Evolving and Versioning Software Architectures Using ATL Transformations

Abdelkrim Amirat[1], Afrah Djeddar[1] and Mourad Oussalah[2]

[1]LiM Laboratory, University Mohamed Cherif Messaadia, Souk-Ahras Algeria
[2]LINA Laboratory, CNRS UMR 6241 University of Nantes, France

Abstract: *Since software architecture has become integral part of software development, managing its evolution has become the concern of most of architecture researchers. In this paper, we define firstly a Generic-ADL (Architecture Description Language) which includes all important and common concepts in the existing ADLs to describe software architectures. Secondly, we propose a second model named EVA-Model (Evolution and Versioning Architecture) to manage the software architecture evolution and their versioning. Based-on the proposed EVA-Model, we implement the evolution and the versioning mechanisms using model transformation approach through ATL language. However, these ATL transformations present tow challenges: the absence of the genericity concept and the rule scheduling mechanism. We address these issues by proposing parameter model to generalize the transformations and by using java technology to allow users managing the parameters and to handle the execution order of evolution transformations where each evolution transformation is followed transparently by a versioning one.*

Keywords: *Architecture Evolution, Versioning, ATL Transformation, Parameterized Transformation, EMF/GMF.*

1. Introduction

Independently of any programming language and execution environment it appeared several languages (e.g. SafArchi [1], DEDAL [2], etc.) for modeling software at a high level of abstraction called ADL. These later allow a designer to describe the architecture of any software. Show and Garland [3] defined *Software Architecture* as a level of design that involves: the description of elements from which the system is build, interaction among those elements and the pattern that guide their composition and construction. These software architectures may be subject to various changes in its structure or/and one of their constituents, here we talk about their evolution. Tracking this evolution is defined as the backup link between the architecture before and after the evolution where each track has its corresponding version. After the analysis of a set of architectural description languages we have found that the traced evolution, in other words versioning the architectural evolutions, is rarely taken into account in the design of ADLs. This research work address this issue by proposing a generic model (i.e. *EVA-Model*) independent from any definition of existing ADLs which deals with traced evolution in a high level of abstraction. In order to show their applicability, we define a generic architecture description language that groups all common concepts between known ADLs. This *generic-ADL* will be associated with our proposed *EVA-Model* to manage the evolution and the versioning of architectures.

The evolution and the versioning mechanisms in the proposed approach are considered as model transformation operations implemented using the transformation language ATL (*Atlas Transformation Language*) [4]. This transformation language presents two challenges that prevent us to implement our generic *EVA-Model*: the absence of the genericity concept and the absence of rule scheduling mechanism. In this work we address these two issues by answering to these two questions: How to parameterize the transformations and making them generic? How to maintain the rule scheduling in the case of a transformation that requires the execution of a set of rules in a specific order?

Parameterizing a transformation means that the latter proceeds according to the criteria given by the user (i.e. the elements meant to be evolved will be chosen separately from the programming of ATL rules). This will render transformations rules more generic. To this end, we write the evolution and the versioning rules once and for all of them we define a *parameter model* which serves as a second entry in our evolution transformation rules in order to support the various elements that can be subject of the evolution.

In order to accomplish a given evolution transformation that requires the execution of a several successive rules we propose to use the java technology to manipulate the parameterized transformation rules and to define their execution order. In this paper we present two kind of the versioning of the evolution transformation: the versioning of the architecture itself that is performed with java and the versioning of the evolved architectural elements that is applied directly and transparently after the execution of each evolution transformation rule by triggering a versioning one.

The rest of paper is organized as follow: in Section 2 we emphasize on some related works to our approach. In Section 3 we present the basic concepts of software architecture. The proposed approach is explained in Section 4. In Section 5 we present the implementation of approach. Finally, we conclude this work by presenting some conclusions and possible guidelines for future work.

2. Related works

This section discusses existing approaches that concentrate on software architecture evolution, its versioning and ADL that support Architecture evolution Barais et *al.* did a comparison of ADL evaluating them according to their abilities of managing software architecture evolution [5]. The studies result of SafArchi [1], C2 [6], ArchStudio [7], and AcmeStudio [8] showed that these ADL support the definition of a static software architecture and in all these language or associated tools; the evolution has not been taken into account. These languages can't describe the dynamic of the system and do not take care of external evolutions. Wright [9] Fscript [10] make dynamic architectures explicit, they currently do not describe the dynamic with the same goal.

