

A Policy-Driven Dynamic Reconfiguration For virtualized Web Services-based architectures

Ismael Bouassida Rodriguez¹, Riadh Ben Halima³, Khalil Drira¹, Christophe Chassot^{1,2} and Mohamed Jmaiel³
{bouassida,khalil,chassot}@laas.fr,
{Riadh.BenHalima,Mohamed.Jmaiel}@enis.rnu.tn

¹ CNRS; LAAS; 7 avenue du colonel Roche, F-31077 Toulouse, France

² Université de Toulouse; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France

³ Enis, Redcad, Route de la Soukra Sfax, Tunisia

Abstract. Adaptation of deployment is required for maintaining the Quality of Service (QoS) in Service Oriented Architectures (SOA). Dynamic reconfiguration of SOA is proposed here to cope with adaptation in reaction to or in prediction of QoS degradation. Handling such an issue needs to detect and to identify the deficiency source, and to reconfigure the architecture implementing service composition. System reconfiguration constitutes a complex activity acting on distributed software entities, and requires to be implemented by correct model-based approaches. We show in this paper how graph grammars can be used to design policy-driven reconfiguration mechanisms of architectures and to rule running applications using reconfiguration laws. We describe how coordinated architectural actions and reconfiguration policies are used to maintain QoS at runtime.

1 Introduction

Building distributed applications by dynamically selecting and connecting web services constitutes a powerful adaptation and reconfiguration mechanism. The high dynamics of the web requires new management infrastructures appropriate for web services. This implies the necessity of supervising architectural configurations of dynamically evolving applications in order to model formally reconfigurable architectures and assist reconfiguration actions for an adaptation purpose.

Software architectures [1] provide a way to reason about software systems at a level of abstraction higher than simple modules, services or lines-of-code. To model systems at this level, architects must have an expressive modeling representation and tools to check models at both design time and runtime [2]. Graphs and graph grammars represent suitable formalisms to specify architectures and to handle the reconfiguration process [3].

Providing generic and scalable solutions for automated reconfiguration can be driven by rule-based architectural reconfiguration policies. This is the approach we adopt and we propose in this paper. We elaborate a dynamic graph-based modeling approach and structural models that may represent the different

interaction-related dependencies: communication flows between services, and the service composition. The provided architecture has to reconfigure communication links proactively or reactively and autonomously in reaction to, or in prediction of the changes in the constraints at communication and processing resources deployment nodes. Reconfiguration may also be necessary to adapt the service composition to the evolution of the requirements of the application level. On the one hand, the provided solution has to react to detected problems. On the other hand, another way to deal with reconfiguration is to predict and to anticipate future QoS degradation like bandwidth overloads or services unavailability.

This can be particularly useful in situation where a given enterprise process executes multiple components and services available for different mashup compositions. In such a situation composing manually components from ad hoc analysis of the context could be a tedious and time-consuming task. Our model-based solution enables automating this task and providing automated reconfiguration of a given composition.

Our solution is based on graph grammars. We handle architectural reconfiguration by defining architectures as graphs where vertices correspond to software services and their operations. We use graph transformation to specify rules for deployment architecture changes (evolutions) while being in accordance with the architectural style. Dealing with dynamically evolving architectures requires characterising the set of consistent architecture instances. This is mandatory to validate the management models and to verify architectural properties preservation.

This paper is organized as follows: related work is presented in Section 2. Section 3 presents the architecture model for composing and adapting applications to QoS degradation. Section 4 details graph grammar productions for architectural reconfiguration following different substitution policies. Section 5 presents the reconfiguration framework associated with our substitution algorithms. Section 6 presents a case study that illustrates our approach. The last section concludes the paper.

2 Related Work

In this section, we present related work in a structured way. In the first part, we detail works dealing with system architecture description and adaptation. In the second part, we focus on autonomic system and we explain how the reconfiguration issue is addressed in autonomic computing. Finally, we present works that present a new approach: “model@runtime”. These approaches focus on managing reconfiguration complexity in a runtime environment.

There are different relevant contributions addressing system architecture adaptation. Such approaches use model-based strategies to apply the required transformations on the system architectures in order to adapt it to environment and requirement changes. These strategies define or reuse models describing the system software architectures. We distinguish between three general categories: formal techniques like graph grammars [4] and Petri nets [5], semantic ADLs

using ontologies [6] and ADLs (Architecture Description Languages) using XML deployment languages in [7]. ADLs can be proprietary or implementing the formal and the semantic architecture description models. These ADLs are used to guarantee the architecture evolving and correctness during the different predictable and unpredictable changes in the systems environment. The necessary actions to achieve such adaptations are specified in rules according to the application run-time context. In the work presented in [8] is an example of these approaches defining a complete model based architecture adaptation at the service, content and user interface levels. Authors in [9] present another model based method using graph grammars to adapt cooperative information systems to situation changes at the communication level.

