SQWRL: a Query Language for OWL

Martin O'Connor, Amar Das

Stanford Center for Biomedical Informatics Research
Stanford, CA 94305 martin.oconnor@stanford.edu

The ability to extract information from OWL ontologies is a basic requirement. While SPARQL and its extensions are being used as an OWL query language in many applications, their understanding of OWL's semantics is at best incomplete. There is a pressing need for a concise, readable, and semantically robust query language for OWL. We describe a query language called SQWRL that we believe provides such a language. SQWRL is based on the SWRL rule language and uses SWRL's strong semantic foundation as its formal underpinning. The resulting language provides a small but powerful array of operators that allows users to construct queries on OWL ontologies. SQWRL also contains novel set operators that can be used to perform closure operations to allow limited forms of negation as failure, counting, and aggregation.

1 Introduction

A number of query languages have been developed to query RDF and OWL, the two dominant Semantic Web languages. SPARQL [1] is currently the *de facto* standard RDF query language. SPARQL has also been pressed in to service as an OWL query language. Since OWL can be serialized as RDF, SPARQL can, in principle, be used to query it. However, SPARQL has no native understanding of OWL. It operates only on its RDF serialization and has no knowledge of the language constructs that those serializations represent. As a result, it can not directly query entailments made using those constructs. Moreover, there is no canonical RDF serialization of some OWL constructs. Different tools can serialize the same OWL constructs in slightly different ways so SPARQL queries may perform differently when applied to ontologies produced by different tools. Custom variants of SPARQL have been developed to facilitate its use with OWL [2], though its essentially syntactic focus means that it is not a natural fit when used with complex OWL constructs. Fundamentally, basing an OWL query language on a particular OWL serialization entails significant language design compromises and unnecessary semantic complexity. A language design approach that builds directly on OWL's DL-based semantics can generally provide a more sound solution. Two such languages, OWL-QL [3] and DIG's ASK protocol [4], are based directly on OWL and have well-defined DL-based semantics. Unfortunately, no usable implementations of OWL-QL were produced and the ASK protocol is too inexpressive to be used as a general OWL query language. There is thus a need for an expressive OWL query language that is built solidly on its DL semantics and that supports comprehensive querying of OWL.

2 SQWRL

SQWRL (**S**emantic **Q**uery-enhanced **W**eb **R**ule **L**anguage; pronounced *squirrel*) is built on the SWRL rule language [5]. SQWRL takes a standard SWRL rule antecedent and effectively treats it as a pattern specification for a query. It replaces the rule consequent with a retrieval specification. SQWRL uses SWRL's built-in facility as an extension point [6]. Using built-ins, it defines a set of operators that that can be used to construct retrieval specifications. The attractiveness of this approach is that no syntactic extensions are required to SWRL. Thus, existing SWRL editors can be used to generate and edit SQWRL queries. In addition, standard SWRL serialization mechanisms can be used so queries can be stored in OWL ontologies.

Core Language Features: Basic Querying

The core SQWRL operator is `sqwrl:select`. It takes one or more arguments, which are typically variables used in the pattern specification of the query, and builds a table using the arguments as the columns of the table. For example, the following query retrieves all persons in an ontology with a known age that is less than 9, together with their ages:

Person(?p) ^ hasAge(?p, ?a) ^ swrlb:lessThan(?a, 9) → sqwrl:select(?p, ?a)

This query will return pairs of individuals and ages with one row for each pair. Results can be ordered using the `orderBy` and `orderByDescending` built-ins. For example, a query to return a list of persons ordered by age can be written:

Person(?p) ^ hasAge(?p, ?a) → sqwrl:select(?p, ?a) ^ sqwrl:orderBy(?a)

The left hand side of a SQWRL query operates like a standard SWRL rule antecedent with its associated semantics. So, for example, the atom `Person(?p)` will match not only all OWL individuals that are directly of class `Person` but will also match individuals that are entailed by the ontology to be individuals of that class. In effect, all variables that would be bound in a SWRL rules antecedent will also be bound in a SQWRL pattern specification. SQWRL places no restrictions on the left hand side of a query—any valid SWRL antecedent is a valid SQWRL pattern specification.

