Open vs Closed world, Rules vs Queries: Use cases from Industry

Gary Ng

Cerebra Inc., Carlsbad, California, USA
gary@cerebra.com

Abstract. This paper is a discussion of the application of OWL driven by commercial use cases. These use cases support the integration of OWL with rule-like extensions, taking advantage of open-world semantics, but with defaults and negation as failure. This paper reports on our analysis of these use cases, current solutions using queries, and outlines the role OWL plays in a semantically driven enterprise architecture.

1 Introduction

The realization of a new ontological web with interoperable services begins from behind enterprises' firewalls. Taking aim at corporate data, the value of semantics is to provide adaptive and agile data management capabilities that will stand up to complex demands. In parallel, enterprise architecture is also moving towards semantically driven services that are discoverable by machines automatically. The goal is to place the control and the agility of business processes directly under the hands of business users, offering appropriate level of information to the right hands at the right time for decision making.

A few of the issues reported in the recent Rules Workshop [3] were related to the differences between declarative knowledge and a programming language. While interoperability of rules across diverse systems is an important issue for the semantic web, maintainability of the rules with precise semantics is more important within a cooperate environment. As such the reliance on rule ordering and conflict resolution should be avoided.

We report on two commercial use cases where rules and OWL-DL complement each other, favoring the "layered" architecture over the "two towers" approach [5]. For each use case we explain why the enterprise is looking into using semantic technologies, and how their use cases are described using OWL-DL. We also propose how OWL, rules with negation as failure (NAF) and other kinds of rule can fulfill the use cases in a single coherent manner.

2 Background

Two use cases are described. The first is on the use of OWL in adaptive enterprise networks, the second is on product classification for market report generation. For each, we describe the purposes and aims, how it is currently solved, and the motivation for using OWL.

2.1 Adaptive Enterprise Network

The day to day operation of an enterprise relies on many individual computers, servers, and IT systems working in concert. Each system must be functional and meet the demands of business processes. Consider a manufacturer with systems dedicated for purchasing and orders, accounting and finance, pricing and discounts, production assembly, packaging and shipping. Each system in turn is made up of different subsystems. For example, a fleet of computers to calculate trading gains and loss every day within accounting; Business to business messaging middleware to contact suppliers' web-services to purchase parts; Business process engines governing the procedural flow of individual operations for ordering and shipping. Some systems are physical hardware entities, some are software and some systems are only conceptual entities. Let's refer to these systems as "nodes" in a network of systems. Each node offers some service and may rely on other nodes. Many may rely on email servers to communicate and report problems back to the employees within the organization. Failure or a poor quality of service in any one node may affect the overall operation of the organization and responsiveness to potentially damaging events.

A business process management tool monitors the performance and health status of all automatic processes within an organization. Human operators are alerted of any failing nodes, which must be investigated and rectified. A self-healing enterprise network is one where the human involvement is minimized, in which backup systems and equivalent services meeting the same demands are brought into service dynamically, replacing those that are failing.

The problem here is more complex than simply providing secondary backup on every system. A failure of a service is not simply a matter of not working, which can be rectified by a secondary backup. There are many conceptual levels of failure, such as failing to meet response time or bandwidth, and each service/system may have additional metrics to define the quality of service. The solution to each kinds of failure is also different. Some solutions involve borrowing additional resources to share the load. Each new resource must also be configured with the right software. Similarly, the definition of "equivalent" services is a conceptual issue in itself, where not just the same functionality has to be met, but it is also a function of response time, bandwidth and reliability.

Vendors offering solutions in this area have begun to look at formal semantics as an alternative to probabilistic methods. These companies see the OWL recommendation as a means to allow more precise definitions of failures for better provenances. The use of formal knowledge representation can potentially lead to a cleaner paradigm to describe, maintain and control the detection and diagnostic processes.

Clearly this problem is large and is not just an issue of conceptual modeling. However, we shall ignore much of the technical details of how such self-healing network is realized. In this paper we focus on the conceptual modeling issue surrounding the detection of root causes of a failing, or sub-optimal network. Having root causes identified could help both the human operator and any automatic processes in rectifying the problem.

In the discussion that follows, the metadata of each network node and their status is captured. Within a business process manager, an ontology with complex axioms is used to determine the "Goodness" of a node. In this use case the ontology with its instance data represents a "snapshot" of the network status. This use case is highly event driven, and inferencing service is used to detect problems by classifying node status. The discussion here focuses on defining the notion of nodes that are root causes of a failure. It is defined as something that is itself failed, but none of the services it relies on has failed. It is here that the element of negation of failure is relevant.

