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

Cristóbal Costa-Soria
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
E-mail: cricosso@upv.es

Jennifer Pérez
E.U. Informática, Technical University of Madrid (UPM)
Carretera de Valencia km.7, E-28031 Madrid, Spain
E-mail: jenifer.perez@eui.upm.es

Jose Ángel Carsí
Dept. of Information Systems and Computation, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
E-mail: pcarsi@dsic.upv.es

Keywords: dynamic reconfiguration, AOSD, autonomic computing, software architecture

The increasing complexity of current software systems is encouraging the development of self-managed software architectures, i.e. systems capable of reconfiguring their structure at runtime to fulfil a set of goals. Several approaches have covered different aspects of their development, but some issues remain open, such as the maintainability or the scalability of self-management subsystems. Centralized approaches, like self-adaptive architectures, offer good maintenance properties but do not scale well for large systems. On the contrary, decentralized approaches, like self-organising architectures, offer good scalability but are not maintainable: reconfiguration specifications are spread and often tangled with functional specifications. In order to address these issues, this paper presents an aspect-oriented autonomic reconfiguration approach where: (1) each subsystem is provided with self-management properties so it can evolve itself and the components that it is composed of; (2) self-management concerns are isolated and encapsulated into aspects, thus improving its reuse and maintenance.

Povzetek: Predstavljen je pristop s samo-preoblikovanjem programske arhitekture.

1 Introduction

The increasing complexity of current software systems is becoming unmanageable: large complex systems are more and more difficult to develop and maintain [35]. One of the most promising techniques to deal with the design of large, complex software systems is Software Architectures [39]. Software Architectures[1] provide techniques for describing the structure of complex software systems (i.e. the key system elements and their organization). Its aim is to hide low-level details and help to understand the system. The structure of a software system is described in terms of architectural elements (components and connectors) and their interactions with each other. This structure can be formally described using an Architecture Description Language (ADL), which is used later to build the executable code of the software system. In addition, most ADLs generally support hierarchical composition (i.e. a composition hiding technique for defining systems of systems), which may be helpful for modelling large-scale complex systems in a scalable way. However, although software architecture helps in the description and development of complex systems, this is not enough: the management and maintenance of these systems still requires a great effort. To minimize such effort, self-managed software architectures were proposed [28]. According to the definition of Kramer and Magee [23], a self-managed software architecture is one in which components automatically configure their interaction in a way that: (i) is compatible with an overall architectural specification, and (ii) achieves the goals of the system. However, the development of self-managed architectures still remains a challenge [23]. Although several works have been

[1] This work has been partially supported by the Spanish Department of Science and Technology under the National Program for Research, Development and Innovation project MULTIPLE (TIN2009-13838), and by the Conselleria d'Educació i Ciència (Generalitat Valenciana) under the contract BFPI06/227.

proposed [4], they generally do not scale well for large systems or do not explicitly consider the maintainability of the self-management subsystem itself.

On designing a self-management infrastructure, also its maintenance, scalability and flexibility must be taken into account. First, maintenance should be improved by isolating dynamic change concerns from functional concerns, as it has been stated from other works [8, 10, 26]. Second, scalability can be improved by providing a decentralized self-management infrastructure. Finally, flexibility should be a fundamental property of the self-management subsystem. It should deal with not only goal-oriented proactive changes (i.e. driven autonomously), but also reactive changes (i.e. driven externally) to cope with unanticipated situations, such as the addition of new functionality.

Our work is focused on the design, construction and maintenance of systems with self-management features, from a Model-Driven Development (MDD) perspective [36]. This paper takes a step forward from a previous work [10], which 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 (i.e. a component composed of other components) capable of reconfiguring its architecture without depending on a unique centralized entity in charge of reconfiguration. In addition, the reconfiguration of composite components is managed 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 techniques [22]. Moreover, dynamic reconfiguration is platform-independent, by identifying the high-level features that a reconfigurable technology should provide. Our approach has been applied to PRISMA [32], which provides a platform-independent Aspect-Oriented ADL and is supported by a MDD framework.

This paper is structured as follows. Section 2 presents the design decisions that guided our approach. Section 3 introduces PRISMA, where this approach has been applied to. Section 4 presents a case study, which is used to illustrate the key ideas of this work. Section 5 describes our approach for supporting autonomic reconfigurations. Section 6 discusses the related works addressing dynamic reconfiguration. Finally, section 7 presents the conclusions and further works.

2 Dynamic reconfiguration of software architectures

Our work defines a design approach to build reconfigurable software architectures, a key issue in the development of self-managed software architectures. *Dynamic reconfiguration* of software architectures [16] is a term that is used to refer, generally, to those changes that are produced in the topology of a composite system at runtime, by preserving the system state and consistency. Those dynamic changes may involve: (i) addition of new functionality (i.e. new components), (ii) replacement and/or removal of existing functionality, and (iii) modification of connections between architectural elements.

A dynamically reconfigurable system is characterized by different dimensions or attributes (e.g. change type, granularity, activeness, impact, management, etc.) [4, 5]. We state here the attributes that we have considered the most important to include in our approach and the reasons that guided such decisions:

Abstraction level. Several works have addressed the support for dynamic change, although at different levels of abstraction. On the one hand, a lot of works focus on the technical feasibility of dynamic updating [25, 33, 34]. These works are generally tied to a specific technology: their reconfigurations are defined at a low abstraction level (e.g. in Java). On the other hand, other works focus on the specification of dynamic reconfigurations at a high abstraction level (i.e. by means of ADLs [4, 7, 12, 16]). However, generally 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 artefacts, we should consider not only the specification of how a system should be reconfigured, but also the mechanisms that support the execution of this reconfiguration process. One of the major contributions of this paper is the definition of a model that bridges the existing gap among high-level reconfigurations and low-level supporting mechanisms.

Activeness of changes. Dynamic reconfigurations can be reactive or proactive. On the one hand, *reactive reconfigurations* are dynamic changes that are driven by an external agent (usually the system architect or developer) and through a user interface. Endler defined them as *ad-hoc reconfigurations* [16]. An example of their utility is to perform component updates: to correct bugs or introduce new unanticipated behaviours. On the other hand, *proactive reconfigurations* are dynamic changes that are autonomously driven 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 (called *programmed reconfigurations* [16]) or synthesized at run-time, according to high-level goals [38]. Both programmed reconfigurations and high-level goals are defined by the architect. An example of their utility is to provide system dependability: if a component instance does not adequately respond, the system might change its connections to another suitable component instance or recreate the instance again. Reactive and proactive reconfigurations should be considered as complementary. Both must be supported to allow a system: to reconfigure itself autonomously (i.e. using programmed

reconfigurations), and to introduce unforeseen changes or updates at runtime (i.e. using ad-hoc reconfigurations). Since both kinds of reconfigurations rely on the same mechanisms to carry out the runtime changes, a way to support reactive and proactive reconfigurations is by explicitly modelling these mechanisms. Thus, the system architect can specify which kinds of reconfigurations are provided, by appropriately enabling or disabling reconfiguration mechanisms and providing proactive behaviour. This provides the architect with a high level of flexibility for defining reconfigurable systems. Our proposal provides support for both reactive and proactive reconfigurations.

