Policy Support for Business-oriented Web Service Management

Stephen Gorton and Stephan Reiff-Marganiec

Department of Computer Science
University of Leicester
University Road,
Leicester LE1 7RH
United Kingdom
Email: {smg24, srm13}@le.ac.uk

Abstract

Policies have been adopted for many reasons within web services and Service-oriented Architecture in general. However, while they are a favoured method of management, this only occurs at the service level and in the software domain. Policies already exist in a narrow variety more focussed on service properties such as authorisation. Business metrics are often overlooked when selecting a service to satisfy a need and these are often different to metrics used by the standard software engineer. As a significant number of web services become available, more emphasis needs to be placed on management of services in the business domain. The web service protocol stack provides only a hint of business management techniques on top of any orchestration or composition mechanisms. In this paper, we propose a policy framework that can be used to express business requirements for web services, at a business level that is more abstract than the current high-level composition and orchestration technologies.

1. Introduction

The deployment of software as agile services available on a network is the core idea behind the loosely-coupled, open standards-based Service-oriented Architecture (SoA), of which Web Services [1] is an implementation. Research continues into this realm that contains technologies for service discovery, description and composition. However, Casati *et al* [6] note that as a substantial number of Web Services become available, so the attention shift will be from service infrastructure to service management.

Often, the enterprise architectural level of web services is regarded as a mangement layer, with technologies such as BPEL[1] providing a means to composition and orchestration of composite services. However, this approach (and various others, such as [5, 10, 15, 21, 22]) is based upon a technical perspective which is of little use in a business domain where the primary user is a business analyst. It also assumes that the underlying business process is pre-defined, despite the fact that business needs include adaptation to technological and political changes. It is exactly for those reasons why management of web services should be from a business perspective, focussing on the agility and adaptability of services that can fulfil business requirements. In addition, there is an increasing requirement to align IT objectives with business objectives. This need has been recognised by industry and a recent report ("IBM has high hopes for 'Next Big Thing' in software", Financial Times Online, April 3, 2006) reported that IBM has doubled its business in SoA and stated that "Things really rub on each other - it's [SoA] the intersection of technology and business". Aligning these objectives cannot easily be done with low-level XML code that refers to port types and composition views!

Services are useful, in that they hide implementation details from the invoking user. Business has little interest in how a service performs its function, but rather that a service does fulfil a need and does it with a certain degree of quality. We imagine that a business may also take little interest in whether or not their software is an atomic or composite service, but rather that the resulting dynamic application does not consume more resources than what was allocated to it (e.g. financial cost and time).

Our approach is to use policies to manage services, in terms of business requirements. Policies are defined in the business domain and express what a service should do. In addition, policies define sequences of events and responses to specific activities. In [9], we presented a context in which

[1] http://www-128.ibm.com/developerworks/library/specification/ws-bpel

policies are used. This paper describes policies in more detail, including their structure and possible implementation with APPEL [20], a policy environment/language designed for the telecommunications environment. Although policies themselves can be defined in low-level languages, they are focussed more towards the business perspective and can be created through the use of wizards.

In section 2, we give a general overview of SoA, together with policies and their current uses. In section 3, we describe the adapted APPEL framework with respect to web services. In section 4, we present an initial APPEL implementation for SoA, giving examples in section 5. Finally, we conclude our research in section 6.

2. Background

SoA, and its implementation as Web Services, provides an opportunity to achieve dynamic applications through automated discovery and composition of services. Services are deployed and made available with well-defined interfaces, so that the implementation details are hidden. This is suitable for the business domain, where the required perspective is a complete picture of the external quality of the interactions, as perceived by the customers [6].

Policies are defined as "...information which can be used to modify the behaviour of a system" [14], without the need for re-compiling or re-deploying. Thus we consider policies as loosely coupled with the systems they interact with. Furthermore, we refine the definition of a policy in the context of our web service management system as *"a high level statement as to how business requirements should be processed in the management system"* (refined from [16]). Policies may be applied to a variety of components within SoA, but in particular they are currently used for access control rules and expressing reactive functionality.

