Software Architecture Evolution

Olivier Barais, Anne-Françoise Le Meur, Laurence Duchien, Julia L. Lawall

1

Software Architecture Evolution

Olivier Barais[1], Anne Françoise Le Meur[2], Laurence Duchien[2], and Julia Lawall[3]

[1] Université of Rennes 1/IRISA/INRIA Triskell project
Campus de Beaulieu . F - 35 042 Rennes Cédex - France
barais@irisa.fr
[2] Université of Lille 1 / LIFL/INRIA ADAM project
Cité scientifique 59655 Villeneuve d'Ascq Cédex - FRANCE
lemeur,duchien@lifl.fr
[3] DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Summary. This chapter provides an overview, comparison and detailed treatment of the various state-of-the-art approaches to evolving software architectures. Furthermore, we discuss one particular framework for software architecture evolution in more detail.

1.1 Introduction

The role of software architecture in the engineering of software-intensive applications has become more and more important and widespread. Component based software architecture models the structure and behaviour of the system, including the software elements and the relationships between them. It becomes a base for the design process, a guide for the software development process and one of the main input to drive the integration tests. There exist currently a lot of Architecture Description Languages (ADLs) [1], which enable the architect to specify his software system. Indeed, during the design process, the architect relies on the ADL to create the architecture of his system by constructing and combining increasingly complex components and connectors.

Despite of the assets of these languages, most of them do not provide means to facilitate the evolution of a software system. However, one of the primary task of the architect is to ensure the quality of a system and its continued existence .Thanks to ADLs, building a functional architecture that contains only business concerns is in fact relatively easy. Supporting the evolution of a software architecture is more pervasive. The essence of the architect's task is to negotiate and balance the conflicting concerns of many diverse stakeholders and to anticipate the possible evolution of their requirements. This characterisation enforces the idea that traditional software architecture languages suffer from a number of key problems that cannot be solved without changing our point of view on the notion of software architecture.

The main problem is the lack of representation of changes in a software architecture description.

In this chapter, we identify two main architectural kinds of changes in software architectures: internal evolution and external evolution. Internal evolution models the changes of the topology of the components and interactions. Components and interactions may be created or destroyed during execution. This kind of evolution captures the dynamic of the system. External evolution allows the specification of the components and interactions to be changed during execution. It captures the needs for an architecture description to be adapted in order to cope with the evolution requirements. In this chapter, we study, through several approaches the lacks and the initial solutions to cope with evolution in a software architecture description. This study proposes a classification of these solutions according to the supported evolution kind.

The external evolution is studied in particular through the separation of concerns issue in a software architecture. Indeed, separation of concerns is traditionally achieved through modularity [2] and encapsulation. If the software component paradigm provides a good support for encapsulation with the help of information hiding, it suffers however from the problems that arise with the tyranny of the dominant decomposition [3]. Indeed, as the object-oriented paradigm, components fail to modularise some concerns because they allow a single dimension of decomposition. As a consequence, some concerns are spread over and repeated in several components in the system [4]. ADLs already provide some implicit separation of concerns: by describing the component configuration and the component interface, they separate the dimensions of composition from interaction. Nevertheless, the separation of these dimensions are not sufficient to modularise concerns such as security that crosscut the software architecture. In these cases, integrating a new concern or modifying an existing one require pervasively modifying the ADL specification, at all points affected by the concern. These modifications are low-level, tedious and error-prone, making the integration of such concerns difficult.

To address the complexity of integrating a new concern and modifying concern into a software architecture, several research works are inspired by Aspect-Oriented Software Development (AOSD) [5], which aims at improving the separation of concerns. In the spirit of Aspect-Oriented Programming (AOP), these approaches put the description of each concern in a separate architecture construct, that can automatically be integrated into an existing software architecture by a *weaver*. However, in specifying the integration of a new concern into an existing architecture, the coherence of the result remains a key issue. Because architectures are complex and aspects are invasive, many transformations caused by the composition may be needed to integrate or modify a concern, making the specification of the transformation highly error prone. Although the global coherence of an architecture can often be checked once the architecture is complete, these verifications are expensive as they consider the entire architecture. Furthermore, the interdependencies between architecture elements may make it difficult to identify the source of an error at this point. A second part of the chapter is focused on the coherence issue when a software architecture description changes. This issue is presented through a detailed presentation of TranSAT [6].

TranSAT proposes, through a specific language for specifying architectural aspects, a solution that ensures a number of coherence properties. This language is carefully designed to make certain unsafe transformations impossible to express, and allows static verification of additional coherence properties before aspect weaving or unweaving. In this approach, there remain, however, some properties that can only be checked dynamically, when integrating a new concern into an existing architecture. For these properties, TranSAT uses information found in the concern specification to limit the cost of the checks, by focusing on the parts of the architecture affected by the architectural aspect, and to present error messages in terms of the aspect elements. Overall, this approach provides verification early in the architecture development process, to enable the architect to rapidly and safely integrate new concerns.

The rest of this chapter is organised as follows. The first section presents several software architecture languages in order to identify the key concepts of these languages and their lacks. The following sections detail several initial solutions to cope with internal and external evolution. These sections provide a classification of these solutions. Section 1.5 presents the TranSAT approach in order to illustrate how addressing aspects at the architectural level can help to solve the AOSD evolution paradox and how an explicit specification of the weaving can help to guarantee the resulting architecture. Section 1.6 describes some related work and finally Section 1.7 concludes and details the remaining critical issues of software architecture.

1.2 Component-based software architecture: concepts and lacks

A software architecture describes the structure and behaviour of a software system. In a software architecture specification, a system is represented as a set of software components, their connections, and their behavioural interactions. Creating a software architecture promotes better understanding of the system, thus facilitates the design process. It also provides a basis for rigorous analysis of the system design, making possible the early detection of design errors and flaws that leads to improvements in software quality and help to ensure correctness.

An architectural description language (ADL) is used to describe a software architecture. It can be a graphical or a textual language, or include both. The advantage of using an ADL lies in the ability of rigorously specifying the global architecture of a system that can be thus analysed. An ADL may be associated with a set of tools that offer useful analysis for architectures specified in the language. An ADL is intended to be both human and machine readable and provides a high level of abstraction. It is a blueprint for the design and can support automatic generation of parts of software systems.

Lots of ADLs have been developed by either academic or industrial groups. If the various ADLs are different in many points, the ADL community generally agrees that the key elements of ADLs are the components, the connectors and the configuration [1]. A component represents a computational element with multiple ports to communicate with its environment. A Connector is a first class element to model

the interaction between components. Finally, the configuration describes how components and connectors are related into a system.

In this section, instead of providing a catalogue of ADL characteristics, we focus on three significant directions for software architecture definitions: (i) the specification and the analysis of distributed software behaviour, illustrated with Darwin [7] and Wright [8], (ii) the strong link with the implementation, that may be found in ArchJava [9], Fractal [10] and Sofa [11], and (iii) the building of architecture-driven software development environment, as promoted by ArchStudio [12], AcmeStudio [13] and SafArchie [14]. Finally, we provide a short evaluation of these works against the evolution issue.

Architecture analysis

Some ADLs, such as Wright [8] and Darwin[7], support the specification and analysis of relatively complex component communication protocols. First, Wright provides a formal basis for architectural description. It can be used to provide a precise, abstract meaning to an architectural specification and to analyse a component assembly. To further aid developers in the realisation and exploitation of architectural abstractions, Wright defines a set of standard consistency and completeness checks that can be used to increase the designer's confidence in the design of a system. These checks are defined precisely in terms of Wright's underlying model in CSP, and can be checked using standard model checking technologies. Darwin has the same background and the same goals than Wright. It is a formal language for describing software structures and network topologies. It models dynamic distributed systems. It possesses both a textual and graphical notation. It uses Finite State Process (FSP) Languages to specify system behaviour [15]. FSP provides a concise way of describing Labelled Transition Systems (LTSs).

System configuration and code generation

Lots of existing approaches decouple implementation code from architecture, allowing inconsistencies and violating architectural properties. ADLs like Fractal [10], ArchJava [9] or Sofa [11] seamlessly unify software architecture with implementation.

ArchJava proposes a hierarchical component model. Components can be either primitive or composite - a composite is built of other components, while a primitive component contains no subcomponents. Components communicate with their environment through ports. Port contains provided or required operations. ArchJava uses a type system to ensure that the implementation conforms to architectural constraints in a strict technical sense known as communication integrity. Communication integrity means that the components in a program only communicate along declared communication channels in their architecture.

Fractal is a general component model part of the OW2 consortium[4]. As ArchJava, it is based on a hierarchical component model. It supports the definition of

[4] http://www.ow2.org

primitive and composite components, bindings between the interfaces provided or required by these components, and hierarchic composition (including sharing). Unlike other Java-based component models, such as ArchJava, Fractal is not a language extension, but a run-time library which enables the specification and manipulation of components and architectures. Fractal distinguishes two kinds of components: primitives which contain the actual code, and composites which are only used as a mechanism to deal with a group of components as a whole, while potentially hiding some of the features of the subcomponents. Primitives can be simple, standard Java classes conforming to some coding conventions. Each Fractal component is made of two parts: a membrane which exposes the component's interfaces, and a cell which can be either a user class in the case of a primitive or other components in the case of a composite. All interactions between components pass through their controller. Finally, the Fractal component model is language independent, and fully modular and extensible. Fractal provides an XML based Architecture Description Language. It is based on three main constructs to specify component types, primitive templates and composite templates. A tool can parse Fractal ADL specification and instantiate the corresponding components. Contrary to Darwin or Wright, ArchJava and Fractal ADL do not provide any behaviour specification. The composition analysis is only structural. However, the abstract Fractal's or ArchJava's component models are efficient and appropriate for the implementation phase.