DEDAL ADL [2] represents explicitly three levels abstraction in the definition of architecture. The specification, configuration and assembly architecture and keeps track of decisions of architects in a process of development (*i.e. forward engineering*). It can be used to support a process of controlled evolution (*i.e. reverse engineering*) also supports element versioning. SOFA [11] introduces the notion of node that supports mechanism for managing multiple version of the same component. The system version is very interesting in monitoring the evolution of software architecture.

Using Graph Transformation, Amirat and Menasria proposed C3Evol [12] that is an extension of C3 [13]. This framework was implemented using AToM^3 [14] exploring all its possibilities, to define their proposed metamodel and all variants of graph grammars implementing evolution operators. Mens and Tamzalit [15] formalized the ADL UML2 using the theory of graph transformation. This allowed specifying the structure and behavior of an architecture, to impose architectural styles constraining the architecture and to specify and execute typical architectural evolution scenarios demonstrating their approach with book store case study.

SAEV [16] manage the evolution using evolution rules and propagation strategies, but do not focus on versioning, it offers a set of concept to manage and describe an evolution of chosen architecture independently of ADL and architecture element behavior.

3. Basic Concepts

Before heading to our proposed approach it is necessary to focus on the basic concepts needed to know throughout this paper.

Versioned software architectures are a combination of the concepts of software architecture and versioning. The versioned software architectures are closely similar to the software architecture, also supports the modeling of components, interactions between them and their behavior, but they modelize in addition versions of the existing architectural elements [17].

Metamodeling: is the construction of a collection of concepts (things, terms, relation, etc.) within a certain domain. A model is an abstraction of phenomena in the real world; a metamodel is yet another abstraction, highlighting properties of the model itself. Each model is conforming to its metamodel in the way that a computer program is conforming to the grammar of the programming language in which it is written.

A model transformation, in Model-Driven Engineering, is an automatable way of ensuring that a family of models is consistent, in a precise sense which the software engineer can define. The aim of using a model transformation is to save effort and reduce errors by automating the building and modification of models where possible.

ATL is a model transformation language specified both as a metamodel and as a textual concrete syntax. It is a hybrid of declarative and imperative. The preferred style of transformation writing is declarative, which means simple mappings can be expressed simply. However, imperative constructs are provided so that some mappings too complex to be declaratively handled can still be specified.

ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. In the scope of the ATL language, the generation of target model elements is achieved through the specification of transformation rules. ATL defines two different kinds of transformation rules: the matched and the called rules. A matched rule enables to match some of the model elements of a source model, and to generate from them a number of distinct target model elements.

In the constitutive block of ATL rules we mention the helpers ATL which can be considered equivalent to methods. They make it possible to define factorized ATL code that can be called from different points of an ATL transformation [18].

4. Proposed Approach

The main interest of our approach is to present a generic model at the architectural level for treating the evolution and the versioning of software architectures. Every software architecture may be evolved over time, it undergoes to different changes in its structure or on any architectural element among their constituents. The proposed generic *EVA-Model* acts as a guide to manage software architecture evolution and implicitly their versioning (i.e. architectural element versioning) through the construction of propagation strategies where each evolution is directly followed by a trace backup in form of version (i.e. architecture versioning). The proposed generic model aims to keep all the modifications performed on all architectural elements throughout their life cycle. The overview of the *EVA-Model* is presented in the Figure1.

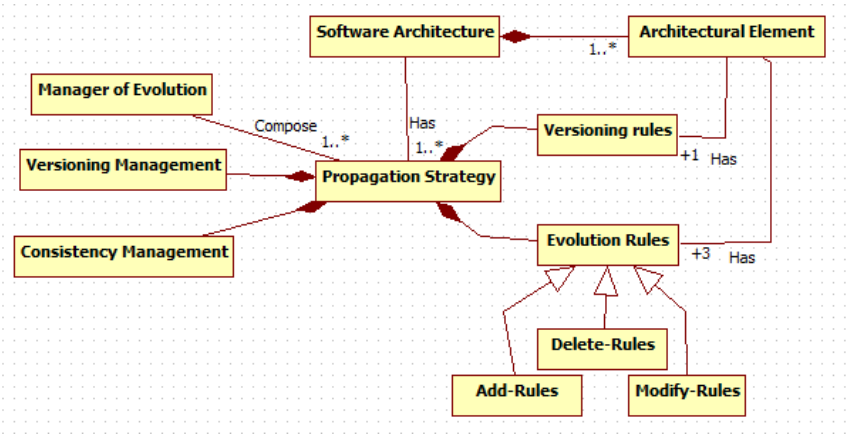

Figure 1- *EVA-Model* overview.

