EVOLUTION FRAMEWORK FOR SOFTWARE ARCHITECTURE USING GRAPH TRANSFORMATION APPROACH

Abdelkrim Amirat, Ahcene Menasria, and Nouredine Gasmallah

University Center of Souk-Ahras, Algeria
amirat_karim@yahoo.fr, ahcene_menasria@yahoo.fr, gasmallahedi@yahoo.fr

ABSTRACT

This paper presents a graph transformation approach to software architecture evolution. Evolution is inevitable over the course of the complete life of complex software intensive systems and more importantly of entire product families. However, not only instance models, but also type models and entire modelling languages are subject to change. Obviously, software architecture is the centerpiece of software systems and acts as reference point for many development activities, and few of today’s software systems are built to accommodate evolution. Evolution is primarily reflected in and facilitated through the software architecture. In this paper we focus on the different dimensions of architecture evolution with an automated evolution process of software architecture using graph transformation. The rules for the potential architecture evolutions operators are defined using AToM³ graph transformation tool.

Keywords: Software Architecture, Architecture evolution, Evolution operator, Graph grammar, AToM³.

1. INTRODUCTION

Software evolution is a process by which system change, adapting marketplace, and inheriting characteristics from pre-existing applications [1]. A key aspect of software evolution is architecture evolution. When a system evolves, its architecture is impacted. On the contrary, planning for architecture evolution is a powerful tool to guide and plan for software evolution.

Most work on software evolution focuses on practices towards managing code. However, the software architecture is the keystone for ensuring that software intensive system achieves its business and quality goals over the lifetime of the system [2]. So, in order to make the software evolution easier and with optimal cost it is recommended that the architecture should be designed with evolution in mind.

Currently, planning for evolution is an ad-hoc activity, not generalizable, and specific to a particular instance of the model. Mostly, the plan is constructed as evolution taking place. This plan is captured in variety of forms: a list identifying key milestones, timeline to achieve them, action to be done at each milestone, intermediate release cycles, etc. Since such planning is not supported by sound engineering methods and tools, the quality of the result depends, almost completely, on the know-how and common sense of the architect of the system. To overcome this drawback, we think that evolution must be planned in a generalizable and reusable manner. This means that the architecture evolution activity should be engineered.

On the other hand, sometimes we are forced to extend existing software in an unforeseen way, e.g., the addition, removal or change of components or entire subsystems should be possible without having to plan this design step before the system is started.

Graphical notation provides an intuitive and flexible approach to describe structural information. Compared with other formal methods, graph transformation [3] is a visual pattern and rule-based manipulation of graph models and is suitable for describing the evolution of software architectures. Graph transformation is theoretically well founded and many matured environments and tools [4] are available to support the design and implementation of graph transformation. Furthermore, since software architectures can be represented as graphs, an evolution process can be simulated with the application of transformation rules on those graphs.

In this paper we define the graph transformation rules that cover the potential evolutions for component-based software architecture. To address this issue, we focus on a particular kind of evolutions – planned evolution – which are amenable to being engineered. An evolution is planned if the properties of the current and target architectures are known a priori. We propose a conceptual framework based on set of generic and reusable operators implemented using AToM³ (A Tool for Multi-formalism and Meta-Modelling) [20]. This framework is named C3Evol, which enables architects to design and make evolve component based systems in an iterative and uniformed process. C3Evol allows software architects to specify their architectures as models and to analyze them with respect to application and platform constraints. In reality C3Evol is an extension of C3 (Component, Connector, and Configuration) metamodel developed to describe hierarchical component-based software architectures [5].

To summarize, our contributions in this paper are:
1. Multi-dimensional identification of aspects to approach the architecture evolution activity,
2. How to use graph transformation as support for architecture evolution, and
3. Proposition of a model to describe architecture evolution operators using graph transformation.

The remainder of the paper is organized as follows. A review of the related work is summarized in section 2
that helps in outlining the research questions in section 3. Section 4 briefly introduces the Triple Graph Grammar formalism and the concept of graph transformation. Section 5 proposes the C3Evol framework of our approach to deal with architecture evolution. Finally section 6 concludes and provides some future work.

2. RELATED WORK
There is a large body of work on maintaining and aligning software to changing requirements, business goals and practices. In this paper we focus only on formal approaches to architecture transformation as related area to architecture evolution.