Other works address software architecture modelling in order to build adaptation systems able to preserve QoS and manage reconfiguration. Rainbow [10] is a framework for self-adaptive systems. It composed of two layers. First, the system layer which collects information about the system and executes repair plan. Second, the architecture layer, which reflects the current architecture model, checks constraint violations while differentiating the real architecture with the conceived architecture, and determines necessary adaptation. The architecture and system layers interact through a *translation infrastructure*. However, experimental work [11] has shown that this approach for self-adaptation causes a significant slowdown of the system behavior. In [12], the authors propose an architecture-based approach to self-adaptative software. The approach has not been implemented and has not been applied to a specific domain or application. It insists on system integrity which requires the assurance of consistency, correctness and coordination of changes. In [7], Taylor and Dashofy refine the architecture proposed in [12]. They exhibit a style-based approach for self-adaptative systems. It focuses on the infrastructure to support the creation, validation and execution of repair strategies.

The autonomic concept and systems suggested in the literature are defined in various ways. Several works focus on the need of autonomic computing and the first concepts. They focus and detail the autonomic control loop. In [13], Autonomic Computing Systems are defined as systems that automatically manage themselves by carrying out tasks that have been traditionally performed by computer specialists.

Other works consider the network point of view of autonomic computing. The work in [14] explains Autonomic Network definition in more detail, links it to the foundational principles of architecture for Autonomic Network Management and provides guidance on how to develop specifications and best practices for the building of Autonomic Communication Systems. Authors of [15] give an overview of the different architectures that support the design, implementation and deployment of autonomic systems. They present issues related to the design, implementation and deployment of autonomic networks-it focus on the autonomic-management approaches. Also, they give the motivation behind the emergence of autonomic, self-managed systems and the required features of such architectures. The applications that focus on group interactions consider usually

mobile or highly mobile entities. The actors of the collaboration may be human or non-human. For the human case, it is about cooperative activities where the awareness is crucial [16]. For the non-human case, the agents are used to support the collaboration [17]. Both of these works consider a Multi-Agent approach for autonomic systems. They consider an agent level on top of a middleware level. The used formalism is graphs with hyper-graph extensions for [17].

Other works address problems related to the management of huge information associated with runtime situations in self-adaptive software. A new approach to manage complexity in runtime environments is referred as “models@runtime”. A “model@runtime is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective.” as defined in [18]. Authors in [19], present an approach for specifying and executing dynamically adaptive software systems. This approach combines model-driven and aspect-oriented techniques. This work, which is a part of the EU-ICT DiVA project (Dynamic Variability in complex, Adaptive systems), addresses two drawbacks related to adaptation management and evolution management by using software models at runtime as well as at design time. The authors intend to tame the explosion in the number of artefacts considered. Authors in [20] present model-based traces as runtime models and traces analysis methods. The focus is on the scenario-based trace as a runtime model and on the metrics and operators to analyze it. Authors illustrate their proposed methods using different application scenarios. The syntax and semantics of various types of the model-based traces in this work are not formally defined. Authors in [21] provide operations control center through an adaptive vision in which human users can understand and manage software systems at runtime. Their approach, entitled “architectural runtime configuration management”, creates a model that captures an adaptive system’s configurations and corresponding behaviors and organizes them in a historical graph of configurations.

3 Architectural model representation approach

In web service-based applications, a set of distributed services collaborate in order to respond to user’s requirements. Such a composition is built while describing the whole process required by the end user, and querying a repository about available web services (The *WS Discovery Service*).

We consider here a generic template of virtual architecture where a Virtual Intermediate Connector (VIC) is used to bind service requesters. The VIC can route the requests to effective service implementations depending on the operation being requested and being provided and depending on the global or operation-specific performances of the requested service.

The VIC asks the *WS Discovery Service* for required web services in order to achieve building the composed application. The research process is based on functionalities and qualities of the desired web service. The architecture graph to which our approach applies may be characterized by any successive compositions of the elementary patterns:

$N_i(\text{ServiceRequester}) \rightarrow N_j(\text{VICType, Interface})$ and $N_j(\text{VIC}) \rightarrow N_k(\text{ServiceProvider, performanceAttribute})$, where N_i, N_j, N_k denote the graph nodes associated, respectively, with the “Service Requester”, the “Virtual Intermediate Connector” and the “Service Provider”.

