Semantics in Service Discovery and QoS Measurement

The industry’s promotion of Web services has prompted providers to develop and publish many Web services. Consequently, service requesters have a choice of offerings that provide similar functions. These offerings can differ significantly in quality of service (QoS). Thus, comparing services now requires more sophisticated patterns of service discovery and negotiation.

In practice, we can divide service discovery into two phases. In the first phase, a requester discovers the service using the functional aspect of the service—what the service does, its input and output parameters, preconditions, and effects. This phase ensures that the returned services meet the requester’s basic requirements. The second phase is to identify the most-appropriate service for the current task; it accounts for the nonfunctional information, such as QoS, about each service returned in the first phase. This second phase has become an important research area in process combination and resource discovery because dynamic binding and invocation are preferable.

The current service discovery process adopts keyword-matching technology to locate published Web services, basing this matchmaking on syntax level, that is, matching according to a set of weighted keywords, a less-than-desirable situation. The returned discovery result might not satisfy the requester’s intended requirements. This leads to a bit of manual work to choose the proper service according to its semantics. From the Web services point of view, the selection criteria should at least include the service’s functional and nonfunctional requirements. To fully integrate service discovery, these domain-specific criteria should be clear and processed automatically. This requires domain-specific knowledge.

In this light, Semantic Web technology is a promising innovation for service discovery. It requires that data is not only human readable, but also machine understandable. Semantic Web concepts can help provide a unifying system that minimizes misunderstandings among different partners. The OWL-S (Web Ontology Language for Services) ontology is one effort by a group of Semantic Web researchers to describe Web services (http://www.daml.org/services/owl-s/1.1/). It aims to enable automated Web service discovery, invocation, composition, and monitoring. It defines the notions of a service profile (what the service does), a service model (how the service works), and a service grounding (how to use the service). Because the service profile defines the service’s functional aspect, it meets the requirements set forth for service discovery’s first phase.

To cover the second phase, OWL-S has defined quality rating ontologies to describe the service’s QoS information. As others have pointed out, these ontologies require significant improvement to supply a realistic solution to DAML-S (Darpa Agent Markup Language for Services) users. One current limitation of DAML-S’ QoS model is that...
it does not provide a detailed set of classes and properties to represent QoS metrics. This motivates us to design the QoS ontology to compensate for this shortcoming of DAML-S in the second phase. In addition to defining the QoS ontology’s use in the second phase of service discovery process, we further define its ability to measure QoS.

**BASIS FOR A QOS ONTOLOGY**

We call our ontology DAML-QoS; it is a complementary ontology that provides detailed QoS information for DAML-S. In addition to covering a system’s nonfunctional aspects, it also covers QoS measurement.

Nonfunctional aspects describe constraints such as service level, management statements, security policies, pricing information, and other contracts between Web services and their users. During the design time, people normally focus more on the system’s functional rather than nonfunctional aspects. However, the best practice is to keep QoS issues in mind early in the design phase. During runtime, QoS specification should provide additional information to support dynamic service discovery, composition, integration, and monitoring. This also requires appropriate language support. As a QoS specification for Web services, our design principles include

- **Ease of use.** Developers should find the specification easy to use and understand. A clear specification helps developers to understand the system’s nonfunctional aspects and guides them toward choosing a proper system design pattern early on in development.
- **Precision and flexibility.** To allow value-added services for dynamic service discovery as well as automatic and customizable composition and integration, the QoS specification should have precision and flexibility. Precision means that it should answer the questions of when, which, where, what, and how the specification should be evaluated against the Web service [Towards Automated SLA Management for Web Services, Akhil Sahai, Anna Durante, and Vijay Machiraju, tech. report HPL-2001-310 (R.1), Hewlett-Packard Labs, 2002]. Precision permits the specification of customized metrics for diverse Web services. Flexibility permits the specification of customized metrics for diverse Web services.
- **Object-oriented style.** Because of the widespread acceptance of object-oriented design principles, it would help if the definitions for QoS requirements were also object oriented. When developers know the overall interface’s QoS requirements, it makes sense for lower-level interfaces to inherit QoS requirements and reuse the QoS constraints from the overall interface.
- **Automatic validation.** As the project grows, automatic validation becomes important. For example, it is desirable to automatically check whether the specification is correct in its syntax as well as its semantics.
- **Separation of design duties.** A design should separate its measurement details from metrics from metric definition. It is the measurement partners’ task to fill in the metric measurement details, according to the service execution environment. A measurement partner is the group that deploys the measurement handlers and performs the service measurement tasks. The service provider or requester can take on this role, or it can be outsourced to a third party.