A number of researchers have proposed formal models to capture structural and behavioral transformation. For example, Wermelinger et al. propose to use a uniform algebraic framework based on Category Theory to describe how transformations can occur in software architecture. Architecture in this sense is defined by the space of all possible configurations that can result from a certain starting configuration [6].

Grunske [7] shows how to map architectural specifications to hyper graphs and uses these to define architectural refactoring that can be applied automatically and also preserve architectural behavior. The approach is based on triple graph grammars and triple graph grammar rules, which provide a deep theoretical concept for data integration between different graph-based structures. An important feature of triple graph grammar rules is the implicit creation of a correspondence graph between the two models. This allows incremental change propagation in case one model evolves.

Tamzalit and others have begun to investigate recurring patterns of architecture evolution, primarily with respect to component-based architectures [8]. They characterize patterns for updating a component-based architecture. They provide a formal approach based on a three tiered conceptual framework. They attempt to capture recurring and reusable patterns of architecture evolution. However, they do not explicitly characterize or reason about the space of architecture paths, or reason about how to select appropriate paths.

Graaf [9] presents a model-driven evolution of software architectures. The goal of this work is to investigate techniques that reduce the risks and costs involved in the evolution of software architectures. To structure this problem he introduces an evolutionary software life-cycle defined by means of four software evolution tasks to be investigated: evaluation, conformance checking, migration, and documentation.

Barais [10] identify two main architectural kinds of changes in software architecture:
1. Internal evolution, models the changes of the topology of the components, connectors, and configuration. Components and connectors may be created or destroyed during execution.
2. External evolution, allows the specification of the components and connectors (i.e., the architecture) to be changed during execution. It captures the needs for an architecture description to be changed in order to cope with the evolution of requirements.

Recently Garlan and Schmerl [11] introduce a tool called Ævol that provides a platform for exploring the foundations of architecture evolution and evolution styles. This tool allows an architect to specify evolution paths, and is integrated with a software architecture design tool to allow the architectures in an evolution path to be visualized and edited. A key feature of Ævol is its support for pluggable analysis of both correctness conditions for evolution, as well as cost-benefit analysis for comparing alternative paths. Ævol is based on an augmented version of linear temporal logic (LTL) which is a modal temporal logic with modalities referring to time. LTL is sufficient to express many interesting properties in software architecture and to specify evolution path constraint.

3. DIMENSIONS OF ARCHITECTURE EVOLUTION
The common types of component-based systems evolution are classified following the dimension of the evolution and the abstraction level of the evolution. Figure 1 illustrates our multi-dimensional classification of software architecture evolution tasks.

3.1. EVOLUTION RESOURCE
Architecture resource, a common issue in evolution resource is the need to differentiate between local changes (e.g., to a component, a connector, or an interface) and architectural changes (e.g., a local subsystem, configuration or system topology) [12]. Architectural changes affect several elements, hence; can have “ripple effects” that need to be accounted for. To account for ripple effects and estimate the effort involved, every step in an evolution plan must be traceable to its source. In this kind of evolution the source and target architectures have the same architecture metamodel, and so the target architecture can be initialised to be the same as the source architecture before the transformation. This kind of evolution is called endogenous architecture evolution as illustrated by Figure 2. We consider that style evolution fall in this category of transformation.
Domain resource, another issue in evolution resource is domain knowledge. This is especially true when adopting a new technical infrastructure, such as moving from client server architecture (CS) to service oriented architecture (SOA). In such cases, the architect needs technical information on the new domain to capture the key architecture properties of the new infrastructure required for evolution resource. As shown in Figure 3, this kind of architecture evolution is called architecture migration from a source infrastructure to another.

Metamodel resource, this kind of resource considers the case of architecture metamodel evolution. So, if the metamodel evolves all architecture instances must evolve too. The source and target metamodels differ, and the target architecture model is constructed entirely by the transformation. This type of evolution is called exogenous architecture evolution. Figure 4 illustrates this category of transformation.

3.2. EVOLUTION TIME

Static evolution, static evolution time refers to changes undertaken during the design phase of the system. Once the new architecture is described a new instance of the system is generated and executed. This kind of evolution is quite simple to realize.