Once the composition process is achieved, the application is ready to be deployed and used. During runtime, quality of service (QoS) offered by each service may be degraded and hence, we need to substitute it by rerouting the requests for one or more operations to different Service providers. A substitution is considered as the elementary architectural reconfiguration action after a QoS degradation detection or prediction on the currently bound service. Searching for an equivalent service using the *WS Discovery Service* is then required to find partially or totally equivalent services.

Our representation approach of architectural configurations relies on graphs and graph grammars. Graph grammar theories represent an appropriate formalism to handle the reconfiguration process. Moreover, graph grammars are a tractable and powerful way of handling complex transformations and characterising the set of configurations without its explicit enumeration. Following the common representation, a node refers a web service and a directed edge designates an invocation link.

4 The Graph Grammar-based rule-oriented reconfiguration

We elaborate here, graph grammars that implement reconfiguration policies and that minimize unnecessary reconfigurations. In the proposed policies, minimize the number of removed edges.

Generative grammars [22] are defined, in general, as a classical grammar system $\langle AX; NT; T; P \rangle$, where AX is the axiom, NT is the set of the non-terminal nodes, T the set of terminal nodes, and P the set of transformation rules, also called grammar productions.

Our approach based on [3] that combines the structure of a Double PushOut (DPO) production structure and an extension of the Neighbourhood Controlled Embedding mechanism (edNCE) connection instructions mechanisms.

An instance belonging to the graph grammar is a graph containing only terminal nodes obtained starting from axiom AX by applying a sequence of productions in P . P specifies the set of grammar productions that are of the form (L, K, R) where L, R and K are subgraphs. Such productions are considered applicable to a given graph G if a graph homomorphism from L to G exists. If a production is applicable, its application leads to the removal of the subgraph occurrence $(L \setminus K)$, the insertion of an isomorphous copy of the subgraph $(R \setminus K)$.

We use the following notations in the sequel: graph nodes are represented by $N_i(\text{att}_1, \dots, \text{att}_n)$ where “ i ” allows a node to be identified in the graph and where $\text{att}_1, \dots, \text{att}_n$ are attributes of the node. Attributes may represent properties of the software entity associated with a given graph node such as: the *role* : (*Requester, Provider*), the *status* : (*Deficient, Non_Deficient*), etc.

4.1 Monolithic service substitution policy

Table 1 shows the graph grammar for substituting of a single WS by another equivalent. Applying the production $p1$ leads to the unbounding of the deficient WS described by the sub-graph containing the node N_2 and the edge $N_1 \rightarrow N_2$, and the bounding the substitute WS described by the sub-graph containing the node N_3 and the edge $N_1 \rightarrow N_3$.

$GG1 = (AX, NT, T, P)$ with: $NT = \{ \}, T = \{N_1(Id_1, Type, interface), N_2(Id_2, SW, State)\}, P = \{p1\}$
$p1 = (L = \{N_1(Id_1, Type, interface), N_2(Id_2, WS, Deficient), N_1 \rightarrow N_2\};$ $K = \{N_1(Id_1, Type, interface)\};$ $R = \{N_1(Id_1, Type, interface), N_3(Id_3, WS, not(Deficient)), N_1 \rightarrow N_3\})$

Table 1. GG1: the monolithic substitution of a WS

Figure 1 shows a visual representation of this production that unbounds the deficient service N_1 and substitutes it by a Non_Deficient service N_3 .

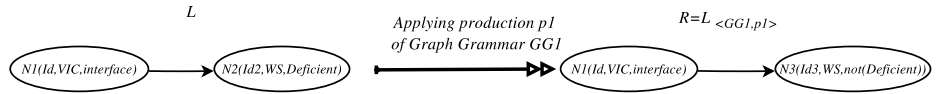


Fig. 1. Graphical representation of GG1 production $p1$

4.2 Cost-aware composite service substitution policy

The graph grammar described in Table 2 describes a composite substitution policy where requests related to a given operation Op_i of the interface of a WS are routed to a different WS implementing the same operation. This may be useful in different situations where a new service is discovered and which implements more efficiently or less costly the given operation. This can also be applied when a particular operation is over-requested and a load balancing is necessary. This can also apply to a situation where an operation is implemented in a way that does not free resources and its repeated invocation leads to an increasing response time. All these situations will be summarized as an “*availability*” property of a service with respect to a given operation.

