A Runtime Architecture-Based Approach for the Dynamic Evolution of Distributed Component-Based Systems

Zhou, Y
A Runtime Architecture-Based Approach for the Dynamic Evolution of Distributed Component-Based Systems

Abstract

Dynamic evolution of distributed component-based systems (DCS) is an important task in software engineering. Several challenges are posed in this process. For example, how to preserve consistency during evolution and how to reflect the abstract evolution specification in the concrete reconfiguration implementation. Having observed the generality of software architecture, researchers have proposed various architectural description languages (ADLs), enabling evolution techniques, etc. to investigate the problem. These approaches typically employ the formal semantics of dynamic ADLs at the incremental levels of refinement in the design phase or the explicit maintenance of software architecture at runtime. However, different ADLs usually address different concerns and the lack of runtime support for the causal relation between ADLs and the running system easily leads to the mismatch between them, thus inevitably sacrifices their usability. We propose an approach based on a runtime architecture which is visually generated from an attributed type graph meta-model, exists through the lifecycle of DCS, establishes the causal relation between architectural topology and system configuration, and directs the dynamic evolution.
A Runtime Architecture-Based Approach for the Dynamic Evolution of Distributed Component-Based Systems

Yu Zhou
State Key Lab. for Novel Software Technology, Nanjing University, China
s.e.a.l, Department of Informatics, University of Zurich, Switzerland
zhou@ifi.uzh.ch

ABSTRACT
Dynamic evolution of distributed component-based systems (DCS) is an important task in software engineering. Several challenges are posed in this process. For example, how to preserve consistency during evolution and how to reflect the abstract evolution specification in the concrete reconfiguration implementation. Having observed the generality of software architecture, researchers have proposed various architectural description languages (ADLs), enabling evolution techniques, etc. to investigate the problem. These approaches typically employ the formal semantics of dynamic ADLs at the incremental levels of refinement in the design phase or the explicit maintenance of software architecture at runtime. However, different ADLs usually address different concerns and the lack of runtime support for the causal relation between ADLs and the running system easily leads to the mismatch between them, thus inevitably sacrifices their usability. We propose an approach based on a runtime architecture which is visually generated from an attributed type graph meta-model, exists through the lifecycle of DCS, establishes the causal relation between architectural topology and system configuration, and directs the dynamic evolution.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Design Tools and Techniques; D.2.12 [Software Engineering]: Design Principles and Methods

General Terms
Design, Theory, Management

Keywords
Software Architecture, Dynamic Evolution

1. MOTIVATION
Dynamic evolution of distributed component-based system (DCS), which is also referred to as adaptation, has been intensively studied in software engineering. With the development of software technology, the DCS ranges from traditional component systems [11] to the nowadays open service-centric systems [8]. Despite the variety of the underlying implementation details, they share a common set of characteristics: high-level separation of computation and coordination, loosely coupling and high autonomous entities, protocol-support interactions, etc. These characteristics can be well captured by the concept of software architecture (SA) which focuses on the constituent entities, their interactions, patterns of composition, and the global constraints [10]. In consideration of the increasing openness and autonomy of the distributed components, dynamic evolution of such systems mainly denotes the addition, removal, substitution, update and the reconfiguration of the constituent entities. SA offers a promising approach to meet the evolution requirements in this level, which is also illustrated by the proliferation of the corresponding works [7].

Consider a simple distributed service-based system: There are several self-contained ticket-booking services scattered in the Internet. They sell different kinds of tickets, such as bus tickets, train tickets, plane tickets, etc. The third-party service assembler wants to integrate these separate services and offer a single entry point to get all the ticket options for clients’ travel routes. To compose these services, another component which interacts with users and invokes separate services is needed. Due to the inherent dynamism of the environment, one component, for example, the bus ticket service is unavailable and needs to be detached from the system. This kind of scenario is quite common in the real world and subject to the coarse-grained, architectural evolution. Generally, the service assembler has to shut down the system and reconfigure the composition logic to evolve the system. This off-line evolution inevitably causes considerable time of system’s unavailability. Imagine that, in several hours, the bus ticket service is recovered, and the system has to be reconfigured again which is rather annoying. As these service entities are autonomous and do not know the existence of others, the central design patterns like publish-subscribe fail to address this problem. To facilitate this process, dynamic addition or removal of constituent entities, i.e., dynamic evolution is highly desirable.