The most accepted policy type to be implemented with web services is the access control rule. Ponder [7] provides a framework for specifying 3 types of policies. Authorisation policies can be either positive (allow) or negative (disallow) and state whether a subject is allowed to perform some form of activity. Obligation and refrain policies specify what actions must (forced) and must not be performed. Delegation policies specify what other subject should have certain privileges delegated to it from this policy. Parameters can be passed to each policy rule, but the definitions of activities are left to the policy implementation. These activities can only occur (or not occur depending on the policy type) when the policy is invoked and conditions are satisfied.

Similarly to Ponder, KAoS policies [4] express constraints on allowable actions performed by subjects (either agents or clients). These are represented using the Web Ontology Language[2] for use in the semantic web. Policies are applied depending on the situation, which includes details of history, state and current actions.

Rein [11] is another policy framework whose representation is ontology-based for use in the semantic web. However, rather than being based on a single ontology, every policy in Rein has its own ontology, with the ability to import other policy ontologies. Policies express constraints over resources such as services and actions.

Rein is a decentralised, distributed framework for representing and reasoning over policies in the semantic web [11]. Policies use information from other policies and web resources. Policies may be represented with different ontologies, such that each policy can potentially have its own ontology. Rein is a combination of Rei [12] and N3 [3] (a human-readable syntax for RDF). While KAoS is concerned with distributed system management and Ponder specifies management policies for distributed object systems, Rei is a framework that integrates support for policy specification, analysis and reasoning in pervasive applications. A more detailed comparison of KAoS, Ponder and Rei can be found in [19].

The Web Services Policy Framework (WS-Policy)[3] expresses capabilities and constraints of a particular web service, in conjunction with various other specifications such as WS-PolicyAttachment[4] and WS-ReliableMessaging[5]. These specifications have service-specific applications in the description and discovery layers of the web service stack. Requirements are expressed as service requirements, as opposed to business requirements.

The Web Services Policy Language (WSPL) [2] is a strict subset of the XML Access Control Markup Language (XACML)[6]. It is also an access control language based on XML and supports various authorisation and allowance policies. Again, policies of these types are applicable at the service level, managing what can be done with a service with what resources, or not.

In addition to access control situations, policies have also been used for developing automated negotiations such as in [13], where the authors highlight the potential of charging a fee for web service usage and the resulting web service marketplace. They note three market phases where the first is matches functional properties to requirements, the second is negotiating non-functional properties and the third is formalisation of agreements. Policies are used to specify formal user preferences, through a framework which provides

[2]http://www.w3.org/TR/2004/REC-owl-features-20040210/

[3]http://www-128.ibm.com/developerworks/library/specification/ws-polfram/

[4]http://www-128.ibm.com/developerworks/library/specification/ws-polatt/

[5]http://www-128.ibm.com/developerworks/library/specification/ws-rm/

[6]http://www.oasis-open.org/committees/xacml/

a generic ontology.

Thus we can see that policies are a popular approach for many aspects of web services. Though despite the number of uses, policies have yet to be applied to a business management framework. Our work is aimed at developing a business policy framework for the management of web services in the business domain.

The use of policies that we propose is orthogonal to a graphical modelling language, such as the one presented in [9] or UML activity diagrams. Each task within a task map represents a unit of business activity that contributes to the satisfaction of the wider business goal. Tasks are encoded with policies and may be subjected to external policy inputs, or global policies. The task map notation includes operators that allow tasks to be carried out sequentially and in parallel, whilst addressing synchronisation issues.

3. APPEL Policy Framework

3.1. Overview

Tonti *et al* [19] suggest that suitable policies and policy frameworks should satisfy, in addition to domain-specific requirements, the following general requirements:

- *Expressiveness* to deal with a wide range of applications;
- *Simplicity* to make it easy for authors to use;
- *Enforceability* to ensure that policies can map to implementation mechanisms;
- *Scalability* to ensure adequate performance;
- *Analyzability* to allow for reasoning about policies.

APPEL was developed in conjunction with the ACCENT project at the University of Stirling, to provide a practical and comprehensive policy language for the call control domain [18]. APPEL is defined by an XML grammar, enabling the use of many common tools and parsers. We believe that this language not only satisfies the above general requirements, but is advantageous in that since web services have many similarities to telecommunications features, we can therefore apply similar cross-domain functionality.