Dynamic evolution, dynamic evolution time is so critique because the architecture should be changed and it should be synchronized with the system in execution (e.g., replace a component in run-time). In former case the execution should be stopped, save the actual state of the system, operate changes, restore the state of the system, and resume the execution of the system. So, the architecture and the system are in permanent co-evolution synchronization. Generally, this type of evolution is required in critical systems (e.g., baking systems, medical monitoring systems, sensor systems, etc.)

3.3. TYPE OF RESOURCE

This dimension takes in consideration the support expected for the object of the evolution. The support can be implicit, predefined, or explicit. Along we know ADLs model architectural elements in various forms and under various names [13]. Architecture evolution should be studied with respect to the support ALDs provide for different aspects of elements.

Explicit resource represents architectural resources which are modelled explicitly. For example, ACME, Aesop, C2, SADL (Simulation Architecture Description Language), UniCon, and Wright model connectors explicitly and refer to them as connectors. Weaves also models connectors explicitly, but refers to them as transport services.

Predefined resource represents architectural resources which are previously predefined by the ADL and the architect has only the possibility to use them in his design. For example, UniCon supports only a predefined set of role types, including Source, Sink, Reader, Readee, Writer, Writee, Definer, and Caller.

Implicit resource represents resources which are not first-class entities. For example Rapide and MetaH connections and Darwin bindings are modeled in-line, and cannot be named, sub-typed, or reused. Darwin and Rapide do allow abstracting away complex connection behaviors into “connector components”.

3.4. EVOLUTION LEVEL

In addition to the previous three dimensions of evolution, we introduce a separate dimension called evolution level which represents the abstraction level of the previous dimensions. Figure 5 represents the four abstraction levels of systems evolution. Our vision to this dimension is rooted in the argument made by Jazayeri – “Software is more than source code; models and meta-models are important to software evolution” [14]. Evolution level and the previous ones are truly orthogonal.

Figure 2: Endogenous architecture evolution.

Figure 3: Architecture migration.

Figure 4: Exogenous architecture evolution.

Figure 5: Abstraction levels of evolution.
System evolution (E0), application evolution is a fine-grained, short-term activity of the product life-cycle that focuses on localized changes of the code. Basically, evolution at this level represents the classical maintenance activity.

Architecture evolution (E1), it is more easy and reasonable to evolve the architecture than its instances (applications). And the synchronisation (co-evolution) mechanism is charged to evolve the application automatically.

Meta Architecture evolution (E2), at this level of abstraction the evolution process focus on the evolution of languages, notation and approaches used to describe the evolution operators applied at the lower level (E1). So, when the syntax of a modeling language evolves (i.e., the meta-model evolves), the most prominent side-effect is that its instance models may no longer conform to the new meta-model. Therefore, the co-evolution (with evolution of their meta-model) of models has become an important research topic. It is widely accepted that a model co-evolution (i.e., migration) is best modeled as a model transformation [15], which we will call the migration transformation.

Meta-meta-architecture evolution (E3), in this work we are not concerned by the evolution at this level. Nevertheless, this kind of evolution is about the MOF [16] if we are in the context of metamodeling or about MADL (Meta Architecture Description Language) [17] if we are in the context of ADL modeling.

It is vital to preserve the structural and semantic properties of the architectural elements during evolution to maintain a consistent architectural representation at all abstraction levels. Basically, the evolution process at each modelling level is guided by the evolution of the above modelling level. In this work we focus on the architecture evolution as a common and an ideal abstraction level for the evolution of systems.

3.5. EVOLUTION PATH AND PLANNING

Another important aspect of software evolution is the evolution path. It represents the sequences of architecture evolution releases starting from the source architecture until the target one. Depending on the target architecture, the evolution itself can fall in the following two kinds [18]:

Open evolution, open evolution is characterized by high uncertainty. While aspects of the new architecture direction are known, business, technical, and market conditions prevent architects to shape clear target architecture a priori as illustrated by Figure 6(a).

Closed evolution, closed evolution is where the characteristics of the current and envisioned system’s architectures are known as indicated by Figure 6(b). This close-ended evolution is motivated by new technical solutions, domain changes, new business models, etc. There are many historical examples of closed evolution, e.g., from mainframe to client-server, from stand-alone application to web-based systems, and from text-based to graphical user interface-based systems.

