Guiding architectural restructuring through architectural styles

Dalila Tamzalit
Nantes University, LINA - CNRS UMR 6241, France
Dalila.Tamzalit@univ-nantes.fr

Tom Mens
University of Mons – UMONS, Place du Parc 20, 7000 Mons, Belgique
tom.mens@umons.ac.be

Abstract

Software architectures constitute one of the main artefacts of software-intensive system development. They outline the essential components and interconnections of a software system at a high level of abstraction, ignoring unnecessary details. How to address the evolution of software architectures, however, is still an important topic of current research. In this article, we use UML 2 as architectural description language notation and formalise it with graph transformation, with a proof-of-concept implemented in the AGG tool. We use this formalisation to express and reason about architectural evolutions that introduce architectural styles.

Key words: Software architecture - Design Evolution - Architectural style - Restructuring - Graph transformation.

1 Introduction

Software architectures describe the high-level structure of software-intensive systems in terms of components and interactions between them [1, 3, 22]. They have become accepted in software development and a common ontology was standardised by [11]: an architecture is “the fundamental organization of a system embodied in its components, their relationships ... and the principles guiding its design and evolution.”

How to specify, guide and control the evolution of an architecture remains an important track of current research. Complex and growing software systems often need to undergo large-scale restructurings. For instance, legacy systems need to be restructured into layered applications or deployed in terms of clients and servers. Their global architecture needs to become compliant to the specificities of these families of systems. This can be ensured by applying architectural styles [22] that they must conform to.

In this article, we propose to evolve architectures to conform to a particular architectural style by applying architectural restructurings. We provide formal specification and execution of such evolutions by leaning on graph transformations. We use UML 2 and present a case study based on a proof-of-concept implemented in AGG [23].

2 Evolving software architecture descriptions

Software architectures facilitate the understanding of software-intensive systems they describe and help to improve their maintenance and evolution. There are several ways to describe software architectures and the most frequently used description is the Component and Connector (C&C) view type [4, 22] (called structural viewpoint in [11]) where software architectures are described in terms of components, that propose or require services, and connected together via their interfaces.

Architecture Description Languages (ADLs)
have been proposed to describe software architectures following the C&C view type. A wide variety of ADLs have been proposed, like Rapide [17], ACME [9], Wright [1], C2SADEL [18], Darwin [13] or UML 2 [20].

As advised by the IEEE standard [11], ADLs should guide, in addition to the description, architectural evolution [19, 24]. Specifying the evolution of software architectures represents a promising way to control costs and challenges of software evolution [24], [15], avoiding modification of isolated parts of the system’s implementation in an ad hoc way.

Unfortunately, few ADLs provide support for architectural evolution. We can cite ACME, Ae- sop [8] or Rapide. Many other ADLs do not support any kind of architectural evolution (MetaH [25], Unicon [21], Darwin or UML 2 [20]). Among the desirable evolution properties, we are particulary interested in the ability to express and apply, by way of evolution, architectural styles [19]. To our knowledge, no ADL proposes to guide evolution through the application of architectural styles on a given architecture.

Among existing ADLs, UML 2 is of particular interest for two main reasons: it is the de facto standard software modeling language; and it provides a useful bridge between the high-level architectural specification and the lower-level detailed design models, and ultimately to source code. Unfortunately, UML is informal and ambiguous ADL [4], proposes no evolution policy and does not support architectural styles.

In this article, we will extend and formalise the UML component diagram notation as architectural description language [12] because of its concepts of component, connector, interface and port. We propose to enrich this ADL with support for architectural styles and explicit evolution towards them through architectural restructurings.

3 Supporting architectural styles with architectural restructurings

Together with the notion of software architectures, the concept of Architectural Style emerged as a disciplined mechanism to guide and constrain the use of software architectures [8, 22]. Following Fielding [6], “an architectural style is a co-ordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style”. These architectural constraints enable the automated analysis of architectures that conform to the architectural style [8].