Basic counting is also supported by SQWRL, provided by a built-in called `sqwrl:count`. A query to, say, count of the number of known persons in an ontology can be written:

Person(?p) → sqwrl:count(?p)

Grouped counts are also supported. For example, a query to count the number of cars owned by each person in an ontology can be written:

Person(?p) ^ hasCar(?p, ?c) → sqwrl:select(?p) ^ sqwrl:count(?c)

This query returns a list of individuals and counts, with one row for each individual together with a count of the number of cars that they own. Individuals that have no cars are not matched by this query. Every person individual is effectively grouped and the `count` built-in keeps track of the number of occurrences of each car associated with each person. This process is analogous to SQL's GROUP BY clause—the only difference being that grouping is implicit.

Basic aggregation is also supported, provided by four built-ins called `min`, `max`, `sum`, and `avg`. Aggregation operators take a single argument which must represent a numeric type. For example, a query to return the average age of persons in an ontology (for which an age is known) can be written:

Person(?p) ^ hasAge(?p, ?age) → sqwrl:avg(?age)

It is important to note that SQWRL's counting and aggregation built-ins operate on the query result itself, not on the underlying ontology. The `count` built-in, for example, keeps track of the number of relevant items matched in a query, not the number of such items in the ontology being queried. For example, the earlier query that determines the number of cars owned by each individual in an ontology will not match individuals that do not own a car because the `hasCar(?p, ?car)` atom in the rule will evaluate to false for those individuals. In other words, the count operator in SQWRL will never return zero. OWL's open world assumption is thus not violated. Also, the result of this built-in is not accessible from within a rule so it can not be used to violate OWL's monotonicity property. A related restriction is that SQWRL adopts the unique name assumption for matched individuals, so will count each individual as distinct even though in the associated OWL ontology these multiple names may refer to same underlying individual.

Like SWRL, SQWRL supports the use of OWL class descriptions. For example, a query to retrieve all individuals in an ontology that are associated with a restriction on an `hasChild` property specifying a cardinality greater than or equal to one can be written:

(hasChild >= 1)(?i) → sqwrl:select(?i)

SQWRL queries can also operate in conjunction with SWRL rules in an ontology and can be used to retrieve knowledge inferred by those rules. Assume, for example, that an ontology contains the following rule to classify persons as adults if they are older than 17:

Person(?p) ^ hasAge(?p, ?age) ^ swrlb:greaterThan(?age, 17) -> Adult(?p)

A query to list all adults in an ontology can then be written:

Adult(?p) → sqwrl:select(?p)

The use of the intermediate inferences made by SWRL rules provides a mechanism to decompose very complex queries. While subqueries are not possible in SQWRL, the use of these intermediate inferences provides an effective equivalent. These inferences can also be used by other rules and queries.

Set Operators: Closing the World

The core SQWRL built-ins support some degree of closure when querying without violating OWL's open world assumption. As shown, queries like *List all patients in an ontology and the number of drugs that they on* can be expressive fairly directly. However, queries with more complex closure requirements can not be expressed using the core built-ins. For example, the query *List all patients in an ontology that are on more than two drugs* can not be expressed. Queries with negation or complex aggregation functionality are similarly not expressible. Disjunction is also an issue in SQWRL: because it does not modify SWRL's syntax in any way, only conjunction is allowed.

We have added set operators to SQWRL to support these additional requirements. Operators to construct and manipulate sets are provided and are used to support the closure operations necessary for these functionalities.

A built-in called `sqwrl:makeSet` is provided to construct a set. Its basic form is:

sqwrl:makeSet(<set>, <element>)

The first argument of this set construction operator specifies the set to be constructed and the second specifies the element to be added to the set. This built-in will construct a single set for a particular query and will place all supplied elements into the set. Operators like `sqwrl:isEmpty` and `sqwrl:size` can then be applied to these sets. A new SQWRL pattern specification clause is provided to contain these set construction and manipulation operators. This clause comes at the end of the standard pattern specification and is separated from it using the ° character. For example, a query to list the number of persons in an ontology can be written:

Person(?p) ° sqwrl:makeSet(?s, ?p) ^ sqwrl:size(?size, ?s) → sqwrl:select(?size)

The new clause may only contain SQWRL set construction and manipulation operators in addition to other built-ins that operate on the results of these operations. It may not contain any other SWRL atom types. The introduction on the new ° separator character does not prevent the use of the standard SWRL serialization for SQWRL queries that use it: the character itself does not have to be saved and its display position can be inferred by tools when reading serialized queries.

