Adding Self-healing Behaviour to Dynamic Web Service Composition

ALEXANDRU MOGA
Computer Science Department
Technical University of Cluj-Napoca
G. Baritiu Str. 26-28
ROMANIA

JANOS SOOS
Computer Science Department
Technical University of Cluj-Napoca
G. Baritiu Str. 26-28
ROMANIA

IOAN SALOMIE
Computer Science Department
Technical University of Cluj-Napoca
G. Baritiu Str. 26-28
ROMANIA

MIHAELA DINSOREANU
Computer Science Department
Technical University of Cluj-Napoca
G. Baritiu Str. 26-28
ROMANIA

Abstract:- The dynamic nature of the Web imposes the automation of web service compositions. Since these compositions serve high level business purposes, it is mandatory to address the reliability issue. The autonomic computing paradigm, introduced by IBM, offers a guideline for building self-managing systems. An important principle of this paradigm is the self-healing behaviour which represents the ability of a system to recover from faulty situations. This paper investigates how to increase the reliability of automatic web service compositions by proposing a framework for adding self-healing behaviour. First, we analyze the general domain of autonomic web services and then, we propose a formal description and architecture for adding self-healing behaviour to web service compositions.

Key-words: - web-service composition, self-healing behaviour.

1 Introduction

An important requirement of modern distributed systems is to run in heterogeneous environments. Web services fulfil this requirement through platform independence by using industry standards for discovery (UDDI), description (WSDL) and communication (SOAP). In order to accomplish a greater functionality, web services are integrated into business processes. This involves a high level complexity which must be controlled especially when dynamic composition of web services is considered. The autonomic computing paradigm [1] introduces the concept of self-management, which addresses the problem of complexity in a user-transparent manner.

The fundamental principles of autonomic computing, also called self-CHOP principles [1], are shortly described below:

Self-Configuring – automatic configuration and reconfiguration of the system under various circumstances in variable environments.

Self-Healing – the system recovers from failures and handles unexpected situations.

Self-Optimizing – the system maximizes its performance by monitoring its components and refining their execution.

Self-Protecting – the system must protect itself from attacks that threaten its security and integrity.

An autonomic element consists of an autonomic manager and a managed element. The lifecycle of an autonomic element is called the MAPE-cycle which stands for Monitoring, Analysis, Planning and Execution [1].

In the literature there are several attempts to bring the benefits of autonomic computing in the world of web services. In [5], Gurguis and Zeid use web services for every phase of the MAPE-cycle and implement a publisher-subscriber pattern for communication among them. Baresi et al. [3][4] focus on web service monitoring and propose several corrective decisions in case of deviation from the specification, such as retry the web service invocation, bind to a different web service or locally restructure the process. In [6], Gekas and Fasli define the problem of dynamic web service

composition by using a semantic web service Current tendencies do not address the reliability issue of the dynamic web service compositions. Our approach to increase composition reliability is to combine automatic composition with self-healing behaviour (Fig. 1).

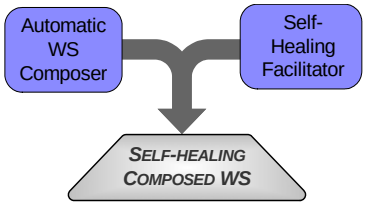

Fig. 1 Self-healing WS Composition

We propose a framework that allows designers to create autonomic managers that deal with dynamic web service compositions. Our final goal is to extend the framework to support other distributed computing paradigms. However, this paper concentrates on the self-healing behaviour of web services. At conceptual level, the self-healing facilitator and the automatic composer are orthogonal components. The outcome of their intersection consists of web service compositions that exhibit self-healing behaviour.

This paper is organized as follows: Section 2 analyses the autonomic behaviour of web services. Section 3 describes the Self-Healing Web Services Framework (SHWSF). The self-healing behaviour algorithm is presented in Section 4, the framework architecture and the implementation details in Section 5. Section 6 concludes the paper and points out future work directions.

2 Autonomic behaviour of web services

In this section we discuss the structure of the autonomic manager and the problem of adding autonomic behaviour to web services.

The autonomic manager implements the MAPE-cycle phases. The *monitoring* phase deals with data gathering. The output of this phase consists of standardized logging data - events or assertion results. The *analysis* phase validates monitoring data based on prior knowledge of the domain and application policies. The *planning* phase calculates the appropriate course of actions in order to optimally solve the problem while the *execution* phase deals with mapping the decisions from the planning phase to concrete actions.