$GG2 = (AX, NT, T, P)$ with: $T = \{N_1(Id_1, Type, interface), N_2(Id_2, WS, State)\}$, $NT = \{ \}$ and $P = \{p1, p2\}$
$p1(Op_i) = (L = \{N_1(Id_1, Type, interface), N_2(Id_2, WS, Deficient), N_1 \xrightarrow{\{Op_1, \dots, Op_n\}} N_2\}$; $K = \{N_1(Id_1, Type, interface), N_2(Id_2, WS, Deficient)\}$; $R = \{N_1 \xrightarrow{\{Op_1, \dots, Op_n\} \setminus Op_i} N_2, N_1(Id_1, Client), N_3(Id_3, WS, not(Deficient)),$ $N_1 \xrightarrow{Op_i} N_3\}$);
$p2(Op_i) = (L = \{N_1(Id_1, Type, interface), N_2(Id_2, WS, Deficient), N_1 \xrightarrow{\{Op_i\}} N_2\}$; $K = \{N_1(Id_1, Type, interface)\}$; $R = \{N_1(Id_1, Client), N_3(Id_3, WS, not(Deficient)), N_1 \xrightarrow{Op_i} N_3\}$);

Table 2. GG2: Service composite substitution policy

Figure 2 and figure 3 show a visual representation of the productions $p1(Op_i)$ and $p2(Op_i)$ of $GG2$.

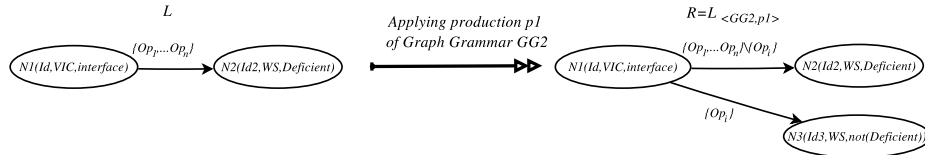


Fig. 2. Graphical representation of $GG2$ production $p1$

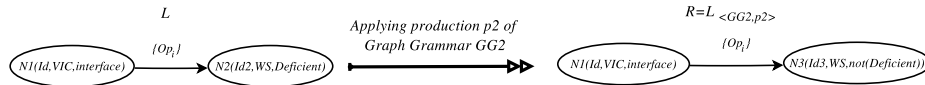


Fig. 3. Graphical representation of $GG2$ production $p2$

Applying the production $p1$ leads to the removal of the the edge $N_1 \xrightarrow{\{Op_1, \dots, Op_n\}} N_2$, and the addition of the substitute WS as described by

the sub-graph containing the node N_3 and the edges $N_1 \xrightarrow{\{Op_1, \dots, Op_n\} \setminus Op_i} N_2$ and $N_1 \xrightarrow{Op_i} N_3$.

Applying the production p_2 leads to the removal of the deficient WS as described by the sub-graph containing the node N_2 and the edge $N_1 \xrightarrow{Op_i} N_2$, and the addition of the substitute WS described by the sub-graph containing the node N_3 and the edge $N_1 \xrightarrow{Op_i} N_3$.

5 Implementation of the reconfiguration management process

This section presents the foundations of the QoS-Oriented Self-Healing middleware (*MARQ*) which implements monitoring and reconfiguration functionalities.

5.1 Architectural Framework

The middleware *MARQ* includes four main components [23]:

- The Monitoring component*: It includes observing and storing relevant QoS parameter values entities using requester-side and provider-side monitors.

- Logging Manager* which manages storing the QoS monitored data.

- The Analysis component*: It inspects the service behavior and detects QoS degradation.

- The Diagnosis component*: It reasons about the degradation by exploiting the stored QoS values and identifies the deficiency source. Then, it sends the diagnostic to the *Reconfiguration* component for repair.

- The Reconfiguration component*: It requests for an equivalent service from the *Substitutable Services WSDLs* and switches requesters to substitutable providers using a dynamic binding connector implementing the VIC.

5.2 Architectural Reconfiguration

The architectural reconfiguration acts by substituting the deficient service by another equivalent from the selected *Substitutable Services WSDLs* or by a composition of several services. We present the algorithms (Algorithm 1 and Algorithm 2) that implements, respectively, the graph grammars specified in Tables 1 and 2.

The reconfiguration execution is performed thanks to the *Dynamic Binding Connector* which unbinds the current connection, and reroutes requests to the new selected services in a seamless way for remote requesters. The *Dynamic Binding Connector* is automatically generated and deployed using runtime compilation and reflective programming.

The substitution of a deficient web service may be done through one or many other services according to a given policy. We consider here two possible policies for monolithic and composite substitutions respectively.

