Policies and Aspects for the Supervision of BPEL Processes

Luciano Baresi, Sam Guinea, and Pierluigi Plebani

Dipartimento di Elettronica e Informazione – Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano (Italy)
{baresi,guinea,plebani}@elet.polimi.it

Abstract. The execution of business processes with BPEL relies on external Web services, which are not necessarily managed by the process owner. This implies the need to constantly verify the correctness of the interactions between the involved parties. This paper proposes a design process model for the definition of supervised processes, in which supervision rules are automatically generated starting from the policies that characterize the external services. These policies exploit WSCoL as a language for describing constraints on the messages exchanged with the business process. In addition, we also present a new version of Dynamo: a prototype of an aspect oriented execution environment that conjugates a BPEL engine and a supervision framework.

1 Introduction

The Service Oriented Computing paradigm is driving the development of a new generation of applications. Here, Web services expose their application logic and interoperate relying on XML-based protocols, such as SOAP, and descriptions, such as WSDL. A central node usually coordinates the interactions according to a predefined process specification in which Web services are used to perform the activities. BPEL represents the de-facto standard for specifying such processes. Among other things, BPEL processes specify when a Web service should be invoked and the data that must be exchanged; they do not, however, specify how the interaction occurs: i.e., if security needs to be considered, transactions enforced, or if messaging should be conducted reliabily.

Since the execution of a BPEL process relies on external Web services, not necessarily managed by the process owner, we need to constantly verify the corectness of the interactions among the invoked services. Moreover, if something goes wrong during the process execution, suitable recovery strategies must be performed. To this end, we propose policies as the means to specify how the interaction with external Web services must occur.

BPEL provides specific compensation handlers, but the supplied features — in their current version— are limited. In particular, all compensation activities are performed using a snapshot of the process state, which precludes the modification of "live" variables. Moreover, the decision to perform a compensation is a business-guided decision which, in contrast, must be hard-coded at design time and must be suitably implemented within the BPEL process.

At this stage, a number of monitoring approaches [1, 2] have been proposed, to realize if a failure occurs during the invocation of the external Web services. Other approaches focus on the recovery problem with the goal of supporting self-healing processes [3]. However, no global solution (i) includes both monitoring and recovery, and (ii) considers design-time specification and run-time management.

The goal of this paper therefore is twofold. On the one hand, we propose a design process model for the definition of supervised BPEL processes. Supervision rules are automatically generated starting from the policies attached to the external Web services and defined by WS-Policy. Both policies and supervision rules exploit *WSCoL* (Web Service Constraint Language): a domain independent language to state monitoring assertions. We also introduce WSReL (Web Service Recovery Language) as the language to define the reaction strategy. On the other hand, we propose a new version of Dynamo (Dynamic Monitoring) [4]: an AOP-based framework for executing supervised BPEL processes, to monitor the execution and enact recovery strategies in case anomalous interactions take place. In this scenario, possible failures arise when: the service is not reachable, the service is down, or the service returns incorrect data. Analogously, possible recovery strategies could be to require a retry, a rollback, or a notification to the process manager.

To better clarify our approach, we introduce a running example. The example takes place in the field of automotive services. The business process, in fact, is intended to be executed on an automobile's onboard device. It provides users with the possibility to integrate a service for searching for parking lots with their navigation system.

When the on-board device is launched, it automatically retrieves —from the navigation system— the coordinates of the destination the user is driving towards, and the coordinates of the current position. We assume that the coordinates are given using the UTM (Universal Transverse Mercator) coordinate system [1]. The system also asks the user to define a maximum radius within which to look for parking lots. Once the system has obtained all the required data, it uses them to call an ActiveBPEL implementation of the process. We assume that the process interacts with a service similar to Microsoft's *Landmark* service to obtain the parking lots the user can choose from. Informally we can state that the Map Point service exposes the following policy: *"The service promises to provide a list of parking lots that are less than x meters from a position indicated using UTM coordinates"*. How this policy is defined service-side in *WSCoL* will be shown in Section 3, while how it is used client-side to define monitoring and recovery will be demonstrated in Section 4.

The paper is structured as follows. Sections 2 discusses the actors and activities required to design a supervised BPEL process. Sections 3 and 4 detail

[1] `http://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system`

the approach at the service and the process side, respectively. Section 5 illustrates an aspect-oriented prototype environment.Finally, Section 6 compares our approach with existing ones, and Section 7conludes the paper.

2 Design Process Model

When defining a BPEL process, *designers* typically follow a standard design model. First, they search for partner services capable of guaranteeing the functionality and QoS needed to implement their systems. This is typically done by looking in UDDI-based service registries, where designers can find service descriptions that also contain policies that regulate how the interactions with that service have to take place. Common examples are directives regarding security (i.e., authentication, encryption, etc.).

Once the designer has found all the required services, the second step is to define the business logic itself, using the constructs offered by BPEL. In our process design model, the responsibility then passes over to to the *deployer*. This actor's main goal is to provide the business process with a valid descriptor for deployment. This role is usually played by a more tech-savvy person, someone who knows how the BPEL engine will have to be configured to comply with the policies attached to the services chosen by the designer. For example, ActiveBPEL uses its deployment descriptor to specify how the engine should behave when trying to contact certain endpoints, and how the message it intends to send them must be built (e.g., a message might need to be encrypted). The last step consists in deploying the process onto the execution engine.