Due to the nature of changing requirements, particularly in the telecommunications domain with switchable features such as call forwarding, it was important to have some control mechanism over call features in such a way that core software did not need changing together with recompiling and redeployment. Web services are similar in that we often need to combine more than one together in order to achieve the results we desire. One specific advantage of SoA is service level reuse, where services are loosely coupled to their clients and can be invoked many times over. In a similar way, web services have the potential to satisfy our software requirements through the use of composition when (atomic) services are not able to individually. One should note at this point that APPEL and other policy languages are *policy description languages* (PDL), and policies are those documents borne out of implementing rules using a PDL.

3.2. APPEL Description

APPEL [18] is a generic policy description language developed for use in telecommunications through language specialisation. More specifically, it is used to specify policies that govern call control with respect to call features such as call forwarding and call barring. APPEL is defined by an XML schema, although a wizard was created for the initial implementation to increase expressive power in the end user domain and allow non-software experts a simpler method of defining their own policies.

A basic APPEL policy defines a number of policy rules, each of which specifies a set of triggers, a set of conditions and a set of actions. The first two sets may be empty, but the last cannot be, thus APPEL has the ability to express ECA rules and user goals.

The policy itself has an owner and may be applied to a user, a set of users or a domain (identified through email-like addresses). Policies can specify modalities through the preferences **must**, **should** and **prefer**, plus their negations. No specification of preference would indicate that the user was neutral about a subject.

Policies are interpreted by a policy server, implying that they have little use other than specification at the end user point. Policy servers are able to link to policy stores that contain information required for policy processing, such as protocol to policy terminology mapping.

3.3. Extension Actions

We extend the initial implementation of APPEL by introducing the following (informally described) actions:

- *Prompt*
  Many service input parameters are required from the user. Previously, the user would specify values through the orchestration script. Now, since the management system is automated and we know no information of any service, we add a function that can prompt a user for a specific data value, based on a simple description. The action `prompt(String dataName)` asks the user for a value that refers to a parameter named `dataName`. All data values are required at the beginning of a task, i.e. before an action is invoked.

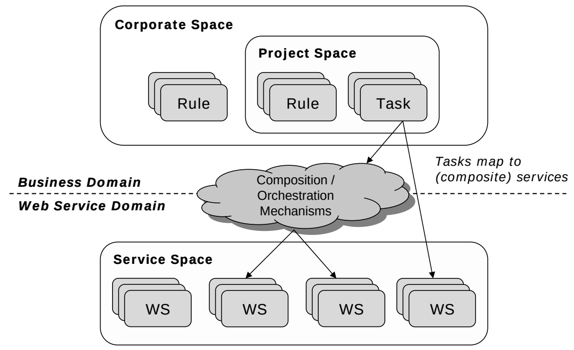

Figure 1. Cross-domain system layout

- *Display*

  Service results or responses will need to be used in some form. The most obvious would be a graphical output to the user. For example, if the user were to request train times, then they would need some method of seeing them. The action `display(Data data)` displays service response data, as either a list or a singular object. The action `display_exception(Exception exception)` outputs exception messages to the user, and the action `display_empty()` outputs a default message in the case of an empty response.

3.4. Management System Architecture

The purpose of our research can be shown in Fig. 1. Most, if not all, current uses of policies in SoA are in the service domain, rather than the business domain. In our approach, policies are used to encode the details of tasks, which themselves are located in coporate space, constrained by individual task rules and overarching corporate rules.

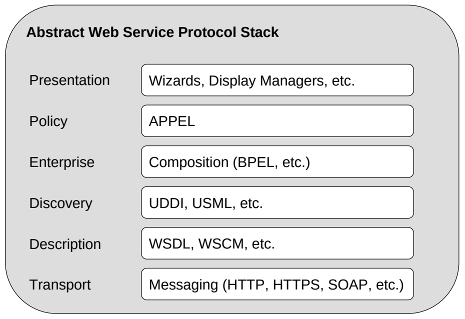

Figure 2. Enhanced web service architecture

We propose an additional two tiers to the more traditional web service architecture (shown in Fig. 2, which includes messaging, discovery, description and composition). On the top of the web service stack, we add a policy server layer and then a user interface layer. In Fig. 3 (adapted from [17]) , the bottom service layer represents the original web service stack. This addition of layers is intuitive, since we are concerned with increasing the level of abstraction towards and into the business domain.

