

An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures*

Cristóbal Costa-Soria¹, Jennifer Pérez², José Ángel Carsí¹

¹Dept. of Information Systems and Computation, Universidad Politécnica de Valencia, Valencia, Spain

²E.U. Informática, Technical University of Madrid (UPM), Madrid, Spain

ccosta@dsic.upv.es, jenifer.perez@eui.upm.es, pcarsi@dsic.upv.es

Abstract. The increasing complexity of current software systems is encouraging the development of self-organizing software systems, i.e. systems capable of reconfiguring their structure at runtime. These systems can be composed of autonomous composite components which also need independent reconfiguration capabilities to evolve their internal compositions. Self-organization in software architectures is described by means of reconfiguration specifications. However, these specifications are often tangled with functional specifications, thus decreasing the maintenance of such systems. This paper presents an approach for supporting autonomic reconfiguration of composite components, taking advantage of aspect-oriented techniques. This approach identifies the common concerns that are necessary to support the autonomic reconfiguration and encapsulates them into aspects, avoiding tangled code. Thus, self-reconfigurable composite components are weaved with these aspects and benefit from: (1) easy maintenance, since the reconfiguration concern is separated from the other concerns; (2) autonomous reconfiguration, since it is provided with dynamic reconfiguration services to change its architecture.

Key Words: dynamic reconfiguration, software architectures, AOSD.

1. Introduction

The increasing complexity of current software systems is becoming unmanageable: large complex systems are more and more difficult to develop and maintain. These systems require design methodologies capable of addressing the increasing complexity of the developments, but also exhibiting maintainable, flexible and scalable properties.

Software Architecture [28] provides techniques for describing the structure of complex software systems. Its aim is to hide low-level details and help to understand the system. The architecture of a software system is described in terms of architectural elements (components and connectors) and their interactions with each other. Software architectures have a hierarchical structure where components can be combined to create composite components which, in turn, can be combined into more composite components. Flexibility can be provided by allowing the software architecture to change the configuration among its architectural elements to a new one which supports the new situation. However, since there are systems that cannot be stopped to perform this configuration, this must be done at runtime. The Dynamic Reconfiguration of software architectures [22] provides support to this, by allowing the runtime creation and destruction of the architectural element instances and links at run-time [15],[22].

However, most of the approaches proposed to support dynamic reconfiguration does not have the properties of being scalable or easily maintainable. On the one hand, some approaches supports the dynamic reconfiguration from a centralized way, thus decreasing the scalability of the solution when the system grows. On the other hand, the specification of the dynamic reconfiguration behaviour is often tangled with other concerns of the system, such as the business logic (the functional concern), thus decreasing the maintainability of the code.

* This work is funded by the Spanish MICYT (META project TIN2006-15175-C05-01) and by the UPV (the project "Quality-Driven Model Transformations").

Our work is focused on the design, construction and maintenance tasks of software architectures with self-management features from a Model-Driven Development (MDD) perspective. This paper takes a step forward from a previous work [8], where it was proposed to isolate the dynamic reconfiguration concern from the rest of the system and its decomposition into reconfiguration specifications and reconfiguration mechanisms. In this paper we detail these ideas, addressing the description and design of composite components capable of reconfiguring themselves in a decentralized way, without tangling evolution and functional concerns. We have called this feature **aspect-oriented autonomic reconfiguration**, since local autonomy for dynamic reconfiguration is provided for each composite component, and separation of concerns is provided by means of Aspect-Oriented Software Development (AOSD) techniques [20]. Moreover, dynamic reconfiguration is addressed in a platform-independent way, by identifying the high-level features a reconfigurable technology should provide. These features have been inferred from platform-specific works supporting dynamic reconfiguration. Our approach has been applied to PRISMA [27], which provides a MDD framework and a platform-independent Aspect-Oriented Architecture Description Language to describe software architectures.

This paper is structured as follows. First the decisions that guide our approach are presented in section 2. Next, the PRISMA model, where we have applied our approach, is introduced in section 3. Then, the approach for supporting autonomic reconfiguration is presented in detail in section 4. Related works that address dynamic reconfiguration are discussed in section 5. Finally, conclusions and further works are presented.

2. Dynamic Reconfiguration of Software Architectures

Our work defines a design approach to build dynamically reconfigurable software architectures, a key aspect towards the development of self-managed software architectures. Dynamic reconfiguration of software architectures [22] provides support for run-time: (i) addition of new functionality (i.e. new components), (ii) replacement and/or removal of existing functionality, and (iii) modification of the connections among the architectural elements. In order to design a dynamically reconfigurable software system, there are some attributes that must be considered. The set of different attributes that characterize a dynamic software system had been studied in different works [3][4][24]. We state here the attributes we have considered the most important to include in our approach, and the reasons that guided such decisions.

The first decision to tackle with is the *granularity of changes*, that is, at what level of abstraction we are going to describe dynamic changes. On the one hand, there are several works that propose different techniques to support dynamic adaptation, as summarized in [24]. However, these works are focused on a specific technology, and for this reason, usually the reconfiguration specifications are specified at a low abstraction level. On the other hand, there are several works that propose formal Architecture Description Languages (ADLs) to describe the dynamic reconfiguration at an high abstraction level, as summarized in [3]. However, most of these works have not addressed how to support the execution of such high level reconfigurations. Since the dynamic reconfiguration of software systems is highly related with the management of running software artifacts, we should consider not only the specification of how a system should be reconfigured, but also the mechanisms that support this reconfiguration process. One of the major contributions of this paper is the definition of a model that bridges the gap between these related areas: formal dynamic ADLs, and platform-specific adaptation mechanisms.

The second decision to tackle with is the *activeness* of the software system [4]: if the dynamic reconfigurations will be reactive or proactive. On the one hand, reactive reconfigurations are dynamic changes that are driven externally by an external agent (usually the system architect or developer) and by means of a user interface. Endler [15] defined them as ad-hoc reconfigurations. On the other hand, proactive reconfigurations are dynamic changes that are driven autonomously by the system when some specific conditions or events apply. Proactive reconfigurations are usually described by means of reconfiguration specifications. A reconfiguration specification

describes *when* the architecture should change (e.g. in response to certain events or state changes) and *what* kind of changes must be performed on the architecture for each situation. Proactive reconfigurations can be described at design-time (also called programmed reconfigurations [15]) or synthesized dynamically at run-time, according to high-level goals [29]. Both kinds of dynamic changes are complementary and must be supported in order to allow a system to reconfigure itself autonomously (e.g. by using programmed reconfigurations) and to introduce unanticipated changes or updates (i.e. ad-hoc reconfigurations). Both reactive and proactive reconfigurations have one point in common: they use the same reconfiguration mechanisms, which actually carry out the run-time changes (i.e. to add/remove components safely at run-time, by preserving the system state). A way to support both kinds of reconfigurations is by explicitly modeling and taking into account the mechanisms that are going to support the dynamic reconfiguration process.

The third decision to tackle with is about the *management* of dynamic reconfigurations. Due to the growing size and complexity of software systems, scalability is also an important issue, as stated by Bradbury et al. [3], and by Kramer and Magee [21]. The management of reconfigurations can be addressed either in a centralized or in a decentralized way. Centralized approaches provide a single, global entity (the *configurator*) that contains (or generates) all the reconfiguration specifications to perform on the whole software architecture. Decentralized approaches distribute reconfiguration specifications across the architectural elements (i.e. components and/or connectors), which are capable of reconfiguring the architecture to which they belong. This is usually carried out by using specific reconfiguration primitives that are provided by the ADL. Our proposal follows a *partial decentralized approach*: each composite component of the software architecture has fully local autonomy to dynamically reconfigure itself independently of the rest of the system. This means that each composite component: (i) contains/generates the reconfiguration specifications to change its architecture, and (ii) is provided with its own reconfiguration mechanisms to evolve independently. This is another key contribution of our work.

Another decision we have taken into account is how to deal with the separation of concerns. Separation of concerns in the context of software evolution allows us to separate parts of the software that exhibit different rates of change [23]. This should be taken into account in order to appropriately separate the different concerns involved and improve the maintenance and reuse. In order to avoid the entanglement of the functional and reconfiguration concerns, and to improve its design and maintainability, aspects are used in our approach. Aspect-Oriented Software Development (AOSD [20]) proposes the separation of the crosscutting concerns of software systems into separate entities called aspects. This separation avoids the tangled concerns of software, allowing the reuse of the same aspect in different entities of the software system as well as its maintenance. Although several proposals have addressed the integration of aspects in software architectures [10], very few of them have considered the encapsulation of the reconfiguration concern into aspects [2], [14]. We consider that the separation among the functional and reconfiguration concerns is a first step to build adaptive systems easier to maintain. Thus, the reconfiguration code will be able to change the functional code without being affected.

Therefore, the key ideas a framework should provide to allow the design of autonomous dynamic reconfigurable systems are the following: (i) Support for both reactive and proactive reconfigurations, (ii) Decentralized reconfiguration specifications and local autonomy for each composite component, and (iii) Separation of the reconfiguration concerns. Our work has been applied to the PRISMA approach [27]. This provides the following benefits: (1) architectural elements are used to model functional decomposition, and aspects are used to model crosscutting-concerns (functionality, reconfiguration, etc.). Thus, components and aspects encapsulate different properties thereby avoiding tangled code; (2) the PRISMA model is completely formalised and its Aspect-Oriented Architectural Description Language (AOADL) is a formal language, so the evolution requirements of our proposal can be easily formalized and executed without ambiguity; (3) PRISMA software architectures can be automatically compiled to a technological platform by using code generation techniques, so a technology-independent approach can be defined; and (4) the PRISMA tool supports the development of aspect-oriented software architectures following the Model-Driven Development (MDD) paradigm. Next, the PRISMA model is briefly described.

3. Background: The PRISMA Model

PRISMA is an Architecture Description Language that has introduced aspects as a new concept of software architectures. They are first-order citizens of software architectures and represent a specific behaviour of a *concern* (safety, coordination, distribution, etc.) that crosscuts the software 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 through their ports.

However, the internal view of these architectural elements differs between simple and composite ones. On the one hand, the internal view of a simple architectural element is an invasive composition [1] 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. Aspects are synchronized among them by means of weavings, that indicate how the execution of an aspect service can trigger the execution of services in other aspects. On the other hand, the internal view of composite architectural elements includes a set of architectural elements (components, connectors and other systems) and the links among them (see Figure 1, right). There are two kinds of links: attachments and bindings.

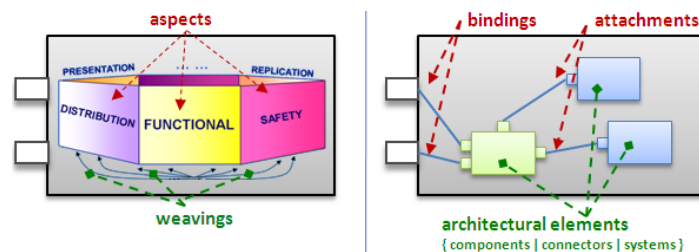


Fig. 1. Internal view of simple and complex PRISMA architectural elements

4. Autonomic Aspect-Oriented Composite Components

As also stated by other authors [3][16][17][21][26], we have observed that self-managed architectures usually follows a closed control loop that periodically monitors the architecture, plans if any (corrective) change needs to be performed, and effects them. This control loop have been taken into account in other disciplines to develop self-managed systems, being the most known the Autonomic Computing discipline [19]. Autonomic Computing (AC) proposes to achieve the self-management of systems by means of an autonomic control loop, which monitors, analyzes, plans and executes control operations on a managed resource to achieve a set of predefined high-level goals. These goals are part of the knowledge of an autonomic element.

Self-managed architectures can be also considered from an autonomic point of view, because: (1) they manage a resource (the software architecture configuration); (2) they have a set of predefined objectives (reconfiguration specifications); and (3) they have a set of control operations that constantly monitor the resource (i.e. the architecture) to detect when a change (i.e. a reconfiguration) is needed. Following the guidelines from Autonomic Computing, we have identified four concerns that dynamic reconfiguration should take into account: (i) architecture *monitoring*, the concern that captures the events that take place in the software architecture; (ii) reconfiguration *analysis*, the concern that analyzes events to detect if a reconfiguration must be done, and that provides the set of reconfiguration actions to be performed on the architecture; (iii) reconfiguration *planning*, the concern that plans or *coordinates* how the set of reconfiguration actions must be applied safely on the architecture without interrupting current transactions, and (iv) reconfiguration *effector*, the concern that applies atomic reconfiguration operations on the software architecture.

Each of the concerns we have identified to perform the reconfiguration have been encapsulated into an aspect (*Monitoring*, *ReconfigurationAnalysis*, *ReconfigurationCoordination* and *ReconfigurationEffector*). Next, each one of these aspects is described in detail below.

4.1 The Monitoring aspect

This aspect *monitors* the architecture of the composite component where it has been imported to. It provides a set of services for collecting information about: (i) the *events* that take place in the architecture, (ii) the current *configuration* of the architecture, and (iii) the *runtime status* of the different elements of the architecture (see Figure 2).

Services 1 to 3 (see Figure 2) allow to capture any event triggered in the composite component. Examples of events are: a service execution request, the creation of either an architectural element instance or a connection, etc. As it will be described in the following section, event capture is used to trigger reconfiguration processes. In addition, not only internal events can be captured, but also external events can be captured to react to changes in the environment. We use a similar approach as [13] does, so it is not described here because it has no relevance.

```

Monitoring Aspect
...
Services
(1)  afterServiceRequest(CONNid, serviceName, out params[], condition);
(2)  beforeServiceRequest(CONNid, serviceName, out params[], condition);
(3)  insteadOfServRequest(CONNid, serviceName, out params[], condition);
(4)  getArchitecturalElementInstances(out IDList [], [AEType]);
(5)  getArchitecturalElement(AEid, out ArchitecturalElement);
(6)  getConnections(out IDList [], [AEid]);
(7)  getConnection(CONNid, out AEid1, out AEid2);
(8)  getStatus(elemID, out status);
...
End_Aspect;

```

Fig. 2. The Monitoring aspect

Services 4 to 7 (see Figure 2) provide information about the current configuration of a composite component. These services provide the references to its architectural element *instances* and to their connections. Since the architecture of a dynamic reconfigurable system can change substantially over time, information about the configuration of the architecture at any given moment is essential. This way, a composite component can be aware of its configuration and use this knowledge to decide if a reconfiguration is needed. Furthermore, this information also allows to verify whether or not a set of reconfiguration actions has been successfully executed, and whether or not the target configuration has been achieved.

Service 8 (see Figure 2) provides information about the runtime status of the elements the composite component is made of: if the elements are idle, they are processing services or they are stopped. This information will allow a composite component to know whether its elements are ready to be reconfigured or not.

4.2. The ReconfigurationAnalysis aspect

This aspect describes the proactive reconfiguration behaviour (see section 2) of the composite component. It defines *when* to perform a reconfiguration process, and *how* the different architectural elements must be reconfigured. The ReconfigurationAnalysis aspect defines the policies that will drive the reconfiguration of the composite component that it belongs to. We have used the PRISMA AOADL to define simple event-condition-action (ECA) policies. These policies are expressive enough to describe how the composite component should react in presence of certain events. These policies are described at design-time. However, policies can be changed at run-time by using reflective dynamic evolution mechanisms, as described in other paper [9].

This aspect is divided into two sections: a set of configuration transactions (i.e. actions) and a set of reconfiguration triggers (i.e. events and conditions). A configuration transaction is a specification that describes a set of (re)configuration actions to be executed transactionally (all or

none) in order to achieve a new valid configuration. Thus, the configuration actions will be executed, and if anything fails, the configuration will be rolled back. A reconfiguration trigger is a condition which, if true, activates a configuration transaction. A condition may use the events captured by the Monitoring aspect (see section 4.1) to check if a service has been called in the architecture, or if a connection has been created or destroyed, etc.

For instance, consider a composite component S consisting of two components, A1 (whose type is A) and oldB (whose type is B), which are interconnected through an attachment (see 1, Figure 3). Assume that we wish to achieve the configuration 2 of Figure 3, if the previous instance (oldB) does not work properly. Figure 3-right, shows the specification of the ReconfigurationAnalysis aspect to perform this reconfiguration. On the one hand, the transaction section specifies a configuration transaction called *repairFaultyB*. First, a reference to the instance of A is obtained (1). Next, a new instance of B, newB, is created (2). Then the connection between A1 and oldB instances is removed (3), and a new one is created between A1 and newB (4). Finally, oldB is destroyed (5). On the other hand, the triggers section activates the reconfiguration transaction *repairFaultyB* when the trigger detects that an instance of B has an Unknown status (6).

