A Language for Quality of Service Requirements Specification in Web Services Orchestrations

Fabien Baligand[1], Didier Le Botlan[2], Thomas Ledoux[2], and Pierre Combes[1]

[1] France Telecom - R&D / MAPS / AMS,
38-40 rue du general Leclerc, 92794 Issy les Moulineaux, France
firstname.name@orange-ftgroup.com
[2] OBASCO Group, EMN / INRIA, Lina
Ecole des Mines de Nantes,
4, rue Alfred Kastler, F - 44307 Nantes cedex 3, France
firstname.name@emn.fr

Abstract. Service Oriented Architectures industry aims to deliver agile service infrastructures. In this context, solutions to specify service compositions (mostly BPEL language) and Quality of Service (QoS) of individual services have emerged. However, architects still lack adapted means to specify and implement QoS in service compositions. Typically, they use ad-hoc technical solutions that significantly reduce flexibility and require cost-effective development. Our approach aims to overcome this shortcoming by introducing both a new language and tool for QoS specification and implementation in service compositions. More specifically, our language is a declarative domain-specific language that allows the architect to specify QoS constraints and mechanisms in Web Service orchestrations. Our tool is responsible for the QoS constraints processing and for QoS mechanisms injection into the orchestration. A key property of our approach is to preserve compatibility with existing languages and standards. In this paper, we present our language and tool, as well as an illustrative scenario dealing with multiple QoS concerns.

1 Introduction

A Web Service is a component accessible over the Web that aims to achieve loose coupling between platforms through the use of XML documents and standardized protocols. As a platform neutral technology, Web Services bring a relevant solution to companies that wish to open their Information System and to allow other businesses to connect to their services. Quality of Service (QoS) is an essential criterion for customers seeking a service. A way to specify the customer-provider relationship is to enrich Web Services with QoS documents that aim to provide some guarantees (*e.g.* throughput, response time, price). For the time being, several approaches [7,3,12,4] allow to publish service QoS so that both customers and providers can agree to a certain QoS level.

Additionally, because of their loose coupling property, Web Services can be easily assembled. This feature is especially useful for business companies that want to

D. Georgakopoulos et al. (Eds.): ICSOC 2006 Ws, LNCS 4652, pp. 38–49, 2007.

integrate several services from different sources. For instance, telecommunication companies plan to provide their products (cell phones) with integrated services, built on top of other services delivered by external businesses. Multiple tools and languages [1,2] are available for Web Service composition purpose.

When composing services, some issues dealing with QoS arise. First, a crucial issue is to guarantee global properties of the assembly. Architects need methods or tools either to deduce the global QoS of a composition where each service QoS is known, or to enforce QoS requirements over the composition. For now, architects still have to deal manually with SLA combinatory of services involved in their composition. The second issue relates to the complexity of adapting a composition to a specific context. At deployment time, the system architect has to ensure that the workflow fits to some specific QoS requirements. Dealing with security, reliability and other QoS mechanisms (*e.g.* load balancing or message queuing) requires a consequent expertise of the platform and of WS-* standards (*e.g.* WS-Security, WS-Reliability). System architects lack abstractions to address such QoS requirements.

In this paper, we present a new approach aiming to provide the architect with adequate means to specify QoS requirements in Web Service compositions. To this end, we design a language, named "QoSL4BP" (Quality of Service Language for Business Process) or "QoSL" for short, that abstracts QoS concerns from the low-level details of Web Services compositions. Additionally, we propose a tool, named "ORQOS" (ORchestration Quality Of Service) that interprets QoSL4BP and that produces an orchestration enhanced with QoS concerns. We illustrate the whole approach with a scenario.

The remainder of the paper is organized as follows: Section 2 gives a brief introduction of Web Service QoS and composition, before we talk about the issues that result from the intersection of these two topics. Section 3 describes the approach we propose to overcome these shortcomings, introducing both the language and the mechanisms related to the implementation of QoS specifications. Section 4 illustrates our approach through an illustrative use case. Section 5 discusses of related works while Section 6 concludes and outlines future work.

