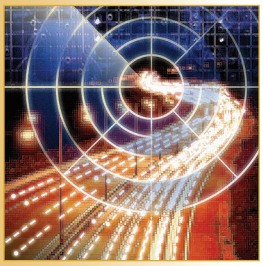

Editor: Michael N. Huhns • huhns@sc.edu

A Framework and Ontology for Dynamic Web Services Selection

Current Web services standards lack the means for expressing a service's nonfunctional attributes — namely, its quality of service. QoS can be objective (encompassing reliability, availability, and request-to-response time) or subjective (focusing on user experience). QoS attributes are key to dynamically selecting the services that best meet user needs. This article addresses dynamic service selection via an agent framework coupled with a QoS ontology. With this approach, participants can collaborate to determine each other's service quality and trustworthiness.

E. Michael Maximilien
IBM Corporation

Munindar P. Singh
North Carolina State University

A service-oriented architecture (SOA) promises the ready creation of applications composed of dynamically selected components. However, service selection also implies an established level of trust between these components: the consumer trusts the service to provide the necessary functionality as well as quality.

Current techniques for publishing and finding services (such as the Web Services Description Language [WSDL] and universal description, discovery, and integration [UDDI]) rely on static descriptions of service interfaces, forcing consumers to find and bind services at design time. Such techniques don't address runtime service selection based on a dynamic assessment of nonfunctional attributes, collectively known as quality of service. Matchmaking techniques based on Semantic Web technologies help fine-tune service interfaces and needs,[1] but such solutions currently ignore QoS and therefore apply only at design time. Service selection based on QoS is challenging: it can't readily be described via an interface because it depends on how, by whom, and where a given service is instantiated. Furthermore, consumers might have specific QoS profiles and requirements.

Dynamic service selection calls for an agent-based solution. Agents can represent autonomous service consumers and providers as well as collaborate to dynamically configure and reconfigure services-based software applications. Most importantly, agencies gather QoS data from agents, and store, aggregate, and present it to agents. Thus they enable agents to share QoS data about different

 Published by the IEEE Computer Society 1089-7801/04/$20.00

services: in principle, each QoS attribute of interest resides in its own agency.

Our approach implements this agent-based architecture and is realized in the Web Services Agent Framework (WSAF). WSAF incorporates service selection agents that use the QoS ontology (described herein) and an XML policy language that allows service consumers and providers to expose their quality preferences and advertisements.

System Architecture, Design, and Use

When a consumer application built with WSAF needs to use a service, it employs its agents to communicate with the service. For each service, WSAF creates a service agent that exposes the service's interface, augmented with functionality to capture the consumer's QoS preferences or policies and to query agencies or other agents for a suitable match. The agent can determine objective QoS-attribute values (such as reliability, availability, and request-to-response time) on its own and get user feedback for subjective attributes (such as the user's overall experience). It then conveys these QoS values to the appropriate agencies.

As Figure 1 shows, WSAF respects the classical SOA[2] but uses agents as broker implementations for service consumers and agencies.

To get a better understanding of our extensions to the classical SOA architecture, let's first take a high-level look at WSAF's main components, followed by a detailed analysis of the typical system usage:

- *Service providers* describe each service via WSDL.
- *Service brokers* augment UDDI broker registries with agencies in which service agents can collaborate and share data.
- *WSAF servers* host service agents, QoS ontologies, configuration details, and host agencies.
- *Consumer applications* contain business objects as well as proxy objects, which act as local proxies to service agents.

Figure 2 (next page) incorporates a UML sequence diagram to illustrate a typical consumer-to-agent interaction and control flow:

- Upon initialization, WSAF sets up all configured agencies (steps 1 and 2).
- Providers register service implementations with WSAF by configuring each service in terms of WSDL URIs, service domains, and the service's advertised QoS policy (steps 3 and 4). Each configured service interface has an agent.
- The consumer application creates a local proxy object for the service agent; the consumer invokes the proxy with its policy (steps 5 to 6).
- The agent uses the policy and its configuration to load and run its script. The script typically consults the QoS and service ontologies to complete its configuration (steps 7 to 7.3). This setup occurs once per consumer-to-agent interaction episode.
- By default, the agent performs a binding operation once configured (steps 8 to 8.3). Consumers can initiate a rebinding or specify an automatic rebinding initiation in their policies. The agent selects a service implementation based on agency data, and then dynamically creates a proxy object for each selected service.
- The consumer invokes the agent's service operations (steps 9 to 13). Each invocation is forwarded to the service proxy, while being monitored by the agent; when the service responds, the agent inserts appropriate data to the relevant agencies.