2.2 Product Classification for Market Reporting

A timely financial report meeting internal demands as well as government regulations is an increasingly important issue. Such financial report contains product revenue data organized into a variety of technology market segments. Each segment represents a market and the company's share in that market is judged based on the report. Misreporting has significant consequences, from affecting the company's market share, to attracting legal allegations.

In one example, there are 50,000 products stored in a database; each product can be classified into multiple market segments depending on the technologies the product is built on. For example, network routers are also switches and a router often contains a firewall in the firmware. Depending on the product line and the embedded technology, products such as routers can be classified as either high-end, low-end, or mid-range; in addition to being a firewall and switches with similar sub-categorization.

Overall there are over 500 market segments into which products are classified. These market segments form a hierarchy. For market analysts, such report also includes data on units sold in each segments.

In the use case being studied, classification rules specified against products were implemented using spreadsheets and PLSQL. Each rule body contains a variety of criteria, from product features, technologies it contains, to criteria based on corporate categories such as product families and the respective business units. There were approximately 200 rules for product classification, and another 200 rules for distributing units sold among market segments when a single product falls into multiple categories. Its translation into OWL features axioms specifying class memberships of individuals, base on the properties of the individuals.

The requirement here is to automatically classify all products correctly into market segments, according to a set of business defined classification rules. These rules may change over time, as segment definitions changed, new products may be introduced and old products made obsolete. The existing implementation of rules based on spreadsheet and PLSQL is complex. The rule set is order dependent, and their specifi-

cation requires technical expertise, akin to application programming. In this use case, much of the complexity in writing and maintaining the rule set comes from the order dependent nature: If a product is not positively identified and dealt with by any previous rules, then it should classified as X. The maintenance and extension of a rule often involve looking at all preceding rules. One must be certain whether or not a particular change will result in (un)desirable classification. Such concern lengthens the time to integrate, test and put changes into production.

The motivation to move away from rule programming towards a declarative semantics such as that offered by OWL holds the potential for a simpler rule language. One that is usable directly by business users and financial analysts, and changes can be made quickly. These users are the domain experts on how these products should be classified and transaction data be reported. Declarative semantics offers precise and easily reconfigurable classification of products into market segments that can ultimately shorten the time to produce such reports.

The example in focus here is on the classification of network routers. The issue here is the use of defaults in classification to deal with incomplete data, or to put things into a broad market category.

3 Analysis

3.1 Adaptive Enterprise Network

The ontology we have constructed contains a class “Node”, which can represent a server or a software system. Different kinds of nodes are modeled as sub-classes of “Node”. Individual nodes are modeled as instances. Each node may have zero, one, or more "reliesOn" relationships to other nodes. For example, a “Bugzilla” node might rely on an “EmailRelay”; an "EmailRelay" node in turn might rely on a "DNSServer" node.

All nodes must fall into two kinds: "GoodNode" and "BadNode". Further, we assert that any node with a "reliesOn" relationship to a "BadNode" means itself is a "BadNode". All these are both intuitive and well within the expressiveness of OWL-DL. In practice, the "reliesOn" relationship can also be asserted as transitive: if A reliesOn B and B reliesOn C, then A reliesOn C, but that property isn't necessary for the discussion here.

The ontology thus contains a graph of instances representing the inter-dependency of nodes in the enterprise network. Each node has its own class of node type. Each type of node has a set of sufficient and necessary conditions under which it becomes a “BadNode”. Some definitions may involve other successor nodes. For this OWL-DL seems perfectly suited, offering formal and complex classification of network nodes.

At runtime, the ontology is updated with status reports from each node. Each time a status change the whole network is reclassified and problems identified. The problem is in the form of the question we wish to ask: "Given that we know there's a problem, what are the possible root causes of that problem?”.

We translated this notion of a "RootCause" to the OWL-DL concept:

$$\text{RootCause} \equiv \text{BadNode} \sqcap \forall \text{reliesOn GoodNode}$$

We are looking for nodes which are “BadNodes”, but not because of another “BadNode” upon which they depend (that “BadNode” dependency would have to be fixed first). In other words, a “BadNode” is one that only depends on “GoodNodes”.

So we're looking for nodes which fit this "RootCause" concept. The trouble comes with what we mean by "fit". Hoping that simply retrieving the members of “RootCause” is the answer is naive. Here we have an open-world view of the world. An OWL-DL engine cannot prove memberships to “RootCause” unless it is known that some nodes definitely do not rely on any “BadNodes”, or only rely on “GoodNodes”.