The problem of adding autonomic behaviour to web services addresses the following aspects: web repository.

service lifecycle, complexity of the managed element, modality of facilitating autonomic behaviour and the functional vs. non-functional perspective.

Regarding the *lifecycle of a web service*, monitoring can be done at different phases. During design time, test vectors might give a hint about the problems that may occur when the service is running. Deployment-time monitoring ensures that the bindings to the involved services are done correctly. Run-time monitoring is mandatory when bindings to services are dynamic and occur before each invocation. Monitors can be configured to track and analyse the conformity of web services to their service level agreement (SLA), and to their functional contract specified in WSDL. If services do not satisfy the agreement or fail to comply with their functional contract, corrective actions must be taken. One could retry to invoke the service, rebind it to an equivalent service (if the first one becomes totally inaccessible) or restructure the web process (when there is no equivalent service) [3].

Judging by *complexity*, the managed element consists of an individual or a composed web service. An individual web service is a black box; therefore the only information available is the web service specification and its input/output data. On the other hand, a composed service is made up of more web services forming a business workflow. When static composition is used, the structure of the process is imposed at design-time, before execution. Dynamic composition occurs at run-time and needs semantically extended information about the domain.

Considering the *modality of adding autonomic behaviour to web services*, two possibilities could be explored. The first one assumes that the autonomic manager is a separate component. A regular web service becomes autonomic by interfacing with this component at run-time. The second possibility is to write an autonomic manager for a specific class of service compositions. By generalizing different aspects of autonomic behaviour for a class of service compositions, one could define an abstract manager that should be derived to specific situations.

It is also important to differentiate between *the functional and the non-functional aspects* associated with web services. The functional features of a web service are specified in WSDL and contain the web service capability. The functional contract is verified against each invocation by monitoring the inputs and outputs. The non-functional features are included in the SLA and refer to quality parameters. The difference between the two aspects shows the

boundary between the self-healing principle (where monitoring is focused on the functional features) and the self-optimizing principle (where monitoring deals with the non-functional aspect).

3 Self-Healing Web Services Framework

The SHWS framework consists of the automatic composer, the self-healing facilitator and the associated resources (Fig. 2).

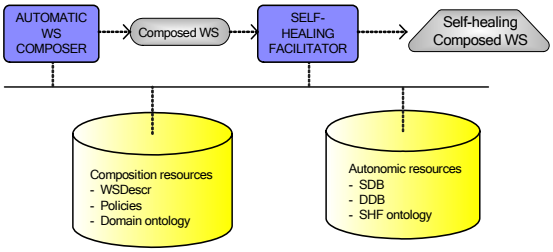

Fig. 2 SHWS framework components

Dynamic composition cannot be separated from the semantic extension of the domain. As shown in Figure 2, the *Automatic Composer* uses a semantic web service repository (*WSDescr*), a set of *policies* and the *domain ontology*. The policies represent web service and process level constraints. The output of the automatic composer is a web service which complies with the imposed policies. The domain of the composed web service is modelled using the ontology.

The main component of the SHWS framework is the *Self-Healing Facilitator* which uses the following resources:

SDB (Symptom Data Base) is part of the knowledge base that contains *symptoms* and *diagnostics*. A symptom corresponds to an event that indicates a malfunction. A diagnostic identifies a problem based on a set of symptoms.

DDB (Decision Data Base). Decisions are inferred from this data base given a certain diagnostic. For each diagnostic there are a set of actions that form a plan P. This data base is also part of the knowledge base.

WSDescr and the set of policies.

SHF ontology is used to model the domain of the self-healing behaviour.

Since dynamic composition of web services is beyond the purpose of this paper, we will concentrate on the self-healing facilitator by exploring three directions. The first one refers to the formal description of the self-healing behaviour. The second one defines the top-view architecture of the SHWS framework, and the last one points out implementation details regarding the integration of the self-healing facilitator with an existing automatic composition framework.

4 Self-healing facilitator

4.1 Self-healing behaviour of web services

Fig. 3 shows the necessary steps involved in facilitating self-healing behaviour while Fig. 4 formally describes the associated algorithm.