In this paper we present a new kind of policy assertion, which can be used to define the functional and non-functional behavioral contract the client and the provider will have to comply with. We also present a client-side management framework that can be used in conjunction with a standard BPEL engine to monitor these policies and to react when they are not satisfied.

This leads to some necessary modifications in the standard process design model we just presented (see Figure 1). The first step still consists in searching for appropriate services in a UDDI registry (Step 1). The only difference is that we assume that the service specifications are augmented with our behavioral contracts specified using *WSCoL* (Web Service Constraint Language) [4]. In practice, the language is here used to define constraints on the messages the client and the service will exchange. Similarly to what happens in classical *design by contract* [5], the service provider promises a certain behavior (specified using post-conditions), if the client complies with certain requirements (specified using pre-conditions). If the client does not comply with the pre-conditions then the service will raise an exception. In contrast, if the service does not respect the post-conditions then the client should identify the violation and react properly.

Once a designer has evaluated the behavioral specifications, and chosen the partner services to use, the next step is to design the BPEL process (Step 2). (Step 3) consists in the deployer configuring the execution environment so that it can interact correctly with the service. This is achieved semi-automatically by

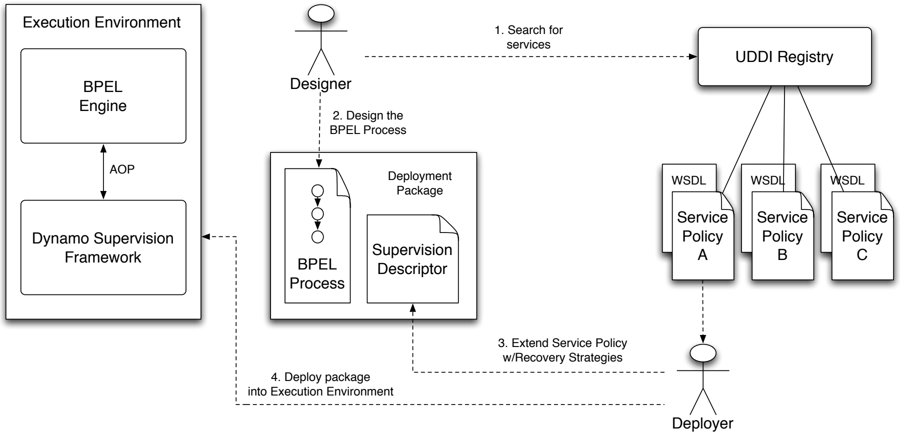


Fig. 1. Process Design Model.

feeding the execution environment with an appropriate *descriptor*, containing a declarative specification of (1) the policies the system will monitor client-side, and (2) the recovery strategies the system will try to undertake in case of invalid interactions. This can happen for two reasons: it can be the process' fault (i.e., the pre-condition is not verified), or it can be the service's fault (i.e., the post-condition is not verified). The descriptor is created by extending the service-side policy definition. We currently only allow for two kinds of extensions. In the first, the designers can modify the conditions obtained from the service-side policy by strengthening the pre-condition and/or weakening the post-condition. Such modifications are considered acceptable, since they ensure that the partner service will receive a message which is compatible with the requirements it has expressed. In the second kind of extensions, the designer can define appropriate recovery strategies (Step 4). These strategies are performed client-side in case a pre-condition is violated (in this case the actions are performed prior to the interaction with the partner service), or in case a post-condition is violated. At this stage, we are only able to consider a set of recovery strategies that are related to a well-defined set of possible failures.

This approach results in a process that clearly separates the business logic (defined in BPEL) from the supervision activities (derived from the specified policy). This provides for greater flexibility, since both the recovery strategies and the modules enacting them can be customized without affecting the business process.