**MODEL**

According to these design principles, we divided the ontology into a three-layer representation:

- The **profile layer** defines QoS matchmaking.
- The **property definition layer** defines properties and the domain and range constraints for such properties.
- The **metrics layer** defines QoS metrics and their measurement. It also defines precise instructions for the measurement partner to measure the service and check against the guarantee.

The QoS profile layer defines the service-level objective (SLO) for the Web service interfaces. An SLO is normally a set of parameters and their values. The model declares the class QoSProfile as a common superclass for all QoS specifications. Additional QoS constraints arise from the QoSProfile’s inherited concepts. Figure 1 gives an example: The service requester requires a Web service that implements the service interface and satisfies the QoS constraints in InquiryQoS, which is the requester’s description. The service provider promises to implement the service interface with the QoS constraints stated in ProviderQoS. These two descriptions are subclasses of QoSProfile but serve different purposes.

The constraint definitions in the SLO specification cover property domain, range, and cardinality. For example, property responseTime’s domain is QoSProfile and its range is the metric RespMSMetric (the response time metric in milliseconds). Cardinality constraints govern the number of metrics values, constraining them to remain within a specified number and range of values.

For example, we want to describe a service’s QoS by stating that its response time is less than 20 seconds and its cost is less than one dollar. We define such an SLO as a concept that allows SLO refinement and reuse based on inheritance:
Advert = QoSProfile
\( \sqcap (\leq 100 \text{cost.CostUSCentMetric}) \)
\( \sqcap (\leq 20000 \text{responseTime.RespMSMetric}) \)

where \( \sqcap \) indicates the conjunction of concept.

The individual SLO concepts link the specification to the DAML-S ontology to answer the “which” question, and it fills the time information to answer the “when” question. In the following example, we defined an agreement based on the Advert concept. The measured service, provider_service, is defined in a DAML-S' concept individual. This specification also calls out the start and end time of the agreement:

\[
\text{Advert(AdvertInstance)}
\]
\[
\text{hasServiceProfile(AdvertInstance, "&provider \_service;#")}
\]
\[
\text{startTime(AdvertInstance, 2004-09-24T09:00:00)}
\]
\[
\text{endTime(AdvertInstance, 2004-09-30T09:00:00)}
\]

We divide QoS metrics into AtomicMetrics and ComplexMetrics. The AtomicMetric’s individual definition provides necessary information to initiate the measurement handler. The measurement system uses the push method to report the collected measurement data. It builds complex metrics from other (AtomicMetric or Complex-Metric) metrics. This layer answers the where, what, and how questions for the measurement. The measurement partner defines this layer; its proper definition is key to effective system design and QoS monitoring.

This SLO definition is object oriented. Because the SLO definitions link directly to the service’s interfaces, object-oriented style is a natural choice for the corresponding SLO’s definition, especially when the project grows. The refinement of inherited SLO concept then defines the SLO. Through the conjunction of the property constraints, refinement tends to replace weaker constraints with stronger ones. All the other inherited QoS constraints defined in the superclass are reusable. Developers with object-oriented programming experience can quickly learn to understand this specification’s semantics to choose the proper design for their system.

The DAML-QoS ontology complements DAML-S. Developers can benefit from DAML-S for semantic matchmaking of service capability, as well as its well-defined process model and the grounding information. Meanwhile, the developer benefits from DAML-QoS for QoS matchmaking and QoS measurement. One Web service’s multiple QoS profiles can refer to the same DAML-S service profile. This provides multiple service-level objectives for the targeted Web service.

**PROTOTYPE SYSTEM ARCHITECTURE**

QoS matchmaking is a process that requires a repository to take an inquiry as input and to return all the published advertisements that conform to the inquiry’s QoS requirement. An SLO specification conforms to another SLO specification only if it is stronger than, or equally as strong as, the other constraint. Based on this definition, we have designed a matchmaking algorithm for DAML-QoS, together with its prototype.

The matchmaking algorithm in the service discovery’s second phase is defined as follows: This algorithm evaluates the relationship between the requester’s and the provider’s SLO concepts. Subsume, Exact, PlugIn, Intersection, and Disjoint are the possible relationships, in order from the
best to the worst match. In our implementation, illustrated in Figure 2, the prototype system contains an ontology repository, converter, and an ontology reasoner. The ontology repository holds all the published Web service descriptions. The converter changes the ontology written in DAML+OIL and OWL to the reasoner’s recognizable syntax. The reasoner is the part of the matchmaking framework that compares a QoS request specification to the service provider’s published specification to determine the degree of match.