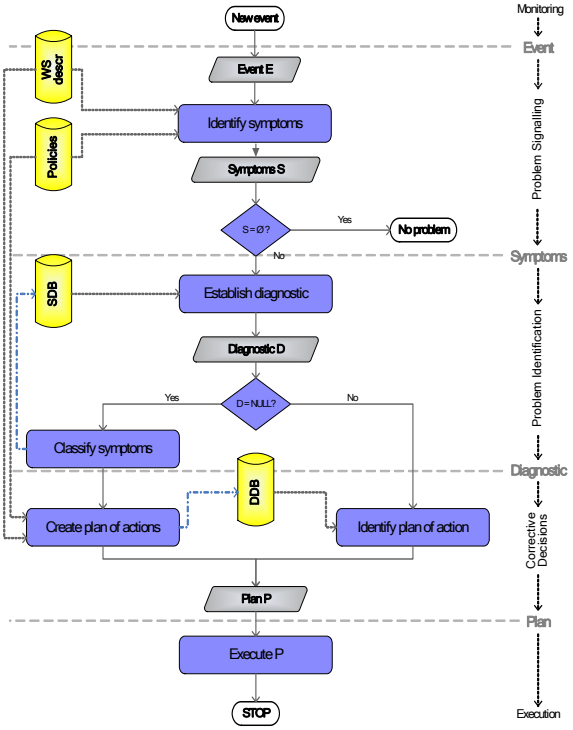

Fig. 3 Self-healing behaviour

The algorithm starts when a new event occurs. An event represents the change of the process execution state and is specified by the event description, event data, the web service and process identifiers. The first phase of the algorithm is the initialization. The symptoms set *S* and the plan *P* are initialized to the empty set and the diagnostic *D* is set to null (lines 1-3). The next phase represents problem signalling (lines 4-10). GET-WS-DESCR() function retrieves the web service description from the *WSDescr* repository. *Policy* contains a set of constraints that results from the combination of the web service level (GET-WS-POLICY()) and process level (GET-PROCESS-POLICY()) policies. VALIDATE() compares the event data with the web service specification and policies, returning a non-empty set of symptoms *S* in case a problem is signalled.

```
SELF-HEALING-FACILITATOR (e)
 1: S ← Ø
 2: D ← null
 3: P ← Ø
 4: wsdescr ← GET-WS-DESCR(wsid[e])
 5: wspolicy ← GET-WS-POLICY(wsdescr)
 6: processpolicy ← GET-PROCESS-
    POLICY(pid[e])
 7: policy ← wspolicy ∩ processpolicy
 8: S ← VALIDATE(e, policy, wsdescr)
 9: if ( S = Ø) then
10:     return "no problem"
11: D ← FIND-BEST-DIAGNOSTIC(S)
12: if (D = null) then
13:     D ← CLASSIFY-SYMPTOMS(S)
14:     P ← GENERATE-PLAN(D)
15: else
16:     P ← FIND-BEST-PLAN(D)
17: for each action a of P do
18:     EXECUTE(a, P)
```

Fig. 4 SHF algorithm for composed web services

FIND-BEST-DIAGNOSTIC() queries *SDB* in order to retrieve the best diagnostic *D* that matches the provided set of symptoms S. In case no diagnostic is found (line 12), CLASSIFY-SYMPTOMS() updates *SDB* with the set of symptoms and a new diagnostic *D* is generated for them. GENERATE-PLAN() creates plan P for diagnostic D. If a diagnostic is found, then FIND-BEST-PLAN() queries *DDB* in order to retrieve the best plan of action P. The final phase is to execute each action from plan P (lines 17-18).

4.2. Generic self-healing behaviour

The SHF algorithm can be slightly modified in order to enable its use on a wider range of distributed systems, resulting in a generic self-healing algorithm. Its integration with other distributed computing paradigms follows the Strategy design pattern [12]. According to this pattern, the system is composed of an invariant part and a changing part. In our case, the invariant is the algorithm while the changing part is represented by the components that support various implementations, such as the policies, the ontological description of the domain, the SHF ontology and the symptom identification, diagnosing, planning and execution components. The generalization of the SHF algorithm is presented in Fig. 5. By generalization, the execution of a distributed system could be considered as a sequence of activities.

```
GENERIC-SELF-HEALING-FACILITATOR (e)
 1: S ← Ø
 2: D ← null
 3: P ← Ø
 4: activity ← GET-ACTIVITY(e)
 5: constraints ← GET-CONSTRAINTS(activity)
 6: S ← VALIDATE(e, activity, constraints)
 7: if ( S = Ø) then
 8:     return "no problem"
 9: D ← FIND-BEST-DIAGNOSTIC(S)
10: if (D = null) then
11:     D ← CLASSIFY-SYMPTOMS(S)
12:     P ← GENERATE-PLAN(D)
13: else
14:     P ← FIND-BEST-PLAN(D)
15: for each action a of P do
16:     EXECUTE(a, P)
```

