MODELING ARCHITECTURAL LEVEL REPAIR IN WEB SERVICES

Francisco Moo-Mena [1,2] and Khalil Drira [1]
[1] *Laboratoire d'Analyse et d'Architecture des Systèmes, 7 Av. du Colonel Roche, 31077 Toulouse, Cedex 04, France*
[2] *Facultad de Matemáticas, Universidad Autónoma de Yucatán, Apartado Postal 172 Cordemex, 97110 Mérida, México*

Keywords: Architecture model, Architecture reconfiguration, Web services, repair actions, UML.

Abstract: Failures during Web service execution may depend on a wide variety of causes, such as network faults, server crashes, or application-related errors, such as unavailability of a requested Web service, errors in the orchestration or choreography of applications, missing data or parameters in an execution flow, or low Quality of Service (QoS). Providing efficient solutions requires handling adaptability not only at the behavioral level, but also at the architectural level. Most of existing standard solutions focus on the behavioral level using reconfiguration mechanisms. In this paper, we propose a model-based approach providing adaptability actions at the architectural level. Typical architectural reconfiguration actions, such as duplication of services or substitution of a non conforming service are introduced. An example is illustrated and discussed with respect to a running example involving coordinated Web services.[a].

[a]This work has been funded by the IST WS-DIAMOND European project.

1 INTRODUCTION

In the WS-DIAMOND (Web service DIAgnosability, MONitoring, and Diagnosis) project (IST, 2006), methodologies and solutions are investigated to design and deploy self-healing Web Services. Namely, Web services able to detect and repair anomalous situations in the execution of distributed Web based applications. WS-DIAMOND studies and develops methodologies and solutions for the creation of self-healing Web Services, able to detect anomalous situations, which may manifest as the inability to provide a service or to fulfill Quality of Service (QoS) requirements, and to recover from these situations. To recover from these situations, various actions can be undertaken, depending on where the fault has occurred. In many cases, such actions lead to rearrangement or reconfiguration of the architecture.

According to (Dashofy et al., 2002), the ability to dynamically repair a system at runtime based on its architecture requires several capabilities: (i) the ability to describe the current architecture of the system, (ii) the ability to express all changes to the architecture that will serve as a repair plan, (iii) the ability to analyze the result of the repair to gain confidence that the change is valid, and (iv) the ability to execute the repair plan on a running system without restarting the system.

Following a model-based development approach, application scenarios are expressed at a conceptual level, for example, by adapting UML diagrams. This representation allows to reason about requirements at a high level of abstraction, without considering implementation-specific details. A conceptual model of the architecture would allow an understanding of the basic mechanisms, and their suitability for a certain task, at a non-technical level (Baresi et al., 2003).

The goal of this paper is to present actions for reconfiguration of QoS mismatches, and to propose a model-based strategy to management and reconfiguration of non conforming Web services applications. The approach here proposed is based on reconfiguration actions defined for different types of mismatches. The QoS mismatches we deal with are at the architectural level, so they involve reconfigurations actions, such as addition and removal of services and connec-

Moo-Mena F. and Drira K. (2007).
MODELING ARCHITECTURAL LEVEL REPAIR IN WEB SERVICES.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Internet Technology*, pages 240-245
DOI: 10.5220/0001280002400245

tions. In particular, we focus on mechanisms based on adaptivity achieved through service duplication or substitution. This model-based approach involves an explicit partitioning between functional architectural issues. In this sense, our first task involves representation of functional (applicative) aspects of architectures by describing a structural specification (across UML extensions) encompassing the functional decomposition of a Web service application. And secondly, reconfiguration management is considered by applying a set of transformation rules (as defined by graph rewriting theory) in order to define reconfiguration actions at the architectural level. In order to facilitate the understanding of our proposal we apply our idea to the context of a *Cooperative Review* application.

The rest of the paper is organized as follows: section 2 introduces our proposal. Section 3 depicts an applicative sample of our approach. Section 4 describes research related to our work. And Section 5 presents conclusions and perspective of this work.

