Journal of Universal Computer Science, vol. 20, no. 8 (2014), 1130-1151
submitted: 4/4/14, accepted: 23/7/14, appeared: 1/8/14 © J.UCS

Extending Policy Languages for Expressing the Self-Adaptation of Web Services

Haithem Mezni
(University of Jendouba, SOIE, Jendouba, Tunisia
haithem.mezni@gmail.com)

Walid Chainbi
(National School of Engineers, SOIE, Sousse, Tunisia
walid.chainbi@gmail.com)

Khaled Ghedira
(Higher Institute of Management, SOIE, Tunis, Tunisia
khaled.ghedira@isg.rnu.tn)

Abstract: With the growing demand on Web Services, self-adaptation in the highly-dynamic environment is becoming a key capability of service-based systems. As a solution for Web services to provide added value and high QoS, combining self-* and policies allows reducing management complexity and effectively drives adaptation. Also, providers must participate in the self-adaptation process as they are aware of the capabilities of their offered services and exceptions that may occur. Despite the important role of service providers, existing approaches did not address this major issue. Thus, the description of self-adaptive Web services must not be limited to functional and QoS data. To address these issues, we extend the WS-Policy framework to represent capabilities and requirements of self-* Web services. We also extend UDDI in order to store and manage service policies, as the current UDDI model does not offer these capabilities. Finally, we propose an ECA-based planning mechanism to specify decision making in the self-adaptation process.

Keywords: Web services, self-adaptation, Autonomic computing, Quality of service, WS-Policy, UDDI, ECA rules
Categories: C.2.4, H.3, H.3.5

1 Introduction

Modern distributed systems require to dynamically take into account at runtime the changes in the users' needs and the execution environment variations in order to improve QoS [Zouari et al. 2014]. In this context, and with the increasing adoption of SOA, service-based systems face an unprecedented level of change and dynamism. Therefore, there is a growing need for efficient mechanisms for service self-adaptation. Although extensive work has been done in this area, existing approaches usually use predefined adaptation strategies to deal with abnormal behavior and do not consider all properties of SOA actors that are subject to change [Kazhamiakin et al. 2010]. Moreover, abnormal behavior should be seen from the perspective of service providers as they are aware of the capabilities and requirements of their published services and the exceptions that may occur. In order for Web services to provide high QoS, providers must participate in the self-adaptation by implementing their specific adaptation policies. Thereby, relieving adaptation system from the situation where no recovery alternative is applicable in a given problem situation.

To tackle these issues, we adopt the autonomic computing paradigm [Kephart and Chess 2003] and we consider Web services and UDDI registries as autonomic systems [Chainbi et al. 2012] i.e., systems endowed with self-* capabilities including self-configuration, self-healing, self-optimization, and self-protection. Additionally, to effectively drive adaptation, we propose a rich information model to allow describing autonomic Web services (AWS) based, not only on functional and QoS data, but also on providers' recommendations. This new kind of information may be used at runtime and gives the adaptation system additional alternatives (i.e. adaptation plans) to manage the executing service and to allow a better precision in driving adaptation.

Since managing Web services requires a structured representation of their capabilities, we adopt a policy-based approach in order to endow services with self-* behavior. Policies have proven their popularity and acceptance in research and academia. They have long been employed in the management of traditional distributed systems [Boutaba and Aib 2007] and become a key for SOA management. In our work, policies have a core position. They are used at different levels including service description, QoS monitoring, service self-adaptation, etc. The adoption of policy specification languages has been suggested in several approaches as a solution to describe QoS. Specifications like WS-Policy [Bajaj et al. 2007] are widely deployed to address SOA issues, as they offer a means to express QoS requirements and capabilities in Web service systems [Phan et al. 2008]. However, existing extensions to WS-Policy [Tosic et al. 2007, Chhetri and Kowalczyk 2010, Badidi and Esmah 2011] did not express information necessary to service self-adaptation. Also, there is no detail about self-* policies and how policies are specified and processed.

The main objective of this paper is to extend WS-Policy in order to express requirements for the run time adaptation of Web services. However, knowing that WS-Policy provides only textual descriptions, this raises a question about how to use adaptation policies in the self-adaptation process. For this reason, a part of our proposed work uses the ECA rule-based approach to convert adaptation policies to executable ECA rules [Bassiliades and Vlahavas 1997]. ECA rules are extracted from the policy documents and can, then, be executed by autonomic service managers.

The rest of this paper is organized as follows. Section 2 introduces a rich AWS information model. Section 3 presents our extension called AWS-Policy. Section 4 presents an extension to UDDI information model. In section 5, we show how specific adaptation policies are used in the self-adaptation process. Section 6 validates the proposed approach through a set of experimental results. In section 7, we discuss the relevance of our approach compared to related work. The last section is devoted to the conclusion and the future work.

2 Autonomic Web Service Information Model

In our previous work, we have shown that an AWS involves two parts: a managed service and an autonomic manager, which endows the invoked service with self-* mechanisms (see [Fig. 1]). AWSs collaborate with autonomic registries (autonomic systems composed of the traditional UDDI registry and a controller named autonomic registry manager, which has the ability to manage the registry content) to ensure self-adaptation and to allow updating registry content [Chainbi et al. 2012].

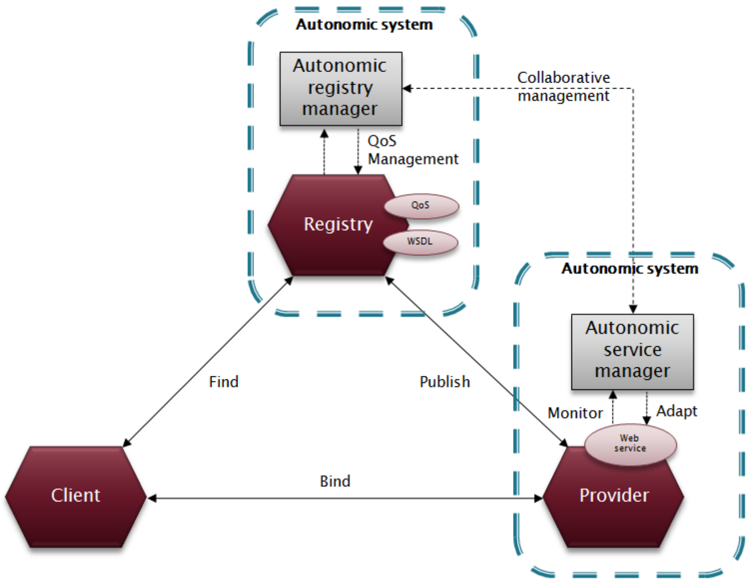


Figure 1: An autonomic Web service system architecture

However, to effectively ensure self-adaptation, an AWS must have a rich description of its capabilities. We believe that functional and non-functional data are not sufficient to effectively drive self-adaptation, and AWSs need to be specified with additional information, such as specific adaptation policies. Unlike other approaches that design their management systems based on predefined management rules, we propose two types of adaptation policies and we give service providers the possibility to define and register their recommended adaptation policies into UDDI registry, in order to effectively drive the autonomic managers in the self-adaptation process. In our work, a service behavior may be adjusted using what we call *default* and/or *specific adaptation plans*. While default plans are predefined actions included in the implementation of autonomic managers as internal knowledge and executed to adapt any failed service, specific plans are priority actions defined by providers at publication time (e.g. specific mediation solutions, invoking a trusted service).

For this purpose, we characterize self-* Web services by three types of information: *Functional properties*, *QoS properties*, and *Specific adaptation plans*. Since managing Web services requires a structured representation of their capabilities, we adopt the WS-Policy framework to treat them as policies [Bajaj et al. 2007]. WS-Policy is a specification that allows Web Services to advertise their characteristics in a flexible and extensible grammar using XML format. A policy is defined as a collection of alternatives which is, itself, defined as a collection of assertions. Assertions are the basic building blocks of policies and are used to represent a requirement, capability or a behavior of a Web service.

However, WS-Policy does not allow defining capabilities and requirements of self-adaptive Web services, and still lacks mechanisms for semantic description and matching of service properties. We believe that the more Web services are attached with QoS and adaptation policies, the more important is the automation of the service execution and adaptation. Indeed, by taking these policies into account, a Web service behavior is always adjusted based on self-* capabilities (e.g. self-healing policies) allowing, thus, a constant execution, even in case of QoS constraints' violation, and a high availability while meeting QoS constraints (e.g. response time and reliability).

To remedy these restrictions, we propose to extend WS-Policy in order to represent AWS capabilities and requirements necessary to trigger self-adaptation and to adjust the behavior of a managed Web service.