Management of dynamic reconfigurations. Due to the growing size of software systems, the scalability of the reconfiguration subsystem is also an important issue [4, 23]. The management of reconfigurations can be addressed either in a centralized or in a decentralized way. On the one hand, centralized approaches (e.g. self-adaptive systems [14, 17, 28]) provide a single, global entity (the *Configuror*) that contains (or generates) both the reconfiguration specifications and mechanisms that will change the overall software system. The main disadvantage is a poor scalability: the larger the system, the more complex and less maintainable the configuror is, since the scope that it must supervise increases proportionally. In addition, a centralized reconfiguration manager turns into a single point of failure: if it fails, the overall system would also lose the ability to reconfigure. On the other hand, decentralized approaches (such as self-organised architectures [18, 37]) distribute reconfiguration management across the elements of the architecture, which are capable of reconfiguring the architecture to which they belong. These approaches have better scalability, since all components can perform reconfigurations. However, a disadvantage is that reconfiguration specifications are spread among different components, thus decreasing maintenance of such specifications. Another disadvantage is that system-wide properties are more difficult to control.

Our proposal follows a *hierarchical decentralized approach*. It is decentralized because each composite component of the architecture has autonomy to reconfigure its internal composition, independently of other components. It is hierarchical because each composite component reconfigures not only its composition, but also drives and coordinates the internal reconfiguration of the composite components it is composed of. That is, a composite component can reconfigure itself autonomously, but in these cases where changes could impact other components of its upper level, the reconfigurations are coordinated by its upper level self-management subsystem, to ensure the architectural consistency.

Separation of concerns. In the context of software evolution, the separation of concerns is important to separate those parts of the software that exhibit different rates of change [26]. This should be considered to appropriately avoid the entanglement of functional and reconfiguration concerns [8, 10], and improve their design and maintainability. Aspect-Oriented Software Development (AOSD) [22] 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, very few of them have considered the encapsulation of the reconfiguration concern into aspects [3, 10, 15]. 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. Our proposal takes advantage of AOSD techniques to improve the reconfiguration management.

This paper provides four contributions to the design of autonomous dynamically reconfigurable systems. First, it defines a model to bridge the gap among high-level reconfiguration specifications and low-level supporting mechanisms. Second, it considers the support for both reactive and proactive reconfigurations, to achieve a better level of flexibility. Third, it describes a hierarchical decentralized approach to tackle the problems of scalability present in self-adaptive approaches and maintainability in self-organizing ones. Fourth, it explicitly separates reconfiguration concerns to improve their maintainability and reuse. These ideas have been integrated in the PRISMA software architecture model, which is briefly introduced next.

3 Background: the PRISMA model

PRISMA provides a model and a language for the definition of complex software systems [30, 32]. Its main contributions are the way in which it integrates elements from aspect-oriented software development and software architecture approaches, as well as the advantages that this integration provides to software development.

Among the different Architecture Description Languages (ADLs) from the literature, the PRISMA ADL was selected because of the benefits it provides for supporting the dynamic evolution of software architectures. First, PRISMA allows modelling the functional decomposition of a system and its crosscutting concerns by using architectural elements and aspects, respectively. Thus, we can easily isolate functional and reconfiguration concerns. Second, PRISMA does not only allow modelling the structure (i.e. the architecture) of a system, but also allows describing precisely the internal behaviour of each architectural element. The behaviour is specified by using a modal logic of actions and a dialect of the polyadic π-calculus. π-calculus is used to specify and formalize the processes of the PRISMA model and mobility capabilities [2], and the modal logic of actions is used to formalize how the execution of these processes affects the internal state of aspects. Thus, since the internal behaviour is formally described, this allows us to automatically interleave the actions required to perform the runtime evolution of its instances. Lastly, the PRISMA ADL is supported by a

Model-Driven Development framework [36], which allows the automatic generation of executable code from PRISMA models/specifications [32]. This also benefits the support for dynamic reconfiguration: the code generation templates will only include reconfiguration mechanisms in the final code when needed. Next, the main concepts of the PRISMA ADL are introduced.

PRISMA introduces aspects as a new concept of software architectures rather than simulating them using other existing architectural terms (components, connectors, views, etc). Aspects are first-order citizens of software architectures and represent a specific behaviour of a concern (safety, coordination, distribution, *reconfiguration*, etc.) that crosscuts the software architecture. PRISMA has three kinds of architectural elements: components, connectors, and composites. 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.

The internal view of **components and connectors** (which are *simple* architectural elements) is an invasive composition [1] of aspects, which can be shown as a prism (see Figure 1). Each side of the prism is an aspect that the architectural element imports. A component differs from a connector in that it imports a functional aspect, whereas a connector imports a coordination aspect. Aspects are synchronized among them by means of weavings, which indicate how the execution of an aspect service can trigger the execution of services in other aspects (see Figure 6).

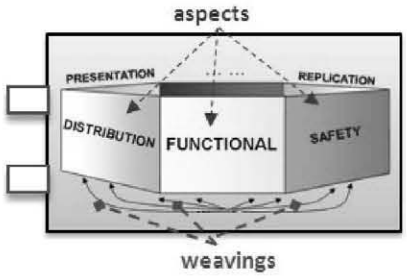

Figure 1: Internal view of *simple* PRISMA elements

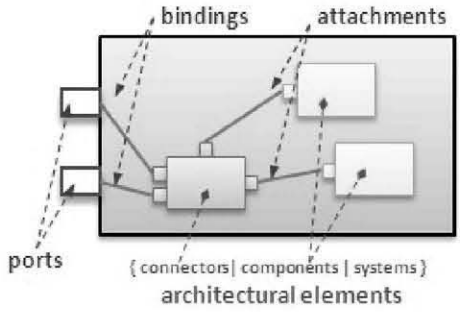

Figure 2: Internal view of *composite* elements

The internal view of **composite components** (called in the PRISMA ADL as *Systems*) consists of a set of architectural elements (components, connectors and other composites) and the links among them (see Figure 2). A link can be of two kinds: an attachment, if it links a component and a connector; or a binding, if it links an (internal) architectural element with one of the ports of the composite (i.e. allowing the communication with external architectural elements). Further details about the semantics of the PRISMA ADL can be found in [30, 32].

4 Case study: Agrobot

To illustrate our approach, we present in this section the software architecture of the *Agrobot*, an autonomous agricultural robot for plague control. Its objective is to patrol -at periodical intervals- a small field or delimited area, looking for pests or disease attacks over a set of growing crops. When a threat is detected, a pesticide is applied to, as a first counter-attack measure, and a real-time alarm is sent to the manager in order to take further specialized actions. The *Agrobot* architecture is hierarchically defined, i.e. as a system of systems. The top level, shown in Figure 3, describes the set of subsystems the robot is composed of and their interactions with each other. Each subsystem is depicted as a component, which provides and requires a set of services through its ports. Each component not only depicts the name of the instance (e.g. *LeftCamera*, see Figure 3, bottom-left), but also the name of its architectural type (e.g. *VisionSystem*), which defines the structure, behaviour and constraints of the component. The interaction among components is coordinated by different connector types (represented as blue small components), which are not detailed in this paper due to space reasons.