Consider the Client-Server (C&S) style: it generally proposes and connects two component types: Server and Client. A server component offers a set of services, and listens for requests for those services from its client(s). We are confronted with the following questions: how can we evolve a given software architecture to make it compliant to this architectural style? How can we guarantee architectural well-formedness? How can we specify, automate and reuse architectural evolution? These questions are crucial for complex software systems, like legacy-systems, that often need to undergo large-scale restructurings.

We are convinced that architectural styles are a promising way to guide architectural evolution. By imposing constraints on the family of architectures they describe, they can constrain the evolution of an architecture towards this family. One specific way to face such evolution that is of particular interest to us is architectural restructuring. Following [14], we define architectural restructuring as the process of transforming the logical structure of an existing software architecture to improve specific attributes and by preserving its external services [10].

We will study how to formally specify architectural restructurings to apply an architectural style to an architecture without modifying existing services of the architecture.

4 Case study

We present the monolithic architecture of an Internet shop application EShop (Figure 1) that needs to evolve to a client-server architecture. The architecture contains three components, Product, Customer and Order, and connectors.

We only explain the most intricate component: Customer has seven ports: p9 to p15. Each port has one interface, representing a service that is provided or required. There are four
connections: OpenOrder, Authenticate, CreateCustomer and Bill. Each one connects a provided and a required interfaces of components. Some component interfaces, such as UserDetails are connexion-free and represent connection points with the environment.

Connectors are used to express dependencies between components. Because internal dependencies among interfaces of the same component are also important for architectural restructurings, we model them as well, like illustrated in Figure 1 as dashed oriented arrows between interfaces inside each component. For example, Product component has interface SelectProduct on port p2 that depends on the interface ViewProduct on p1.

The C&S style is introduced by applying architectural restructurings that ultimately leads to the final architecture in Figure 2: dependencies (expressed through connectors and internal dependencies) between services (interfaces) might be rearranged but they should be preserved at the end. The application of an architectural style is defined as a set of automatic transformations achieved by the framework, mainly restructurings, and a set of manual changes asked by the user:

1. The framework creates and appoints one server and n clients (n ≥ 1). For our EShop architecture, n = 1: EShop_Server server component and EShop_Client client component are created.

2. The user selects existing components to be moved into a client or into the server: move the Order component in EShop_Server, and the Product and Customer components in EShop_Client.

3. The framework moves the selected components in the server and client(s) and transforms all connectors and dependencies consistently. The EShop architecture obtained with gray internal components in Figure 2 is the result of these first three steps. Please note that the intermediate state, after these three steps, of the Product component is to be moved in EShop_Client before being splitted in two components (like explained hereafter).

4. If needed, the user manually triggers further desired changes. In the example, the user wishes to have a Product component in both EShop_Server and EShop_Client. It is thus necessary to split the Product component in two components, Product (that allows the user to view and select products) and Product_Server (that encapsulates product information in the server). The user also wishes to provide the Cancel service from the EShop_Client by delegating it from the EShop_Server.

5. The framework automatically executes the requested change and checks the architecture according to the ADL and the C&S style.

This process is visualised as a UML activity diagram in Figure 3. Each of the five steps is indicated by a dashed rounded rectangle. In addition, the activity diagram is decomposed into two phases (separated by the dashed horizontal line): steps 1 to 3 represent the mandatory first phase conforming the C&S architectural style. Phase 2 is optional and allows to execute specific changes. It leads to the final architecture of Figure 2.

Applying an architectural style on a given architecture is defined as an application of several architectural restructuring transformations. We have identified the following restructurings: Move Interface, Move Port, Move Service from a component to another, Split Component into two or more components, Merge Components into one component, Move in Component as a subcomponent of another and Move out Component from its containing component. In the previous example, we applied several of these restructurings.