3 AWS-Policy: An Extension for Autonomic Web Services' Description

The *AWS-Policy* specification allows providers to define the behavior of their offered services. It makes Web services expose autonomic behaviors in response to changes detected in the execution environment and predicted by a service provider (e.g. self-healing and self-configuration behavior, in case of substitution and mediation actions). Each construct in AWS-Policy is considered as a service behavior (e.g. *unavailability* QoS attribute, *timeout* event, *SLA Negotiation* action, *bandwidth* context). Separation of concern and flexibility of AWS-Policy allows the services' implementations to focus only on the functional capabilities, whereas policies are responsible for providing and guaranteeing a certain behavior and a constant service execution, while taking into account the changing conditions and the required QoS.

AWS-Policy offers an extensible grammar based on XML format. The building blocks in AWS-Policy are assertions, which allow defining a QoS preference, a QoS capability, an event or a self-adaptation action. Assertions are grouped using a set of policy operators which can be recursively nested to express complex policies.

3.1 QoS Policies

This section presents the AWS-Policy document model that may be used by service providers to describe the QoS capabilities of their offered Web services, or by users to specify their QoS requirements. The QoS policies XML schema is shown in [Fig. 2].

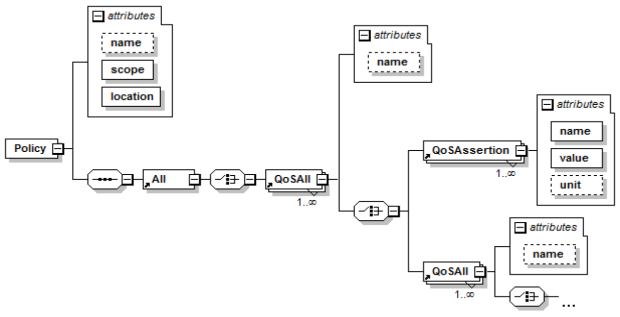
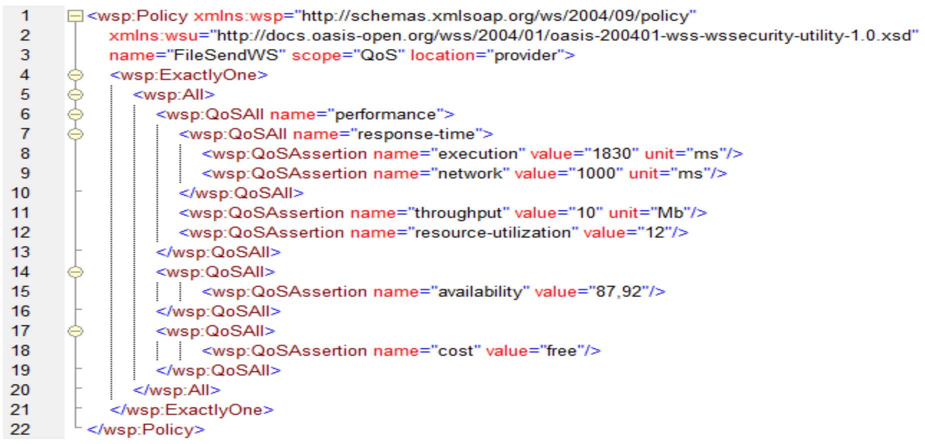


Figure 2: QoS policies XML schema.

As shown in [Fig. 2], an AWS must be described by a set of QoS attributes. Each one is described in a *QoSAssertion* element and is characterized by three properties (*name*, *value*, and *unit*) containing, respectively, the name, the value, and the measure unit of the QoS attribute. Since providers may describe their services using complex QoS attributes, we define the *QoSAll* alternative element to group the QoS attributes into a composite one. Fig. 3 shows an example of AWS-Policy QoS document.



```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    name="FileSendWS" scope="QoS" location="provider">
    <wsp:ExactlyOne>
        <wsp:All>
            <wsp:QoSAll name="performance">
                <wsp:QoSAll name="response-time">
                    <wsp:QoSAssertion name="execution" value="1830" unit="ms"/>
                    <wsp:QoSAssertion name="network" value="1000" unit="ms"/>
                </wsp:QoSAll>
                <wsp:QoSAssertion name="throughput" value="10" unit="Mb"/>
                <wsp:QoSAssertion name="resource-utilization" value="12"/>
            </wsp:QoSAll>
            <wsp:QoSAll>
                    <wsp:QoSAssertion name="availability" value="87,92"/>
            </wsp:QoSAll>
            <wsp:QoSAll>
                    <wsp:QoSAssertion name="cost" value="free"/>
            </wsp:QoSAll>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 3: QoS policies of the FileSendWS Web service.

An overall view shows that the offered *FileSendWS* service is a free Web service characterized by a high degree of performance and availability.

3.2 Specific Adaptation Policies

As shown in [Fig. 4], a specific adaptation policy is defined as a set of plan alternatives. Each one contains two policy alternatives: *ExactlyOneEvent* and *ExactlyOnePlan*. The first element specifies the events that may affect service execution, whereas the second element describes specific adaptation plans that may be triggered by the events defined in the *ExactlyOneEvent* alternative.

AWS-Policy allows specifying execution events in a flexible manner. Indeed, self-adaptation may be triggered by a primitive or a composite event. For primitive events, we define the *EventAssertion* element which may contain a set of attributes depending on the mentioned event. Composite events are of type “AND” ($e_1 \wedge e_2 \wedge \ldots e_n$). They require that all of the sub-events must occur during service execution so that they may trigger some potential adaptation actions. A composite event is expressed using the *AllEvent* policy alternative. Since different events may trigger the same adaptation actions, we also consider the events of type “OR” ($e_1 \vee e_2 \vee \ldots e_n$) which means that at least one of these events must occur during invocation so that it may trigger adaptation. Such events are expressed using the *ExactlyOneEvent* alternative.

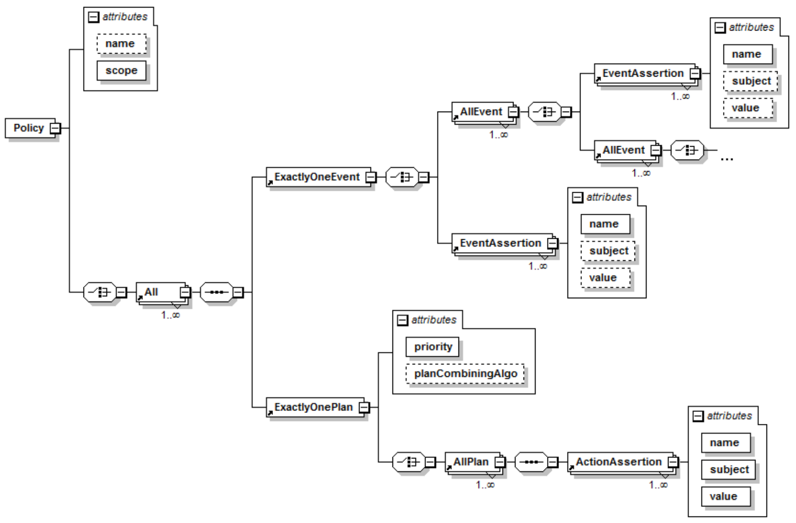
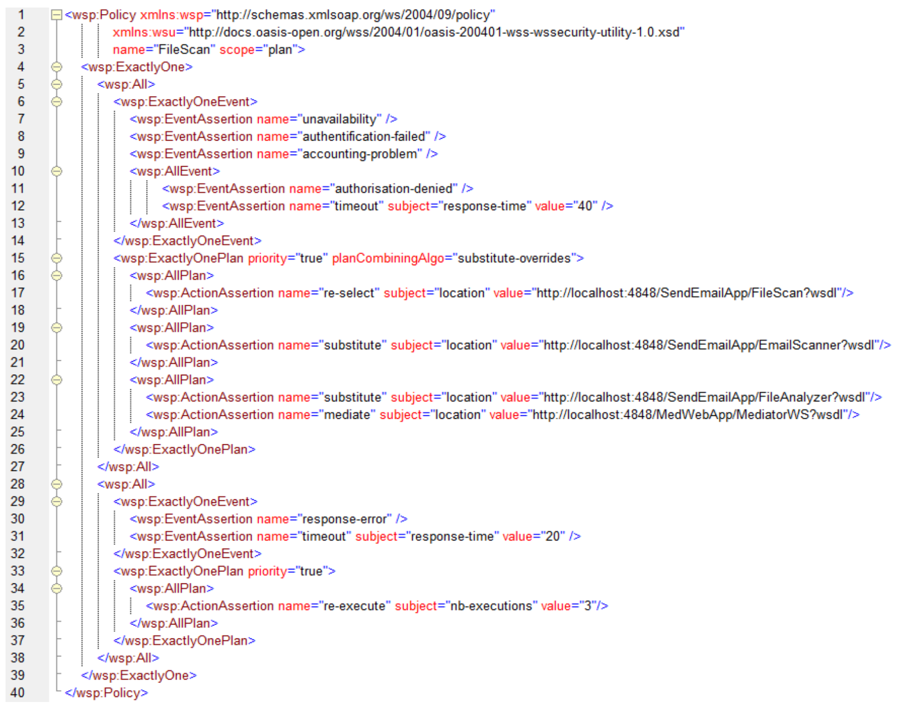


Figure 4: Specific adaptation policies XML schema.

The second part of the self-adaptation policy is an *ExactlyOnePlan* alternative defining self-adaptation actions that may be executed when one of the events described in the *ExactlyOneEvent* alternative occurs. The *ExactlyOnePlan* element indicates that only one specific plan must be executed by the autonomic manager. We define the *ActionAssertion* assertion to describe an atomic self-adaptation action. Since an adaptation plan may be composed by a set of adaptation actions, it is expressed by grouping *ActionAssertion* elements in the *AllPlan* alternative. The *AllPlan* alternative requires all primitive actions to be executed in the specified order.

The *priority* attribute in the XML schema is used by a provider to indicate that a specific adaptation action has to be firstly executed by the autonomic manager, or executed when the default adaptation plans fail to resolve a service failure. This allows a better precision in choosing adaptation actions. Another important attribute is the *planCombiningAlgo* which is used by planning agents to select between more than one suitable plan and to resolve conflicts between self-adaptation plans. Details on plan combining mechanisms will be presented in [section 5.2].

The AWS-Policy document in [Fig. 5] describes specific adaptation policies of a published service. An overall view shows that the service provider defines a set of self-adaptation plans for two possible events. The first set of plans may be triggered when a "binding fault" (lines 8 to 13) is detected by the autonomic manager or when the executing service is not available. The second plan may be triggered in case of an "execution fault" (see "response error" or "timeout" in lines 30 and 31).



