A Metamodeling Approach for Software Architecture Evolution

Amirat Abdelkrim, Mourad Chabane Oussalah

A Metamodeling Approach for Software Architecture Evolution

Abdelkrim Amirat[1] and Mourad Oussalah[2]
[1]University Mohamed Cherif Messaadia, Souk-Ahras, Algeria
[2]LINA Laboratory, University of Nantes, France[2]
{abdelkrim.amirat ; mourad.oussalah}@univ-nantes.fr

Abstract: *As software architecture evolution has become an integral part of the automated software engineering lifecycle, reuse, modularization and composition of evolution rules becomes more important. This paper aims to generalize the architecture evolution model by defining evolution rules and propagation strategies on graphs describing software architectures. We aim to define a user-definable means to manage software architecture evolution model.*

Keywords: *Architecture evolution, Component evolution, Evolution rules, Propagation strategies, Graph, Semantic relations.*

1. Introduction

Architecture evolution plays a central role in software development and has become an integral part of the automated software engineering lifecycle. In order to keep this automated lifecycle maintainable, evolution rules will have to be reusable, modular and composable [4] [6].

It is natural that we can represent a software architecture by a graph of nodes. However, due to genericity reasons, we can reduce the problem of evolution of architecture in a graph evolution problem where evolution rules and propagation strategies can be applied easily on the graph [7].

In this work we are essentially interested in the evolution of a graph of elements as long as software architecture can be described by graphs. We consider a graph as a semantic graph composed of nodes and edges. Where nodes represent architectural elements (component, connector, and configuration) and edges represent the semantic relations among these elements like inheritance, composition, and association relations, which themselves (nodes and edges) can be described as graph of elements and so on [3].

Change management in software architectures can take several forms:

- Architecture *modification*: changing the architecture without bothering about its consistency.
- Architecture *evolution*: changing the architecture while keeping it consistent so the evolution concerns the changes without trace (changes within the architecture).
- Architecture *versioning*: building and managing different versions of an architecture and providing access to these versions; so the evolution concerns the versioning of the architecture (change with trace) [1].

The main motivation factor is to maintain in a uniform way the consistency of the evolution of a graph of elements by permitting the changes of its structure and its versioning while respecting its semantic.

In our context a graph is defined by a set of nodes with a set of edges among them, where each node or an edge can be primitive or composite. Composite nodes or edges are defined by other graphs. So, we deal with an hierarchical graph where each node or edge can be described by means of another graph and so on.

Evolution is described by the changes of the structure of the graph describing the architecture or by its versioning process. To summarise these definitions we introduce the following equations:

Evolution = {propagation strategy} + {evolution rules} (1)

Evolution without trace (changes within the architecture) (2)

Evolution with trace (versioning the architecture) (3)

The idea that we want to introduce through this work lies in developing an approach to evolve systems with a backup track (versioning) [2]. So we want that our approach will be as generic as possible with the aim of applying it to all modelling levels (M3, M2, M1, and M0) defined by OMG. As consequence the mechanical operations developed to this purpose can be applied at all modelling levels. This goal is possible because the approach is focused around the graph concept which is generic too.

The remainder of the paper is organized as follows. Some motivations of our work are summarized in section 2. The necessary basic concepts required in this work are outlined in section 3. Section 4 introduces the proposed approach to deal with architecture evolution. Section 5 proposes the evolution mechanisms used in our approach. A short case study is presented in section 6. Finally section 7 concludes and provides some future work.

2. Motivations

The main motivation is to maintain in a uniform way the consistency of the evolution of a graph by permitting the changes of its structure and its versioning in respect of its semantic. This consistency is achieved via a perturbation model: starting from a graph which is initially consistent, an element of this graph evolves (node, semantic relation, attribute …) and the task of the system is to find back a new consistent graph.

Many applications require the use of graphs and their evolution. So we have defined some objectives to be reached for our graph element evolution model:

- An abstraction level of the evolution must be provided in order to allow evolutions process to be reusable and more generic.