The policy server layer contains the policy management systems, together with web service search and matching engines. There may be repositories which can store persistent policies as required by policy servers, who can also share repositories. Many policy servers may exist, depending on demand and processing ability. Other aspects, such as mediation and negotiation, can also be addressed in this layer.

The user interface layer allows business users to create new policies and activate their business requirement model. Due to the diversity of services that can exist, we must allow for a number of different interfaces, for example PDAs, PCs, mobile phones and televisions. The interface should include wizards to aid in the creation of policies as we do not expect business users to be familiar with low-level code. Wizards may be customised to their particular platform.

4. Language Specialisation for Web Services

4.1. Events

An event can be defined as either a message being passed in a system (as in event-driven programming) or a change of properties (as in telecommunications). A specified event acts as a trigger to a policy. In the context of web service management, we define simple events to be:

- *Message events*: occur when a message is sent or received. A message can be modelled as a set

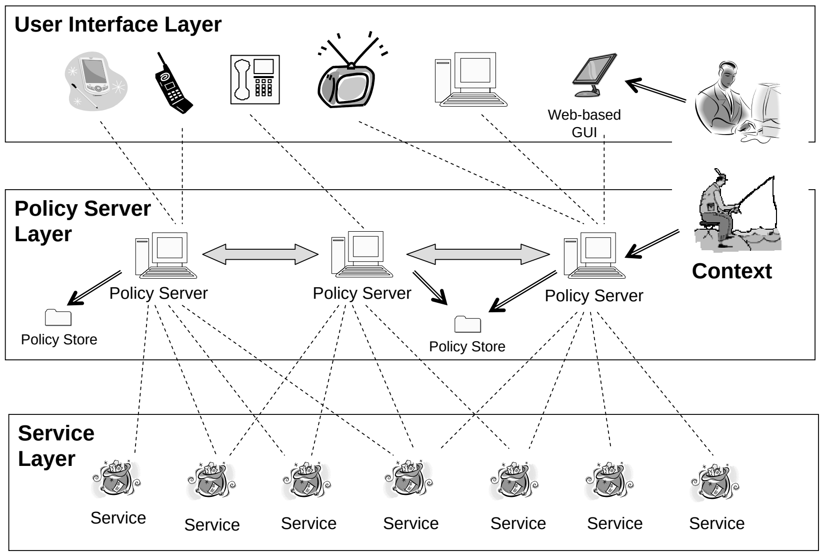

Figure 3. Policy architecture

$\{source, destination, description, data\}$, with the source and destination identified by URIs, the description being meta-data for semantic markup and the data section being the actual message passed.

- *Time events*: can be either absolute, periodic or relative time events. An absolute time event is one where a prescribed time is reached and can be specified with a standard XML timestamp object. A periodic time event represents a regular period of time, with a starting time and either duration or ending time. Duration is modelled by the set $\{years, months, days, hours, minutes, seconds\}$ to comply with XML timestamp formats. Relative time events represent an absolute time that can be calculated according to some criteria, usually a start time and duration. These are modelled by the set $\{initialTime, duration\}$, with the former element a timestamp and the latter the previously mentioned duration object.

- *Change events*: occur when properties are changed either internally or externally. A change can be to one or more property, and can reflect that a property has changed, or changed to a specific value. The set $\{source, Prop, Val\}$ describes a change, instigated by the $source$, in terms of the properties in question and the prescribed values, such that $Prop \stackrel{\text{def}}{=} \{p_1, ..., p_n\}$, $Val \stackrel{\text{def}}{=} \{v_1, ..., v_n\}$ and $\forall x \in \{1, ..., n\}$ p_x takes value v_x. The source and properties are identified by URIs.

- *Service events*: are generated before a service is invoked or afterwards (potentially before or after a response is received). These events are based on the afore-mentioned message events.