2 OUR APPROACH

2.1 Architectural Description

A first step consists in modeling of architectures representing a Web service application. For this, some model types are defined as follows:

- ***The Cooperative role*** represents a logical deployment unit associated to sites wherein participating actors are connected. It may be that a *Cooperative role* is identified in practice with a particular runtime entity (i.e. Web service application server, orchestration engine, etc.) but no correlation exists between *Cooperative role* and runtime entity in general, so a *Cooperative role* can either correspond to several runtime entities or a runtime unit can match the services associated with several *Cooperative roles*. The concept of *Cooperative role* fulfills inherent needs in any Web service application by distinguishing logical distribution of sites hosting Web services. Specific *Cooperative role* examples would be: “Author Site”, “Supplier Site”, “TrackChair Site”, etc.
- ***The Service category*** represents a global and common functionality offered by a set of Web services distinguished by their class. Web services belonging to the same *Service category* can be deployed on one or more *Cooperative roles*. We distinguish, for example, *Service categories* such as “Travel Agency Management”, “Food Shopping Management”, “Cooperative Review Management”, etc.
- ***The Service class*** refers to the functional behavior offered by a particular type of Web service as a basic logical unit associated to a *Service category*. In a deployment scenario, a *Service class* represents a specific part of the business logic offered by a *Service category* deployed into a *Cooperative role*. For the sake of simplicity we consider a unique *Service class*, of the same category, being deployed into the set of *Cooperative roles*. We distinguish, for example, *Service classes* such as “Booking Manager”, “Costumer Manager”, “Reviewing Manager”, etc.

A graphical notation is defined in order to represent static architectural specification of a Web service application. This notation is based on issues defined by the UML deployment diagram (OMG, 2005) and extends it as shown in figure 1. This metamodel define each of the architectural types as extensions of the Node metaclass. A relation of aggregation exists between the *Cooperative role* and the *Service category* since the latter may be deployed on different *Cooperative roles*. As well, a relation of composition is enabled between the *Service category* and the *Service class* meaning the latter may only be deployed on a unique *Service category*. Services (or Web services) are defined (besides of the architectural types) by extending the Component metaclass.

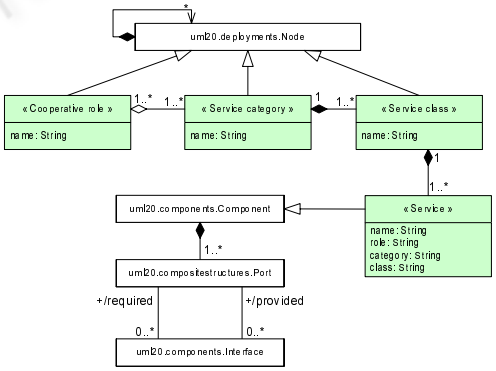

Figure 1: Metamodel extending UML deployment view.

According to the proposed metamodel, the notions of *Cooperative Role*, *Service Category* and *Service class* are all represented by a Node symbol. A tag allows to distinguish each element by its name. Dependencies between these architectural elements are implicitly represented by embedding each element into its counterpart. As well, we use *Communication paths* in order to represent interconnection at the *Service class* level. Services and specific interactions between

them are based on component metaclass as defined by UML 2 (OMG, 2005). Each service is represented by a component design, in addition to some tags remembering dependencies with regard to the architectural elements (Cooperative role, Service category and Service class).

2.2 Basic Rule Definition

This step consists in the definition of dynamic architectures in order to represent reconfiguration actions. To make it possible we define some basic rules based on graph transformation. Definition of rules is merely based on the basic model types. Namely, each rule must involve either a *Cooperative role, Service category, Service class* or a mixture of them into its specification. This strategy aims to provide some important advantages such as: (i) definition of simpler and generic rules, (ii) definition of more powerful rules, as a single model type (Cooperative role, Service category or Service class) generally involves multiple instances of Web services.

In order to build basic rules we define a simple graph, where *vertices* represent Web services and *edges* represent connections between Web services. Each vertex of the graph is defined by:

$WS_{(i,j,k,l)}$ *where* :
i represents a Cooperative role,
j represents a Service category,
k represents a Service class, and
l represents the identifier of the WS.

In what follow we depict three basic rules which involve some typical actions required to build reconfiguration actions. In these rules, variables *X,Y* and *Z* match graph vertices, where each vertex represents one or more instances of Web services. These basic rules are defined as follows:

```
Rule [name-R1](i,j,k,l,i',j',k',l')
 Match Vertices:
   X with parameters i,j,k,l;
 Add Vertices:
   Y with parameters i',j',k',l';
```

```
Rule [name-R2](i,j,k,l,i',j',k',l',i'',j'',k'',l'')
 Match Vertices:
   X with parameters i,j,k,l;
   Y with parameters i',j',k',l';
   Z with parameters i'',j'',k'',l'';
 Match Edges:
   X->Z, Z->X
 Add Edges:
   Y->Z, Z->Y
 Restriction Edges:
   Y->Z, Z->Y
```

- *Rule R1* deals with addition of Web services. In practice that means activation or deployment of Web services instances.

```
Rule [name-R3](i,j,k,l)
 Delete Vertices:
   X with parameters i,j,k,l;
```

- *Rule R2* deals with addition of connections. A restriction is imposed as the connections between a pair of elements must remain unique, so this rule is only applicable if declared edges have no been created previously, and
- *Rule R3* deals with removal of elements and associated connections. In practice removal could be interpreted as deactivation of implied elements. It must be noted due to parameters are associated to model types each variable would instantiate several elements (Web services instances and connections).