- Evolution must be managed outside the entities concerned by the evolutions; indeed merging the evolution behaviour and the methods which describe the behaviour specific to each element runs against the behaviour abstraction of evolution model.
- The evolution model must be open to the addition of new external methods of evolution.
- The evolution model must be able to take into account the semantics of various types of relation of a graph element and not impose fixed evolution police.
- The management of the evolution must be easy and flexible.
- The evolution model must be capable to take advantage of the features of object-oriented paradigm such as abstraction, polymorphism, and encapsulation. More precisely, the principle of reusability must be widely exploited. To begin with, an evolution can concern several distinct sets of classes. Moreover, a new evolution can be defined by combining evolutions which have already been defined using inheritance or/and composition relations [8].

3. Basic concepts

The concept of graph element, which is the support of our modeling, is a semantic graph composed of nodes "architectural elements" and edges "semantic relations" like inheritance, composition or association relations. These semantic relations specify the quality of existing interactions between nodes or graphs. In our model each kind of class (graph, node or relation) is reified and then owns its structure and its behavior and in this case its evolution.

In order to express this evolution, the designer is able to attach evolution capabilities directly to his applications entities concerned by the evolution; of course he can also, by default, keep the evolution police provided by the system. Indeed, i our model, the evolution of an element is based on two components: evolution rules and propagation strategies. A propagation strategy groups together the set of evolution rules which define the operations of creation, destruction, modification, derivation, versioning applicable to a given element (graph, node or relation).

A propagation strategy, if it exists is therefore associated with each element graph, node or relation; it can be reused or redefined in the corresponding sub-element hierarchies. An evolution rule defines declaratively the actions that must be triggered on the elements concerned by the evolution. The evolution rules are defined as active rules and re reified so they can be hierarchical; they are based on the formalism of ECA rules (*Event*/*Condition*/*Action*) and are hierarchical via the inheritance mechanism. For example, the version creation or the version destruction rules of a node via an *Action* part of its evolution rule will trigger the evolution rules of the corresponding afferent and efferent relations associated with the processed node. For the relations these rules can be propagated in four directions and according to two modes [7].

The propagation direction of a relation evolution rule can be *Forward*, *Backward*, *Bidirectional*, or *None*. *Forward*, for example, means that the propagation takes place from the source of the relation to its destination. The propagation mode can be *Restricted* or *Extended*. If it is *Restricted*, the operation propagates from the extremity on which it is triggered to the relation element. If it is *Extended*, the operation propagates from the extremity on which it is triggered to both the relation element and the other extremity of the relation. The use of propagation strategies containing evolution rules allows a more flexibility because rules can be defined and carried out according to the context and needs of an information system.

4. Our approach to architecture evolution

The basis for our approach to architecture evolution centers on the concept of a graph evolution. Basically, our graph element evolution model is based on the key concepts of modeled graph element, evolution manager, propagation strategies and evolution rules as illustrated by the metamodel depicted in Figure 1.

In modeled graph element we use nodes to represent the architectural elements and edges to represent the semantic relations among these architectural elements. We rely on an object oriented modelling concepts (class diagram) to describe the metamodel of our approach.

The concept of architectural element represents any reified entity of the architecture to evolve. With each architectural element are associated some evolution strategies. A strategy consists of a set of evolution rules of an architectural element. An evolution rule describes the application of an evolution operation on an architectural element. A rule is triggered if the corresponding event occurs under predefined conditions. A rule can trigger other rules, if necessary, to spread the impact of the operation it describes. Thus, an action of a rule may correspond to an event. Furthermore, rules have a name which is unique in the namespace its grammar and can have a number of super-rules.

Rules can be abstract, which means that they are only applied in combination with non abstract sub-rules. Finally, rules have an execution mode, which can be either manual, automatic single, or automatic recursive. Manual rules have to be explicitly invoked. Automatic single rules are matched once, and then applied once by the automatic matching framework. Automatic recursive rules are matched and applied by the automatic matching framework until there are no matches.

It is only possible to define super-rule relation between rules of the same kind: manual, automatic, or recursively automatic.

