Towards a Graph Grammar Based Verification Approach for Runtime Constrained Evolution of Service-Oriented Architectures *

Yongwang Zhao, Bingyang Zhao, Min Liu, Chunyang Hu, Dianfu Ma
Institute of Advanced Computing Technology
School of Computer Science and Engineering, Beihang University, Beijing, China
{zhaoyw,zhaoby,liumin,hucy}@act.buaa.edu.cn, dfma@buaa.edu.cn

Abstract

Service Oriented Architecture (SOA) is a new form of distributed software architecture. It promotes loose coupling, services distribution, dynamicity and agility. Runtime architecture of new generation service based system should be evolutionary for flexible application requirement, instability of composing service nodes and Internet environment. Modeling of runtime SOA and verifying consistency of evolution could improve system dependability and adaptability. The main contribution of this paper is a graph grammar based modeling and verification approach for constrained evolution of SOA at runtime. System specification described by SOA style and structural constraints and their satisfaction checking algorithms are proposed. We have implemented an initial constrained evolution verification tool that allow us to model runtime SOA, SOA style and constraints and verify consistency.

1 Introduction

A service-oriented architecture is essentially a collection of services which communicate with each other. The communication can be simply between two services or it can be multiple services coordinating an activity by orchestration or choreography[14]. WS-BPEL (Web Services Business Process Execution Language) [1] is typical language for orchestration and WS-CDL (Web Service Choreography Description Language) [10] for choreography. Future complex software for various application domains should rely on dynamic architectures to fulfill evolving-related requirements which WS-CDL and WS-BPEL do not support. For instance, a business system of purchasing goods among buyers, sellers, a broker and shippers presented in Fig.1.

This system permits Sellers and Shippers joining or leaving the system at runtime, Broker will transmit ReqQuote message to all Sellers when receiving it from Buyer and select the cheapest one for Buyer. Then Broker will also select the best Shipper by geographical location of Buyer, Seller and Shipper. For this requirement, because of statically modeling of architecture by WS-BPEL and WS-CDL, when new services joining the system, WS-BPEL do not support new instantiation of partnerLink and can not change the behavior of Broker dynamically to fit this situation. For WS-CDL, new instantiation of Participant do not be supported too. On the other hand, adaptive service-based systems should deal with instability of composing service nodes and Internet environment. Crashing of services or destroying of network connections may turn the running system into failure since the runtime architecture is inconsistent with system requirements. This dynamic architectural changing is most commonly known as run-time evolution or dynamism. These situations require modeling service oriented architectures and verifying consistency of evolution with system requirements.

In this paper, we proposed a graph grammar[15] based
verification approach for evolution of runtime service-oriented architecture. It represents the SOA using graph grammar and architectural style[13]. A graph expresses an architecture instance by nodes(services) and edges(channels) while production rules of a graph grammar define the SOA style which means the set of reachable architectures derived from initial graph by production rules. We also introduced topological constraints of runtime SOA. Constraints are expressed by graph morphisms mapping the nodes and edges of a graph to those of another one, preserving source and target of each edge. Constraints of SOA are properties of architectures that have to be preserved when evolving at runtime. We used topological constraints to express basic restrictions such as the existence or uniqueness of certain services and channels in SOA. To verify the architectural consistency, SOA style and topological constraints satisfaction checking algorithms were proposed. A demonstrated constrained evolution verification tool(CEVT) for service-based systems was implemented to support software development.

The paper is organized as follows. Some related works are discussed in Section 2. In Section 3, we propose a graph grammar based approach to model service oriented architectures and specify their styles and topological constraints. In Section 4, we describe our constrained evolution verification tools for service-based systems and Section 5 concludes.