Based on this principles, for instance, a basic rule describing something like: "add Web services with Service category = *AuthorManagement*", would be carried out by applying the Rule R1 with $X = WS_{(i=*,j=AuthorManagement,k=*,l=*)}$, where "*" matches any value in all cases. That involves all elements relied on this Service category, namely, the Service classes and services.

2.3 Reconfiguration Actions

In order to model reconfiguration actions for a non conforming Web service application, we use the basic rules aforementioned and define policies for their right application. Namely, we establish an ordered sequence of basic rules to enable architectural reconfiguration actions such as duplication or substitution of Web services.

- *Duplication*(or replication) involves addition of services representing similar functionalities. This aiming at improving load balancing between services in order to better adapt, for instance, to QoS requirements.
- *Substitution* encompasses redirection between two services. Applying this action means the first one is deactivated and relieved by the second one.

Hence, the *Duplication* and *Substitution* reconfiguration actions are defined as follows:

$$Duplication = R_1 R_2$$

$$Substitution = R_1 R_2 R_3$$

That means reconfiguration action of Duplication implies the application sequence of Rule R1 and Rule R2. And reconfiguration action of Substitution implies the application sequence of Rule R1, Rule R2 and Rule R3.

3 ILLUSTRATION

This example illustrates the functioning of a common collaborative activity addressing the problem of "review process". This problem has several instantiations in different activity areas. In industrial activities such as engineering of complex and embedded systems, the "design review" activity is known to be as one of the most complex activities in aerospace industry. The "cooperative review" application scenario we study in the context of the WS-DIAMOND project aims to support such activities. Most of the defined services, parameters, actors and processes are generic and may be applied to different domain-specific review processes. In order to facilitate the understanding and the collaboration within our project, we choose to instantiate the review process by the well-known scenario of the review process in scientific publishing activities. Namely, we consider the specific case of the scientific conferences looking for describing the automatic functioning of their different steps within a service-oriented approach.

Several actors collaborate in order to accomplish the various management tasks of the cooperative reviewing process. The different types of actors are presented as follows: (i) the *Conference Chair* represents the principal actor in charge of organization of the conference. (ii) The *Track Chair* has in charge to manage a particular conference's session, (iii) the *Author* represents each potential contributor to the different topics of the conference, (iv) the *Reviewer* is an expert in one or more of the domains defined by the different topics of the conference.

3.1 Architecture Description

For this example an architectural description is depicted in figure 2. This description involves:

- 4 Cooperative Roles: (i) *AuthorSite* considers participant site for authors, (ii) *ConfChairSite* considers participant site for conference Chairs, (iii) *TrackChairSite* considers participant site for track Chairs, (iv) *ReviewerSite* considers participant site for reviewers.
- 3 Services Categories: (i) *AuthorMgmt* deals with general functionality management regarding authors, (ii) *ConfMgmt* deals with general functionality management regarding conferences, (iii) *ReviewerMgmt* deals with general functionality management regarding reviewers.
- 7 Services Classes: (i) *AuthorMgr* defines specific functionality concerning authors, (ii) *ConfInfoProv* defines specific functionality concerning information providing of conferences, (iii) *ApprovalMgr* defines specific functionality concerning approval process of publications, (iv) *PublishingMgr* defines specific functionality concerning publishing management of proceedings, (v) *SubmisMgr* defines specific functionality concerning submission management of papers, (vi) *ReviewingMgr* defines specific functionality concerning reviewing management of papers, (vii) *ReviewerMgr* defines specific functionality concerning reviewers.

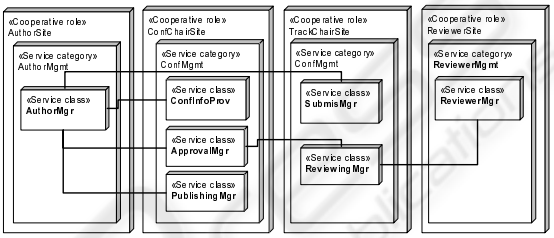

Figure 2: Architecture of the Cooperative Review application.