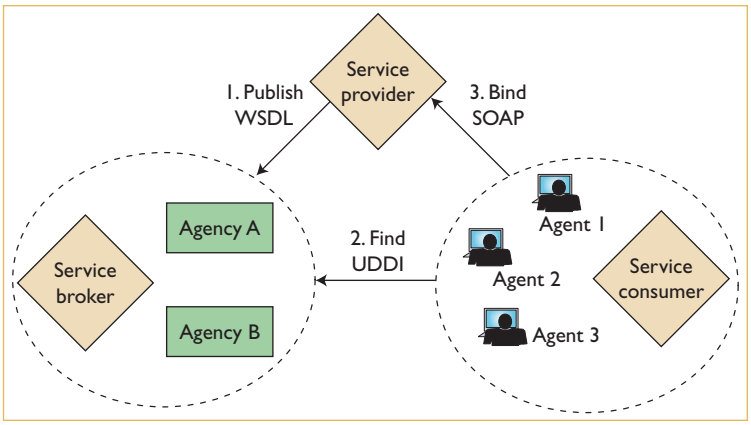

Figure 1. Agents and agencies in a service-oriented architecture. In classic SOAs, service providers publish to registries (and agencies, in our case), and service consumers query the agencies and then select and bind to a service provider. Because agents and agencies reside in a known application server, agents don't impose any additional computational burden on the consumer's resources.

Because WSAF is a framework, it provides basic structures for agents and agencies that can be extended for various purposes in an SOA solution.

Agent and Agency Design

WSAF agents are autonomous: they can participate in agencies, behave reactively, and serve as mediators between a given application and the

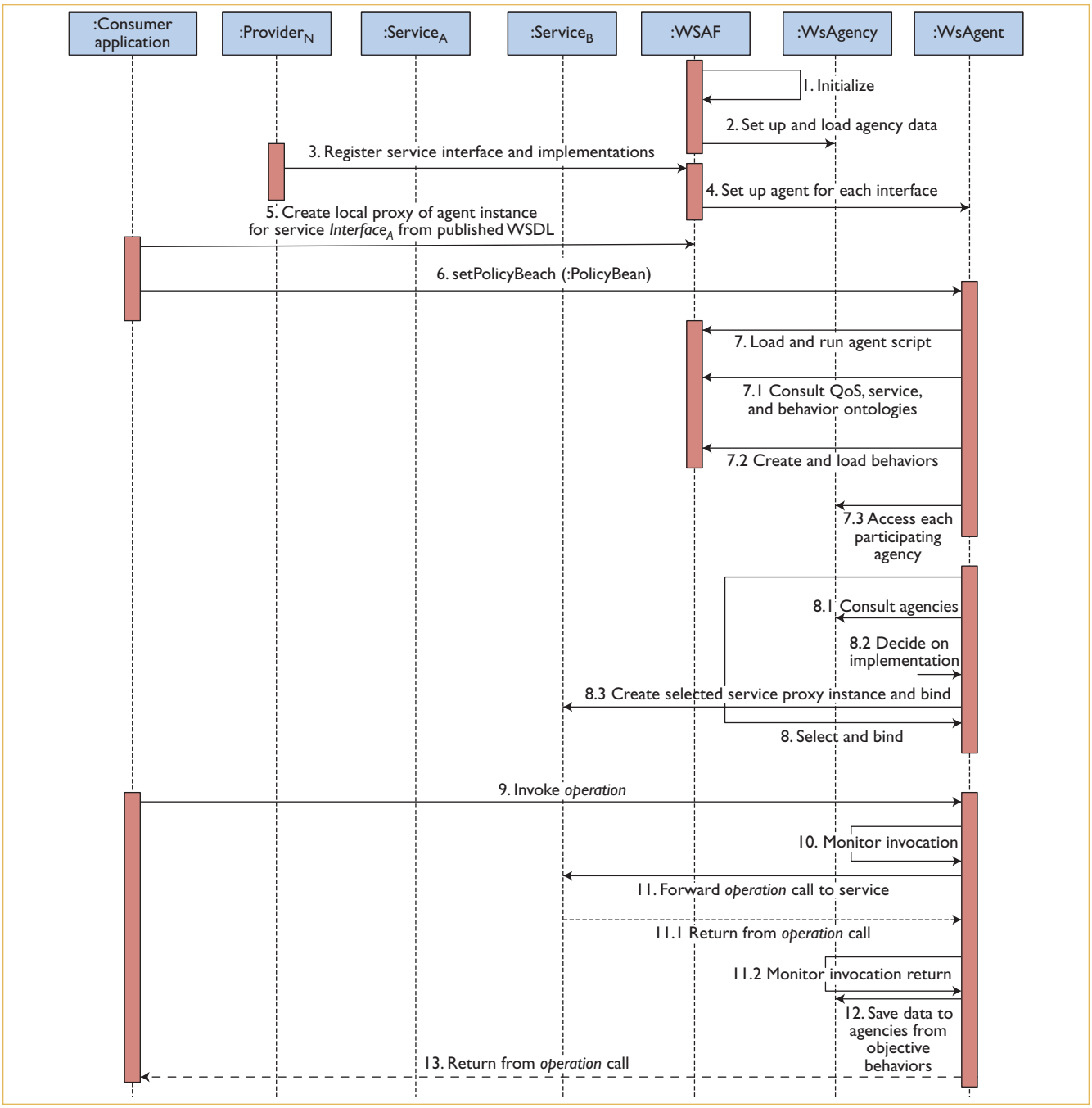

Figure 2. Typical agent usage control flow. Using a UML sequence diagram, we see the different sequence of actions among the main actors (the consumer application, service agents, service providers, and our framework).

implementations of the consumed services. WSAF agents also provide methods that let consumers set their QoS preferences and rank services.

