A Generic Model for Managing Software Architecture Evolution

Mourad OUSSALAH, Nassima SADOU, Dalila TAMZALIT
LINA-FRE CNRS 2729
Faculty of Sciences Nantes University
2,Rue de la Houssinière, BP 92208
44322 Nantes Cedex 03 France

Abstract: Evolution becomes an important concern of software architectures, as well at architectural level as at application one. In addition, such evolution can be rather static (at specification time) than dynamic (at execution time). To face this important problem of software-architecture evolution, it is necessary to consider the evolution in a generic and uniform way by : defining the same concepts to manage the evolution of any architectural elements at any level of abstraction and independently of the software architectures description or implementation language. Our work aims to reach these objectives through the proposed model, called SAEV(Software Architecture EVolution Model). SAEV offers evolution operations described by evolution strategies and evolution rules to manage the architectural elements evolution. These rules and strategies must respect all invariants defined on each architectural element to safeguard the architecture coherence across the evolution. SAEV proposes also an evolution mechanism, which describes the execution process of the evolution model.

*Keywords :*Software architecture, static evolution, dynamic evolution, evolution operation, evolution strategies.

1. Introduction

Software architectures takes a predominant importance within software engineering area for its characteristics of re-use, assembly and deployment of its entities. It improve the development and the evolution of large and complex systems. Software architecture offers a high abstraction level for the description of systems, by defining its architecture in terms of components describing the systems functionalities and the connectors which express the interactions among these components [19]. Several Architecture Description Languages (ADLs) are proposed to aid the architecture-based development, such as C2 [1], ACME [5], Darwin [9]. Most of their efforts focus on the systems specification, development and deployment, few works are devoted to their evolution problems. For the ADLs that approach this problematic, there proposals are even limited to some techniques such as subtyping, inheritance, composition [12].

We aim in this work, to enhance software evolution, by proposing an evolution model called SAEV (Software Architecture Evolution Model). We consider the software architecture through three abstraction levels namely, from the most abstract one: the meta level, the architectural one and the application level. SAEV aims to describe and manage the evolution of software architecture at these different levels in a uniform way: as well the evolution of architectures as the evolution of applications. For this, software architecture elements (like component, interface, connector and configuration) are considered as first-class entities and SAEV leans on its own concepts and evolution mechanism.

The remainder of this paper is organized as follows: section 2 describes the related work, section 3 presents our mains objectives and motivations; section 4 presents the minimal and consensual architectural elements of ADLs and their abstraction levels; section 5 describes the proposed evolution model. Section 6 describes the

application of SAEV to the different abstraction levels, before concluding and presenting our perspectives.

2. A state of the art

Evolution is considered a key aspect of architecture-based development, because design decisions at the architectural level have far reaching consequences on the resultant code [8]. In the research literature, we can distinguish two kinds of architectural evolution, static evolution and dynamic evolution.

2.1 Static evolution

Static evolution concerns with modifying the architecture of system at the time of its specification. Then, once the system is implemented, we must stop it to make modifications, then produce another executable system. This category of evolution is supported by operational mechanisms often inspired by those of object-oriented evolution and often influenced by the programming language which will implement the specification . We present in the following some of these mechanisms:

Instantiation: software architecture distinguishes between component and connector types, where component types are abstractions that encapsulate functionalities into reusable blocks, and connectors types are abstractions that encapsulate components communication. A component type can be instantiated multiple times in a single system. Regarding connectors, only ADLs that model connectors as first-class entities support their instantiations. ADLs such Darwin[10], Metah[20], and Rapide[9] don't support connectors instantiations, since they don't model them as first-class entities.

