

WS-Policy for Service Monitoring

L. Baresi, S. Guinea, and P. Plebani

Dipartimento di Elettronica ed Informazione,
Politecnico di Milano, Piazza L. da Vinci, 32 - 20133 Milano, Italy
{bares, guinea, plebani}@elet.polimi.it

Abstract. The paper presents a monitoring framework for WS-BPEL processes. It proposes WS-CoL (Web Service Constraint Language) as a domain-independent language, compliant with the WS-Policy framework, for specifying user requirements (constraints) on the execution of Web service compositions. WS-Policy and WS-CoL provide a uniform framework to accommodate both functional and non-functional constraints, even though the paper only addresses non-functional requirements. It concentrates on security, which is one of the most challenging QoS dimensions for this class of applications.

1 Introduction

Originally, *service-centric* computing relied on the simple and essential service-oriented paradigm, where service providers, service users, and service directories were the only players. Recently, many proposals have tried to extend the service-oriented approach with issues related to composition, conversation, monitoring, and management [1]. In particular, this paper focuses on extending the basic features with the capability of monitoring the execution of composed Web services (i.e., WS-BPEL processes), as a way to assess both their functional correctness and quality of service. Monitoring should address both functional and non-functional aspects and might involve different parties: clients may be interested in probing the services they use, providers may assess the services they offer, but also third party entities might be involved to offer neutral monitoring capabilities and collect historical data.

The paper introduces a monitoring approach capable of probing both functional and non-functional requirements. Functional requirements predicate on the correctness of the information exchanged between the WS-BPEL orchestrator and the selected services; non-functional requirements are about aspects directly related to how well the service works in terms of, for example, security, transactionality, performance, and reliable messaging. In order to probe such a wide range of requirements, the execution must be analyzed: (1) before invoking the service, that is, before the message to invoke it exists, (2) after producing the message, but before reaching the target service, (3) before the return message reaches its destination, and (4) after reaching it. The first two cases cover the flow from the WS-BPEL orchestrator to the target service, while the other two cases deal with the opposite flow.

The approach presented in this paper concentrates on client-side monitoring and relies on WS-Policy [2], the emerging standard to define Web service requirements, to express the *monitoring policies* associated with WS-BPEL processes, that is, the user

requirements (constraints) on running Web services compositions. All constraints are written in WS-CoL (Web Service Constraint Language), a domain-independent language for monitoring assertions.

The paper also describes a prototype component, called *Monitoring Manager*, that can be used to extend existing platforms for service offering and invocation¹ with monitoring capabilities.

Even though the approach is general, the paper only addresses non-functional aspects, and specifically it concentrates on security, one of the most challenging QoS dimensions for deploying Web services systems. The approach is exemplified on a simple case taken from the common scenario of online book shopping. BookShop is an online bookshop that uses a WS-BPEL process to coordinate all the steps that must be taken to interact with its clients. Here, we concentrate on the service invocation the process makes to OnlineBank to register credit-card transactions. We require that this invocation be encoded using the 3DES algorithm and be pursued only if the total amount to be charged is less than the amount defined in the user's preferences. In fact, BookShop maintains a repository of user preferences to simplify the process of buying books and registers the client's credit-card and a money cap. A money cap is useful when a client wants to avoid spending more than a certain amount of money in a single transaction.

The paper is organized as follows. Section 2 briefly discusses the WS-Policy framework and how related specifications can be used along the Web service life-cycle. Section 3 introduces the monitoring approach adopted to check the proposed policies for monitoring. Section 4 introduces WS-CoL, our assertion language adopted to express non functional requirements. Section 5 presents the architecture of the monitoring framework and exemplifies how it works. Section 6 briefly surveys related approaches and Section 7 concludes the paper.

2 WS-Policy and Policy Lifecycle

WS-Policy [2] is emerging as the standard way to describe the properties that characterize a Web service. By means of this specification, the functional description of a service can be tied to a set of assertions that describe how the Web service should work in terms of aspects like security, transactionality, and reliable messaging. According to [3], an assertion is defined as “an individual preference, requirement, capability or other property”, and the WS-Policy document is in charge of composing such assertions to identify how a Web service should work. These assertions can be used to express both functional aspects (e.g., constraints on exchanged data), and non-functional aspects (e.g., security, transactionality, and message reliability). So far, a couple of languages, namely WS-SecurityPolicy and WS-ReliableMessaging Policy, have been proposed as a set of WS-Policy-compliant domain dependent assertions. Similarly, as discussed in Section 4, we propose WS-CoL (Web Service Constraint Language), as a domain-independent language to express monitoring constraints.