2 Related Works

Architectural evolution in software can occur at design time, pre-execution time, or run-time. Dynamic software architectures are those that modify their architecture and enact the modifications at run-time [11]. Graph-based, process algebra and logic-based formalisms are three basic formal approaches to dynamic software architectures[4]. In particular, the graph-based approach allows for a natural way of describing styles and configurations and has been largely used for specifying architectures. Le Métayer[13] uses context-free graph grammars to represent architectural styles, graphs to represent an architectural instance, and standard rewriting rules to represent the reconfiguration. The COMMUNITY approach[18, 19] has a formal specification language based firmly in category theory with architectural reconfiguration specified using the double-pushout graph transformation. Brunia et al. in [5] use typed graph grammars to model dynamic architectures in which a type graph is used to express the architectural styles and typed graphs to architectural instances.

For modeling SOA, UML as a graph-based modeling language is adopted by many researches. Heckel et al.[8] propose a UML profile for dynamic service discovery in an SOA by providing stereotypes that specify the relationships among service implementations, service interfaces, and requirements. In [2], authors develop a semi-formal platform independent and an SOA-specific meta model to capture service architectures on various levels of abstraction in a model-driven service development process using UML. The reconfigurations for service publishing, querying and binding are captured by graph transformation rules [7]. This combination of meta modeling and graph transformation rules is suitable to a model-based development process for service-oriented systems.

Software patterns are proposed as a way to support the general understanding of the generic and stable aspects of SOA as an architectural style. A survey on SOA patterns can be found in [20]. On top of these patterns,[21] have described a pattern language that explains proven practices for process-oriented integration of services in an SOA. These patterns have been identified and validated using a broad foundation of industrial and open source systems, tools, and case studies.

3 Verification Approach

3.1 Modeling Runtime Service Oriented Architecture

Our evolution model considers services and their communication dependency represented as channels in our approach. A channel is directed and realizes a point of collaboration between services by specifying where and how information is exchanged. Fig.2 shows the main concepts of our model.

A service has different ports and each of them presents a role of services in the system. A channel connects two ports of different services, and the source port can initiate interactions to target port. Channels limit the direction of interactions, but not the times. We formalized SOA based on edNCE grammars(Graph Grammars with Neighborhood Controlled Embedding and Dynamic Edge Relabelling) [7].

Definition 1 (SOA Instance). An SOA instance is an architecture instance of an SOA system at runtime. Let \( \Sigma \) be an alphabet of service labels, \( K \) be an alphabet
of port labels, $\Gamma$ be an alphabet of channel labels and $\Pi$ be an alphabet of service-port relation labels. An SOA instance over $\Sigma$, $K$, $\Gamma$ and $\Pi$ is a tuple $\text{SOAI} = (\text{Service}, \text{Port}, SP, \text{Channel}, \lambda, \kappa)$, where Service is finite set of services, Port is finite set of ports of services, $SP \subseteq \{(s, \gamma, p) | s \in \text{Service}, p \in \text{Port}, \gamma \in \Pi\}$ is set of relations between services and their ports, $\text{Channel} \subseteq \{(p, \varphi, p') | p, p' \in \text{Port}, \varphi \in \Gamma\}$ is the set of channels, $\lambda : \text{Service} \rightarrow \Sigma$ is the service labeling function, and $\kappa : \text{Port} \rightarrow K$ is the port labeling function.

An SOA instance defines the structure of a service system including services, their ports and channels between ports. But for composition, a composed service should be reused by other services. Actually, a composed service should expose its ports so that other services may communicate with it by channels. Therefore an SOA instance of a composed service may have external ports and channels for being composed further. For instance, a WS-BPEL based process is composed of some web services and itself is also described by a WSDL document for reusing. We define embedding SOA instance to describe a composed service.

**Definition 2 (Embedding SOA Instance).** An embedding SOA instance is a kind of SOA instance with external ports and channels. It is a tuple $\text{SOAI} = (\text{SOAI}, C)$, where $\text{SOAI}$ is an SOA instance and $C \subseteq K \times \Gamma \times \Gamma \times \text{Port} \times \{\text{in}, \text{out}\}$ is connection relation of $(\text{SOAI}, C)$. Each element $(\sigma, \beta, t, x, d) \in C$ with $\sigma \in K, \beta \in \Gamma, t \in \Gamma, x \in \text{Port}, d \in \{\text{in}, \text{out}\}$ is a connection instruction of $(\text{SOAI}, C)$.