Coarse granularity, represents the case where the architectures within the trajectory are only A0 and An. This approach does not provide the architect with the planning path.

Low level of granularity, representing every key structural change as a state within the trajectory. This second approach is not feasible and leads to “analysis paralysis”.

A complete path of the evolution can be constructed with a suitable level of abstraction. To do so, we need a mechanism to capture key structural changes between intermediate architectures. To achieve this goal, we extend the path with evolution operators. Formally, a path is constructed from an alternative sequence of architectures and operators as illustrated in Figure 6. Where each Ai represents a version of the architecture and each Op_i is an operator. Intuitively, an operator Op_i represents the steps taken between architectures Ai and Ai+1 which represent successive architecture evolution releases. Each operator type is described at an abstract level by one or more semantic actions. The operator instances are augmented by structural transformation rules of an architecture, such as “identify a service”, and structural transformation of an architecture, such as “some components and/or connectors are introduced and/or removed”. A subset of evolution operators are described in Table 1.

Developing a meaningful and reusable set of evolution operators is a fundamental activity in this context. Currently, we define a basic set of transformation operators, as indicated by Table 1, where ‘e’ represents an architectural element (component, connector, a configuration).

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add (e)</td>
<td>Add operator enable the architect to add ‘e’ to the architecture</td>
</tr>
<tr>
<td>Remove (e)</td>
<td>Remove enable the architect to remove ’e’ from the architecture</td>
</tr>
<tr>
<td>Modify (e, p)</td>
<td>Modify specify a new interface ‘p’ for the element ‘e’</td>
</tr>
<tr>
<td>Replace (e1, e2)</td>
<td>Replace ‘e1’ by ‘e2’ in the architecture</td>
</tr>
<tr>
<td>Merge (e1, e2, e)</td>
<td>Merge replaces elements e1 and e2 by the e</td>
</tr>
<tr>
<td>Split(e, e1, e2)</td>
<td>Split divides e in two elements e1 and e2</td>
</tr>
</tbody>
</table>

Table 1: Basic evolution operators
To achieve a sequence of evolution actions on the architecture, a sequence operator can be used. The transformation operators along with architectural element are composed into the transformation patterns. During transformation these operators should preserve the structural properties of architectural elements.

4. TGG AND GRAPH TRANSFORMATION
In this section we give an overview of the Triple Graph Grammar (TGG) formalism and graph transformation, which are the theoretical foundation of the proposed approach in section 5.

4.1. TRIPLE GRAPH GRAMMAR
Graph grammars extend the generative grammars of Chomsky into the domain of graphs. Different from string grammar expressing sentences in sequence of characters, graph grammars are suitable for specifying visual information in a multi-dimensional fashion.

Triple Graph Grammars (TGG for short) [3] can be used to specify translators of data structures, check consistency, propagate small changes, and defining the dynamic evolution of a single model.

The main idea of graph transformation is the rule-based modification of graphs (rewriting) as shown in Figure 7.

![Figure 7: Rule-based modification of graphs.](image)

The core of a rule (production) \( p = (L, R) \) is pair of graphs \( (L, R) \) called left hand side LHS and right hand side RHS. Applying the rule \( p = (L, R) \) in a graph instance (called host graph) means to find a match of \( L \) in the host graph and to replace \( L \) by \( R \) leading to the target graph of the transformation. Any transformation of graphs can be realized by applying a sequence of productions.

4.2. GRAPH TRANSFORMATION
Software architectures may be described by component diagrams thus it is natural to use graphical notation to depict and transform them through graph transformation.

Graph transformation is the application of a sequence of rules on a given host graph. The transformation process produces a new graph from the input host graph after applying transformation rules. Moreover, the transformation process termites when no more transformation rules can be applied.

Let \( L \) be the LHS of a grammar rule \( r \) and \( R \) be the RHS of the rule. Let \( G \) be a host graph. The transformation from graph \( G \) to graph \( H \) by rule \( r \) can be achieved through the following steps:
1. Recognize sub-graph \( L \) in the host graph \( G \).
2. Check if the transformation rule can be applied.
3. Replace sub-graph \( L \) by sub-graph \( R \).

4. Connect the dangling edges if the node is marked to preserve the association surrounding of the replaced sub-graph \( L \).