Despite the simplicity of the scenario, it can reflect essential requirements and challenging issues of dynamic evolution. Among the many, we list several requirements: (1) Global constraints should be specified and maintained through

---

\( ^1 \)Note that the service assembler is not necessarily the service programmer.
the evolution process to ensure consistency. (2) An enabling mechanism should be provided to conduct the evolution. (3) The evolution steps should be traceable.

Current research effort on architecture support for dynamic evolution falls in two categories, i.e., theoretical, and technical, respectively. The former exploits algebra notions or formal language to deduce the structure specification and impose the abstract constraints and evolution rules, while the latter investigates the enabling technology to explicitly instantiate and maintain the architectural model, or automatic generation of architecture style applied code skeleton. The advantages of each category are the disadvantages of the other: the specification is sound and traceable but difficult to implement, while the manual, ad-hoc implementation and evolution lack the formal basis, thus easily become inconsistent during evolution. They fall short to meet the requirements listed above.

This observation motivates us to combine the two categories to obtain the complementary advantages of both: using a formalism to derive the runtime architecture model, which is then instantiated and existing through the life cycle of the DCS. The contextual changes are acted upon the meta-model which in turn, is reflected on the runtime architecture model and then in turn, reflected on the running DCS. This process has a transition-closure property. As each step is complied with the configuration constraints and traceable, the evolved architecture preserves its consistency.

The remainder of the paper is organized as follows: The proposed solution is explained stepwise in Section 2 followed by the validation in Section 3. Related work and expected contributions are given in the Section 4 and 5.

2. PROPOSED SOLUTION

In this section, we further explain the proposed solution. To clarify the idea, we recall the scenario. The system conforms to a typical master/slave architecture style. The entry point component functions as the master, while the specific ticket components serve as slaves. In an abstract structural level, the systems need to dynamically evolve from 1 master with \( n \) slaves to 1 master with \( (n - 1) \) slaves. Meanwhile, the constraints, for example, that it must have at least one master node and one slave node, no dangling arcs and nodes exist, etc., must be preserved during evolution. In a concrete application level, the behavior that the entry point receives client requests, invokes tickets functions of the current attached components and returns the results should be the same after the reconfiguration.

Our proposal is to support the above kind of dynamic evolution without violation of the architectural constraints (consistency) by using the automatically generated runtime architecture model from a rigorous graph meta-model and establish the causal relation between specification and implementation without the loss of formal semantics.

To achieve the above vision, we propose the following 3-step approach.

1. Establish the mapping from an attributed type graph (ATG) meta-model to the runtime architecture model which is in the form of a class.
2. Runtime architecture instance generation and interaction with the DCS. This step also concerns the distribution

of the instance to the involved containers.

3. Reconfiguration of the graph meta-model and the generation, interaction of evolved architecture model.

Figure 1 sketches the principle process of our proposed solution. The reasons why we use ATG as a formalism basis mainly lies in three considerations: 1. Graph representation has an advantage of an intuitive visual correspondence to the common practice of box-line notation of SA; 2. The widely recognized importance of connector as a first-class entity can be well captured by the notion of an edge attribute in ATG; 3. By production, the graph rewrite techniques can be employed to describe the evolution and to check the consistency formally.

2.1 Establishing the Mapping from ATG Meta-Model to the Runtime Architecture Model

An ATG is an attributed graph integrated with a data signature algebra which consists of unary operations over the attribute sorts. Here we do not dive into the theoretical details, for interested readers we refer to [2]. We propose to map the constituent elements of ATG to different classes: graph topology information and data signature operations can be directly mapped to the corresponding properties and methods in the runtime architecture. The above information is mainly in the meta-level, and more static, similar to the template during the generation of a runtime architecture. Besides this information, a runtime architecture also reserves space tailored for needs from a specific component model of the application.

This step is illustrated in Figure 1 (Step 1). A system assembler composes the graph elements through a GUI according to the corresponding styles or patterns and binds the application-specific information to them. Constraints and rules can be specified separately in the graph grammar format. After successful checking, the graph meta-model will get mapped to the architecture model.

In our scenario, the application follows a master/slave architectural style. This style can be modeled as an ATG. The different components have different names, port categories (provide/require) of string type and port quantity of a natural number type. The connector which is modeled as a line has the attribute of invoking direction: in or out (or caller, callee roles). The data signature contains the basic

\( \text{CH} \) appeared in the figure stands for the involved components hosts; ‘RA’ stands for runtime architecture.