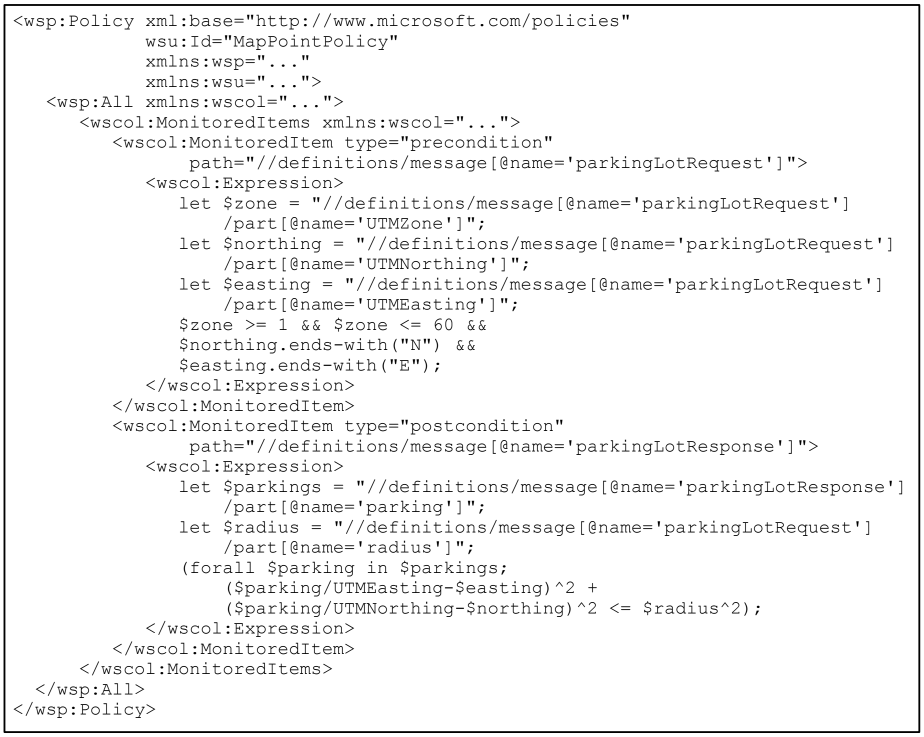
3 Service-side policies and WSCoL

According to the BPEL terminology and to the design process model introduced in the previous section, partners are Web services capable of performing one or more activities included in the process. These Web services are usually described by WSDL documents that define the operations, messages, and data-types involved during the invocation are defined. In this work, we aim at extending such a description by considering pre- and post-conditions on the incoming and outgoing messages. WSDL, indeed, cannot define, given a input parameter, which are the admissible values. In the same way, given an output parameter, the service client is not aware of the possible returning values.

For this reason, we assume that —along with the WSDL document— a WS-Policy document is provided. WS-Policy is a machine-readable language for representing the capabilities and requirements of a Web service. According to this specification both the service provider and the service user are able to argue about the behavioral aspects of a Web service. Briefly, a WS-Policy document is a composition of assertions, each of them representing an individual preference, requirement, capability or other property of the Web service. Assertions are organized according to two main operators: `ExactlyOne` and `All`. Given a set of assertions, the `ExactlyOne` operator states that only one assertion must hold at the same time, whereas, with the `All` operator, each assertion must hold.

WS-Policy is a part of the Web Services Policy Framework [6]. This framework also includes WS-PolicyAttachment [7] that specifies how a policy document can be attached to WSDL documents, UDDI entries, and generic XML files. In addition, the framework includes the guidelines for defining domain specific assertions. As listed in [8], some domain-specific assertions are now available to describe capabilities and requirements as security, reliable messaging, transactionality, and more. At this stage, no efforts have been done to describe pre- and post-conditions. For this reason, we aim at proposing such an assertions set, and to use *WSCoL* in a way that complies with the guidelines defined in [6]. These guidelines state that policy assertions representing opt-in, shared, and visible behaviors are useful pieces of metadata. In our case, pre- and post-conditions predicate on the incoming and outgoing messages of the Web services that are partners of our process (the client-side of the interaction). Pre-conditions obligate the service client to send correct data if it aims at obtaining useful results. In the same way, post-conditions make the service client aware of the range of possible values the Web service can send as a valid invocation response. Therefore, pre- and post-conditions defined in *WSCoL* affect the interaction among the parties. This means that *WSCoL* complies with the WS-Policy guidelines.

WSCoL has been previously introduced in [9] to express monitoring policies. In this work we aim at including an improved version of *WSCoL* to be used server-side. As discussed in the next section, the *WSCoL* expressions defined server side will inspire the supervision we provide with our client-side framework. *WSCoL* was originally intended for the monitoring of BPEL processes. It defines what should be monitored and how to collect the data required for such monitoring. *WSCoL* uses three different ways of collecting data: *internal variables* are part of the state of the running process, *external variables* are obtained externally by means of specific constructs for getting data from any remote com-



```
<wsp:Policy xml:base="http://www.microsoft.com/policies"
            wsu:Id="MapPointPolicy"
            xmlns:wsp="..."
            xmlns:wsu="...">
   <wsp:All xmlns:wscol="...">
      <wscol:MonitoredItems xmlns:wscol="...">
         <wscol:MonitoredItem type="precondition"
                path="//definitions/message[@name='parkingLotRequest']">
            <wscol:Expression>
               let $zone = "//definitions/message[@name='parkingLotRequest']
                   /part[@name='UTMZone']";
               let $northing = "//definitions/message[@name='parkingLotRequest']
                   /part[@name='UTMNorthing']";
               let $easting = "//definitions/message[@name='parkingLotRequest']
                   /part[@name='UTMEasting']";
               $zone >= 1 && $zone <= 60 &&
               $northing.ends-with("N") &&
               $easting.ends-with("E");
            </wscol:Expression>
         </wscol:MonitoredItem>
         <wscol:MonitoredItem type="postcondition"
                path="//definitions/message[@name='parkingLotResponse']">
            <wscol:Expression>
               let $parkings = "//definitions/message[@name='parkingLotResponse']
                   /part[@name='parking']";
               let $radius = "//definitions/message[@name='parkingLotRequest']
                   /part[@name='radius']";
               (forall $parking in $parkings;
                   ($parking/UTMEasting-$easting)^2 +
                   ($parking/UTMNorthing-$northing)^2 <= $radius^2);
            </wscol:Expression>
         </wscol:MonitoredItem>
      </wscol:MonitoredItems>
  </wsp:All>
</wsp:Policy>
```

Fig. 2. Landmark Ws-Policy example

ponent, which exposes a WSDL interface, and *historical variables* are obtained from previous process executions.