The traced evolution mechanism is implemented as a set of ATL transformation rules that will be applied on the software architecture meant to be evolved. According to *EVA-Model*, each *propagation strategy* represents a traced evolution transformation of the software architecture (i.e. the transition from software architecture version N to software architecture version N+1). This propagation strategy is composed of a set of evolution *rules* and *versioning rules* where each architectural element has three evolution rules modeling *Add*, *Delete*, or *Modify* operations and one versioning rule.

However, *evolution manager* (i.e. the person how will change the architecture) will take the responsibility to produce theses propagation strategies by choosing the set of evolution rules that will be applied on its architecture according to a specific order. Regarding the versioning rules, *versioning management* will trigger this set of rules after each the execution of an evolution rule in order to produce the versioning the evolved architectural element. Thus, it plays the role of the versioning of the target architecture after the execution of each rule (i.e. sub-versioning of the architecture).

Consistency management is specialized in the verification of the consistency of the produced architecture after the execution of each evolution operation in order to ensure the correct functioning of the evolved architecture.

To describe the *software architecture*, subject of the evolution process, we define a generic description language that regroups all common elements defined in large set of ADLs. As illustrated in the Figure 2. The metamodel of this *generic-ADL* supports the description of the five principal architectural elements: *Component*, *Connector*, *Configuration*, *Component and configuration Interface* (i.e. Ports), and *Connector Interface* (i.e. Roles). Also this metamodel presents the different relations among these elements: *Binding* that represents the relation between a component and its configuration through theirs interfaces (i.e. Port/Port relation), and *Attachment* that represents the relation between component and connector through theirs Interfaces (i.e. Port/Role relation). The metamodel is defined with EMF (*Eclipse Modeling Language*) [19]. It is implemented following all steps defined in GMF [20] (*Graphical Modeling Language*). The result is the description pallet representing all elements and relations used to describe any software architecture model. This described model is considered as instance of the generic metamodel.