Several interactions, structured into activities, between Web services would be enabled on this application. Some typical examples of these activities are: (i) Conference Search Activity, (ii) Reviewers Search Activity, (iii) Report Transmission, (iv) Author Notification, etc.

3.2 Application of Reconfig. Actions

In order to better illustrate effects due to application of reconfiguration action we use a Left-Right style (namely, each reconfiguration action is composed by a left side and a right side). In the left side, we describe architectural elements (services and connections) as necessary to apply the rules. In the right side, we describe architectural elements, as wished after application of the rules. In practice, architectural changes yielded by passing from a state to another are carried out by a set of atomic actions (like activation, binding or deactivation of Web services and connections).

A first example illustrating reconfiguration actions for the Cooperative Review application considers duplication of Web services. In fact, many reasons would be behind decision for applying this action, for instance, QoS mismatches as overhead, load balancing, low throughput, etc. As depicted in figure 3 it is possible that in any time many *AuthorMgrServ* services try to address a same instance of *ConfInfoProvServ* service, so it is pertinent to improve performance by deploying new instances of the latter ser-

vice. Hence, the duplication action is applied as described in section 2.3 ($R_1 R_2$).

For all Web services instances defined by variables X, Y, and Z; apply rules R1 and R2 with the following parameters.

Rule	X()	Y()	Z()
R1	i=ConfChairSite j=ConfMgmt k=ConfInfoProv l=*	i'=ConfChairSite j'=ConfMgmt k'=ConfInfoProv l'=new	
R2	i=ConfChairSite j=ConfMgmt k=ConfInfoProv l'!=new	i'=ConfChairSite j'=ConfMgmt k'=ConfInfoProv l'=new	i"=* j"=* k"=* l"=*

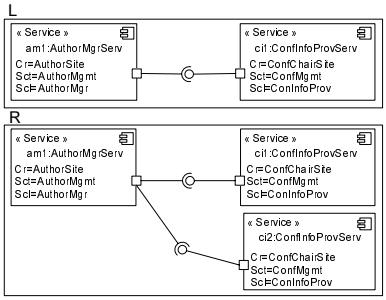

Figure 3: Applying duplication action.

A second example depicting reconfiguration actions for the Cooperative Review system deals with application of the substitution action. Recurrent QoS mismatches as Web services delivering a wrong service, low throughput, or yet denial of service, would carry the reconfiguration module to apply this kind of reconfiguration action. As depicted in figure 4 it is possible that the group of Web services related to the same Service category get in any time a non conforming state, so a pertinent and efficient way of repairing this matters is by substituting all the Web services concerned with this Service category (*ConfMgmt*). It must be clear interactions between services involved at the moment of applying the reconfiguration action, must be taken into account in order to restore them with the new instances of involved services. In this example, this is the case between the *AuthorMgrServ* service and the *ConfInfoProvServ* service, when applying the action of substitution as defined in section 2.3 ($R_1 R_2 R_3$). Many other scenarios would be illustrated by assigning different values of variables concerning each architectural type.

For all Web services instances defined by variables X, Y, and Z; apply rules R1,R2, and R3 with the following parameters.

Rule	X()	Y()	Z()
R1	j=ConfMgmt	j'=ConfMgmt l'=new	
R2	j=ConfMgmt l'!=new	j'=ConfMgmt l'=new	i"=* j"=* k"=* l"=*
R3	j=ConfMgmt l'!=new		

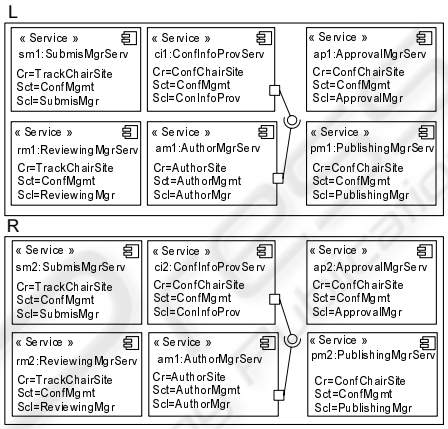

Figure 4: Applying substitution action.