When using *WSCoL* to define service-side policies, however, a few considerations must be made. First of all, instead of predicating on BPEL internal variables, we predicate on the messages being received and sent by the service. Secondly, both external and historical data are only considered at the client-side, to effectively monitor and enforce the constraints defined at the server-side.

To check whether collected data comply with defined constraints, *WSCoL* offers the typical boolean operators, such as `&&` (and), || (or), `!` (not), => (implies), and <=> (if and only if), relational operators, such as <, >, ==, <=, and >=, and mathematical operators such as +, −, ∗, /, and %. The language also supports predicate on sets of values through the use of universal and existential quantifiers, and other constructs, such as `max`, `min`, `avg`, `sum`, and `product`.

Considering our running example, Figure 2 shows the WS-Policy document attached to the Landmark Web service [2]. As pre-condition, the policy requires a valid UTM coordinate. A UTM coordinate is composed of three main infor-

[2] In this paper, for the sake of clarity, we use a simplified WSDL

mation: *zone*, *easting*, and *northing*. Zone is a number from 1 to 60. Northing is a string consisting of six digits and an ending ‘N’ character. Finally, easting is similar to northing except it ends with an ‘E’. The post-condition guarantees that the parking lots found are no more than x meters away from the specified UTM location.

4 Client-side descriptors and WSReL

As previously stated, the client-side descriptor that instructs the Dynamo Supervision Framework is built as an extension of the service-side policies defined by the service provider. Two possible extensions are possible. In the first, the designer can take the policy defined in *WSCoL* and strengthen the pre-condition or weaken the post-condition. This guarantees run-time conformance to the original policy. In the second, the designer can add client-side recovery strategies to be performed if either the client or the service is not complying with the joint-behavioral contract. Due to lack of space we will not consider the first kind of extension but concentrate on how recovery is defined in *WSReL*.

4.1 Recovery Strategies

The recovery strategies in WSReL are based around the definition of a finite (but extensible) set of *Atomic Actions*. These actions are considered the building blocks we want to mix and match to define complex strategies. Our way of intending recovery is that these atomic actions work on a single process instance. They do not have access to the process definition. Therefore, the performed recovery is only valid for the life-span of a single process instance[3]. Moreover, the recovery strategies are performed synchronously (i.e., while the process is momentarily blocked).

The current set of *Atomic Action*s comprises: *ignore*, to simply ignore the anomaly, *notify*, to communicate to a user that something wrong happened, *halt*, to stop the process execution, *retry*, to impose that the system retry to execute the invocation a user-defined number of times, *rebind*, to indicate that the currently used Web service must be substituted with another service. At this stage we assume that the designer of the recovery strategy must indicate the endpoint of an equivalent service. For example, [10, 11] discuss approaches for QoS-based Web service selection and Web service substitution with different interfaces are discussed. Additional Atomic Actions are: *changeSupervisionRules*, to modify how supervision is achieved and therefore to relax or tighten some constraints, *changeParams*, to modify the parameters associated with the considered supervision rule, *changeProcessParams*, to modify the parameters associated with the

[3] This allows us to have different client-side monitoring and recovery specifications for different stake-holders. However, it could be interesting to investigate recovery strategies, defined by the process provider itself, that have access to the process definition.

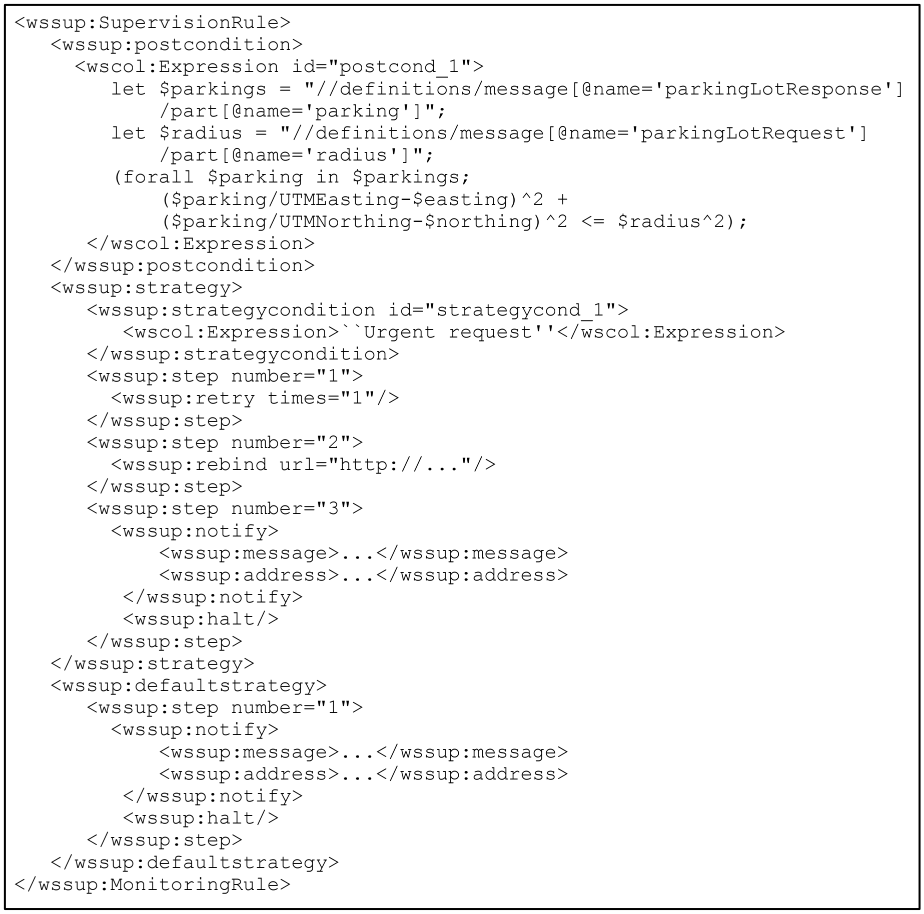
executing process, *call*, to call an external Web service, and *processCallback*, to directly invoke one of the event handlers supplied by the BPEL process.