Step 1 and 2 can be classified as a pattern matching process and step 3 and 4 are used to update the surrounding connections with the host graph. The Figure 8 shows a process of graph transformation. In this figure we introduce the grammar rule specified to remove the component \( B \) and its replacement by two other components \( B1 \) end \( B2 \) (split operator).

![Figure 8: A graph transformation process.](image)

5. C3EVOL EVOLUTION MODEL
In our study we describe architectures using C3 architecture metamodel defined in our previous work [5]. Figure 10 illustrates the UML class diagram of C3 metamodel.

5.1. C3 META MODEL
As in most ADLs, including UML 2.0, in C3 metamodel Components correspond to the major run-time computational elements and data stores of a system, while connectors define the pathways of interaction between components. Components and connectors can be primitive or composite elements. Interfaces of components and connectors are termed ports. Architectures may be defined hierarchically where elements may be defined as configurations as indicated by Figure 9.

![Figure 9: First class architectural elements are annotated with properties that provide more detailed semantics.](image)

First class architectural elements are annotated with properties that provide more detailed semantics. Software architecture can be represented as a graph in which the nodes represent first class entities (e.g., component, connector, configuration, ports, etc) and edges represent relationship between these entities.

In our context of evolution, TGG allow us to define the relation between two different kinds of models and to transform one kind of model into the other. As long as the syntax and semantics of ADLs are precisely defined by means of meta-modelling [19] and model transformation, we can use AToM³ (A Tool for Multi-formalism and Meta-Modelling) [20] as graph transformation [21] tool to implement architecture evolution tasks. In the annex given in the last of this paper, we define the metamodel of C3, we introduce client-server example as a source architecture. To produce the target architecture we apply rules (1, 2 and 3) defined using AToM³ Tools with some python
constraints defined in pre-condition and post-condition associated with each rule.

![Diagram](image)

**Figure 9: Core elements of C3 meta-model.**

Basically, AToM³ is a tool for the design of Domain Specific Visual Languages. It allows defining the abstract and concrete syntax of the Visual Language by means of meta-modelling and expressing model manipulation by means of graph transformation. With the meta-model information, AToM³ generates a customized modelling environment for the described architecture language. Evolution operators are performed by means of rules application. Rules should be applied according to their priority fixed by the architect. Also, rules are grouped in separate sets represented by graph grammars. Each graph grammar is designed to perform the semantic actions of an evolution operator.

After the rules (1, 2 and 3) presented in the annex using AToM³ tool, we present in the following other rules to realise some evolution operation like connector replacement (rule 4), split of a component (rule 5), merge of components (rule 6) and delete a component from a configuration (rule 7) as illustrated by Figure 10. Other rules are defined but for space constraint reason we cannot represent them in this paper.

Generally the rules of the grammar are ordered with a priority property. And most of them parameterized with the variability information about the architectural element to be handled (e.g., the deletion of an element by its name or by its type), so if we use the name only the component having this name is deleted but if we the type all components of this type should be deleted from the architecture.

**Rule 4: Connector replacement**

![Rule Diagram](image)
6. CONCLUSION

We introduced our concept of visual model evolution and illustrated the basic notions using a small client-server example. The main ideas are on the one hand a concept of visually representing the architecture of a system and on the other hand a rule-based approach for the description of consistent system evolution.

To provide reliability to the design and evolution process, a graph transformation tool enables architects to specify the structure of application architecture, as well as its properties. Actually we explore all possibilities of AToM³ to define our meta-model and all variants of graph grammars implementing evolution operators.

After all ideas, in order to make easer the evolution process, the architecture of the system should be designed with evolution in mind (i.e. the architecture should evolve).

Future work includes several extensions. Indeed, using the formal basis of our approach to formalize and check e.g., invariants of an architecture. Furthermore, our incremental evolution approach considers the structure of the application only. It would be interesting to address also the evolution of the behaviour.

REFERENCES


ANNEXE

Figure A1: C3 Metamodel described using AToM³ Tool.

Figure A2: Source Architecture.

Figure A3: Target Architecture.

Figure A4: Rule 1- Delete a component

Figure A5: Rule 2- Addition of a component.


Figure A6: Rule 3- Addition of a component with its connector.