Towards Architectural Evolution through Model Transformations

João Pimentel, Emanuel Santos, Diego Dermeval, Jaelson Castro
Centro de Informática
Universidade Federal de Pernambuco
Recife, Brazil
{jhcp, ebs, ddmcm, jbc}@cin.ufpe.br

Anthony Finkelstein
Department of Computer Science
University College London
London, United Kingdom
a.finkelstein@ucl.ac.uk

***Abstract*—The increasing need for dynamic systems, able to adapt to different situations, calls for underlying mechanisms to support software evolution. In this sense, model-based techniques can be used to automate one of the evolution aspects – the modification of models. However, the use of model-based techniques is tailored to the specific modeling languages being used. Thus, efforts to automate the modification of models can be undermined by the several different modeling languages used in different approaches for software evolution. Aiming to facilitate the use of model-driven development in the context of architectural evolution, we propose an approach to define basic transformation rules. This novel approach relies on a conceptual model and a set of basic operations for architectural evolution, which are used to define transformation rules for a specific architectural modeling language of interest. We illustrate the application of our approach by defining transformation rules for Acme using QVT.**

Keywords-software architecture; architectural evolution; autonomic systems

I. INTRODUCTION

Software evolution has become a key research area in software engineering [7]. Software artifacts and systems are subject to many kinds of changes at all levels, from requirements through architecture and design, as well as source code, documentation and test suites. Since the abstraction level of software architecture is adequate for identifying and analyzing the ramifications of changes [11], it could be one of the software evolution pillars [17]. As the architecture evolves, mechanisms are required for supporting these dynamic changes [1][14][19]. There are several approaches for tackling different aspects of architectural evolution, often relying in some kind of model transformation. However, these approaches do not use of model-driven engineering techniques, which provide underlying mechanisms for model transformation.

In this paper we present a novel approach for creating basic transformation rules with the focus of facilitating model-based architectural evolution. We defined a conceptual model and a set of basic operations that can be applied to the different languages used for architectural modeling. We illustrate our approach by defining transformation rules for architectural evolution on a specific ADL – Acme [9]. The contribution of this paper is twofold. On one hand, our basic operations can be used as a common vocabulary for the different architectural evolution approaches – facilitating their integration. On the other hand, it can guide the creation of transformation rules for a specific modeling language.

The remainder of this paper is structured as follows. In Section 2 we present the background for this work. Our conceptual framework is described in Section 3. Section 4 illustrates the use of our approach in Acme. Lastly, Section 5 concludes the paper.

II. BACKGROUND

Architectural evolution has been acknowledged as a key element for achieving software evolution [17]. According to [6], there are 5 types of software evolution: Enhancive, Corrective, Reductive, Adaptive and Performance. In the case of autonomic, self-adaptive or self-managing systems this evolution is performed at runtime, with some degree of automation. Architectures that can evolve at runtime are classified as dynamic architectures. Our approach supports evolution both at design time and at runtime.

A survey on formal architectural specification approaches, regarding their ability to enact architectural changes at runtime is reported on [5]. That survey analyzes the approaches regarding the type of changes they support: Component Addition, Component Removal, Connector Addition and Connector Removal. As result, 9 out of 11 approaches were found to support all these basic operations: CommUnity, CHAM, Dynamic Wright, PiLar, Gerel, ZCL, Rapide, as well as the approaches by Le Métayer and by Aguirre-Maibaum. Moreover, all these 9 approaches have some kind of support for composing these operations. The need for structural change is clear in the architectural deployment view, for example, by replicating application servers in order to improve the system performance. Additionally, the use of subtyping mechanisms to enable architectural evolution has also been suggested [15]. From the 9 architectural description languages analyzed in [15], 4 show some kind of support for evolution through subtyping mechanisms: Aesop, C2, SADL and Wright.

As the elements and connectors of an architecture evolve, their properties may be modified as well. These properties may be related to the element itself (e.g. performance), or to the use of the element (e.g. workload). The modification of the properties may happen at design time or at runtime. For instance, at runtime, we may consider a connector (in a

deployment view) that is reified in a physical network. Properties such as reliability and bandwidth of this connector may be subject to change over time. This kind of changes in properties of architectural elements is often ignored in the architectural models. We advocate that the evolution of these properties should be reflected in the models, as it allows us to (i) monitor the evolution of these properties, which in turn can be used to trigger adaptations, and to (ii) analyze the actual characteristics of a system using its architectural models. This alignment between what was designed (the initial model) and the actual implementation/deployment can help identify and reduce architectural erosion [15]. A particular research field that considers the evolution of the properties of architectural elements is that of service-oriented architectures. For instance [4][8] match services properties with non-functional requirements for selecting which services to use.

