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)
baresi|guinea|plebani@elet.polimi.it

Abstract. The paper presents a monitoring framework for WS-BPEL processes. It proposes WS-Policy and WS-CoL (Web Service Constraint Language) as domain-independent languages for specifying the user requirements (constraints) on the execution of Web services compositions. These languages provide a uniform framework to accommodate both functional and non-functional constraints, but 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 [5]. 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 access 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 able to probe both functional and non-functional requirements. Functional requirements predicate on the correctness of the information exchanged between the WS-BPEL orchestrator and selected services; non-functional requirements are about the aspects directly related to how well the service works in term of, for example, security, transactionality, performance, and reliable messaging. The capability of probing such a wide range of requirements imposes that the execution 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. Moreover, cases 1 and 4 assume client-side monitoring, while the other two cases are wider and open to the options introduced above.

The approach concentrates on client-side monitoring and relies on WS-Policy [3], 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 invocation[1] with monitoring capabilities.
Even if 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, and Section 4 presents the architecture of the monitoring framework and exemplifies how it works. Section 5 briefly surveys related approaches and concludes the paper.

2 WS-Policy and WS-CoL

WS-Policy [3] 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 [1], 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, we propose WS-CoL (Web Service Constraint Language), as domain-independent language to express monitoring constraints.
As stated in [4], policies can be defined at different phases, and by several actors, of the Web service life-cycle (Figure 1). Besides implementing the application, service devel-

[1] For example, existing service buses.

opers also specify the properties that must hold during the execution independently 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 that support the deployment of the services (*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 the services they want to invoke should support (*requested policies*). By combining requested policies and supported policies, we obtain the so called *effective policies*, that is, 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.

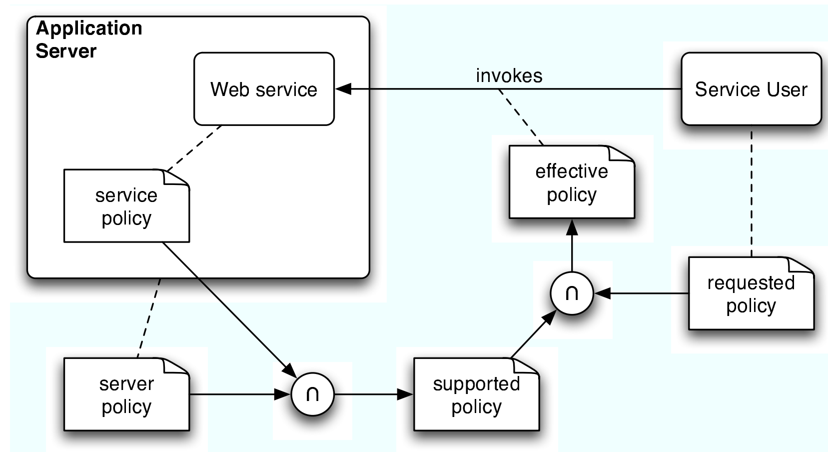

Fig. 1. Ws-Policy definitions and attachments

WS-PolicyAttachment [2], one of the elements of the WS-Policy framework, supports the scenario described above by introducing 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: the whole process, a branch of execution, a service invocation, a single message, or a single internal variable. Hereafter, for simplicity, we suppose that all the effective policy assertions work at the same level and, more precisely, at service invocation level.

If the considered service is a WS-BPEL process, policies can be attached to some of the service invocation activities. Figure 2 shows a possible effective policy attachment[2], where policy `BookShopPolicy` is applied to all the subjects identified by the XPath

[2] Namespaces are not included for the sake of readability.

expression in the MonitoredItem tag. The type attribute specifies when the expressions included in the policy must hold.
The effective policy, which must be satisfied when the credit card is about to be charged, is defined in the second part of Figure 2: the BookShopPolicy states both functional and non-functional properties. Non-functional requirements impose that all exchanged messages be encrypted using 3DES as the encryption algorithm. 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 WS-CoL 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).

1. Policy attachment:

```
<wsp:PolicyAttachment xmlns:wsp="...">
  <wsp:AppliesTo xmlns:wsal="...">
    <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:wsse="..."
           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 <= uP.moneyCap;
    </wscol:Expression>
  </wsp:All>
</wsp:Policy>
```

Fig. 2. Ws-Policy example

WS-CoL (Web Service Constraint Language) borrows many concepts from JML. It distinguishes between *data collection* and *data analysis*. Data can come from the process directly (e.g., input and output messages), but they can also come from any external

source (e.g., exchanged SOAP messages, metering tools). This is possible because of a set of keywords representing ways of obtaining data from external data sources. A different extension is introduced for each of the standard XSD types that can be returned by external data collectors: `\returnInt`, `\returnBoolean`, `\returnString` provide data according to the specified format. These extensions can be nested to make a service filter (or compose) the data gathered from other 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` [10] and `CLiX` [11].

3 Monitoring approach

Runtime monitors [6] 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 [7]) and JML (Java Modeling Language [8]), and proposes the use of special-purpose assertions to check the correctness/quality of running WS-BPEL processes. It is also based on the idea that we want to reuse as much existing technology as possible as means to increase its diffusion and acceptability[3].

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 BPEL^2, 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 BPEL^2 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. BPEL^2 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 resources can be released.

[3] 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.

$BPEL^2$ 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. $BPEL^2$ only adds calls to the monitoring manager. This means that policies can change without re-instrumenting the process. If locations changes, then $BPEL^2$ would produce a different specification.

After the weaving process at deployment-time, monitoring policies can be switched on and off at runtime [9]. 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 that oversees the application of the monitoring policies, has a dedicated user interface that lets the designer change its current priority and thus modify the impact that monitoring has on the execution dynamically.

4 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 sequence diagram of Figure 4 shows how such components interact while executing a WS-BPEL process if the monitoring of pre-conditions is required. When $BPEL^2$ produces the instrumented version of the process, it adds an initial call to the manager that sets up the monitoring activities by creating a specific configuration in the Configuration Manager. This configuration contains all the policies that are selected for the process.

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. 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 our example of 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.

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

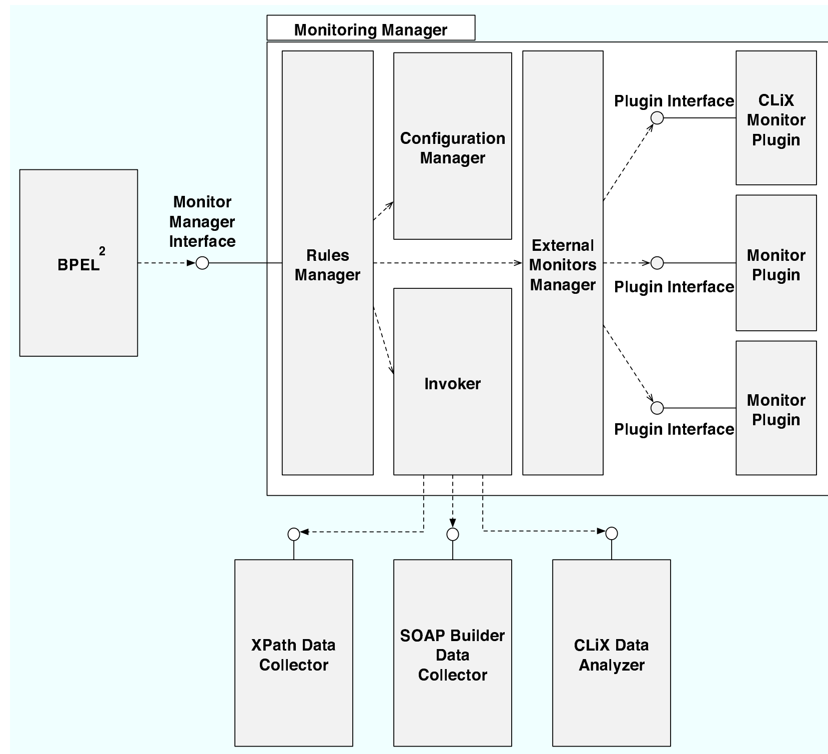

Fig. 3. Interaction with the Monitoring Manager

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. 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. 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. 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. Once this translation is completed, the appropriate data analyzer is invoked 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. If the data analyzer responds by saying that the

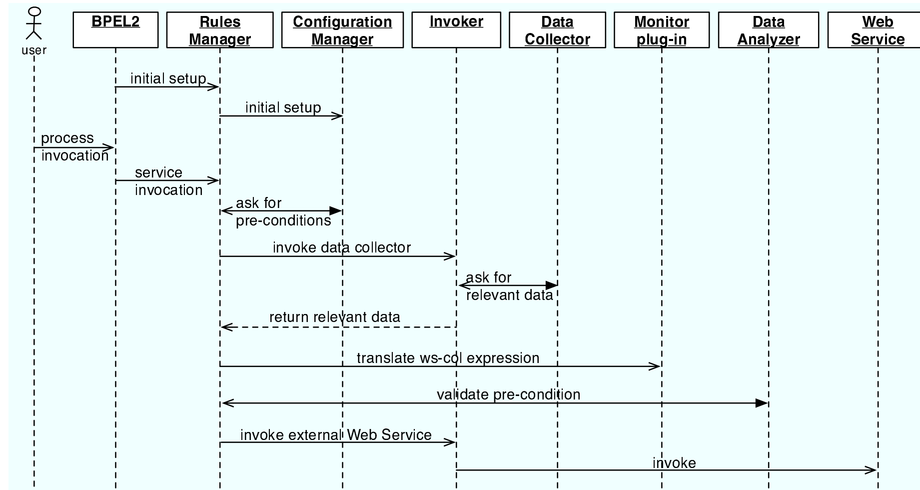

Fig. 4. Interactions among the main elements of the monitoring manager

property is not valid, a standard exception is raised to the instrumented process which can then decide for some recovery strategies[4].

