Disambiguating Availability Specification through the use of OWL

Glen Dobson
Computing Department
Lancaster University
g.dobson@comp.lancs.ac.uk

Ian Sommerville
School of Computer Science
St Andrews University
ifs@dcs.st-and.ac.uk

Abstract

Many Quality of Service (QoS) languages exist. However, not only do few encompass dependability, none acknowledge the semantic complexity of the vocabulary they provide. This paper presents a Quality of Service ontology which provides not only an extensible syntax for expressing dependability, but also rich, well-defined semantics. These semantics avoid ambiguity and misunderstanding as well as facilitating translation where possible. To demonstrate these features, this paper examines in depth how to use our QoS ontology along with the built-in capabilities of the Web Ontology Language (OWL) to capture the semantics of availability.

1. Introduction

In service-based systems there are a number of stakeholders who wish to express, record and communicate about Quality of Service (QoS):

- The service provider may wish to advertise their service QoS to potential customers or make use of QoS measurements internally for quality assessment and planning.
- The service integrator may wish to express their QoS requirements. These may be applied at the traditional requirements stage of a development process, or, due to the possibility of late binding, may be applied at runtime. Equally the integrator may wish to advertise the QoS of their application to its potential users.
- The end user may also wish to express their QoS requirements.
- Where some guarantee of required QoS levels being met is desired, an integrator or end user may wish to negotiate a Service Level Agreement (SLA) with the provider. A chain may even be formed where an integrator's SLA with their customer depends upon the SLAs that they have agreed with one or more providers.

As well as these, many other communications regarding QoS may need to be performed such as to initiate and perform measurements, to signal breach of an SLA, to discover or differentiate services, etc. All of these tasks require a concrete means of expressing aspects of QoS.

Many languages have been developed which provide a concrete syntax for exactly these purposes (e.g [1], [2], [3], [4]). However, such languages often have a focus on performance and efficiency and therefore fail to encompass the full spectrum of non-functional properties which may be used to judge service quality. In particular, few allow the specification of service dependability – with most having a general bias towards network performance characteristics. Moreover, in this paper we demonstrate that the semantics of service dependability are complex enough that, in many cases, a concrete syntax is insufficient to successfully perform the tasks listed above. Therefore, extending an existing language may not be a sufficiently powerful solution to fill this gap.

To focus the discussion we concentrate on service availability in particular. The solution we present makes use of the Web Ontology Language (OWL) to provide a QoS ontology, i.e. an extensible vocabulary for expressing QoS (including dependability) with rich, well-defined semantics. We demonstrate the advantages of this solution in terms of accurately specifying availability metrics and interrelating them. We also consider how the machine interpretable nature of OWL makes tasks such as metric conversion and requirements matching possible.

The paper is structured as follows: in the following section we look at existing work on QoS specification languages. In Section 3 we move on to discuss OWL in order to set the scene for explaining our QoS ontology. The core of this ontology, which we call QoSOnt, is the subject of Section 4. Section 5 contains the crux of our argument - demonstrating the complexity of expressing availability and how our ontology aids in this. In Section 6 we draw conclusions and suggest future directions for this work.

2. QoS Specification Languages

There is a wide range of research which touches upon Quality of Service specification. At one end of the spectrum languages suitable for specifying the requirements of multimedia applications regard QoS as concerned largely with network performance and synchronisation issues. At this level, protocols such as IntServ (Integrated Services) [5], DiffServ (Differentiated Services) [6] and RSVP (Resource Reservation Protocol) [7] allow packets in a flow to be differentiated and prioritised allowing an IP network to offer levels of service beyond best-effort. These protocols essentially provide a means to specify required bandwidth, latency and jitter.

On top of these, languages such as XQoS [1] and QuAL [2] allow the specification of the QoS requirements of a multimedia application and encompass synchronisation of streams. These application-level specifications can be mapped onto the underlying QoS protocols mentioned above with the aim of ensuring that the network meets the application's requirements.

In the case of these QoS languages the network is seen as providing the service of which the quality is being judged. However, for service-based applications, the services being consumed are software services. The quality issues are therefore much wider, including performance in a broader sense and service dependability.