- *Interaction events*: occur depending on what type of service operation has been invoked. If the operation returns a response, a call event is generated after receiving the reply. If the operation does not return a response, then a signal event is generated on invocation.

Complex events can be built from simple events using event composition primitives (e.g. conjunction), resulting in event (or trigger) groups.

4.2. Conditions

Conditions are boolean values that must equate to `true` when the policy is triggered in order for the action section to be invoked. It is either a simple parameter value check or a condition group using condition composition primitives. Parameter values may be subject to operators such as $>, \geq$ $, <$ or $\leq$. Parameters are generally local variables that may declared in the policy, or the wider policy system.

4.3. Actions

An action is a step towards fulfilling a business goal. In the context of web services, an action may include the invocation of a service. Actions may be atomic or composite (action group) through the use of action composition primitives. These primitives include `and`, `or`, `andthen` and `else` as defined by the APPEL language.

5. Example Usage

5.1. Telecommunications Scenario

We present a simple scenario where a company director named John wishes to call his wife Mary with his mobile phone. He asks his personal assistant Fiona to set up the call although Fiona only knows Mary's email address. This scenario makes use of two services. Firstly, a service to map email addresses to telephone numbers. Secondly, a service to create phone calls between two endpoints. The simple task map is shown in Fig. 4.

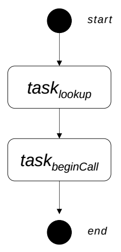

Figure 4. Telecommunications scenario task map.

The policy is structured with preamble, including preference, followed by a set of policy rules, which each consist of a set of triggers, a set of conditions and a set of actions. The attribute `owner` is the author, identified with a URI normally, but simplified in this case. In this case, Fiona is the policy author, whereas the policy itself applies to John. We attach the preference `should` to allow other policies with stronger preference to overrule it (e.g. perhaps John has gone over his personal call allowance and an automated system puts a stop to any further calls). The preamble is expressed as follows:

```
<policy name=''policyOne''
  owner=''fiona'' applies_to''john''
  id=''Telephone Recipient from Email Address''
  enabled=''true'' changed=''2006-05-15T12:25:23''>

  <preference>should</preference>

  <policy_rule>
```

The policy rule is then described as a sequence of the component triggers, conditions and actions. In our example, Fiona has specifed two triggers for the policy. Firstly, John may invoke the policy by sending a start message to the management system. The `<message>` tag indicates that a message is received and its `<data>` node indicates what the message was. Secondly, should John forget to phone his wife, the system will automatically start the process at 5pm on the same day. Should the policy be enabled after 5pm, the only trigger left would be John's start message. The triggers are expressed as follows:

```
<triggers>
  <or />
  <trigger>
    <message>
      <source>john</source>
      <data>start</data>
    </message>
  </trigger>
  <trigger>
    <parameter>localTime</parameter>
    <value>2006-05-18T17:00:00</value>
  </trigger>
</triggers>
```

Following the triggers, the set of conditions, under which actions may be executed, are described. In this case, Fiona has attached two conditions to the policy rule. Both conditions must be satisfied before the actions can commence and this is specified through the `<and />` tag. The first condition specifies that the phone call cannot be until after 3:30pm (perhaps Mary is picking up children from school). Thus the phone call must happen between 3:30pm and 5pm. Secondly, the call cannot start if John is already on the phone, expressed through the local variable `callStatus`. Conditions are expressed as follows:

```
<conditions>
  <and />
  <condition>
    <parameter>localTime</parameter>
    <operator>gt</operator>
    <value>2006:05:18T15:30:00</value>
  </condition>
  <not />
  <condition>
    <parameter>callStatus</parameter>
    <operator>eq</operator>
    <value>engaged</value>
  </condition>
</condition>
```

Finally, the actions of the policy rule are described. This section specifies two actions upon successful commencement, with the second action occurring on completion of the first, as denoted by the `<andthen />` tag.

```
<actions>
  <andthen />
  <action arg1=''mary@johnandmary.home.com''>
    <serviceType>
      <domain>Lookup</domain>
      <subDomain>Telephone Directory</subdomain>
    </serviceType>
    <functionality>
      <input name=''from''>from(arg1)</input>
      <postConditions>
        <postCond>
          <output>
            <type>integer</type>
          </output>
          </postCond>
        </postConditions>
        <output type=''integer''>
          copy_to(policyTwo:arg1)
        </output>
      </functionality>
      <qualities>
        <quality>
          <parameter>price</parameter>
          <operator>leq</operator>
          <value>0</value>
        </quality>
        <quality>
          <parameter>availability</parameter>
          <operator>eq</operator>
          <value>now</value>
        </quality>
      </qualities>
    </action>
```