Figure 1: Proposed Solution Illustration

---

CH1 CH2
CHn CH3
CH1 CH2
CHn

Step 1
Step 2
Step 3

Graph Composition → App-Specific Info Binding
Constraints/Var Definition (Graph Grammar)
Grammar Checking → Architecture Class Mapping

2.1 Establishing the Mapping from ATG Meta-Model to the Runtime Architecture Model

An ATG is an attributed graph integrated with a data signature algebra which consists of unary operations over the attribute sorts. Here we do not dive into the theoretical details, for interested readers we refer to [2]. We propose to map the constituent elements of ATG to different classes: graph topology information and data signature operations can be directly mapped to the corresponding properties and methods in the runtime architecture. The above information is mainly in the meta-level, and more static, similar to the template during the generation of a runtime architecture. Besides this information, a runtime architecture also reserves space tailored for needs from a specific component model of the application.

This step is illustrated in Figure 1 (Step 1). A system assembler composes the graph elements through a GUI according to the corresponding styles or patterns and binds the application-specific information to them. Constraints and rules can be specified separately in the graph grammar format. After successful checking, the graph meta-model will get mapped to the architecture model.

In our scenario, the application follows a master/slave architectural style. This style can be modeled as an ATG. The different components have different names, port categories (provide/require) of string type and port quantity of a natural number type. The connector which is modeled as a line has the attribute of invoking direction: in or out (or caller, callee roles). The data signature contains the basic
2.2 Runtime Architecture Instance Generation and Interaction with the DCS

In the design phase, the assembler configures the ATG meta-model through an integrated graphic editor, including drawing nodes and edges, assigning attributes, defining the isomorphism relations. As introduced before, the runtime architecture also contains information about the application, such as the component location, component identification, connector identification, specific protocol, etc. This information is bound to the attributes of components and connectors of the graph model and will be mapped to the properties of the runtime architecture object. Afterwards, the graph meta-model is checked against the accompanied rule-set (written in attributed graph grammar). The accepted one would be mapped to the specific runtime architecture object as a typed attributed graph instance.

This generated object contains the topology information and application-specific information. That’s why it is termed runtime architecture model. The copies of the object will be distributed to the place where the involved components reside in a transaction. Then the architecture object will intercept the incoming invocation to the corresponding component interfaces, process it according to the protocols specified in the connector. This can be realized through reflection and introspection mechanism commonly provided in programming languages.

This step is concisely illustrated in Figure 1 (Step 2). The runtime architecture object reinterprets the references after instantiation and dispatching.

In our scenario, the connector of the architecture object intercepts the ticket searching method invocation from master component and forwards the parameters to different attached slaves and then sends back the final result to the master component. As the connector has the architecture information, it knows the number of slave components and their interaction protocol. The above process is transparent to the involved components. The master component invokes as if only one slave component exists. This can be achieved by adding a layer to the component container using reflection. In our preliminary work [6,14], we added an interceptor layer to the JBoss container, and all the invocations are captured and processed in the architecture object and then forwarded to the corresponding component interfaces.

2.3 Reconfiguration of the Graph Meta-Model, Generation and Interaction of the Evolved Architecture Model

In this step, users reconfigure the graph meta-model according to the contextual or requirement changes. The ATG defines a set of graphs which have the same kinds of nodes and arcs, the topology constraints, and attributes constraints of the nodes and arcs. The evolved or the reconfigured graph which is an instance of the ATG can be regarded consistent. To ensure consistency, the validity of the reconfiguration must be checked against the rule set of graph transformation. The rule set can be defined as \( R = \{ r_1, r_2, \ldots, r_n \} \), in which \( r_i = (L \xrightarrow{q_i} R, A(r_i)) \), \( q : L \rightarrow R \) is attributed graph morphism, and \( A(r) \) is the application condition of \( r_i \).

Based on graph parsing, the above checking can be reduced to a reachability problem. For more details, we refer to [1]. If the checking result is negative, the reconfiguration will abort. Otherwise a new architecture object is generated and distributed to the involved components in a transaction. The new architecture replaces the outdated one and directs the interaction of the evolved running system.