As Figure 3 shows, these agents consist of several components:

- *Input ontologies* (the shared conceptualizations among agents, consumers, and providers). The agent-behavior ontology specifies the behaviors with which an agent is configured. An agent that participates in a performance agency, for instance, uses the defined behaviors for collecting performance QoS data.
- *Agency data*. An agent can query or insert data into each agency in which it participates.
- *Augmented service interface*. A proxy agent

exposes the service's interface but is augmented with agent-specific methods.
- *Scripting engine.* A script specifies an agent's behavior. Jython (Python in Java; see www.jython.org), for example, supports rapid prototyping and object-oriented paradigms.

Specifically, an application uses the augmented interface to specify its QoS preferences (used by the agent to select services) prior to using the service's methods. The application can then interact with the agent to select appropriate services. The application uses the agent interface to provide feedback on QoS attributes that apply to its use of the service. This feedback can be explicit (the consumer fills out a form in consultation with the human user) or implicit (the agent infers the consumer's rating based on heuristics such as repeated usage).

Figure 4 (next page) illustrates a WSAF agent's key interfaces:

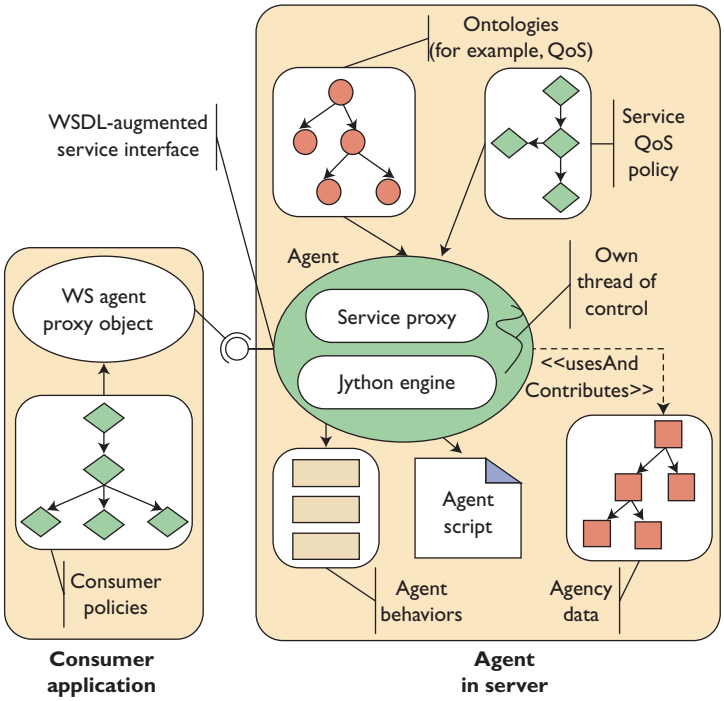

Figure 3. Agent design. The input ontologies conceptualize the knowledge needed for the agent's tasks; the agency data is the structured data about QoS shared among the agents; and the scripting engine provides a rapid-prototyping programming environment.

- `Ws` is a surrogate for the available services.
- `WsAgentConfig` is used to configure an agent at deployment or during runtime.
- `WsAgent` is the primary interface to a WSAF agent. This interface aggregates the other objects and acts as a proxy for the service. The agent dynamically generates a proxy object with the same interface as the service, intercepts all method calls, and then forwards them to the `ServiceInterceptors` prior to forwarding calls to the service proxy object. This interception lets the agent monitor the service, add behaviors to service methods, and dynamically select new service instances.
- `WsAgency` provides a facade to the data shared between `WsAgents`. Agencies enable the persistence of QoS information.
- `ServiceInterceptor` provides a generic way for `WsAgent` to intercept a consumer's method call to a service.
- `AgentBehavior` extends `ServiceInterceptor`; `ServiceInterceptors` are notified of all service method calls, before and after invocations. By virtue of being part of an agent, a behavior can contribute data to the agencies in which the agent participates.
- `AgentScript` abstracts the agent's programs. For a service selection agent, for example, the script contains the selection algorithm. It provides the lifecycle methods called for the script along with the agent lifecycle methods, including `init()` and `dispose()`. The abstraction enables the agent to be implemented dynamically.

Any WSAF agent can participate in an agency as long as it respects the agency's database schema. Figure 5 (next page) illustrates an agency's key interfaces. The interface `WsAgencyPopulator` abstracts methods that enable flexible population of agencies; WSAF implements this interface via Apache's Xindice XML database (http://xml.apache.org/xindice/).