2 Background

2.1 Quality of Service in Web Service World

Web Services are built on standards widely adopted. Best known among these standards are WSDL (interface specification), SOAP (communication protocol), and UDDI (service description and discovery). There also exist multiple WS-* standards that address non functional properties (*e.g.* WS-Security or WS-Addressing). Although most of these definitions have resulted into solid standards, there still is no consensus to the question of how one should publish the QoS of Web Service.

Among different approaches to predict the QoS of a service, current works focus on promoting the use of Service Level Agreement (SLA). SLA consists in a prescription concerning QoS between a consumer and its provider. For the time

being, there is no standard SLA definition and each vendor has designed and implemented its platform specific solution (IBM Web Service level Agreement, HP Web Service Management Language, Web Service Offering Language, WS-Agreement).

2.2 Web Service Composition

Using several Web Services as building blocks, architects may design more elaborated Web Services by composing them. A possible way to achieve Web Service composition is to design a message workflow and specify the services to be called within the workflow. Such a composition is called an orchestration. Orchestration processes are centralized and one entity is responsible for the execution logic of the whole workflow. This method is convenient for maintenance and evolution since the architect can design the workflow in the boundaries of his own business.

To design such orchestrations, many languages have merged into a standard and widely agreed language named "Business Process Execution Language for Web Services" (BPEL4WS or BPEL for short) [2]. This language is based on a small set of primitives (*e.g.* invoke, receive, reply, flow, throw), hence allowing any architect to specify the structure of his orchestration.

2.3 Current Issues

At pre-deployment time, the SLA of local services (*i.e.* services that take part in the composition) is computed to deduce the SLA of the composite service (bottom-up approach), or, conversely, the architect must resolve a set of local services whose SLA aggregation match the SLA of the composite service (top-down approach). However, both approaches do not take into account architects advanced QoS requirements. For instance, the architects may want to guaranty the SLA of their orchestrations, while specifying QoS of some parts of their orchestrations and requiring that some local services are discovered to match the global SLA. In this case, SLA documents do not provide expressivity to address such QoS requirements. Furthermore, being able to specify QoS mechanisms, such as security, over parts of the orchestration is a major concern that SLA cannot address either.

Because BPEL language does not provide expressivity for QoS management, and since SLA are limited, architects cannot easily declare QoS requirements and logic in their orchestrations. Instead, they specify QoS management at the message level, using multiple frameworks and languages. Making all these frameworks work together leads to code that lacks flexibility and portability, that decreases loose coupling nature of the composition, and which is error-prone.

3 Quality of Service Requirements Specification

3.1 Motivation

BPEL is a language allowing the architect to design orchestrations. From the outside, an orchestration can be seen as a composite service with a WSDL functional interface. A SLA document can be associated to a service or a composite

Table 1. Web Services related Languages

	Functional Description	QoS Description
External Interface	WSDL	SLA
Composition Implementation	BPEL	Lack of expressivity

service to specify its QoS properties. However, there is a lack of expressivity for architects willing to specify QoS objectives and mechanisms in the orchestration (*e.g.* the architect may want to specify security, performance or even pricing requirements over parts of his orchestration). We give an illustration of this discussion in Table 1.

To overcome this lack of expressivity, we can consider several guidelines, such as specifying a QoS extension for BPEL, or implementing a QoS-aware BPEL engine, or designing a specific language and platform for QoS requirements specification and enactment in BPEL. Because a key decision of our approach was not to be intrusive, in order to preserve existing infrastructures and languages, we choose not to extend BPEL language or to implement a new BPEL engine. Moreover, because languages improve reusability and portability, we decided to provide an appropriate language, namely "QoSL4BP" (Quality of Service for Business Process) or QoSL for short, for QoS requirements specification in Web Service compositions. More specifically, QoSL allows to specify QoS constraints and mechanisms in Web Service orchestrations.