Algorithm 1 The monolithic substitution algorithm (algorithm 1)

```
/*Considered WSDL interface of the deficient WS to substitute*/;
WSDL_Interface( $S_{deficient}$ ) =  $\{Op_1, \dots, Op_N\}$ 
/*Browse the available services*/;
Analyze_Available_WS_Interfaces();
/*Considered web services for the substitution*/;
 $S \leftarrow \{S_1, \dots, S_M\}$ 
(1) Single_Substitution  $\leftarrow \exists S_t$  such that:
WSDL_Interface( $S_t$ )  $\supseteq$  WSDL_Interface( $S_{deficient}$ )
if (Single_Substitution) then
  for each ( $Op_i$ )-request do
    Reroute_request_to  $S_t$ 
  end for
end if
```

Algorithm 2 The composite substitution algorithm (algorithm 2)

```
/*Considered WSDL interface of the deficient WS to substitute*/;
WSDL_Interface( $S_{deficient}$ ) =  $\{Op_1, \dots, Op_N\}$ 
/*Browse the available services*/;
Analyze_Available_WS_Interfaces(); Composite_Substitution  $\leftarrow \exists S_h, \dots, S_k$  such
that:  $\left( \bigcup_{j=h}^k \text{WSDL\_Interface}(S_j) \supseteq \text{WSDL\_Interface}(S_{deficient}) \right) \wedge$ 
 $(\exists Op_i \in \text{WSDL\_Interface}(S_{deficient}), \exists (x, y)$  such that:
 $Op_i \in (\text{WSDL\_Interface}(S_x) \cap \text{WSDL\_Interface}(S_y)))$ 
if (Composite_Substitution) then
  /* Consider service availability*/;
  if (Service_Availability_Policy) then
    for each ( $Op_i$ )-request such that:  $Op_i \in \text{WSDL\_Interface}(S_{j_1}) \cap \dots \cap$ 
 $\text{WSDL\_Interface}(S_{j_a})$  do
      Reroute_request_to  $S_{j_k}$ , such that:
       $S_{j_k}$  has the Highest Availability among all  $S_{j_1}, \dots, S_{j_a}$  wrt  $Op_i$ 
    end for
  else
    /*trivial load balancing*/;
    for each ( $Op_i$ )-request such that:  $Op_i \in \text{WSDL\_Interface}(S_{j_1}) \cap \dots \cap$ 
 $\text{WSDL\_Interface}(S_{j_a})$  do
      Reroute_request_to  $S_{j_k}$ 
       $S_{j_k} \leftarrow \text{nextElementOf}\{S_{j_1}, \dots, S_{j_a}\}$ 
    end for
  end if
end if
```

The first policy, implemented by algorithm 1, handles the monolithic substitution. All requests are routed to a new web service S_t , which offers the same operations as the deficient WS $S_{deficient}$. In such a case, $S_{deficient}$ is entirely replaced by S_t .

The second policy, implemented by algorithm 2, considers the composite substitution by two or more services that implement the interface offered by the currently bound service.

6 The FoodShop Case study: a web application

The FoodShop application depicted in Figure 4, has been studied in the framework of the WS-DIAMOND project.

6.1 Description of the case study

The FoodShop application is concerned with a web service-based application representing a company that sells and delivers food. The company has online Shops and several warehouses ($warehouse_1, \dots, warehouse_n$) located in different areas that are responsible for stocking imperishable and perishable goods and delivering items to customers.

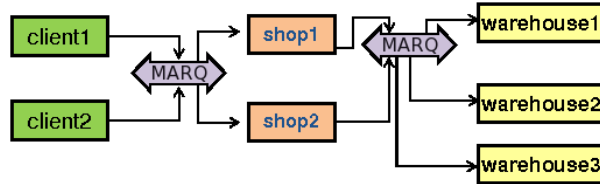


Fig. 4. Details of global distributed self-healing architecture applied to the FoodShop case study

The *MARQ* middleware is deployed within the FoodShop application between each pair of provider/requester as shown in Figure 4. The *Diagnosis* modules exchange information, in order to coordinate the healing actions. For instance, for the two linked services $shop_1$ and $warehouse_1$, the QoS degradation of $warehouse_1$ may propagate to $shop_1$ from the *client* point of view. This triggers a healing process within the two *MARQ* middleware instances. If not in coordination, each *MARQ* middleware substitutes its provider. However, the global reasoning about the degradation deduces that the $shop_1$ QoS degradation is due to the propagation and only $warehouse_1$ has to be substituted. The deployment of the *FoodShop* within the middleware *MARQ* enables the monitoring at the HTTP level. It extracts parameters like IP address, the deployment host, the communicating WS names, the invoked operations, the execution time and the communication type (synchronous or asynchronous). These information allow the dynamic discovery of involved parties in the application and the automatic building of the application profile.