The first action includes one input argument, which is Mary's email address. For added simplicity, we assume that she has one email address and one telephone number. Fiona wants to invoke a service to search for a telephone number so she specifies that the service should be in the "lookup" domain and also the "telephone directory" subdomain. The functionality is described as providing one output from providing one input, known as "from". When the output is received, it is copied to the input argument of policyTwo. In order to ensure the policy can be completed immediately, Fiona states that the service should be immediately available. Futhermore, it should cost nothing, or provide some form of credit (if perhaps this was possible!). The second action generates a system message that is passed to the second task in order to activate it:

```
      <action>
        <message>
          <source>this</source>
          <destination>this:policyTwo</destination>
          <description />
          <data>start</data>
        </message>
      </action>
    </actions>
  </policyrule>
</policy>
```

The second policy has similar preamble to the previous, and states that the only event that will trigger it is the receiving of a message from policyOne.

```
<policy name=''policyTwo''
  owner=''fiona'' applies_to''john''
  id=''Telephone Recipient from Email Address part 2''
  enabled=''true'' changed=''2006-05-15T12:25:23''>
  <preference>should</preference>
  <policy_rule>
    <trigger>
      <message>
        <source>policyOne</source>
        <data>start</data>
      </message>
    </trigger>
```

The second policy requires two input parameters: the first is received from another source (in this case policyOne) and the second is the source telephone number, as previously specified by Fiona. This way, the system knows both endpoints of the telephone call to set up. Fiona states what type of service is required, and the policy inputs are mapped to the service inputs.

```
    <action arg1=''receive()''
            arg2=''0116 222 1234''>
      <serviceType>
        <domain>Telephone</domain>
        <subDomain>Create Call</subdomain>
      </serviceType>
      <functionality>
        <inputs>
          <input name=''from''>from(arg2)</input>
          <input name=''to''>to(arg1)</input>
        </inputs>
      </functionality>
```

Further to the functionality required, some qualitative aspects are included to specify that the price of using the service should be less than 1 unit of local currency (e.g. GBP) and that the service should be available immediately.

```
      <qualities>
        <quality>
          <parameter>price</parameter>
          <operator>leq</operator>
          <value>1</value>
        </quality>
        <quality>
          <parameter>availability</parameter>
          <operator>eq</operator>
          <value>now</value>
        </quality>
      </qualities>
    </action>
  </policyrule>
</policy>
```

Thus the scenario that generates a telephone call between two endpoints, knowing only the source telephone number and the destination email address, can be achieved in two tasks, encoded by two policies.

5.2. Train Enquiry Scenario

We consider now a second scenario in which a person requires prices for a particular train journey. There is only one task involved so we do not include a task map. The policy begins with typical preamble.

```
<policy owner=''stephen@mcs.le.ac.uk''
  applies_to=''@mcs.le.ac.uk''
  id=''Query for cheapest train ticket (UK)''
  enabled=''true''
  changed=''2006-05-08T15:51:00''>
  <preference>must</preference>
  <policy_rule>
```

The policy was written by the user `smg24@mcs.le.ac.uk` and is available for anyone in the `mcs.le.ac.uk` domain. The policy is active and is triggered when a system `start` message is received. Since no source is defined, the message could come from anywhere. However, one limitation of the policy is that a user can only invoke it if a local variable that keeps track of the local country has value "UK".

```
    <trigger>
      <message>
        <data>start</data>
      </message>
    </trigger>
    <condition>
      <parameter>location</location>
      <operator>eq</operator>
      <value>UK</value>
    <condition>
```