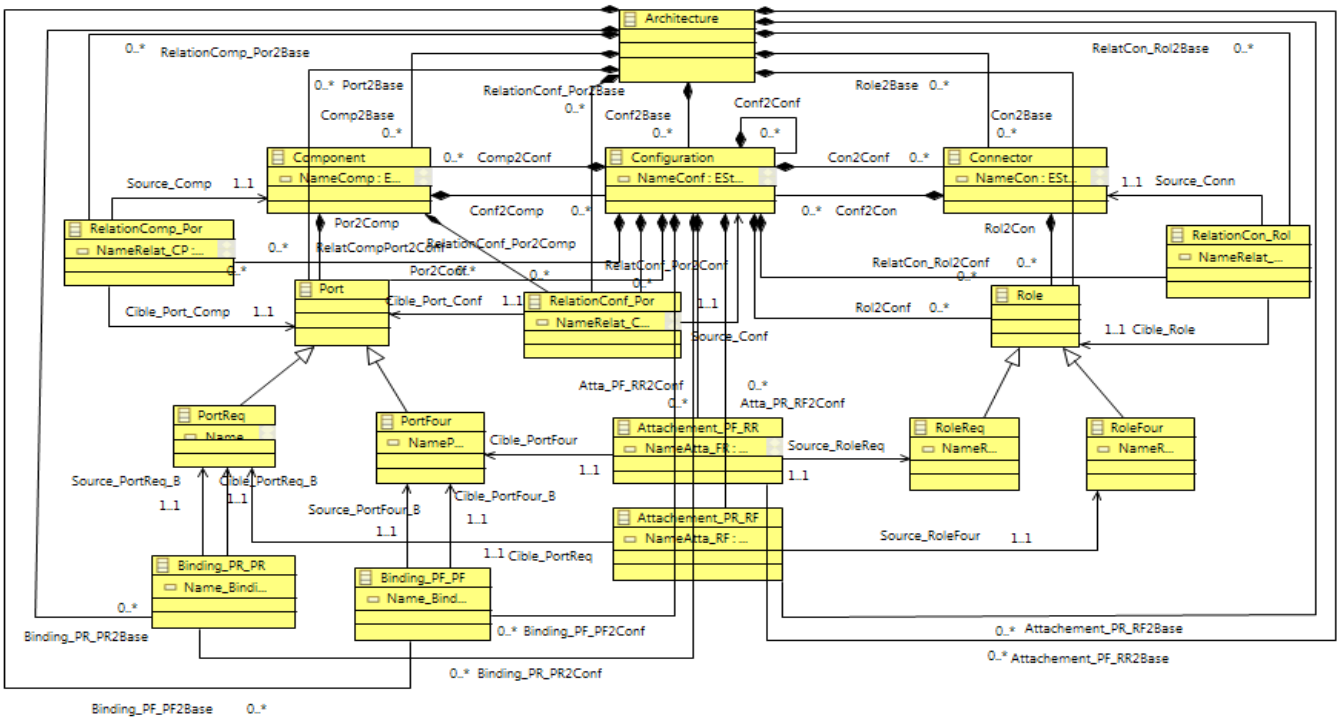

Figure 2- Generic metamodel for architecture description.

In order to make the mechanism of the evolution and the versioning generic and due to fact that ATL does not support the parameterization of transformation rules we have defined a *Parameter metamodel* as illustrated in Figure 3. Every architectural element presented in the *Generic-ADL* has its corresponding parameter class where theirs attributes support the names of the architectural elements meant to be evolved except for the *Number-Version* class that is used to versioning the architectural elements. The *ID attribute* carries the actual version value of the architectural element. The values of these different attributes are given by the manager of evolution (i.e. evolved architectural elements names) or manipulated via java (i.e. version value).

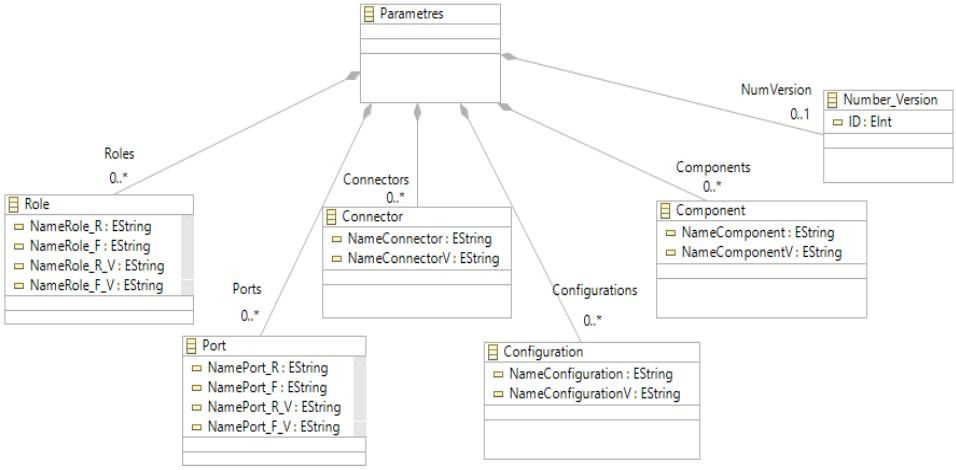

Figure 3- Parameter Metamodel**.**

However, the evolution and the versioning rules are written once for all where in each of them we defined a set of helpers that serve to extract the values of the attribute presented in the *parameter model* (i.e. the name of the architectural element on which we will apply the evolution rule or the version value which the versioning rule will use to do versioning the architectural element) or to perform various test (e.g. the test if the component that we will add is not already exist).