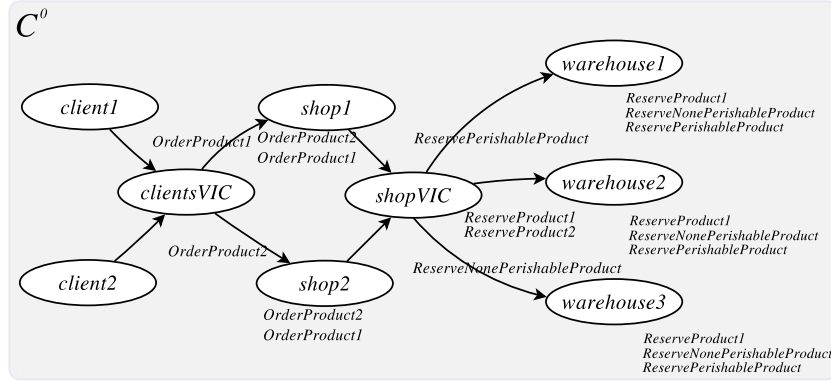


Table 3. Initial Architectural configuration

We have developed a monitoring graphical interface with the middleware *MARQ*, in addition to the self-healing features. The collected monitoring data enable us to draw up dynamically a visualization window of WS hosts and invoked operations.

6.2 Application of our approach

To illustrate our approach, we propose the following scenario. Figure 3 shows the initial configuration C^0 . We consider two clients, two shops and three warehouses. Each shop implements two operations: Op_1 and Op_2 . These operations are *OrderProduct1* and *OrderProduct2*. Each warehouse implements four operations: Op_1 , Op_2 , Op_3 and Op_4 . These operations are *ReserveProduct1*, *ReserveProduct2*, *ReservePerishableProduct* and *ReserveNonePerishableProduct*.

$Client_1$ invokes the operation $shop_1/OrderProduct1$ that invokes the operation $warehouse_1/ReserveProduct1$. $Client_2$ invokes the operation $shop_1/OrderProduct2$ that invokes the operation $warehouse_2/ReservePerishableProduct$. $Client_2$ invokes $shop_2/OrderProduct2$ that invokes the operation $warehouse_1/ReserveNonePerishableProduct$.

To illustrate the reconfiguration policies, three events are considered. The events are “ $shop_1/OrderProduct1$ degradation”, and “ $warehouse_2/ReserveProduct1$ degradation”.

In the following, we show the reconfiguration of the architecture when the first policy is used. The architecture is reconfigured according to the reconfiguration algorithm that implements in this case the graph grammar GG1.

Monolithic service substitution policy When the $shop_1/OrderProduct1$ performance is considered as degraded, the WS $shop_1$ is disconnected and WS