Knowledge Representation

So far, we've shown how our framework uses agents and agencies to address the challenges of dynamic service selection in a manner that respects QoS. But to realize such agents and agencies in a principled manner presupposes that we have rich knowledge representations for services and qualities. Such representations would help us capture the most important requirements and engineer agents and agencies that behave as desired. These representations are the service and

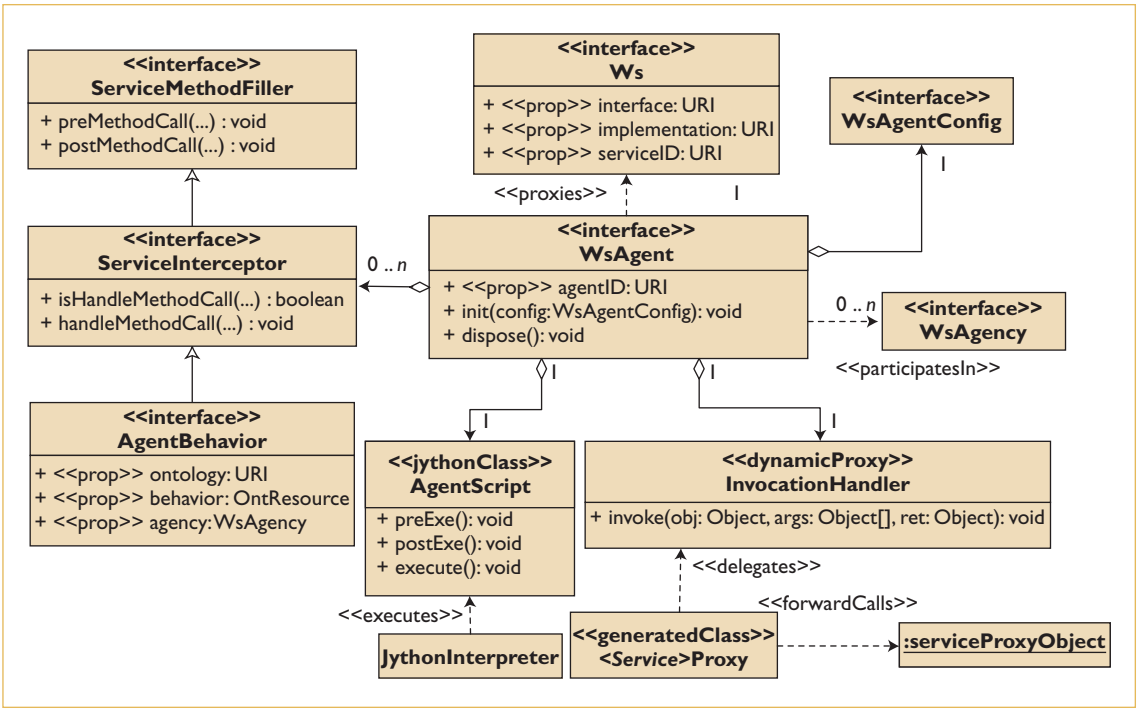

Figure 4. Agent UML static class diagram. This diagram shows an agent's main interfaces and classes. By emphasizing interfaces, we allow different implementations of the framework to be provided.

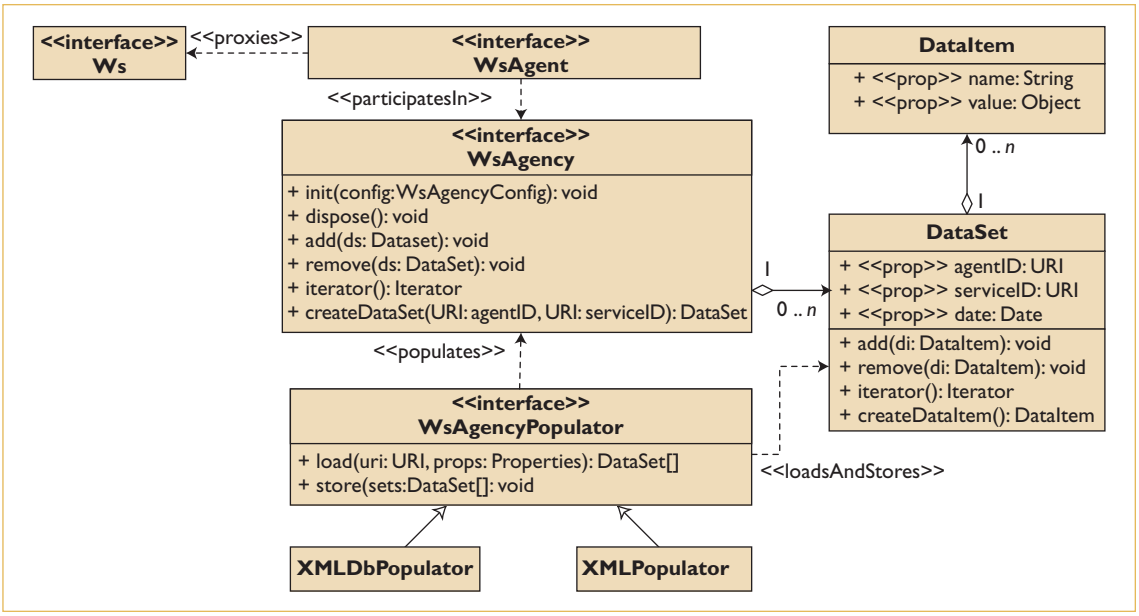

Figure 5. Agency design. At its core, an agency is a collection of agency `DataSets`*, which in turn maintain a collection of* `DataItems`*, which can be composed of additional* `DataItems`*. Each* `DataSet` *is associated with the agent and service that it proxies.*

QoS ontologies, respectively. The service ontology relates services to QoS whereas the QoS ontology nails down the quality concepts. Let's look at such a representation's main components.

Service Ontology

Figure 6 illustrates our service ontology. Each service is associated with a service domain and has an interface and potentially many implementa-

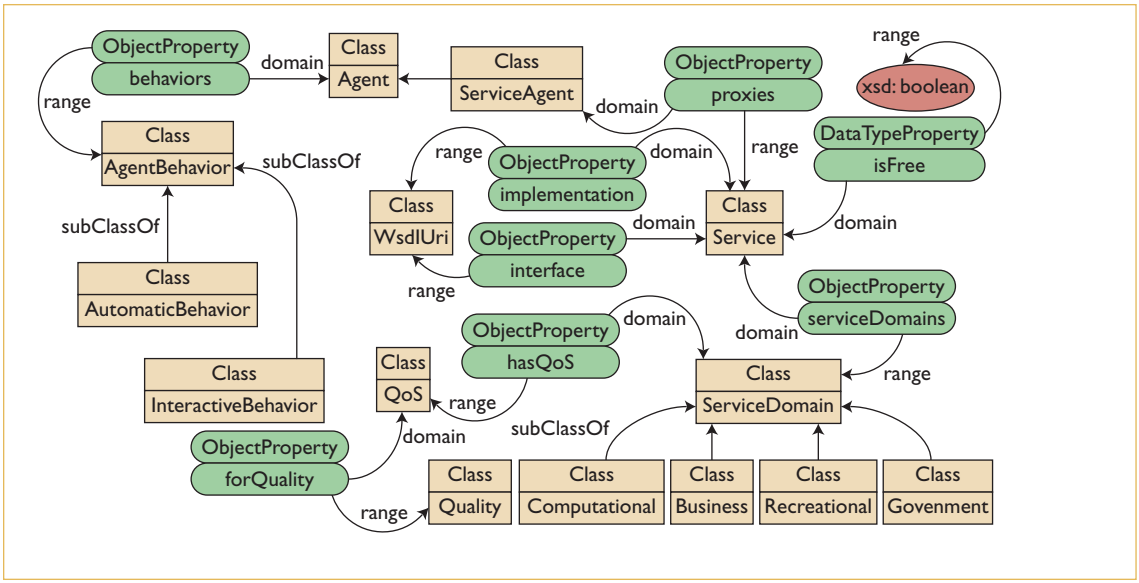

Figure 6. Service ontology. The service ontology classifies services into domains and associates qualities with domains. We can then capture the agents that make use of services (service agents) and their behaviors.

tions. A `ServiceDomain` aggregates services that have common qualities.

QoS Ontology

Our QoS ontology lets service agents match advertised quality levels for its consumers with specific QoS preferences. Providers express policies and consumers express preferences using the QoS ontology, which also enables the consumers to configure service proxy agents so that they have the necessary behaviors to monitor and record consumer and service interactions. It helps to distinguish three ontologies for QoS: upper, middle, and lower.

Figure 7 (next page) reviews the key aspects of our QoS upper ontology. The upper ontology captures the most generic quality concepts and defines the basic concepts associated with a quality, such as quality measurement and relationships:

- `Quality` represents a measurable nonfunctional aspect of a service within a given domain. Quality attributes relate to each other. Figure 8 (next page) describes our middle ontology with specific quality concepts. The middle ontology differs from upper and lower ontologies in that it captures the quality concepts that are applicable to multiple domains (such as performance qualities).
- `QAttribute` captures a given quality's type — for example, whether it's a monotonic float attribute (a quality whose value is a floating-point number and increases in the value reflect improvements in the quality).
- `QMeasurement` measures a `Quality` objectively or subjectively. Agents make objective measurements automatically, whereas subjective measurements involve humans. A measurement has a validity period and can be certified.
- `QRelationship` describes how qualities are correlated. Service response time, for example, could be negatively correlated to throughput. Such quality relationships often reflect the trade-offs providers make in their service implementations. Qualities are potentially related in terms of *direction* (opposite, parallel, independent, or unknown) and *strength* (such as weak, mild, strong, or none).
- `AggregateQuality` is a quality composed from other qualities. The price–performance ratio, for instance, aggregates price and performance.

The QoS middle ontology incorporates several quality aspects encountered in distributed systems.[3–5] Figure 8 (next page) defines the middle ontology for Web services QoS:

- *Availability* is the probability that a service can respond to consumer requests. It has two subclasses: `MTTR` (mean time to repair, meaning the average time for restoring a failed service) and `UpTime` (the duration for which the service has been operational continuously without failure). Availability is mildly parallel to reliability and