We designed four interfaces for the system: inquiry, publish, browse, and administration. When service providers publish their service QoS profiles through the publish interface, the ontology parser will first parse the ontology, making sure that its format follows the DAML+OIL syntax. The reasoner is the part of the matchmaking framework that compares a QoS request specification to the service provider’s published specification to determine the degree of match.

If the parsing process ends successfully—that is, either reports errors or send the parsed ontology to the converter—the parser stores the ontology in the server’s ontology repository. Meanwhile, the converter renders the ontology into a description acceptable to the reasoner. When the requester submits the inquiry, the converter changes the inquiry and passes it to the reasoner. The reasoner then returns the matching result back to the requester. The current system is suitable for a small- or medium-sized (500 or so advertisement QoSProfile concepts) advertisement repository. For example, the service requester wants to select the service provider whose response time is less than 40 seconds and whose cost is less than five dollars. The provider can represent this requirement as follows:

\[
\text{Inquiry} = \text{QoSProfile} \\
\quad \cap (\leq 500 \text{cost}.\text{CostUSCentMetric}) \\
\quad \cap (\leq 40000 \text{responseTime}.\text{RespMSMetric})
\]

According to the matchmaking algorithm, the matchmaker will return the Advert concept because this is a \textit{Subsume} match. If the published advertisement does not satisfy the inquiry, the matchmaking algorithm will judge the degree of match accordingly. For example, disjoint is the worst degree of match. The validation of the specification’s correctness becomes more important as the project grows. Development requires the validation of two factors: syntax and semantics. Ontology parsers, such as Jena (“Jena: A Semantic Web Toolkit,” Brian McBride, \textit{IEEE Internet Computing}, Nov.-Dec. 2002) check syntax. The reasoner checks for semantic correctness.

The measurement system itself should be able to plug into the measured Web service system with minimum influence. Figure 3 shows sample architecture for the measurement system. Each measurement handler corresponds to one AtomicMetric individual, and the measurement system pushes the measured data to the collector service. Each collector corresponds to one ComplexMetric and will store the received data series for calculation. When the summary collector has all the required report data, it generates the measurement SLO in the same form as that of the service provider’s SLO. The service judges whether the measured SLO concept conforms to the agreed SLO.

The code generating process helps the measurement partner to develop its measurement system. After the parser checks the ontology and stores it, the code generator starts from the SLO for the top-level metrics. Then the code generator traverses the second-level metrics contained in the first-level ComplexMetrics, and so on. Code generation finishes when all the metric levels are complete.

**ADVANTAGES**

In contrast to the earlier work summarized in the “Other QoS Ontology Work” sidebar, we base our QoS language on the ontology layer instead of the pure-XML layer. This has some advantages related to the Semantic Web:

- **Interoperability.** In a complex scenario such as an enterprise application, various management systems and tools assist the system’s discovery, design, and running. These
Realizing the second phase requires the availability of two items: the description that describes the service’s QoS information and the corresponding matchmaking algorithm.

QoS Descriptions
Much research targets methods for describing, advertising, and signing up for Web or grid services at defined QoS levels. These methods are normally application-layer specifications that are hardware and platform independent. We list several here.

➤ Quo framework. This framework uses its own QoS Description Language (QDL) to describe services. It is an example of an aspect-oriented approach because it follows that paradigm to define the system components’ QoS properties (“Architectural Support for Quality of Service for CORBA Objects,” John A. Zinky, David E. Bakken, and Richard E. Schantz, Theory and Practice of Object Systems, Apr. 1997).


➤ WS-Policy. This framework comes from a collaboration of large vendors (BEA, IBM, Microsoft, and SAP). It is based on a general framework for the specification of policies for Web services. The particular categories of policies will be defined in specialized languages. (Web Services Policy Framework (WS-Policy) Version 1.0, Maryann Hondo and Chris Kaler (eds.), Dec. 2002; (http://www.verisign.com/ wss/WS-Policy.pdf)

Several frameworks for description rely on XML:

➤ WSML. Another HP product, the Web Services Management Language, relies on detailed QoS constraints and custom-made SLA. It is oriented towards management applications in inter-enterprise scenarios. (Towards Automated SLA Management for Web Services, Akhil Sahai, Anna Durante, and Vijay Machiraju, tech. report HPL-2001-310 (R.1), HP Laboratories, 2002).