Some modeling languages that are not considered ADL can also be used for architectural modeling. Architectural behavior has been defined in languages such as Statecharts, ROOMcharts, SDL, Z, Use-Case, Use-Case Maps, Sequence diagrams, Collaboration diagrams and MSCs [2]. For instance, Statecharts can be used to describe the different states of a component, as well the transitions between them; Use-Case diagrams can express the different activities performed by an element of a system, from the user' point-of-view. Thus, when dealing with architectural evolution, we cannot neglect these languages. Similarly, the use of goal-based notations such as Kaos and *i** for architectural modeling has been promoted in some research endeavors [12][18].

III. Conceptual Framework

Based on a review of the architectural modeling approaches mentioned in the previous section, we devised a framework for empowering architectural evolution through model transformations, which can be applied to different modeling languages. This framework comprises a conceptual model, used to classify the different constructs of a modeling language, as well as a set of basic architectural evolution operations defined upon the conceptual model.

Architectural models are composed of architectural elements – e.g., components, services, classes and states – and links that define connections between these elements – e.g., connectors, requests, association links and events. Both elements and links may have properties, which can be used to provide a detailed description of elements and links. Moreover, elements may have sub-elements, i.e., elements that are part of them. Fig. 1 shows a model that represents these concepts. Using the graph terminology, elements can be considered typed nodes, links can be considered directed edges and properties can be considered labels of a node/edge.

This conceptual model can be applied to different architectural description languages, as well as to other modeling languages that are used for architectural modeling (e.g., Statecharts, Sequence diagrams, Use-Case and *i**). The (meta-)metamodel of the VPM language [3] resembles our conceptual model, being essentially composed by elements (entities) and links (relations) but neglecting their properties. That work provides a metamodeling language, whereas our conceptual model is used to classify the constructs of existing

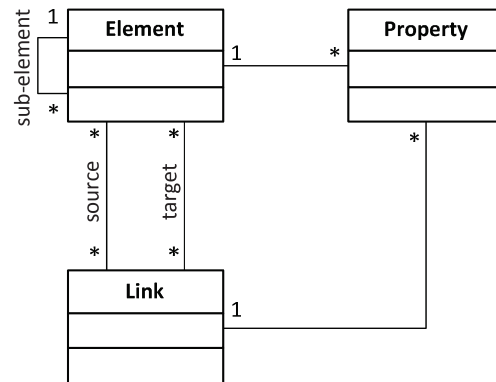

Figure 1. Conceptual Model

metamodels in order to guide the creation of transformation rules. Thus, our approach is not tied to any particular metamodeling nor transformation specification language.

A. Basic Architectural Evolution Operations

This conceptual model enabled us to define 7 basic operations required to support architectural evolution with model transformations: Add Element, Remove Element, Add Link, Remove Link, Add Property, Remove Property and Change Property. These operations support the 2 different types of architectural model changes described in Section 2 – structural/topology changes and property changes. These operations are described next.

1) Add Element

Inserts a new element in a model. A particular case is when the new element is a sub-element of another element. Usually, an element has a name or an identifier.

2) Remove Element

Deletes an element from a model. Caution should be taken when removing elements from a model, as it is important to maintain the integrity of the model. For instance, if an element is removed, it is most likely that it will also be necessary to remove the links associated with that element.

3) Add Link

Inserts a new link connecting elements of the model.

4) Remove Link

Deletes a link from a model. Here again caution should be taken, as elements without links may result in invalid models.

5) Add Property

Inserts a property in an element or in a link. Usually, the property has a name or an identifier, as well as other attributes such as value, default value and type.

6) Remove Property

Deletes a property from an element or from a link.

7) Change Property

Modifies the content of an attribute of a property, e.g., the actual value, or its type. A similar effect could be achieved by combining the *Remove Property* and the *Add Property* operations. However, as modifying a property is conceptually

different from replacing it, we decided to define this specific operation.

We decided against the definition of *Change Element* and *Change Link* operations as they essentially consist of changing their properties or adding/removing sub-elements.

These operations are similar to the five basic operations for graph-based model transformations [13]: create node, connect nodes, delete node, delete edge and set label. Our notion of property can be seen as an elaborated kind of label, leading us to defining three different property-related operations. Moreover, our conceptual model also supports the definition of sub-elements, which is an important feature in some architectural modeling languages.

B. Model Transformation Rules

In order to use the power of model-driven development for architectural evolution, it is necessary to define transformation rules for a particular modeling language. In order to describe these rules, the first step is to classify the constructs of the language based on the conceptual model of Fig. 1 - i.e., to identify which are the elements, sub-elements, links and properties of that particular language. Then one can proceed to instantiate the basic architectural evolution operations for each element, link and property of the language.