The Listing 1 presents an example of *component versioning rule* which contains three helpers, the first aims to verify if the component already exists in the architecture and the second one aims to extract the name of this component from the *parameter model* and put it in a variable

that will be used in the rename rule for versioning the component, and the third helpers aims to extract the actual version value from the *parameter model* that will be assigned to the component.

```
-- @path modele=/Evolution/MetaModel/GenericMetaModel.ecore
-- @path ModeleParam=/Evolution/MetaModel/ParametreMetaModel.ecore

Module CreatVersionComponent;
create OUT : modelerefining  IN : modele  , Parametre: ModeleParam;
helper context modele!Componentdef : ModifyComp : Boolean = let x :String =
ModeleParam!Component.allInstances()->collect(p| p.NameComponent)->first() inif(self.NameComp= x)
then true else false endif;
helperdef : NameComponent: String =ModeleParam!Component.allInstances()->collect(p|
p.NameComponent)->first();
helperdef : NumberVersion: Integer =ModeleParam!Number_Version.allInstances()->collect(p| p.ID)-
>first();

ruleRennomer {
        from E :modele!Component(E.ModifyComp)
        to T:modele!Component (NameComp<-thisModule.NameComponent + 'V' + thisModule.NumberVersion)
             }
```

Listing 1- Component Versioning Rule.atl.

Thus, the necessity behind the definition of the *parameter model* and write the ATL rules a more generic way is to answer the limit of ATL transformation mechanism: the lack of genericity in order to be able to realize parameterized transformations.

Another problem that can prevent us to implement correctly our generic *EVA-Model* is the absence of scheduling mechanism because our traced evolution approach needs implicit and explicit trigger of our evolution and versioning rules in according to a specific order. Implicit triggering reflects the automatic execution of the versioning rules by the versioning management in contrary of explicit triggering that represents the execution of the evolution rules by the manager of evolution. For this, once the ATL rules are written, we convert them to a java code in order to exploit them via an interrogation menu implemented in java in order to realize our generic traced evolution mechanism. From the generated java code of each transformation rule, we need only to manipulate the code part shown in the Listing 2 to invoke the execution of the corresponding evolution or versioning rule.

```
AddComponent runner = newAddComponent();
runner.loadModels(Source1, Source2);
runner.doAddComponent(newNullProgressMonitor());
runner.saveModels(Target);
```

Listing 2- Java code of *Add-Component* ATL rule.

In this paper, we use the transformations of type *N to 1* [17] because all our evolution and versioning rules need two source models (i.e. *Parameter Model* and *source software architecture*) in order to be executed and to generate the target model.

In our approach we presents to kind of versioning: the first kind addresses the architectural element and the second one concerns the versioning of the architecture itself. The execution of any evolution rule triggered by the manager of evolution leads to give the target architecture assigned with the sub-version Vn-m. Our mechanism aims to generate the architecture version Vn+1 when the execution of all the evolution rules of the same propagationstrategy is completed. Unlike to the versioning of the architectural element that is performed with the collaboration of java method and versioning ATL rules. Firstly, the java method extracts the least version given to the architectural element that is saved in a *versioning table* in order to increment it and after it reinserts again the incremented version in the *versioning table* and thus in*parameter model*. Now, it comes the role of the versioning rules that aim to extract the value (i.e. the last version) assigned to the ID attribute from the *parameter model*and affects it to the architectural element meant to be versioned.

5. Operative Mechanism

In order to explain the workflow of our proposed generic *EVA-Model* we present the following scenario: We assume that we have an architecture that carries the version N as presented in the Figure 4. This architecture model is instantiated from our generic metamodel defined previously.