Inheritance and subtyping: they are two different ways of reusing models. Inheritance permits the reuse of a model itself; meanwhile subtyping supports the reuse of objects of a model. The inheritance improves reusability and evolution, allowing the replacement of components and connectors within the system by specialized versions, which maintain some of the properties of the original ones. Subtyping is where an object of one type may safely be substituted where another type was expected. In software architecture components and connectors are architectural types and they are distinguished from the basic types(e.g., integers, characters, array, etc.). We can see three different components subtyping relationships: interface subtyping, behavior subtyping and implementation subtyping [16]. For example, Interface subtyping (Int) requires that if a component Cp1 is an interface subtype of another component Cp2, then Cp2 must specify at the least the provided and most the required interface elements from Cp1.The ADL C2[2] supports multiple subtyping by offering a mechanism to select what parts of a component can be changed (behavior, interface, implementation) and what can't, using keys such as *and*, *or*, *not* . Whereas, ADL like ACME[5] supports strictly subtyping using its extends features. Unicon[18], Metah[20] for example define components by enumeration, so they don't provide any mechanism to evolve them. Only ADLs that don't model connectors as first-class entities do not provide a mechanism to inherit them, these include ADLs such Darwin[10], Metah[20], and Rapide[9]. Several ADLs such as ACME support connectors' inheritance using mechanism identical to components inheritance.

2.2 Dynamic evolution

Dynamic evolution is a new concern in ADLs, it is called also "active evolution" or "run-time evolution" in [15]. It means the possibility of introducing modifications in system during its execution. This is an important characteristic, as some critical systems can't be stopped, to evolve them. Dynamic changes of an architecture may be either planned at architecture specification time or unplanned [12]. In the first case the changes likely to occur during the execution of the system must be known initially, so they must be specified at the description of architecture. Rapide[9] supports conditional configuration, using its clause *where* which enables a form of architectural rewiring using the *link* and *unlink* operators[13]. Dynamic ACME associates with each element a multiplicity which can indicate the number of instances of this element or if it is defined, undefined, optional, etc. It associates also to the specification a clause *opened* or *closed*. The specification is considered *closed* if all the elements likely to be added are defined in the architecture specification. It is considered *opened* if the structure or elements specification are uncompleted.

The unplanned evolution places no restrictions at the architecture specification time on the kinds of allowed changes[12]. Thus, the ADL supporting this kind of evolution must offer architecture modification features, which allows the architect to specify changes during the system execution. C2[1] for example specifies a set of operations for insertion, removal and rewriting of elements in architecture at runtime. C2 offers also the ArchShell tool which enable interactive specification, execution and run-time-modification of C2-style architectures by dynamically loading and linking new architectural elements[15].

3. Motivations and main objectives

Developed systems evolve as well as their architectures. We may need, for instance to add new components, to modify the existing components or to modify the connections between these components. This evolution must be identified and managed to maintain the architecture coherence of the evaluated system. We propose SAEV as a solution to face this problematic. We are interested more precisely by the structural evolution of architectures. For that SAEV must :

- abstract the evolution, from the specific behaviors of architectural elements. That allows to:

- define mechanisms for the description and the management of the evolution independently of the architectural elements and their description languages;
  - support the re-use of these evolution mechanisms in several cases of evolution.
- be open to the addition of new evolutions, in particular those that are not envisaged initially by the model. It must be for that reflective and adaptive.
- Support static evolution (at the architecture specification time) as well as dynamic one (at the application execution time).

To achieve these objectives, SAEV must take into account all architectural elements proposed by ADLs. We present the most important elements in the following section.

4. Main architectural elements

4.1 Presentation of the architectural elements

We present hereafter the main architectural elements commonly supported by the majority of ADLs ([5], [6],[11], [19]). We present first their definitions (most accepted by the software architecture community), their Meta model, then we position them according to different abstraction levels.

Component represents the computational elements and data stores of a system. It is described by an interface and one or more implementations. **Connector** represents the interaction among components as well as the rules that control this interaction. It is mainly represented by an interface and one or more implementations. **Interface** is the only visible part of components and connectors. It provides the set of services (provided or required) and interaction points. The interaction points of component are called ports (provided or required port). Those of the connectors are called roles (provided or require roles). **Configuration** describes how components and connectors are fastened to each other. It is described also by an interface which provides a set of interaction points (provided and required ports) and a set of services. We distingue two kinds of links used to fasten configuration's elements the *attachments* and the *bindings*. The *attachements* express which ports of a given component connected to which roles of a connector (a provided port can be attached only to a required role and a required port can be attached only to a provided role). The *bindings* define links among: a port of a composed component and those of its subcomponents, a role of a composite connector and those of its subconnectors or among the ports of a configuration and those of its components These elements are represented by the following meta model described using the class diagram of UML[4].