To improve readability, a connection instruction $(\sigma, \beta, t, x, d)$ will be written as $(\sigma, \beta/t, x, d)$. For example, $(\sigma, \beta/t, x, \text{in}) \in C$ means that if there was a $\beta$-labelled channel between port $v$ and $w$ with label $\sigma$, where $v$ is to be substituted by $(D, t)$, then the embedding process will establish a $t$-labelled channel from $w$ and $x$.

Fig.3 shows a case described in section 1 of the composed services which has external ports and channels. Buyers can communicate with. The formal description is $\text{SOAI}_1 = (\text{SOAI}_1 = (\text{Service}, \text{Port}, SP, \text{Channel}, \lambda, \kappa), C)$, where $\lambda$ Service = \{ BrokerService, ShipperService, SellerService \}, Port = \{ P_{brk}, P_{sell}, P_{ship} \}, SP = \{ (\text{BrokerService}, \theta_1, P_{brk}), (\text{SellerService}, \theta_2, P_{sell}), (\text{ShipperService}, \theta_3, P_{ship}) \}, \lambda(\text{BrokerService}) = A, \lambda(\text{ShipperService}) = C, \lambda(\text{SellerService}) = D, \kappa(P_{brk}) = m, \kappa(P_{sell}) = k, \kappa(P_{ship}) = l, C = \{(n, \delta_1/\gamma_1, P_{brk}, \text{in}), (\sigma, \delta_2/\gamma_2, P_{ship}, \text{out})\}.

### 3.2 Service Oriented Architectural Style

An SOA instances describe an architecture instance at runtime, while an architecture style is a class of architectures exhibiting a common shape[13]. An architectural style consists of a set of architectural elements and operations called *productions* which define the well-formed compositions of architectures. It can be seen as the “type” that the architecture must have at run time, that is to say the possible interconnections between its individual services and two services can communicate only through the channels specified by the style. Technically, architecture styles are defined as graph grammars. We define the SOA style as follow:

**Definition 3 (SOA Style).** An SOA Style is tuple $\text{SOA}_S = (\Sigma, \Sigma_\Delta, K, K_\Delta, \Gamma, \Gamma_\Delta, \Pi, \Pi_\Delta, P, S)$ where $\Sigma$ is the alphabet of service labels, $\Sigma_\Delta \subseteq \Sigma$ is the alphabet of terminal service labels, $K$ is the alphabet of port labels, $K_\Delta \subseteq K$ is the alphabet of the terminal port labels, $\Gamma$ is the alphabet of channel labels, $\Gamma_\Delta \subseteq \Gamma$ is the alphabet of final channel labels, $\Pi$ is the alphabet of relation labels, $\Pi_\Delta \subseteq \Pi$ is the alphabet of final relation labels, $P$ is the finite set of productions, and $S \in \Sigma - \Sigma_\Delta$ is the initial nonterminal.

A production is of the form $X \rightarrow \text{SOAI}_C$, where $X \in \Sigma - \Sigma_\Delta$ is nonterminal service label and $(\text{SOAI}_C, C)$ is an embedding SOA instance. It means that the node $X$ in an SOA instance can be substituted by another instance $\text{SOAI}$ and $C$ shows the channels connection between source SOA instance and replacing one. We use styles of a box with a label on top of it to represent a production which means that the label node in an SOA instance can be replaced by the SOA instances defined in the box.