Fig. 5 Generic SHF algorithm

Each activity produces a set of events that are intercepted by the monitoring component. Compared to the algorithm in Figure 4, *activity* would contain the web service semantic description. The policies have been replaced with the set of constraints associated with the activity that generated the event. Thus, VALIDATE() (line 6) becomes a constraint satisfaction problem. The rest of the algorithm remains unchanged.

5 Framework architecture and implementation

The SHWSF architecture is presented in Fig. 6. The central part of the architecture is the *SHF Core* component which implements the flow of activities presented in Figure 3. These activities, i.e. monitoring, problem signalling and identification, decision making and execution, are all functionally independent. Therefore, their implementation is separated from the SHF Core. Each activity is serviced by a component with a public interface, which facilitates a loosely-coupled environment and encourages reusability.

SHF Core is the invariant part of the system by the fact that the SHF algorithm is hard-coded into it. Also, SHF Core must undertake the tasks of synchronization and communication with the other components.

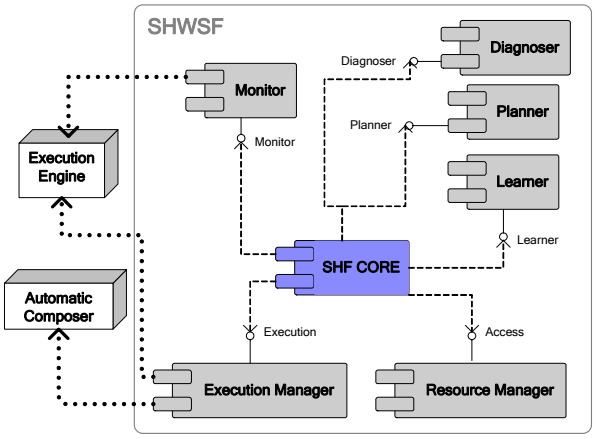

Fig. 6 SHWS framework architecture

The *Monitoring* component (or simply the Monitor) interfaces with the engine in which the web process runs. It intercepts events and stores them in a log file using a standard format. The Monitor can be viewed as a layer built on top of the execution engine. This component is activated when the execution state of the process (composed web service) is changed. The responsibility of the Monitor is to log the event and then notify the SHF Core that a new event has occurred.

The *Diagnoser* component deals with problem signalling and identification. It is activated after the SHF Core is notified of a new event. At this point, the SHF Core formulates a request to the Diagnoser which must validate the event and find a diagnostic in case the validation fails. This component implements the VALIDATE() and FIND-BEST-DIAGNOSTIC() functionalities. The response to the SHF Core can contain one of the following: the diagnostic, "NO PROBLEM" or "UNKNOWN PROBLEM" messages. In the latter case the Diagnoser also returns the set of identified symptoms.

The *Planner* implements the algorithms for decision making. The SHF Core invokes this component only if a diagnostic is formulated in the previous stage. The input is the diagnostic description while the output is a plan of actions P. This component implements FIND-BEST-PLAN(*D*) and GENERATE-PLAN(*D*).

The *Learner* integrates new information into the knowledge base and it implements CLASSIFY-SYMPTOMS(*S*). The inputs of the Learner consist of the set of symptoms while the output is the generated diagnostic. When the SHF Core is notified by the Diagnoser with the "UNKNOWN PROBLEM" message, it invokes the Learner in order to find a diagnostic based on previous experience. The SHF Core coordinates the Planner and the Learner when a new plan is due to be generated.

The *Execution Manager* interfaces with both the execution engine and the automatic composer. It receives from the SHF Core the plan containing corrective actions and executes it. Each action consists of a set of commands which are sent to the execution engine (retry/rebind actions) or to the automatic composer (restructuring actions). The Execution Manager implements EXECUTE().