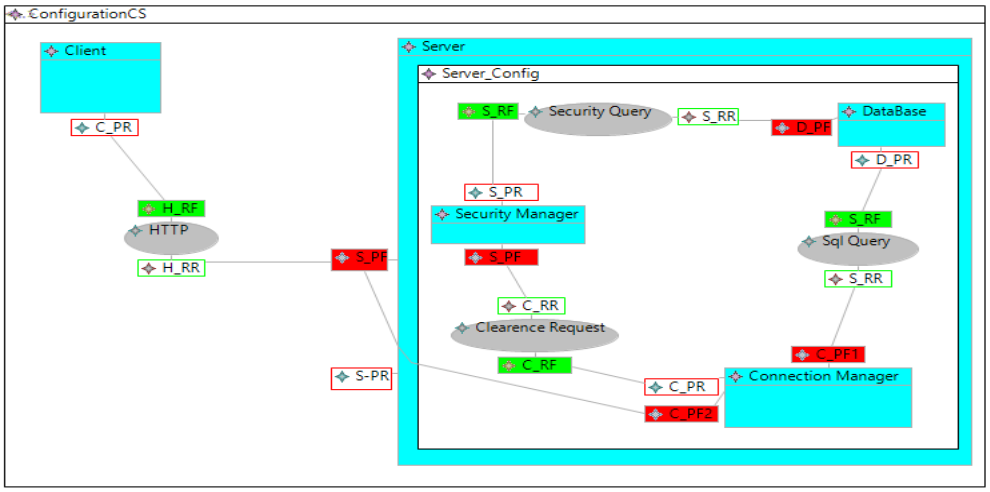

Figure 4- the software architecture Version N.

If the *manager of the evolution*launches the evolution rule *Add-Component* (Client 2) the *versioning management* triggers automatically the versioning rule *Configuration-Versioning*(ConfigurationCS) where the component was added (i.e. versioning of the configuration *ConfigurationCS-1*). After, it versions the target architecture (i.e. sub-versioningof the architecture *Vn.1*). Butbefore the *versioning management*performs its task it comes the role of the *consistency management*in order to verify the consistency of the generated architecture. Here, it launches the evolution rule *Add-Port* to the added component for satisfy the constraint: "Each component must have at least one port". Another rule will be executed that is *Modify-Port*to assign the component with a specific port (i.e. modify the default name of the added port). Now, after the *versioning management*performed its role the *consistency management* asks from the*manager of evolution*for add a connector in order to satisfy the constraint: "each component must be connected at least with other component".

If the user will connect the added component with another component that already exist it will launch the *Add-Connector* rule here all the steps explained aboveto add the componentwill be repeatedin order to add this connector. In the case where the manager of evolution indicates that the evolution is terminated, the *versioning management* generates the final version of the architecture V_{n+1}.

It should be noted that the *versioning management*and the *consistency management*work by faction. Before the *versioning management*does its workit just waits if the *consistency management*has evolution rules to execute as illustrated the Figure 5.

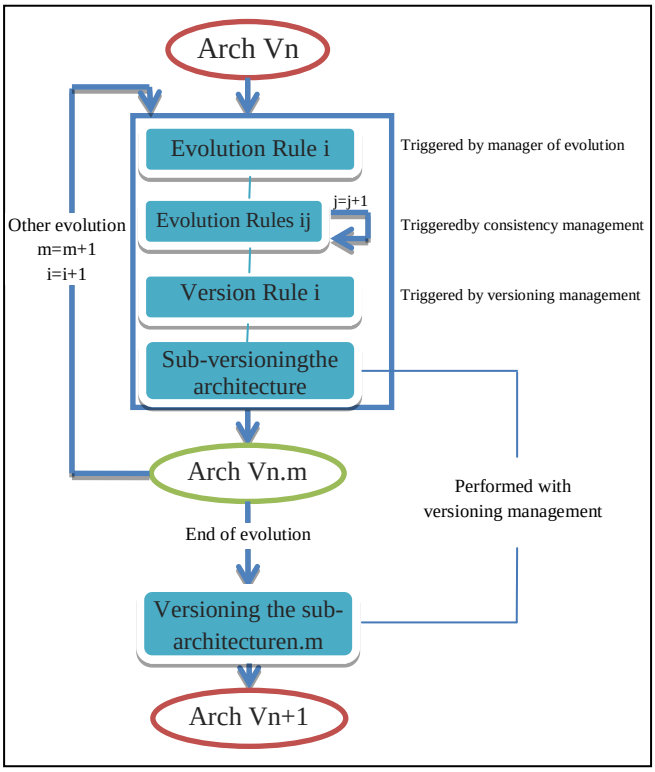

Figure 5- Execution sequence of the traced evolution mechanism.