Putting elements into sets provides a closure mechanism. Clearly, two phase processing is required for these queries—a query cannot, say, determine how many elements there are in a set until the set has been constructed. As mentioned, the language restricts the atoms that are processed in the second phase to set built-ins and other built-ins that operate on the results of these set built-ins. That is, the first phase of query execution is analogous to standard rule execution. The second phase consists of operations on the sets constructed as a result of that execution.

In addition to basic counting, query features such as negation as failure and disjunction can also be provided by additional set operators. In SQWRL, these operators include `sqwrl:union`, `sqwrl:difference`, `sqwrl:intersection` and so on. Using these operators, queries can effectively examine the results of two or more closure operations, permitting the writing of far more complex queries.

For example, in an ontology with a class `Drug` and its subclass `BetaBlocker`, a query to list the number of *non* beta-blocker drugs can be written:

Drug(?d) ^ BetaBlocker(?bbd) °
sqwrl:makeSet(?s1, ?d) ^ sqwrl:makeset(?s2, ?bbd) ^
sqwrl:difference(?s3, ?s1, ?s2) ^ sqwrl:size(?n, ?s3)
→ sqwrl:select(?n)

Using this set difference approach, SQWRL can effectively provide negation as failure in queries. Again, the results of these operators can not be used in rules so monotonicity is ensured.

Disjunction can be supported using the set union operator. For example, a query to list the number of beta-blocker *or* anti-hypertensive drugs can be written:

AntiHypertensive(?htnd) ^ BetaBlocker(?bbd) °
sqwrl:makeSet(?s1, ?htnd) ^ sqwrl:makeSet(?s2, ?bbd) ^
sqwrl:union(?s3, ?s1, ?s2) ^ sqwrl:size(?n, ?s3)
→ sqwrl:select(?n)

This example assumes that the class `AntiHypertensive` is also a subclass of `Drug`.

These types of sets support some fairly basic counting and aggregation operators. For example, the earlier query to list all patients in an ontology on more than two drugs is still not expressible using this approach. Additional set construction operators are required to allow more complex queries that group related sets of entities. This additional expressivity is supplied by sets that are partitioned by a group of arguments. A built-in called `sqwrl:groupBy` provides this functionality. The general form of this grouping is:

sqwrl:makeSet(<set>, <element>) ^ sqwrl:groupBy(<set>, <group>)

This group can contain one or more entities. The first argument to the `sqwrl:groupBy` built-in the set and the second and (optional) subsequent arguments are the entities to group by. Only one grouping can be applied to each set. This grouping mechanism is again analogous to SQL's GROUP BY clause.

Using this grouping mechanism, the construction of a set of drugs taken by each patient can be written:

Patient(?p) ^ hasDrug(?p,?d) °
sqwrl:makeSet(?s, ?d) ^ sqwrl:groupBy(?s, ?p)

Here, sets are built for each patient matched in the query and all the drugs for each patient are added to their set. Using this grouped set, the query to list all patients on more than two drugs can then be written:

Patient(?p) ^ hasDrug(?p,?d) °
sqwrl:makeSet(?s, ?d) ^ sqwrl:groupBy(?s, ?p) ^
sqwrl:size(?n, ?s) ^ swrlb:greaterThan(?n, 2)
→ sqwrl:select(?p)

Here, the `sqwrl:size` operator will apply to each individual grouped set. In general, operators applied to grouped sets will automatically consider the grouping.

More complex groupings will require multiple grouping entities. For example, to build sets that contain the doses of each drug taken by each patient the sets must be grouped by both patients and drugs:

Patient(?p) ^ hasDrug(?p,?d) ^ hasDose(?d, ?dose) °
sqwrl:makeSet(?s, ?dose) ^ sqwrl:groupBy(?s, ?p, ?d)

Here, sets will be constructed for each patient and drug combination and the all doses for that combination will be added to them.

Aggregation operators can then be applied to these grouped sets if the elements have a natural ordering. For example, a query to return the average dose of each drug taken by each patient can be written:

Patient(?p) ^ hasDrug(?p,?d) ^ hasDose(?d,?dose) °
sqwrl:makeSet(?s, ?dose) ^ sqwrl:groupBy(?s, ?p, ?d) ^ sqwrl:avg(?avg, ?s)
→ sqwrl:select(?p, ?d, ?avg)

SQWRL's grouping mechanism dramatically expands the power of the language. This mechanism effectively allows queries to perform closure by selectively partitioning OWL entities into sets. It then supports an array of standard set operations on these partitioned entities, which allow it to answer very complex queries.