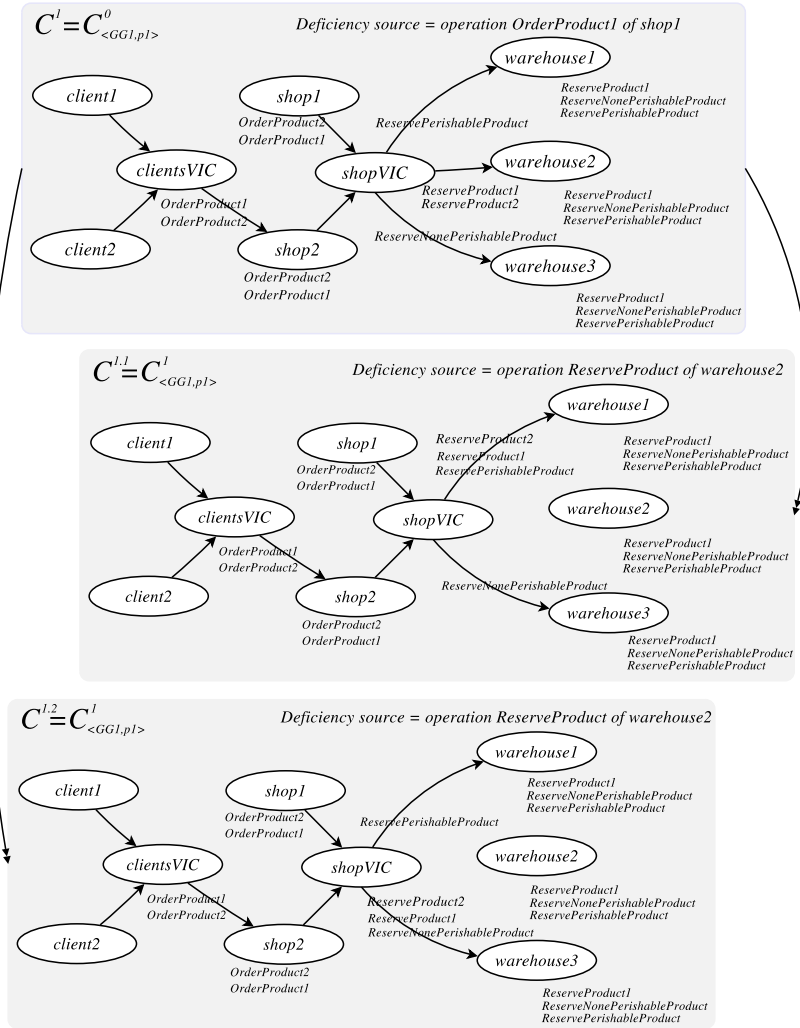


Table 4. Monolithic service substitution

$shop_2$ is bound to requester, in order to provide the service requested by the client as shown in $C^1 = C^0_{\langle GG1, p1 \rangle}$ (see Figure 4).

When the $warehouse_2/ReserveProduct1$ performance is considered as degraded, the WS $warehouse_2$ is disconnected and WS $warehouse_1$ provides the service requested by the $shop_2$ as shown in $C^{1.1} = C^1_{\langle GG1, p1 \rangle}$ (see figure 4). We can also have a WS $warehouse_3$ that provides the service requested by the $shop_2$ as it is shown in $C^{1.2} = C^1_{\langle GG1, p1 \rangle}$ instead of $warehouse_1$.

This policy represents a basic solution that is efficient in case of a limited service number but in some cases generates unnecessary reconfiguration. This

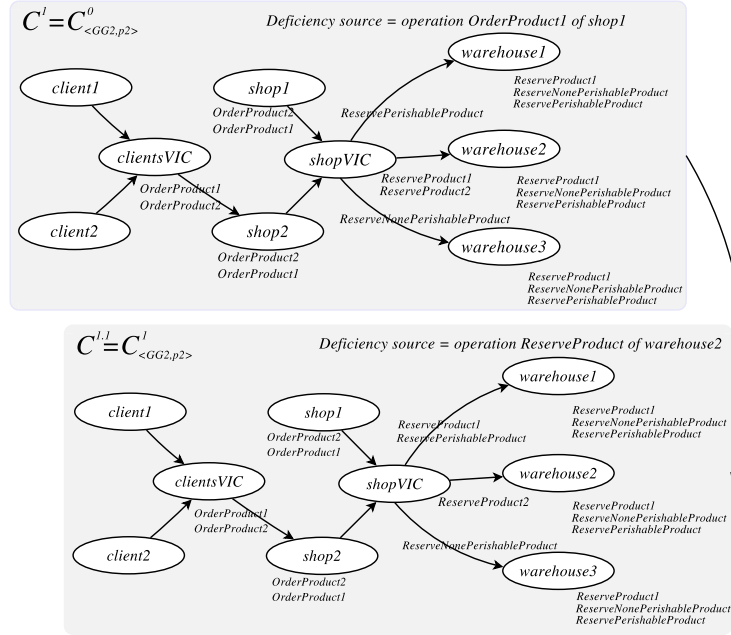


Table 5. Composite service substitution according to WS cost

is the case, for example, when *warehouse2* is unbound with only one operation (*ReserveProduct1*) at the origin of QoS degradation. This provokes the routing request to the equivalent operation *ReserveProduct2* of *warehouse2* that is unnecessary (*ReserveProduct2* is working).

6.3 Composite service substitution according to WS cost policy

To avoid unnecessary configuration enumeration, we propose a policy that replaces the degraded operation based on its cost. The architecture is reconfigured according to the reconfiguration algorithm implementing the graph grammar GG2.

When the *shop1/OrderProduct1* performance is considered as degraded, the *OrderProduct1* of WS *shop2* provides the service requested by the client as shown in $C^1 = C_{<GG_2, p_2>}^0$ (see Figure 5).

When the *warehouse2/ReserveProduct1* performance is considered as degraded, the *ReserveProduct1* of WS *warehouse1* provides the service requested by the *shop2* as shown in $C^{1.1} = C_{<GG_2, p_2>}^1$ (see Figure 5). Here there is an other alternative using *ReserveProduct1* of *warehouse3* but *ReserveProduct1* of WS *warehouse1* costs less. Hence, the use on this policy avoids the generation of useless reconfigurations and *warehouse2* still providing the operation *ReserveProduct2*.

7 Conclusion

In this paper, we studied the dynamic reconfiguration of Service Oriented Architectures for maintaining the Quality of Service in perturbation-prone environments. We adopted a virtualized architectural model which can cope with the problem of adapting the requester-provider communication in different situations and at different levels of interaction. The virtual entities introduced can be implemented at different communication levels: at the HTTP level using proxies, at the SOAP level using interceptors and at the TCP transport protocol level using Middleboxes. Our proofs of concepts and experiments cover these three levels of implementation.