Complex *Recovery Strategies* are not direct aggregations of atomic actions. Instead they are defined as multi-step processes, in which each step (i.e., each *Recovery Step*) attempts to fix the problem before giving up and passing on to the next. If a step is unsuccessful, it is rollbacked so that the next step can be attempted. If a step is successful, the system skips the others. A single step is defined as a conjunction of atomic actions that have to be executed. The way the system knows if a step was successful depends on the actions it contains. Some of the actions, in fact, require monitoring to be re-performed, while others are always successful (e.g., the *ignore* action). Another thing to keep in mind is that these actions have the power to modify the set of monitoring data being used, and the monitoring and recovery specifications themselves. Therefore, every time a step terminates unsuccessfully, all values are reverted to the original situation, as if no recovery had been attempted. This way, we deal with possible severe situations where the failure is not cause by the process to be monitored. On the other hand, the error might come by other software, such as operating systems, application servers, or BPEL engines.

WSReL allows more than one recovery strategy to be defined for a given erroneous situation. In fact, each strategy is always accompanied by a condition expressed in WSCoL and specified by the designer. Strategies also have an implicit prioritization, given by the order in which they are defined. The first strategy that can be applied is executed and the others are ignored. This is a simple way to avoid problems with multiple strategies enabled at the same time and to relax the constraint that *conditions* must be mutually exclusive.

4.2 Example descriptor

Figure 3 illustrates an example of a client-side monitoring and recovery descriptor. It illustrates what is defined client-side for the server-side post-condition. First of all, the supervision rule contains the same WSCoL post-condition included in the server-side policy. This defines what the client-side framework will look out for.In case the post-condition fails, the strategies included in the descriptor are considered. Specifically, each strategy has a `<strategyCondition>` expressed in WSCoL. If the related expression holds then the specified `<step>`s are successively performed, until one of them results in an effective recovery. The `number` attribute indicates the order in which the steps are performed. In the example, there are two recovery strategies. The first, which is performed when the request is considered urgent by the user, consists of three recovery steps. The framework first tries to re-invoke the service, and then tries to dynamically bind to an equivalent service (i.e., to a service with the same WSDL interface). If neither is successful, its last resort is to notify the process provider via e-mail and halt the process execution. With the `<defaultstrategy>` we define a recovery strategy to be performed when all previous conditions do not hold. In this case, this strategy consists of only one step, which is to immediately notify the problem to the process provider and halt the execution. For the sake of simplicity,



```
<wssup:SupervisionRule>
   <wssup:postcondition>
     <wscol:Expression id="postcond_1">
        let $parkings = "//definitions/message[@name='parkingLotResponse']
            /part[@name='parking']";
        let $radius = "//definitions/message[@name='parkingLotRequest']
            /part[@name='radius']";
        (forall $parking in $parkings;
            ($parking/UTMEasting-$easting)^2 +
            ($parking/UTMNorthing-$northing)^2 <= $radius^2);
      </wscol:Expression>
   </wssup:postcondition>
   <wssup:strategy>
      <wssup:strategycondition id="strategycond_1">
         <wscol:Expression>``Urgent request''</wscol:Expression>
      </wssup:strategycondition>
      <wssup:step number="1">
        <wssup:retry times="1"/>
      </wssup:step>
      <wssup:step number="2">
        <wssup:rebind url="http://..."/>
      </wssup:step>
      <wssup:step number="3">
        <wssup:notify>
            <wssup:message>...</wssup:message>
            <wssup:address>...</wssup:address>
         </wssup:notify>
         <wssup:halt/>
      </wssup:step>
   </wssup:strategy>
   <wssup:defaultstrategy>
      <wssup:step number="1">
        <wssup:notify>
            <wssup:message>...</wssup:message>
            <wssup:address>...</wssup:address>
         </wssup:notify>
         <wssup:halt/>
      </wssup:step>
   </wssup:defaultstrategy>
</wssup:MonitoringRule>
```

Fig. 3. WSReL example

the strategy conditiosn are reported informally. Corresponding WSCoL expressions predicate on user context variables, which are external variables, stating the urgency of the request.