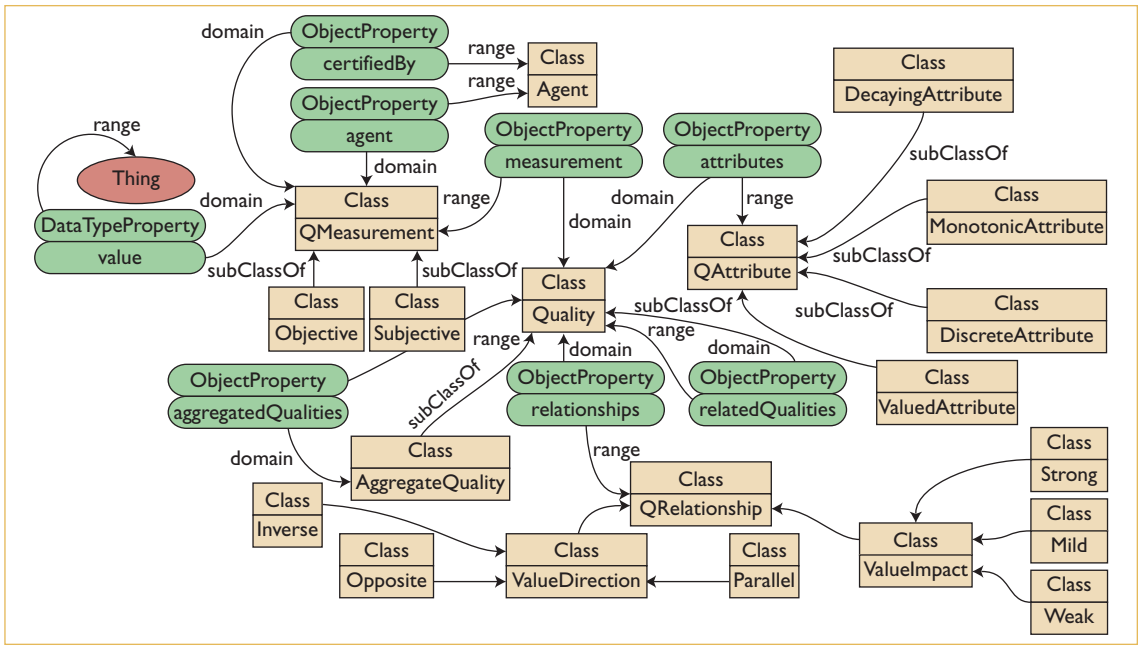

Figure 7. QoS upper ontology. This ontology includes the basic characteristics of all qualities and the main concepts associated with them.

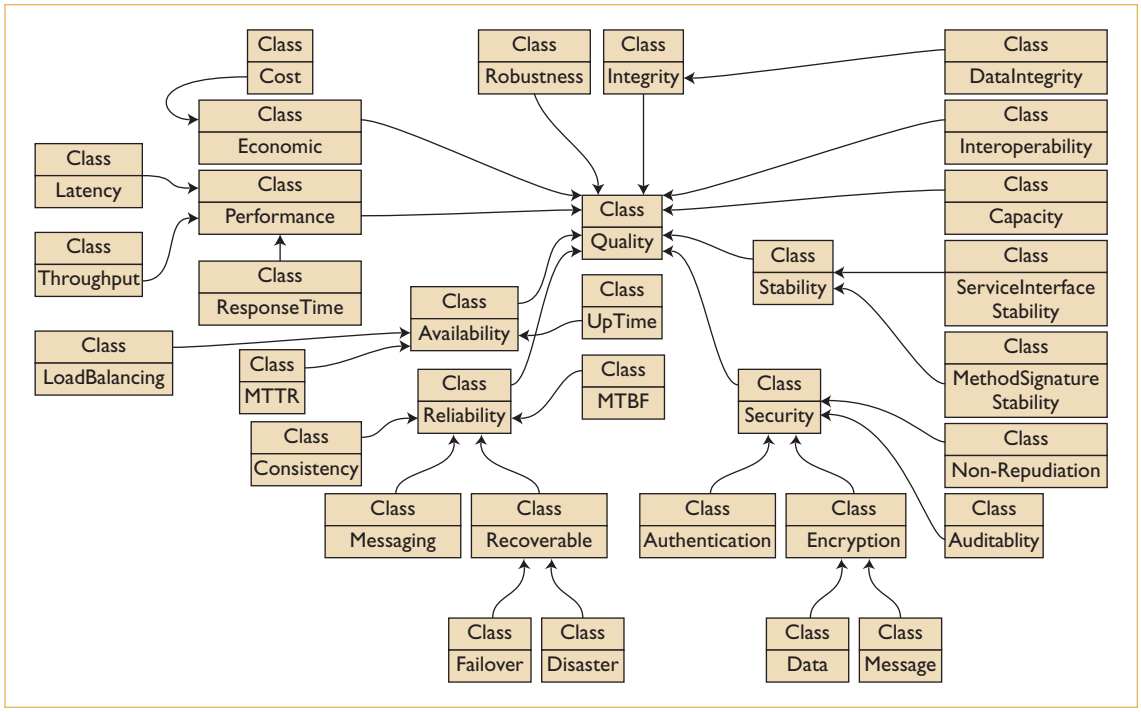

Figure 8. QoS middle ontology (arrows indicate `subClassOf`*). This ontology specifies domain-independent quality concepts and is typically completed by a domain-specific lower ontology.*

typically mildly opposite to capacity.

- *Capacity* is the limit on the number of requests a service can handle. When a service is operated beyond its capacity, its availability and reliability are negatively affected.
- *Economic* captures the economic conditions of using the service. Usage cost is a key economic attribute.