One application of a production on an SOA instance is defined Derivation Step. The application of production $p = X \rightarrow \text{SOAI}_2, C_2$ on SOA instance $\text{SOAI}_1$ is noted as $\text{SOAI}_1 \Rightarrow_p \text{SOAI}_2$. A sequence of derivation steps is defined as Derivation of SOA instances which is noted as

![Figure 3. The embedding SOA instance of the business system.](image-url)
Algorithm 1: Substitution of embedding SOA instance

1. $Port_3 = (Port_1 - v) \cup Port_2$
2. $Service_3 = Service_1 \cup Service_2$
3. $SP_3 = SP_1 \cup SP_2$
4. $Channel_3 = \{x, \gamma, y \in Channel_1 | x \neq v, y \neq v\} \cup Channel_2 \cup \{(w, \gamma, x) | \exists \beta \in \Gamma : (w, \beta, v) \in Channel_1, (\kappa_1(w), \beta, \gamma, x, in) \in C_2\}$
5. if $x \in Service_1$ then
   \[ \lambda_3(x) = \lambda_1(x) \]
6. else if $x \in Service_2$ then
   \[ \lambda_3(x) = \lambda_2(x) \]
7. if $x \in Port_1 - \{v\}$ then
   \[ \kappa_3(x) = \kappa_1(x) \]
8. else if $x \in Port_2$ then
   \[ \kappa_3(x) = \kappa_2(x) \]
9. $C_3 = \{(\sigma, \beta, \gamma, x, d) \in C_1 | x \neq v\} \cup \{(\sigma, \beta, \gamma, x, d) \exists \gamma \in \Gamma : (\sigma, \beta, \gamma, v, d) \in C_1, (\sigma, \gamma, \delta, x, d) \in C_2\}$

$SOA_0 \Rightarrow p_1 SOA_1 \Rightarrow p_2 ... \Rightarrow p_n SOA_n$. It is noted simply as $SOA_0 \Rightarrow \ast SOA_n$. $SOA_n$ is the result of application of derivations of SOA instances on $SOA_0$.

Reachable Set of SOA Style $SOA_S = (\Sigma, \Sigma_\Delta, K, K_\Delta, \Gamma, \Gamma_\Delta, \Pi, \Pi_\Delta, P, S)$: The set of SOA instances which can be derived from the initial nonterminal by application of productions in $P$.

$R = \{SOA_0 | \Rightarrow_{p*} SOA_1 \text{ and } (\forall p \in (p*)) \in P\}$

3.3 Style Checking on SOA Instances

For style checking, we first explain substitution of embedding SOA instance which means one service and its ports in instance $A$ is replaced by another instance $B$. Let $(SOA_1, C_1)$ and $(SOA_2, C_2)$ be two embedding SOA instances over the same alphabet, such that $SOA_1$ and $SOA_2$ are disjoint, and let $v$ be the replaced service of $SOA_1$. The substitution of $(SOA_2, C_2)$ for $v$ in $(SOA_1, C_1)$, denoted $(SOA_1, C_1)[v/(SOA_2, C_2)]$. Let $(SOAI_3, C_3) = (Service_3, Port_3, SP_3, Channel_3, \lambda_3, \kappa_3, C_3) = (SOA_1, C_1)[v/(SOA_2, C_2)]$

We show the computing method of $(SOAI_3, C_3)$ as algorithm 1.

The algorithm to verify whether an SOA instance belongs to reachable set of an SOA style is listed in Algorithm 2.

This algorithm takes an SOA instance $SOAI$, production rules set $PRs$ and initial nonterminal $S$ of the style as input parameters. For $SOAI$, we used reversed production on it by institution algorithm above, that means if the right graph matched on $SOAI$ it will be instituted by left graph of this production. This matching is a problem of subgraph isomorphism[16] explained in section 3.5 detailedly. If $SOAI$ reached $S$, it means that $SOAI$ could be generated by applications of rules in $ProductionRules$ from initial SOA $S$.

3.4 Topological Constraints of SOA Evolution

Constraints of SOA evolution are properties on SOA instances which have to be satisfied at runtime. An SOA style gives the reachable set of SOA instances when being reconfigured. Constraints are introduced to restrict this reachable set, because some reachable architecture is also not proper for different SOA applications. For instance, the amount of seller instances in the business system mentioned in section 1 can not be larger than one hundred. We use graph morphism to express constraints of SOA.