For example, we've asserted that the “Bugzilla” node “reliesOn” "EmailRelay" and "EmailRelay" “reliesOn” a "DNSServer". From a closed world view, a malfunctioning “Bugzilla” and a malfunctioning “EmailRelay” with a functional “DNSServer” would suggest “EmailRelay” is a root cause in this simple network.

In an open world however, it is assumed that something is possible unless explicitly stated otherwise. We have no definitive assertions on whether “EmailRelay” rely on *only* good nodes, or *only* bad nodes. Thus it is entirely correct for an OWL-DL engine to conclude that a malfunctioning “EmailRelay” with a functional “DNSServer” does not necessarily mean that the “EmailRelay” itself is the root cause.

With an open world system, what we're really looking for are nodes that might be the root cause. The “EmailServer” might be the cause of the whole problem in this case. To explain the notion of “might be”, consider the following. For all “Nodes” and a sub-class C, we can partition all nodes into three distinct sets:

1) Those that are known to lie within C, said to be a member of C.
2) Those that are known to lie outside of C, said to be a member of ¬C.
3) Those that might lie inside or outside C, isn't a member of either C or ¬C.

In this case, let C be “RootCause”, we're looking for everything except group 2: those things that are definitely root causes and things that may or may not be root causes. Since everything must fall into one of these three groups, all we have to do is subtract everything from group 2 from the set of nodes: subtract all members of "¬RootCause" from the set of nodes. In our example, "Bugzilla" and "DNSServer" would be returned as the subs of "¬RootCause", and we would return only "EmailServer" to the user.

This satisfies the use case by delivering the expected results. However, the way the question is asked and the result is derived is not as straight forward as the retrieval of other classes.

It should be pointed out that the hierarchy of bad nodes in terms of their “reliesOn” relationship could be traversed to work out the root cause, which is a much simpler method than classification. The suggested solution here represents what one can do if it has been mandated to use an OWL reasoning engine. It was a worthy exercise to explore the boundary of an OWL reasoning system as a black box, uncovering “how much inferencing it can or cannot do” for a given problem. To a lot of commercial customers this is a mystery. It is clear here that some custom code has to be written on

top in order to solve the problem completely: either to perform graph traversals, or to perform additional satisfiability tests.

3.2 Product Classification for Market Reporting

The promise of complex classification by formal definitions has attracted this use case to OWL. Formal semantics hold the key to complex product typing involving product features, a hierarchy of product families as well as a hierarchy of business units. Once the formal definitions of the core operational data is devised, other aspects such as secure data access and separation of duties (such as financial analysts, corporate users) can be defined on top; forming an information infrastructure based on semantics.

An abstraction of the issue can be described as follows. Let $\{X_1, X_{2...}, Y_1, Y_{2...}, Z_1, Z_{2...}\}$ be the set of technology features in the domain, each are represented as concepts, and $p_1, p_2, ...$ be the individual products in the domain. Each product p_i may be a member of one or more of these features. Router, HighEnd, LowEnd, and MidRange are concepts representing technology market segments in the domain:

1) Router = $\{ p_i \mid p_i \in Z_1, p_i \in Z_{2...} p_i \in Z_n \}$. Products fall within certain criteria are routers.
2) Routers are disjoint covered by HighEnd, LowEnd and MidRange
3) HighEnd = $\{ p_i \mid p_i \in X_1, p_i \in X_{2...} p_i \in X_n \}$. Those contain certain technology features are high end routers.
4) LowEnd = $\{ p_i \mid p_i \in Y_1, p_i \in Y_{2...} p_i \in Y_n \}$. Those contain certain technology features are low end routers.
5) The remaining Routers are considered MidRange.

The last rule is a default classification. The class of "MidRange" routers represents those that are identifiable neither as HighEnd nor LowEnd. Representing this requires negation-as-failure. This group of Routers cannot be provably identified as "MidRange" within an open world assumption with respect to the set of axioms above.

With the disjoint covered axioms (2), a product that does not fit the sufficient condition of a "HighEnd" router is automatically in the complement set of "HighEnd", and likewise for "LowEnd". However, it does not follow that any routers which might conceptually be "MidRange" are necessarily always identifiable as either "¬HighEnd" or "¬LowEnd". In some cases where the criterion is in the concrete domain this is trivially satisfied. However, not all criteria are in the concrete domain and incomplete information is often expected. A solution is to explicitly assert that such an instance does not satisfy at least one criterion within the set of X or Y. However, such amount of explicitness is hard to maintain and defeats the point of using an open-world system that is capable of handling incomplete information.

It can be shown that this is in fact similar to the previous example. Here we are looking for members of Routers which might be "¬HighEnd" or "¬LowEnd". Thus this can also be retrieved through result set manipulation.