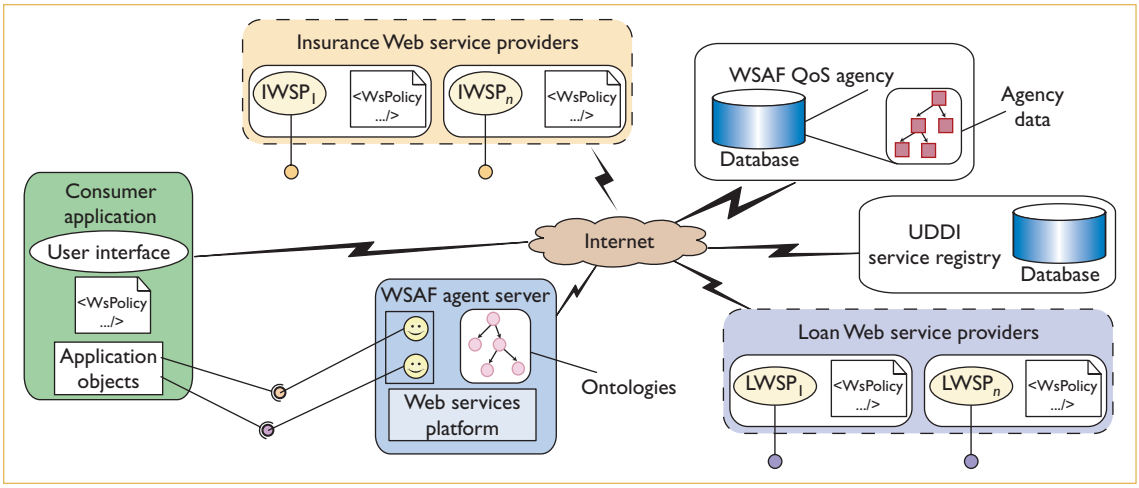

Figure 9. Components for the insurance and loan example. The providers are shown in two parts (insurance and loan services), and agencies are shown as the UDDI registry and the WSAF QoS agency. An alternative topology is to deploy the agents in the consumer application.

- *Interoperability* is the ease with which a consumer application or agent interoperates with a service. It defines, for example, whether the service is compliant with a specified standard, such as the WS-I Basic Profile,[6] or specific versions of standards like WSDL.
- *Performance* characterizes performance from the consumer's perspective. Examples are `Throughput` (the rate of successful service-request completion) and `ResponseTime` (the delay from the request to getting a response from the service).
- *Reliability* is the likelihood of successfully using a service. Typically, it parallels availability, but its main aspects also include `Fault Rate` (the rate of invocation failure for the service's methods); `MTBF` (mean time between failures); `Consistency` (the failure rate's lack of variability); `Recoverability` (how well the service recovers from failures); `Failover` (whether the service employs failover resources, and how quickly); and `Disaster` resilience (how well the service resists natural and human-made disasters).
- *Robustness* is resilience to ill-formed input and incorrect invocation sequences.
- *Scalability* defines whether the service capacity can increase as needed.
- *Security* captures the level and kind of security a service provides. Its key components include `Auditability` (the service maintains auditable logs); `Authentication` (the service either requires user authentication or accepts anonymous users); `Encryption` (the type and strength of encryption technology used for storage and messaging); and `NonRepudiation` (whether consumers can deny having used the service).
- *Integrity* is a measure of the service's ability to prevent unauthorized access and preserve its data's integrity.
- *Stability* is the rate of change of the service's attributes, such as its service interface and method signatures.

Let's now apply the above framework and ontology using a realistic example.

Comprehensive Example

We can demonstrate our approach with a comprehensive scenario that involves a consumer shopping for financing and insurance to buy a car. Standardized Web services interfaces for insurance and loan providers already exist, as do lower QoS ontologies for the insurance and loan domains. Figure 9 shows our approach's main components as applied here.

The insurance QoS ontology contains the concepts of `PremiumPrice` (subclass of `Price`) and `Deduction` (specific to the insurance domain). Similarly, the loan QoS ontology includes the domain-specific concept `InterestRate`. Naturally, both domains also use concepts such as security and performance, as in the middle ontology illustrated in Figure 8.

Figure 9 shows a consumer application using two service agents to select the best available insurance and loan service implementations. The

following is a snippet of the advertised policy for an insurance provider:

```
<WsPolicy name=
   ”insuranceServiceProvider1” …>
  <Services><Service name=”isp1”
     interfaceUri
=”http://.../insurance?wsdl”/>
   … </Services>
   <Ontologies>
   <Ontology name=”InsuranceOnt”
     uri=”http://.../insurance-
qos.owl”/>
   … </Ontologies>
  <QoSPolicy ontologyName=”InsuranceOnt”
     serviceNames=”isp1” methods=”*”>
   <QoS name=”PremiumPrice”><qValue>
     <unit>USD</unit><min>2000</min>
<max>6000</max><typical>4000</typical>
     </qValue></QoS>
   <QoS name=”Deduction”><qValue>
     <unit>USD</unit><min>50</min>
   <max>1000</max><typical>500</typical>
     </qValue></QoS>
   </QoSPolicy>
</WsPolicy>
```