```xml
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
      name="FileScan" scope="plan">
  <wsp:ExactlyOne>
    <wsp:All>
      <wsp:ExactlyOneEvent>
        <wsp:EventAssertion name="unavailability" />
        <wsp:EventAssertion name="authentification-failed" />
        <wsp:EventAssertion name="accounting-problem" />
        <wsp:AllEvent>
            <wsp:EventAssertion name="authorisation-denied" />
            <wsp:EventAssertion name="timeout" subject="response-time" value="40" />
        </wsp:AllEvent>
      </wsp:ExactlyOneEvent>
      <wsp:ExactlyOnePlan priority="true" planCombiningAlgo="substitute-overrides">
        <wsp:AllPlan>
          <wsp:ActionAssertion name="re-select" subject="location" value="http://localhost:4848/SendEmailApp/FileScan?wsdl"/>
        </wsp:AllPlan>
        <wsp:AllPlan>
          <wsp:ActionAssertion name="substitute" subject="location" value="http://localhost:4848/SendEmailApp/EmailScanner?wsdl"/>
        </wsp:AllPlan>
        <wsp:AllPlan>
          <wsp:ActionAssertion name="substitute" subject="location" value="http://localhost:4848/SendEmailApp/FileAnalyzer?wsdl"/>
          <wsp:ActionAssertion name="mediate" subject="location" value="http://localhost:4848/MedWebApp/MediatorWS?wsdl"/>
        </wsp:AllPlan>
      </wsp:ExactlyOnePlan>
    </wsp:All>
    <wsp:All>
      <wsp:ExactlyOneEvent>
        <wsp:EventAssertion name="response-error" />
        <wsp:EventAssertion name="timeout" subject="response-time" value="20" />
      </wsp:ExactlyOneEvent>
      <wsp:ExactlyOnePlan priority="true">
        <wsp:AllPlan>
          <wsp:ActionAssertion name="re-execute" subject="nb-executions" value="3"/>
        </wsp:AllPlan>
      </wsp:ExactlyOnePlan>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Figure 5: Specific self- adaptation policies of the FileScan Web service.

Fig. 5 also shows that, in case of a detected binding fault (binding denied may occur if the authorization component denies the access, that is, either authorization denied, authentication failed, or accounting problems occur), the autonomic manager features two possible priority self-adaptation actions: *reselect* a new composition for the executing composite service or, in case of restarting failure, *substitute* the service with one of two preferred and trusted services (a mediation action may be performed in case of incompatibility in services' interfaces). Fig. 5 also shows that the provider recommends *re-executing* the service operations in case of a delivered incorrect result.

The next section reveals how AWS policies are published in the service registry.

4 Enhancing UDDI Information Model with Policies

In our previous work, we have shown the way autonomic registries interact with each other in the execution environment, to allow exchanging and processing different kinds of management tasks, such as QoS updating and service discovery [Chainbi et al. 2012].

However, representing and storing providers' policies is a main challenge, as the current UDDI model does not offer these capabilities and as service metadata are basic resources in driving the adaptation process. The problems of storing QoS and specific adaptation policies in UDDI may be resolved using *tModel* concept. tModels are core components of UDDI. They represent unique concepts or constructs and they are used to describe compliance with a specification, a concept, a category, or an identifier system. Each tModel should contain an *overviewURL*, which references a document that describes the tModel and its usage in more details [OASIS 2004]. We use tModels as a categorization solution to facilitate service discovery by the autonomic registry managers (also called ARMs). Since tModels are considered as containers for references to service descriptions, they may be used to represent QoS policies and adaptation policies in UDDI.

Fig. 6 shows an extension of the original UDDI information model with two new data structures for describing AWS related information. Unshaded boxes and solid lines represent data structures in the original model and the association between them, whereas shaded boxes and dashed lines illustrate our extension. *QoS policies* data structure represents QoS information of a managed Web service, whereas *specific adaptation policies* data structure represents actions defined by a service provider that will eventually be used in priority to manage service deviations.

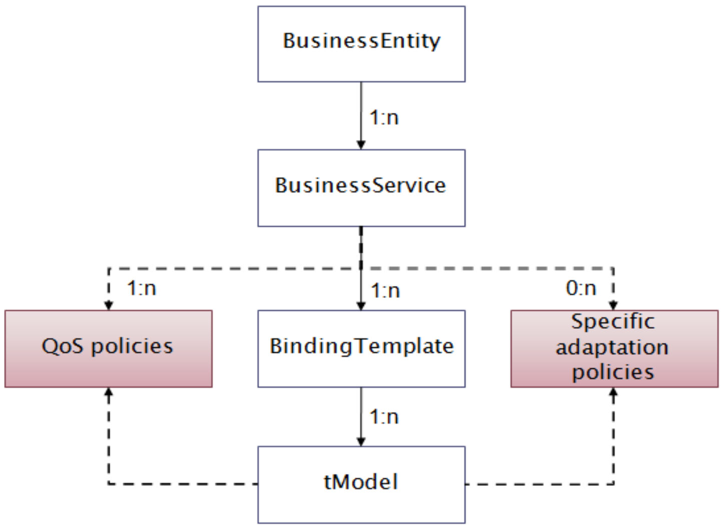
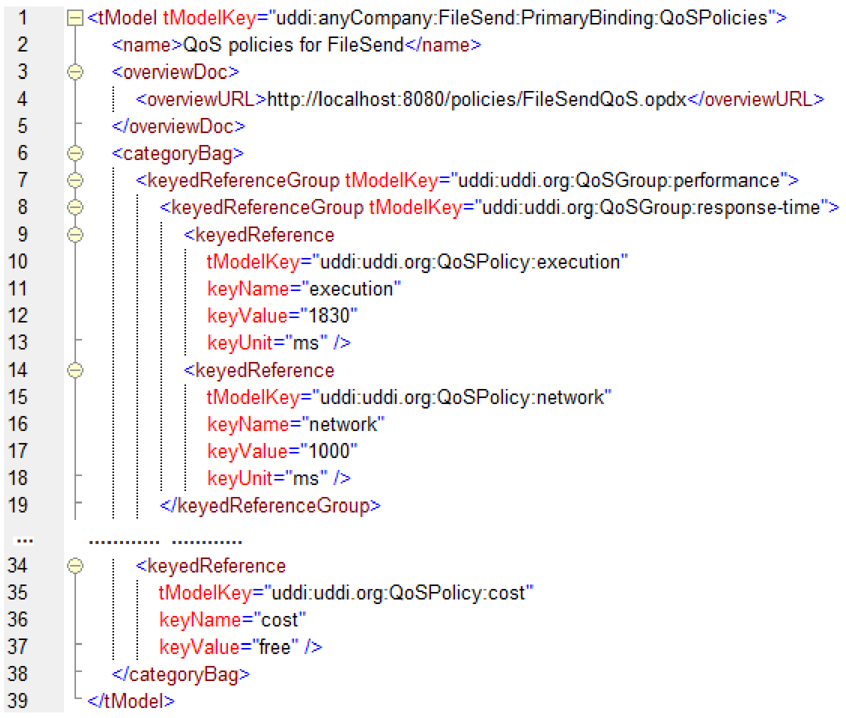


Figure 6: Extended UDDI information model.