5 Experience and Discussion

5.1 Negation as failure (NAF)

Both examples exhibit some form of closed-world assumption. In the root cause analysis example, in order to identify something as a root cause of a problem over a chain of faulty "nodes", one needs to explicitly assert that something is in fact at the end of the faulty dependency chain. This essentially closes the world for that node, i.e. we have complete information on that node with respect to its dependencies. In the product classification, where the conjunction of a set of criteria is involved, it is not necessary to have complete information on a product for it to be classified as expected, but it must have sufficiently complete information (failing at least one of the criteria) for the inference to occur. This makes it very difficult to determine just how much information is required in each individual case.

It would be convenient for modelers in these cases to use a concept constructor $\neg_{naf}$, with which the concept "RootCause" node can be defined as:

$$\text{BadNode} \sqcap \neg_{naf} (\exists \text{ reliesOn BadNode}) \quad (1)$$

In other words, a "RootCause" node is a "BadNode" that has zero "reliesOn" relationship to a "BadNode", with respect to the current known state of the universe. Simply put, it is a bad node that is *not known* to rely on any bad node.

Similarly, the definition of "MidRange" could be reformulated as:

$$\text{Router} \sqcap \neg_{naf} \text{HighEnd} \sqcap \neg_{naf} \text{LowEnd} \quad (2)$$

In other words, a "MidRange" router is a router that is neither proven to be "HighEnd" nor "LowEnd", but does not necessarily possess any information in the current known state of the universe that it is within their complement set either. Simply put, it is a router that is *not known* to be high end, and is *not known* to be low end.

Such operator can be supported by a powerful query language. Query languages already have a closed world flavor, as distinguished variables can only bind to named individuals. As already discussed in the previous section, it is natural to implement this by way of query subtraction.

This operator is in fact the epistemic operator **K** as described by Donini *et al* [1]. The **K**-operator is a formalization of negation as failure, closed-world queries, and other popular features of non-monotonic systems. It has been shown that it can be added onto open-world concept languages while preserving the open-world semantics of ordinary (non-K) queries. Their work gives a clear formal semantics and the computational properties of its algorithms for query answering are known. However, to date we are not aware of **K**-operator support in any implemented systems, it is not in the specification of OWL, and that it isn't clear there exists an efficient procedure for its implementation.

Another approach which can positively identify the desired set of instances without formal support for the **K**-operator is to exploit the satisfiability of these instances with the complement of certain classes. For example, consider those instances inferred to be members of the expression (∃ reliesOn BadNode), let's call this expression rBN. It is known that rBN(x) is satisfiable and that ¬rBN(x) is not satisfiable. We say it is *provable* that rBN(x) is true, i.e. rBN(x) is true in all models of the KB. For the remainder of the instances that offer no evidence it is relying on any BadNode, both rBN(x) and ¬rBN(x) are *satisfiable,* i.e. either rBN(x) and ¬rBN(x) is true in any models of the KB. Thus our desired set of instances which are potentially root causes are exactly those where ¬rBN(x) is satisfiable, i.e. (∀ reliesOn GoodNode) is satisfiable. We introduce two query predicates: *Provably* and *Satisfiably* in the remainder of this discussion to capture the above notions.

Thus the set of "RootCause" nodes {x} is:

$$\text{Provably}(\text{BadNodes}(x)) \cap \text{Satisfiably}(\forall \text{ reliesOn GoodNode}(x)) \quad (3)$$

Similarly in the product example, for those instances inferred to be a members of either "HighEnd" or "LowEnd", lets call them HE(x) and LE(x) respectively. It is known that A(x) is satisfiable and that ¬A(x) is not satisfiable, where A can be HE or LE. For the remainder of the instances that are intended to be "MidRange", both A(x) and ¬A(x) are satisfiable.

Thus the set of "MidRange" routers {y} is:

$$\text{Provably}(\text{Router}(y)) \cap \text{Satisfiably}(\neg\text{HighEnd}\ (y)) \cap \text{Satisfiably}(\neg\text{LowEnd}\ (y)) \quad (4)$$

Note the symmetry of (1) and (3), as well as (2) and (4). Also note that in both cases Satisfiably (¬A) here includes those instances that are provably in ¬A, as well as those that might possibly be ¬A.

The above has demonstrated that such form of negation-as-failure can be implemented on top of purely open-world systems using queries. It is worth noting that to our knowledge, none of the query languages: RDQL [8], SPARQL [6] and nRQL [2] currently has any means to retrieve satisfiable results. It is probably easy for nRQL to provide such features since it is based on an open world system, whereas RDQL and SPARQL are based on a closed world of instance graphs in the first place.