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. The WS-CoL expression is extracted from the Configuration Manager component and the Rules Manager discovers that the amount of money, that has to be charged to the client’s credit-card, and the money cap that the client has set in his/her user preferences are required. Both of these data originate in the instrumented process that is in execution, and as such, have already been sent to the manager during the request for monitoring. No extra data collection is necessary. The WS-CoL expression and the data are translated into formats interpretable by the CLiX monitor and sent to the appropriate data analyzer that responds that everything is fine.

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”. When the instrumented version of the WS-BPEL process is created, the encryption policy is translated into WS-CoL format, to make it interpretable by the manager. 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. The WS-CoL expression for the outgoing message is presented in Figure 5.

The WS-CoL expression makes use of two nested `\returnStrings`. The inner one is used to ask the SOAP Builder Data Collector to produce an encrypted SOAP message using the Encryption Policy stated in the initial WS-Policy file, and the data received from the instrumented process. Amongst these data is the WSDL of the service

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

```
<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>
```

Fig. 5. WS-CoL encryption expression for the outgoing message

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`, on the other hand, is used to extract a value (the location of which is specified using an XPath expression) contained in the header of the just built SOAP message. In the meanwhile, the encrypted SOAP message, as built by the SOAP Builder Data Collector, is kept untouched in the Invoker component. This prevents it from being modified by anyone, which 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 finally confronted with "3DES" by the CLiX Data Analyzer. If the message results to be encrypted correctly, the Invoker is instructed to forward the message it has been holding to the OnlineBank service. If the message is not encrypted correctly, an exception is raised and passed to the instrumented process.
The return message received by OnlineBank must also be monitored for correct encryption. Once the return message has been received by the Invoker component, it is copied and passed to the XPATH Data Collector which extracts the header element to confront it with "3DES". Once again a WS-CoL expression containing a `\returnString` call to the XPath Data Collector is used. 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 Builder for decryption, after which the result of the decryption is finally forwarded to the instrumented process. If the message is not correctly encrypted, the usual exception is raised and passed to the instrumented process.
Generally speaking, given a generic WS-Policy assertion to be monitored, if a data source able to identify the effects of such an assertion exists, we can derive a Ws-Col expression. So, a WS-Policy assertion results in an expression like the one presented in Figure 5. This expression drives the monitoring manager to state if the non-functional properties the user requires are satisfied.

5 Conclusions and future work

Lack of space precludes a thorough survey of all the approaches that address Web services monitoring. Here, we only concentrate on some relevant initiatives.
Even if WSDL represents the standard way to define what a Web service does, many efforts are now focusing on languages able to complete such a description by considering aspects 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, it 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 domain-independent assertion language.
Similarly, WSLA [12] and WS-Agreement [13] propose domain-independent frameworks capable of collecting properties. Also RuleML [14] can be used to express constraints in terms of facts and rules.
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 the monitoring framework.

References

1. A. Nadalin (ed.). Web Services Policy Assertions Language (WS-PolicyAssertions). `www.ibm.com/developerworks/library/ws-polas/`, May 2003.
2. C. Sharp (ed.). Web Services Policy Attachment (WS-PolicyAttachment). `www-128.ibm.com/developerworks/library/specification/ws-polatt/`, September 2004.
3. J. Schlimmer (ed.). Web Services Policy Framework (WS-Policy Framework). `www.ibm.com/developerworks/library/specification/ws-polfram/`, September 2004.
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. M. P. Papazoglou and G. Georgakopoulos. Service-oriented computing: Introduction. *Communication ACM*, 46(10):24–28, 2003.
6. 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.
7. D.C. Luckham. Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs. *Texts and Monographs in Computer Science*, Oct 1990.
8. Gary T. Leavens, Albert L. Baker, and Clyde 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.
9. L. Baresi, C. Ghezzi and S. Guinea. Smart Monitors for Composed Services. *In Proceedings of the 2nd International Conference on Service Oriented Computing*, 2004.
10. XlinkIt: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Software Engineering and Methodology*, pages 151–185, May 2002.
11. CLiX: Constraint Language in XML. `www.clixml.org/clix/1.0/`.
12. A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Technical Report RC22456(W0205-171), IBM Research Division, T.J. Watson Research Center, May 2002.
13. Web Services Agreement Specification (WS-Agreement), 2005. `ws.apache.org/wsif/`.
14. The Rule Markup Initiative. `www.dfki.uni-kl.de/ruleml/`.