➤ WSOL. Unlike the preceding company-related frameworks, the Web Services Offer Language (WSOL) is open source; it provides a rich set of reusability constructs and lightweight management infrastructure. (WSOL: A Language for the Formal Specification of Classes of Service for Web Service, Vladimir Tosić, Bernard Pagurek, and Kruti Patel, tech. report OCIECE-02-06, Dept. of Systems and Computer Eng., Carleton Univ., 2002).

Matchmaking
Most of the previous QoS specifications define their own language syntax and semantics, so they need a special engine for processing the syntax and its semantics.

The QML project targets a mapping of QML to Java and C++ so Java objects can represent QML specifications at runtime. However, QML’s mapping process and matchmaker are not yet clearly defined, to our knowledge.

WSLA has one proposed matchmaking algorithm that checks whether two SLOs are compatible (Compatibility Analysis of WSLA Service Level Objectives, Weilai Yang, Heiko Ludwig, and Asit Dan, tech. report RC22800 (W0305-082), IBM, 2003]. This algorithm uses a syntax tree to compare QoS requirements.

Another approach relies on an attribute constraint definition (An Internet-Based Negotiation Server for E-Commerce,” Stanley Y.W. Su and colleagues, VLDB J., Aug. 2001). It calculates all the combinations of the possible intervals for all the attributes to generate interval records. (The interval defines the upper and lower bound values of the particular attribute.) Matchmaking relies on these interval records to judge whether there’s a conflict between two constraint definitions. To our knowledge, previous work only judges whether two SLOs are compatible or not. Our matchmaking algorithm divides SLO compatibility into five degrees to allow for a more precise selection.
ever, requires a common ontology for each domain so that all partners in the same domain use a consistent terminology.

• Automation. Having various logical views of the system’s knowledge permits high-level automation, permitting the system, for example, to aggregate knowledge from various components. Matchmaking, validation, decision making, and so on are based on the logic in the collected knowledge rather than that in the hard-coded programs. Reasoners or rule engines help the system to achieve better automation.

• Extensibility. Different Web services have various requirements for their QoS descriptions. It’s relatively easy to add new requirements onto a knowledge base. When you introduce the new definition for a QoS property, you must also introduce its related metric’s definition to prevent the parser from raising logical warnings. The openness of ontology definition facilitates the sharing of experiences and quickens the development cycle. Similar to the QoS Modeling Language, our specification’s definition uses an object-oriented approach. Any programmer familiar with object-oriented design principles will find it easy to learn how to design and reuse our QoS specification. However, our specification does not address the problem of what actions to trigger at runtime if the QoS requirements are unsatisfiable. Other specifications, such as the QoS Description Language, are more expressive in this regard. The QoS measurement system based on an ontology layer is a new research effort, which will initially design the measurement specification and generate the measurement code base.

In the future, Semantic Web technology will help to integrate system components and tools into a unified system. As part of our future work, we would like to convert our current system into an OWL-based one. We also want to incorporate this QoS matchmaking framework into our previous QoS-aware service discovery framework (“QoS-Aware and Federated Enhancement for UDDI,” Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, Int’l J. Web Services Research, Apr. 2004, pp. 58-85). To maintain the current specification’s relatively simple syntax and short reasoning time, it does not use a rule-based language. Therefore, we do not yet define triggers and adaptation actions for unexpected events, such as the problem of unfulfilled QoS specifications mentioned earlier. Currently, the Semantic Web Rule Language is the one specification that combines an ontology (OWL) with the ability to write rules. This could be a nice solution for adaptation definition and requires future investigation.

Chen Zhou is a PhD student in the School of Computer Engineering, Nanyang Technological University, Singapore. Contact him at pg04878518@ntu.edu.sg.

Liang-Tien Chia is the director for the Centre of Multimedia and Network Technology and also an associate professor in School of Computer Engineering at Nanyang Technological University. Contact him at asltchia@ntu.edu.sg.

Bu-Sung Lee is an associate professor at the Nanyang Technological University and the vice dean of research for the School of Computer Engineering. He is also the technology area director of the Asia-Pacific Advance Network (APAN) and an associate with the Singapore Research and Education Networks (SingAREN). Contact him at ebslee@ntu.edu.sg.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.