3.2 Design

To design the QoSL language, we focussed on a couple of properties that seem relevant to our application domain:

- QoSL is a domain-specific language (DSL) [9]. Domain expertise is captured in the language implementation rather than being coded explicitly. The domain of QoSL corresponds to "Quality of Service applied to Web Service Orchestrations", encompassing sub domains (*e.g.* security or performance).
- QoSL is a declarative language [10] as it meant to be goal driven. Control is not the concern of the architect who does not need to provide a fully detailed list of instructions to specify QoS objectives and mechanisms.
- QoSL is modular. Separation of concerns is the process of breaking a program into distinct features, hence increasing code clarity and evolutivity. In order to separate concerns, the QoSL language isolates the code into several modules (policies), each of them capturing one particular QoS concern.

3.3 Specification

Policies are the basis of QoSL language structure. A policy represents a single QoS concern of the service and contains a set of rules, which are composed of QoS constraints and mechanisms. In QoSL, a policy has a name, targets a scope of the orchestration (a scope represents a single activity, multiple activities or

Table 2. QoSL Language Entities

Language Entity	Meaning
scope (Where)	abstracts away an orchestration subset. represents a single activity, multiple activities, or even the whole orchestration.
concern (What)	specifies which QoS concern is addressed in a specified scope. enforces separation of concerns.
policy (How)	consists in a module encompassing QoS constraints and mechanisms in a scope for a specific QoS concern.

even the whole orchestration) and refers to a particular QoS concern to enforce homogeneity of the body of the policy. Since QoSL aims to bind BPEL, SLA and WS-* standards, then QoSL borrows some abstractions from these languages. Table 2 shows entities of the QoSL language.

To implement the body of policies, we focussed on adapted constructs. QoSL allows to bind variable names to values (using "let variableName = *value*"), to define constraints ("check QoSproperty > *quantitativeRequirement*") and to set mechanisms ("set QoSmechanism(parameterList)"). It allows the architect to specify essential parameters, related to the policies he defines, as well as parameters that would otherwise be inferred (*e.g.* weights for weighted round-robin load balancing algorithm). We present here a first version that will be refined and extended later on (*e.g.* to integrate policy composition primitives). Figure 1 illustrates the structure of actual QoSL language.

```
orqos orqos_name {
    policy policy_name scope BPEL_scope_name concern QoS_concern {
        // specify some variable
        let variable_name = value ;
        // specify some constraint
        check QoS_property > quantitative_requirement ;
        // specify some mechanism
        set QoS_mechanism( parameterList ) ;
    }
}
```

Fig. 1. QoSL Policy Template

It is worth noting that although our approach enables the architect to write policies, it is different of WS-Policy. WS-Policy provides a grammar for expressing the capabilities and requirements of a single entity, whereas QoSL allows to set objectives and actions over parts of a workflow.

3.4 Interpretation

Figure 2 shows the QoSL interpretation process. This process is static and occurs before the workflow is instantiated in the BPEL engine.