Graphs are related by graph morphisms, mapping the nodes and edges of a graph to those of another one, preserving source and target of each edge. Graphs together with graph morphisms form the category Graph. We use graph morphism to define the morphism of SOA instances.

Firstly, we add source and target function of channels and relations in SOA instances. $s_{sp} : SP \rightarrow Service, t_{sp} : SP \rightarrow Port, s_{ch} : Channel \rightarrow Port$ and $t_{ch} : Channel \rightarrow Port$ are source and target functions.

**Definition 4 (SOA Morphism).** Given two SOA instances $SOAI_1 = (Service_1, Port_1, SP_1, Channel_1, \lambda_1, \kappa_1) \in \{1, 2\}$ an SOA morphism $f : SOAI_1 \rightarrow SOAI_2$,

- $f = (f_s, f_p, f_{sp}, f_{ch})$ consists of four functions $f_s : Service_1 \rightarrow Service_2, f_p : Port_1 \rightarrow Port_2, f_{sp} : SP_1 \rightarrow SP_2$ and $f_{ch} : Channel_1 \rightarrow Channel_2$ that preserve the source and target functions, i.e. $s_{sp} \circ f_{sp} = f_s \circ s_{sp}, t_{sp} \circ f_{sp} = f_p \circ t_{sp}, s_{ch} \circ f_{ch} = f_p \circ s_{ch}$ and $t_{ch} \circ f_{ch} = f_p \circ t_{ch}$.
Next, we will consider the structural constraints of SOA. We first define the structural constraints as follow based on graph constraints[6]:

**Definition 5 (Structural Constraints).** Structural Constraints over SOA instance $P$ are defined inductively as follows: For an arbitrary SOA morphism $x: P \rightarrow C$, $\exists c$ is a basic constraint over $P$. For an arbitrary SOA morphism $x: P \rightarrow C$ and a constraint $c$ over $C$, $\forall (x,c)$ and $\exists (x,c)$ are conditional constraints over $P$. For constraints $c, c_i (i \in I)$ over $P$, true, false, $\neg c$, $\land_{i \in I} c_i$ and $\lor_{i \in I} c_i$ are boolean constraints over $P$.

To restrict the element count in SOA instances, counting constraints are introduced.

**Fact 1 (Counting Constraints).** The following properties of SOA instances can be expressed as counting constraints: For a given $n \in \mathbb{N}$, there exists a pt type port which has number of $\{=,<,>\} n$ cht type channels:

$$\exists (\top_{pt} \rightarrow \top_{pt}, \text{cht}_{\{=,<,>\}} n)$$

All pt type port has number of $\{=,<,>\} n$ cht type channels:

$$\forall (\top_{pt} \rightarrow \top_{pt}, \text{cht}_{\{=,<,>\}} n)$$

We will show some cases of constraints on SOA instances. The first case is that if there exists a channel from port$_1$ to port$_2$, there must exist another channel from port$_2$ to port$_1$. We can define the SOA morphism as $x: \text{port}$_1 \rightarrow \text{port}$_2 \rightarrow \text{port}$_1 \rightarrow \text{port}$_2 and constraint $c: \text{port}$_1 \rightarrow \text{port}$_1 \rightarrow \text{port}$_2 \rightarrow \text{port}$_2. An SOA instance $SOA I$ satisfies $\exists (x,c)$ means that there could be no channel between port$_1$ and port$_2$ or bidirectional channel. While $SOA I$ satisfies $\forall (x,c)$ means there must be a bidirectional channel between port$_1$ and port$_2$.

Next, counting constraints are used to restrict the amount of Seller instances in motivation example. If we define the channel type as sch from port $P_{brk}$ of the broker to port $P_{sell}$ of the seller. The counting constraint $\forall (P_{brk} \rightarrow P_{brk}, \text{sch}_{<100})$ means that at runtime the amount of seller instances should not exceed 100.