SOFA (SOFtware Appliances) is close to Fractal. It provides a platform for developing with software components. Like in Fractal, a SOFA application is viewed as a hierarchy of nested components. The component model is hierarchical, components can be a primitive or a composite. A component is described by its frame and its architecture. The frame is a component interface and the architecture is an abstract implementation. A frame defines provides-services and requires-services of the component. The frame can be implemented by more than one architecture. The architecture of a composite describes the structure of the component by instantiating direct subcomponents and specifying the interconnections between these subcomponents. The architecture reflects a particular grey-box view of the component - it defines the configuration of an architecture. Sofa provides a text-based ADL called Component Definition Language (CDL) which is based on OMG IDL. Communication among SOFA components can be captured formally. CDL embeds a process algebra called *behaviour protocols* to express the behaviour of each component. In this algebra, every method call or a return from a method call forms an event identified by an event-token. Behaviour protocols are regular-like expressions on the set of all event tokens, generating the set of admissible traces of the component.

Architecture Centric Integrated Development Environments

Most ADLs works today have been undertaken with academic rather than business goals in mind. As a result, the use of architecture languages and tools in industrial project is limited. On the fringes of ADLs, some projects aim at improving the use of software architecture concepts in software engineering industry. For example, ArchStudio [12] mainly developed by the Institute for Software Research

at the University of California, Irvine, is an architecture-driven software development environment. Indeed, while most development environments, like Microsoft Visual Studio and IBM Eclipse, are code-driven development environments, ArchStudio focuses on software development from the perspective of software architecture. It supports the C2 architectural style [16]. A C2 architecture is a hierarchical network of concurrent components linked together by connectors (or message routing devices) in accordance with a set of style rules. C2 communication rules require that all communication between C2 components be achieved via message passing. ArchStudio is extensible, it has lots of extensions to analyse, refine, or deploy an architecture specification. ArchStudio approach is a good way to improve the use of Software Architecture result in industrial project. With the same goal, AcmeStudio [13] is a customisable editing environment and visualisation tool for software architectural designs based on the Acme architectural description language (ADL). Acme is an ADL that can be used as a common interchange format for architecture design tools and as a foundation for developing new architectural design and analysis tools. AcmeStudio allows the designer to define new Acme families and customise the environment to work with those families by defining diagram styles.

Finally, SafArchie [14] is an abstract component model for designing a software architecture. The SafArchie component model describes the structure of a piece of software in terms of components, ports, operations and bindings. A component provides some services and may require some services from other components. Services can only be accessed through explicitly declared ports. A port is a binding point on a component that defines two sets of operations: provided operations and required operations. These operations make the dependencies between a component and its environment explicit. The set of operations provided by a port forms a service. An operation represents an action performed by a component. It is specified by its signature, which includes the name, the types of the parameters and the result of the operation, as well as the exceptions that it may raise. A binding associates a component's port with a port located on another component. There may only be one binding attached to a port. Two ports can be bound with each other only if the operations required by one port are provided by the other and vice-versa.

Like ArchJava, Fractal or Sofa, SafArchie is hierarchical in that a component is either primitive or composite. A primitive component can be seen as a basic building block in the component assembly. A composite component defines a given combination of primitive and composite components. The services provided and required by the child components of a composite component are accessible through delegated ports, which are the only entry points of a composite component. A delegated port of a composite component is connected to only one child component port. In SafArchie, each component interface is defined with contracts. These contracts clarify the structure but also the external behaviour of the components, which describes the component's interactions with its environment. SafArchie also provides a tool suite called SafArchie Studio which is built as a set of modules for ArgoUML. SafArchie Studio allows the designer to describe its architecture. It provides some connections with model checkers. Finally, it has a module to generate code towards ArchJava or Fractal. Even if, the architecture style and the refinement approach are different in

SafArchie Studio, the goal is the same as AcmeStudio or ArchStudio: To provide a complete tool suite to build, deploy and refine a software architecture and to transform ADLs in an effective vehicle for communication and analysis of a software system.

Evaluation: Managing software architecture evolution

Among the different problems which are not correctly addressed by the languages presented in this Section, the software architecture evolution is still a critical issue for the community. Indeed, the various languages presented in this section support the definition of a static software architecture. From this description, tools can check the correctness of the model. They can generate code. They can guarantee the consistency between a design and an implementation. However, a consequent problem is that a software architecture, once implemented in the software system, is, sometimes prohibitively, expensive to change. Worst, in all these language or associated tools, the evolution has not been taken into account. Due to the lack of first-class that represents the evolution, the models becomes obsolete quickly and their use is limited to an outdated documentation of the system.

If we confront the different languages presented in this section against the evolution issue, we can notice that:

- these languages can not describe the dynamic of the system. They give a snapshot view of the system that can become obsolete.
- these languages do not take care of external evolutions. The architecture analysis tools do not support incremental checks. Consequently, for each modification, the model checker has to re-check the entire system. At the implementation level, component-based software platforms suffer greatly from tangled code because a lot of functions, that belong to crosscutting concerns, are spread and repeated over different components. Consequently, the integration or the modification of a new concerns is difficult and error-prone. Finally, the different architecture development environments do not provide any shortcut to easily integrate or modify a concern which crosscuts several components in the architecture.

Following this observation, we will study in the next sections several initial solutions to handle the dynamic and to manage external evolutions of a component-based software architectures.

1.3 Dynamic software architecture description

A component-based software architecture can not be monolithic. The modularity brought by components drives the system to be dynamic. The topology of the components and interactions can be changed dynamically. New components and interactions may be created during execution. As the dynamic changes, applied to the architectural structure, may interact in subtle ways with the on-going computations of the system, software architectures have to take into account these changes. As

there is no consensus between the existing approaches that support the expression of the software architecture dynamism [17], this section argues that the dynamic issue can be tackled in two ways: either the ADL can support an explicit specification of the software architecture's dynamic in which case all of the possible evolutions of the system are foreseen in the software architecture description, or the ADL can define a frame for dynamic software architecture. This frame confines the potential evolution of the software architecture.

1.3.1 Explicit specification of the software architecture dynamic

Wright

The first approach which has worked on the expression of the dynamic in a software architecture is an extension of Wright [18]. This extension reuses the behaviour notation of Wright to model the reconfiguration. It allows the architect to view the architecture in terms of a set of possible architectural snapshots, each with its own steady-state behaviour. Transitions between these snapshots are accounted by special reconfiguration-triggering events. To introduce the dynamism in an architecture description, the architect has to modify the component's alphabet, and allow new messages to occur in port descriptions. Through this approach, the interface of a component is extended to describe when reconfigurations are permitted in each protocol in which it participates. Thanks to these new events, a "reconfiguration view" consumes these events to trigger reconfigurations. This extension allows the designer to simulate the evolution of its software architecture. Each potential snapshot can be checked by the model checker of Wright. This extension is especially tailored for dynamic software architectures. However two main problems limit its use in a concrete system development. First, the modification of the component is really heavy for the architect. This solution breaks the separation of concerns principle. Indeed, the reconfiguration is expressed at the same level as the functional behaviour of the component. Second, this approach is limited to model and to simulate dynamic systems with a finite number of configurations.

Fractal/FScript

Fractal ADL can not capture the dynamic of the system. Indeed, Fractal ADL is an XML configuration file used for the instantiation of the system. This file can really be compared to the configuration file of the Spring Framework[5] in which all the Beans/Components[6] are instantiated and interconnected. However, Fractal is a good candidate for the expression of the dynamic of the system. First, its run-time model is highly dynamic. Components or bindings can be instantiated at run-time programmatically. The configuration of a composite can be changed. Besides, Fractal is now associated with a scripting language, named FScript [19], used to program reconfigurations of Fractal components. The language and its implementation are designed to

[5] http://www.springframework.org/
[6] Bean for Spring and Component for Fractal

offer certain guarantees on the reconfigurations, by considering them as transactions. More precisely, the guarantees are Termination (a reconfiguration can not be infinite), Atomicity (reconfiguration is executed either completely or not at all), Consistency (the Fractal system resulting from a successful FScript reconfiguration is structurally consistent) and Isolation (there are no concurrent reconfigurations). Each FScript program can be triggered by an event occurring inside the application itself using reactive rules modelled after the *Event-Condition-Action* paradigm (ECA). Associated to Fractal ADL, FScript provides an interesting power of expression to model the dynamic of the system.

ArchJava

ArchJava is really close to the implementation (see section 1.2). To model the dynamic in ArchJava, only statically defined components can be dynamically instantiated and connected. At creation time, each component records the component instance that created it as its parent component. A component will eventually be garbage collected if there are no references to the component. Dynamically created components can be connected together at run-time. Communication integrity requires each component to explicitly document the kinds of architectural interactions that are permitted between its subcomponents. A connection pattern is used to describe a set of connections that can be declared at run-time. Whether a component or a connection can be dynamically created, ArchJava does not support the explicit component or connector destruction.

AADL

The AADL is a new international standard for predictable model-based engineering of real-time and embedded software [20]. Mainly inspired by MetaH [21], its fields of application are automotive, avionics, space and industrial control systems. AADL is a lower-level modelling language than the different ADLs presented in the previous section. Main concepts manipulated by this language are components, ports, threads, communication bus, *etc*. It models software topologies bound to execution platform topologies. AADL is interesting for two reasons. Far-off the concern of this chapter, it was one of the first AADL to model the quality of service in a component based software architecture. It can model times properties or latency. Secondly, more relevant for this chapter, it provides a mechanism of *mode* to model the reconfiguration of statically-known systems. Indeed, each AADL component can have modes. Modes represent alternative configurations of the component implementation. Only one mode is active at a time. At the level of system and process a mode represents possibly overlapping (sub-)sets of active threads and port connections, and alternative configurations of execution platform components, as well as alternative bindings of application components to execution platform components. As in Fractal, mode changes are specified as a state transition diagram whose states are the modes, and the transitions are triggered by events. Thus, AADL can model the reconfiguration of statically-known systems.