During step 3, Move in Component was used to move components into EShop_Client or EShop_Server, automatically followed by restructuring Delegate Component Interface to create necessary delegations, dependencies and connectors using primitive restructurings Move Interface, Move Port and Move Service (coupled with the creation of corresponding ports and interfaces on EShop_Client and EShop_Server).

During step 5, the Split component restructuring was applied to achieve specific changes requested by the user: splitting
the Product component (belonging to EShop_Client) in two components Product and Product_Server. The latter one is subsequently moved out the EShop_Client and moved in the EShop_Server.

5 Formalising Architectural Description

A prerequisite for automating support for architectural evolution is the ability to specify architecture descriptions, architectural styles and architectural transformations precisely and unambiguously. For this purpose, we propose to use graph transformation theory. The analogy between software architecture restructuring and graph transformation is quite natural: a software architecture can be expressed as a graph containing a set of components interrelated by connectors. Graph transformations allow us to express the evolution of these architectural graphs in a precise way. In addition, it enables formal analysis and reasoning about architectural evolutions.

In this section we will formally specify the ADL, architectural styles and architectural restructurings to introduce these styles. To do this, we will use graph transformations specified in AGG, a Java-based graph transformation tool [23]. In section 7 we will exploit the formal analysis mechanisms of AGG to reason about architectural restructurings and their relation to the architectural styles.

In order to formally specify the ADL, we need to express its metamodel as a type graph [5]. This type graph, shown in Figure 4, specifies how to create well-formed architectures (compliant to the ADL). Any well-formed architecture can be represented as a graph that conforms to this type graph.

All architectural concepts we want to reason about (e.g., Component, Port, Interface) are represented as a graph containing a set of components interrelated by connectors. Graph transformations allow us to express the evolution of these architectural graphs in a precise way. In addition, it enables formal analysis and reasoning about architectural evolutions.
Figure 2. Resulting client-server EShop architecture.

represented by a node type in the type graph. The associations or relations between the architectural concepts (e.g., dependencies, connectors) are represented by edge types. Similar to the UML metamodel specification, the type graph imposes (lower and upper) multiplicity constraints on edge types. Node and edge types may contain additional attributes, and one can use a subtyping relationship between nodes.

In the type graph, we represented the concept of provided and required interface, respectively, by node types ProvInt and ReqInt that are both concrete subtypes of their common abstract supertype Interface. The Interface type contains an attribute $n$ of type String to represent the name of the interface. An Interface must be attached to a Port. An Interface can contain one or more Operations. A Port should always be connected to exactly one Component, and a Component may have any number of Ports (even zero).

In UML 2, the notion of connector is not a first-class entity but a link. It is here specified by edge type connectAssembly connecting node type ReqInt to node type ProvInt. The delegate edge type between two interfaces of the same type represents the delegation\(^3\) relation between an interface of a component and an interface of one of its subcomponents. Internal dependencies are represented by edge type depends on node type Interface.

Because the type graph does not allow us to express all constraints imposed by the ADL, we added additional graph invariants, some of which are listed below:

- a depends-link is only allowed between interfaces belonging to a different port of the same component.
- a connectAssembly-link is only allowed between interfaces of a different type (between a required and a provided interface) and belonging to different components that are not connected by a contains-link.
- a delegate-link is only allowed between interfaces of the same type belonging to different components that are connected by a contains-link, i.e. between a component and one of its subcomponents.

An architectural description can now be specified formally as a graph conforming to the type graph together with all of its graph invariants. An example is given in Figure 5\(^4\). It shows a graph representing the abstract syntax of the EShop architecture. AGG can automatically verify that this graph conforms to its type graph and that all imposed graph invariants are satisfied.

\(^3\)Generally called ‘binding’ in software architectures.

\(^4\)For reasons of clarity, Port names, Service nodes and Operation nodes are not shown, and some Interface names have been abbreviated.
To formalise an architectural style, we proceed in exactly the same way as for formalising the ADL: we extend the type graph and add extra graph invariants that express the additional constraints imposed by the architectural style. As an example, the C&S architectural style requires us to add two new Component subtypes to the type graph: a Client node type and a Server node type. This is shown in Figure 6. The node multiplicities state that a C&S-style architecture should always have exactly one Server and at least one Client.