The loan service providers specify policy advertisements similarly. The consumer's QoS needs are as follows:

```
<WsPolicy name=”consumer” …>
   <Services><Service name
      =”insuranceService”
   interfaceUri=”http://.../insurance?
      wsdl”/>
<Service name=”loanService”
   interfaceUri=”http://.../loan?wsdl”/>
   … </Services>
<Ontologies><Ontology name=”InsuranceOnt”
   uri=”http://.../insurance-qos.owl”/>
<Ontology name=”PerfOnt”
   uri=”http://.../perf-qos.owl”/>
<Ontology name=”BindingOnt”
   uri=”http://.../wsaf.owl”/>
      </Ontologies>
<BindingPolicy ontologyName
      =”BindingOnt” services=”*”>
    <Bind when=”onConnect”
       type=”bestMatch”/>
    … </BindingPolicy>
<QoSPolicy ontologyName=”LoanOnt”
   serviceNames=”loanService”>
   <QoS name=”InterestRate”><qValue>
   <unit>percent</unit><max>5</max>
     <preferred>2.5</preferred>
   </qValue></QoS>
… </QoSPolicy>
<QoSPolicy ontologyName=”PerfOnt”
      serviceNames=”*”>
   <QoS name=”Availability”><qValue>
     <unit>percent</unit><min>98</min>
     <preferred>100</preferred>
     </qValue></QoS>
… </QoSPolicy>
   …
</WsPolicy>
```

Some of the consumer's needs in this code snippet refer to subjective qualities (such as `PremiumPrice`); others refer to objective qualities (such as `Service Availability`).

The service agent finds services matching the given interface in the traditional manner (using UDDI), and then applies the consumer's policy on the available quality data to rank the service implementations. The ranking is computed from the quality–degree match, which is based on what the provider advertises along with the provider's reputation for the given quality, and how the quality in question relates to other needed qualities. Because the insurance premium price is opposite of the deductible, for example, the agent would adjust the degree match to trade off the premium with the deductible if the consumer wanted a low premium and low deductible. The match for a service implementation is an aggregation of the matches for the applicable qualities.

Discussion

To evaluate WSAF and its QoS ontology, we built a simulation testbed on top of WSAF and evaluated our approach with scenarios involving consumers with different policies and services with different QoS advertisements. By artificially controlling the services qualities (such as accuracy, response time, reliability, and availability), we could verify whether each consumer selected the right services (given its preferences). The preliminary results are promising and suggest that this approach can support dynamic trust establishment. In ongoing work, we are attempting to show how it achieves self-adjusting trust in the sense of autonomic computing.[7] As a service QoS drops below its consumer's QoS requirements, the agent will no longer select it; when the service's qualities improve, it

will again be considered for selection.

An important technical direction is to select services in the face of multiple qualities (which might be mutually independent or dependent). We're developing an approach based on multiattribute utility theory, which accommodates the relevant relationships, both statistical (as inferred from agency data) and qualitative (as given by a conceptual modeler).

Of the other work in this area, OWL-S is the most relevant (see www.daml.org/services/owl-s/1.0/). The OWL-S service ontology captures the service profile as well as the service-process model. Our approach complements OWL-S by emphasizing the quality aspects.

An area of future research opened by our approach is how to prevent malfeasant agents from biasing the agencies' data. Current security infrastructures are not sufficient because they're vulnerable to attacks such as spoofing. Solutions to combat this problem could include those based on reputation and social networks.[8]

References

1. K. Sycara et al., "Automated Discovery, Interaction and Composition of Semantic Web Services," *J. Web Semantics*, vol. 1, no. 1, 2003, pp. 27–46.
2. *Web Services Conceptual Architecture* (WSCA 1.0), IBM Corp. specification, 2001; http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf.
3. S. Ran, "A Model for Web Services Discovery with QoS," *SIGEcom Exchanges,* vol. 4, no. 1, 2004, pp. 1–10.
4. B. Sabata et al., "Taxonomy for QoS Specifications," *Workshop on Object-Oriented Real-Time Dependable Systems* (WORDS '97), IEEE CS Press, 1997.
5. K.-C. Lee et al., "QoS for Web Services: Requirements and Possible Approaches," World Wide Web Consortium (W3C) note, Nov. 2003; www.w3c.or.kr/kr-office/TR/2003/ws-qos/.
6. K. Ballinger et al., *WS-I Basic Profile Version 1.0a*, Web Services Interoperability Org., 2003; http://www.ws-i.org/Profiles/.
7. J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003, pp. 41–50.
8. B. Yu and M.P. Singh, "An Evidential Model of Distributed Reputation Management," *Proc. 1st Int'l Joint Conf. Autonomous Agents and Multiagent Systems,* ACM Press, 2002, pp. 294–301.

E. Michael Maximilien is a PhD candidate in computer science at North Carolina State University, Raleigh, where he received an MS in computer science. His research interests include Semantic Web services, multiagent systems, and software engineering. Maximilien also serves as an advisory software architect for IBM. He is a member of the IEEE and the ACM. Contact him at maxim@us.ibm.com.

Munindar P. Singh is a full professor of computer science at North Carolina State University. His research interests include multiagent systems and Web services – specifically, the challenges of trust, service discovery, and business processes and protocols in large-scale open environments. Contact him at singh@ncsu.edu.