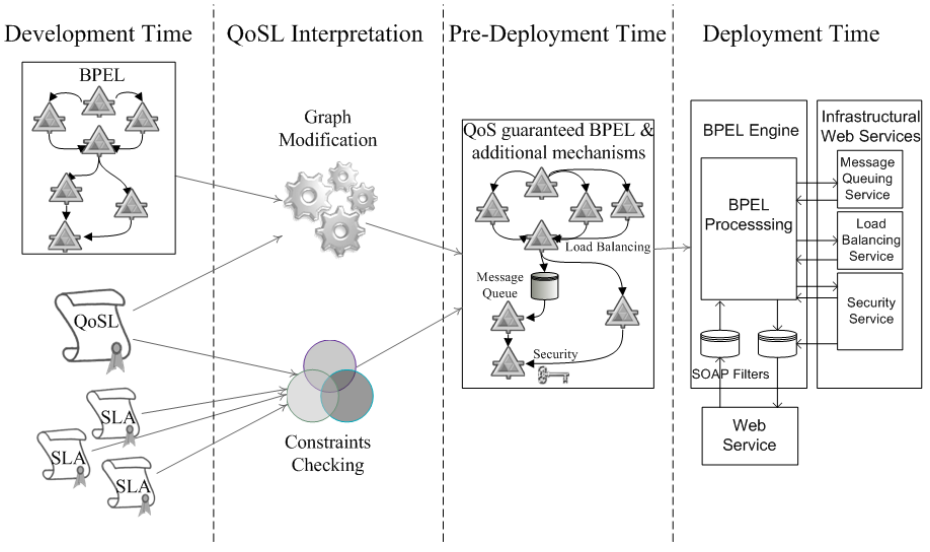

Fig. 2. QoSL Interpretation Process

At development time, the business architect is responsible for designing the workflow, using BPEL language. Then the system architect has to specify the non functional properties of the architecture. He collects SLA related to services involved in the orchestration and specifies QoS policies, using QoSL language. QoSL interpretation is performed by our tool ORQOS. The result is a new BPEL document and its associated SLA. BPEL document contains QoS mechanisms, and is ready to be deployed on a BPEL engine. Once deployed the BPEL workflow can be executed. While running, calls are made to some Web Services that contain QoS mechanism logic. For the time being, runtime monitoring and SLA violation are not handled by ORQOS.

3.5 ORQOS Process

ORQOS aims to process the QoSL policies specified by the architect. A major issue consists in dealing with policies composition to avoid possible interactions during the processing. For now, each policy is processed sequentially based on their order in the QoSL document. We plan to improve this model in our future works. So far, processing a QoSL document consists in five steps.

- First, ORQOS checks policies consistency to ensure that the code contained by a policy is homogeneous and that it addresses a single concern. For instance, a security mechanism should not be called within a policy whose concern is performance. By clearly separating the different concerns, it becomes easier to figure out how policies should be composed one with another.
- Second step corresponds to modifying the original BPEL graph in such a way that QoS mechanisms are introduced at specific places determined by

the scope of the policy (application domain). Such mechanisms are specified by the policies and are injected into the BPEL as "extra BPEL" activities. Potentially, extra BPEL activities set can include a wide range of QoS mechanisms such as traditional QoS mechanisms found in regular Web application servers, as well as mechanisms involving WS-* standards.
- Once the new workflow is generated, ORQOS sets constraints on each activity. Constraints are provided by SLA of local services and composite service, and by QoSL specifications concerning QoS requirements over scopes.
- Constraints are checked by a constraint solver. At this step, if the constraints defined by the architect cannot be satisfied, ORQOS stops and returns an exception. In the opposite case, ORQOS keeps processing and may use some of the results as parameters for QoS mechanisms settings (*e.g.* balance weights or capacities for message queuing) and to generate the SLA of the composition.
- Final step is the production of a BPEL document that can be read by any BPEL engine. Extra BPEL activities are refined into plain BPEL activity sequences that can involve some "invoke" activities that reach infrastructural Web Services containing mechanism logic (*e.g.* for security concern, an infrastructural Web service is responsible for WS-Security implementation logic). Thus, the potential boundaries of the approach correspond to the set of actions that can possibly be performed by modifying a BPEL document and calling logic contained in infrastructural Web Services.

4 Illustrative Scenario

4.1 Urban Trip Planner Scenario

Depicted in Figure 3, the "Urban Trip Planner" (UTP) scenario illustrates a Web Service orchestration. It aims to plan trips in big cities, by using transportation services. By calling the UTP service, a client gets the complete transportation route, as well as a map showing the path from the last station to his final destination.

As can be seen on Figure 3, the Urban Trip Planner Service is composed of multiple services. It requires both a destination and a device identification number as inputs. Next, the request is sent to two different services in parallel. These services belong a flow named "OrangeScope". The first service uses the device identification number and returns the client current location (for instance, using a Wifi access point location service). The second service takes the destination in input and returns the exact address, using the Yellow Pages service. Upon reception of both replies, the UTP service sends both addresses to a Transportation service that returns the route and commutes details. The final station address and the destination address are sent to a Grapher service that delivers a map of the path from the station to the destination. Eventually, both the route details and the map are returned to the user.