The evolution strategy that represents the set of evolutions rules triggered by the manager of evolution and the versioning one that are triggered by *management versioning* is composed automatically as an interrogation via a java menu. The different required parameters to execute the evolution and versioning rules are eventually assigned to the different attributes presented in *parameter model* as indicated in Figure 6. The evolution strategy corresponding to the scenario presented above is illustrated as following:

- ⇨ R1 :Add component (Client2, ConfigurationCS)
  - ○ //enter the name of the new component?
  - ○ // Where do you need to add it?
- ⇨ R2 :AddPortF(PF, Client2)// add port with a default name PF
- ⇨ R3 :Modify-Port (C2-PF) // Modify Port Name
- ⇨ R4 :Versioningr-Configuration (ConfigurationCS)
  - ○ //As a result « ConfigurationCS 1»
  - ○ //Versionningthe architecture Vn.1
- ⇨ R5 :Addconnector(RPC, ConfigurationCS , CC-PR, CC-PF)
- ⇨ R6:AddRolR(RR, CC-PF)
- ⇨ R7 :ModifyRol (R-RR)
- ⇨ R8 :AddRolF(RF,CC-PR)
- ⇨ R9 :Modify Rol(R-RF)
- ⇨ R10: Versionner-Configuration (ConfigurationCS)
  - ○ //Result « ConfigurationCS2»
  - ○ //Versioning the architecture Vn.2
  - ○ //indicate the final of the evolution
  - ○ //Versionning the architecture Vn+1.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<xmi:XMIxmlns:xmi="http://www.omg.org/XMI" xmlns="Parametres" xmi:version="2.0">
<Parametres>
<NumVersionID= "10" />
<ComponentsNameComponent="Client2" NameComponentV=""/>
<ConfigurationsNameConfiguration="ConfigurationCS" NameConfigurationV=""/>
<ConnectorsNameConnector="HTTP" NameConnectorV=" "/>
<PortsNamePort_F="C-PF "NamePort_F_V=" " NamePort_R="C-PR" NamePort_R_V=""/>
<RolesNameRole_F="H-RF" NameRole_F_V="" NameRole_R="HT-RR" NameRole_R_V=""/>
</Parametres></xmi:XMI>
```

Figure 6- Parameter Model.

.

The source architecture Vn passes with several immediate version (i.e. Vn.1 when the adding of the component and the version Vn.2 when the adding of the connector) in order to have the final oneVn+1. The result of the evolution scenario presented above is represented in the Figure 7.

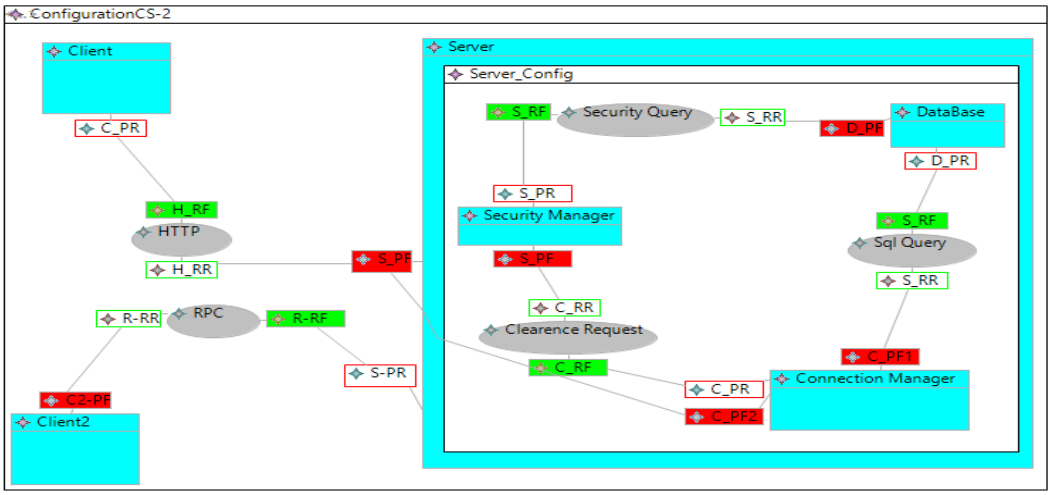

Figure 7- The versioned architecture N+1.

.

6. Conclusion

In this paper, we presented our contribution for resolution the traced evolution of software architecture's issue at architectural level. Therefore, we have proposed a generic model for managing software architecture's evolution and versioning called *EVA-Model.* To well illustrate the applicability of this proposed generic model we have presented a *generic-ADL*for describe the software architectures that are the evolution object of our *EVA-Model.*We have implemented the traced evolution mechanism as ATL transformations using parameterized ATL rules manipulated via java menu while ensuring the architecture's consistency. Thus, we have presented two kinds of versioning the first one is carried on the architectural element and the second one on the architecture itself. As future work, we plan to address the conflict problems caused by the presence of multiple versions and managing the traced evolution via a graphic interface and no via java menu.

References

[1] Barais, O. and Duchien, L., "SafArchie studio: Argouml extensions to build safe architectures", In *Architecture Description Languages*, pp. 85-100, SpringerUS, 2005.

[2] Zhang, H. Y., Urtado, C., and Vauttier, S., "Dedal: un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants", Proceeding *Australian Software Engineering Conference*, pp 246-255, 2010.