The action section specifies a set of input arguments that must be obtained from the user through the extended `promptUser()` function. The purpose for each parameter should be intuitive from the names.

```
    <action arg1=''promptUser(Departure Station)''
            arg2=''promptUser(Arrival Station)''
            arg3=''promptUser(Date of Travel)''
            arg4=''promptUser(Fast or Cheap)''
            arg5=''promptUser(Railcard)''>
```

The policy requires the use of a service that should be listed under the domain "Travel" and the subdomain "Ticket". The service is invoked at the end of the action section, with defined functionality and quality as necesary parameters.

```
      <serviceType>
        <domain>Travel</domain>
        <subDomain>Ticket Vendor</subdomain>
      </serviceType>
```

Functionality is defined according the the inputs, which will require some form of mediation to match user given values to service input values. Mediation ensures that the user-given values are semantically-compatible with service required inputs, and furthermore applies any appropriate transformation to data values to ensure syntactic compatibility. Presuming that the inputs are correct, no preconditions or assumptions are specified.

```
      <functionality>
        <inputs>
          <input name=''from''>
            from(arg1)
          </input>
          <input name=''to''>
            to(arg2)
          </input>
          <input name=''date''>
            date(arg3)
          </input>
          <input name=''preference''>
            preference(arg4)
          </input>
          <input name=''railcard''>
            railcard(arg5)
          </input>
        </inputs>
```

The outputs should only be in two forms: firstly a list object and secondly an empty object, where the former can contain one or more sub-objects. Depending on the output type, the object is passed to a respective `display()` or `display_empty()` function.

```
        <postConditions>
          <postCond>
            <output>
              <or />
              <type>list</type>
              <type>empty</type>
            </output>
          </postCond>
        </postConditions>
        <outputs>
          <output type=''list''>
            display(this)
          </output>
          <output type=''empty''>
            display_empty()
          </output>
        </outputs>
        <exceptions>
          <exception name=''default''>
            display_exception(this)
          </exception>
        </exceptions>
```

Exceptions are addressed in a generic sense, i.e. if any exception occurs, the function `display_error()` is called. Any negative side effects (i.e. those that impose financial penalties) are disallowed while any bonus side effects (i.e. any that offer rewards) are allowed.

The side effects considered are bonuses which are allowed and penalities which are not allowed.

```
        <sideEffects>
          <sideEffect>
            <penalty>
              <type>default</type>
              <permission>disallow</permission>
            </penalty>
            <bonus>
              <type>default</type>
              <permission>allow</permission>
            </bonus>
          </sideEffect>
      </functionality>
```

In terms of quality, the policy states that the price of using the service should be free (or offer a financial bonus indicating a negative cost). Additionally, the service should be available immediately upon policy activation.

```
      <qualities>
        <quality>
          <parameter>price</parameter>
          <operator>leq</operator>
          <value>0</value>
        </quality>
        <quality>
          <parameter>availability</parameter>
          <operator>eq</operator>
          <value>now</value>
        </quality>
      </qualities>
    </action>
```

Now, upon specification of functional and non-functional requirements, the policy is complete.

6. Conclusion

Policies are already in use for a wide variety of applictions, specific to the management of distributed systems.

Futhermore, policies are often used in SoA to define access control constraints on services. We have presented an event-condition-action policy framework, derived from the APPEL policy language, to lift policies from the service domain into the business domain.

Our work is novel in that policies are used only in the service domain thus far, rather than in the business domain. The latter domain requires more of a business perspective and the input of business metrics, such as quality. It also requires agility in order to continually align corporate IT systems with key business objectives.

We have presented two scenarios in which policies have been defined and explained to the reader. We envisage that business users will not create policies using such low level code without assistance. Therefore we need to introduce wizards to simplify this process.

We have concentrated on the policy description language here, but the framework includes an architecture that allows for policies to be used in the selection of suitable services and also to govern service execution.

The gain achieved by our work is that policies are a powerful and flexible means of management in the face of dynamic environments, such as service composition. While services may change through updates, removals, etc., policies can continue to express the constraints as required by the end user. Therefore, the potentially long process of discovering and composing the "ideal" service for a requirement is significantly reduced. APPEL, although a generic policy framework, can be specialised to the service domain, as it was with telecommunications. It allows us to express a range of policies such as goals and ECAs, whilst using XML as a widely-accepted interchange format.