To these ends, when a Web service is published in the UDDI registry, a set of tModels is created to represent the QoS and specific adaptation policies of this service. To include Web service policies in UDDI, different ways may be envisioned, such as defining a tModel that points to an external resource or defining a tModel containing multiple categories, each reflecting a different policy. The first method has the disadvantage of requiring navigation to external resources to retrieve information. In addition, service data are not stored in UDDI in this case, which makes their management a difficult task. This is the case, for example, when services' providers may restrict access to their servers, prohibiting ARMs to update QoS values.

We adopt the first method to store specific adaptation policies, and we integrate QoS policies using the second method. This choice is justified by the fact that specific adaptation policies do not interest the user in the discovery process and are only processed by autonomic managers. Furthermore, they do not require a representation or a classification scheme like QoS policies, which need to be stored in the registry in order to facilitate service discovery. For this purpose, specific adaptation policies are considered as external resources and referenced in the *overviewURL* tag of a tModel. The UDDI *bindingTemplate* may, then, contain a reference to these policies, which is added to its *tModelInstanceDetails* collection.

Regarding QoS, we make a mapping between QoS policies and UDDI, in order to give service providers a way to store their QoS descriptions into UDDI registry. The QoS policies description is referenced in a tModel and each QoS assertion in the policy document is mapped to a *category* in the tModel structure (see [Fig. 7]). In UDDI, *categoryBag* tag allows to add categorization information into UDDI data structures to make a given entity a member of one or more categories. Therefore, ARMs can find the desired entity based on some classification scheme. In our work, categorization may be ensured on the basis of QoS policies. The categoryBag element in the QoS policies tModel acts as a collection of *keyedReference* structures (line 6), each containing a single QoS attribute-based categorization and characterized by a set of properties (*keyName*, *keyValue*, and *keyUnit*) describing respectively the name, value and measure unit of a QoS attribute (lines 11, 12, and 13).



```
1   <tModel tModelKey="uddi:anyCompany:FileSend:PrimaryBinding:QoSPolicies">
2     <name>QoS policies for FileSend</name>
3     <overviewDoc>
4       <overviewURL>http://localhost:8080/policies/FileSendQoS.opdx</overviewURL>
5     </overviewDoc>
6     <categoryBag>
7       <keyedReferenceGroup tModelKey="uddi:uddi.org:QoSGroup:performance">
8         <keyedReferenceGroup tModelKey="uddi:uddi.org:QoSGroup:response-time">
9           <keyedReference
10            tModelKey="uddi:uddi.org:QoSPolicy:execution"
11            keyName="execution"
12            keyValue="1830"
13            keyUnit="ms" />
14          <keyedReference
15            tModelKey="uddi:uddi.org:QoSPolicy:network"
16            keyName="network"
17            keyValue="1000"
18            keyUnit="ms" />
19        </keyedReferenceGroup>
...  ............ ............
34      <keyedReference
35        tModelKey="uddi:uddi.org:QoSPolicy:cost"
36        keyName="cost"
37        keyValue="free" />
38    </categoryBag>
39  </tModel>
```

Figure 7: tModel with QoS policies.

To categorize a managed service as being described by a composite QoS attribute, we propose to add a *keyedReferenceGroup* element (lines 7 and 8) to the categoryBag. This element is a collection of keyedReference structures that logically belong together and correspond to the component QoS attributes for the categorization.

5 Self-adaptation Planning: from AWS-Policy to ECA Rules

In this section, we propose a planning technique that allows driving Web service self-adaptation based on predefined and specific self-adaptation policies. Despite the importance of this phase in the autonomic cycle, few approaches have been proposed to deal with the problem of autonomic decision making [Lu 2011].

As we mentioned above, providers can define their specific self-adaptation plans in the form of policies. This raises a question about how to use these policies by the autonomic manager to adapt a service. The effective use of adaptation policies in the autonomic cycle requires that the policies be captured and translated into actions within the autonomic manager. Since adaptation policies are textual descriptions, we propose to convert them to executable rules that may be executed by autonomic managers on their rules' engines. Rules are increasingly being used to specify a variety of situations, such as business needs, conventional behavior, and policies [Sing and Huns 2005]. Rules are also desirable because they are executable, unlike textual descriptions or even some formal specifications. There is no additional step of converting specifications into formally executable implementations. We adopt the classical ECA (Event-Condition-Action) form [Bassiliades and Vlahavas 1997], where the event is a triggering condition. Such rules can be read as:

on *event* **if** *condition* **then** (*perform*) *action*

As an example of policies' transformation, the policy elements contained in the AWS-Policy document in [Fig. 5] will be converted as follows:

If the binding is denied or the authentication fails, then composition should be reselected instead of substituted whenever possible.

Here, *binding fault* or *authentication failure* are the events. The condition is true (*priority="true"*), and the action is re-selection of the composition (line 16) or *substituting* the failed service with specific one (lines 19 and 22).

Next, we present a transformation algorithm that allows generating self-adaptation ECA rules from the AWS-Policy documents.

5.1 Specific Adaptation Policies Transformation

In order to effectively monitor and adapt the executing Web service, the autonomic service manager starts by interacting with the autonomic registry (see [Fig. 1]) to get the AWS-Policy documents describing QoS and specific self-adaptation policies. This step is important as the autonomic manager must be aware of the capabilities of its managed service. Self-adaptation plans defined by the provider are then extracted and stored in the form of ECA rules. Connection between possible events (AWS-Policy constructs) and the corresponding adaptation actions is established when generating the different parts of each new rule. First, a specific self-adaptation rule is defined as follows:

Definition 1 (*Self-adaptation rule set*). A self-adaptation rule set for a managed Web service, $SARS^{WS}$, is a finite set of ECA rules, R = (E,C,A,P,RCA) such as :

1. E = {e | e = e_1;…;e_n}. Event set *E* consists of one or more serializations of events, where the serialization is expressed as e_1; . . . ; e_n.
2. Condition set *C* consists of the disjunction of zero or more conjunctions of condition predicates. Each conjunction is expressed as $c_1 \wedge \ldots \wedge c_m$, and condition predicate c_j is expressed as r1 θ r2, where θ is an operator from the set $\{=, \neq, <, \leq, >, \geq\}$, and r_i, i = 1, 2, is a constant, an event variable, or a QoS attribute.
3. Action set *A* consists of a serialization of primitive self-adaptation actions, where the serialization is expressed as a_1; . . . ;a_n.
4. Priority P is a Boolean variable indicating whether or not the primitive actions in *A* are executed in priority instead of triggering default self-adaptation rules.
5. Rule combing algorithm *RCA* defines a strategy of combing a set of concurrable rules in a single decision. *RCA* indicates if the self-adaptation rule overrides the other rules in the $SARS^{WS}$ rule set.

The $SARS^{WS}$ rule set contains two self-adaptation sub-sets: the $SARS^{WSspecific}$ rule set which contains specific self-adaptation rules transformed from an AWS-Policy document defined by the provider, and the $SARS^{WSdefault}$ rule set containing self-adaptation actions predefined for each autonomic manager instance.

Definition 2 (*Self-adaptation constraints*). Self-adaptation constraints SAC^{WS} for a managed Web service is a power set of the self-adaptation actions' set such that AC = $\{a_1, \ldots a_n \mid a_1 \wedge \ldots \wedge a_n$ are not allowed to be executed simultaneously $\} \in SAC^{WS}$.

Definition 3 (*Rule conflict*). Let $SARS^{WStrig}$ be a rule set containing the rules that are triggered on the occurrence of a primitive event. It is said that $SARS^{WStrig}$ has a rule conflict if two or more service actions of $SARS^{WStrig}$ are contained in one of the element of SAC^{WS}.

Our algorithm for specific self-adaptation policies transformation (see [Fig. 8]) starts with an event alternative set EAS^{WS} and a plan alternative set PAS^{WS}, as inputs. The output of the algorithm is a specific self-adaptation ECA rule set $SARS^{specific}$.