4 RELATED WORK

Many research efforts contribute to the topic considered by this work. Regarding Architecture Description Language (ADL) as a support for achieving self-adaptive or self-configuration software, (Georgiadis et al., 2002) proposed the use of architectural constraints as the basis for self-organising architectures in distributed systems. In this proposal structural specification of architectures is carry out through the Darwin ADL (J. Magee, N. Dulay, S. Eisenbach and J. Kramer, 1995). (Garlan and Schmerl, 2002) depict an approach for self-adaptation of architectures based on the ACME ADL (Garlan et al., 2000). The CHAM ADL (Inverardi and Wolf, 1995) offers an approach in terms of molecules and reactions, where components are represented as atoms connected into molecules. Reconfigurations are specified as reactions. Provided with an appropriate set of reaction rules, it is possible to determine whether a certain reconfiguration is valid or not.

(Le Metayer, 1998) adopted graph grammar for modeling of software architectures and associated reconfiguration. With this formalism, he expresses architectural styles and reasons about conformance of change or evolution with respect to structural constraints.

Regarding frameworks for supporting self-healing systems, (Wile and Egyed, 2004) propose a framework for self-healing systems through monitoring, interpretation, analyzing and reconfiguration of running systems. (Gurguis and Zeid, 2005) propose a solution based on an architecture encompassing a Monitor, an Analyzer, a Planner and an Executive.(Shin, 2005) proposes a strategy for self-healing components in distributed systems based on an architecture with two layers. The services layer provides functional services to the other components, whilst the self-healing layer implements mechanisms for monitoring and repairing.

In the particular case of reconfiguration in Web services, (Baresi et al., 2003) propose a model for a SOA style. Architectures are specified by adapting UML diagrams and reconfiguration is achieved through reconfiguration rules. An important difference with our work is that their model, for representing architectures, is more based on behavioral elements of service-oriented architectures.

The goals in the WS-DIAMOND go beyond these approaches, namely a dynamic reconfiguration of Web service architectures whenever some QoS mismatches occur, considering both functional and non functional features of Web services.

5 CONCLUSION

The architectural aspects aforementioned may be translated and implemented by using the architectural tools being developed at LAAS-CNRS. In concrete, the architectural description, as well as, the basic rules are described by XML files defined across XML Schemas. Initial specification and reconfiguration actions could be mapped into graphs and graph transformation actions, respectively. The main idea is to use a graph transformation engine in order to manage dynamic changes in the Web services application whenever a reconfiguration action is triggered.

The approach herein described enables modeling of dynamic reconfiguration of Web services at the architectural level. Following the goals of the WS-DIAMOND project we consider its integration to the monitoring, diagnosis and repair mechanisms being developed, in order to satisfy requirements of a global self-healing solution (Koopman, 2003). An extended version of the Cooperative Review example might introduce other interacting scenarios by handling additional architectural elements.

REFERENCES

Baresi, L., Heckel, R., Thöne, S., and Varró, D. (2003). Modeling and validation of service-oriented architectures: application vs. style. In *ESEC / SIGSOFT FSE*, pages 68–77.

Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). Towards architecture-based self-healing systems. In *WOSS*, pages 21–26.

Garlan, D., Monroe, R., and Wile, D. (2000). Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems. Leavens, G.T., and Sitaraman, M.(eds)*, pages 47–68. Cambridge University Press.

Garlan, D. and Schmerl, B. (2002). Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA. ACM Press.

Georgiadis, I., Magee, J., and Kramer, J. (2002). Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA. ACM Press.

Gurguis, S. A. and Zeid, A. (2005). Towards autonomic web services: achieving self-healing using web services. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–5, NY, USA. ACM Press.

Inverardi, P. and Wolf, A. (1995). Formal specification and analysis of software architectures using the chemical abstract machine. *In IEEE Transactions on Software Engineering*, 21(4):373–386.

IST (2006). The web services diagnosability, monitoring and diagnostic project. http://wsdiamond.di.unito.it.

J. Magee, N. Dulay, S. Eisenbach and J. Kramer (1995). Specifying distributed software architectures. In *Proceeding of the 5th European Software Engineering Conference, ESEC '95*.

Koopman, P. (2003). Elements of the self-healing system problem space. In *Proceeding of the Workshop on Software Architectures for Dependable Systems*.

Le Metayer, D. (1998). Describing software architecture styles using graph grammars. *In IEEE Transactions on Software Engineering*, 24(7):521–533.

OMG (2005). Unified modeling language specification. Technical Report Infrastructure Version 2.0.formal/05-07-04, Object Management Group.

Shin, M. E. (2005). Self-healing components in robust software architecture for concurrent and distributed systems. *Sci. Comput. Program.*, 57(1):27–44.

Wile, D. S. and Egyed, A. (2004). An externalized infrastructure for self-healing systems. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*. IEEE Computer Society.