As stated in [4], policies can be defined by several actors and during different phases of the Web service life-cycle (Figure 1). Besides implementing the application, service developers also specify the properties that must hold during the execution regardless

¹ For example, existing service buses.

of the platform on which the services will be deployed (*service policies*). On the other hand, service providers specify the features supported by the application servers on which services are deployed. (*server policies*). The intersection of service and server policies results in *supported policies*, which define the properties of the services deployed on a specific platform. Finally, Web service users state the features that should be supported by the services they want to invoke (*requested policies*). By combining requested policies and supported policies, we obtain the so called *effective policies*. Approaches to policy intersection are discussed in [4, 5, 6].

In this paper, we do not concentrate on policy intersection, but on the result produced by policy intersection, that is, the effective policies. Effective policies represent the set of assertions that specify the properties of a Web service deployed on a particular server and invoked by a specific user. The Web service to which effective policies apply is linked by definition and it can be a simple Web service or a WS-BPEL process. Once effective policies are derived, services should be monitored at runtime to guarantee that they offer the service levels stated by their associated policies. So, in this paper we propose a framework capable of monitoring effective policies expressed using WS-Policy.

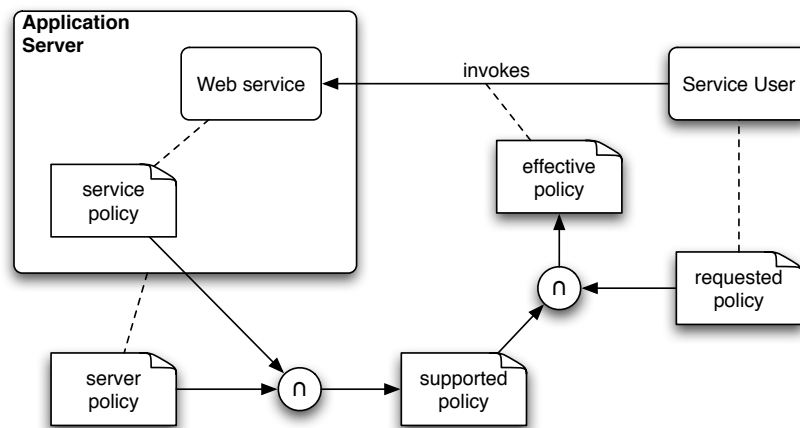


Fig. 1. Ws-Policy definitions and attachments

WS-PolicyAttachment [7], one of the elements of the WS-Policy framework, supports the policy life-cycle described above by defining how a WS-Policy document can be tied to an XML document that represents the subject for which the policy holds. Notice that the assertions included in the effective policy can be applied at different levels of granularity: process level, branch of execution level, service invocation level, message level, or internal variable level. Hereafter, for simplicity, we suppose that all the effective policy assertions work at the same level and, more precisely, at the service invocation level. If the considered service is a WS-BPEL process, policies can be attached to some of the service invocation activities.

3 Monitoring Approach

Runtime monitors [8] are the “standard” solution to assess the quality of running applications where suitable probes control the functional correctness and the satisfaction of QoS parameters. Our monitoring approach borrows its grounding from assertion languages, like Anna (Annotated Ada [9]) and JML (Java Modeling Language [10]), and is also based on the idea that we want to reuse as much existing technology as possible as a means to increase its diffusion and acceptability².

The tradeoff between monitoring and performance might be influenced by many different factors. We cannot define a strict relationship between WS-BPEL processes and monitoring directives. Users must be free to change them to cope with new and different needs. For example, the execution of these processes in different contexts might require a heavier burden in terms of monitoring, while when selected services are well-known and reliable, users might decide to privilege performance and adopt a looser monitoring framework.

These considerations led us to propose monitoring directives as stand-alone (external) *monitoring policies* rendered in WS-Policy (see Section 2). These constraints do not belong to the workflow description, that is, the WS-BPEL process, but they are weaved with it at deployment-time. Besides the gain in flexibility, with different sets of monitoring policies that can be associated with the same process, this solution also allows us to keep a good separation between business and control logics.