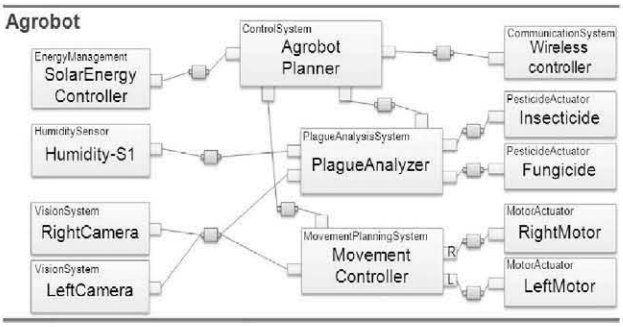

Figure 3: Software architecture of the Agrobot

Each component of the *Agrobot* architecture provides a different set of actions (e.g. image capturing, pattern analysis, movement, sensing, communication, pesticide activation, etc.), which are combined appropriately to fulfil a task (e.g. to supervise a growing crop, go to another crop, recharge energy, etc.). For instance, task planning and selection is performed by the *AgrobotPlanner* component, the energy management is performed by the *SolarEnergyController*, the communication is performed by the *WirelessController*, etc. Among the different components, we will focus on the image capturing subsystem, provided by the *RightCamera* and *LeftCamera* components. Both components are instances of the same architectural type, *VisionSystem*, but are parameterized to use a right camera

or a left camera, respectively. These components capture and pre-filter real-time images from the environment, which are used by other components to look for crop diseases (i.e. the *PlagueAnalyzer* component) or to guide the movement (i.e. the *MovementController* component). The *RightCamera* and *LeftCamera* components are composite components, i.e. their behaviour is provided by a composition of other architectural elements. They are mainly composed of a video capture component, *VideoCaptureCard*, a hardware device which captures images from the environment at a constant frame rate; and an image processing component, *ImageProcCard*, a hardware device which pre-processes the images captured. For instance, Figure 4 shows the internal structure of the *RightCamera* component: an instance of a *VideoCaptureCard* component, *Right-VCapt*, sends the captured images to an instance of an *ImageProcCard* component, *ImgProc-1*. These components are coordinated by a connector, *VCC-Conn*. The pre-processed images are sent to other subsystems by means of another connector, the *IPC-Conn* connector.

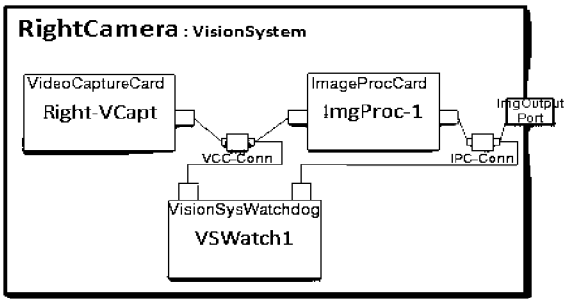

Figure 4: Architecture of the RightCamera component

Self-management is used in the *RightCamera* and *LeftCamera* components to reconfigure the internal architecture when a fault is detected in one of its components. Fault detection is performed by a watchdog component, *VisionSysWatchdog*, which periodically checks if images are being correctly captured and processed. In case misbehaviour is detected, this component sends an event to notify a failure. For instance, if the image processing component does not correctly process images or has a negative performance, then the *VisionSysWatchdog* component sends an event, which contains the name of the failing architectural type:

```
faultyOutput!(output `ImageProcCard`)
```

The events raised by the *watchdog* component will be captured by the self-management mechanisms, reacting appropriately for each kind of event. For instance, in case of an occurrence of the previously described event (i.e. *faultyOutput*), the failing component instance must be removed (i.e. the hardware image-processing device is deactivated) and another, different, component must be used instead: the *ImageProcSoftware* component. This component implements another (compatible) image processing algorithm, but with less performance than the removed one. Thus, the image capturing subsystem can continue seamlessly working.

In the next section, the self-management mechanisms that support the dynamic reconfiguration of a composite component are described in detail.

5 Autonomic reconfiguration of composite components

One of our previous works was the study and identification of the active concerns in evolution processes. As other authors also stated [4, 17, 20, 23, 28], we observed that self-managed architectures usually follow a closed control loop that periodically supervises the architecture, plans if any (corrective) change needs to be performed, and effects them. Similar control loops have been proposed to develop autonomous systems (e.g. robots), being the most extended the autonomic control loop [21], which is usually referred to as the MAPE loop (Monitor, Analyse, Plan, Execute). This loop performs control operations on a managed resource to achieve a set of predefined high-level goals, which are part of the knowledge of an autonomic (i.e. self-controlled) element. The autonomic control loop has the advantage that clearly isolates the main concerns commonly present in every process of (self-)change. Other architecture-based proposals for self-management generally merge analysis and planning, or planning and execution, or do not explicitly model the knowledge required to perform the changes.

Our proposal uses the autonomic control loop as a reference model to define how a system reconfigures itself, bridging the gap among high-level specifications (i.e. ADLs) and technology-specific (dynamic updating) mechanisms. We have adapted the original MAPE loop for this purpose: the managed resource is the architecture of a system, and the control operations performed on this resource are mainly introspection operations (for monitoring the architecture) and reconfiguration operations (for changing the architecture). Another adaptation that has been done to the original MAPE loop is its implementation by means of aspects: each one of the different controlling components (i.e. Monitor, Analyse, Plan, etc.) has been encapsulated in a different aspect. Next subsections describe the details of the approach.

5.1 Aspects for reconfiguration

Our approach defines four aspects to encapsulate the reconfiguration concerns. They are the following (see Figure 5): (i) *Monitoring*, the concern that captures the events that take place in the architecture of a composite component (i.e. the managed resource); (ii) *Reconfiguration Analysis*, the concern that analyses the different events to detect if a reconfiguration must be done, and that defines the set of reconfigurations to be performed on the architecture; (iii) *Reconfiguration Coordination*, the concern that plans/ coordinates how the reconfigurations must be applied safely to the architecture without interrupting current transactions, and (iv) *Reconfiguration Effector*, the concern that applies atomic reconfiguration operations on the running system.

Each one of these aspects will be described in the different subsections.

The reason of using aspects instead of modules for encapsulating dynamic reconfiguration behaviour is because of the advantages that AOSD provides [22], i.e. better reuse and maintenance of the different concerns. Although modules can be used to separate concerns, the invocations among different modules (e.g. procedure calls) are explicitly defined inside each module, thus making each module dependent of the other. However, in PRISMA aspects are, by definition, independent of each other: there are not invocations among aspects, but there are synchronizations among aspects. An aspect defines provided and required services, and each service is treated as a *hook* which can be intercepted. These interceptions are performed by weavings, which are defined outside the aspects and define how two aspects are bound together (i.e. synchronized). Thus, aspects are completely independent of each other: modifying an aspect will only impact the weavings that are specifically related to this aspect, but not other aspects.