In addition, we need to add two extra graph invariants that further constrain the type graph:

- A Client component must always be connected to Server via a connectAssembly-link. This graph invariant is shown in Figure 7.
- Only Client and Server are allowed as top-level components. This constraint is specified similar to the previous one.

6 Formalising Architectural Evolution

In order to specify architectural evolution we need to use the notion of a graph transformation. It takes a graph as input and produces another
Figure 5. A graph representing the abstract syntax of the architecture of Figure 1.

Figure 6. Formalisation of the C&S architectural style.

Figure 7. Connection between a client and the Server.

Figure 8. Three transformation rules to move a component into the server.

Graph as output. It is specified by means of a graph transformation rule that must conform to the type graph and all graph constraints.

Figure 8 shows three examples of graph transformation rules that formalise activities of Figure 3: Create Server, Move Component To Server and Delegate Provided Interface to Server. In general, the specification of a graph transformation rule is composed of three different parts (displayed from left to right): a number of optional negative application conditions (NAC), a left-hand side (LHS) and a right-hand side (RHS).

Applying the transformation proceeds as follows:

1. An occurrence (or “match”) of the LHS needs to be found in the host graph (for example, the one of figure 5). If more than one match is found, the user selects the desired match or the tool chooses a match non-deterministically.

2. If successful, the NAC verifies that cer-
tain “forbidden constructs” do not appear in the match. In rule CreateServer of Figure 8, NAC NoServer states that a server cannot be created if there is already one. NAC NoContainment of rule MoveComponentToServer states that the component to be moved is not allowed to be contained in any other component.

3. If the NAC is satisfied, the transformation rule is applied by “replacing” the match corresponding to the LHS in the host graph by its RHS. Identical numbers in LHS and RHS are used to identify nodes and edges that are preserved by the transformation. Nodes or edges only appearing in the RHS are newly added; nodes or edges only appearing in the LHS are removed by the transformation.

Section 5 formalised the structure of the C&S architectural style. Its introduction can be formalised as the application of a sequence of architectural restructurings expressed as graph transformation rules. The rule sequence below specifies the order in which the transformation rules of Figure 8 must be applied to move components into the server:

CreateServer; (MoveComponentToServer)*;
(DelegateProvInterfaceToServer)*;
(DelegateReqInterfaceToServer)*

The ; symbol is used to specify the order in which to apply the rules, whereas the * is used to specify a repetition of a rule (or even a subsequence), by applying it as long as a new match can be found in the graph.

Such rule sequences can be used to formalise the activity diagram specification that represents the application of the C&S architectural style in Figure 3. An activity diagram represents a set of possible execution paths. Each such path can be expressed by a rule sequence. As such, the set of rule sequences represents all possible execution scenarios represented by the activity diagram. The advantage is that rule sequences are amenable to formal analysis.

7 Analysing Architectural Evolution

A distinct advantage of formalising architectural styles and architectural evolution through graph transformation is the fact that we can carry out an automated form of formal analysis.

A first, and obvious, kind of support is the verification of whether a given architecture is well-formed (i.e., it conforms to the ADL specification), and whether the architecture conforms to an architectural style. Such verification is directly supported by AGG: if suffices to check that a graph conforms to its type graph, as well as all specified graph constraints. This checking was successfully done on the EShop architecture.

A second way to analyse architectural evolution is through AGG’s built-in support for critical pair analysis (CPA) between graph transformation rules. Such analysis enables the detection of parallel conflicts and sequential dependencies between pairs of transformation rules $R_1$ and $R_2$.