There is still one remaining problem however. Using a powerful query interface to collect instances that fall into such a "satisfiable" category is all well and good, but there is still an asymmetry in the retrieval method of these results, compare to instance retrieval with any other classes. For external entities to know the specifics of how the knowledge is defined and to issue context dependent queries is a non-starter. From the end user's point of view, the ability to make assertions using these query predicates or the **K**-operator will be invaluable.

5.2 Knowledge Base as a Black Box

The ideal case is where axioms such as (1) and (2) can be specified among other OWL-DL axioms and/or SWRL rules [4], and the results are simply retrieved by asking for members of "RootCauses" and "MidRange" respectively. Furthermore, members of such classes may in turn participate in further classification by other axioms in the ontology. That way, the best of both open and closed worlds is available to the user; the user can specify knowledge in a way more natural to him/her. For example, at the language level a user could say that the default for the class of routers is "MidRange". How such "rule" is implemented should be opaque to the user. One approach is for any new instances of routers that are not known to be High/Low-End be explicitly asserted to be instances of "MidRange"; Thereby achieving encapsulation from the point of view of specification, as well as symmetry from the point of view of querying.

Note that the notion of "if x is a result of a query q then assert something about x" is essentially a rule. Thus what we are proposing here is inline with the layered approach as expressed in [5]: rules are implemented on top of an open-world language, taking advantage of the formal definition of atoms that such language offers.

5.3 Semantic Model Driven Architecture

The use cases reported here are only two of many areas where the use of OWL is explored in the commercial world. It is becoming clearer to many enterprises that existing integration technologies are not reducing costs and are not delivering the required information. Technologies such as data warehousing, extract, transform and load (ETL), enterprise application integration (EAI) and enterprise information integration (EII) have created more versions of the truth about their data than is desirable. Each technology requires a separate modeling and mapping effort, with individual proprietary models of business domain knowledge. With most technologies focusing mainly at the syntactic level, semantic integration is becoming a hot topic [7].

The vision of a semantically integrated infrastructure has two parts: 1) All subsystems employ a single formalism to specify knowledge; 2) all data meaning are mapped to a single semantic model, but operational data remains decentralized residing within their respective operational systems. All existing integration technologies will remain in operation, but with their models mapped to each other via the semantic model. This model serves as the central component offering a single version of truth of data. This is the vision of a semantic model driven architecture. Being an international standard grounded on formal logic, OWL is at the heart of this vision.

6 Conclusions

Commercial enterprises are currently exploring semantic knowledge driven architecture and web services. The main contribution of this paper is the detail description of two commercial use cases, both aimed to exploit the power of open-world semantics and the OWL-DL standard. We have outlined the necessity of a notion of nega-

tion-as-failure within these use cases. A concept constructor $\neg_{naf}$ is introduced which applies equally to classes as well as instances. It is similar to the **K**-operator and we have provided an implementation approach using only open-world query answering services. We believe an extension to OWL with these features will allow a wider and faster adoption of the standard.

References

1. Donini, F. M., Lenzerini, M., Nardi, D., Schaerf, A., Nutt, W.: An epistemic operator for description logics. Artificial Intelligence. 100(1-2):225-274. (1998)

2. Haarslev, V., M¨oller, R., Wessel, M.: Querying the Semantic Web with Racer + nRQL. In *Proc. of the KI-2004 Intl. Workshop on Applications of Description Logics (ADL'04),* (2004.)

3. Hawke, S.: Report from the W3C Workshop on Rule Languages for Interoperability. Available at: http://www.w3.org/2004/12/rules-ws/report/. (2005)

4. Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Available at: http://www.daml.org/2003/11/swrl/. (2003)

5. Horrocks, I., Parsia, B., Patel-Schneider, P., Hendler, J.: Semantic Web Architecture: Stack or Two Towers? Available at: http://www.cs.man.ac.uk/~horrocks/Publications/download-/2005/HPPH05.pdf. (2005)

6. Prud'hommeaux, E., Seaborne, A.,: SPARQL Query Language for RDF. W3C Working Draft, Available at http://www.w3.org/TR/rdf-sparql-query/. (2005)

7. Raden, N.: Start Making Sense: Get From Data To Semantic Integration. Intelligent Enterprises. October, 2005. Available At: http://www.intelligententerprise.com/showArticle.jhtml ?articleID=171000640. (2005)

8. Seaborne, A.: RDQL - A Query Language for RDF. W3C Member Submission. Available at: http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/. (2004)