The weaving process is governed by BPEL², which instruments the original WS-BPEL specification to make it apply the monitoring policies. The pre-processor parses all the monitoring policies selected for the particular process. For each policy, the embedded location indicates the point of the process in which BPEL² substitutes the WS-BPEL invoke activity with a call to the monitor manager, which is then in charge of evaluating the policy and call the service if it is the case. BPEL² also adds an initial call to the monitoring manager, to send the initial configuration (such as the priority at which the process is being run) to initialize it, and a final call to communicate it has finished executing the business logic and that resources can be released.

BPEL² produces a fully-compliant WS-BPEL specification, which is deployed instead of the original one. Monitoring policies are not actually intertwined with the original process. BPEL² only adds calls to the monitoring manager. This means that policies can change without re-instrumenting the process.

After the weaving process at deployment-time, monitoring policies can be switched on and off at runtime [11]. Special-purpose parameters, like *priority*, allow the designer to select those policies that are to be checked at run-time (they must be a subset of those selected at deployment time). Notice that the priority associated with monitoring policies must not be confused with the `preference` defined in the WS-Policy framework. The preference defines the internal order among policies, while the priority is used to define if a policy must be monitored. For example, if a policy has priority lower than the current one (i.e., the one set by the monitoring manager), the manager skips its execution and calls the actual service directly. The monitoring manager, the component

² The current implementation of the approach as “external” component can be seen as a feasibility study before embedding this technology in a standard WS-BPEL engine.

that oversees the application of the monitoring policies, has a dedicated user interface that lets the designer change its current priority and thus dynamically modify the impact that monitoring has on the execution.

4 Web Service Constraint Language

The *Web Service Constraint Language*, hereafter WS-CoL, is a domain-independent policy assertion language for specifying user requirements (constraints) on the execution of Web services. WS-CoL is a standard assertion language augmented with special-purpose features to retrieve “external” data. It distinguishes between *data collection* and *data analysis* to differentiate the phase in which information is collected (from external sources, if needed) from the phase in which stated expressions are evaluated against collected values. Data can be collected from the process directly (e.g., internal variable), but they can also come from external sources (e.g., exchanged SOAP messages, metering tools).

Internal variables are accessed by means of the following instruction:

```
$<name_of_variable>\<part_of_variable>
```

As in the WS-BPEL specification, a variable is an instance of an XML schema. Since a variable can be composed of several parts, this instruction allows us to access the different parts.

If data come from external sources, called data collectors, we use the following instruction:

```
\return[Int|String|Boolean](WSDL, OpName, <parameters>)
```

It defines how to retrieve the information that originates outside the process. We suppose that data collectors are Web services, therefore the instruction’s parameters have the following meanings:

- WSDL represents the URL of the WSDL related to the requested data collector. For example, in Figure 2, WSDL_XPATH indicates a data collector capable of extracting data from XML snippets according to an XPath expression.
- OpName represents the operation supported by the data collector. In the example of Figure 2, applyXPath is an operation that returns the value corresponding to an XPath expression.
- <parameters> represents the set of values requested by the operation. In the example of Figure 2, the applyXPath operation requires two arguments: an XPath expression and the file in which the information is stored (we suppose that up.xml contains the user preferences).

So far, we support data collectors returning an integer, string, or boolean. These instructions can be nested to filter (or compose) the data gathered from different sources.

Data analysis can be carried out by different data analyzers. The WS-CoL concrete syntax can be translated into different abstract representations that correspond to different analysis engines. In this paper, we concentrate on a specific engine implemented using `xlinkit` [12] and `CLIX` [13]. WS-CoL complies with the WS-Policy framework, and assertions based on WS-CoL can be included in a WS-Policy file.

1. Policy attachment:

```

<wsp:PolicyAttachment xmlns:wsp="...">
  <wsp:AppliesTo xmlns:wsp="...">
    <wscol:MonitoredItems xmlns:wscol="...">
      <wscol:MonitoredItem type="precondition"
        path='XPATH expression to WS-BPEL invoked activity' />
    </wscol:MonitoredItems>
  </wsp:AppliesTo>
  <wsp:PolicyReference
    URI="http://www.bookshop.it/policies#BookShopPolicy"/>
</wsp:PolicyAttachment>

```

2. Policy definition:

```

<wsp:Policy xml:base="http://www.bookshop.it/policies"
  wsu:Id="BookShopPolicy"
  xmlns:wsp="..."
  xmlns:wsu="...">
  <wsp:All xmlns:wsp="..."
    xmlns:wscol="...">
    <wsse:Confidentiality>
      <wsse:Algorithm type="wsse:AlgSignature"
        URI="http://www.w3.org/2000/09/xmlenc#3des-cbc" />
    </wsse:Confidentiality>
    <wscol:Expression>
      ($ChargeRequest\amount) <=
        \returnInt(WSDL_XPATH, applyXPATH,
          '\userpref\moneyCap', up.xml)
    </wscol:Expression>
  </wsp:All>
</wsp:Policy>

```

Fig. 2. Ws-Policy example

Figure 2 shows a possible effective policy attachment³, where policy `BookShopPolicy` is applied to all the subjects identified by the XPath expression in the `MonitoredItem` tag. The type attribute specifies when the expressions included in the policy must hold. More in details, the effective policy, which must be satisfied when the credit card is about to be charged, is defined in the second part of the Figure: the `BookShopPolicy` states both functional and non-functional properties. In the example, we use an assertion that complies with `Ws-SecurityPolicy` to specify that all exchanged messages be encrypted using “3DES” as the encryption algorithm. Moreover, functional requirements impose that every time clients are ready to pay for their books, the order cannot exceed the money cap. This last constraint is rendered in the `WS-CoL` assertion included in the `Expression` tag: the amount of money of the current purchase (`ChargeRequest`) must be less than or equal to the `moneyCap` of the current user’s preferences (`uP`).

³ Namespaces are not included for the sake of readability.

Notice that the expression `$ChangeRequest\amount` retrieves the cost of the purchase from the corresponding WS-BPEL internal variable, while `\returnInt(WSDL_XPATH, applyXPath, '\\userpref\moneyCap', up.xml)` retrieves the maximum amount the user is willing to spend, from the preferences file named `up.xml`.

5 Monitoring Manager

The proposed monitoring component, called *Monitoring Manager*, is simple and extensible in terms of the data analyzers it can use for verifying functional and non-functional properties at run-time. Simplicity has been chosen over other guidelines, such as performance, due to its prototypical nature. The *Monitoring Manager* is composed of four principal components (see Figure 3): the *Rules Manager*, the *Configuration Manager*, the *External Monitors Manager* and the *Invoker*.

The UML collaboration diagram of Figure 4 shows how such components interact during the execution of a WS-BPEL process if the monitoring of pre-conditions is required. When BPEL² produces the instrumented version of the process, it adds an initial call to the manager (1) that sets up the monitoring activities by creating a specific configuration in the Configuration Manager (2). This configuration contains all the policies that are selected for the process.

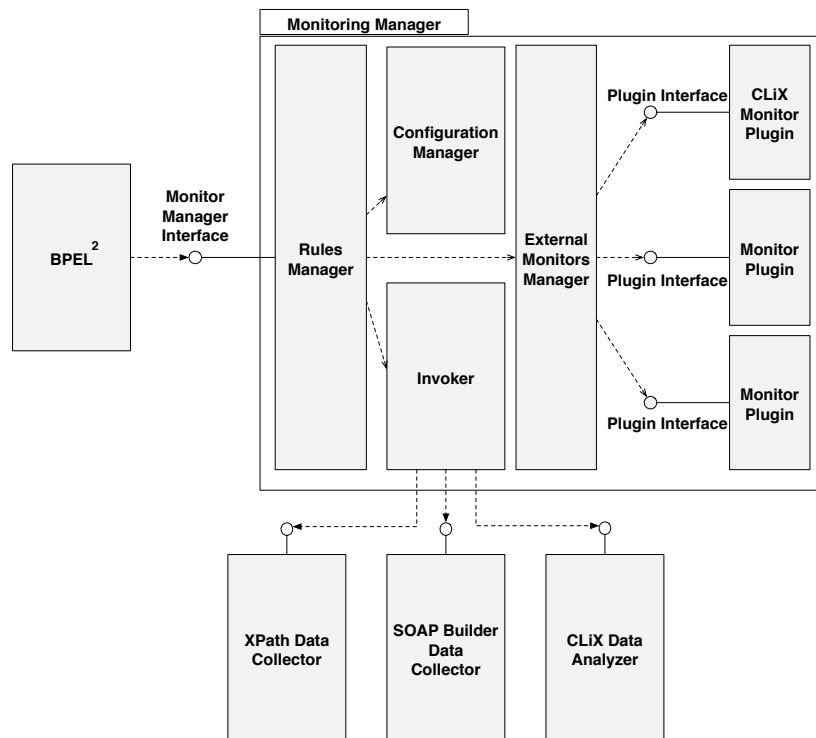


Fig. 3. Interaction with the Monitoring Manager

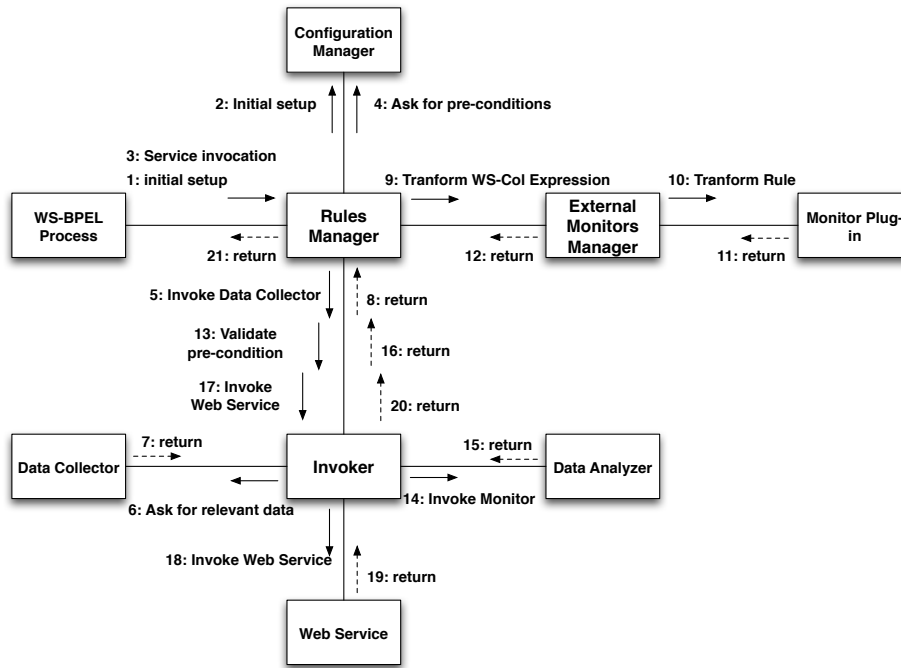


Fig. 4. Interactions among the main elements of the monitoring manager

After setup, the execution of the actual business logic commences. If the instrumented process needs to invoke a service that must be monitored, it invokes the Monitoring Manager in its place (3). The manager is sent the data that are to be analyzed and the information required to invoke the Web service that the manager is wrapping. The Rules Manager extracts the expressions associated with the service invocation from the Configuration Manager. In Figure 2, an encryption policy and a functional pre-condition are associated with the OnlineBank service invocation. This means that when the monitoring of these properties is requested by the instrumented process, their appropriate expressions are extracted from the Configuration Manager (4).

The pre-condition is a functional property that must be verified prior to constructing the SOAP message that must be sent to the OnlineBank service. The encryption policy is a non-functional property that must be verified after the SOAP message has been constructed and prior to sending it to the OnlineBank service. If we consider return messages, the approach works similarly.

When a policy has to be checked, the Rules Manager starts by confronting the policy’s property with the global process execution priority. This is done to decide whether the policy should be monitored or if the requested monitoring activity can be ignored. In our example, we can imagine the process execution priority is “4” while the monitoring rule’s priority is “5”. This means that the required monitoring activity cannot be ignored.

If a policy is to be monitored, the Rules Manager analyzes the expressions to see if additional data must be obtained prior to effective analysis (5,8). If additional data

is needed (meaning a `\returnString`, `\returnInt`, etc. is present in the WS-CoL expression), the Invoker is called to interact with the specified external data collectors (6,7). Once all the data has been obtained, the Rules Manager asks the appropriate external monitor plugin to translate the WS-CoL expression and the data into the formats the external monitor (in this case the CLiX monitor) is capable of interpreting (9,10,11,12). Once this translation is completed, the appropriate data analyzer is invoked (13,14,15,16) and the Rules Manager waits for a response. If the response is that the property is valid, (this is the case in Figure 4) the Rules Manager proceeds by asking the Invoker component to call the Web Service that would have been called originally (17,18,19,20,21). If the data analyzer responds by saying that the property is not valid, a standard exception is raised to the instrumented process which can then decide for some recovery strategies⁴.

At this point the manager proceeds to the monitoring of the SOAP message that is to be sent to the OnlineBank service, to see if it is encrypted as stated in the encryption policy (see the policy example in Figure 2), in other words using “3DES”. Notice that even for this policy the monitoring approach is the same as described before. In fact, when the instrumented version of the WS-BPEL process is created, the encryption policy is translated into WS-CoL, to make it interpretable by the manager. This means that, potentially, a generic WS-Policy assertion can be translated into a WS-CoL assertion in order to express how the assertion can be monitored.

In particular, the policy is translated into two different WS-CoL expressions, one for the outgoing message and one for the returning message. Both are sent to the manager during the initial setup phase and stored in the Configuration Manager. If we consider the WS-SecurityPolicy assertion included in our example:

```
<wsse:Confidentiality>
  <wsse:Algorithm type="wsse:AlgSignature"
    URI="http://www.w3.org/2000/09/xmlenc#3des-cbc"/>
</wsse:Confidentiality>
```

The corresponding WS-CoL expression for the outgoing message is:

```
<wscol:Expression>
  \returnString(WSDL_XPATH, applyXPATH,
    '\\Envelope\body\EncryptedData\EncryptionMethod\@Algorithm',
    \returnString(WSDL_SOAP_DC, getSOAP,
      'BookShopPolicy', 'Data')
  ) == 'http://www.w3.org/2000/09/xmlenc#3des-cbc';
</wscol:Expression>
```

In this expression, we suppose to have an XPath data collector (`WSDL_XPATH`) and a SOAP data collector (`WSDL_SOAP_DC`). The first retrieves data corresponding to XPath expressions. The second generates the SOAP message that correspond to invocations of external Web Services.

The WS-CoL expression uses two nested `\returnStrings`. The inner one, by using the SOAP data collector, asks to produce an encrypted SOAP message using the encryption policy stated in the initial WS-Policy file and the data received from the

⁴ Recovery strategies are not part of this paper and are our future work.

instrumented process. Amongst these data is the WSDL of the service that must be invoked with the encrypted message. This is needed for understanding the structure of the SOAP message that has to be built. The outer `\returnString`, extracts a value (whose location is specified using an XPath expression) contained in the header of the just newly built SOAP message. In the meanwhile, the encrypted SOAP message, as built by the SOAP data collector, is kept untouched in the Invoker, preventing it from being modified. This is fundamental since it represents the actual message that will be sent to OnlineBank, once its correct encryption is proven. The value extracted by the XPath data collector is eventually confronted with “3DES” by the CLiX data analyzer. If the message results to be encrypted correctly, the Invoker is instructed to forward the message to the OnlineBank service. If the message is not encrypted correctly, the system raises an exception that is passed to the instrumented process.

The return message received by OnlineBank must also be monitored for correct encryption. Once the return message is received by the Invoker, it is copied and passed to the XPath data collector that extracts the header element to confront it with “3DES”. Once again, we use a WS-CoL expression containing a `\returnString` call to the XPath data collector. The extracted values are then passed to the CLiX data analyzer. If the message results to be correctly encrypted, it is passed to the SOAP data collector for decryption, after which the result of the decryption is forwarded to the instrumented process. If the message is not correctly encrypted, an exception is raised and passed to the instrumented process.

Generally speaking, given a generic WS-Policy assertion to be monitored, if a data source capable of identifying the effects of such an assertion exists, we can derive a WS-CoL expression. This expression instructs the monitoring manager to state if the non-functional properties the user requires are satisfied.

6 Related Work

The research initiatives undertaken in the field of web service monitoring share the common goal of discovering erroneous situations during the execution of services. They differ, although, in a number of ways: degree of invasiveness, abstraction level at which they work, reactivity or pro-activeness, and nature of erroneous situations they are capable of discovering.

For example, Spanoudakis and Mahbub [14] have developed a framework for monitoring requirements of WS-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 deployed services. The first can be extracted automatically from the WS-BPEL specification of the process, while the latter must be specified by the user. Events are then observed at run-time. They are stored in a database and the run-time checking is done by an algorithm based on integrity constraint checking in temporal-deductive databases.