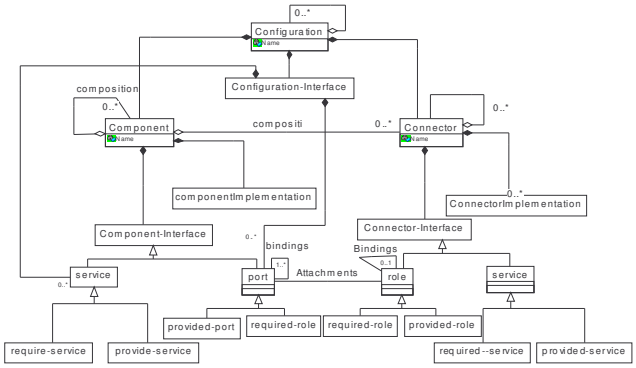

Figure 1. Architectural elements Meta model

Each architectural element is represented by a class and each element can be connected to other elements by an association of composition. Thus each element is characterized by an interface and its contents (elements which compose it).

4.2 Software Architecture abstraction levels

Some of the surveyed ADLs such as C2 [1], ACME [5] and Rapide [8] consider components and connectors as first-class entities and distinguish respectively the component-type and the connector-type from their component-instances and connector-instances. But, this distinction is not valid for the *configuration* which is often considered only at the application level as a graph of components-instances and connectors-instances. In our Work, we consider all architectural elements as first-class entities and with three abstraction levels: the Meta level, the Architectural level and the Application level.

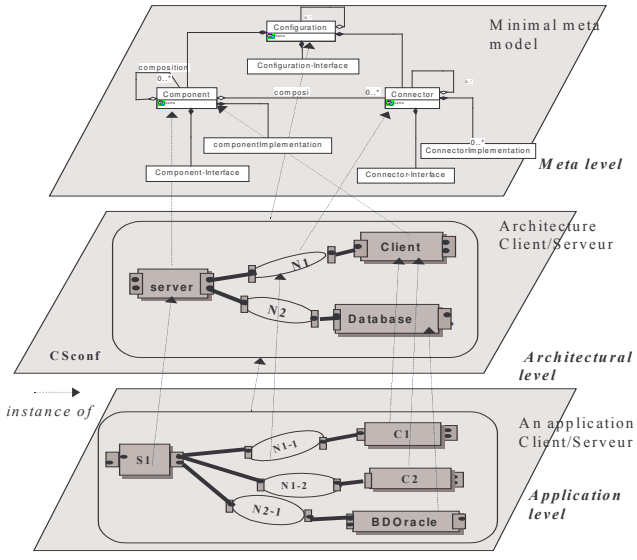

Figure 2. Architectural elements abstraction levels

Meta level: it is the level of definition of all ADLs architectural elements, like configuration, component, connector, interface, etc.

Architectural level: In this level a system architecture is describe using one or more instances of the architectural

elements of the meta level. The Figure 2, presents a Client/Server architecture with a *Configuration*: CSConf; three *components* : client, server, Database and of two *connectors* N1 and N2.

Application level: In this level we can define one of more applications in accordance with their architecture defined at level in the top. For example, from the preceding architecture client/server, we can build an application made up of: one instance of the configuration CSConf: Cf, two instances of the component client: C1, C2 and one instance of the component Database: DBoracle, one instance of the component server: S1; two instances of connector N1: N1-1, N1-2 and one instance of connector N2: N2.1.

5. SAEV: Software Architecture EVolution model

The evolution of software architecture is reflected through the different changes carried out on its elements. These changes can be, for example the addition of a component, the deletion of one of its components,etc. Each change may cause also impacts that should be managed to maintain the whole architecture in a coherent state.

Basing on these concerns and the previous objectives, SAEV offers a whole of concepts to describe and manage the software architecture evolution.