QML (Quality Modelling Language) [3] is a language which allows QoS specification in this broader sense, although it is not designed with services in mind specifically – but as a general-purpose means of describing the QoS properties of software components. The basic element in a QML specification is known as a *contract*. Each *contract* is of some specified *contract type*. The *contract type* specifies the *dimensions* that can be used to specify QoS properties within some category (e.g performance, availability, security, timing). The simplified section of QML below is an example defining a Dependability *contract type* with a single *dimension* (availability), which is a numeric value where an increasing value indicates increasing quality. A *contract* of this type is then specified (systemAvailability) stating that availability > 0.9 is required.

```
type Dependability = contract {
        availability : increasing numeric;
};

systemAvailability = Dependability contract {
        availability > 0.9;
};
```

In practice the contract is often bound to a software interface, operation, operation argument or operation result using the language element known as a *profile*. This example shows however, that despite being generic and extensible enough to encompass availability, it is only really possible to specify it in a very shallow way – basically as a bound on a number about which you know nothing other than its value (See Section 5 for more discussion of this).

QML also attempts to rigorously define its semantics and therefore has much in common with our QoS Ontology. However, it perhaps leaves too much to be specified by a user, whilst not giving them enough power to specify everything they should. We seek to provide foundation layers providing common conceptual building blocks to minimise user effort, whilst maximising expressivity. Also, by seeking to make use of semantic web technologies we enable much closer integration with the web services for which QoS is being specified.

On this note it is worth questioning what QoS specification technologies exist in the field of web services. WSLA [4] is an XML-based specification language for (Web) Service Level Agreements (SLAs). A section of WSLA might look like:

```
<Expression>
  <Predicate xsi:type="Less">
    <SLAParameter>Availability</SLAParameter>
    <Value>0.9</Value>
  </Predicate>
</Expression>
```

Again, this expression could be tied to a particular operation (through the service's WSDL). A great deal of other detail is also missing from this snippet. For instance, for each QoS parameter a metric is defined and for each metric a Measurement Directive is specified. This is a step in the right direction in terms of providing sufficient detail in the specification of a given attribute (compared to, e.g. QML). However WSLA only applies to SLA specification and therefore cannot be easily applied to the other tasks mentioned in the introduction. It also remains difficult to interrelate metrics.

In this section we noted the need for a QoS specification language with richer semantics than existing ones provide. To create something like this, a descriptive meta-language with its own formal semantics is highly desirable. In the next section we discuss a language which is exactly this: the Web Ontology Language (OWL).

3. OWL

An ontology is a machine interpretable description of the terms which exist in some domain and the relationships between them. The aim of an ontology is to give machines some depth of domain "understanding" beyond the syntactic. The degree to which this is achieved depends upon success in eliciting domain knowledge and representing it in some formal way. Numerous ontology languages have developed to facilitate such formal description, of which OWL is a relatively new example. Its emphasis is on providing a way of distributing and sharing ontologies via the web.

To this end, OWL is built upon RDF (the Resource Description Framework) [8]. RDF is an XML (eXtensible Markup Language) [9] vocabulary for describing resources on the web, and as such has certain commonalities with ontology languages. Statements about resources in RDF are expressed using triples, which consist of:

- The resource being described (the subject)
- The specific property (the predicate)
- The value of that property for that resource (the object)

RDF vocabularies can be specified using RDF Schema (RDFS). The chief concepts introduced for this purpose are Classes (i.e. the types of things which can be described) and their Properties. OWL builds upon RDF and RDFS to add a greater level of expressivity and machine interpretability as well as providing formal semantics for the language. It gains from RDF the ability to distribute an ontology across many systems (as resources are identified and accessed by URI).

An OWL ontology consists of Classes and their Properties. Instances of OWL Classes are called Individuals. OWL Individuals are very much like resources described using RDF although they may have further OWL-specific *facts* expressed about them - namely:

- That the URI indicated and some other URI actually refer to the same Individual (the OWL sameAs construct)
- That the URI indicated and some other URI do not refer to the same Individual (the OWL DifferentFrom construct)
- That each URI in a specified list refers to a different Individual (the OWL AllDifferent construct)

An OWL Class is a specialisation of RDFS Class, which can be specified in new ways beyond simply stating its name. These added means of class description are:

- Enumeration of all Class members (i.e. OWL Individuals) using the OWL oneOf construct.
- As an anonymous Class of all Individuals that satisfy a Property restriction using the various OWL value and cardinality constraint constructs: allValuesFrom ($\forall$), someValuesFrom ($\exists$), hasValue, maxCardinality, minCardinality, cardinality
- By combining existing Classes using set operators (the OWL intersectionOf ($\sqcap$), unionOf ($\sqcup$), and complementOf ($\neg$) constructs)

These Class *descriptions* can be nested to create arbitrarily complex new *descriptions*. *Descriptions* can then be combined into a Class *definition* using the OWL subClassOf ($\sqsubseteq$), equivalentClass ($\equiv$) and disjointWith constructs. The Class *definition* specifies all of the necessary and/or sufficient conditions for Individuals to be members of a Class. A Class can therefore be viewed as defining a set of individuals (the *class extension*).

A key feature of OWL is that it takes the form of a Description Logic (DL) [10] and therefore has formally stated semantics. A Description Logic is a logic that focuses on concept descriptions as a means of knowledge representation and has semantics which can be translated to first-order predicate logic. Some of the DL notations for various OWL constructs are shown in parentheses above. The nature of DLs means that classification, subsumption and satisfiability can be automatically computed by a reasoner. OWL can be seen as a trade-off between expressivity and decidability. In this context decidable means that inference algorithms exists for the language and are known to terminate.

In DL reasoning, an open world assumption is made. This means that no assumptions are made about anything which is not asserted explicitly. This contrasts with the closed world assumption used in data modeling, where anything unstated is taken to be false.

4. The QoS Ontology

Our QoS Ontology (QoSOnt) [11] should be regarded as an "upper ontology" for QoS. That is, its aim is to provide the definition of the general concepts of QoS without reference to any particular domain. These general concepts can then act as a common foundation for describing particular QoS attributes, metrics, etc. (i.e. for building one's own QoS vocabulary). QoSOnt was developed by a process of examining existing QoS specification languages [12] and represents many of the commonalities found.

From Section 3 one can see that an OWL ontology consists of a group of Class (or concept) descriptions which are reducible to predicate logic; a set of combinatorial statements defining new Classes from existing Classes (i.e. set operations); and a Class hierarchy (or taxonomy). The asserted taxonomy is formed from the use of the subClass construct explicitly stated in the OWL, however a reasoner might infer a different hierarchy based upon the properties of a given Class. What this means is that, as long as you have stated the relevant properties the reasoner can be left to deal with classification where necessary.

The core of QoSOnt, from a taxonomical point of view, consists of two hierarchies – one rooted at the Class QoSAttribute and one rooted at the Class QoSMetric (See Figure 1). Many actual attributes would populate a complete hierarchy. Here we only show Availability as it is the focus of this paper. The same is true of metrics. Here, we show DowntimeHistory as an example.