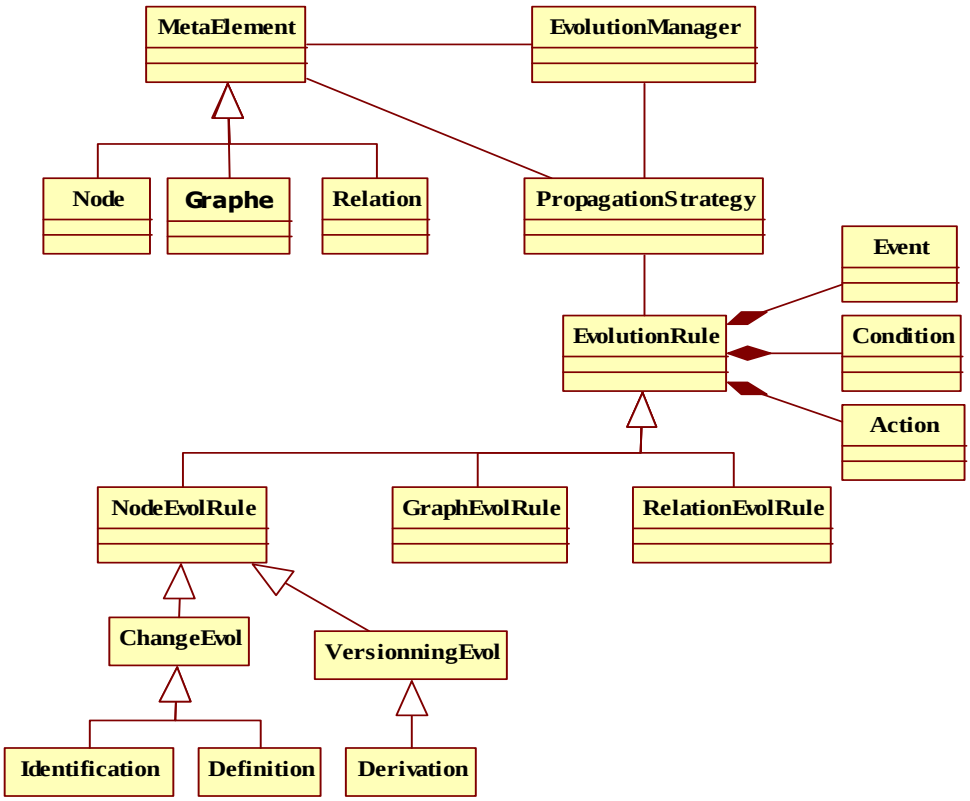

Figure 1. Evolution metamodel for software architecture.

5. Evolution mechanism

The operating mechanism describes the execution process of the evolution model. It is defined by means of four steps.

5.1. Interception of the event

An event can be intercepted in two different ways:

a- After a user request, indeed, the user selects both the element (graph, node, relation) concerned by the evolution and the rule to apply on it (deletion, modification, versioning ...etc.). The evolution manager intercepts the message representing the user choice.

b- after the execution of an evolution rule (action part), Indeed, the execution of an action of an evolution rule can involve the call of another, and so on, until the propagation is over. So, the evolution manager is responsible of the interception of any new event.

5.2. Research of the propagation strategy

The evolution manager having received a request of an evolution of an element, then looks for the corresponding propagation strategy (if it exists) and then applies this strategy to the element and triggers the corresponding evolution rules.

5.3. Execution of the evolution rules

Rules are identified by the event type to execute (for example for a node evolution the corresponding event is: delete-node, create-node-version, delete-attribute-node …) and are applied after the condition are checked. Actions of these rules cans be a program code or eventually a list of events to be executed on other elements.

5.4. Propagation

The triggering of evolution rules in the execution of their action part. This execution raise new events that will be executed in the same way, and recursively propagate other evolution rules.

In order to avoid cycles in the execution of rules, the evolution manager stores the names of elements that have been treated during a given propagation. This prevents messages concerning the same element from being taken into account more than once.

6. Case study

The example of the Figure 2 illustrates a proposed graph Gr0 to be evolved.

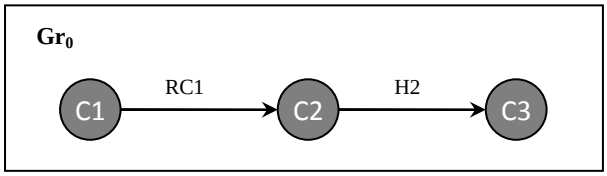

Figure 2. Gr_0 elements before evolution.

We propose the following evolution scenario: the user selects the C2 element and decides first to delete it and then create a version of the C1 and C3 elements. The results of this evolution scenario (illustrated by Figure 3) depend on the different evolutions rules described below by the designer.