The *Resource Manager* wraps access to resources such as *SDB*, *DDB*, *Policies* and *WSDescr*. Beside the main inputs and outputs discussed above, each component requests access to resources from the SHF Core which forwards them to the Resource Manager. This component contains data representation wrappers that allow the SHF Core and the other framework components to exchange data in a standard format.

In order to test our conceptual model we use MWSCF (METEOR-S Web Services Composition Framework) [13] for dynamic discovery, selection and composition of web services. METEOR-S uses BPEL abstract processes for specifying the functional requirements. Dynamic composition refers to run-time binding of web services to the abstract process activities. The automatic discovery implies a semantically enhanced UDDI, and the selection phase represents a constraint satisfaction problem. METEOR-S uses the BPWS4J implementation of the BPEL4WS standard specification. METEOR-S defines a layer on top of BPWS4J allowing run-time binding. The result of dynamic composition is an executable BPEL process which runs in the BPWS4J engine [14].

The integration of SHWSF with MWSCF is done by interfacing the Monitoring component and the Execution Manager with the METEOR-S engine. The enhanced UDDI represents the *WSDescr* repository and uses OWL-S representation of web services. The ontological resources presented in Figure 2 are described in OWL while the information stored in SDB and DDB is specified in XML format.

6 Conclusions and future work

This paper has presented a way to improve automatic web service compositions by defining a conceptual model and a framework that includes the necessary steps to facilitate self-healing behaviour. We have also defined an architecture that encourages reusability by separating the SHF algorithm flow from the components that

encapsulate the monitoring, analyzing, planning, execution and resource access logic. We conclude that by adding self-healing behaviour, web service compositions are more reliable and adaptable to faulty situations.

Through generalization, we have shown that the conceptual model and the SHF algorithm can be extended to support other distributed system paradigms, such as mobile agents. Our future work involves the refinement of the algorithm to the extent where it can be used as a standalone component. In addition, using the concepts presented in this paper we envision building facilitators for self-configuration, self-optimization and self-protection.

References

[1] The Vision of Autonomic Computing, IBM, http://www.research.ibm.com/autonomic.

[2] IBM Autonomic Computing, "Automating problem determination: A first step toward self-healing computing systems", October 2003.

[3] L. Baresi, C. Ghezzi and S. Guinea, "Towards Self-healing Service Compositions".

[4] L. Baresi, C. Ghezzi and S. Guinea, "Smart Monitors for Composed Services", *ICSOC'04,* New York, USA, November 15–19, 2004.

[5] S. A. Gurguis and A. Zeid, "Towards Autonomic Web Services: Achieving Self-Healing Using Web Services", *DEAS 2005*, St. Louis, Missouri, USA, May 21, 2005.

[6] J. Gekas and M. Fasli, "Automatic Web Service Composition Using Web Connectivity Analysis", *W3C Workshop on Frameworks for Semantics in Web Services 2005,* Position Paper.

[7] D. Bridgewater, "Standardize messages with the Common Base Event model", http://www-128.ibm.com /developerworks/library/ac-cbe1/.

[8] K. Birman, R. van Renesse and W. Vogels, "Adding High Availability and Autonomic Behavior to Web Services", *Proceedings of the 26th International Conference on Software Engineering*, ICSE 2004.

[9] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull and M. Macella, "Towards Automatic Web Service Discovery and Composition in a Context with Semantics, Messages, and Internal Process Flow (A Position Paper)", April 29, 2005.

[10] K. Verma and A. P. Sheth, "Autonomic Web Processes", *Proceedings of the Third International Conference on Service Oriented Computing*, ICSOC 2005.

[11] V. Kappor, "Services and Autonomic Computing: A Practical Approach for Designing Manageability", *Proceedings of the 2005 IEEE International Conference on Services Computing*, SCC 2005.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*".

[13] METEOR-S Web Services Composition Framework, http://lsdis.cs.uga.edu/proj/meteor/mwscf/mwscf.html.

[14] R. Aggarwal, K. Verma, J. Miller, W. Milnor, "Dynamic Web Service Composition in METEOR-S", Technical Report, LSDIS Lab, University of Georgia, Athens.

[15] Business Process Execution Language for Web Services version 1.1 (BPEL4WS), http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[16] Business Process Execution Language for Web Services Java Run Time (BPWS4J), http://www.alphaworks.ibm.com/tech/bpws4j.

[17] OWL-S: Semantic Mark-up for Web Services, http://www.w3.org/Submission/OWL-S.

[18] Web Ontology Language (OWL).