Our further work involves improving the language specialisation for APPEL by defining more accurately the extended functions as previously described. We also plan to add operational semantics to the language in an effort to formalise it further. Finally, in conjunction with a business modelling notation, we will look to integrate a system that combines graphical modelling with underlying policies.

Acknowledgements

This work is funded by the IST-FET IST-2005-16004 project SENSORIA (Software Engineering for Service-Oriented Overlay Computers).

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraiu. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[2] A. H. Anderson. An introduction to the web services policy language (wspl). In *POLICY*, pages 189–192. IEEE Computer Society, 2004.

[3] T. Berners-Lee. Notation 3. http://www.w3.org/DesignIssues/Notation3.html, 1998. accessed on 24-May-2006.

[4] J. M. Bradshaw, A. Uszok, R. Jeffers, N. Suri, P. J. Hayes, M. H. Burstein, A. Acquisti, B. Benyo, M. R. Breedy, M. M. Carvalho, D. J. Diller, M. Johnson, S. Kulkarni, J. Lott, M. Sierhuis, and R. van Hoof. Representation and reasoning for daml-based policy and domain services in kaos and nomads. In *AAMAS*, pages 835–842. ACM, 2003.

[5] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW*, pages 403–410, 2003.

[6] F. Casati, E. Shan, U. Dayal, and M.-C. Shan. Business-oriented management of web services. *Commun. ACM*, 46(10):55–60, 2003.

[7] N. Dulay, N. Damianou, E. Lupu, and M. Sloman. A policy language for the management of distributed agents. In M. Wooldridge, G. Weiß, and P. Ciancarini, editors, *AOSE*, volume 2222 of *Lecture Notes in Computer Science*, pages 84–100. Springer, 2001.

[8] D. Fensel, K. P. Sycara, and J. Mylopoulos, editors. *The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003, Proceedings*, volume 2870 of *Lecture Notes in Computer Science*. Springer, 2003.

[9] S. Gorton and S. Reiff-Marganiec. Towards a task-oriented, policy-driven business requirements specification for web services. In S. Dustdar, J. Fiadeiro, and A. P. Sheth, editors, *BPM*, Lecture Notes in Computer Science. Springer, 2006. To appear.

[10] IBM. Web service flow language 1.0, May 2001. accessed at http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf on 23-May-06.

[11] L. Kagal and T. Berners-Lee. Rein: Where policies meet rules in the semantic web. Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA, 2003.

[12] L. Kagal, T. W. Finin, and A. Joshi. A policy based approach to security for the semantic web. In Fensel et al. [8], pages 402–418.

[13] S. Lamparter and S. Agarwal. Specification of policies for automatic negotiations of web services. In L. Kagal, T. Finin, and J. Hendler, editors, *SWPW*, 2005.

[14] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Software Eng.*, 25(6):852–869, 1999.

[15] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *KR*, pages 482–496. Morgan Kaufmann, 2002.

[16] S. Reiff-Marganiec and K. J. Turner. Use of logic to describe enhanced communications services. In D. Peled and M. Y. Vardi, editors, *FORTE*, volume 2529 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2002.

[17] S. Reiff-Marganiec and K. J. Turner. A policy architecture for enhancing and controlling features. In D. Amyot and L. Logrippo, editors, *FIW*, pages 239–246. IOS Press, 2003.

[18] S. Reiff-Marganiec, K. J. Turner, and L. Blair. APPEL: The ACCENT policy environment/language. Technical Report CSM-164, University of Stirling, Jun 2005.

[19] G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In Fensel et al. [8], pages 419–437.

[20] K. J. Turner, S. Reiff-Marganiec, L. Blair, J. Pang, T. Gray, P. Perry, and J. Ireland. Policy support for call control. *Computer Standards and Interfaces*, August 2005.

[21] W3C. Web service choreography interface 1.0, Aug 2002. accessed at http://www.w3.org/TR/wsci/ on 23-May-06.

[22] J. Yang and M. P. Papazoglou. Web component: A substrate for web service reuse and composition. In A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. T. Özsu, editors, *CAiSE*, volume 2348 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2002.