[3] Shaw, M. and Garlan, D. *Software architecture: perspectives on an emerging discipline*, Vol. 1, p. 12, Englewood Cliffs: Prentice Hall. Inc., Upper Saddle River, NJ, USA, 1996.

[4] http://www.eclipse.org/atl/.

[5] Barais, O., Le Meur, A. F., Duchien, L., and Lawall, J., "Software architecture evolution", In *Software Evolution*, pp. 233-262, Springer Berlin Heidelberg, 2008.

[6] Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. "Using object-oriented typing to support architectural design in the C2 style", In *ACM SIGSOFT Software Engineering Notes*, Vol. 21, No. 6, pp. 24-32, October 1996.

[7] Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., and Taylor, R., "Archstudio 4: An architecture-based meta-modeling environment", In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pp. 67-68, IEEE Computer Society, May 2007.

[8] Schmerl, B. and Garlan, D., "AcmeStudio: Supporting style-centered architecture development", In *Software Engineering, ICSE 2004*, IEEE *Proceedings 26th International Conference on*, pp. 704-705, May 2004.

[9] Allen, R., Douence, R., and Garlan, D., "Specifying dynamism in software architectures", in *Proceeding of foundations of Component-Based Systems Workshop*, Set. 1997.

[10] David, P. C. and Ledoux, T., "Safe dynamic reconfigurations of fractal architectures with fscript", In *Proceeding of Fractal CBSE Workshop, ECOOP*, Vol. 6, July 2006.

[11] Bures, T., Hnetynka, P., and Plasil, F., "Sofa 2.0: Balancing advanced features in a hierarchical component model", In IEEE *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pp. 40-48, August 2006.

[12] Amirat, A., Menasria, A., and Gasmallah, N., "Evolution Framework for Software Architecture Using Graph Transformation Approach", In *Proceeding of International Arab Conference on Information Technology (ACIT)*, Riyadh, Saudi Arabia, 2011.

[13] Amirat, A. and Oussalah, M., "First-class connectors to support systematic construction of hierarchical software architecture", *Journal of Object Technology*, *8*(7), pp. 107-130, 2009.

[14] De Lara, J. and Vangheluwe, H., "AToM3: A Tool for Multi-formalism and Meta-modelling", In *Fundamental approaches to software engineering*, pp. 174-188, Springer Berlin Heidelberg, 2002.

[15] Tamzalit, D. and Mens, T., "Using graph transformation to evolve software architectures", *ence, Eindhoven University of Technology, Netherlands*, pp. 31, 2008.

[16] Oussalah, M., Sadou, N., and Tamzalit, D., "SAEV: A model to face evolution problem in software architecture", In *Proceedings of the International ERCIM Workshop on Software Evolution*, pp. 137-146, April 2006.

[17] van der Hoek, A., Heimbigner, D., and Wolf, A. L., "Versioned software architecture", In ACM *Proceedings of the third international workshop on Software architecture*, pp. 73-76, November 1998.

[18] Jouault, F. and Kurtev, I., "Transforming models with ATL", In *Satellite Events at the MoDELS 2005 Conference*, pp. 128-138, Springer Berlin Heidelberg, January 2006.

[19] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M., *EMF: eclipse modeling framework*, Pearson Education, 2008.

[20] Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., and Weiss, E., "Graphical definition of in-place transformations in the eclipse modeling framework", In *Model Driven Engineering Languages and Systems*, pp. 425-439, Springer Berlin Heidelberg, 2006.