5 Prototype

The prototype implementation we present in this section is based on AOP techniques. Its main goals are to provide BPEL process providers with the tools they need to deploy and manage processes that are aware of the kind of supervision rules presented in Section 4.

In this solution, business logic and supervision policy enforcement are defined and treated separately, since we advocate that *separation of concerns* facilitates

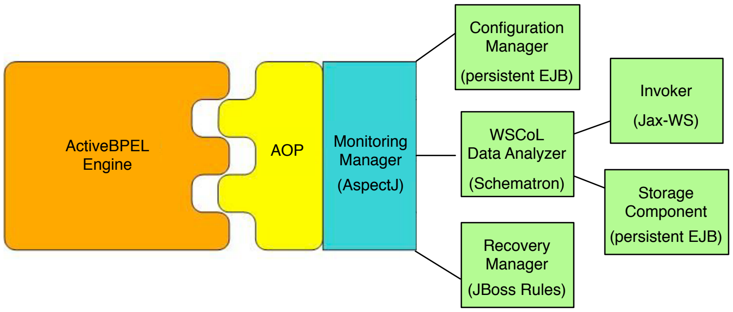


Fig. 4. The Architecture of the Dynamo Prototype.

both the design itself and later management. In this architecture, we augment —using AOP technology [12] (i.e., AspectJ [13])— a standard BPEL engine (i.e., ActiveBPEL) with notions on how to verify monitoring expressions and how to perform recovery. Business processes go unmodified and are deployed as usual, while supervision policies are deployed to a persistent component where they await activation.

Figure 4 illustrates the overall design of the prototype. It is made up of seven main components.

1) The *ActiveBPEL Engine* is the BPEL engine we have chosen for our prototype, due to the fact that it is currently the most mature open-source engine available. Its implementation revolves around the run-time visit (using the *Visitor* pattern [14]) and management of an internal tree-based representation of the process. A thorough study of the platform led us to define our pointcuts[4] as (1) after the engine visits a *Receive* node, (2) before and after it visits an *Invoke* node, and (3) before and after it visits a *Pick* node. These were chosen since they represent the points in which the process interacts with the outside world.

2) The *Monitoring Manager* represents the main *advice*, that is to say the component that is weaved into the execution environment. The result is that —after the weaving— this component has direct access to the internal representation of the process in execution, and to its state (i.e., the set of instantiated BPEL variables). This allows it to collect data from the process itself, and provide them for analysis. This component is also responsible for managing all the steps in the monitoring process. We will defer a more in depth analysis of its behavior to Section 5.1.

3) The *Configuration Manager* is a persistent component in which we store all the supervision descriptions that have been devised, and that are waiting to be activated. The *Monitoring Manager* can query its contents by specifying

[4] This term indicates —in the standard AOP terminology— the points in which we are interested in inserting our cross-cutting concern. In our case, it indicates the points in which we want to activate supervision.

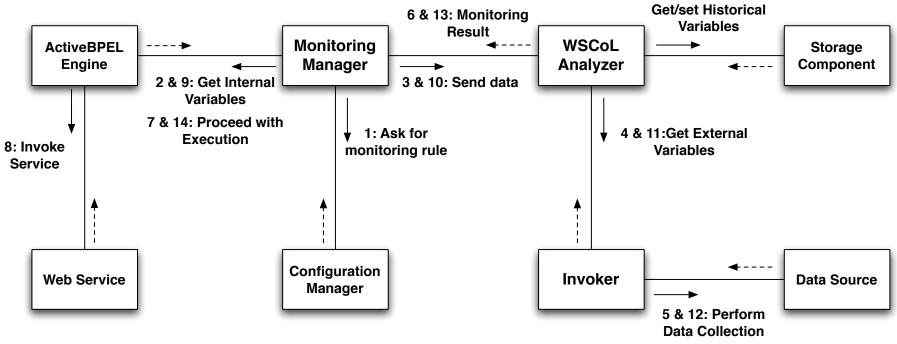

ActiveBPEL Engine
Monitoring Manager
WSCoL Analyzer
Storage Component
Web Service
Configuration Manager
Invoker
Data Source
6 & 13: Monitoring Result
Get/set Historical Variables
2 & 9: Get Internal Variables
3 & 10: Send data
7 & 14: Proceed with Execution
8: Invoke Service
1: Ask for monitoring rule
4 & 11:Get External Variables
5 & 12: Perform Data Collection

Fig. 5. Monitoring pre- and post-conditions.

the process it is executing, the unique id of the user of the business process, and the *Receive, Invoke*, or *Pick* activity being executed. As the reader can see, these allow the system to distinguish between different supervision policies for different users, and to guarantee personalized supervision. 4) The *WSCoL Data Analyzer* is the component responsible for actually verifying the monitoring expressions. The component takes the data collected from within the process, and the monitoring rules extracted from the *Configuration Manager*, and provides a monitoring result. If it needs extra data to perform its analysis (e.g., external or historical WSCoL variables), it can interact directly with the *Invoker* component to obtain data from external data sources, or with the *Storage Component* to obtain data pertaining to previous supervision activities.