Our approach uses graph grammar theories to implement rules that characterize the set of the different configurations candidate to solving composite or monolithic reconfiguration. Contrarily to enumerative approaches that define extensively the different equivalent and valid configurations, our approach is more appropriate for handling the scalability. It can handle Web-based applications in an open world view such as in collaborative activities where the number of participants may vary in a uncontrolled and unpredictable way.

Future research actions include the definition of reconfiguration rules that consider additional QoS-related information and more context parameters (e.g. by using context and QoS ontologies). Moreover, we aim to provide the appropriate reconfigurations rules where adaptation may be constrained by the evolving of the supported activity requirements and by the changes of the communication resources constraints.

References

1. Garlan, D., Perry, D.: Introduction to the special issue on software architecture. *IEEE Transactions On Software Engineering* **21**(4) (1995) 269–274
2. Abi-Antoun, M.: Static extraction and conformance checking of the runtime architecture of object-oriented systems. In: *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, New York, NY, USA, ACM (2008) 911–912
3. Guennoun, K., Drira, K., Diaz, M.: A proved component-oriented approach for managing dynamic software architectures. In: *Proc. 7th international conference on software engineering and application*, Marina Del Rey, CA, USA (2004)
4. Hirsch, D., Inverardi, P., Montanari, U.: Graph grammars and constraint solving for software architecture styles. In: *ISAW '98: Proceedings of the third international workshop on Software architecture*, New York, NY, USA, ACM (1998) 69–72
5. Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* **77**(4) (1989) 541–580
6. Zhou, Y., Pan, J., Ma, X., Luo, B., Tao, X., Lu, J.: Applying ontology in architecture-based self-management applications. In: *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, New York, NY, USA, ACM (2007) 97–103

7. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of xml-based architecture description languages. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, New York, NY, USA, ACM (2002) 266–276
8. Chaari, T., Laforest, F., Celentano, A.: Adaptation in Context-Aware Pervasive Information Systems: The SECAS Project. *Int. Journal on Pervasive Computing and Communications(IJPCC)* **3**(4) (2007) 400–425
9. Chassot, C., Guennoun, K., Drira, K., Armando, F., Exposito, E., Lozes, A.: Towards autonomous management of qos through model-driven adaptability in communication-centric systems. *ITSSA* **2**(3) (2006) 255–264
10. Cheng, S.W., Garlan, D., Schmerl, B.R.: Making self-adaptation an engineering reality. In: *Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations*. Volume 3460 of *Lecture Notes in Computer Science.*, Springer (2005) 158–173
11. David Garlan, Shang-Wen Cheng, B.S.: Increasing system dependability through architecture-based self-repair. *Appears in Architecting Dependable Systems* (2003)
12. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems* **14**(3) (1999) 54–62
13. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* **36** (2003) 41–50
14. Agoulmine, N., Balasubramaniam, S., Botvitch, D., Strassner, J., Lehtihet, E., Donnelly, W.: Challenges for autonomic network management. In: *In 1st IEEE International Workshop on Modelling Autonomic Communications Environments (MACE)*. (2006)
15. Johnson, J.H., Irvani, P.: The multilevel hypernetwork dynamics of complex systems of robot soccer agents. *ACM Trans. Auton. Adapt. Syst.* **2**(2) (2007) 5
16. Locatelli, M.P., Vizzari, G.: Awareness in collaborative ubiquitous environments: The multilayered multi-agent situated system approach. *ACM Trans. Auton. Adapt. Syst.* **2**(4) (2007) 13
17. Liu, H., Parashar, M., Member, S.: Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems*, Editors: R. Sterritt and T. Bapty, IEEE Press **36** (2006) 341–352
18. Blair, G., Bencomo, N., France, R.: Models@ run.time. *Computer* **42**(10) (2009) 22–27
19. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. *Computer* **42**(10) (2009) 44–51
20. Maoz, S.: Using model-based traces as runtime models. *Computer* **42**(10) (2009) 28–36
21. Georgas, J., van der Hoek, A., Taylor, R.: Using architectural models to manage and visualize runtime adaptation. *Computer* **42**(10) (2009) 52–60
22. Chomsky, N.: Three models for the description of language. *IEEE Transactions on Information Theory* **2**(3) (1956) 113–124
23. Halima, R.B., Drira, K., Jmaiel, M.: A qos-oriented reconfigurable middleware for self-healing web services. In: *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services, Beijing (Chine)*, IEEE Computer Society (2008) 104–111