Evaluation

If the languages presented in this section make dynamic architectures explicit, they currently do not describe the dynamic with the same goal. The Wright extension or AADL model the dynamic to be able to simulate the evolution of the software architecture to check it. Fractal and ArchJava are more dedicated to implementing a dynamic software architecture. Among the shared features, we can observe than these approaches are based on a limited version of the CRUD[7] primitives, *i.e.*, they can create or destroy component or connection. However, although the separation between the reconfiguration policies and the rest of the software architecture is correct for Fractal/FScript and AADL, these policies are completely tangled into the components for ArchJava, and partially for Wright. If we consider the specification of the dynamic in the real-world, we can argue that the specification of all the possible reconfigurations is fastidious and limit the real dynamic of the software architecture.

1.3.2 A frame for dynamic software architecture

Contrary to an explicit specification of all the potential snapshots of the system configuration, other languages try to confine the potential evolution of the software architecture in what we call a frame for dynamic software architecture.

UML 2.0

UML 2.0 [22] permits the specification of logical components, i.e. specification level components (e.g., business components, process components) as well as deployed components (such as artifacts and nodes). It proposes to model the system as a hierarchy of nested components that provide and require interfaces. It provides support for decomposition through the new notion of *structured classifiers*. A structured classifier is a classifier (a type) that can be internally decomposed (`Classes`, `Collaboration`, and `Components`). New constructs to support decomposition have been introduced: `Part`, `Connectors`, and `Ports`. In UML 2.0, a component is viewed as a "*self-contained unit that encapsulates state + behaviour of a set of classifiers*". It may have its own behaviour specification and specifies a contract of provided/required services, through the definition of ports. To model the nested hierarchy, a component can be seen as parts because a component is a structured classifier. In this case, a part has type and a lower/upper bound multiplicity. Consequently, a connector does not represent a connection at the instance level but a potential connection at the type level. This kind of diagram can be really interesting to design a frame for software architecture. The variability of the software architecture is confined with the lower and the upper bound of subcomponents. Besides, each connection between component instances must match a connection pattern declared in the enclosing component between component types. However, UML 2.0 provides

[7] Create, Retrieve, Update, Delete

usual intentional flexibility. This kind of diagram is optional, and the nested hierarchy can be modelled only with instances that have a fix cardinality. In this last case, UML 2.0 does not provides any frame to confine the software architecture dynamic.

SafArchie

In the same lineage, SafArchie defines the concept of architecture type. An architecture type defines a set of possible configurations that must be respected by the software architectures. The defined constraints deal with component interfaces and identify relations and interactions between these component interfaces. Therefore, an architecture type represents a static view of a software architecture. These architecture types are used to check the structural and behavioural compatibilities between components. An architecture type is composed of six main elements: component type, composite type, bindings, port type, operation, and attribute. Designing a port type consists of identifying a set of operations that the port should provide or require. A port type corresponds to a set of operation signatures and their gathering together is guided by the system design. Component type defines all port types of the component and the minimum and maximum cardinality for each one. Composite type also identifies all the component types that it should contain and the minimum and maximum cardinality for each one. It defines the allowed interactions between these component types through the binding concept. A binding defines a possible interaction between two port types belonging to one or two component types that belong to the same component type. By this way, software architecture type is a set of structured constraints in terms of composite type, component type, and port type. Each typed software architecture should respect these constraints.

ACL

In [23], Tibermacine et al present an approach to preserve the architectural choices throughout the component-based software development process. They present an Architectural Constraint Language (ACL) as a means to formally describe architectural choices at all the stages. This language is based on the UML's Object Constraint Language (OCL) [24]. ACL limits the scope of an OCL constraint to a particular component, by slightly modify the syntax and semantics of the context part in OCL. At the syntactic level, every constraint context should introduce an identifier. This identifier corresponds to the name of a particular instance of the meta-class cited in the context. At the semantic level, ACL interprets a constraint with the meaning it would have in the context of the metaclass but limiting its scope only to the instance cited in the context. With a tiny modification of OCL, a component is able to define a constraint on its own structure. In ACL, only invariants can be expressed. Pre- and post-conditions are removed of the language. In fact, ACL can be used to define a frame for software architecture by defining a set of invariants that has to be respected by all the configuration of a system. The architect effort is more important than in UML 2.0 or SafArchie but the power of expression of ACL is better. Indeed, in the lineage of OCL, which has proved its benefits for the comprehension and the maintainability of models [25], ACL is easy to read for the designer.

ArchStudio

In one of the first version of ArchStudio, an effort has been done to govern runtime change [26]. They propose a mechanism for restricting changes that compromise system integrity. They use constraints to confine the different change that can occur but also to constrain when particular changes may occur. They support also transactions modifications. Consequently, during the course of a complex modification, the system's architecture may be in an invalid state before reaching a final valid state. In the same trend as ACL, Constraints legitimately restricts certain modifications paths.

1.3.3 Evaluation

These four works tackle the issue of the software architecture dynamic with the limitation of the allowed variability. The two main problems concern firstly, the lack of connection with component-based platforms. Indeed, these models could be seen as a repository which could evaluate if an explicit evolution is permitted. But, currently, no approach combines a scripting language to make explicit the dynamic at the platform level and an architecture type or a set of constraints to check if the proposed evolutions are correct from the modelling point of view. The second problem concerns the number of valid architectures that are defined with a set of constraints or an architecture type. In lots of case, this number is infinite. Consequently, it is impossible to check the correctness of all of these architectures. Currently, model checkers do not support the evaluation of an infinite architecture family.

1.4 Aspect-oriented architectures description language

1.4.1 Issue

The notion of *architectural view* / *architectural layer* / *architectural aspect* [8] comes from a very natural analogy: Just like in an house architecture we have distinct *view*/*plan*/*blueprints* describing distinct concerns of the same house (walls and spaces, electric wiring, water conducts), it seems reasonable to conceive a software architecture description as the composition of several concerns specifications (*view*, *aspect*, *plan*) reflecting several perspectives (viewpoint, concern) of the same software system. Indeed the target audiences for an architecture description are the various stakeholders of the system. Explicitly identifying these stakeholders reflects the multi-dimensional, multi-disciplinary nature of defining a software architecture. A stakeholder is any person, organisation or other entity with a particular interest in the architecture of the system. The reason to identify each stakeholder is to facilitate the comprehension of the system and its properties.

A software architecture description already provides an implicit separation of concerns: by describing the component configuration and the component interface,

[8] depending on the community

they separate the dimensions of composition from interaction. Nevertheless, the separation of these dimensions are not sufficient to modularise concerns such as security that crosscut the software architecture. The insufficient crosscutting concerns modularity complicates software evolution. To overcome this tension, this section presents several approaches that propose to promote Aspect-Oriented Software Development (AOSD) principles into ADLs. Through the description of these approaches, we will see how the improvement of the separation of concerns in a software architecture description can ease its evolution. We will also discuss the main issue raised by the introduction of AOSD into a software architecture.

1.4.2 Using Aspects in Architectural Description

IEEE 1471

IEEE Std 1471, named *Recommended Practice for Architectural Description of Software-Intensive Systems* [27] was the first formal standard to address what an architectural description (AD) is. It was developed by the IEEE Architecture Working Group between 1995 and 2000 with representation from industry, other standards bodies and academia. In 2006, IEEE 1471 became a draft international standard (ISO/IEC DIS 42010) and is now undergoing joint revision by IEEE and ISO.

IEEE 1471 is a conceptual framework. It establishes a set of content requirements on an architectural description. An architecture description contains any collection of products used to document an architecture. IEEE 1471 details how architecture descriptions should be organised, and their information content. The three main principles of this framework are:

- abstracting away from specific media (e.g., text, HTML, XML);
- being method-neutral: It is being used with a variety of existing and new architectural methods and techniques);
- being notation-independent: IEEE 1471 recognises that diverse notations are needed for recording various facets of architectures.

An architecture description in IEEE 1471 is governed by a set of rules. These rules define what it means for an AD to conform to the Standard. Even if IEEE 1471 does not provide the concept of aspect, it identifies the concept of architectural concerns which include: functionality, security, performance, reliability. All these concerns are generally regarded as early aspects. Under the rules of IEEE 1471, an architectural description must explicitly identify the stakeholders of the system's architecture and enumerate each architectural concern. If an AD does not address all identified stakeholders' concerns, it is, by definition, incomplete.

In IEEE 1471, an AD is organised into one or more architectural views. An architectural view is defined to be *a representation of a whole system from the perspective of a related set of concerns*. Each view has a governing architectural viewpoint. The viewpoint provides the set of conventions for constructing, interpreting and analysing a view, including the rules for determining whether it is well-formed. Each identified

stakeholder concern must be framed by at least one of the architectural viewpoints selected for use in an AD; if not, the AD is incomplete.

With regard to this conceptual framework, we can see that this standard has identified as a key concepts the issue of the different stakeholder management and the separation of concerns in a software architecture description. Currently, they do not propose to use Aspect-Oriented Modelling methods to compose this view. Consequently, they do not propose a clear join point model. They do not propose any pointcut language. The composition phase is not really formalised in the standard. This approach is not operational. However, we can really imagine to use IEEE 1471 as a framework associated to an ADL that supports AOSD.

Aspect-Oriented ADLs

Recently, to improve modularity and component reusability, several ADLs are motivated by the integration of new Aspect-Oriented (AO) abstractions such as, aspects, joinpoints, pointcuts and advices into the ADLs in order to address the modelling of crosscutting concerns in an architecture.

As software architecture descriptions rely on a connector to express the interactions between components, an equivalent abstraction must be used to express the crosscutting interactions. An Architectural Aspect, which is composed of *aspectual connector* and *aspectual component*, is a component that represents a crosscutting concern in a component-based architecture. The traditional connector can not model the crosscutting interaction because the semantics between a binding of two components is different than the semantics of a binding between an aspect and a base component. The first one defines usually a contract between a client and a supplier. The second one is more invasive. Due to the obliviousness principle, the base component must not be aware of the fact that it might be modified by an aspect component.

In order to express the crosscutting interaction, AspectualAcme [28] defines the Aspectual Connector, an architectural connection element that is based on the connector element but with a new kind of interface and a different semantics. The new interface makes a distinction between the different elements playing different roles in a crosscutting interaction, *i.e.*, affected traditional components and aspectual components; and captures how components are interconnected. The interface of an aspectual component contains some base roles, some crosscutting roles and a glue clause.

The glue clause specifies how an aspectual component affects regular components. There are three types of glue clause: after, before, and around. The semantics is similar to the one of advice composition from AspectJ [5]. The base roles can be linked to ports with a pointcut description. This expression matches the different ports affected by the aspectual component. Base role identifies the aspectual component that affects the base components.

Similarly, Fractal Aspect Component (FAC) [29] extends the Fractal ADL with Aspect Components (AC). Aspect Components are responsible for specifying existing crosscutting concerns in software architecture. Each aspect component can affect components by means of a special interception interface. Two kinds of binding between components and ACs are offered: a direct crosscut binding by declaring the

component references and a crosscut binding using pointcut expressions based on component names, interface names and service names.

A third approach, named PRISMA [30], should be mentioned. It integrates the software architecture and the AOSD approaches. Contrary to FAC or AspectualACME, PRISMA is a symmetrical approach because it does not consider functionality as a kernel entity different to aspects and it does not constrain aspects to specify non-functional requirements; functionality is also specified as an aspect. As a result, PRISMA provides a homogeneous treatment to functional and non-functional requirements. In PRISMA, aspects are first-order citizens of software architectures and represent a specific behaviour of a concern (e.g., safety, coordination, etc) that crosscuts the software architecture.

1.4.3 Evaluation

An interesting analysis and comparison of Aspect-Oriented ADL to complete this section can be found in [31, 32]. Complementary to these two studies, the approaches presented in this section illustrate that there is currently no consensus between existing approaches about the way to define an aspect in a software architecture. Among the other differences, some approaches consider that an aspect is composed of components, is a kind of component or a component is composed of aspects. However, most of them agree on that the semantics of the composition has to be extended to incorporate aspects into an ADL. As in software architecture there is a consensus that a software connector is the element that mediates interactions between components, several approaches modify the semantics of the connector to reflect the concepts of AOSD in a software architecture description.

As illustrated in [33], in addition to separating the different concerns during software development, AOSD can help to overcome many of the problems related to software evolution. Improving the separation at the architecture level will help to coordinate the different stakeholders of the system and improve the ability to modify only one concern independently of the others. Nevertheless, integrating or modifying a concern requires invasively modifying the ADL specification, at all points affected by the concern. These modifications are low-level, tedious and error-prone, making the integration of such concerns difficult. As pointed out by the AOSD evolution paradox [34], the evolution of a concern can break the consistency of the software architecture. For this reason, we claim in the second part of this chapter that the consistency of a base architecture modified by an aspect is a key issue for the software architecture community. To illustrate the problem and evaluate an initial solution, we propose to study in depth TranSAT: a framework for integrated stepwise new concerns in a software architecture.

1.5 Safe integration of new concerns in a software architecture

1.5.1 Overview of TranSAT

In this section, to motivate the breaking consistency issue, we present an overview of the TranSAT framework, through the example of a banking software architecture. We first describe the architecture and then show how to use the TranSAT framework to extend this architecture with an atomicity concern. Finally, we consider some of the issues that confront an architect when specifying a crosscutting concern.

Example

Our example banking application manages the withdrawal and deposit of money between savings and checking accounts. This application is represented by the software architecture shown in Figure 1.1, which is specified using the SafArchie ADL (see Section 1.2).

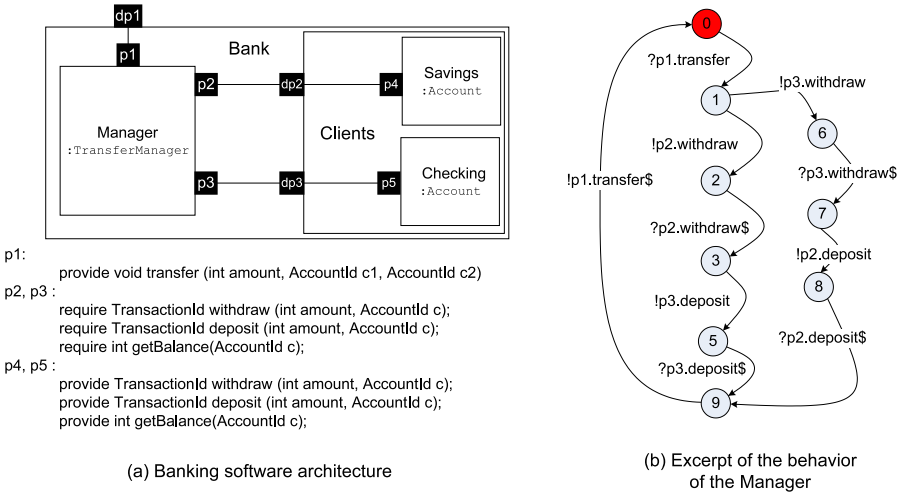


Fig. 1.1. Banking software architecture

Figure 1.1(a) gives the structural description of the banking architecture. The structure is described in terms of composites (`Bank`, `Clients`), components (`Manager`, `Savings`, `Checking`), ports (`p1` to `p5`), delegated ports (`dp1` to `dp3`) and bindings. Ports contain operations; for example, the operations `withdraw` and `deposit` are provided by the ports `p4` and `p5` respectively. A port must contain at least one operation, must be part of exactly one component, and must be bound to exactly one other port, in some other component. Operations are either provided or required. Bound ports must contain compatible operations; for example, port `p2` requires the operations provided by port `p4`. Delegated ports do not contain any

operations; they define the interface of a composite, exporting the operations of the composite's components.

Figure 1.1(b) gives the behavioural description of one of the components, `Manager`. The behaviour is specified in terms of an Input/Output Automaton [35] that describes the sequences of messages that a component may receive and emit. The notation used in these automata is as follows. For a provided operation `op1`, the message `?op1` represents the receipt of a request and the message `!op1$` represents the sending of the response. `?op1` must precede `!op1$`, but they can be separated by any number of messages, representing the processing of `op1`. For a required operation `op2`, the message `!op2` represents the sending of a call and the message `?op2$` represents the receipt of the response. Sending a call is a blocking operation, and thus `!op2` must always be immediately followed by `?op2$`. Using this notation, the behaviour shown in Figure 1.1(b) specifies that when the `Manager` receives a transfer request, it makes a withdrawal from one of the two accounts and a deposit to the other one.

Integrating an atomicity concern using the TranSAT framework

The TranSAT framework manages the integration of a new concern, represented as an *architectural aspect*, into an existing architecture, referred to as a *basis plan*. The software architectural aspect represents the new concern in terms of a *plan*, a *join point mask*, and a set of *transformation rules*. The plan describes the structure and behaviour of the new concern. The join point mask defines the structural and behavioural requirements that the basis plan must satisfy so that the new concern can be integrated. The transformation rules specify the means of composing the new

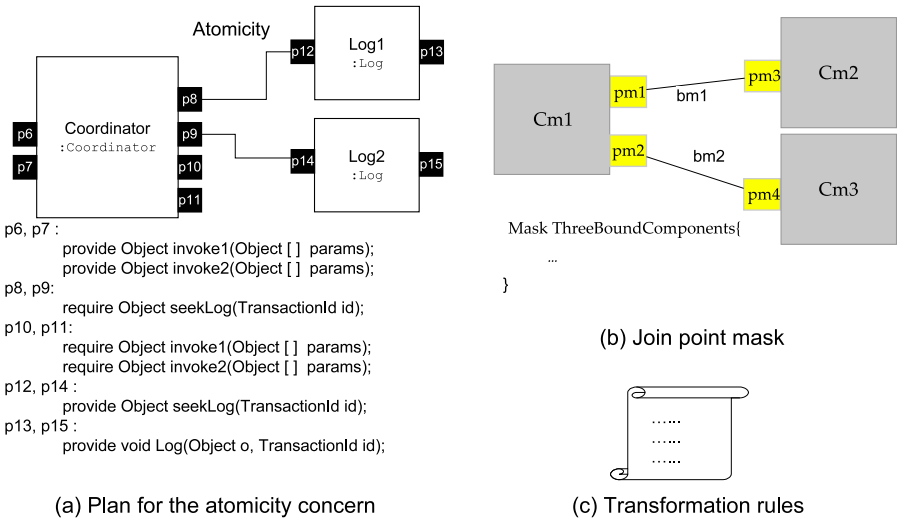
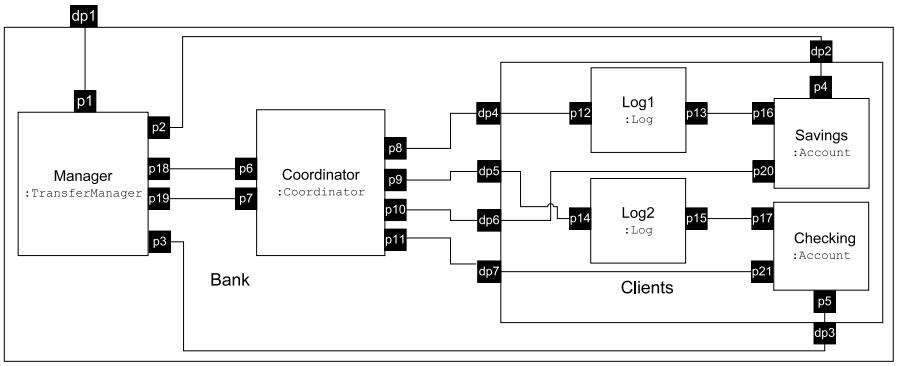


Fig. 1.2. Architectural aspect for the atomicity concern

plan with the basis plan. Given a software architectural aspect, the architect specifies where it should be added to the basis plan. The TranSAT *weaver* then checks that the selected point in the basis plan matches the join point mask, instantiates the transformation rules according to the architectural entities matched by the join point mask, and executes the instantiated transformation rules to compose the new concern into the basis plan.

As an example of the use of these constructs, we consider how to make the banking transactions atomic. This concern is crosscutting, in that it affects both the `Manager` and the savings and checking accounts. The architectural aspect related to atomicity is shown in Figure 1.2. The new plan corresponding to the atomicity concern keeps a log of certain operations and enables these operations to be rolled back when an error occurs. Specifically, the `Log` components provide operations to keep a log and to retrieve information from this log, and the `Coordinator` component triggers rollbacks when appropriate, guaranteeing the atomicity property. The join point mask specifies that this plan can be composed in a context consisting of one component `Cm1` attached to two other components `Cm2` and `Cm3`. Some constraints (not shown) are also placed on the operations in the ports connecting these components. In the banking software architecture, the join point mask is compatible with the integration site consisting of the `Manager`, `Savings` and `Checking` components. Finally, the transformation rules connect the ports of the plan to the ports of the selected integration site, and make other appropriate adjustments. In the case of the banking architecture, the result of the composition is shown in Figure 1.3.




```
p2, p3:
        require int getBalance(AccountId c);
p4, p5:
        provide int getBalance(AccountId c);
p16, p17 :
        require void log(Object o, TransactionId id);
p18, p19 :
        require TransactionId withdraw (int amount, AccountId c);
        require TransactionId deposit (int amount, AccountId c);
p20, p21 :
        provide TransactionId withdraw (int amount, AccountId c);
        provide TransactionId deposit (int amount, AccountId c);
```

Fig. 1.3. Transformed banking software architecture

Issues

To specify the integration of a crosscutting concern, the architect must describe how to modify the component structure, behaviour, and interfaces. This task is highly error prone, as many modifications are typically required, and these modifications can have both a local impact on the modified elements and a global impact on the consistency of the architecture.

Typically, a component model places a number of requirements on local properties of the individual architectural elements. For example, in SafArchie, the ADL on which TranSAT is built, it is an error to break a binding and then leave the affected port unattached, or to remove the last operation from a port, and then leave the port empty. The construction of the behaviour automaton associated with each component is particularly error prone, because it must be kept coherent with the other elements of the component and because of the complexity of the automaton structure. For example, in SafArchie, all of the operations associated with the ports of a component must appear somewhere in the component's behaviour automaton. When the ADL separates the structural and behavioural descriptions, it is easy to overlook one when adding or removing operations from the other. An automaton must also describe a meaningful behaviour; at a minimum that for each operation, a call precedes a return and every call is eventually followed by a return from the given operation.

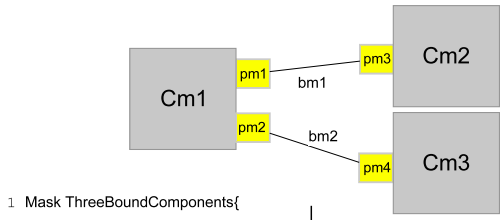
The architecture must also be globally coherent. The most difficult point raised by this coherence issue lies mainly in the behaviour of the architecture. So that the application can run without deadlock, it must be possible to synchronise the behaviour of each component with that of all of the components to which it is bound by its ports. Any change in the behaviour of a single component can impact the way it is synchronised with its neighbours, which in turn can affect the ability to synchronise their behaviours with those of other components in the architecture. The interdependencies between behaviours can make the source of any error difficult to determine.

1.5.2 A specific language for Software Architecture Transformation

In this subsection we present the TranSAT's transformation language for specifying the elements of an architectural aspect: plan, join point mask and transformation rules. The component assembly shown in Figure 1.2 (a) is an example of a plan, showing only structural information. We also present the join point mask and the transformation rules. The use of the language is illustrated through the definition of the atomicity aspect.

The join point mask

The join point mask describes structural and behavioural preconditions that a basis plan must satisfy to allow the integration of the new concern. It consists of a series of declarations specifying requirements on the structure and behaviour of the components available at the integration site.




```
1  Mask ThreeBoundComponents{
2        … //Component and port
           declarations corres-
         -ponding to the diagram
3        OperationMask opm1, opm2,
           opm3, opm4, opm5, opm6,
           opm7, opm8;
4        opm1.polarity = require;
5        opm2.polarity = require;
6        opm3.polarity = require;
7        opm4.polarity = require;
8        opm5.polarity = provide;
9        opm6.polarity = provide;
10       opm7.polarity = provide;
11       opm8.polarity = provide;
12   pm1.operations = {opm1, opm2, *};
13   pm2.operations = {opm3, opm4, *};
14   pm3.operations = {opm5, opm6, *};
15   pm4.operations = {opm7, opm8, *};
16   opm1 = inverse(opm5);
17   opm2 = inverse(opm6);
18   opm3 = inverse(opm7);
19   opm4 = inverse(opm8);
20 … //Binding declarations}
```

Fig. 1.4. Join point mask definition

Figure 1.4 illustrates a join point mask suitable for use with the atomicity plan (Figure 1.2 (a)). For readability, some of the declarations are elided or represented by the diagram at the top of the figure. The diagram specifies that some component `Cm1` must be connected to two other components `Cm2` and `Cm3`. The remaining declarations define a series of placeholders for operations (line 3), specify whether these operations must be declared as provided or as required (lines 4-11) and specify that they must be associated with the ports `pm1` to `pm4` (lines 12-15). Finally, lines 16-19 ensure that the operation `opm1` is the inverse of operation `opm5` in the bound port, and similarly for `opm2` and `opm6`, `opm3` and `opm7`, and `opm4` and `opm8`. Operations are inverse if they have the opposite polarity, the same name and compatible types. In the banking architecture, these constraints would, for example, allow the architect to select the required operation `withdraw` in port `p2` as `opm1` and the provided operation `withdraw` in port `p4` as `opm5`. In this example, the join point mask does not specify any behavioural requirements. If needed, the constraints on the behaviour of a component mask can be specified in terms of a sequence of messages.

The transformation rules

The transformation rules describe precisely how to compose the new plan with a basis plan. They specify the various transformations to perform on the elements defined in the new plan and the join point mask, as well as their application order. The language provides two kinds of transformation primitives: *computation transformation primitives* and *interaction transformation primitives*. The computation transformation primitives specify the introduction of new ports and operations in primitive components, in order to adapt the component behaviour. The interaction transformation primitives manage the insertion and deletion of component bindings and man-

age the composite content, in order to reconfigure the software architecture. Overall TranSAT is targeted towards introducing new concerns into existing architectures rather than removing existing functionalities. Thus, the language has been designed to prevent transformations that remove existing behaviours.

Computation transformation primitives

Table 1.1 shows the primitives used to manage the structural transformation of primitive component interfaces. These primitives allow the architect to create new ports and operations, to destroy empty ports and to move an operation from one port to another.

	Port	Operation
create	Port *Pr* in *Cp*;	Operation *Or* = *op* in *Pr*; Operation Or_1 = *op* replaces Or_2;
destroy	*Pr*.destroy();	N/A
move	N/A	*Or*.move(*Pr*);

Cp: ComponentRef, Pr: PortRef, Or: OperationRef,
op ::= *Or* | inverse(*Or*), N/A: Not applicable

Table 1.1. Computation transformations

Adding an operation to a port has an impact on the behaviour of the associated component. When a new copy of an operation is added to a port using the operation `Operation Or = op in Pr`, the architect must explicitly specify how the messages associated with the newly added operation `op` fit into the behaviour of the component to which the operation is attached. The transformation of the behaviour automaton is specified using the pattern-matching syntax *template* `=>` *result*. Such a rule inserts the messages associated with the new operation, `op`, before, after, or around the calling or responding messages associated with some existing operation, `m`. The template specifies the sequence of messages on `m`, possibly separated by any sequence of messages, `x`. The result describes how messages associated with the new operation, `op`, are interleaved with this sequence.

The following lines illustrate the use of the automaton transformation rules:

```
?m → x → !m$ ⇒ ?m → !op → ?op$ → x → !m$;                    1
?m → x → !m$ ⇒ ?m → (!op → ?op$ → x | x) → !m$;              2
```

In line 1, the template describes the receipt of a call to `m` followed by any number of messages, followed by the sending of `m`'s response. The result specifies that following the receipt of the call to `m`, the component sends a call to `op` and waits for the response before performing any further computation. The use of the new operation `op` at runtime can also be conditional. In line 2, the transformed component either calls `op`, waits for the response, and then performs the sequence `x`, or performs `x` alone, ignoring the added `op` operation.

Interaction transformation primitives

The interaction transformation primitives manage the reconfiguration of the software architecture. As shown in Table 1.2, operators are provided to create and destroy bindings, to create composites either at the top level or within another composite, and to move one composite $\texttt{Cr}_1$ or one component *Cp* into another composite $\texttt{Cr}_2$.

	Binding	Composite	Component
create	Binding *Br* = $\{Pr_1, Pr_2\}$;	Composite *Cr*; Composite Cr_1 in Cr_2;	N/A
destroy	*Br*.destroy();	N/A	N/A
move	N/A	Cr_1.move(Cr_2);	*Cp*.move(Cr_2);

Cp: ComponentRef, Cr: CompositeRef, Pr: PortRef,
Br: BindingRef, N/A: Not applicable

Table 1.2. Interaction transformations

Example

We use the atomicity example to illustrate the use of the computation and interaction transformation primitives. In this example, composing the new plan requires (i) interposing the `Coordinator` component between the original component `Cm1` (instantiated as `Manager` in the banking case) and the operations that are to be made atomic, and (ii) inserting the `Log` components in front of the components `Cm1` and `Cm2` providing these operations (instantiated as `Savings` and `Checking` in the banking case). Figure 1.5 shows the rules that carry out these transformations.

In the join point mask, the operations to be made atomic are specified to be in a port that may contain other operations, *e.g.*, port `pm1` includes the operations `opm1`, `opm2`, and some unknown list of operations `*` (line 12 in Figure 1.4). So that the atomicity concern does not have to take into account these other operations, lines 2-11 in Figure 1.5 move the operations to become atomic into newly created ports, `p18` to `p21`. This transformation may cause the ports matched by the join point mask to become empty. Accordingly, lines 13-16 apply the `destroy` operation to these ports, causing them to be destroyed if they are empty. When the atomicity concern is composed into the banking software architecture, the ports matched by `pm1` to `pm4` are not destroyed because they contain the operation `getBalance`.

The ports of the `Coordinator` are then updated with references to the operations to be made atomic. For each port, `p6`, `p7`, `p10`, and `p11`, the generic operations `invoke1` and `invoke2` are replaced by the inverses of the corresponding operations in the ports `p18` to `p21` (lines 18-25). These transformations implicitly update the `Coordinator`'s behavior automaton by replacing the messages associated with the `invoke` operations by the messages associated with the new operations.

To insert the `Log` components in front of `Cm2` and `Cm3`, new ports must be added to `Cm2` and `Cm3` and these ports must be instantiated with references to the `log`

```
// Cm1 transformation
Port p18 in Cm1;
opm1.move(p18);
opm2.move(p18);
... Similarly for the port p19 and the operation masks opm3 and opm4 of pm2

// Cm2 transformation
Port p20 in Cm2;
opm5.move(p20);
opm6.move(p20);
... Similarly for the port p21 in Cm3 and the operation masks opm7 and opm8 of pm4

// Port destruction
pm1.destroy();
pm3.destroy();
... Similarly for the ports pm2 and pm4

// Coordinator transformation
Operation o6a = inverse(opm1) replaces p6.invoke1;
Operation o6b = inverse(opm2) replaces p6.invoke2;
... Similarly for the operations of the port p7

Operation o10a= inverse(opm5) replaces p10.invoke1;
Operation o10b= inverse(opm6) replaces p10.invoke2;
... Similarly for the operations of the port p11

// Introduction of p16 within Cm2
Port p16 in Cm2;
Operation o16 = inverse(p13.log) in p16;
?opm5 → x → !opm5$
        ⇒ ?opm5 → x → !o16 → ?o16$ → !opm5$;
?opm6 → x → !opm6$
        ⇒ ?opm6 → x → !o16 → ?o16$ → !opm6$;

// Introduction of p17 within Cm3
... Similarly to Cm2 for the transformation of the port p17

// Component introduction
Coordinator.move(Cm1.parent);
Log1.move(Cm2.parent);
Log2.move(Cm3.parent);

// Binding creation
Binding b6 = {p18, p6};
Binding b7 = {p19, p7};

Binding b10 = {p10, p20};
Binding b11 = {p11, p21};

Binding b13 = {p13, p16};
Binding b15 = {p15, p17};
```

Fig. 1.5. Transformation rules for the atomicity concern

operation. We focus on the transformation of `Cm2`, as the transformation of `Cm3` is similar. Lines 28-29 add the port `p16` and copy the `require` counterpart of the `Log` component's `log` operation into this port. Because `log` is a new operation for `Cm2`, we must specify where it fits into `Cm2`'s behaviour. Lines 30-33 specify that `Cm2` sends a call to this new operation whenever it is about to return from either of the operations to be made atomic.

The remaining rules transform the interaction between components. Lines 39-41 add the components of the plan to the basis plan. In these rules, for any outermost component or composite referenced by`C` in the join point mask, `C.parent` represents the parent of the element to which `C` is matched in the basis plan. As the component model is arborescent, each component or composite has at most one parent. If there is no parent, the enclosing transformation is not performed. Finally, lines 43-50 connect the components at the various ports. TranSAT automatically adds delegated ports, *e.g.*, `dp4` in Figure 1.3, as needed. This behaviour of the transformation engine improves the genericity of the architectural aspect. Applying these transformation rules to the join point between the `Manager`, `Savings` and `Checking` components shown in Figure 1.1 (a) produces the software architecture shown in Figure 1.3 (structural information only).

1.5.3 Static checking of the transformation

A goal of TranSAT is to ensure that the composition of a new concern produces a valid software architecture. Accordingly, TranSAT statically checks various properties of the aspect at creation time and dynamically checks that the aspect is compatible with the insertion context when one is designated by the architect.

Static properties and checks

Given an aspect, TranSAT first checks that its various elements are syntactically and type correct. For example, a join point mask must declare that a port contains elements of type `Operation` and a `Binding` transformation must connect two ports. TranSAT then performs specific verifications for the plan, the join point mask, and the transformation rules.

Plan. TranSAT requires that the plan be a valid software architecture according to the component meta-model of SafArchie, except that it may contain unattached ports. For example, TranSAT checks that all bindings connect ports that contain compatible operations and that the automata describing the behaviours of the various components in the plan can be synchronised.

Join point mask. The variables declared by the join point mask represent the fragments of the basis architecture that can be manipulated by the transformation rules. Unlike the plan, the join point mask need not be an enriched architecture specification and thus TranSAT does not check that *e.g.* operations are specified for all ports or automata can be synchronised. These properties are, however, assumed to be satisfied by the elements matched in the basis architecture. TranSAT does verify the consistency of the information that is given, for example that any automaton provided uses operations in a manner consistent with their polarity.

Transformation rules. TranSAT ensures the safety of the transformation process by a combination of constraints on the transformation language and verifications performed statically on the transformation rules.

Several features of the transformation language have been designed to prevent the architect from expressing unsafe transformations. For example, the SafArchie

component meta-model requires the insertion of delegated ports whenever a binding crosses a composite boundary. TranSAT introduces these delegated ports automatically, relieving the architect of the burden of identifying the composites between two ports, reducing the size of the transformation specification, and eliminating the need to fully specify composite nesting in the join point mask. The SafArchie component model also requires that each architectural element have a parent, except for the outermost components or composites. The transformation language enforces this constraint by combining the creation of a new element with a specification of where this element fits into the architecture; for example, `Port Pr in Cr` both creates a new port `Pr` and attaches this port to the composite `Cr`. Finally, a common transformation is to replace an operation in a port by another operation, which requires updating both the port structure and the automaton of the associated component. The transformation language combines both operations in the declaration `Operation` $\texttt{Or}_1$ `= op replaces` $\texttt{Or}_2$.

Other safety properties are not built into the syntax of the transformation language, but are checked by analysis of the transformation rules. To do so, the operational semantics of the transformation language is formalised. Thanks to these formalisation, the analysis simulates the execution of the transformation rules on the various elements identified by the plan and the join point mask. At the end of the simulation, global post-conditions are checked to guarantee that the pattern will not break the software architecture consistency. For example, a post-condition guarantee that every element has at least one subelement except operations and join point mask elements for which no subelements are initially specified. A similar analysis checks various properties of bindings: every port is connected to some other port by a binding, the connected ports are not part of the same component, the operations of the connected ports are compatible, etc. Another analysis checks that for each component, the automaton and the set of operations in the various ports are kept consistent. A more detailed description of these checks is provided in [6].

1.5.4 Dynamic checks

An architect integrates an aspect by designating a fragment of the existing architecture to which the aspect should be applied. TranSAT checks that the fragment matches the join point mask, to ensure that the fragment satisfies the assumptions under which the safety of the transformation rules has been verified. However, because the join point mask does not describe the entire basis architecture, the static checks of the different elements of the aspect are not sufficient to guarantee the correct composition of a new plan into a basis plan. Consequently, dynamic verifications of some structural and behavioural properties of the architecture are performed during the composition process.

The dynamic structural verification consists of checking the compatibility between the newly connected ports, according to the definition of the port compatibility of SafArchie [14]. Concretely, based on transformation rules that have been applied, the analysis builds a list containing the newly created connections as well as the connections between ports that have been modified by the transformations.

For each of these connections, the connected ports are verified to contain compatible operations. The other connections do not need to be checked as they are not affected by the transformations and their correctness has been previously verified during the analysis of the basis plan or the aspect plan.

Adding new components and behaviours to a fragment of an architecture can change the synchronisation at the interface of the fragment, and thus have an effect on the synchronisation of the rest of the architecture. The use of a architectural aspect localises the modifications to a specified fragment of the existing architecture. The process of resynchronisation thus starts from the affected fragment and works outward until reaching a composite for which the interface is structurally unchanged and the new automaton is bisimilar to the one computed before the transformation. The bismilarity relation ensures that the transformation has no impact on the observable behaviour of the composite, and thus the resynchronisation process can safely stop [36].

If the transformation of the architecture fails, any changes that were made must be rejected. Before performing any transformations, TranSAT records enough information to allow it to roll back to the untransformed version in this case.

1.5.5 Assessment

In Section 1.1, we observed that the architect who integrates a new concern without a dedicated framework, can use the general architecture analysis tools to check the validity of the resulting architecture after the composition is complete. This approach, however, can give imprecise error messages, because the resulting architecture does not reflect the transformation step that caused the problem, and can be time consuming, due to the automaton synchronisation that is part of this validation process. In this section, we briefly describe how a composition framework like TranSAT can address these issues.

Because the static verifications have a global view of the transformations that will take place, they can pinpoint the transformation rules that can lead to an erroneous situation. For example, if an operation is moved from a port of the join point mask, the port may become empty, resulting in an erroneous software architecture. While SafArchie would simply detect the empty port, TranSAT can, via an analysis of the complete set of transformation rules, detect that there is a risk that a port contains only one operation, that a `move` is performed on the operation in this port, and that a `destroy` is not subsequently applied to this port. Using this information, TranSAT can inform the architect of problems in the transformation rules, before any actual modification of the architecture has taken place. Obtaining this feedback early in the composition process can reduce the overall time required to correctly integrate the new concern.

Because the dynamic verifications are aware of the exact set of components that are modified by the composition, they can target the resynchronisation of the automata accordingly. As synchronisation is expensive, reducing the amount of resynchronisation required can reduce the amount of time required to integrate a new concern, making it easier for the architect to experiment with new variants.

1.5.6 Discussion and tool suite

TranSAT is really different compared to other AO-ADLs. Contrary to most AO-ADL that try to create new first-class entities that extend the concept of Component and Connector, TranSAT architectural aspect is a composite entity that contains a set of components and connectors that must be inserted. Consequently, in TranSAT the weaving is not a composition operator with a fix semantics. The transformation rules contained in the architectural aspect description refine this semantics. TranSAT's use of transformations to weave aspects model raises the issue about the difference between model weaving and model transformation. This issue is not limited to the TranSAT approach. Indeed, in many Aspect-Oriented Modelling approaches (AOM), a design is presented in terms of multiple user-defined views (aspects) and model composition is often carried out to obtain a model that provides an integrated view of the design. In this different approaches, model composition involves merging or weaving two or more models to obtain a single model. The apparent similarities between model weaving and model transformations have already been discussed in [37]. As a result, even if TranSAT can not be compared directly to others AO-ADLs, it can be classified without restriction as an AOM approach.

The TranSAT framework enriches the SafArchie tool suite to assist the architect during the specification of the system. For TranSAT, three main static modules have been developed. The first one permits the static checking of an architectural aspect. The second one assists the architect to compose an architectural aspect with an existing architecture by highlighting the different join points matched by the join point mask. Finally, the transformation engine performs the transformation to weave an architectural aspect into an architecture.

The static checking of the architectural aspect has been developed as an if-then clause in Drools[9] and in Prolog[10]. The comparison of the Drools rules and the Prolog rules shows that you find the same conditions and the same consequences. The difference is that the rule is called explicitly in Prolog whereas it is chosen by the rule engine in Drools. Contrary to the Prolog implementation, the Drools implementation does not output the reason of the failure of a transformation rule. Although it is possible to check the reasons of the failure, it is not handy to do so with forward-chaining. The time spent for the verification using the Prolog implementation is quite similar for both implementations. The rules check the initial state and the final state of the transformation environment and perform the structure and connection analysis. The Drools implementation relies on 39 Drools rules. The Prolog implementation is composed of 21 Conditional Transformations[11] and 18 Prolog rules.

To detect the join point that can be matched by a join point expression, a module has been implemented in three ways with AGG[12], DROOLS and Prolog. The idea is the same for all the implementations. First we fill the knowledge base with facts that correspond to the elements of the software architecture. Then we transform the

[9] http://legacy.drools.codehaus.org/
[10] http://www.swi-prolog.org/
[11] http://roots.iai.uni-bonn.de/research/jtransformer/cts
[12] http://tfs.cs.tu-berlin.de/agg/

join point mask into a set of rules. Finally we make the rule engine use these generated rules to find all the matching facts in the knowledge base. The main difference between those implementations concerns the efficiency of the search. The AGG implementation takes more time than the two others because of the graph matching process. Drools arrives at the second position. The extra time for the Drools implementation comes mainly from the compilation of the Drools rules. Since the rules are generated from the join point mask, the cost of the rules compilation can not be reduced.

Finally, the transformation engine has been developed with two concurrent techniques: AGG and Prolog. The AGG rules are generated from the TranSAT transformation rules. There are 167 graph transformation rules created for the atomicity composition. The host graph generated from the software architecture is composed of 77 nodes and of 90 edges. The transformed host graph holds 115 nodes and 138 edges. There are 34 conditional transformations that perform the TranSAT transformations. The software architecture is described by 145 predicates in the knowledge base. After the atomicity integration, the transformed software architecture is specified by 210 predicates. The atomicity integration in the reservation software architecture takes 50 times more time with AGG than with Prolog. Once again, the graph matching is responsible for the effeciency difference.

1.6 Related work

Separation of concerns in software modelling

Our work relates in several points to the works in Aspect Oriented Modelling. In this domain, the composition of the different concern models identified in the early stage of the development process is an important issue when building a model with aspects. Main works consist in being able to compose UML diagrams. For example, France et al [38] have developed a systematic approach for composing class diagrams in which a default composition procedure based on name matching can be customised by user-defined composition directives. These directives constrain how class diagrams are composed. The framework identifies automatically conflicts between models that have to be composed and it solves them thanks to the composition directives. Composition directives address the weaving only from the structural point of view. It considers the composition as a model transformation. Besides, it is a symmetric AOM approach in which they do not differentiate between aspect model and base model. Consequently, they do not provide currently a pointcut language to manage the composition.

Close to model composition directives, Muller et al [39] present a means to build an information system with parametrised models. They use a model composition operator to combine models. This work focuses on the idea that model parametrisation allows the reuse of models in multiple contexts. The need to compose parametrised models and apply them to a system according to alternative and coherent ordering rules is highlighted. However, as with model composition directives, their work only supports the composition of class diagrams and can not compose dynamic diagram. This approach does not provide aspectual composition operators and as such does not support the composition of aspect models.

In the same idea, Theme/UML extends the UML to support the specification of symmetric concern models . A symmetric decomposition model is one in which both base and aspect concerns are defined in separate models at the same level of abstraction. At the modelling level, a base concern represents behaviours that are not crosscutting. An aspect concern represents behaviours that are primarily crosscutting. The Theme/UML approach introduces a *theme module* that can be used to represent a concern at the modelling level. Themes are declaratively complete units of modularisation in which any of the diagrams available in the UML can be used to model one view of the structure and behaviour the concern requires for execution. The structure and behaviours, that are needed to execute a concern, are specified within the theme module. In Theme/UML, a class diagram and sequence diagrams are typically used to describe the structure and behaviours of the concern being modelled. Classes and Methods defined on these diagrams describe the structure of these entities in the scope of the concern. Sequence diagrams describe the behavioural interactions that can occur between classes when the concern is executed. Aspect concerns are represented as parametrised themes. These themes are parametrised with templates that represent the join points at which behaviours in other themes are crosscut. Theme/UML is really close of approaches like TranSAT presented in this paper. The main difference is on the target domain model. However, unlike TranSAT, Theme does not guarantee currently the result of the composition even if, a first work, called kerTheme [40], proposes to validate the composition result through testing.

Klein et al [41] defines an asymmetric operator that introduces the semantic-based weaving of scenarios. In this approach, an aspect is defined as a pair of scenarios, one scenario for the join point designation (the ”pointcut”), i.e., a scenario interpreted as a predicate over the semantics of MSCs [42] satisfied by all join points (specification of the behaviour to detect), and the second one for an advice representing the expected behaviour at the join point. Similarly to Aspect-J, where an aspectual behaviour can be inserted around, before or after a join point, with this approach, an advice may indifferently complete the matched behaviour, replace it with a new behaviour, or remove it entirely to create composed behaviour. The operator, proposed by Klein et al, is generic enough to be re-used to compose the behavioural part of an architectural aspect. It can be for example adapted to compose UML 2.0 sequence diagrams.

Less connected with UML, Roberto Lopez-Herrejon et al [43] proposed an approach based on algebraic foundations. Here aspects are seen as a model transformation function, or a function that maps models to models, and the effects of the weaving process can be understood in terms of algebraic transformations. Around this definition, theoretical properties (commutativity, associativity and identity) are assigned to aspect compositions, and rules are generated (in ex. precedence rules for compositions). This allows one to reason about composition, exposing its problems and leading to a partial solution for aspect reusability and problems that derive from the weaving process. The transformation language proposed in TranSAT to express the composition aims to the same goal.

The aim of the different approaches presented in this subsection is to safely compose models. In considering software architecture description as a model, several

ideas can be inspired by these works to guarantee the correctness of an evolution step in a software architecture description.

AOSD Evolution Paradox

Despite all the help provided by aspects in modularisation, AOSD paradoxically becomes a threat for software evolution, and consequently for software reliability. This problem is called AOSD-Evolution Paradox [34]. Managing the separation of concerns at the architecture level does not solve this issue.

Indeed, if we focus in long-term behaviour of a system and we depict what can happen when it evolves, we argue that aspects can apply in a badly way, resulting in a system that is inconsistent and exhibits an incorrect behaviour. This problem can be described as a consequence of the obliviousness property of aspects, which tries to make all aspects transparent for the base model. Indeed the aspects have to include a description of each place at which this crosscutting concern occurs and thus rely on the existing structure of the system. This leads to a tight coupling between the system and the aspects that advise it. When the system evolves, its structure changes and every crosscutting concern in every aspect needs to be checked to avoid an undesired behaviour, meaning that AOSD leads to software that is robust in modularisation, but contrary to what is expected it reduces the evolvability. A special case of this problem, called fragile pointcut[44], arises in the explicit designation of pointcut target location by naming corresponding elements to the base model. This pointcut becomes fragile when it unintentionally captures or misses particular join points, after seemingly safe modifications to the base model (tight coupling between aspects and the base model). Hence it makes the reuse difficult because non-local changes may break pointcut semantics. This problem introduces an overhead to the programmer and leads to a potential failure because the programmer has to make sure the pointcuts that he designates do not accidentally match another method.

In [45] a Model-based pointcut definition is proposed. These poincuts are defined in terms of a conceptual model of the base program, rather than referring directly to the implementation structure. This results in joint points based on conceptual properties instead of structural properties of the base program and thus leads to a low coupling of the pointcut definition and base model. These poincuts are called view-based pointcuts, because they use the formalism of intentional views to both express a conceptual model of a program and keep it synchronised with the source mode of that program. These model-based pointcuts are useful to avoid the AOSD Evolution paradox, and a promising approach to unveil the hard-link between aspects and base, allowing to reason about aspects in different dimensions. In TranSAT, the join point mask is based on the semantic of a Software architecture. Consequently, for example, when the join point mask defines two connected components. Every component directly or indirectly connected are matched. Furthermore, a component mask can match indiscriminately a component or a composite. This behaviour of the join point mask improve the architectural aspect genericity and limits the impact of a structural change on the weaving semantics.