Once all basic operations are defined, the transformation rules can be developed using a model transformation framework (such as QVT) or using a general-purpose programming language. The development of these rules can be quite straightforward, as presented in the next section. However, the complexity of the language in focus may pose some additional challenges.

IV. Model Transformations for Architectural Evolution on Acme

In this section we illustrate the use of our conceptual framework to enable architectural evolution on a specific ADL: Acme [9]. Acme components characterize computational units of a s ystem. Connectors represent and mediate interactions between components. Ports correspond to external interfaces of components. Roles represent external interfaces of connectors. Ports and roles are points of interaction, respectively, between components and connectors – they are bound together through attachments. Systems are collections of components, connectors and a description of the topology of the components and connectors. Systems are captured via graphs whose nodes represent components and connector and whose edges represent their interconnectivity. Properties are annotations that define additional information about elements (components, connectors, ports, roles, representations or systems). Representations allow a component or a connector to describe its design in detail by specifying a sub-architecture that refines the parent element. The elements within a representation are linked to (external) ports through bindings.

A. Applying the conceptual framework

Considering our conceptual framework, we identified the basic operations required to evolve an architectural model in Acme. The Acme elements are *Component*, *Connector*, *Role, Port* and *Representation*. Particularly, the last three elements are sub-elements – *Role* is a sub-element of *Connector*, *Port* is a sub-element of component and *Representation* is a sub-element of both *Component* and *Connector*. Please note that *Connector* is not an actual link – instead, it is an element that is (indirectly) linked to *Component* through *Attachment*. The only links in Acme are *Attachment* and *Binding*. An attachment links an internal port of a component to a role of a connector, while a binding links an internal port to an external port of a representation. Lastly, *Property* expresses the properties of each element or of a *System*.

Thus, the basic operations for architectural evolution in Acme were defined: Add Component, Add Port, Add Connector, Add Role, Add Representation; Remove Component, Remove Port, Remove Connector, Remove Role, Remove Representation; Add Attachment, Add Binding; Remove Attachment, Remove Binding; Add Property; Remove Property; Change Property. Since all properties in Acme share the same structure, we did not need to define different property operations – e.g., there is no benefit in defining different *Add Component Property* and *Add Connector Property* operations.

B. Acme evolution with QVT

In this section we present how the basic operations can be implemented using a model transformation framework. Here we are using QVT, which comprises a language for specifying model transformations based on the Meta Object Facility – MOF and on the Object Constraint Language – OCL.

Fig. 2 s hows the QVT Operational code for the *Add Component* operation. The first line states the metamodel that will be used in the transformation, which is the Acme metamodel. The second line declares the *Add Component* transformation, informing that the same model (aliased *acmeModel*) will be used both as input and as output. Other than the input models, QVT transformations accept input through the mechanism of configuration properties. Line 3 presents the input variable of this transformation, which is the component name. In QVT, the entry point of the transformation is the signature-less *main* operation. Our *main* operation (lines 4-7) calls the mapping of the root object of the model, which is *System*. The transformation itself is performed by the *Apply Add Component* mapping (lines 8-12), which inserts a new component. The component constructor is defined in lines 13-16, it simply assigns the given name to the component.

Fig. 3 presents the mapping of the *Remove Port* transformation (the other elements of the transformation are very similar to that of Fig. 2). In this mapping we traverse all ports of the model (Line 3), so that when port with the given name is found (Line 5) it is deleted from the model (Line 7).

Here we have described a straightforward implementation of the basic architectural evolution operations for Acme. However, there is room for improvement. For example, *where* clauses can be defined for preventing the addition of a component with an empty name, or the removal of a port that is still attached to some role.

V. Conclusion and Future Work

The conceptual framework defined in this paper is generic enough to allow its application on different architectural

```
1  modeltype Acme uses Acme('acme2');
2  transformation AddComponent(inout acmeModel : Acme);
3  configuration property componentName : String;

4  main()
5  {
6    acmeModel.rootObjects()[System].map
   applyAddComponent();
7  }

8  Mapping inout System::applyAddComponent()
9  {
10     self.acmeElements += new
11 Component(this.componentName);
12 }

13 constructor Component::Component(myName : String)
14 {
15     name := myName;
16 }
```

Figure 2. QVT code for the Add Component operation

```
1  mapping inout System::applyRemovePort()
2  {
3     self.allInstances(Port)->forEach(p)
4     {
5        if (p.name.=(this.portName)) then
6        {
7           acmeModel.removeElement(p);
8        }
9        endif;
10    };
11 }
```