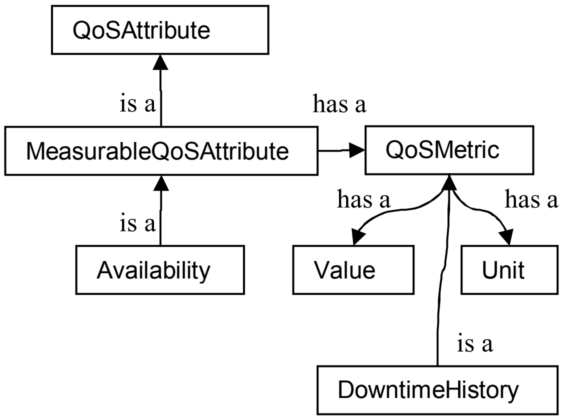

Figure 1. Core QoSOnt Classes

These two hierarchies are joined together by an OWL ObjectProperty (a binary relation between Classes) named hasMetric. The subclass MeasurableQoSAttribute of QoSAttribute is specifically defined in terms of this property as MeasurableQoSAttribute $\equiv$ QoSAttribute $\sqcap$ $\forall$hasMetric.QoSMetric $\sqcap$ $\exists$hasMetric.QoSMetric (remembering that the $\exists$, and $\forall$ restrictions essentially define anonymous Classes in OWL). In OWL RDF/XML syntax this looks like:

```
<owl:Class rdf:ID="MeasurableQoSAttribute">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#QoSAttribute"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#hasMetric"/>
          <owl:allValuesFrom rdf:resource="#QoSMetric"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#hasMetric"/>
          <owl:someValuesFrom rdf:resource="#QoSMetric"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

This gives some indication that, for human-to-human communication of OWL, the DL notation is more succinct and readable (once understood). Therefore no more RDF/XML shall be presented here. This example also shows that semantic rigour comes at the cost of brevity and simplicity.

We highlight the MeasurableQoSAttribute in particular as for most applications it is important that there is an unambiguous way of measuring a specified QoS attribute. The aim of the above definition is to state that a MeasurableQoSAttribute is any QoSAttribute where all metrics are QoSMetrics and there is at least one of these (i.e. there exists at least one metric which is a QoSMetric). If not constrained any ObjectProperty in OWL can have zero or more values in any given Individual.

A QoSMetric is essentially defined as having a numerical value, or alternatively a QoSMetricComplexValue. The latter allows for arbitrarily complex values to be defined using OWL. For instance the example metric in Figure 1 is intended to model the case when a provider may simply advertise their complete history of downtime and no other metric. This would require the use of a QoSMetricComplexValue rather than a single number.

This situation is depicted in a simplified form in Figure 1 by the box "Value". Where a simple value is used, a unit can be associated with it. It is useful, e.g. if you are stating Mean Time to Repair to know what units of time you are using. In fact we define our own time ontology since relating concepts of time is such a pervasive issue in QoS. An acceptability direction (equivalent to increasing/decreasing in QML) can also be associated with a QoSMetric Class, which would indicate to somebody unfamiliar with the metric whether a higher or lower value was better. The issue of non-linear variance of quality where a "sweet spot" may exist is not yet handled.

In the original version of the ontology we also have ConversionRate classes which can be used for conversion purposes when specifications are stated in different units. However, these rely on associating some external meaning with the Class in question, which is not the aim of an ontology. As an alternative, we have found that the Semantic Web Rules Language (SWRL) [13] allows us to explicitly state the semantics of conversion, as it adds built-in arithmetic functions as well as implication.

On top of QoSOnt we have also created a dependability ontology based upon the IFIP Working Group 10.4 taxonomy [14]. This provides the general concepts for describing dependability. Full descriptions of particular attributes of dependability (i.e. reliability, availability, security) should be provided at another layer still. In Section 5 we begin to demonstrate how to do this for availability in particular. On top of these it is still expected that many people will have their own vocabularies and their own metrics in particular. However, one reason for creating an ontology for a particular attribute is to allow the concepts that metrics of that attribute refer to, to be defined. For instance, the concept "failure" comes in useful in order to define Probability of Failure on Demand (POFOD) and allows POFOD to be defined for specific types of failure rather than just referring to failure in some more nebulous way.

As an example of the advantages of the formal semantics of OWL – consider the task of matching a QoS requirement on some metric to that advertised by a provider or QoS monitor. A QoS requirement might take the form: $\text{RequiredAvailability} \equiv \text{SomeAvailabilityMetric} \sqcap \forall\ \text{hasValue.RequiredValueRange}$. This could be read as RequiredAvailability is anything that is a SomeAvailabilityMetric and also has all values in the required value range. This range is actually stated as a custom XML datatype in practice (i.e. by restricting the range of a built-in XML datatype). An advertisement essentially takes the same form (replace "required" with "advertised" in the above piece of DL notation). It can be seen that checking whether a requirement is met by an advert is simply a case of checking whether RequiredAvailability is a subclass of AdvertisedAvailability, which can be answered by a reasoner. The nature of requirements matching remains the same with complex values although the class descriptions involved will become more complicated.

5. The Semantics of Availability