The algorithm starts by parsing the set of events that may trigger self-adaptation plans recommended by the service provider (line 2). For each event, the algorithm creates the event part of the new ECA rule, by transforming event assertions and alternatives to simple (line 12) or composite events (lines 4-7). Next, the algorithm extracts, for each generated event, the condition part, according to the constraints of the specified events (e.g. "response time" subject for the "timeout" event) (lines 8-9). Note that the priority attribute in the plan alternatives is considered as a condition when executing the policies transformation process (line 25). Then, for each plan in PAS^{WS}, the action part of the new rule R_{new} is created according to the nature of the plan alternative, by taking the union of the simple actions specified in each plan in PAS^{WS} (lines 20-23). Finally, the algorithm, based on the set of *self-adaptation constraints* (i.e. adaptation actions that are not allowed to occur at the same time such as re-execute operation and skip operation) checks if the new rule has any conflict and eventually creates a conflict resolution rule for the detected conflict (line 27).

Performance of the policy transformation algorithm depends on the size of the events and actions' sets, and on the self-adaptation constraints' set. Time complexity of the algorithm is $O(n_e n_a n_c)$, where n_e, n_a and n_c denote the number of events, the number of adaptation actions and the number of adaptation constraints, respectively. Function checkConflict (R_{new}, $SARS^{WS}$) requires $O(n_a n_c)$ since for each adaptation action, the function checks the conflict with each action in the constraint set.

Algorithm 1 Specific_SelfAdaptation_Policies_Transformation (**in** (EAS^{WS}, PAS^{WS}), **out** ($SARS^{WSspecific}$))

Input: An event alternative set EAS^{WS} and a plan alternative set PAS^{WS}
Output: A provider defined self-adaptation rule set $SARS^{WSspecific}$, where $SARS^{WSspecific}$:= { (Evt, Cond, Act, Pr, RCA) | triggering event Evt, provider constraint Cond, provider actions Act, provider priority Pr and rule combining algorithm RCA}

```
1: $SARS^{WSspecific}$:= {};
2: for each E ∈ $EAS^{WS}$ do
3:     Cond := {};
4:     if E.Type = ALL then
5:         Evt := {};
6:         for each EA ∈ E do
7:             Evt :=Evt∪ EA;
8:             C := new Condition (EA.Subject, EA.SubjectValue);
9:             Cond := Cond ∪ C;
10:        end for
11:        else
12:        Evt := E;
13:        Cond := new Condition (Evt.Subject, Evt.SubjectValue);
14:    end if
15:    for each P ∈$PAS^{WS}$ do
16:        $R_{new}$ := new ECARule ();
17:        $R_{new}$.setEvent(Evt);
18:        $R_{new}$.setCondition (Cond);
19:        Act := {};
20:        for each A∈ P do
21:            Action Ac := new Action (A.Name, A.Subject, A.SubjectValue);
22:            Act.addAction (Ac);
23:        end for
24:        $R_{new}$.setAction (Act);
25:        $R_{new}$.setPriority (P.Priority());
26:        $R_{new}$.setCombiningAlgo (P.CombiningAlgo());
27:        checkConflict ($R_{new}$, $SARS^{WS}$);
28:        $SARS^{WSspecific}$ :=$SARS^{WSspecific}$ U $R_{new}$;
29:    end for
30: end for
31: return $SARS^{WSspecific}$;
```

Figure 8: Algorithm Specific_Adaptation_Policies_Transformation.

5.2 Plan Selection and Conflict Resolution of Self-adaptation ECA Rules

As shown in the previous section, specific self-adaptation policies are transformed to ECA rules to be used by autonomic managers in the self-adaptation process. As a result, some rules may have inconsistencies with each other or with the predefined self-adaptation plans and may cause conflicts when triggered and executed.

To select the suitable self-adaptation plan, we propose a solution that allows reconciling self-adaptation policies when their evaluation is contradictory. The proposed AWS-Policy specification supports what we call *plan combining algorithms*, which allow resolving self-adaptation rule conflicts by representing a way of combining multiple adaptation plans into a single plan. The proposed method is based on "*event dominance*" and "*self-adaptation rule (plan) overriding*". Event dominance is defined to analyze the relationship between the events that occur during service execution. It is used to identify the rules that can be triggered together by a dominant event. Rule overriding consists of combining a set of rules that can be triggered by the same event into a single rule. These basic concepts are defined as follows:

Definition 4 (*Event dominance*). Let e^i and e^j be two serializations of events. e^i is said to be dominant over e^j(represented as $e^i \rightarrow^{E} e^j$) if the occurrence of e^i implies the occurrence of e^j. That is, if $e^i = e^i_1;\ldots;e^i_n$, and $e^j = e^j_s;\ldots;e^j_t$ $(1 \leq s \leq t \leq n)$.

Definition 5 (*Rule overriding*). Let $SARS^{WStrig}$ be a self-adaptation rule set such that $SARS^{WStrig} \subset \mathrm{SARS}^{WS}$ and contains the rules triggered by occurred events, $SARS^{WStrig} = \{R \mid R = R_1;\ldots;R_n\}$ and let R_i and R_j be two triggered rules. R_i is said to override R_j (represented as $R_i \rightarrow^{O} R_j$) if R_i belongs to $SARS^{WStrig}$, R_i is triggered and each rule R_j($1 \leq j \leq n$ and $j \neq i$) contained in $SARS^{WStrig}$ is rejected. $\forall R_i \in \mathrm{SARS}^{WStrig}$, $\forall R_j \in \mathrm{SARS}^{WStrig}$, $\exists a$, $a \in A_i \wedge a \notin A_j$. That is, if $A_j = a^j_s;\ldots;a^j_t$, $A_i = a^i_1;\ldots;a;\ldots;a^i_n$.

To avoid conflict or violation of service constraints, the autonomic manager must take into account only the rules triggered by the dominant events. If none of the occurred events dominates each other, a technique called "*self-adaptation rule overriding*" is used to process the plans in order to choose a single plan.

We propose a dynamic conflict resolution algorithm which is based on event dominance and self-adaptation rule overriding. The algorithm is inspired from the combining algorithms defined for XACML access control policy language [XACML Technical Committee 2008]. To manage conflicts between access control rules, XACML supports different combining algorithms, each representing a way of combining multiple decisions, which are often conflicting, into a single decision. Possible decisions are: permit, deny, indeterminate and not applicable.

To this purpose, the planning technique uses the *Priority* and *PlanCombiningAlgo* attributes [section 3.2] to indicate an algorithm for combining the possible self-adaptation plans from the evaluation of a set of rules. Such algorithms can be:

- *Retry-overrides:* If any rule evaluates to *retry*, then the final generated plan is also re-executing the operations of the managed service. For example, When a detected event triggers three possible plans (e.g. retry, substitute and skip), and if the *planCombiningAlgo* attribute is set to "retry-overrides", then the autonomic manager decides to re-execute the Web service operation.
- *Substitute-overrides:* If any rule evaluates to *Invoke*, then the final decision is also substitute Web service.
- *FirstPlan-applicable:* In this case, the self-adaptation rule is picked among the rules; according to the order given in the AWS-Policy document.

Our proposed algorithm for the selection and conflict resolution of self-adaptation ECA rules (see [Fig. 9]) starts with an event set Evt^{WS}, a self-adaptation rule set $SARS^{WS}$ and the plan combining algorithm RCA, as inputs. The output of the algorithm is a single decision R_{select} from the multiple self-adaptation rules.

Knowing that the self-adaptation rules' set $SARS^{WS}$ is triggered by a set of events Evt^{WS}, a rule set $SARS^{dom}$, which is included in $SARS^{WS}$, is only triggered by the most dominant events Evt_{dom} (line 1). The rule set $SARS^{dom}$ can be refined to a single self-adaptation rule R_{select}, which is the only rule that meets two criteria: "event dominance" and "plan overriding". In other words, R_{select} is triggered by the most dominant events and is the result of combining the rules, in $SARS^{dom}$, based on the given combining algorithm (lines 15 or 17).

```
Algorithm 2 SelfAdaptation_RuleSelection_ConflictResolution (in ($Evt^{WS}$, $SARS^{WStrig}$, RCA), out ($R_{select}$))
Input: An event set $Evt^{WS}$; a self-adaptation rule set $SARS^{WStrig}$ and a rule combining algorithm RCA.
Output: a single self-adaptation ECA rule $R_{select}$ representing a decision from multiple adaptation rules.
1: $Evt_{dom}$ :=findDominantEvents ($Evt^{WS}$);
2: if $Evt_{dom} \neq \emptyset$ then
3:     $SARS^{dom}$ := {};
4:     for each R $\in SARS^{WStrig}$ do
5:         exists := true;
6:         for each e $\in Evt_{dom}$ do
7:             if not Trigger (e, R) then
8:                 exists := false;
9:                 break;
10:            end if
12:        end for
13:        if exists then $SARS^{dom} := SARS^{dom} \cup R$;
14:    end for
15:    $R_{select}$ :=PlanCombiningAlgo ($SARS^{dom}$, RCA);
16:    else
17:    $R_{select}$ :=PlanCombiningAlgo ($SARS^{WStrig}$, "FirstPlan-Applicable");
18: end if
19: return $R_{select}$;
```

```
Function PlanCombiningAlgo(SARS, RCA)
1: for each Rule R ∈ SARS do
2:     Act :=R.Actions();
3:     for each A∈ Act do
4:         if A.CombiningAlgo = "FirstPlan-Applicable" OR  A.CombiningAlgo = RCA then
5:             return R;
6:         end if
7:     end for
8:     if R.Priority = true and $R_{priority}$ = null then
9:         $R_{priority}$ := R;
10:        else if $R_{default}$ = null then $R_{default}$ := R;
11:    end if
12: end for
13: if $R_{priority} \neq$ null then return $R_{priority}$;
14: if $R_{default} \neq$ null then return $R_{default}$;
15: return "Indeterminate";
```