1.7 Conclusion

Software architectures have the potential to provide a foundation for managing software evolution. However, if many ADLs support static description of a system, most of them currently provide no facilities for specifying architectural changes. In this chapter, we have identified two kinds of change: runtime architectural changes called internal software architecture evolution and changes managed by the architect called external evolution changes. For the first kind of evolution, two subcategories have been identified. The first one can express run-time modifications to architectures but requires that the modifications be specified explicitly. In contrast, other ADLs can accommodate unplanned modifications of an architecture and incorporate behaviour not anticipated by the original developers. These works propose to define architectural constraints to confine the potential evolution of the software architecture. On the other hand, to manage external evolution, ADLs suffer from the lack of support for modularity. This leads to a number of architectural breakdowns, such as increased maintenance overhead, reduced reuse capability, and architectural erosion over the lifetime of a system. As AOSD allows designer to modularise crosscutting concerns, promoting aspect-oriented software development principles into ADLs seems to be an attractive solution to overcome this external issue.

However, if applying AOSD to ADLs can help to overcome many of the problems related to software evolution, it pervasively modifies the semantics of the composition of software components. In the second part of this chapter, we argue that the integration of new concerns in a software architecture can break the software architecture consistency. Since a majority of existing ADLs have focused on design issues, they provide advanced static analysis and system generation mechanisms. These mechanisms must be adapted to manage the new composition paradigm between aspects and components. Through SafArchie and TranSAT, this chapter proposes an initial solution to statically check that an aspect will not break the software architecture consistency. TranSAT is based on a specific architecture transformation language to describe the weaving. This language is carefully designed to make certain unsafe transformations impossible to express. Besides, it allows static verification of additional coherence properties before aspect weaving or unweaving to be performed. However, TranSAT and its transformation language are currently highly coupled with the SafArchie semantics.

To conclude this chapter, we claim that one of the future main steps of software architecture is to propose (i) a way to describe homogeneously internal and external software evolutions. (ii) This evolution description should be associated to a powerful analysis model in order to be able to guarantee the software architecture consistency by checking only the parts of an architecture impacted by the changes. (iii) This approach should be generic in order to be adapted depending on the ADLs semantics. (iv) Any changes should be represented as first-class entities in the software architecture and it should, at least before system-deployment time, be possible to add, remove and modify a concern with a limited effort. The approaches presented in this chapter propose initial solutions to achieve these requirements. However, none addresses the evolution issue in its entirety in considering both the software evolution description, the analysis impact of a change and its projection on a targeted platform.

References

1. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26** (2000) 70–93
2. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15** (1972) 1053–1058
3. Tarr, P.L., Ossher, H., Harrison, W.H., Jr., S.M.S.: N degrees of separation: Multi-dimensional separation of concerns. In: International Conference on Software Engineering. (1999) 107–119
4. Vanderperren, W.: Combining Aspect-Oriented And Component-Based Software Engineering. PhD thesis, Faculty Of Science Department Of Computer Science System And Software Engineering Lab, Bruxelles, Belgium (2004)
5. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin., J.: Aspect-Oriented Programming. In Akcsit, M., Matsuoka, S., eds.: Proceedings ECOOP. Volume 1241., Springer-Verlag (1997) 220–242
6. Barais, O., Le Meur, A.F., Duchien, L., Lawall, J.: Safe integration of new concerns in a software architecture. In: ECBS, IEEE Computer Society (2006) 52–64
7. Magee, J.: Behavioral analysis of software architectures using ltsa. In: Proceedings of the 21st international conference on Software engineering, IEEE Computer Society Press (1999) 634–637
8. Allen, R.: A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon, School of Computer Science (1997) Issued as CMU Technical Report CMU-CS-97-144.
9. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering (ICSE-02), New York, ACM Press (2002) 187–197
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in java. In Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C., eds.: CBSE. Volume 3054 of Lecture Notes in Computer Science., Springer (2004) 7–22 *ISBN: 3-540-21998-6*.
11. Bures, T., Hnetynka, P., Plasil, F.: Sofa 2.0: Balancing advanced features in a hierarchical component model. In: SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications, Washington, DC, USA, IEEE Computer Society (2006) 40–48
12. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of xml-based architecture description languages. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, New York, NY, USA, ACM Press (2002) 266–276
13. Yan, H., Garlan, D., Schmerl, B., Aldrich, J., Kazman, R.: Discotect: A system for discovering architectures from running systems. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland (2004)
14. Barais, O., Duchien, L. In: SafArchie Studio: An ArgoUML extension to build Safe Architectures. Springer (2005) 85100 ISBN: 0387245898.
15. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour analysis of software architectures. In: WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), Deventer, The Netherlands, The Netherlands, Kluwer, B.V. (1999) 35–50
16. Taylor, R.N., Medvidovic, N., Anderson, K.M., Jr., E.J.W., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering **22** (1996) 390–406

17. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, New York, NY, USA, ACM Press (2004) 28–33
18. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal (1998)
19. David, P.C., Ledoux, T.: Safe dynamic reconfigurations of fractal architectures with fscript. In: Proceeding of Fractal CBSE Workshop, ECOOP'06, Nantes, France (2006)
20. SAE, A..E.C.S.C.: Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506 (2004)
21. Vestal, S.: Fixed-priority sensitivity analysis for linear compute time models. IEEE Transactions on Software Engineering **20** (1994)
22. OMG, O.M.G.: Unified Modeling Language: Superstructure. (2005) Version 2.0.
23. Tibermacine, C., Fleurquin, R., Sadou, S.: Preserving architectural choices throughout the component-based software development process. In: WICSA, IEEE Computer Society (2005) 121–130
24. OMG, O.M.G.: UML 2 Object Constraint Language Specification. (2006) Version 2.0.
25. Briand, L.C., Labiche, Y., Yan, H.D., Pent, M.D.: A controlled experiment on the impact of the object constraint language in uml-based development. icsm **00** (2004) 380–389
26. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE '98: Proceedings of the 20th international conference on Software engineering, Washington, DC, USA, IEEE Computer Society (1998) 177–186
27. Maier, M.W., Emery, D., Hilliard, R.: Ansi/ieee 1471 and systems engineering. Syst. Eng. **7** (2004) 257–270
28. Garcia, A., Chavez, C., Batista, T., Sant'Anna, C., Kulesza, U., Rashid, A., de Lucena, C.J.P.: On the modular representation of architectural aspects. In Gruhn, V., Oquendo, F., eds.: EWSA. Volume 4344 of Lecture Notes in Computer Science., Springer (2006) 82–97
29. Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L.: A model for developing component-based and aspect-oriented systems. In: Proceedings of the 5th International Symposium on Software Composition (SC'06). Volume 4089 of Lecture Notes in Computer Science., Vienna, Austria, Springer-Verlag (2006) 259–273
30. Perez, J., Navarro, E., Letelier, P., Ramos, I.: A modelling proposal for aspect-oriented software architectures. In: ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06), Washington, DC, USA, IEEE Computer Society (2006) 32–41
31. Batista, T., Chavez, C., Garcia, A., Rashid, A., Sant'Anna, C., Kulesza, U., Filho, F.C.: Reflections on architectural connection: seven issues on aspects and adls. In: EA '06: Proceedings of the 2006 international workshop on Early aspects at ICSE, New York, NY, USA, ACM Press (2006) 3–10
32. Quintero, C.E.C., Rodríguez, M.P.R., de la Fuente, P., Barrio-Solórzano, M.: Architectural aspects of architectural aspects. In Morrison, R., Oquendo, F., eds.: EWSA. Volume 3527 of Lecture Notes in Computer Science., Springer (2005) 247–262
33. Mens, T., Mens, K., Tourw'e, T.: Aspect-oriented software evolution. ERCIM News (2004) 36–37
34. Tourwé, T., Brichau, J., Gybels, K.: On the existence of the AOSD-evolution paradox. In Bergmans, L., Brichau, J., Tarr, P., Ernst, E., eds.: SPLAT: Software engineering Properties of Languages for Aspect Technologies. (2003)

35. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2** (1989) 219–246
36. van Glabbeek, R.: The linear time - branching time spectrum I. The semantics of concrete, sequential processes. In J.A. Bergstra, A.P..S.S., ed.: Handbook of Process Algebra, Elsevier (2001) 3–99
37. Baudry, B., Fleurey, F., France, R., Reddy, R.: Exploring the relationship between model composition and model transformation. In: 7th International Workshop on Aspect-Oriented Modeling (AOM 2005), MoDELS 2005, Montego Bay, Jamaica (2005)
38. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for composing aspect-oriented design class models. T. Aspect-Oriented Software Development **vol 3880** (2006) 75–105
39. Muller, A., Caron, O., Carré, B., Vanwormhoudt, G.: On some properties of parameterized model application. In Hartman, A., Kreische, D., eds.: ECMDA-FA. Volume 3748 of Lecture Notes in Computer Science., Springer (2005) 130–144
40. Jackson, A., Klein, J., Baudry, B., Clarke, S.: Testing aspect models. In: Model Driven Development and Model Driven Testing workshop at ECMDA. (2006)
41. Klein, J., Hélouêt, L., Jézéquel, J.M.: Semantic-based weaving of scenarios. In: AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2006) 27–38
42. ITU: Recommendation Z.120: Message Sequence Chart (MSC). Haugen (ed.), Geneva (1999)
43. Lopez-Herrejon, R.E., Batory, D.S., Lengauer, C.: A disciplined approach to aspect composition. In Hatcliff, J., Tip, F., eds.: PEPM, ACM (2006) 68–77
44. Koppen, C., Störzer, M.: PCDiff: Attacking the fragile pointcut problem. In Gybels, K., Hanenberg, S., Herrmann, S., Wloka, J., eds.: European Interactive Workshop on Aspects in Software (EIWAS). (2004)
45. Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the evolution of aspect-oriented software with model-based pointcuts. In Thomas, D., ed.: ECOOP. Volume 4067 of Lecture Notes in Computer Science., Springer (2006) 501–525