Lazovik et al. [15] proposes another approach based on operational assertions and actor assertions. The first can be used to express properties that must be true in one state before passing to the next, to express an invariant property that must hold throughout all

the execution states, and to express properties on the evolution of process variables. The second can be used to express a client request regarding the entire business process, all the providers playing a certain role in the process execution, or a specific provider. The system then plans a process, executes it, and monitors these assertions. This approach shares with ours the fact of being assertion-based. Once the assertions are inserted, it is completely automatic in its setup and monitoring. It lacks although the possibility of dynamically modifying the degree of monitoring. It also lacks adoptability since it is based on proprietary solutions.

A third approach, Cremona (Creation and monitoring of WS-Agreements) project [16] is the only one which, so far, is based on WS-Policy, embedded in WS-Agreement declarations, to express the non-functional requirements. WS-Agreement [17] is a standardization effort of the Global Grid Forum that defines an agreement protocol based on XML. This standard defines agreements for interfaces, security and quality of service properties. Cremona provides a framework that simplifies the definition, the management and the run-time monitoring of the state of the agreements.

7 Conclusions and Future Work

Even if WSDL represents the standard way to define what a Web service does, many efforts are now focusing on languages that can complete such a description by considering aspects that are not directly related to how a service should be invoked. WS-Policy, and all the other languages included in the WS-Policy framework, represent one of the most well-known attempts and, due to its flexibility, could be a candidate to become the future standard. For these reasons, in this work, we decided to extend the WS-Policy framework by proposing WS-CoL, as a domain-independent assertion language.

This paper is only a first proposal to embed monitoring directives into policies. The first implementation of the Monitoring Manager and the experiments with the (complete) example presented in this paper gave promising results, but the approach needs further analysis and a wider set of case studies to fully assess its soundness. Rules driving the automatic translations from WS-Policy assertions to Ws-CoL expressions are under development. All these activities are facilitated by the availability of our monitoring framework.

References

1. M. P. Papazoglou and G. Georgakopoulos. Service-oriented computing: Introduction. *Communication ACM*, 46(10):24–28, 2003.
2. J. Schlimmer (ed.). Web Services Policy Framework (WS-Policy Framework). www.ibm.com/developerworks/library/specification/ws-polfram/, September 2004.
3. A. Nadalin (ed.). Web Services Policy Assertions Language (WS-PolicyAssertions). www.ibm.com/developerworks/library/ws-polas/, May 2003.
4. N. Mukhi, P. Plebani, T. Mikalsen, and I. Silva-Lepe. Supporting Policy-driven behaviors in Web services: Experiences and Issues. In *Proceedings of the Second International Conference on Service Oriented Computing (ICSOC2004)*, New York, NY, USA, 2004.

5. P. Nolan. Understand WS-Policy processing. Explore Intersection, Merge, and Normalization in WS-Policy <http://www.ibm.com/developerworks/webservices/library/ws-policy.html>.
6. K. Verma, R. Akkiraju and R. Goodwin. Semantic Matching of Web Service Policy. <http://lsdis.cs.uga.edu/kunal/publications/SemanticPolicy-SWDP-final.pdf>.
7. C. Sharp (ed.). Web Services Policy Attachment (WS-PolicyAttachment). www-128.ibm.com/developerworks/library/specification/ws-polatt/, September 2004.
8. N. Delgado, A. Q. Gates and S. Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Transactions on software Engineering*, pages 859-872, December, 2004.
9. D. C. Luckham. Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs. *Texts and Monographs in Computer Science*, Oct 1990.
10. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *Department of Computer Science, Iowa State University, TR 98-06-rev27*, April, 2005.
11. L. Baresi, C. Ghezzi and S. Guinea. Smart Monitors for Composed Services. *In Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.
12. XlinkIt: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Software Engineering and Methodology*, pages 151-185, May 2002.
13. CLiX: Constraint Language in XML. www.clixml.org/clix/1.0/.
14. K. Mahbub and G. Spanoudakis. A Framework for Requirements Monitoring of Service Based Systems. *In Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.
15. A. Lazovik, M. Aiello and M. Papazoglou. Associating Assertions with Business Processes and Monitoring their Execution. *In Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.
16. H. Ludwig, A. Dan, R. Kearney. Cremona: an architecture and library for creation and monitoring of WS-Agreements. *In Proceedings of the Second International Conference on Service Oriented Computing (ICSOC2004)*, New York, NY, USA, 2004.
17. Web Services Agreement Specification (WS-Agreement), 2005. ws.apache.org/wsif/.