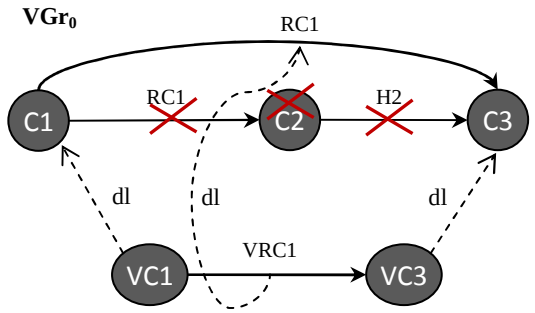

Figure 3. GR0 elements after evolution.

The different elements acting in this evolution scenario are:

Gr_0 : Graph
Nodes : C1, C2, C3
Relations : RC1, H2

C1 : Node	***C2 : Node***	***C3 : Node***
Aff. relation : - *Eff. relation: RC1* *Structure: …..* *Behavior: …..*	*Aff. relation : RC1* *Eff. relation: H2* *Structure: ……* *Behavior: ……*	*Aff. relation : H2* *Eff. relation: -* *Structure: ……* *Behavior: ……*

RC1 : Relation	***H2 : Relation***
Type : Composition *Source: C1* *Target : C2* *Exclusive : true* *Dependent : false* *Predominent : false* *Card : 1* *Reverse_card :1*	*Type : Inheritence* *Source: C2* *Target : C3* *Exclusive : true* *Dependent : false* *Predominent : false* *Card : 1* *Reverse_card :1*

Propagation Strategy	***S1 : Graph***	***S2 : Node***	***S3 : Relation***
TheDefaultStrategyForElement	*GR0*	*C1,C2,C3*	*RC1, H2*
HasAsCreationRules	*R9*	*R7*	*R1, R8*
HasAsDescructionRules		*R2*	*R4*
HasAsModificationRules	*R3,R5*	*R6*	

The different rules defined to deal with the evolution process of a graph are:

R1: Relation evolution rule	***R2: Node evolution rule***
Event: addRelation(R,N1,N2,G) **Condition:** Belong (N1, G) Belong (N2, G) **Actions:** InstantiateRelation(R,N1,N2,G)	**Event:** deleteNode(N) **Condition:** Not (Shared (N)) G ← Graph(N) **Actions:** modifyGraph(G, N, ()) executeDeleteNode(N)

R3: Graph evolution rule
Event: modifyGraph(G,N,()) **Condition:** Belong(N,G); R1 ← N.afferent; R2 ← N.efferent **Actions:** deleteRelation(R1) modifNode(R1.source, efferent, R1) deleteRelation(R2) modifNode(R2.destination, afferent, R2) G.Relations ← G.Relations – {R1, R2} addRelation(R.name, R1.source, R2.destination, G) G.Node ← G.Node – {N} G.Relations ← G.relations + R

R4: Relation evolution rule	***R5: Graph evolution rule***
Direction : forward **Mode:** extented **Event:** deleteRelation(R) **Condition:** G ← Graph (R) **Actions:** modifyGraph(G,R,()) executeDeleteRelation(R)	**Event:** modifyGraph(G,R,()) **Condition:** Belong (R, G) N1 ← R.source N2 ← R.destination **Actions:** modifyNode(N1, efferent, R) modifyNode(N2, afferent, R)

R6: Node evolution rule	***R7: Node evolution rule***
Event: modifyNode (N, type, R) **Condition:** (Belong (R, N.afferent)) **or** (Belong (R, N.efferent)) **Actions: Case** type **of** Afferent:N.afferent←N.afferent-R Efferent: N.efferent←N.efferent-R	**Event:** createVersionNode(N) **Condition:** Versionable (N) **Actions:** V(N) ← executeCreateVersion(N) G ← Graph(N) createVersionGraph(G , N)

R8: Relation evolution rule	***R9: Graph evolution rule***
Direction: forward **Mode:** **e**xtended **Event:** CreateVersionRelation(R,N,N1) **Condition:** **Exists**(V(N)) **Actions:** V(R) ← derive (R) V(N1)←createVersionNode(N1) V(R).source ← V(N) V(R).destination ← V(N1)	**Event:** CreateVersionGraph(G , N) **Condition:** **Belong**(N,G) **Let** R(N,N1) **and** R.relationOperationRule.mode= extended **Actions:** createVersionRelation(R,N,N1) V(G) ← executeCreateVersion(G)

6.1. Actions triggered

The deletion of the C2 element consists not only in deleting it, but also in propagation (using the propagation strategy) the deletion of the other elements which depend on it, like the composition relation RC1 and the inheritance relation H2.