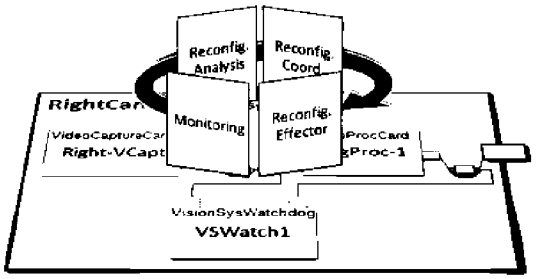

Figure 5: Aspects for autonomic reconfiguration

For instance, Figure 6 shows some of the weavings that have been defined in the VisionSystem architectural type. The first weaving intercepts the execution of the service *create-ImageProcSoftware* (provided by the Reconfiguration Analysis aspect), and replaces it with the execution of the service *createArchElement* (provided by the Reconfiguration Coordination aspect). In other words, this weaving binds the execution of a domain-specific reconfiguration service (i.e. create-ImageProcSoftware) to a generic reconfiguration service (i.e. createArchElement). This weaving will be invalidated if any of these services has its signature changed. For instance, if a parameter is removed, the weaving definition can be modified to provide a default value to the other service (or the result of applying a function). In both cases, the modification of an aspect does not necessarily impact to the other aspects. The analysis of this impact is outside the scope of this work; however other authors have conveniently addressed this problem, such as Pérez-Toledano et al. [31].

Another reason for using aspects is to avoid that changes (i.e. maintenance operations) on technology-specific reconfiguration mechanisms may have an impact on the technology-independent reconfiguration specifications, and vice versa. Each aspect has a different role in the MDD process. On the one hand, the *Reconfiguration Analysis* aspect is domain-specific: it is defined by the architect and contains the high-level specific reconfiguration policies (in terms of PRISMA concepts) for the composite component it is weaved to. On the other hand, the aspects *Monitoring* and *Reconfiguration Effector* (depicted in Figure 5 in dark grey) implement the technology-specific mechanisms that provide support for supervising/changing the architecture. They model the low-level services that are provided by the infrastructure and allow us to combine them to perform high-level reconfiguration operations. This is performed by the aspect *Reconfiguration Coordination*: it encapsulates the mappings from high-level PRISMA concepts to low-level technological services. Thus, code that has different rates of change [26] is explicitly separated: dynamic updating mechanisms (i.e. Monitoring and Effector aspects), reconfiguration specifications (i.e. Analysis aspect), and the mappings among them (i.e. Coordination aspect).

```
Weavings
  ...
  ReconfCoord.createArchElement("ImgProcSW",
       params, newID)
     insteadOf
  VisionSysRecAnalysis.create-ImageProcSoftware(
       params, newID);

  Monitoring.beforeServiceRequest(*, eventName,
       eventParams)
     insteadOf
  VisionSysRecAnalysis.beforeEvent(eventName,
       eventParams);
  ...
End_Weavings;
```

Figure 6: Example of weavings among aspects

5.1.1 The monitoring aspect

This aspect *monitorizes* 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/messages* 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. These services are shown in Figure 7. Next, they are explained in detail.

Services 1 to 3 are provided to intercept any event triggered in the composite component and act *before*, *instead of*, or *after* the event. Since this aspect supervises the architecture of a composite component, the events that it can capture are only those that take place at the level of interactions, i.e. service requests among internal components (i.e. through attachments), or coming from/to external ports (i.e. through bindings). Thus, internal components/connectors remain unaffected by interception mechanisms (i.e. encapsulation is preserved). For instance, the service 2 (see Figure 7) intercepts a service request just before it is delivered to the service provider. The parameter *serviceName* defines the request to intercept, whereas *elemID* defines which element sent the request (an external port, an architectural instance, or a connection). As it will be described in the following section, event capturing is used to trigger reconfiguration processes.

Services 4 to 8 provide information about the current configuration of the managed architecture. Since the architecture of a dynamic reconfigurable system can change substantially over time, information about the configuration at any given moment is essential. For instance, the service 4 returns a PRISMA specification with the current configuration, so it can be analysed at runtime. The other services are auxiliary and allow us to get the instances of a particular type, the connections to a given instance, etc. In this way, a composite component can be aware of its internal configuration and use this knowledge to decide if a reconfiguration is necessary. Moreover, this information also allows us to verify whether or not a set of reconfiguration actions has been successfully executed.

Finally, service 9 provides information about the runtime status of the elements the composite component is made of: if the elements are idle, processing services or stopped. This information allows a composite component to be aware of whether its elements are ready to be reconfigured or not.

```
Monitoring Aspect
  ...
  Services
(1)  afterServiceRequest(elemID, serviceName,
        output paramList);
(2)  beforeServiceRequest(elemID, serviceName,
        output paramList);
(3)  insteadOfServRequest(elemID, serviceName,
        replacingService, output paramList);
(4)  getConfigSpecification(output PRISMASpec);
(5)  getArchElementInstances(typeName,
        output instanceList);
(6)  getConnections(archElemId,
        output connectionList);
(7)  getArchElementProperties(archElemID,
        output propertiesList, output portList);
(8)  getConnectionProperties(connID,
        output archElem1, output archElem2);
(9)  getStatus(elemID, output status);
  ...
End_Aspect;
```

Figure 7: Services of the Monitoring aspect

For instance, to be subscribed to (i.e. intercept) the event *faultyOutput* when it is triggered by the *VisionSysWatchdog* component (i.e. before the event is processed), the following code must be executed:

```
beforeServiceRequest!("VisionSysWatchdog",
        "faultyOutput", output paramList)
```

5.1.2 The reconfiguration analysis aspect

This aspect describes the *proactive* reconfiguration behaviour of a composite component. This aspect is application-specific: it is defined for a specific composite component, and contains the policies that will drive the reconfiguration of this component. The *Reconfiguration Analysis* aspect defines *when* to perform a reconfiguration, and *how* the different architectural elements must be reconfigured. We have used the PRISMA AOADL to define event-condition-action (ECA) policies. These policies are expressive enough to describe how a composite component should react in presence of certain events or conditions. In our approach, these policies are described at design-time, although they can be changed at runtime by using reflective dynamic evolution mechanisms, as described in a previous work [11]. The system architect defines ECA policies by means of *configuration transactions* (i.e. Actions) and *reconfiguration triggers* (i.e. Events and Conditions). An example of this aspect is shown in Figure 8: it shows the *Reconfiguration Analysis* aspect of the *VisionSystem* architectural type.

A **configuration transaction** is a specification that describes an ordered set of domain-specific reconfiguration operations to be executed transactionally (all or none), in order to achieve a new type-conformant configuration. Thus, reconfiguration operations will be executed, and if anything fails, the reconfiguration will be rollbacked. For instance, the transaction *RepairImageProcessUnit* (see Figure 8, transactions section) describes how the component *ImageProcCard* must be replaced by the component *ImageProcSoftware* in case of malfunction. The transaction consists of two processes. The first one (see *BEGIN* process) obtains the references to the instances that are going to be affected by the reconfiguration process. Then, the second process (see *RECONF* process) performs a set of configuration actions: creates a new instance of the *ImageProcSoftware* component, attaches this instance to the instance of the *VCC-Conn* connector, detaches the failing *ImageProcCard* instance from the *VCC-Conn* connector instance, etc.

```
ReconfigurationAnalysis aspect VisionSysRecAnalysis
  ...
  Triggers
    RepairImageProcessingUnit() when
      {eventParams==["ImageProcCard"]}
      beforeEvent!("faultyOutput", out eventParams);
    ... [more reconfiguration triggers]

  Transactions
    in RepairImageProcessingUnit():
      BEGIN::=
      // Get IDs of instances subject to changes
      oldImProcCardID=imageProcCard-list[0] →
      VCCConnID=VCC-Conn-list[0] →
      IPCConnID=IPC-Conn-list[0] → RECONF;
      RECONF::=
        create-ImageProcSoftware!(cameraPos,
           output newImProcID) →
        attach-Att_VCCConn_IPCSW!(VCCConnID,
           newImProcID, output newAttID) →
        attach-Att_IPCSW_IPCConn!(newImProcID,
           IPCConnID, output newAttID) →
        detach-Att_VCCConn_IPC!(VCCConnID,
           oldImProcCardID) →
        detach-Att_IPC_IPCConn!(oldImProcCardID,
           IPCConnID) →
        destroy-ImageProcCard!(oldImProcCardID) →
        END;

    ... [more transactions]
End_Aspect VisionSysRecAnalysis;
```

Figure 8: Example of a Reconfiguration Analysis aspect

A **reconfiguration trigger** is a condition which, if true, activates a configuration transaction. This condition may evaluate user-defined attributes (e.g. performance), or be true when a certain event is intercepted (e.g. an exception, a service request, the creation or destruction

of connections, etc.). For instance, the reconfiguration trigger shown in Figure 8 (see *triggers* section) activates the configuration transaction *RepairImageProcessUnit* when a certain event is intercepted in the architecture *and* a certain condition is fulfilled. The event to intercept is the service request *faultyOutput*, and the condition is that one of the parameters of this service is "ImageProcCard". This denotes that an instance of ImageProcCard is failing (see section 4).

Note that this aspect does not directly invoke services from other aspects. For instance, the reconfiguration trigger intercepts services by means of the service *BeforeEvent*. This service is really a hook that is bound to the Monitoring aspect by means of a weaving relationship (see the second weaving in Figure 6). Without this weaving, the service *BeforeEvent* does nothing.

5.1.3 The reconfiguration coordination aspect

This aspect is in charge of driving the successful execution of the reconfiguration plans that have been triggered by the *Reconfiguration Analysis* aspect. It ensures that these plans are transactionally performed (all or none), and that the current state of the architecture is preserved. When a reconfiguration transaction is triggered, the service *beginConfigurationTransaction* 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 configuration transaction implicitly triggers the execution of one of the generic reconfiguration services provided by the *Reconfiguration Coordination* aspect. The headers of these services are shown in Figure 9. Their behaviour is defined using the PRISMA AOADL syntax. For illustration purposes only these services for creating and destroying architectural elements are completely shown.

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. For instance, the destruction of an instance and its connections. These low-level changes are performed by the Reconfiguration Effector aspect (see section 5.1.4). 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 5.1.1). 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 rollbacked.

```
ReconfigurationCoordination Aspect
  BeginConfigurationTransaction():
     ... // Initialisation of auxiliary structures
  EndConfigurationTransaction():
     CHECK::=  |transState=valid|COMMIT  +
        |transState=fail|ROLLBACK.
     COMMIT::=
        Destroy!(DestructionStack_popElement())→
     ... // [Commit and Rollback processes]
  CreateArchitecturalElement(AEType, params,
    output newID):
    CREATE::= CreateInstance!( typeof(AEType),
          params, output newID) → CHECK;
    CHECK::= CheckConsistence!(output transState) →
     |transState=fail|EndConfigurationTransaction!()
          + CONTINUE;
    CONTINUE::= ElementCreated(newID) →
          Start!(newID).
  DestroyArchitecturalElement?(id):
     STOP::= CheckConnections!(id) →
        Stop!(id) → Status!(id,status) →
        |status="Blocked"|DESTROY + STOP;
     DESTROY::= DestructionStack_pushElement(id).
  CreateAttachment(sourceArchElemID, srcPort,
     targetArchElemID, trgPort,output attID):
        [... body ommitted for space reasons ]
  DestroyAttachment(attachmentID):  [...]
  CreateBinding(sysPortName, archElemID,
     archElemPortName, out bindingID):  [...]
  DestroyBinding(bindingID):  [...]
  ReplaceArchitecturalElement(IDToBeReplaced,
     newAEType, [initializationValues], out newID):
     [...]
End_Aspect;
```

Figure 9: Services of the Reconfig. Coordination aspect

5.1.4 The reconfiguration effector aspect

This aspect *effects*, or performs, changes on the architecture it manages. It provides a set of atomic, simple reconfiguration services to interact with the other high-level aspects. These services are simple because they do not take into account the status (i.e. whether the element has been previously stopped or not) and/or the relations with the adjacent architectural elements. They must be correctly coordinated to carry out a safe reconfiguration: this is performed by the Reconfiguration Coordination aspect (see section 5.1.3). The most relevant services are shown in Figure 10.

The implementation of each reconfiguration service is technology-dependent: depending on the technology selected and how the component execution model has been implemented, the dynamic updating mechanisms to use will be different. For instance, the current implementation of the PRISMA model, PRISMANET, has been done using .NET technology and a concurrent, event-based, aspect-oriented execution model [29]. The management of connections at runtime has been done by the use of indirections and publish-subscribe mechanisms, which are implemented in ports. Among the available strategies for implementing the quiescence of

running, stateful components [19, 24, 40], finally a variation of the tranquillity approach was implemented. The support for instance replacement requires the implementation of three features: type replacement, state mapping and interface adaptation. Our current implementation only provides type replacement and state mapping, in a similar way as described by Ritzau et al. [33], but adapted for event-based, aspect-oriented components. An example of how interface adaptation can be provided is described in Cámara et al. [6].

```
ReconfigurationEffector Aspect
  ...
  Services
    StartElement(elemID);// Reach an Active status
    StopElement(elemID); // Reach a Quiescent status
    CreateInstance(componentType, initParams,
       out componentID);
    DestroyInstance(componentID);
    Connect(componentID1, port1, componentID2,
       port2, out connectionID);
    Disconnect(connectionID);
    ReplaceArchitecturalElement(ID,type,[params]);
  ...
End_Aspect;
```

Figure 10: Services of the Reconfig. Effector aspect

5.2 The evolver component: weaving the reconfiguration aspects

The previously described aspects provide autonomic reconfiguration capabilities to those composite components that import them. However, the infrastructure for supporting dynamic reconfiguration is not costless: it may introduce a performance overhead of 2% [41]. Since not all the components of a system require this degree of flexibility, and to optimize performance and system resources, the decision of which composite components will support dynamic reconfiguration or not is left to the architect. This decision is reflected by importing the reconfiguration aspects in those composite components that may undergo dynamic changes. Only when the specification of a composite component imports these aspects, the PRISMA Model Compiler [32] includes the reconfiguration mechanisms in the generated code of the composite component.

To synchronize appropriately the aspects for autonomic reconfiguration and ease their maintenance, these aspects have been encapsulated into a component called *Evolver*[2]. This component provides autonomic reconfiguration capabilities to the composite component that it has been imported to. It is integrated in the architecture of a composite component like another component, but it provides services that belong to the meta-level. That is, it offers services that introspect and change the architecture within the *Evolver* resides (i.e. a composite component).

By default, the *Evolver* only imports the aspects that support dynamic reconfigurations, i.e. *Monitoring*, *Reconfiguration Coordination* and *Reconfiguration Effector*. The activeness of change (i.e. proactive, reactive or both) is specified by the architect, depending on its needs. On the one hand, to introduce proactive reconfigurations, a Reconfiguration Analysis aspect must be defined. This is done by completing an automatically generated, empty Reconfiguration Analysis aspect with the reconfiguration policies needed. On the other hand, to allow reactive reconfigurations, two ports must be added to the Evolver: one for introspection and another for changing the architecture. The former publishes the introspection services provided by the *Monitoring* aspect (i.e. the services 4 to 8 shown in Figure 7). The latter publishes the generic reconfiguration services provided by the *Reconfiguration Coordination* aspect. These ports allow performing unanticipated reconfigurations on a composite component. These reconfigurations could be requested by another component (such as another Evolver, which would act as a *configuror* of other elements), or by the architect itself (e.g. by connecting these ports to a component that provides a user interface).

Thus, a reconfigurable composite component will have a fixed part, i.e. the Evolver, and a variable part where the Evolver will act upon, i.e. all the other components and connections of the composite component. However, this does not mean that the reconfiguration process is unconstrained. A reconfiguration is limited by the constraints defined in the type of a composite component [9]. This type defines which components can be used in the architecture and how they can be interconnected. Thus, although different instances of the same composite component reconfigure its architecture, they will always maintain type conformance, so that the overall composition is preserved.

5.3 Hierarchically decentralized evolvers

The Evolver provides a composite component with dynamic reconfiguration capabilities, which can be initiated both proactively (i.e. autonomously-driven) and reactively (i.e. externally-driven). These kinds of activeness are combined to build a *hierarchical decentralized approach* for self-management.

Each reconfigurable composite component is provided with an Evolver that proactively manages its architecture. This proactivity makes a composite component autonomous, and allows us to distribute (and *decentralize*) reconfiguration policies among the different composite components that build a system. In addition, the decentralization we propose is *hierarchical*. Since not all the reconfiguration policies are confined to a single composite component, but can span different composites, a coordination structure among different Evolvers is needed. This coordination is performed hierarchically: the Evolver of a composite component coordinates the reconfigurations of lower-level Evolvers, i.e. those that manage the reconfigurations of composite components integrated in the architecture of the upper-

[2] This name has been chosen because this component also imports other aspects, related to the dynamic evolution of architectural types. See [11] for further details.

level Evolver. For instance, the *Agrobot* system has an Evolver that manages not only the reconfiguration of the *Agrobot* architecture, but that also coordinates the reconfigurations of the composite components that compose this architecture (see Figure 11): e.g. *RightCamera, LeftCamera,* the *MovementController*, etc.

This hierarchical decentralized reconfiguration is supported by means of reconfiguration goals, reactive reconfiguration ports (i.e. introspection and reconfiguration ports), and reconfiguration events. The details of this approach are described below.

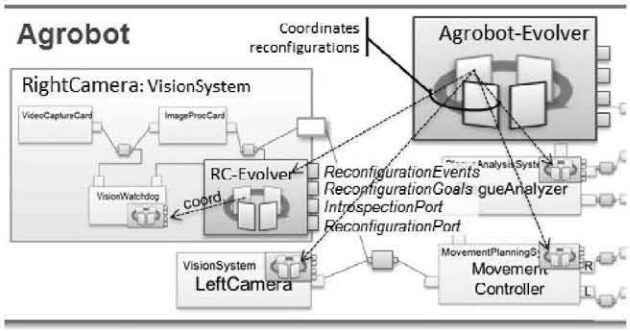

Figure 11: Agrobot and the coordination of Evolvers

5.3.1 Hierarchical change coordination

Although reconfigurable composite components may proactively reconfigure themselves, in certain cases these changes cannot be only performed locally. This is the case when reconfigurations impact several composite components simultaneously. For instance, the introduction of a new image encoding algorithm in the *Agrobot* will not only impact the image capturing subsystem (i.e. the *VisionSystem*), but also those components that decode and analyse the images captured (e.g. the *PlagueAnalyzer*). In these cases, changes must be done in a coordinated manner among the different composite components to preserve the architecture consistency. Otherwise, a *VisionSystem* component may produce images that other subsystems would be unable to decode. In our approach, this coordination of changes is performed hierarchically: the Evolver of a composite component (i.e. the upper-level Evolver) drives the reconfiguration of other composite components, through their respective Evolvers. This can be done in two ways: non-intrusively or intrusively.

Non-intrusive reconfigurations are driven by changing the **reconfiguration goals** of reconfigurable composite components. These goals are provided by the Evolver of a composite component to allow its upper-level Evolver to set reconfiguration preferences or to initiate internal proactive reconfigurations. A reconfiguration goal is an attribute defined by the architect in the *Reconfiguration Analysis* aspect that: (1) is externally visible and modifiable, and (2) is evaluated in either: (i) a *reconfiguration trigger*, to determine if a configuration transaction must be initiated; or (ii) a *configuration transaction*, to decide how a reconfiguration must be performed.

For instance, the Evolver of a *VisionSystem* composite component provides a reconfiguration goal to define the minimum performance that the *VisionSystem* must provide. Depending on the value of this goal, certain reconfigurations will be done or not. This goal is set by means of an attribute called *min_frame_rate*, which defines the minimum rate for producing images. This attribute is evaluated to decide whether a reconfiguration should be initiated to increase performance or, by the contrary, to release resources. To increase performance, the Evolver instantiates additional image processing components, whereas to release resources removes them and disables the watchdog component (thus decreasing reliability). This way, the upper-level Evolver, i.e. the *Agrobot* Evolver, can drive how the reconfiguration of the *VisionSystem* should be performed: preserving performance or reliability.

The advantage of using reconfiguration goals is that they allow us to drive the reconfiguration of a composite component without breaking its encapsulation, i.e. without directly accessing its internal composition. However, the disadvantage is that only anticipated reconfigurations can be done (i.e. those defined by reconfiguration goals). Unanticipated changes, such as the addition of a new component to a composite component, must be done intrusively. This is done through **reactive reconfiguration ports**. These ports are provided by the Evolver of a composite component to allow externally-driven reconfigurations (see section 5.2). In this context, these ports are used to allow an upper-level Evolver to explicitly introspect and change the internal composition of a composite component. Moreover, since reconfiguration services are internally provided by the *Reconfiguration Coordination* aspect, transactional reactive reconfiguration support is also provided: even all the changes externally requested are successfully executed, or all the changes are undone.

This way, an upper-level Evolver can reconfigure in a coordinated way the internal composition of different reconfigurable composite components. A coordinated change can be also transactionally performed. Each reconfiguration transaction initiated in a reactive port is considered as a subtransaction of the coordinated change transaction. If a subtransaction fails (i.e. a set of reconfigurations cannot be performed inside a composite component), then the coordinated change transaction can be entirely aborted, by deferring the commits of each subtransaction until the end of the coordinated change process.

Note that both non-intrusive and intrusive reconfigurations can be performed in a composite component if and only if its Evolver has enabled them (i.e. by exporting reconfiguration goals or reactive reconfiguration ports, respectively). This way, the architect of a reconfigurable composite component has a great level of flexibility to determine whether a composite component can be managed from outside or not, and how it can be managed.

5.3.2 Bottom-up change notifications

Another functionality that is provided by an Evolver component is the notification of changes to its upper-level Evolver. This is needed when the Evolver of a composite component has initiated changes that may impact the upper-level, i.e. the architecture where the composite component is located. For instance, consider the removal of an internal component whose functionality was being exported to other elements (e.g. the removal of the *ImageProcCard* component in the *VisionSystem*, due to a failure). These changes must be notified to its upper-level Evolver, so it can initiate additional actions to preserve architecture consistency: disabling the *VisionSystem* instance that has reduced its functionality. These changes are notified by means of *reconfiguration events*.

A **reconfiguration event** is used to communicate internal changes to outside, and has the following signature: `ReconfigurationEvent(type, message)`. The *message* parameter gives a descriptive code about the reconfiguration performed. The *type* parameter describes the impact of change, i.e. what kind of change is going to be performed: (i) *local*, an internal change: the existing interfaces remain unchanged; (ii) *medium*, a conservative change: new interfaces are added, or existing interfaces are extended with new services (i.e. existing interactions are still valid, but additional functionality is provided); and (iii) *system-wide*, a potentially disruptive change: existing interfaces are deleted, or some services removed.

Reconfiguration events can be triggered by the Reconfiguration Coordination or the Reconfiguration Analysis aspects. The Reconfiguration Coordination aspect triggers a reconfiguration event automatically when an external port or a binding to an internal component are added, changed, or removed. The reason is that external ports and binding are the means by which a composite component interacts with its environment. If an internal change impacts a port or a binding, this change will also impact the environment, so it must be notified. The Reconfiguration Analysis aspect may also trigger reconfiguration events to notify about a situation or reconfiguration performed. This is specified by the architect in proactive specifications. For instance, in a *VisionSystem* composite component, the *VideoCaptureCard* component is a critical element. If this element fails, and since the *VisionSystem* cannot perform its functionality, then the environment (i.e. the Agrobot architecture) must be notified about. This is specified in the Reconfiguration Analysis: when the event `faultyOutput?(`VideoCaptureCard`)` is intercepted, then the following event is triggered: `ReconfigurationEvent!( "system-wide", "VIDEOCARD FAILURE")`. This event will be captured by the upper-level Evolver, which will disable the composite component that has triggered this event to avoid processing its results. Thus, although one *VisionSystem* composite component failed, the robot would be able to continue working, because it is provided with two replicas of this component.

6 Related work

In the last years, a lot of research efforts have been done to address the dynamic evolution of software systems [5, 25, 34] and the reconfiguration of software architectures [4,16,19,24]. Some works have addressed the integration of AOSD techniques in software architectures [13, 32], although most of them have been mainly focused on modelling the separation of concerns at the architectural level. Only a few proposals have explicitly addressed the use of aspects to separate the evolution concerns in software architectures. AO-Plastik [3] isolates the reconfiguration concern by using aspectualized components and connectors to encapsulate the reconfiguration specifications. SAFRAN [15] 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 [20] 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 that provide dynamic reconfiguration support through specific language primitives, such as Gerel [16], Darwin [24], LEDA [7] or PiLaR [12]. These primitives are used in component specifications to describe when and how the architecture should be reconfigured. However, these works only focus on reconfiguration specifications but do not address how these specifications are finally applied on the architecture. In addition, their functional specifications are tangled with reconfiguration specifications. Several architecture-based approaches that provide self-adaptation capabilities have emerged [28]. Dashofy et al. [14] and the Rainbow framework [17] describe an architecture-based approach to provide self-healing and self-adaptation of running systems, respectively. However, both approaches use external and centralized reconfiguration mechanisms instead of using localised mechanisms to each composite component.

Morrison et al. [27] describe a conceptual framework where evolvable systems are structured in Evolver-Producer pairs (E-P). A *Producer* is a process that carries out productive functionality. An *Evolver* is a process that monitors the Producer and/or environmental stimulus, and uses this information to generate a new version of the Producer or even the *locus* (i.e. the context) where the E-P pair is located. These concepts are recursively applied to build composite systems: both an Evolver and a Producer may be internally composed of an E-P pair. Our approach shares several ideas with this conceptual framework: (i) a composite component is the locus where an E-P pair is located; (ii) the architectural elements composing a composite component represent a Producer process; and (iii) the Evolver component of a composite component behaves as an Evolver process (i.e. it can change the entire locus or generate a new version of the Producer). Another similarity with our work is that each

locus is provided with localised reconfiguration capabilities, explicitly isolating functionality from evolution. However, the framework is only conceptual, the high-level mechanisms for change are not described, and coordination issues among evolvers are not addressed.

7 Conclusion and future work

This paper has described an approach for supporting the autonomic reconfiguration of hierarchical software architectures. Instead of using a centralized self-management infrastructure to supervise the entire system and its subsystems, a hierarchical decentralized approach is proposed. Each subsystem (i.e. a composite component): (i) manages its internal reconfiguration independently of other subsystems, and (ii) provides reconfiguration events and goals to its upper level (i.e. the architecture within which it is used), to allow its integration and management. The upper level then: (i) uses these events to be informed about changes which may affect other elements, and (ii) according to the new situation, it reconfigures its architecture and/or changes the reconfiguration goals of components to fit the new needs. This approach can be recursively applied, because the same set of aspects is used at each level (i.e. *Monitoring*, *Reconfiguration Coordination* and *Reconfiguration Effector* aspects). Only the architecture-specific aspect (i.e. the *Reconfiguration Analysis* aspect) changes at each level, because the context to manage (i.e. the architecture) is different. Thus, this approach provides a software architecture with the following properties: (i) flexibility, due to the use of dynamic reconfiguration mechanisms; (ii) maintainability, because aspect-oriented techniques are used to separate reconfiguration concerns from other concerns, and (iii) scalability, because management is decentralized.

Further works remain, as the dynamic generation of reconfiguration plans from high-level goals. We have used the PRISMA AOADL to define simple ECA policies, although other kind of approaches may be used, such as those related to the synthesis of tasks from high-level goals [38]. Our contribution is not the definition of the reconfiguration specification, but the explicit separation between the reconfiguration specifications and the mechanisms that support them. This way, business logic, reconfiguration specifications, and reconfiguration mechanisms can be maintained separately. The business logic can be dynamically changed by reconfiguration specifications, by means of reconfiguration mechanisms. And reconfiguration specifications can also be dynamically changed by using the reconfiguration mechanisms, treating them as any other concern of the system, as we stated in [11].

References

1. Aßmann, U.: *Invasive Software Composition*. Springer, 2003.
2. Ali, N., Ramos, I., Solís, C.: *Ambient-PRISMA: Ambients in mobile aspect-oriented software architecture*. Journal of Systems and Software 83(6): 937-958, 2010.
3. Batista, T., Tadeu, A., Coulson, G., et al.: On the Interplay of Aspects and Dynamic Reconfiguration in a Specification to Deployment Environment. In: *2nd European Conf. on Software Architecture*. LNCS, vol. 5292. Springer, 2008.
4. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. In: *Workshop on Self-Managed Systems*. Newport Beach, CA, 2004.
5. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution*, 17(5). Wiley, 2005.
6. Cámara, J., Salaün, G., Canal, C.: Composition and Run-time Adaptation of Mismatching Behavioural Interfaces. *J. of Universal Computer Science*, 14(13), Springer, 2008.
7. Canal, C., Pimentel, E., Troya, J.M.: Specification and Refinement of Dynamic Software Architectures. In: *Working IFIP Conference on Software Architecture (WICSA'99)*. San Antonio, Texas, USA, 1999.
8. Cazzola, W., Chiba, S., Saake, G.: Guest Editors' Introduction: Aspects and Software Evolution. *Transactions on Aspect-Oriented Software Development*, 4: 114-116. Springer, 2007.
9. Costa-Soria, C., Heckel R.: Modelling the Asynchronous Dynamic Evolution of Architectural Types. In: *Self-Organizing Architectures*. LNCS, vol. 6090, pp. 198-229. Springer-Verlag, Berlin Heidelberg, July 2010.
10. Costa-Soria, C., Pérez, J., Carsí, J.A.: Handling the Dynamic Reconfiguration of Software Architectures Using Aspects. In: *13th European Conf. on Software Maintenance and Reengineering*. Kaiserslautern, Germany, 2009.
11. Costa-Soria, C., Hervás-Muñoz, D., Pérez, J., Carsí, J.A.: A Reflective Approach for Supporting the Dynamic Evolution of Component Types. In: *14th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'09)*. 2-4 June 2009.
12. Cuesta, C.E., Romay, P., Fuente, P., Barrio-Solórzano, M: Reflection-Based Aspect-Oriented Software Architecture. In: *European Workshop on Software Architecture (EWSA'04)*. LNCS, vol. 3047. Springer, 2004.
13. Cuesta, C.E., Romay, P., Fuente, P.d.l., Barrio-Solárzano, M.: Architectural aspects of architectural aspects. In proc. of: *2nd European Workshop on Software Architecture (EWSA'05)*. LNCS, vol. 3527. Springer, 2005.
14. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards Architecture-Based Self-Healing Systems. In: *Workshop on Self-Healing Systems*. Charleston, South Carolina, 2002.
15. David, P., Ledoux, T.: An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. *5th Symp. on Software Composition (SC'06)*. Vienna, Austria, 2006.
16. Endler, M., Wei, J.: Programming Generic Dynamic Reconfigurations for Distributed Applications. In: *First International Workshop on Configurable Distributed Systems*. London, UK, 1992.
17. Garlan, D., Cheng, S., Huang, A., et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37:46-54. IEEE, 2004.
18. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: *Workshop on Self-Healing Systems*. Charleston, South Carolina, 2002.
19. Gomaa, H., Hussein, M.: Software reconfiguration patterns for dynamic evolution of software architectures. *4th Int. Conf on Software Architecture (WICSA'04)*. IEEE, 2004.
20. Greenwood, P., Blair, L.: A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects. *Transactions on AOSD II*. LNCS, vol. 4242, pp. 30-65. Springer, 2006.
21. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. *Computer*, 36(1):41-50. IEEE, 2003.

22. Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect-Oriented Programming. In *11th ECOOP'97*.
23. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: *ICSE - Future of Software Engineering (FOSE'07)*, pp. 259–268. IEEE, 2007.
24. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. *Transactions on Software Engineering*, 16(11):1293-1306. IEEE, 1990.
25. McKinley, P.K., Sadjadi, S., Kasten, E., et al.: Composing Adaptive Software. *Computer*, 37(7). IEEE, 2004.
26. Mens, T., Wermelinger, M.: Separation of concerns for software evolution. *Journal of Software Maintenance and Evolution*, 14(5):311-315. Wiley, 2002.
27. Morrison, R., Balasubramaniam, D., Kirby, G., et al.: A Framework for Supporting Dynamic Systems Co-Evolution. *Automated Software Engineering*, 14(3):261-292. Springer, 2007.
28. Oreizy, P., Gorlick, M., Taylor, R.N. et al: An Architecture-Based Approach to Self-Adaptive Software. *Intelligent Systems*, 14:54-62. IEEE, 1999.
29. Pérez, J., Ali, N., Costa, C., et al.: Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. In: *3rd International Conference on .NET Technologies*. Pilsen, Czech Republic, June 2005.
30. Pérez, J: *PRISMA: Aspect-Oriented Software Architectures*. PhD Thesis, Universidad Politécnica de Valencia, 2006.
31. Perez-Toledano, M.A., Navasa, A., Murillo, J.M., Canal, C.: TITAN: a Framework for Aspect Oriented System Evolution. In: *International Conference on Software Engineering Advances (ICSEA'07)*. IEEE, 2007.
32. Pérez, J., Ali, N., Carsí, J.A., et al.: Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. *Information & Software Technology*, 50(9-10):969-990. Elsevier, 2008.
33. Ritzau, T., Andersson, J.: Dynamic Deployment of Java Applications. In: *Java for Embedded Systems Workshop*. London, 2000.
34. Segal, M.E., Frieder, O: On-the-Fly Program Modification: Systems for Dynamic Updating.*IEEE Software*, 10(2) 1993.
35. Software Engineering Institute: *Ultra-Large-Scale Systems: Software Challenge of the Future*. Technical Report. Carnegie Mellon University, Pittsburgh, USA, 2006.
36. Selic, B.: The pragmatics of model-driven development. *Software*, 20(5). IEEE, 2003.
37. Serugendo, G.D.M., Gleizes, M.P., Karageorgos, A.: Self-organisation and emergence in MAS: An Overview. *Informatica (Slovenia)*, 30(1):45-54. 2006.
38. Sykes, D., Heaven, W., Magee, J. et al.: From goals to components: a combined approach to self-management. *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08)*. Germany, 2008.
39. Taylor, R.N., Medvidovic, N., et al.: *Software Architecture: Foundations, Theory and Practice*. Wiley, 2009.
40. Vandewoude, Y., Ebraert, P. et al.: Tranquillity: A low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *Transactions on Software Engineering*, 33(12):856-868. IEEE, 2007.
41. Wang, Q., Shen, J., Wang, X., Mei, H.: A Component-Based Approach to Online Software Evolution. *J. of Software Maintenance and Evolution*, 18(3). Wiley 2006.