As an example, consider step 1 of the activity diagram of Figure 3. Starting from an initial software architecture, one can execute two parallel sequences of activities (step 2). The first sequence starts with Create Server and is followed by Move Component in Server and Delegate Component Interface to Server (step 3). The second sequence is similar but for Client instead of Server.

We can formally verify that the way of expressing the order between the different activities actually makes sense. To do so, we represent each activity as a graph transformation rule, as shown partly in Figure 8. Next, we run the critical pair analysis to detect parallel conflicts and sequential dependencies between pairs of rules. The result of this automated analysis is depicted in Figure 9 where dotted arcs between transformation rules represent sequential dependencies and solid arcs represent parallel conflicts.

The analysis corresponds to our intention: Create Client and Create Server are parallel independent, so they can be applied in parallel without any problem. Create Server is in parallel conflict with itself because only one server can be introduced into the C&S architecture. Move Com-
ponent to Client and Move Component to Server have a potential parallel conflict if one tries to move the same component in the client and in the server. This is why, in the activity diagram of Figure 3, we have inserted an intermediate activity select component to be moved in server (resp. client) where the user intervenes to decide which components to move in the server (resp. client). Move Component to Server (resp. Client) is in conflict with itself because one cannot move the same component twice. We also find all expected sequential dependencies: Delegate (Provided or Required) Interface to Server causally depends on Move Component to Server that causally depends on Create Server.

Thanks to this automated formal analysis, the process we specified in Figure 3 for applying the C&S architectural style is well-defined. Even better, the formal analysis allowed us to identify some constraints that we forgot to specify in the C&S style: while analysing all sequential dependencies, we found a number of false positives (i.e., dependencies that did not correspond to our intuition) that represented a situation where the Server component was contained in the Client component (or vice versa). Clearly, such a situation should be prohibited, so we added an extra constraint in the architectural style formalisation to reflect this.

8 Related Work

A variety of related work focuses on the individual topics that form part of our architectural evolution framework (i.e., architectural style patterns, formal architectural specifications, graph transformation, architectural restructuring). Our contribution is that we combine all of these topics together by studying the use of architectural styles and restructurings to guide software architecture evolution in a formal way using graph transformation.

Le Metayer [16] uses graph transformation theory to describe software architecture and architectural style as graph grammars. Other related work has been carried out to enrich and enforce the ADL UML 2 by extending it or combining it with existing formal ADLs (e.g., Acme, Wright) to ensure formal architectural descriptions and specifications like [2]. Existing ADLs do not seem to provide any technique to specify architectural style changes when an architecture evolves.

The idea of considering architectural styles as typical “evolution patterns” was first introduced by [15] that proposes the concept of Evolution Style as an architectural evolution pattern. Garlan [7] defines an evolution style as a set of evolution paths among different kinds of systems (from a simple system to a web-service architecture).

9 Conclusion and perspectives

In this article, we explored the relation between software architecture evolution and architectural styles. In particular, we focused on how to evolve a software architecture by applying an architectural style to it. We proposed to use architectural restructurings to specify such evolutions, unrolled on a case study involving UML 2 ADL. We used graph transformation theory as a formal foundation for specifying the ADL and the architectural style, as well as to specify how to evolve architectures in terms of architectural restructurings that satisfy the consistency constraints imposed by the ADL and the architectural style. We defined, applied and formally analysed sequences of architectural restructurings.

The formal specification and analysis of architectural styles and architectural restructurings is also beneficial for reuse purposes. Architectural styles represent structural patterns that can be (re)used and adapted in many different contexts, and that can form the basis for other architectural styles. Similarly, architectural restructurings can be used and adapted in many different situations. How to manage this reuse capability in...
presence of continuously evolving architectures remains a topic of future work.

Acknowledgments. This work was partially supported by University of Nantes, France and by the Belgian research project AUWB-08/12-UMH19 “Model-Driven Software Evolution”, Action de Recherche Concertée financed by the Ministère de la Communauté française - Direction générale de l’Enseignement non obligatoire et de la Recherche scientifique

References