Figure 3. QVT code excerpt for the Remove Port operation

modeling languages of different architectural views (e.g., module, components and connectors, and allocation views [10]). By classifying the constructs of the modeling language according to our conceptual model, and then defining its basic operations, one can systematically develop transformation rules for that particular language. These rules can then be used to automate model transformation, in the context of model-driven engineering. Moreover, by defining transformation rules based on a common set of basic operations, the integration of different architecture evolution approaches can be facilitated. It is worth noting that our approach is not intended to replace current approaches for architectural evolution, but instead to empower them by facilitating the use of model transformations on the diverse set of modeling languages in use.

One of the limitations of our approach is that it does not consider architectural evolution as a whole, focusing solely on the modification of models. In future works we intend to explore the use of triggers to initiate the modification of the architectural model, as well as the selection of which modification to perform [20]. Additionally, we intend to automate the creation of the basic transformation rules for a given language, based on its metamodel. Moreover, we will investigate how this approach can be used in connection with modeling languages other than architectural ones.

REFERENCES

[1] Allen, R., Douence, R., Garlan, D. Specifying and Analyzing Dynamic Software Architectures. Proc. of 1998 Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal, March, 1998.

[2] Bachmann, F., Bass, L., Clements, P., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J. Documenting Software Architecture: Documenting Behavior. CMU/SEI-2002-TN-001, January 2002.

[3] Balogh, A., Varró, D. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. ACM SAC, pp. 1280-1287, France, 2006.

[4] Baresi, L., Heckel, R., Thöne, S., Varró, D. Modeling and validation of service-oriented architectures: application vs. style. SIGSOFT Softw. Eng. Notes 28, 5 , pp. 68-77, September 2003.

[5] Bradbury, Jeremy S.; Cordy, James R.; Dingel, Juergen and Wermelinger, Michel (2004). A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems, November 2004.

[6] Chapin, N., Hale, J., Khan, K., Ramil, J., Than, W.. Types of software evolution and software maintenance. Journal of software maintenance and evolution, pp. 3–30, 2001.

[7] Fernandez-Ramil, J., Perry, D., Madhavji, N.H. (eds.) Software Evolution and Feedback: Theory and Practice, Wiley, Chichester (2006).

[8] Franch, X., Grünbacher, P., Oriol, M., Burgstaller, B., Dhungana, D., López, L., Marco, J., Pimentel, J. Goal-driven Adaptation of Service-Based Systems from Runtime Monitoring Data. In: Proceedings of the 5th International IEEE Workshop on R equirements Engineering for Services (REFS), Munich, Germany, July 2011.

[9] Garlan D, Monroe R, Wile D (1997) Acme: An Architecture Description Interchange Language. In: Proceedings of the 1997 co nference of the Centre for Advanced Studies on Collaborative research (CASCON'97). Toronto, Canada.

[10] Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P., Merson, P. Documenting software architectures: views and beyond, 2nd ed. Addison-Wesley Professional, 2010.

[11] Garlan, D., Perry, D. Introduction to the Special Issue on Software Architecture. In: Journal IEEE Trans. on Soft. Eng. Vol. 21, Issue 4 (1995)

[12] Grau, G., Franch, X. On the Adequacy of *i** Models for Representing and Analyzing Software Architectures. Proceedings of the ER Workshops 2007, LNCS 4802, pp. 296-305 (2007).

[13] Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H. A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. GraBaTs'04.

[14] Magee, J., Kramer, J. Dynamic Structure in Software Architectures. Proceeding SIGSOFT '96 Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering 1996.

[15] Medvidovic, N. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, University of California, February 1996.

[16] O'Reilly, C., Morrow, P., Bustard, D. Lightweight prevention of architectural erosion. Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSW), pp. 59-64, September 2003.

[17] Oreizy, P., Medvidovic, N., Taylor, R. Architecture-Based Runtime Software Evolution. Proceedings of the International Conference on Software Engineering 1998 (ICSE'98). Kyoto, Japan, April 1998.

[18] Pimentel, J., Franch, X., & Castro, J. (2011). Measuring architectural adaptability in i* models. In Proceedings of the XIV Ibero-American Conference on Software Engineering, pp. 115-128, 2011.

[19] Pimentel, J., Lucena, M., Castro, J., Silva, C., Santos, E., Alencar, F. Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach. In: Requirements Engineering Journal, published online, 2011.

[20] Pimentel, J.; Santos, E.; Castro, J. Conditions for ignoring failures based on a requirements model. In: Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE). p. 48-53, 2010.