5) The *Recovery Manager* is responsible for the execution of recovery strategies when monitoring has signaled an anomaly. It is based on the ECA rules paradigm [15] (i.e., *event-condition-action*), and was built using JBoss Rules [16] (formerly known as Drools). The *event* is implicit and consists in the anomaly itself being signaled. The *condition* consists of a two-level nesting of *if-then-else* clauses that allow the system to distinguish between different actions depending on the extent to which a monitoring expression is unverified. Both the clauses are expressed in WSCoL. The former reply the pre- or post-condition included in the service-side policy. The latter allows us to distinguish aong different reaction strategies. Finally, the *action* is a recovery strategy, as defined in Section 4.

6 and 7) The *Invoker* and the *Storage Component* are utility modules. The former allows to dynamically bind and invoke any Web service. The latter is used to store data collected during previous activations of the supervision framework.

5.1 Monitoring Manager

Figure 5 illustrates how all the aforementioned components come together to provide supervision. When the process execution is intercepted and the *Monitoring Manager* is activated, the first thing it does is to obtain the *processID*, the *userID*, and the *invokeID* needed to query the *Configuration Manager*. All these data are automatically provided by the execution engine, except for the *userID* which is provided by the user in the SOAP message that instantiates the process[5]. In our current implementation, the userID must be provided by the user when a new instance of the process is requested. However, in the future, the ID could be provided automatically by the system through authentication.

The *Configuration Manager* is then queried for supervision rules that have been defined by that user, for that process, and in particular for that activity (Step 1). If none are found the execution is immediately returned to the engine.

However, if a rule exists, and it defines a pre-condition, the *Monitoring Manager* reads the monitoring expression to see what WSCoL internal variables have been defined, and need to be collected (Step 2). This is simplified by the fact that, thanks to the AOP weaving, the *Monitoring Manager* lives in the same execution space where the ActivbeBPEL variables are stored. Once all the data collection has been achieved, the data is formatted into XML and sent to the *WSCoL Analyzer* together with the monitoring expression itself (Step 3). The *WSCoL Analyzer* proceeds to finish data collection (external and historical variables) and perform verification. Once it has finished, the monitoring result is returned to the *Monitoring Manager* (Steps 4, 5, 6). At this point, if no error has been discovered the framework returns control to the execution engine which performs the service invocation (Steps 7, 8). When the supervision framework is re-activated, after completing the execution of the *Invoke* activity, it checks whether there is a post-condition. Its data collection and analysis are performed in the same way as for the pre-condition (Steps 9, 10, 11, 12, 13).

5.2 Recovery Manager

The current version of the recovery manager has been implemented using the JBossRules ECA rule engine. We use data collected during monitoring, and the monitoring results themselves, to produce JBoss rules that will “fire” according to the WSCoL strategy conditions in WSReL.

In order to guarantee the correct activation order for the recovery strategies, defined explicitly by the WSCoL conditions associated with the single strategies and implicitly by prioritization, we make use of: (1) the concept of *salience*, an integer value that gives a rule a certain priority (higher priority rules are executed before lower priority ones, while rules with the same priority are executed in a LIFO manner), (2) the concept of *activation-group*, a value which groups rules into sets in which only one rule can be activated, and (3) the concept of *agenda-group*, which allows the engine to discriminate between rule sets, and to execute only those actions that belong to the agenda-group that is said to be in “focus” (this can be set programmatically).

In our example we have two recovery strategies (see Figure 6). They are defined with the same agenda-group, meaning that the rule engine will try to

[5] This is the only modification that needs to be performed on the process definition to enable supervision

```
rule "Strategy_1"
   salience 2;
   agenda-group="postcond_1"
   activation-group="ag1"
   when
       strategycond_1
   then
       recov_strategy_1();
end
```

```
rule "Strategy_2"
   salience 1;
   agenda-group="postcond_1"
   activation-group="ag1"
   when
       strategycond_2
   then
       recov_strategy_2();
end
```

Fig. 6. JBoss rules example.

activate them at the same time. However, they have two different salience values, meaning that `strategy_1` will be considered first. An activation-group is made explicit since the two strategies are mutually exclusive. If the first fails to "fix" the problem, then the second is activated.

Finally, the `recovery manager` performs the single recovery strategies by invoking a Java application (`recov_strategy_1()` and `recov_strategy_2()`) which contains the strategy steps and their atomic actions. In addition, the `Monitoring Manager` is also responsible for re-evaluating monitoring to see if recovery was successful. Monitoring being re-evaluated also translates into a complete cleansing of the JBoss Rules working space.

6 Related work

Much work has been accomplished in the field of the specification and monitoring of service level agreements for Web services. Keller and Ludwig [17] advocate the need for a framework that can provide tools for the specification, measurement, and monitoring of QoS parameters. Ludwig et al. also present a revisitation of their work in [1], in which they adopt WS-Agreement [18] as their agreement language. They propose Cremona (Creation and Monitoring of Agreements) as an architecture that can facilitate the design and management of agreements through the use of templates. The architecture is mainly composed of two parts: an Agreement Protocol Role Management component, intended to help create and access agreements at run-time, and an agreement Service Role Management component, required to trigger agreement-driven provisioning of a service and to monitor their compliance. With respect to Cremona, which concentrates on QoS, our approach can be used to define more general properties. This guarantees a more widespread solution which can be adapted to many different needs. Spanoudakis and Mahbub [2] have also developed a framework for monitoring requirements of BPEL-based service compositions. Their approach uses event-calculus for specifying the requirements that must be monitored. Requirements can be behavioral properties of the coordination process or assumptions about the atomic or joint behavior of the deployed services. The system observes system events during execution, and stores them in a database. Run-time check-

ing is then interpreted as integrity constraint checking in a temporal deductive databases. Like our approach, they also provide reactive monitoring since erroneous situations can be found only after they have occurred. It is a less intrusive approach that proceeds in parallel to the execution of the business process. This leads to a lesser impact on performance but also to a lesser responsiveness in discovering run-time erroneous situations.
In our work, pre- and post-conditions are expressed using WSCoL since it provides compatibility with the rest of the proposed solution. Nevertheless, the policies can include conditions expressed according to different languages such as OCL (Object Constraint Language) or specific logics (e.g. temporal or descriptive). Even if some work, such as OWL-S [19] and WSDL-S [20], include pre- and post-conditions directly into the functional specification, we prefer to exploit WS-Policy. This way, we separate the technical details of the invocation from the constraints on exchanged data.

7 Conclusions and future work

In this paper we have presented an approach to supervise BPEL processes by exploiting policies and aspects. Policies are involved at design-time, when the process owner selects the external Web services to be invoked during the process execution. WS-Policy has been adopted as the language for expressing the behavioral aspects of the external Web services in term of pre- and post-conditions. In particular, WSCoL inspires a new set of assertions compliant with the WS-Policy framework. Policies also drive the process deployer during the configuration of the BPEL process. Using a WSReL descriptor the process is instructed to check the pre- and post-conditions during the service invocation and to properly react in case of violation. The descriptor is semi-automatically generated by starting from the policies attached to the external services. Finally, we present an AOP-based prototype, which is responsible for the execution of the business logic, and for its monitoring and recovery.

References

1. Ludwig, H., Dan, A., Kearney, R.: Cremona: an architecture and library for creation and monitoring of ws-agreements. In: Proceedings of the 2nd International Conference on Service Oriented Computing, ACM (2004) 65–74
2. Mahbub, K., Spanoudakis, G.: A framework for requirents monitoring of service based systems. In: Proceedings of the 2nd International Conference on Service Oriented Computing, ACM (2004) 84–93
3. Modafferi, S., Mussi, E., Pernici, B.: SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In: 1st workshop on Middleware for Service Oriented Computing (MW4SOC '06), Melbourne, Australia (2006) 48–53
4. Baresi, L., Guinea, S.: Towards dynamic monitoring of ws-bpel processes. In: Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings. Volume LNCS 3826. (2005) 269–282

5. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10) (1992) 40–51
6. Vedamuthu, A., Orchard, D., Hondo, M., Boubez, T., Yendluri, P.: Web Services Policy 1.5 - Primer. `http://www.w3.org/TR/2006/WD-ws-policy-primer-20061018` (2006)
7. C. Sharp (ed.): Web Services Policy 1.2 - Attachment (WS-PolicyAttachment). `http://www.w3.org/Submission/WS-PolicyAttachment/` (2006)
8. VV.AA.: Web Service Policy Framework. `http://www-128.ibm.com/developerworks/library/specification/ws-polfram/` (2006)
9. Baresi, L., Guinea, S., Plebani, P.: WS-Policy for Service Monitoring. In: Technologies of e-Services (TES). LNCS 3811 (2005) 72–83
10. Antonellis, V.D., Melchiori, M., Santis, L.D., Mecella, M., Mussi, E., Pernici, B., Plebani, P.: A layered architecture for flexible web service invocation. Softw., Pract. Exper. **36**(2) (2006) 191–223
11. Fugini, M., Plebani, P., Ramoni, F.: A user driven policy selection model. In Dan, A., Lamersdorf, W., eds.: Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings. LNCS, Chicago, Springer (2006) 427–433
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP. (1997) 220–242
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP. Volume 2072 of Lecture Notes in Computer Science., Springer (2001) 327–353
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1995)
15. McCarthy, D., Dayal, U.: The architecture of an active database management system. Proceedings of the 1989 ACM SIGMOD international conference on Management of data (1989) 215–224
16. Proctor, M., Neale, M., Lin, P., Frandsen, M.: Drools documentation. Technical report, JBoss.org (2006)
17. Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Journal of Network and Systems Management **11**(1) (2003) 57–81
18. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web Services Agreement Specification (WS-Agreement). Global Grid Forum GRAAP-WG, Draft, August (2004)
19. Martin (ed.), D.: OWL-S: Semantic Markup for Web Services. W3C Submission. `http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/` (2004)
20. Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.T., Shet, A., Verma, K.: Semantic Annotations for WSDL. `http://www.w3.org/Submission/WSDL-S/` (2005)