5.1 SAEV Meta model

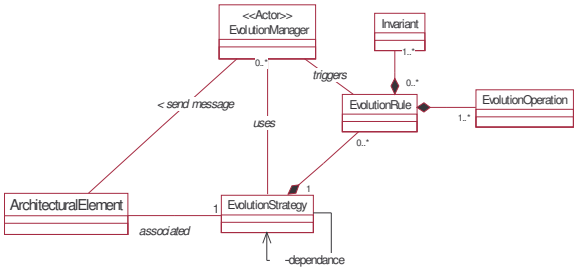

Figure 3. SAEV Meta –Model

We associate to each architectural element an evolution strategy. A strategy gathers the whole of evolution rules which describe the operations that can be applied to this architectural element. Thus, each evolution rule must respect the invariants defined on this architectural element. The evolution execution process is managed by the evolution manager.

We detail each concept in the following section. We illustrate only the evolution of the architectural level but the principle remains the same for the application level evolution.

5.2 SAEV concepts

a. Architectural element : represents any element of software architecture. It can be for example a configuration, a component, a connector, interface, etc.

b. Invariant : represents an architectural element constraint which must be respected throughout its life cycle. Any change in the architecture must maintain the correctness of this invariant. We present hereafter the invariants associated with the *configuration*:

A.Element	Invariants
Configuration	– a configuration must have an interface, by which it can delegate with other configurations or with its components. – a configuration must be composed at least of one component; – a connector must connect at least two components. – a component can not be related directly to an other component.

TABLE 1. EXAMPLES OF ARCHITECTURAL ELEMENTS INVARIANTS

c. Evolution operation : is an operation which can be applied to the architectural element or to its sub-elements and which cause its evolution. We have identified the following evolution operations: Addition, Deletion, Modification, Substitution. For example the evolution operations of configurations are :

- Addition / deletion / modification / substitution of a provided / required *port* or *service*.
- Addition / deletion / substitution of a *component* or *connector*.
- Modification of the name of a *component* or *connector*

d. Evolution Rule : describes the execution of an operation on a given architectural element. It expresses the necessary conditions to execute this operation as well as the rules to be triggered if necessary on the other architectural elements, to propagate the rule impacts.

The evolution rules are based on the ECA formalism (Event /Condition /Action). Thus each evolution rule is made of:

- an event: is the evolution invocation coming from the designer or from another rule. It is intercepted by the evolution manager;
- one or more conditions: that must be satisfied to execute the action part of the evolution rule.
- one or more actions, an action can be an:
  - event, in this case, it will be redirected toward another rule;
  - elementary action to be executed on the architectural element. We note them: *Architectural-element-name.Execute.operation-name (parameters);*

We give in the following example of evolution rule :

Event : delete-component(**Cf**: Config, **C**: comp);

Condition: **C** ∈ comp(**Cf**), provided-interface(**C**) connected to provided-interface(**Cf**), ∃ NC⊂ connect(Cf) and ∀ **N**∈ NC **N** is connected to **C** and **N** is not charred

Action:

For **N** ∈ NC delete-connector (**Cf**,N)

For b ∈ bindings(**Cf**,**C**) delete-binding(Cf,C,b)

For I∈ interface-comp(**C**) delete-interface-comp(Cf,C,I)

*C.Execute-delete-component(**Cf**, **C**)*

The rule **R1** describes the deletion of the component **C** belonging to the configuration **Cf**. This rule triggers firstly the deletion of connectors connected to **C**, then the deletion of bindings between the component **C** and configuration **Cf,** the deletion of the interface of **C** and finally the deletion of the component **C**.

The whole of the defined evolution rules is stored in a rules base. The designer will be able to re-use these rules or to create his own evolution rules.

e. Evolution strategy : we associate with each architectural element an evolution strategy. A strategy gathers the whole of the evolution rules which describe all the evolution operations that can be applied to this architectural element. The table 3 presents example of a strategy S1 associated with the configuration.

Strategies	A. Elements	Evolution operations	Evolution rules
		Addition	R1, R10
		Deletion	R2,R3
		Substitution	R6, R9
		Modification	R3,R4

f. Evolution manager: is an actor, representing the processing system of SAEV. Its role is intercepting the events emanating from the designer or the evolution rules towards an architectural element. Then it triggers the execution of the corresponding evolution rules, according to the evolution strategy associated with this architectural element. We detail this process in the following section.

5.3 SAEV Evolution mechanism

The evolution mechanism describes the execution process that must be followed to carry out a given evolution. We describe this process using the following UML sequence diagram [4].