Let us now give an example of our approach through a scenario involving three QoS properties: throughput, capacity and authentication properties. Throughput is the number of requests by second that a service can process. Capacity

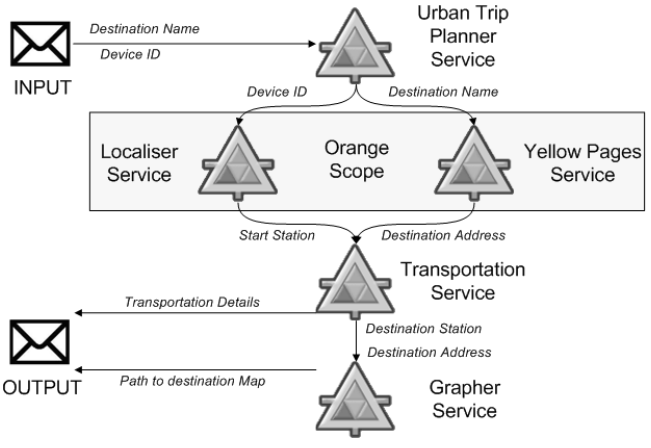

Fig. 3. Urban Trip Planner Service Workflow

relates to the limit of the number of simultaneous requests which should be provided with guaranteed performance. Authentication is the ability of a client to show credentials to access a service. To address these QoS properties, the architect would like to specify some QoS constraints and some QoS mechanisms in the UTP orchestration.

4.2 Specification of QoS Concerns

Once the business architect has implemented the workflow using BPEL, the system architect has to ensure it fits to some QoS requirements. Thus, before deploying the orchestration, the system architect reads each SLA document, then deduces and validates the global QoS of the assembly.

To adjust to services offers and requirements, he also needs to implement QoS mechanisms such as load balancing, message queuing and username token authentication. Such information can be specified using QoSL policies. For this, constraints and mechanisms targeting a same QoS concern in a same scope are gathered in a same policy. The policies corresponding to the requirements of the architect are shown on Figure 4.

- The first policy is called "IncreaseGrapherThroughput" and targets the single "InvokeGrapher" activity. It specifies that the throughput property of the Grapher service should be more than 5 000 requests/sec and introduces a load balancing mechanism using the weighted round robin algorithm over Mappy, GoogleMap, and MapQuest services with weights 3, 2 and 5.
- The second policy, called "ManageGlobalCapacity" aims to force the orchestration capacity to be more than 200 parallel requests. It specifies that a queuing mechanism can be used to increase the capacity.
- The third policy affects the security settings when calling services of the "OrangeScope" (which are the Localiser and the Yellow Pages services). The

```
orqos UTPpolicies {
    policy IncreaseGrapherThroughput scope InvokeGrapher concern Performance {
        check Throughput ≥ 5000 ;
        let $MyLinks={Mappy, GoogleMap, MapQuest} ;
        set LoadBalancing(Algorithm=WeightedRoundRobin, PartnerLinks=$MyLinks,
            Weights={3, 2, 5}) ;
    }
    policy ManageGlobalCapacity scope UTPFlow concern Capacity {
        check Capacity ≥ 200 ;
        set Queue() ;
    }
    policy OrangeAuthentication scope OrangeScope concern Security {
        let $MyToken = {ORQOSuser, mypassw} ;
        set Authentication(Token=$MyToken) ;
    }
}
```

Fig. 4. QoSL Document for UTP Orchestration

architect defines a username token (with name "ORQOSuser" and password "mypassw") and specifies the authentication mechanism to use.

4.3 Interpretation of QoSL

As said earlier, ORQOS processes QoSL documents through five steps. At first step, it takes the BPEL document of the UTP orchestration document, as well as the QoSL document stating the three policies (IncreaseGrapherThroughput, ManageGlobalCapacity and OrangeAuthentication) and the SLA of each service (Localiser, Yellow Pages, Transportation and Grapher services). These documents are translated to objects and policies homogeneity is verified.

Next, the graph describing the UTP workflow is modified. A "Load Balancing activity" replaces the Grapher invoke activity and encapsulates a Mappy, MapQuest and GoogleMap invoke activities. Each invoke activity is encapsulated in a "Message Queuing activity". The OrangeScope is encapsulated in an "Authentication Activity".

Next step consists in setting constraints on each activity of the workflow. In this example, we focus on throughput and capacity properties. Each activity encapsulates both a throughput and a capacity parameter. Simple activities (invoke activities to services involved in UTP workflow) are linked to SLA information related to these properties, while composite activities (Load Balancing activity, Message Queuing activity, Orange Scope parallel branches) aggregates encapsulated activities parameters. Eventually, it results in a constraint network that is solved by a java constraint solver (Choco constraint solver).

The solver returns solutions to determine the capacity of the Message Queues activities, as well as weights for the weighted round robin algorithm of the Load Balancing activity. It also computes the SLA of the composite service so that the service customer can get a guaranty of the service QoS.

Finally, the graph is transformed back to plain BPEL. First, the Load Balancing activity gets replaced by a sequence of two activities: an invoke activity to a load balancing infrastructural Web Service to figure which Grapher service should be called for this specific instance of the workflow, then a switch activity tests the output and calls the appropriate Grapher service. Message queuing activities get replaced by an invoke activity to a message queue infrastructural Web Service before each invoke activity of the workflow. With regards to performance issues, tests have shown that load balancer service or message queues, using a cache mechanism for state management, do not significantly impact the BPEL engine performance. Then the Authentication activity gets replaced by two invoke activities, located around the Orange Scope, to an authentication infrastructural Web Service. This security service injects a security token through the filters of the BPEL engine when the orchestration reaches the Orange Scope. Because infrastructural Web Services are located on the same server as the BPEL engine, performance overhead is minimal. The new BPEL document is eventually generated and ready to be deployed.

5 Related Works

"Aspect Oriented for Business Process Execution Language" (AO4BPEL) [6] aims to bring AOP (Aspect Oriented Programming) mechanisms to BPEL. The authors have given examples of their solution with security aspects and use deployment descriptors to generate aspects. Although very promising, AO4BPEL is an imperative language and we do not believe that architects want to deal with implementation details in the BPEL process. We also think that deployment descriptors are limited when it comes to expressing mechanisms over multiple activities of the workflow and, since AO4BPEL approach does not benefit of SLA works, they do not address QoS constraints specification. Our belief is that architects would rather use a declarative language to specify constraints and mechanisms over different scopes of their orchestrations.

In [5] the authors propose a policy assertion language, "WS-CoL" (Web Services Constraint Language), based on WS-Policy and specifically designed for user requirements (constraints) specification on the execution of Web service compositions. This language is meant to be compliant with the WS-Policy framework, and its process requires a transformation step of the BPEL document to integrate some monitoring activities. This approach is similar to ours in that it provides expressivity concerning requirements on BPEL orchestrations. However, the authors only considered security assertions, using WS-SecurityPolicy.

"Self-Serv" [11] solution includes a platform to compose Web Services, a declarative language based on state charts and a "community of service" concept (containing alternative services), to add a layer between composition level and applicative services level. Composition execution is controlled by "coordinators" components that are in charge of initializing, controlling and monitoring the composition. Although this solution brings interesting elements to QoS management, the fact that BPEL is not supported is quite limitative.

[8] offers to extend BPEL with QoS attributes in order to manage SLA data written using the WSLA language. A new tag, named "agreement", has been inserted into BPEL schema. Architect should be able to specify some QoS constraints while actual QoS properties are evaluated through Computational Quality Attributes elements. Because this work extends BPEL, it is not compliant with regular BPEL engines. Moreover, these extensions do not deal with QoS mechanism and so the architect still has to implement logic of QoS mechanisms. Also, our approach allows the orchestration logic and QoS logic to be separated, hence improving flexibility and reusability.

6 Conclusion and Future Works

We first outlined a crucial issue concerning Web Service compositions. Although solutions exist to compose Web Services and to publish the Quality of Service of individual services, there is still no solution for QoS requirements specification in Web Service compositions. For now, system architects have multiple languages and frameworks to deal with, at deployment time. They also have to check each SLA to deduce the global outcome, making it a complex task to guarantee QoS properties of an assembly.

Our solution aims to provide abstractions for system architects, so they can specify and implement QoS requirements before deployment time. Our first contribution is a language, called "QoSL4BP" (Quality of Service Language for Business Process), that allows the architect to specify QoS constraints and mechanisms in their orchestration. Secondly, a tool, called "ORQOS" (ORchestration Quality Of Service), validates these constraints (by analyzing both the QoSL document as well as the multiple SLA documents) and injects QoS mechanisms (*e.g.* security mechanisms). QoS mechanisms logic is implemented by some infrastructural Web Services that can be reached via BPEL invocations. A key decision for the design of our approach was to not modify any existing language or standard from the Web Service world.

So far, we have built a first prototype that tackles with three different QoS concerns (throughput, capacity and security) and that is able to inject load balancing, message queuing and authentication mechanisms into an orchestration. Because QoS concerns are likely to be tangled up, we plan to study composition of policies in a more formal way as we extend our approach to other concerns.

References

1. Web service choreography interface (wsci) 1.0 (2002), `http://www.w3.org/TR/wsci/`
2. Business process execution language for web services, version 1.1 (2003), `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`
3. Keller, A., Ludwig, H.: The wsla framework: Specifying and monitoring service level agreements for web services. In: Journal of Network and Systems Management, March 2003, vol. 11, Plenum Publishing (2003)

4. Sahai, A., Machiraju, V., Sayal, M., van Moorsel, A., Casati, F.: Automated sla monitoring for web services. In: 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM, Montreal, Canada, pp. 28–41. Springer, Heidelberg (2002)
5. Baresi, L., Guinea, S., Plebani, P.: Ws-policy for service monitoring. In: Bussler, C., Shan, M.-C. (eds.) TES 2005. LNCS, vol. 3811, pp. 72–83. Springer, Heidelberg (2006)
6. Charfi, A., Schmeling, B., Heizenreder, A., Mezini, M.: Reliable, secure, and transacted web service compositions with ao4bpel. In: ECOWS. Proceedings of the 4th IEEE European Conference on Web Services, Zurich, Switzerland, December 2006, IEEE Computer Society Press, Los Alamitos (2006)
7. Li Ji Jin., et al.: Analysis of service-level agreement for web services. Technical Report HPL-2002-180 (2002)
8. Fung, C.K., Hung, P.C.K., Linger, R.C., Walton, G.H.: Extending business process execution language for web services with service level agreements expressed in computational quality attribute. In: HICSS-38. IEEE Thirty-Eighth Hawaii International Conference on System Sciences, Big Island, Hawaii, IEEE Computer Society Press, Los Alamitos (2005)
9. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. csur 37(4), 316–344 (2005)
10. Sethi, R.: Programming languages: concepts and constructs. Addison-Wesley Longman Publishing, Boston, MA (1989)
11. Sheng, Q.Z., Benatallah, B., Dumas, M., Mak, E.O.-Y.: Self-serv: A platform for rapid composition of web services in a peer-to-peer environment. VLDB02 (2002)
12. Tosic, V., Patel, K., Pagurek, B.: Wsol - web service offerings language. In: Bussler, C.J., McIlraith, S.A., Orlowska, M.E., Pernici, B., Yang, J. (eds.) CAiSE 2002 and WES 2002. LNCS, vol. 2512, pp. 57–67. Springer, London (2002)