The basic problem our work highlights with existing QoS languages is that they make use of fixed syntactic tokens to represent complex concepts. For instance there may be a single token for availability as in the QML given in Section 2. There is an obvious difficulty with this: the attribute availability may be measured in many ways. The first level of distinction necessary is therefore between a dependability attribute and a specific metric of that attribute. As discussed in the previous section this distinction is provided by QoSOnt.

The next, and perhaps the most important, level of distinction required is precisely what availability metric is being used. Generally, if only a single token exists for availability in a language then it is taken to refer to the ratio uptime/(uptime + downtime). However, without knowing something more about it, this ratio may be meaningless. In the worst case one may misunderstand the meaning of the specified availability metric entirely and it may not be a ratio at all – but, for instance a probability distribution. In specifying an ontology for availability based upon QoSOnt we seek to avoid this ambiguity, whilst also providing a richer availability vocabulary. The following subsections highlight various aspects of the semantics of this vocabulary before an OWL model of availability is presented at the end of the section.

5.1. Temporal Issues

One key piece of information missing from the simple ratio approach to stating availability is the period over which it was calculated. One would normally assume that it represented availability over the lifetime of the system – but it is not obvious that this will always be the case. Depending on the exact period measured how useful it is as an indicator of current (and future) availability will vary. If the period over which availability was measured was a long time ago, or only represents a very short period of time then there can be little confidence in the figure as an indicator.

The temporal pattern of downtime to uptime is also important to most customers – but is completely lost by stating availability as a ratio. For instance, the ratio 0.98 might mean that out of a whole year, the service has only been down for a week. If this week was in a contiguous period, and the customer only wishes to perform a relatively short operation (in the order of seconds or minutes) then this could potentially indicate to them a lower risk. On the other hand if the 0.98

indicated a downtime of a second out of every minute then this would appear more risky. If the customer's operation was to take longer than a minute then this can certainly be seen to be the case, as a downtime could be *expected* to occur during *every* service call.

In any usage scenario it would always be useful to know of planned downtimes (or risk periods). If the customer is making a single service call and it falls in a planned downtime period then they can effectively regard availability for that service as zero. In a more common situation, a customer will know the likely pattern of service usage. If a high proportion of this usage falls in planned downtimes or "at risk" periods this will significantly change the availability the customer perceives. Another use of this planned downtime information might be to anticipate the need to switch to a backup service.

5.2. Measurement Issues

From different viewpoints, the way in which downtime can be measured also differs. For a party other than the provider to measure downtime it must perform some kind of regular health check. The nature of these health checks is of interest – e.g. is it a ping, an attempted HTTP connection, a genuine call to a service operation or some more indicative suite of test calls? Also relevant is how frequently this health check is performed (indeed it may not be done regularly, but at some irregular time such as in the process of performing a normal service call).

Even if the provider is stating their own availability (ignoring the issue of why they should be trusted to do so) then it is worth questioning how they are distinguishing downtime from uptime. If it is based upon based upon human observation then it may not be entirely accurate, whereas if it is based upon, e.g. server logs, it might be more so.

On the other hand, this brings up a further issue about stating availability – stating server downtime does not give the entire picture with regards to service availability. Even whilst the web or application server appears to be up the module specific to web services may have failed. For instance if a provider is running Apache Axis on top of Apache Tomcat then they may make a configuration change which breaks Axis (the SOAP implementation), whilst Tomcat still runs OK. Further to this, the service implementation itself may become unavailable due to some fault in its implementation, change of configuration, etc.

From a customer's point of view the situation is even more complicated as the provider or QoS monitor is unlikely to be able to tell them anything about the availability of the network intervening between the client and service – yet in terms of specifying the availability of an entire system this might be important to the customer. This is also one reason one must be wary about the reports of availability from third parties (i.e. QoS monitors). If they are including network failures in their availability measurement then this will not be particularly indicative of the availability that will be achieved at a different location on the Internet. In the ideal world it would therefore be nice to separate out availability into a network, server hardware, server software, and a service implementation part. There seems to be no widely adopted solution to separate the network and server-side components of availability out from a third-party point of view however, and it may be argued that it is only the aggregate figure of service availability which will ever be useful anyway. On the other hand, by not stating explicitly what part/s of a system the availability stated refers to, a provider could be misleading a customer.

Taking this granularity argument further, it may be useful in some cases to state availability of a given operation. For instance, a service may make use of an unreliable subsystem (be it a database backend, some mechanical control system, or whatever). The availability of operations which make use of the unreliable subsystem may vary from those which do not. The variance may perhaps be more obvious if one pictures some unreliable control system. Again, it is hard to see this fine granularity being used in practice – but to be able to state this level of granularity should perhaps be available in a complete specification vocabulary.

5.3. Adding Availability Concepts to QoSOnt

Sections 5.1 and 5.2 highlighted the following requirements about what should be specifiable for availability metrics:

- The period of measurement.
- The actual downtime periods.
- Planned downtime periods.
- The party responsible for the measurement.
- The method of measurement
    - Including timing of health checks where relevant
- The system component/s that the measurement applies to.

Note that, at this stage, no attempt has been made to investigate the availability metrics people use in practice beyond noting that the ratio method (how much of a given time period is uptime) is widely used. We have named this class of metric ProportionOfUptime for want of a more succinct description. Given the temporal issues mentioned in section 5.1 we suggest just one more broad class of "metric", which may in fact not be regarded as a metric at all. We introduce the DowntimeHistory as a means of supplying all downtime data without summarising it. The reason we introduce this in particular is that this could then be used to infer the values of other more specific metrics which do summarise the data.

A UML sketch of the structure of our availability ontology is shown in Figure 2. This is an overview of some of the relevant taxonomical and relational aspects of the ontology (the logical Class definitions are not shown).

To QoSMetric we have added the property measuredBy to indicate who measured it. We have added it at this generic level as it could be of interest for any metric, and this does not make the use of this property obligatory. AvailabilityMetric is defined as a convenience Class, allowing the options of defining planned downtimes, how downtime is distinguished from uptime, and the time interval over which this metric was computed. This means that we have nothing to further distinguish the ProportionOfUptime Class other than its restriction to have only decimal (or float or double) values.

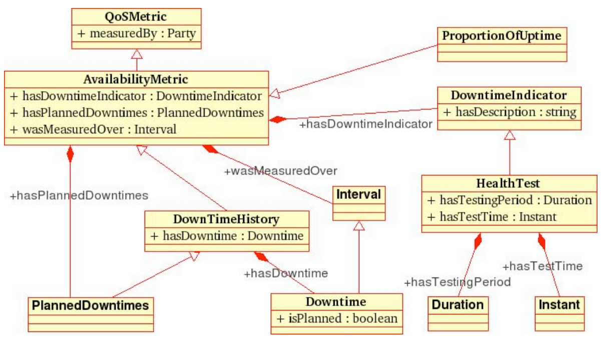

Figure 2. Classes in the Availability Ontology

Note that the PlannedDownTimes Class is shown as a subclass of DownTimeHistory. The information missing from the diagram is the Class definition: $\mathrm{PlannedDowntimes} \equiv \mathrm{DownTimeHistory} \sqcap \forall\, \mathrm{hasDowntime.PlannedDowntime}$, where $\mathrm{PlannedDowntime} \equiv \mathrm{DownTime} \sqcap \forall\, \mathrm{isPlanned}.\mathit{true}$, i.e. a PlannedDownTimes is any DownTimeHistory containing only planned downtimes. DownTimeHistory is defined such that there must be at least one Downtime listed – but is essentially an Interval (of time) that can be labelled as either planned or not planned.

The Interval, Duration and Instant classes come from the OWL-Time ontology [15] (to which we add only a few extensions to allow us to represent extra time units). One advantage of using this ontology is the formal definition of the notions of temporal relations, e.g. before, after, and for intervals: overlapped-by, contains. We envision that together with a rules language such as SWRL [13] this would give a good basis for evaluating indicators of the effective availability a service might offer in practice, by e.g comparing a customer's usage intervals with planned down time intervals. Similarly one can imagine rules being written to translate from the complete DownTimeHistory to some specific summary metric, including the ProportionOfUptime.

We have not gone into the level of describing health tests or other downtime indicators in detail. A health test may either be periodic (i.e. regular). Specific subclasses of this might be Ping or other heartbeat-type tests. The health tests may also occur sporadically. This is specified by stating the test times as an OWL-Time Instant. This applies in the situation where, e.g. the health tests are performed whilst proxying actual service calls (and therefore at unpredictable times).