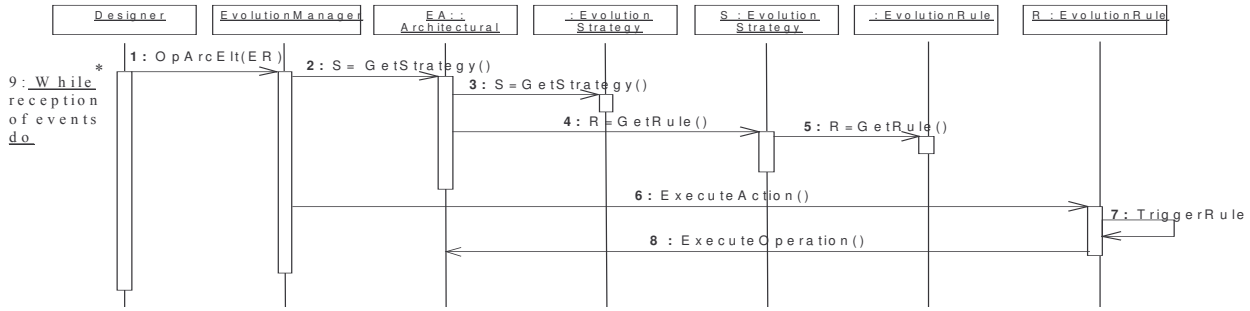

Figure 4. SAEV execution process

The evolution is triggered automatically by any event emitted by the designer towards an architectural element. The evolution manager :

- **1:** intercepts this event;
- **2:** and **3:** selects the evolution strategy associated with the architectural element invoked by the event ;
- **4**: and **5:** selects in this strategy the evolution rule that correspond to the event and which have a satisfied conditions;
- **6:** triggers the execution of the action part of the selected rule. Two cases can arise:
  - **7:** if the action corresponds to an event, the manager intercepts it also and follows the preceding steps (**1:** to **6:**).
  - **8:** if it corresponds to an elementary action, it triggers then its execution.
- **9:** These steps are renewed as much as the manager intercepts new events;

6. SAEV and the abstraction levels

The model SAEV must be able to describe and manage the evolution of software architecture at the architectural level as well as at the application level. Thus, it can be positioned at the meta level to manage the evolution of the architectural level and it can be positioned at the architectural level to manage the evolution of the application level. This is illustrated by the following figure:

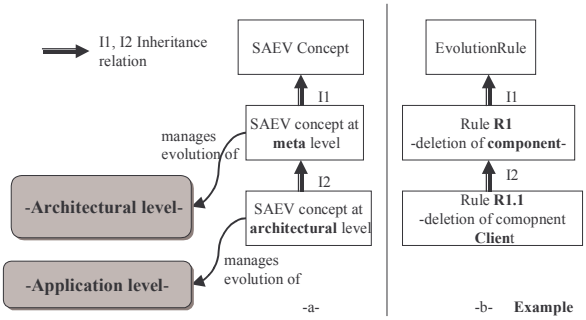

Figure 5. SAEV and the abstraction levels

At the meta level, SAEV can extend its concepts (relation I1) to manage the evolution of any architecture described with any ADL. It proposes, for that the same evolution rules, invariants, evolution strategies. For example, SAEV proposes the evolution rule **R1**, which describes the deletion of any **Component** of any architecture. At the architectural level, SAEV allows the designer to extend the concepts of SAEV at the meta level (I2 relation) by considering the elements of its own architecture. For example, for the client/server architecture (figure 5-b), the designer can extend the evolution rule R1 proposed at the meta level, with the *rule R1.1* which describes the deletion

of the specific component **C*lient***. In addition of the deletion of component, the rule *R1.1* specifies what is necessary to delete a component **Client** from an architecture client/sever.

7. Conclusion

We have proposed in this article SAEV a model for software architecture evolution independently of their description languages. SAEV offers a set of concepts to describe and manage the evolution of a given architecture. We worked on the most common architectural elements of ADLs but it can be applied to any other architectural element of any ADL. SAEV associates an evolution strategy to each architectural element. A strategy gathers evolution rules describing all evolution operations of this architectural element.