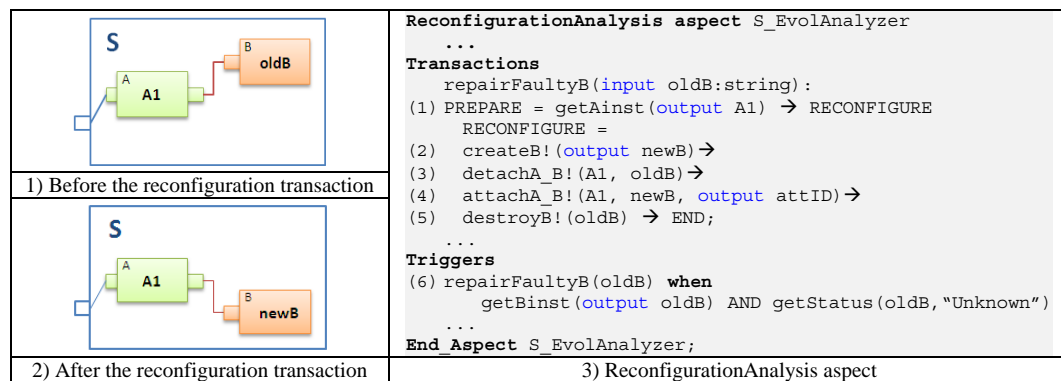


Fig. 3. A simple Reconfiguration Example

4.3. The ReconfigurationCoordination aspect

As described above, the ReconfigurationAnalysis aspect provides a set of domain-specific reconfiguration specifications. Whereas the ReconfigurationAnalysis aspect is responsible for triggering these reconfigurations when certain conditions apply, the ReconfigurationCoordination aspect is responsible for driving the successful execution of these reconfiguration specifications. It ensures that the reconfigurations triggered are performed transactionally (all or none).

When a reconfiguration transaction is triggered in the ReconfigurationAnalysis aspect, the service *beginConfigurationTransaction()*, which is provided by the ReconfigurationCoordination aspect, is implicitly executed. The execution of this service prepares the architecture of the composite component to be reconfigured. Then, the execution of each configuration action belonging to a reconfiguration transaction implicitly triggers the execution of one of the generic reconfiguration services provided by the ReconfigurationCoordination aspect (see Figure 4).

These generic reconfiguration services describe the set of low-level actions to perform for each different kind of reconfiguration action (i.e. creating instances, disconnecting instances, replacing instances, etc.). Each generic reconfiguration service performs three steps. First, the running transactions of the elements affected by a reconfiguration action are finished in a consistent way. For instance, the affected elements when performing the *destroyArchitecturalElement* operation are the instance to destroy, its connections, and its adjacent architectural element instances. Second, the set of required low-level changes are applied (e.g. destroying the required instance and its connections). These low-level changes are performed by the ReconfigurationEffector aspect (see section 4.4). Third, when the reconfiguration has been realized, it is verified whether or not the desired configuration has been achieved, by querying to the Monitoring aspect about the

configuration information (see section 4.1). Each generic reconfiguration service successfully executed is registered in a data structure, in order to undo the operation if anything fails.

Finally, if a reconfiguration transaction ends successfully, the service *endConfigurationTransaction()* is implicitly executed. Then, all the elements that were stopped are restarted. It only makes sense to start reconfigured elements when all the reconfiguration operations have been performed successfully. If any of the reconfiguration services fails, the configuration transaction is rolled back.

```
createArchitecturalElement(architecturalElementType, initializationValues,
    out architecturalElementInstanceID);
destroyArchitecturalElement(architecturalElementInstanceID);
createAttachment(archElementInstanceID1, portName1, archElementInstanceID2,
    portName2, out attachmentID);
destroyAttachment(attachmentID);
createBinding(systemPortName, archElementInstanceID, archElementPortName,
    out bindingID);
destroyBinding(bindingID);
replaceArchitecturalElement(archElementInstanceIDToBeReplaced, newArchElementType,
    [initializationValues], out newArchElementInstanceID);
```

Fig. 4. Reconfiguration services provided by the ReconfigurationCoordination aspect

4.4. The ReconfigurationEffector aspect

This aspect *effects*, or performs, the changes on the architecture it manages. It provides low-level services (i.e. those that interact with the middleware and the technological platform) to modify the architecture without taking into account the status (i.e. whether the element has been previously stopped or not) and/or the relations with the adjacent architectural elements. These low-level services must be correctly coordinated to carry out a safe reconfiguration: this is performed by the ReconfigurationCoordination aspect (see section 4.3). The most relevant low-level services provided are the following, being omitted most of the implementation details due to space reasons.

- Services to change the execution status of an element: (i) *StartElement(elemID)*, the element is activated and can process services; (ii) *StopElement(elemID)*, the element is passed to the blocked status (either tranquillity or quiescence is achieved [30]).
- Services to reconfigure the architecture: (i) *CreateInstance(compType, initParams, out compID)*, (ii) *DestroyInstance(compID)*, (iii) *Connect(compID1, port1, compID2, port2, out connID)*, (iv) *Disconnect(connID)*, and (v) *ReplaceArchitecturalElement(compID, type, [params])*.

4.5. The Evolver component: Placing all the Reconfiguration Aspects together

As described in the previous sections, the Dynamic Reconfiguration concern has been implemented by using four different aspects. The reason of this decomposition is to separate code that has different rates of change, as stated by Mens and Wermelinger [23]. The objective is to avoid that changes (i.e. maintenance operations) on the platform-specific reconfiguration mechanisms may have an impact on the platform-independent reconfiguration specifications, and vice versa. Each aspect has a different role in the MDD process. The Monitoring and ReconfigurationEffector aspects encapsulate the specific mechanisms that provide support for supervising/changing the architecture of a specific technological platform. These aspects can be easily replaced when another technological platform is selected (e.g. from .NET to Java), without affecting the high-level reconfiguration specifications. In addition, these aspects can be used in its own in different component models (i.e. not only in PRISMA). The ReconfigurationCoordination aspect encapsulates the mappings from the high-level PRISMA concepts to low-level technological services. This allows that changes on low-level mechanisms does not affect the high-level reconfiguration specifications, and viceversa. Finally, the ReconfigurationAnalysis aspect is different and specific for each composite component, and encapsulates the high-level reconfiguration specifications of the component. This aspect is specified by the software architect.

The reason to use aspects and not modules to encapsulate the dynamic reconfiguration behaviour is because of the advantages AOSD provides. Although modules can be used to separate concerns, the invocations among different modules are defined explicitly inside each module, thus making each module dependent of the other. However, aspects are, by definition, independent of each other. In addition, in symmetrical aspect-oriented models [10] like PRISMA, we cannot talk about invocations among aspects, but synchronizations among aspects. Aspects are synchronized with each other by means of weavings, which are defined outside the aspects and perform interceptions of aspect services to coordinate the execution of aspects. This way, aspects are completely independent of each other. In this way, modifying an aspect will only impact the weavings that are specifically related to this aspect. Following the example of Figure 3, some of the weavings between these aspects are shown in Figure 5.b (S_ReconfCoord and S_EvolAnalysis are instances, for the system S, of the ReconfigurationCoordination aspect and the ReconfigurationAnalysis aspect, respectively).

These aspects are provided to each PRISMA composite component intended to be dynamically reconfigurable. In the PRISMA model, aspects are imported by simple architectural elements (components and connectors); composite components do not import aspects. For this reason, the aspects related to Dynamic Reconfiguration have been imported by a simple component that is named *Evolver*[†] (see Figure 5.a).

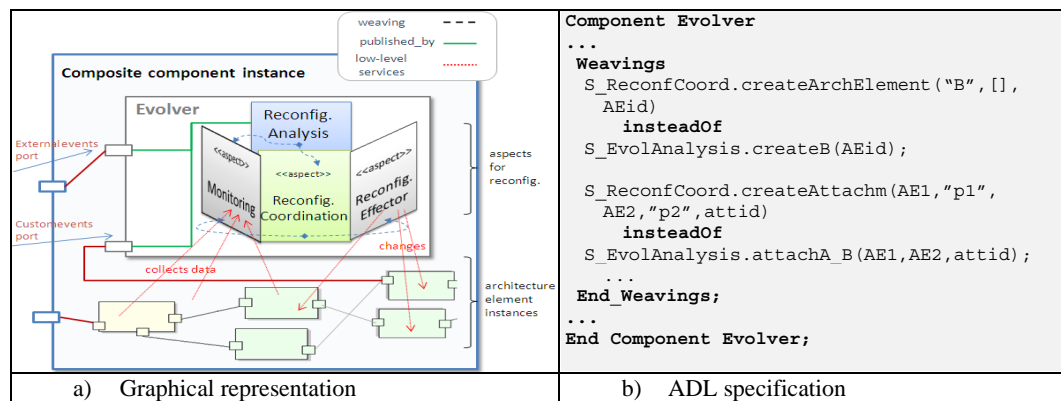


Fig. 5. The Evolver component (a) and an example of the weavings among the Reconfiguration aspects (b)

The Evolver component provides the composite component with fully autonomy to dynamically reconfigure its architecture independently of the rest of the system, eliminating the need for a centralized reconfiguration manager. Thus, a reconfigurable composite component will have a fixed part, the Evolver component, and a variable part, all the other architectural elements and connections of the composite component, that the Evolver will act upon. It is important to note that this component can only manage the architecture that it belongs to (see Figure 5.a). However, this fully autonomy does not mean that the reconfiguration process is unconstrained. The reconfiguration is limited by the constraints defined in the architecture pattern each PRISMA composite component defines [7]. Thus, although different instances of the same composite component may reconfigure its architecture, they will always maintain their conformance to their architectural pattern, so that the overall composition is preserved.

This fully autonomy is optional: only those types of composite components which may be subject to changes in the future are created (i.e. automatically generated in the context of a MDD paradigm) with reconfigurable mechanisms, in order to optimize system resources and performance. Thus, we can design software systems that are made of static architectural elements (i.e. not evolvable) as well as autonomously evolvable architectural elements.

[†] This name has been chosen because this component also imports other aspects, related to the dynamic evolution of architectural types. However, since this is outside the scope of this paper, we refer the reader to [9] for further details.

5. Related works

In the last years, a lot of research efforts have been done to address dynamic evolution of software systems [4][24], and dynamic reconfiguration of software architectures [3][11]. A few proposals have explicitly addressed the use of aspects to separate the evolution concerns in software architectures. AO-Plastik [2] isolates the reconfiguration concern by using aspectualized components and connectors to encapsulate the reconfiguration specifications. SAFRAN [14] has extended the FRACTAL component model to introduce adaptation aspects, which decouple reconfiguration from functional concerns. However, these approaches do not take into account all the concerns involved in the autonomous control loop, such as monitoring and effecting changes. Greenwood and Blair [17] proposed the use of dynamic aspects for monitoring and effecting changes. However, this work is focused on a particular technology whereas our approach is based at the architecture level in a MDD context. There are many ADLs, such as Darwin[22], LEDA[6], PiLaR [11] or SOFA [5], which provide dynamic reconfiguration support to software architectures through specific language primitives. However, these works only focus on reconfiguration specifications but do not describe how reconfiguration primitives reconfigure the architecture at runtime. In addition, its functional specifications are tangled with reconfiguration specifications. Several architecture-based approaches [12], [16] that provide self-adaptation capabilities [26] have emerged. However, these approaches use external and centralized reconfiguration mechanisms instead of using localised mechanisms to each system instance. Morrison et al. [25] have used the term *autonomic evolution* for describing those systems which can make changes to themselves based on evolutions that have previously been programmed. This approach is closely related to ours, but it is not described how the high-level mechanisms for change interact with the runtime infrastructure and if some aspects of the reconfiguration process are addressed.

6. Conclusions and further works

This paper has described an approach for supporting dynamic reconfiguration of composite components in an autonomic way. The guidelines for building autonomous systems, described by the Autonomic Computing approach, have been applied to identify the concerns involved in the dynamic reconfiguration process. These concerns have been encapsulated into aspects to avoid reconfiguration code from being tangled with system functionality. These aspects are: the Monitoring aspect, the ReconfigurationAnalysis aspect, the ReconfigurationCoordination aspect, and the ReconfigurationEffector aspect. These aspects are imported by the Evolver component, which is imported by each composite component with autonomic reconfiguration needs. Maintenance and reuse is improved as the aspects defining reconfiguration mechanisms (ReconfigurationCoordination, Monitoring, and ReconfigurationEffector) are described just once in the system and are imported by all composite components which are going to be reconfigurable.

We have used the PRISMA AOADL to define simple event-condition-action (ECA) policies, although any other kind of policies could have been defined [18]. Other promising approaches that could be encapsulated inside the ReconfigurationAnalysis aspect are those that carry out task synthesis from high-level goals [29]. Our contribution is not the definition of the reconfiguration specification, but the explicit separation between the reconfiguration specifications and the mechanisms that provide support to them. This way, the different concerns (i.e. the business logic, reconfiguration specifications, and reconfiguration mechanisms) can be maintained separately since they have different rates of change. The former is changed by the reconfiguration specifications together with the reconfiguration mechanisms. Moreover, the reconfiguration mechanisms can also be used to change the reconfiguration specifications, treating them as any other concern of the system, as we stated in [9].

References

1. U. Assmann. Invasive Software Composition. Springer, 2003.
2. T. Batista, A. Tadeu, G. Coulson, et al. On the Interplay of Aspects and Dynamic Reconfiguration in a Specification-to-Deployment Environment. In proc. of *2nd European Conference on Software Architecture (ECSA'08)*. LNCS, vol. 5292. Springer, 2008.
3. J.S. Bradbury, J.R. Cordy, J. Dingel, M. Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. *1st Workshop on Self-Managed Systems*. Newport Beach, CA, 2004.
4. J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniessel. Towards a taxonomy of software change. *Journal of 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. Conf. on Software Engineering Research, Management and Applications*. USA, 2006.
6. C. Canal, E. Pimentel, J.M. Troya. Specification and Refinement of Dynamic Software Architectures. *First Working IFIP Conference on Software Architecture (WICSA'99)*. San Antonio, Texas, USA, 1999.
7. C. Costa-Soria, J. Pérez, J.A. Carsí. Managing Dynamic Evolution of Architectural Types. In *2nd European Conf. on Software Architecture (ECSA'08)*. LNCS, vol. 5292. Springer, 2008.
8. C. Costa-Soria, J. Pérez, J.A. Carsí. Handling the Dynamic Reconfiguration of Software Architectures Using Aspects. *13th Eur. Conf. on Soft. Maintenance and Reengineering*. Kaiserslautern, Germany, 2009.
9. C. Costa-Soria, D. Hervás-Muñoz, J. Pérez, J.A. Carsí. A Reflective Approach for Supporting the Dynamic Evolution of Component Types. In proc. of *14th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'09)*. Potsdam, Germany, 2-4 June 2009.
10. C.E. Cuesta, M.d.P. Romay, P.d.l Fuente, M. Barrio-Solárzano. Architectural aspects of architectural aspects. *2nd European Workshop on Software Architecture (EWSA'05)*. LNCS, vol. 3527. Springer, 2005.
11. C.E. Cuesta, P.d.l. Fuente, M. Barrio-Solárzano. Dynamic Coordination Architecture through the use of Reflection. *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. WildCAT: a generic framework for context-aware applications. In proc. of *3rd Int. Workshop on Middleware For Pervasive and Ad-Hoc Computing (MPAC'05)*. Grenoble, France, 2005.
14. 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.
15. M. Endler & J. Wei. Programming Generic Dynamic Reconfigurations for Distributed Applications. In proc. of *First International Workshop on Configurable Distributed Systems*. London, UK, 1992.
16. D. Garlan, S. Cheng, S. Huang, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37:46-54. IEEE, 2004.
17. P. Greenwood and L. Blair. A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects. *Transactions on AOSD II*. LNCS, vol. 4242, pp. 30-65. Springer, 2006.
18. M.C. Huebscher, J.A. McCann. A survey of autonomic computing-degrees, models, and applications. *ACM Comput. Surv.* 40(3), 2008.
19. J.O. Kephart & D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41-50. IEEE, 2003.
20. G. Kiczales, J. Lamping, A. Mendhekar, et al. Aspect-Oriented Programming. In *11th ECOOP'97*.
21. J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In proc. of *ICSE - Future of Software Engineering (FOSE'07)*, pp. 259-268. IEEE, 2007.
22. J. Magee & J. Kramer. Dynamic Structure in Software Architectures. *ACM Soft. Eng. Notes*, 21(6), 1996.
23. T. Mens and M. Wermelinger. Separation of concerns for software evolution. *J. of Software Maintenance and Evolution*, 14(5):311-315. Wiley, 2002.
24. P.K. McKinley, S. Sadjadi, E. Kasten et al. Composing Adaptive Software. *Computer*, 37(7). IEEE, 2004.
25. 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.
26. P. Oreizy, M. Gorlick, R.N. Taylor et al. An Architecture-Based Approach to Self-Adaptive Software. *Intelligent Systems*, 14:54-62. IEEE, 1999.
27. 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.
28. D.E. Perry & A.L. Wolf. Foundations for the Study of Software Architecture. In *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, 1992.
29. D. Sykes, W. Heaven, J. Magee et al. From goals to components: a combined approach to self-management. *Workshop on Soft. Eng. for Adaptive and Self-Managing Systems (SEAMS)* Germany, 2008.
30. 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.