This step is concisely illustrated in Figure 1 (Step 3). Before reinterpreting the reference of involved components, reconfiguration must follow the successful checking against the rules and constraints.

In our scenario, the reconfiguration includes the removal of the disconnected slave component. The new architecture is still an instance of the master/slave style. Thus the new architecture object is generated, cloned and distributed to the involved hosts. Now incoming retrieval requests will be delivered to the two available services, the results are sent back to the master component which forwards them to the clients. Like Step 2, this process is transparent to the involved components, as all the necessary information is maintained by the architecture object.

3. VALIDATION

The validation of our work will be checked against the requirements listed in the first section.

In our previous work [6,14], we have implemented a middleware system supporting the development of architecture based self-adaptive systems as a proof of concept. We use the service component system based on EJB and Axis as an application testbed. The currently achieved progress is that it validates parts of the key ideas presented in this paper. Especially it proves the feasibility that the internal runtime architecture can interact with the running system and direct the evolution. Through the mechanism of reflection and introspection, this interaction is transparent to the involved components. To further support runtime architecture persistency, we propose using Hibernate to facilitate a high-level object data retrieval/storage.

Currently, we have included the mapping from ATG to architecture model in our ongoing NG (next-generation) project. The resulting software will be a middleware which allows defining the isomorphism rules, consistency checking, attribute binding, runtime architecture generation and manipulation. To facilitate the consistency checking, we prepare to use AGG [12] as the underlying engine. The fact that AGG contains a set of validation tools and has a tight relation with Java quite suits our needs. Checking the evolved attributed graph to be an instance of the original ATG is a graph case of formally called ‘membership problem’. The challenging issue is the time cost when the number of graph elements grows. However, in a context-sensitive graph grammar, this problem can be transformed to the reachability checking, i.e., from the start graph, by production, checking whether the derived graph is isomorphic to the target graph. As this step can be separately evaluated, we will write transformation rules and configuration constraints for
a selected set of known architectural styles. We will simulate reconfiguration patterns and measure the temporal cost.

After the above two steps, we will conduct case studies to further validate and evaluate the whole approach. We plan to apply the approach to applications similar to the scenario mentioned at the beginning, but not necessarily with the same style constraints. Generally, several demonstrations will be developed in different component technologies. For each application, we will test by at least two reconfiguration attempts, one is graph grammar-complied; the other is grammar violated to check the ability of our system.

4. RELATED WORK

Various researchers have investigated the architectural support for the evolution of software systems in the literature. In the theoretical aspect, Le Metayer [5] first proposed to use graph grammar to describe software architecture styles and reconfiguration, and Baresi et al. [1] also provided a style-based refinement model of dynamic software architecture. Recently, the notion of component architecture “modes” [3] is proposed as a promising concept to model specific domains. These models leveraged the visibility of graph grammar or formal domain semantics but had less corresponding runtime support in the software system. In the technical aspect, Oreizy et al. [9] first presented a C2-style approach supporting architecture-based runtime evolution. Wang and Mei [13] also proposed a component-based approach to online evolution. Despite the fact that these techniques facilitate the dynamic evolution process, they fall short in the consistency control during evolution.

5. EXPECTED CONTRIBUTIONS

In this paper, we presented an architecture-based approach to the runtime evolution. It will bridge the gap between the architectural specification and the running systems. In [4], Kramer and Magee present a three-layer reference model as a context for the research on self-managed systems. We believe our work will be a complement to this model. The expected contributions include:

1. A graph-based notation offers a visually intuitive view to generate the architecture model without loss of formal semantics;
2. DCS constitutes a wide range of applications. The proposed approach has a broad application domain;
3. Evolution chain can be recorded through architecture object persistence technology. The proposed approach offers a higher traceability;
4. The change of the meta-model will be dynamically reflected on the evolution of the system, thus the causal relation is established.

6. ACKNOWLEDGMENTS

The research is funded by the 973 program of China (2002-CB312002), JSNSF (BK2006712) and the NSFC (60403014, 60736015, 60721002).

I am grateful to the China Scholarship Council for supporting me to conduct research as a joint PhD student at the University of Zurich.

Sincere thanks are given to my advisor at Nanjing University, Prof. Jian Lu, associate Prof. Xiaoxing Ma, my joint advisor at the University of Zurich, Prof. Harald Gall, and Beat Fluri.

7. REFERENCES


