecture (ECSA'07)*. LNCS, vol. 4758. Springer, 2007.

[10] G. Coulson, G.S. Blair, P. Grace et al. OpenCOM v2: A Component Model for Building Systems Software. In proc. of *IASTED Software Engineering and Applications*. Cambridge (MA), USA, 2004.

[11] C.E. Cuesta, P.d.l. Fuente, M. Barrio-Solárzano. Dynamic Coordination Architecture through the use of Reflection. In proc. *2001 ACM Symposium on Applied Computing*. Las Vegas, Nevada, United States, 2001.

[12] E.M. Dashofy, A. van der Hoek, R.N. Taylor. Towards Architecture-Based Self-Healing Systems. In proc. of *First Workshop on Self-Healing Systems (WOSS'02)*. Charleston, South Carolina, 2002.

[13] P. David, T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *5th Symp. on Software Composition (SC'06)*. Vienna, Austria, 2006.

[14] D. Garlan, S. Cheng, S. Huang, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37:46-54. IEEE, 2004.

[15] W.H. Harrison, H.L. Ossher, P.L. Tarr. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. *Technical Report RC22685 (W0212-147)*. Thomas J. Watson Research Center, IBM, 2002.

[16] J. Keeney, V. Cahill. Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework. In: *4th IEEE International Workshop on Policies for Distributed Systems and Networks*, p.3, June 04-06, 2003

[17] G. Kiczales, J. Lamping, A. Mendhekar, et al. Aspect-Oriented Programming. In *11th European Conf. on Object-Oriented Programming (ECOOP'97)*. Lecture Notes on Computer Science, Vol. 1241. Springer, 1997.

[18] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293-1306, 1990.

[19] P. Maes. Concepts and Experiments in Computational Reflection. In: *SIGPLAN Not.*, Vol. 22 (12), pp. 147-155. ACM Press, New York, NY, USA, 1987.

[20] P.K. McKinley, S.M. Sadjadi, E.P. Kasten and B.H.C Cheng. Composing Adaptive Software. *Computer*, 37(7):56-64. IEEE, 2004.

[21] T. Mens, and M. Wermelinger. Separation of concerns for software evolution. *J. of Software Maintenance and Evolution*, 14(5):311-315. Wiley, 2002.

[22] R. Morrison, D. Balasubramaniam, G. Kirby et al. A Framework for Supporting Dynamic Systems Co-Evolution. *Autom. Software. Eng*, 14(3):261-292. Springer, 2007.

[23] A. Nicoara, G. Alonso, T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *ACM SIGOPS Operating Systems Review*, Vol.42 (4). May 2008.

[24] J. Pérez, N. Ali, J.A. Carsí, I. Ramos et al. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. *Information & Software Technology*, 50(9-10):969-990. Elsevier, 2008.

[25] J. Pérez, N. Ali, J.A. Carsí, I. Ramos. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In proc. of *9th Int. Symp. on Component-Based Software Engineering (CBSE06)*. LNCS, Vol. 4063. Springer, 2006.

[26] J. Pérez, N. Ali, J.A. Carsí, I. Ramos. Dynamic Evolution in Aspect-Oriented Architectural Models. In *2nd European Workshop on Software Architecture (EWSA'05)*. LNCS, vol. 3527. Springer, 2005.

[27] J. Pérez, N. Ali, C. Costa, J.A. Carsí, I. Ramos. Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. In proc. of *3rd International Conference on .NET Technologies*, pp. 97-108. Pilsen, Czech Republic, June 2005.

[28] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. In *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, 1992.

[29] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, NJ, USA, 1996.

[30] Y. Vandewoude and Y. Berbers. Component state mapping for runtime evolution. In *Proc. of Int. Conf. on Programming Languages and Compilers*. Las Vegas, Nevada, USA, 2005.

[31] Y. Vandewoude, P. Ebraert, et al. Tranquillity: A low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering*, 33(12):856-868, 2007.

[32] Q. Wang, J. Shen, X. Wang, H. Mei. A Component-Based Approach to Online Software Evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol.18(3), pp.181-205, May 2006.