Figure 9: Algorithm SelfAdaptation_RulesSelection_ConflictResolution

The algorithm starts by calling the function *findDominantEvents* (*Evt^{WS}*) in order to get the most dominant events among the triggering events in the Evt^{WS} event set (line 1). For instance, let e_1, e_2 and e_3 be the occurred events and suppose that a set of rules {R_1, R_2, R_3, R_4, R_5} are trigged as follows: R_1 by {e_1, e_2}, R_2 by {e_1}, R_3 by {e_1, e_3}, R_4 and R_5 by {e_1, e_2, e_3}. The most dominant events in this case are {e_1; e_2; e_3}. Consequently, *$SARS^{dom}$* includes only the self-adaptation rules R_4 and R_5 (lines 6-13).

The next step of the algorithm is to check the self-adaptation rules in *$SARS^{dom}$*. This is by evaluating each resulting rule against the plan combining algorithm. To do that, the algorithm calls the *planCombiningAlgo* function (line 15) in order to combine the *$SARS^{dom}$* rule set (R_4 and R_5) into a single decision.

Performance of the algorithm depends on the size of the triggered rule set and the occurred event set. Time complexity of the algorithm is O($n_r n_e^2 n_a$), where n_r, n_e and n_a denote the number of triggered self-adaptation rules, the number of events and the number of adaptation actions, respectively. Creating the $SARS^{dom}$ rule set takes the time O($n_r n_e^2$) as the size of Evt_{dom} set is smaller than n_e. Function planCombiningAlgo (SARS, RCA) takes the time O($n_r n_a$) as the size of $SARS^{dom}$ is smaller or equal to n_r.

6 Implementation and Experimental Results

To validate our goal of making Web services self-*, we have used JADE platform to implement autonomic managers. The autonomic registry is implemented by extending Apache jUDDI which is an open source UDDI implementation compliant with version 2.0 specification. We have used JADE platform and JAX-WS to implement autonomic registry managers as agent-based Web service clients to the UDDI registry. To support AWS-Policy descriptions, we have implemented the main policy management aspects by extending Apache Neethi 2.0, which provides a general framework for developers to use WS-Policy.

We have evaluated the availability, reliability and response time of a set of Web services. We started the invocation without injecting any autonomic capability. Then we performed the same steps by associating an autonomic manager to each service.

Compared with normal execution, self-healing actions (retry or substitute in our case) have reduced the execution failures and consequently have improved the availability of the managed Web service. As an example, the EmailValidator Web service is executed 1000 times and fails to return a result 143 times (see [Fig. 10]). Using an autonomic manager to invoke Web service functionalities has allowed resolving 137 detected deviations from the 143 occurring events. However, the contract violation still happens (6 invocation failures in these experiments). This is explained by the failure of some generated self-healing plans.

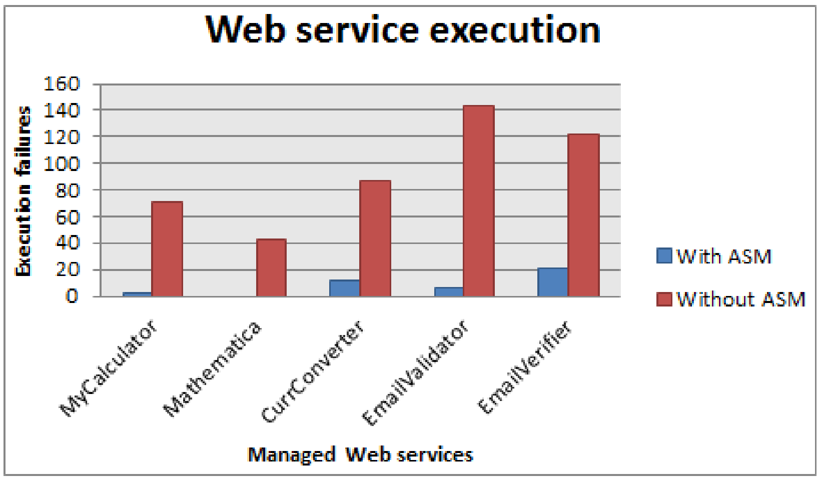


Figure 10: Evaluation of Web service execution with autonomic behavior.

Regarding availability and response time (see [Fig. 11]); we can see that, without autonomic behavior, the response time, in most cases, is higher than the required response time (increased by 104 milliseconds). In the same way, availability is not constant when we vary the number of invocations. Self-healing actions have improved the availability of *EmailValidator* (between 82% and 97%). As shown in [Fig. 11 (b)], during the trials, the availability is increased by 14% and more.

We can also see that self-optimization improves the response time as autonomic manager continuously looks for service opportunities. However, when executing a self-healing action, the autonomic service can exceed the response time fixed in the original contract (e.g. case of 50 and 60 queries). This is explained by the cost of performing such action (i.e. time taken to select and execute self-healing actions).

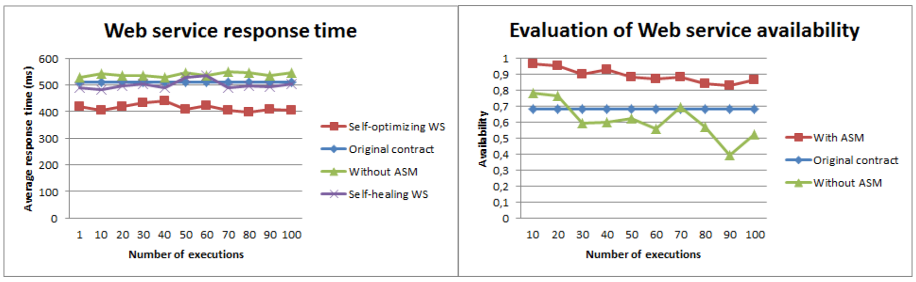


Figure 11: (a) Web service response time (b) Web service availability.

We have compared our work to the approaches proposed in [Ben Halima et al. 2008, Cardellini and Iannucci. 2010]. In the case of the QOSH middleware [Ben Halima et al. 2008], the execution time with a “service monitor” is higher than the response time returned without a service monitor (increase between 7.5% and 37.22% and in some cases 43.6%). In our work, self-healing actions have increased the response time by 7.2%, only in few cases. Regarding the MOSES system [Cardellini and Iannucci 2010], 75% of the time is wasted in analyzing data, unlike our approach, which transforms self-adaptation policies into executable rules, before starting invocation, to reduce the processing time during autonomic execution.

We have also evaluated the results of using default and specific adaptation plans in the autonomic execution (see [Fig. 12]). We have measured the time taken by autonomic managers to decide about an adaptation and to execute the generated plan (in these experiments, retry execution, skip execution, and substitute a failed service).

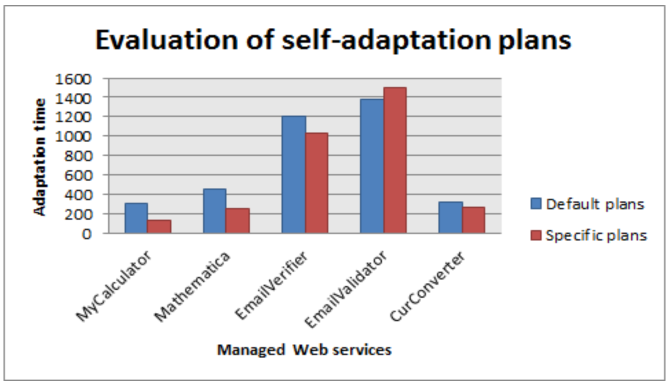


Figure 12: Evaluation of default and specific self-adaptation plans.