The one thing missing from this picture is the indication of which system component/s the metric is relevant to. The reason this is not shown here is that this is a feature that can be achieved through the use of the concepts in the core QoSOnt ontology. Here, metrics may be associated with system components through the use of other Ontologies. For instance, metrics can be associated with web services by making use of the OWL-S ontology [16] – an ontology for service specification. There are facilities in QoSOnt which provide this link (a lightweight Service ontology provides the linkage layer). For an example of this see [11].

6. Conclusion and Future Work

In this paper we have discussed the shortcomings of existing QoS languages, firstly with regards to their lack of coverage of dependability, and primarily with regards to their lack of semantic depth. Our work uses OWL to explicitly describe the semantics of dependability. OWL's formally defined semantics (reducible to fragments of predicate logic) lead to unambiguous meaning, whilst providing a framework for describing the domain in question. No attempt is made to relate the ontology to details of providing the levels of QoS specified, which itself is an open research area.

In previous papers we have detailed the QoSOnt ontology, i.e. that providing the basic concepts of QoS and dependability. Here, by concentrating on availability we have shown how to build upon this ontology to describe metrics in greater detail. We have provided one particular availability vocabulary which we by no means regard as exhaustive – but in doing so have demonstrated the concepts of extending QoSOnt and perhaps provided a number of building blocks for other availability vocabularies.

Given that OWL is a logic-based language, with a somewhat obscure syntax (partly due to its concentration on machine interpretability) it cannot be expected that there will be many experts in a position to extend upon QoSOnt as we describe here. Therefore, a future direction of this work is to provide an interface for non-experts to define their own dependability vocabulary. This should insulate them entirely from the details of OWL if possible. Such a tool, along with the ontology should provide an excellent framework to develop a widely usable set of QoS vocabularies which are easily translated and interrelated. We see this as a key to enabling QoS in service-centric systems due to the inevitable involvement in such systems of multiple organisations.

7. Acknowledgements

This work is funded as part of the Service Centric Systems Engineering (SeCSE) EU Integrated Project 511680. With regards to the initial work on QoSOnt the authors also acknowledge the support of EPSRC award no. GR/S04642/01, Dependable, Service-centric Grid Computing; and particularly the initial contribution of Russell Lock to the ontology.

8. References

[1] Ernesto Exposito et al, “XML QoS specification language for enhancing communication services”, in Proceedings of the *15th International Conference on Computer Communication*, pp. 76-90, 2002

[2] Patrícia Gomes, Soares Florissi, “Quality of Service Management Automation in Integrated Distributed Systems”, in Proceedings of the *Conference of the Centre for Advanced Studies on Collaborative Research*, p. 18, 1994

[3] Svend Frolund, Jari Koisten, “QML: A Language for Quality of Service Specification”, HP Labs Technical Report, http://www.hpl.hp.com/techreports/98/HPL-98-10.html, 1998

[4] Alexander Keller, Heiko Ludwig, “The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services”, in the *Journal of Network and Systems Management*, vol. 11, No. 1, pp. 57-81, 2003

[5] R. Braden, D. Clark, S. Shenker, “Integrated Services in the Internet Architecture: an Overview” (IETF RFC1633), http://www.ietf.org/rfc/rfc1633.txt, 1994

[6] K. Nichols, S. Blake, F. Baker, D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers“ (IETF RFC2474), http://www.ietf.org/rfc/rfc2474.txt, 1998

[7] L. Zhang,S. Berson, S. Herzog, S. Jamin, “Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification” (IETF RFC2205), ftp://ftp.isi.edu/in-notes/rfc2205.txt, 1997

[8] W3C, “Resource Description Framework”, http://www.w3.org/RDF/

[9] W3C, “Extensible Markup Language (XML)“, http://www.w3.org/XML/

[10] Franz Baader, Ian Horrocks, and Ulrike Sattler, “Description logics for the semantic web”, in *Künstliche Intelligenz*, Vol. 16, No. 4, pp. 57-59, 2002

[11] Glen Dobson, Russell Lock, Ian Sommerville, “QoSOnt: a QoS Ontology for Service-Centric Systems”, 31st Euromicro Conference on Software Engineering and Advanced Applications, pp. 80-87, 2005

[12] Glen Dobson, "Quality of Service in Service-Oriented Architectures”, http://digs.sourceforge.net/papers/qos.pdf

[13] Ian Horrocks et al, “SWRL: A Semantic Web Rule Language Combining OWL and RuleML”, W3C Member Submission, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, 2004

[14] Jean-Claude-Laprie, Brian Randell, Carl Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing”, in *IEEE Transactions on Dependable & Secure Computing*. Vol. 1, No. 1, pp. 11-33, 2004

[15] Jerry R. Hobbs, Feng Pan, “An Ontology of Time for the Semantic Web”. In *ACM Transactions on Asian Language Processing (TALIP)*: Special issue on Temporal Information Processing, Vol. 3, No. 1, pp. 66-85, 2004

[16] DAML, “DAML Services”, http://www.daml.org/services/owl-s