### 3.5 Constraints Checking on SOA Instances

To check whether an SOA instance satisfies the topological constraints, the satisfaction rules are listed in Fact.2

**Fact 2 (Constraints Satisfaction).** An SOA morphism $p: P \rightarrow G$ satisfies a basic constraint $\exists x$ if there exists an SOA morphism $q: C \rightarrow G$ with $q \circ x = p$. $p$ satisfies a conditional constraint $\exists (x,c)$ or $\forall (x,c)$ if for some [all] morphisms $q: C \rightarrow G$ with $q \circ x = p$ satisfy $c$. Every morphism satisfies true, and no morphism satisfies false. Morphism $p$ satisfies a boolean constraint $\neg c$ if $p$ does not satisfy $c$. $p$ satisfies $\land_{i \in I} c_i$ [some] if $p$ satisfies all [some] $c_i$ with $i \in I$. An SOA instance $G$ satisfies a topological constraint $c$ of the form $\exists x, \exists (x,c)[\forall (x,c)]$ if all [some] morphisms $p: P \rightarrow G$ satisfy $c$. Every SOA instance satisfies true, no instance satisfies false. $G$ satisfies $\neg c$ if $G$ does not satisfy $c$ and $\land_{i \in I} c_i$ if it satisfies all [some] $c_i$ with $i \in I$. We write $G \models c$ to denote that $G$ satisfies $c$.

Fact.1 shows the satisfaction of counting constraints itself. In the implementation of satisfaction checking, SOA morphism is a key problem. According to definition of SOA Morphism, computing of morphism function and computing of morphism function $f : SOAI_1 \rightarrow SOAI_2$ is to find a subgraph in $SOAI_2$ which matches the $SOAI_1$ and this matching is $f$. Some subgraph isomorphism algorithms have been proposed[16, 12, 17], and we used a variation of Valiente’s maximum common subgraph algorithm[17] which implemented in SimPack tools[3].

### 4 Verification Tool

An initial constrained evolution verification tool(CEVT) for service-based systems has been developed on Eclipse platform[v3.4] by Java SDK v1.6. Fig.4 shows the main screen of our CEVT.

An embedding SOA instances editor has been developed to edit SOA instances, production rules, left and right SOA instance in SOA morphisms. MVC programming style is adapted, and there are a view file(.soamtdiagram file) and a model file(.soamt file) for each SOA instance. The file name is the only identity for each SOA instance. When .soamtdiagram files are modified using editor,.soamt files will be modified automatically. Production rules and constraints of a system are organised in the “ProductionRule” and “Constraint” folder separately. Production rules and constraints of a system are organised in the “ProductionRule” and “Constraint” folder separately. Production rules and constraints of a system are organised in the “ProductionRule” and “Constraint” folder separately. Production rules and constraints of a system are organised in the “ProductionRule” and “Constraint” folder separately.

---

tion rules are edited directly, while constraints created by a constraint language which based on SOA morphisms that could be created by assigning two SOA instances name. This constraint language is very simple, for example, the “\(\exists(x, \exists(x_1, \text{true}))\)” statement means the constraint \(\exists(x, \exists(x_1, \text{true}))\). All terms in the language are names of morphisms or keywords (exists, all, true, false, not, and, or, (, ) ) and constraint variables are not supported now. Constraints of a system are stored in text files under the “Constraint” folder. For verification, SOA instances representing runtime architectures of a service based system should be created in CEVT. Then, they could be checked consistency with specifications by selecting ProductionRule folders and constraints files.

5 Conclusion and Future Work

For improving adaptability of service based system, a graph grammar based modeling and verification approach is proposed in this paper. Service oriented architectures at runtime are modelled by SOA instance graph and system specifications are presented as architectural style and topological constraints. The checking algorithms for satisfaction of SOA instances on style and constraints are given and implemented in the CEVT verification tool.

This paper implemented consistency checking of architecture instances with system specification. While in general service based systems, system evolution at runtime will be modelled by evolution policy (or graph rewrite rules) which are generally given by event-condition-action style at design time. Thus, modeling of this evolution policy and theirs consistent checking with style and constraints should be implemented in the future.

References