We can see that a specific plan–driven adaptation requires a time lower than the time spent by the ASM to adapt a Web service using predefined plans. For example, in case of a predefined substitution plan, the ASM has to start a new Web service discovery process that consists of searching in the whole registry. However, using a specific adaptation plan in which the provider has defined a set of trusted substituting services, the ASM can rapidly locate the WSDL file and thus, perform substitution without starting a new expensive discovery process. For the five Web services, we note that the gain in self-adaptation time varies between 14% and 44,22%, except for the EmailValidator Web service (see [Tab. 1]).

Web services	**Self-adaptation failures**			**Self-adaptation time (ms)**		
	default	**specific**	**total failures**	**default**	**specific**	**gain (%)**
MyCalculator	2	0	2	100	70	30
Mathematica	0	0	0	450	250	44,22
CurConverter	10	2	12	1200	1031	14,08
EmailValidator	6	0	6	1683	1802	-7,07
EmailVerifier	17	4	21	122	101	17,21

Table 1: Evaluation of Default and Specific Self-adaptation plans

Tab. 1 gives an idea about the nature of plans that caused adaptation failure and shows the importance of involving providers in the self-adaptation process. Unlike specific self-adaptation plans, default plans are frequently prone to failures. This can be explained by the fact that providers are aware of the capabilities and requirements of their published services and exceptions that may occur. Consequently, they may predict the possible deviation and fix the suitable self-adaptation actions. For instance, the effectiveness of specific adaptation plans for EmailValidator and EmailVerifier Web services is about 100% and 96,72%, respectively.

However, although our planning technique has significantly improved the availability of the managed services (see [Fig. 10]) and seems to be effective in terms of decision and adaptation quality, it presents some limitations regarding the time spent in plan selection (especially for services with short execution time) and requires a considerable processing time when the decision making complexity increases.

In the future, we plan to improve the planning algorithm in order to optimize the selection process of the self-adaptation rules and to reduce the adaptation time.

7 Related Work

Several approaches have been proposed to deal with Web service self-adaptation. These approaches discuss various issues, such as service interactions [Kongdenfha et al. 2009, Taher et al. 2011], monitoring and prevention of SLA violations [Fugini and Siadat 2010, Mahbub and Spanoudakis 2010, Schmieders et al. 2011], proactive adaptation based on online testing [Hielscher et al. 2008, Sammodi et al. 2011], multi-layered monitoring [Mos et al. 2009, Guinea et al. 2011], etc.

Regarding policy-based management, WS-Policy has been extended in several works to describe QoS [Tosic et al. 2007, Chhetri and Kowalczyk 2010, Badidi and Esmah 2011]. However, WS-Policy is not used to specify adaptation policies and existing approaches lack mechanisms for using and managing the specified policies during the execution.

A recent trend is to complement approaches for reactive adaptation with proactive capabilities [Baresi et al. 2012]. Different solutions have been proposed such as equipping particular execution points in a composition with a set of alternative behaviors [Leitner et al. 2010, Aschoff and Zisman 2011], the use of testing to anticipate problems in service compositions and trigger adaptation requests [Hielscher et al. 2008, Tosi et al. 2009, Metzger et al. 2010], the combination of reactive and proactive adaptation to support self-protection and self-healing [Na et al. 2010], the use of semi-Markov models for performance predictions [Dai et al. 2009], the use of a composition tree to determine the impact of each service in a composition on its overall performance [Bodenstaff et al. 2009], risk-driven management [Ma et al. 2014]. However, these approaches lack reactive behavior and do not support changes that may occur in any part of the composition. In addition, they lack dynamism regarding test cases and adaptation strategies, when a composition is modified, in order to take into account changes and new requirements for the prediction process.

Recently, some approaches have been proposed to allow a monitoring and adaptation across multiple layers [Gjørven et al. 2008, Mos et al. 2009, Guinea et al. 2011, Zengin et al. 2011]. Their aim is to combine and correlate monitoring data from different sources and to avoid potential conflicts that may arise due to uncoordinated adaptations in different layers. These approaches use different techniques such as event monitoring and logging (e.g. [Zeginis et al. 2011]), detection of event patterns, use of dependency graphs, correlation and mapping between events and adaptation strategies (e.g. [Popescu et al. 2012]), etc. The major limitation of these approaches is that the control of execution is performed at each layer in isolation, and the events are processed independently of each others. Consequently, the monitored data are not correctly analyzed, which may lead to the wrong identification of the source of problem. Also, monitoring is often based on static event correlation rules. Finally, cross-layer approaches do not take into account properties and needs of other layers. This leads, in most cases, to incompatibility and conflict problems.

Our approach is different as it encompasses the main requirements for adaptive service-oriented systems [Metzger and Marquezan 2011], in particular autonomic decision making, the way Web service capabilities and adaptation requirements are specified, the involvement of different actors (autonomic registries, service providers and autonomic services themselves) in the self-adaptation process, self-adjustment of the adaptation system through the dynamic integration of providers' policies.

8 Conclusion

In this paper, we have proposed an approach for policy-based self-adaptation of Web services. As a main contribution, we have proposed AWS-Policy, an extension for the description of autonomic Web services. The extension, based on a rich Web service information model, allows not only specifying QoS capabilities, but also provider's recommendations. Indeed, as part of our work, service providers have the ability to participate in the self-adaptation process by publishing their specific adaptation policies, which will be used by autonomic managers during service execution.

To allow representing and storing AWS policies as well as achieving their self-management, we have extended the original UDDI information model with new data structures. We have also proposed a planning technique, which allows transforming specific adaptation policies to executable ECA rules, in order to enable the autonomic managers to determine an order of the actions that achieve the specified goal.

In the future, we intend to eliminate the limitations of this work through the introduction of semantics in all self-management steps. Indeed, semantics play an important role in the autonomic service behavior, and self-adaptation requires a semantic understanding of QoS and autonomic capabilities. AWS-Policy allows only for a syntactic description which may lead to inefficient matching of service policies. For this reason, we will integrate our *AWS-Ont* ontology [Mezni 2014], which is a complete ontology that covers all the behavioral aspects of self-* Web services. Using this ontology brings autonomic managers to a common conceptual space and helps to apply reasoning mechanisms to find a better match. *AWS-Ont* allows to reason about Web services' policies and to carry out more intelligent tasks on behalf of service providers. We suggest adding semantics to AWS-Policy, using extensibility in policy assertions and alternatives. Also, a well-defined semantic matching algorithm is being implemented to check the compatibility between QoS policies. In this case, comparison between AWS-Policy constructs is performed based on assertions' QNames and semantic extensions, i.e. domain concepts of our *AWS-Ont* ontology.

We also look for enhancing the process of creating specific self-adaptation policies by giving providers the possibility to define their recommended plans based on existing plans published by other providers. The idea of discovering autonomic Web services based on their self-* capabilities is underway.

Finally, for the future Internet, cross-layer self-adaptation is an emerging research issue in service-based systems. We intend to inject self-* properties in multiple layers, in order to support autonomic execution across the whole service technology stack.

References

[Aschoff and Zisman 2011] Aschoff, R. R., Zisman, A.: "QoS-driven Proactive Adaptation of Service Composition"; Proc. Int. Conference on Service Oriented Computing, (2011), 421-435.

[Badidi and Esmah 2011] Badidi, E., Esmah, L.: "A Scalable Framework for Policy-based QoS Management in SOA Environments"; J.S (Journal of Software), 6, 4 (2011), 544-553.

[Bajaj et al. 2007] Bajaj, S., et al.: "Web Services Policy 1.5 – Framework"; W3C Candidate Recommendation 04 September 2007. http://www.w3.org/TR/ws-policy/.

[Baresi et al. 2012] Baresi, L., Georgantas, N., Hamann, K., Issarny, V., Lamersdorf, W., Metzger, A., Pernici, B.: "Emerging Research Themes in Services-oriented Systems"; Proc. 2012 Annual SRII Global Conference, (2012), 333-342.

[Bassiliades and Vlahavas 1997] Bassiliades, N., Vlahavas, I.: "DEVICE: Compiling Production Rules into Event-driven Rules Using Complex Events"; J.IST (Journal of Information and Software Technology), 39, 5 (1997), 331–342.

[Ben Halima et al. 2008] Ben Halima, R., Drira, K., Jmaiel, M.: "A QoS-oriented Reconfigurable Middleware for Self-Healing Web Services"; Proc. IEEE International Conference on Web Services, (2008), 104-111.

[Bodenstaff et al. 2009] Bodenstaff, L., Wombacher, A., Reichert, M., Jaeger, M. C.: "Analyzing Impact Factors on Composite Services"; Proc. IEEE International Conference on Services Computing, (2009), 218-226.

[Boutaba and Aib 2007] Boutaba, R. Aib, I.: "Policy-based Management: A Historical Perspective"; J.NSM (Journal of Network and Systems Management), 15, 4 (2007), 447-480.

[Chainbi et al. 2012] Chainbi, W., Mezni, H., Ghedira, K.: "AFAWS: An Agent-based Framework for Autonomic Web Services"; J. MAGS (Journal of Multiagent and Grid Systems), IOS Press, 8, 1 (2012), 45-68.

[Cardellini and Iannucci. 2010] Cardellini, V., Iannucci, S.: "Designing a Broker for QoS-driven Runtime Adaptation of SOA Applications"; Proc. IEEE International Conference on Web Services, (2010), 504-511.

[Chhetri and Kowalczyk 2010] Chhetri, M. B., Vo, B. Q., Kowalczyk, R.: "Policy-based Management of QoS in Service Aggregations"; Proc. 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, (2010), 593-595.

[Dai et al. 2009] Dai, Y., Yang, L., Zhang, B.: "QoS-driven Self-healing Web Service Composition Based on Performance Prediction"; J.CST (Journal of Computer Science and Technology), 24, 2 (2009), 250-261.

[Fugini and Siadat 2010] Fugini, M., and Siadat, H.: "SLA contract for Cross-layer Monitoring and Adaptation". In S. Rinderle-Ma, S. Sadiq, and F. Leymann, editors, Business Process Management Workshops, Springer Berlin/Heidelberg, 43, (2010), 412-423.

[Gjørven et al. 2008] Gjørven, E., Rouvoy, R., Eliassen, F.: "Cross-layer Self-adaptation of Service-oriented Architectures"; Proc. 3rd workshop on Middleware for service oriented computing, Leuven, Belgium, (2008), 37-42.

[Guinea et al. 2011] Guinea, S., Kecskemeti, G., Marconi, A., Wetzstein, B.: "Multi-layered Monitoring and Adaptation"; Proc. 9th International Conference on Service-Oriented Computing, Paphos, Cyprus, (2011), 359-373.

[Hielscher et al. 2008] Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M.: "A Framework for Proactive Self-adaptation of Service-based Applications Based on Online Testing"; Proc. 1st European Conference on Towards a Service-Based Internet, (2008), 122-133.

[Kazhamiakin et al. 2010] Kazhamiakin, R., et al.: "Adaptation of Service-based Systems"; Service Research Challenges and Solutions for the Future Internet. Springer Berlin Heidelberg, (2010), 117-156.

[Kephart and Chess 2003] Kephart, J. O., Chess, D. M.: "The Vision of Autonomic Computing"; IEEE Computer, 36, 1 (2003), 41-50.

[Kongdenfha et al. 2009] Kongdenfha, W., Motahari-Nezhad, H. R., Benatallah, B., Casati, F., Saint-Paul, R.: "Mismatch Patterns and Adaptation Aspects: A Foundation for Rapid Development of Web Service Adapters"; Trans. on Services Computing, 2, 2, (2009), 94-107.

[Leitner et al. 2010] Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: "Monitoring, Prediction and Prevention of SLA Violations in Composite Services"; Proc. IEEE International Conference on Web Services, (2010), 369-376.

[Lu 2011] Lu, Q.: "Autonomic Business-driven Decision Making for Adaptation of Web Service Compositions"; Proc. 7th IEEE World Congress on Services, USA, (2011), 73-76.

[Ma et al. 2014] Ma, S. P., Yeh, C., Chen, P. C.: "Service Composition Management: A Risk-Driven Approach"; J. UCS (Journal of Universal Computer Science), 20, 3, (2014), 302-328.

[Mahbub and Spanoudakis 2010] Mahbub, K., Spanoudakis, G.: "Proactive SLA Negotiation for Service-based Systems"; Proc. IEEE 6th World Congress on Services, (2010), 519-526.

[Metzger et al. 2010] Metzger, A., Sammodi, O., Pohl, K., Rzepka, M.: "Towards Proactive Adaptation with Confidence: Augmenting Service Monitoring with online Testing"; Proc. ICSE Workshop on Software Engineering for Adaptive and Self-managing Systems, (2010), 20-28.

[Metzger and Marquezan 2011] Metzger, A., Marquezan, C. C.: "Future Internet Apps: The next wave of adaptive service-oriented systems?"; Proc. Towards a Service-Based Internet, Springer Berlin Heidelberg, (2011), 230-241.

[Mezni 2014] Mezni, H.: "Towards Trustworthy Service Adaptation: An Ontology-based Cross-layer Approach"; Proc. 5th IEEE International Conference on Software Engineering and Service Science, Beijing, China, (2014), To appear.

[Mos et al. 2009] Mos, A., Pedrinaci, C., Rey, G. A., Gomez, J. M., Liu, D., Vaudaux-Ruth, G., Quaireau, S.: "Multi-level Monitoring and Analysis of Web-scale Service-based Applications"; Proc. Int. Conf. on Service-Oriented Computing, Stockholm, Sweden, (2009), 269-282.

[Na et al. 2010] Na, J., Zhang, B., Zhang, X., Zhu, Z., Li, D.: "Two-stage Adaptation for Dependable Service-oriented System"; Proc. Int. Conf. on Service Sciences, (2010), 143-147.

[OASIS 2004] OASIS: "UDDI Version 3.0.2"; UDDI Spec Technical Committee, (2004). http://uddi.org/pubs/uddi_v3.htm

[Phan et al. 2008] Phan, T., Han, J., Schneider, J. G., Ebringer, T., Rogers, T.: "A Survey of Policy-based Management Approaches for Service-oriented Systems"; Proc. 19th Australian Conference on Software Engineering, (2008), 392-401.

[Popescu et al. 2012] Popescu, R., Staikopoulos, A., Liu, P., Brogi, A., Clarke, S.: "A Formalised Taxonomy-driven Approach to Cross-Layer Application Adaptation"; ACM Transactions on Autonomous and Adaptive Systems, 7, 1 (2012), 7-36.

[Schmieders et al. 2011] Schmieders, E., Micsik, A., Oriol, M., Mahbub, K., Kazhamiakin, R.: "Combining SLA Prediction and Cross-layer Adaptation for Preventing SLA Violations". Proc. 2nd Workshop on Software Services: Cloud Computing and Applications based on Software Services, Timisoara, Romania, (2011), 1-8.

[Sammodi et al. 2011] Sammodi, O., Metzger, A., Franch, X., Oriol, M., Marco, J., Pohl, K.: 'Usage-based Online Testing for Proactive Adaptation of Service-based Applications"; Proc. IEEE 35th Annual Computer Software and Applications Conference, (2011), 582-587.

[Sing and Huns 2005] Singh, M.P., Huns, M.N.: "Service-oriented Computing: Semantics, Processes and Agents"; Wiley, Chichester (2005).

[Taher et al. 2011] Taher, Y., Parkin, M., Papazoglou, M. P., Van den Heuvel, W. J.: "Adaptation of Web Service Interactions Using Complex Event Processing Patterns"; Proc. Service-Oriented Computing, Springer Berlin Heidelberg, (2011), 601-609.

[Tosi et al. 2009] Tosi, D., Denaro, G., Pezze, M.: "Towards Autonomic Service-oriented Applications"; J. AC (Journal of Autonomic Computing), 1, 1 (2009), 58-80.

[Tosic et al. 2007] Tosic, V., Erradi, A., Maheshwari, P.: "WS-Policy4MASC - A WS-Policy Extension Used in the MASC Middleware"; Proc. IEEE International Conference on Services Computing, (2007), 458-465.

[XACML Technical Committee 2008] XACML Technical Committee: "eXtensible Access Control Markup Language (XACML) Version 2.0"; Specification Errata 29, OASIS, (2008).

[Zeginis et al. 2011] Zeginis, C., Konsolaki, K., Kritikos, K., Plexousakis, D.: "ECMAF: An Event-based Cross-layer Service Monitoring and Adaptation Framework"; Proc. Service-Oriented Computing - ICSOC'11 Workshops, Springer, (2011), 147-161.

[Zeginis et al. 2012] Zeginis, C., Konsolaki, K., Kritikos, K., Plexousakis, D.: "Towards Proactive Cross-layer Service Adaptation"; Proc. WISE, Lecture notes in computer science, Springer, Paphos, Cyprus, 7651, (2012), 704–711.

[Zengin et al. 2011] Zengin, A., Kazhamiakin, R., Pistore, M.: "CLAM: Cross-Layer Management of Adaptation Decisions for Service-Based Applications"; Proc. IEEE International Conference on Web Services, (2011), 698-699.

[Zouari et al. 2014] Zouari, M., Diop, C., Exposito, E.: "Multilevel and Coordinated Self-management in Autonomic Systems based on Service Bus". J. UCS (Journal of Universal Computer Science), 20, 3, (2014), 431-460.