The propagation of this modification is managed by the propagation strategy "S2" and more precisely by its destruction rule R2. Indeed, the evolution manager applies the strategy S2 which consist in bringing back its operation rule R2 dealing with the deletion of a node and then triggers it.

The description of the rule R2 consists, before deleting the node C2, to verify the conditions of this deletion (the afferent and efferent relations of the node C2 must be exclusive), and then in executing the actions "*modifyGraph(G,N,())*" and "*executeDeleteNode(N)*". So, the evolution manager incepts the next event consisting in: "*modifyGraph(G,N,())*". This event is send to the graph entity GR0 to which we have associated the strategy S1 which owns two modification rules R3 and R5. In this case, the rule R3 is selected by the evolution manager. The other operations follow these steps:

- ✓ *Strategy S1, rule R3 on graph GR0*
  - o *Strategy S3, rule R4 on relation RC1*
    - *Strategy S1, rule R5 on graph GR0*
      - *Strategy S2, rule R6 on node C1*
      - *Strategy S2, rule R6 on node C2*
  - o *Strategy S2, rule R6 on node C1 // if needed*
  - o *Strategy S3, rule R4 on relation H2*
    - *Strategy S1, rule R5 on graph GR0*
    - *Strategy S2, rule R6 on node C2 // if needed*
    - *Strategy S2, rule R6 on node C3*
  - o *Strategy S2, rule R6 on node C3 // if needed*
  - o *Strategy S3, rule R1 on relation CR1*

Concerning the creation of the version of the C1 node, the following rules are triggered:

- ✓ *Strategy S2, rule R7 on node C1*
  - o *Strategy S1, rule R9 on graph GR0*
    - *Strategy S3, rule R8 on relation RC1*
  - o *Strategy S1, rule R9 on graph GR0*

By default, the new creating elements (VRC1, VC1, VC3 and VGR0) are associated to a predefined strategies and rules of elements types which they depend on. However, the designer is free to redefine or to specialize them for a targeted application.

7. Conclusion

The proposed evolution model respects most of the objectives we determined before the design process. In addition to the mechanisms which are inherent in the representation of the evolution (propagation strategies and evolution rules) by objects of the first class, the specialization of the evolution and application graph classes may be dealt with independently. The principal originality of our model lies in the fact that different semantics of graph element evolution can be taken into account.

Moreover, it differs from the existing models in two points: 1- It proposes a uniform way to manage both changes and versioning in a same objects base. 2-It permits extensibility and the reusability of the different rules and strategies of a graph of elements evolution.

References

[1] Oussalah M., "Changes and Versioning in complex Objects", *International Workshop on Principles of Software Evolution*, IWPSE 2001, Sep. 10-11, Vienna University of Technology, Austria.

[2] Chaki S., Diaz-Pace A., Garlan D., Gurfinkel A., and Ozkaya I., "Towards engineered architecture evolution", in MiSE, *ICSE Workshop on Modeling in Software. Engineering*, pp.1-6, 2009.

[3] Taylor R.N., Medvidovic N. and Dashofy E., "*Software Architecture: Foundations, Theory, and Practice"*, Wiley-Blackwell, 2009.

[4] Barais O., Le Meur A., Duchien L., and Lawall J., "Software architecture evolution", *Software Evolution*, Springer, 2008.

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

[6] Jazayeri M., "On architectural stability and evolution", *Proceeding of Ada-Europe'02*, 2002.

[7] Tamzalit D., Sadou N. and Oussalah M., "Evolution problem within component-based software architecture", in *Proceeding of International Conf. on Software Engineering and Knowledge Engineering (SEKE'06)*, 2006.

[8] Amirat A., Menasria A., and Gasmallah N., "Evolution Framework for Software Architecture using Graph Transformation Approach", *The 12th International Arab Conference on Information Technology (ACIT'2011)*, December 11-14, Riyadh, Saudi Arabia, pp. 75-82, 2011.