SAEV answers objectives, that we fixed (section 3): the evolution is described independently of the architectural elements behavior; the evolution mechanisms (evolution strategies, evolution rules, invariants) are the same to manage the evolution of any architectural element at the architectural level as well as at the application level. We have chosen to implement our model with UML 2.0[14] which is a good standard for software development. This implementation will offers to UML2.0 an architecture evolution support. It will also allow SAEV to be reflexive, so it can be used to evolve its own architecture.

We aim to study the influence of the architecture evolution on the application level and vice versa, next we will study the application of the evolution model on the meta level. This later perspective will allow the evolution of a given ADL, for example by adding, or redefining of new concepts.

References

[1] R. Allen, R. Douence, D.Garlan: Specifying and analyzing Dynamic Software Architectures, In proceeding of the Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal Mars 1994.

[2] L F. Andrade, J.L. Fiadeiro: Architecture based evolution of software systems, LNCS 2804 : pp 148-181, 2003.

[3] T. Batista, A. Joolia, G. Coulson: Managing Dynamic Reconfiguration in Component based systems, in Europeen workchop on software architecture, Pisa, Italy, June 2005.

[4] G. Booch, J. Rumbaugh, and I. Jacobson: The Unified Modeling Language User Guide, Addison-Wesley Professional, Reading, Massachusetts, 1998.

[5] D. Garlan, R. Monroe, D. Wile: ACME: Architectural Description Of Components-based systems, Leavens Gary and Sitaraman Murali, Foundations of component-Based Systems, Cambridge University Press, 2002,pp. 47-68.

[6] D. Garlan, D. Perry: introduction to the special issue on software architecture, IEEE Transactions on Software Engineering, 21(4), April 1995.

[7] D. Garlan, R. Allen, and J. Ockerbloom: Architectural mismatch: Why reuse is so hard. IEEE software, November 1995;

[8] C. B. Jakman. A. Maintenance check for evolving a product-line architecture by determining the indicators of erosion. In proceeding of workshop on Empriccal Studies of Software maintenance (WESS98), Bethesda, Maryland, November 1998

[9] D. Luckham, M. Augustin, J. Kenny, J. Vera, D. Bryan, W. Mann: Specification and analysis of System Architectures using Rapide", IEEE Transaction on Software Engineering, vol.21, N° 4, April 1995,336-355.

[10] J. Magee, N Dulay, S. Eisenbach, J. Kramer : Specifying Distributed Software Architectures, In Proceedings of the fifth European Software Engineering conference

[11] N. Medvidovic, D.S.Rosenblum, and R.N. Taylor: A Language and Environment for Architecture-based Software Development and evolution. In proceeding of the 21st international conference en Software engineering , pp.44-53, may 1999.

[12] N. Medvidovic, D.S.Rosenblum: A Domains of Concern in Software Architectures and Architecture Description Languages. In Proceedings of the 1997 USENIX Conference on Domain-Specific Languages, October 15-17, Santa Barbara, California.

[13] N. Medvidovic, R. N. Taylor : A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, Vol. 26, 2000.

[14] OMG, UML 2.0 infrastructure specification, Technical Report ptc/03-09-15,Object Management Group (2003).

[15] P. Oreizy: Issues in the runtime modification of software architectures. Technical Report, UCI-ICS 96-35, University of California, Irvine, August 1996.

[16] D.E. Perry, A.L. Wolf : Foundations for study of software Architecture, In ACM/SIGSOFT Software Engineering Notes, volume 17, pages 40-52, October 1992

[17] R. Roshandel, A.V.D. Hoek, M. Miki-Rakic, N. Medvidovic : Mae-A System Model and Environment for Managing Architectural Evolution : ACM Transactions on Software Engineering and Methodology, April 2004.

[18] M. Shaw, R. Deline, D. V. Klein, T.L. Ross, D. M. Young and G. Zelesnik: Abstractions for Software architecture transactions on Software Engineering, page 314-335, April 95.

[19] A.Smeda, M. Oussalah, T. Khamaci : A Multi-Paradigm Approach to Describe Complex Software System", WSEAS Transactions on Computers, Issue 4, Volume, 3, Department of Computer Engineering, Mälardalen University, September 19th 2003.

[20] S. Vestal: Programmer's Manual, version 1.09. Technical report, Honeywell technology center. April 1996