More complex queries can then be constructed by combining these grouping and aggregation mechanism with the earlier negation and disjunction mechanisms. For example, a query to list the average doses of drugs taken by patients that are on more than two drugs and where none of those drugs is a beta blocker or an anti-hypertensive can be written:

Patient(?p) ^ hasDrug(?p,?d) ^ hasDose(?d,?dose) ^
BetaBlocker(?bb) ^ AntiHypertensive(?ahtn) °
sqwrl:makeSet(?s1, ?dose) ^ sqwrl:groupBy(?s1, ?p, ?d) ^
sqwrl:makeSet(?s2, ?drug) ^ sqwrl:groupBy(?s2, ?p) ^
sqwrl:makeSet(?s3, ?bb, ?ahtn) ^
sqwrl:avg(?avg, ?s1) ^
sqwrl:size(?n, ?s2) ^ swrlb:greaterThan(?n, 2) ^
sqwrl:intersection(?s4, ?s2, ?s3) ^ sqwrl:isEmpty(?s4)
→ sqwrl:select(?p, ?d, ?avg)

This query illustrates the use of counting, aggregation, negation as failure and disjunction. As can be seen from this query, set operations can be applied to grouped and non grouped sets simultaneously and both types of sets can be used in the same operator.

3 Implementation

An implementation of SQWRL has been developed in the SWRLTab plugin [7] in Protégé-OWL. The implementation provides a graphical interface to edit and execute SQWRL queries and also provides a JDBC-like Java interface to execute SQWRL queries in Java applications. SQWRL queries also have access to all available SWRL built-in libraries, which provides a means of continuously expanding the power of the query language. A TBox built-in library, for example, has been developed and allows direct querying of OWL classes and properties. Custom built-in libraries can also provide functionality tailored to querying non OWL information sources. For example, an XML built-in library has been developed to query an OWL representation of XML documents. Built-in libraries for querying spreadsheets and RDF ontologies are also available in the SWRLTab.

An OWL reasoner that wishes to support SQWRL must obviously implement its built-in operators. These built-ins are somewhat unusual in that, unlike typical built-ins, they do not examine their arguments and evaluate to true if their arguments satisfy some predicate. Instead, these built-ins always evaluate to true. Their presence in a SWRL rule thus has no effect on the semantics of the rule. However, their behavior does not violate SWRL's semantics.

Of course, an implementation must also provide access to the results generated by each query. During query execution implementations of SQWRL built-ins must effectively accumulate the information passed to them and place it in a table-based data structure, with one table for each SQWRL query. Once a query has finished executing an implementation must also provide mechanisms to access the query result held in this data structure. An implementation is also responsible for SQWRL built-in argument error checking.

SQWRL queries can be serialized using the standard SWRL serialization mechanism. With the exception of the ° set antecedent separator, it also adopts a relatively conventional presentation syntax. Standard editor tools can thus work with it and may render the ° separator as the standard conjunction symbol. With minor modifications, tools can easily support this separator and can also ensure that set built-ins and operators on them only occur after it in a SQWRL query.

4 Conclusion

We have described a SWRL-based query language called SQWRL that provides a simple yet expressive language for performing queries on OWL ontologies. In addition to basic ontology query functionality, it provides an array of powerful set operators that support negation as failure, disjunction, counting, and aggregation functionality. A free open-source implementation of the core language features is included in Protégé-OWL 3.4.1 and the set operations will be available in an upcoming release.

Acknowledgments

This research was supported in part by grant 1R01LM009607 from the National Library of Medicine.

References

[1] SPARQL: http://www.w3.org/TR/rdf-sparql-query/

[2] E. Sirin, B. Parsia (2007). SPARQL-DL: SPARQL query for OWL-DL. *Third OWL Experiences and Directions Workshop (OWLED-2007)*.

[3] R. Fikes, P. Hayes, I. Horrocks (2005). OWL-QL - a Language for Deductive Query Answering on the Semantic Web. *Journal of Web Semantics*, 2(1).

[4] S. Bechhofer, R. Moller, P. Crowther (2003). The DIG Description Logic Interface. *Proceedings of the International Description Logics Workshop (DL 2003)*.

[5] SWRL Submission: http://www.w3.org/Submission/SWRL

[6] SWRL Built-ins: http://www.daml.org/2004/04/swrl/builtins.html

[7] SWRLTab Plugin: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab