# SWRL-IQ: A Prolog-based Query Tool for OWL and SWRL

Daniel Elenius

SRI International, Menlo Park, California, USA
elenius@csl.sri.com

**Abstract.** We present SWRL-IQ (SWRL Inference and Query Tool), a Protégé plug-in that allows users to create, edit, save, and submit queries to an underlying inference engine based on XSB Prolog. The tool distinguishes itself from other reasoning tools by a number of features, including goal-oriented backward-chaining reasoning, flexible constraint handling that allows for very declarative rules and queries, powerful SWRL extensions, and tracing and debugging features for explanation of reasoning results. Together, these features allow SWRL to be used as a powerful Logic Programming language that is tightly integrated with OWL ontologies. SWRL-IQ was motivated by the needs of a very complex problem domain: distributed military training and testing. The system is implemented in a flexible way to allow for different syntax front ends and reasoning back ends.

## 1 Introduction

SWRL-IQ[1] (Semantic Web Rule Language Inference and Query tool) is a plug-in for Protégé 3.x that allows users to create, edit, save, and submit queries to an underlying inference engine based on XSB Prolog[2]. The inference engine supports an expressive fragment of OWL, SWRL, and some extensions. The tool has a powerful set of features not found together (or at all) in other query and reasoning tools:

- Goal-oriented backward-chaining Prolog-style reasoning (as opposed to the forward-chaining paradigm used by most Semantic Web rule engines.
- Constraint-solving based on CLP(R) (Constraint Logic Programming with Reals). This allows for more declarative and powerful rules and queries.
- User-defined predicates with arbitrary arity.
- Powerful SWRL extensions for non-monotonic aggregation, limited higher-order logic, and more.
- Saving query results to XML or CSV format.
- Tracing and debugging of inference results.
- A simple-to-use Java Attachments mechanism.

Together, these features allow SWRL to be used as a powerful Logic Programming language that is tightly integrated with OWL ontologies.

[1] SWRL-IQ can be downloaded at protegewiki.stanford.edu/wiki/SWRL-IQ
[2] xsb.sourceforge.net

In Section 2 we briefly discuss our use cases that motivated the development of SWRL-IQ. Section 3 forms the bulk of this paper, and describes our system and how it differs from other reasoning engines. Section 4 discusses some related work. Section 5 discusses extensions and improvements that we have planned for the future.

## 2 Motivation

SWRL-IQ is part of a suite of solutions developed by SRI International to support military training and testing events, most recently in the Open Netcentric Interoperability Standards for Training and Testing (ONISTT) and Analyzer for Net-centric Systems Confederations (ANSC) programs, where we apply Semantic Web technologies to address problems in this domain [1].

These events make use of many distributed, heterogeneous systems and resources, such as simulation programs, virtual trainers, and live training instrumentation, connected to play different roles. The different systems are not usually designed to be used in this fashion, and the domain is replete with interoperability problems.

One of the most problematic areas is that of terrain representation [2]. For most training and testing purposes, it is crucial for the simulation software to have a realistic representation of the environment in which the events take place, including elevation data, imagery, and 3D models of trees and buildings. It is difficult to determine whether terrain data is adequate for a particular purpose, because it lacks sufficient metadata, and has often gone through many processing steps, some of which affect accuracy in subtle ways understood only by a handful of experts. To analyze such problems, we trace the processing history of terrain data in order to calculate the contribution to different types of error measures by each processing step, using SWRL rules. The expressivity we need for these kinds of problems goes beyond what existing Semantic Web tools provide. Thus, we created SWRL-IQ, the subject of this paper.

## 3 Architecture and Features

Here, we describe the architecture and features of SWRL-IQ. For a more thorough description, see the user manual bundled with the tool.

SWRL-IQ is a query answering system. As usual, the answer to a query consists of all the possible bindings for the variables in it[3]. SWRL-IQ does not answer questions about class subsumption (TBox reasoning), although TBox axioms are still taken into account in the query reasoning. The syntax for queries is the same as for *SWRL rule bodies*. The semantics of such a SWRL query is straightforward. In Section 4 we discuss the differences with SPARQL queries. SWRL-IQ supports most of the standard SWRL Built-ins (see the user manual

[3] There is also support for anonymous variables, i.e. variables whose bindings are not shown in the result. This is useful to avoid intermediate results cluttering up the display of the final results. Anonymous variables have an underscore in the beginning of the variable name, e.g., `?_x` instead of `?x`.

for full details). We describe the support for arithmetic, constraints, and lists in more detail below, as it goes beyond what might be expected.

SWRL-IQ is based on the Description Logic Programs (DLP)[3] approach to bridging Description Logics (DL) and Logic Programs (LP). Roughly speaking, DLP is the intersection of LP and DL, which corresponds to the definite equality-free Datalog Horn fragment of First-Order Logic (FOL). In this approach, DL sentences are translated to Horn clauses. For example, the subclass axiom $C \sqsubseteq D$ (where $C$ and $D$ are named classes) corresponds to the Horn clause $D(x) \leftarrow C(x)$. A large part of SWRL can also be supported by the DLP approach since SWRL rules already correspond to Horn clauses, assuming certain class constructors are not used in the rules.

SWRL-IQ uses the Horn clauses generated from OWL and SWRL directly as Prolog rules. The language is limited in several ways. As in [3] and OWL 2 RL, the language is assymetrical in that certain constructs are allowed in the "head" (or superclass) position only, and others in the "body" (or subclass) position only. For example, `someValuesFrom` and `unionOf` are disallowed in the head, and `allValuesFrom` is disallowed in the body. Negation and equality reasoning are not supported (`complementOf`, `disjointWith`, functional properties, cardinalities, `sameAs`, `differentFrom`). Supported features include domain and range, `hasValue`, and transitive and inverse properties.

The OWL part of the SWRL-IQ language is similar to OWL 2 RL (OWL 2 RL does not inherently include user-defined rules as in SWRL). The OWL 2 RL specification provides a rule set that can be used to implement the reasoning inside a rule language, whereas SWRL-IQ uses the Horn meaning of OWL and SWRL sentences directly as rules - in other words, it uses Prolog as the reasoner in a "native" way. This difference has some implications on the supported language constructs and reasoning. For example, OWL 2 RL supports some equality reasoning, whereas SWRL-IQ does not, and the OWL 2 RL approach supports TBox reasoning, whereas the SWRL-IQ approach only supports querying.

The back end of SWRL-IQ is XSB Prolog, which provides powerful goal-oriented reasoning. Combined with the flexible support for constraints and lists (Sections 3.2, 3.3), and our SWRL extensions (Section 3.4), SWRL-IQ enables a powerful Logic Programming style of specification. The *tabling* feature of XSB is used to avoid infinite loops that could otherwise occur in a naive translation of, for example, equivalent classes and properties. The SWRL built-ins are implemented directly in Prolog.

The main interface to SWRL-IQ is through a plug-in to Protégé 3. The reasoner can also be used programmatically, or through a Web Service interface, although these usage modes are not officially supported at this time. The high-level architecture is shown in Figure 1.

The OWL-Prolog translator shown in Figure 1 has several parts. It uses an intermediate representation of the knowledge base (KB), which is modeled on first-order logic (FOL). The purpose of this is to make the system as flexible as possible. It allows us to support and combine many different language syntaxes as long as they fall within first-order logic, by adding additional front ends. It

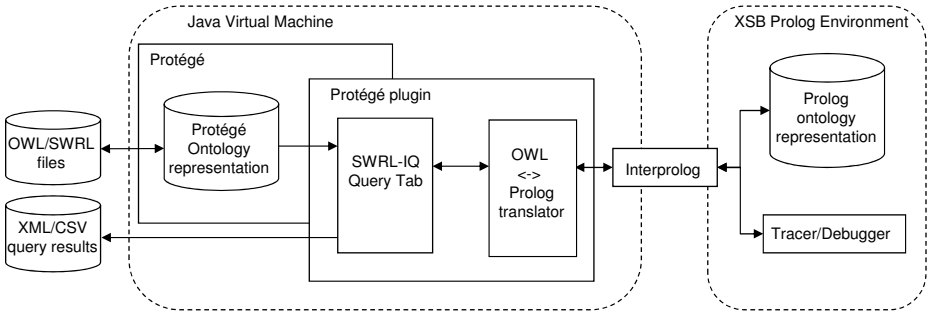


**Fig. 1.** Implementation architecture

also allows us to plug in different reasoners, as long as they operate on some fragment of first-order logic, by writing additional back ends. The FOL representation serves as the common nexus of all the different front and back ends. Going beyond the triple representation also makes it easy to support predicates with an arbitrary number of arguments (see Section 3.4). The downside of this architecture is that it is less integrated with the triple model. It is not easy to find out which triple(s) map to which Prolog rule(s). Incremental updates therefore become slightly more complicated (but not impossible; SWRL-IQ automatically reacts to changes in the KB in an incremental way). In addition, working with several different representations (triples, FOL, Prolog) makes the system more complex generally, and consumes more memory.

### 3.1 Interface

The Protégé-based user interface (Figure 2) has three main parts: The query instance browser, the query editor, and the query results panel. The query instance browser is used to create new query individuals, and to select among existing ones. The query editor to the right of the query instance browser is used to edit query content. The query results panel at the bottom of the screen shows the bindings for all the query variables once the query has been submitted, with one row in the table per solution. Queries with no variables just return "SUCCEEDED" or "FAILED". Results can also be saved to XML (we use the SPARQL XML results format[4] for this) and CSV (comma-separated values) files. The latter can be opened in spreadsheet tools and, for example, used to generate reports or graphs.

### 3.2 Arithmetic and Constraints

Many of the built-ins can generate numerical *constraints*, and are implemented to do so in a very flexible way, using a Prolog subsystem known as Constraint Logic Programming (CLP). Constraints are maintained throughout the query and simplified as much as possible. If any constraints cannot be simplified away, they are returned in the query result, along with the variable bindings. One consequence of this is that these built-ins can be used "backwards," and in

[4] `www.w3.org/TR/rdf-sparql-XMLres/`

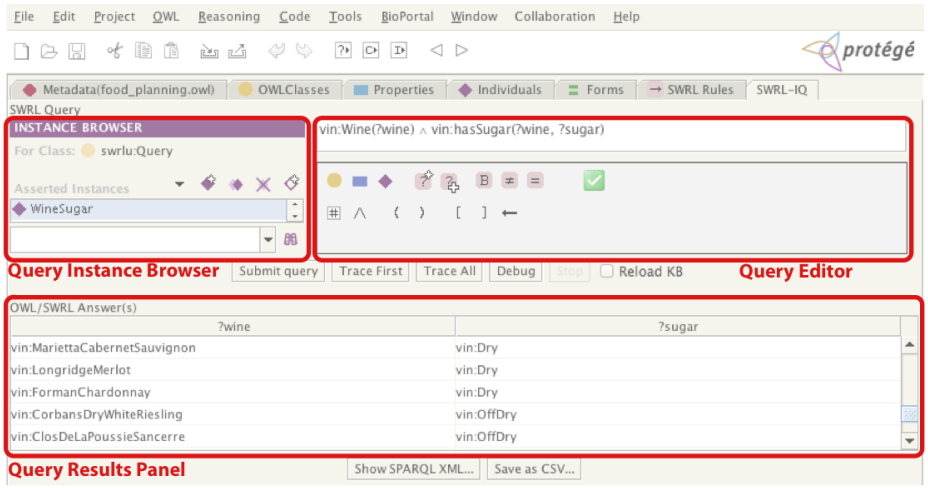


**Fig. 2.** SWRL-IQ Protégé user interface

effect to solve systems of equations. This makes rules that use these built-ins more declarative and powerful. The constraint mechanism is best understood through examples (Table 1).

| Query | Bindings | Constraints |
|---|---|---|
| `swrlb:equal(?x,3)` | ?x = 3 | none |
| `swrlb:notEqual(?x,3)` | ?x = ?Var0 | ?Var0 =\= 3 |
| `swrlb:lessThan(?x,3)` ∧ `swrlb:lessThan(?y,?x)` | ?x = Var0, ?y = ?Var1 | ?Var0 < 3.0<br>?Var1 - ?Var0 < 0.0 |
| `swrlb:lessThanOrEqual(?x,3)` ∧ `swrlb:greaterThanOrEqual(?x,3)` | ?x = 3 | none |
| `swrlb:pow(?_sq,?x,2)` ∧ `swrlb:multiply(10,?_sq,5)` | ?x = 1.4142135623730951,<br>?x = -1.4142135623730951 | none |
| `swrlb:pow(?_sq,?x,2)` ∧ `swrlb:multiply(10,?_sq,5)` ∧ `swrlb:greaterThanOrEqual(?x,0)` | ?x = 1.4142135623730951 | none |

**Table 1.** Examples of SWRL queries with the returned bindings and constraints

### 3.3 Lists

Our implementation of the built-ins for lists has a number of interesting features:

– `rdf:Lists` can be created on-the-fly in SWRL queries, using the syntax `rdf:List(m1,m2...)`, where `m1,m2...` are the list members. The same is true for SWRL rules in Protégé.

– The list built-ins can be used to *generate* lists, not just to check properties or retrieve values of existing lists.
– The reasoner can operate on (and return) partially instantiated lists, e.g., a list of three elements where the second element is unknown.
– Lists in the variable bindings returned by queries are presented in a different syntax, using square brackets for brevity (as in Prolog), e.g., `[m1,m2..]`. If the tail of the list is unspecified (i.e., the list has unknown size), the Prolog bar syntax is used, e.g., `[m1,m2|?tail]`.

These features are illustrated in Table 2.

| **Query** | **Bindings** |
|---|---|
| `swrlb:listConcat(?l,rdf:List(1,2),rdf:List(3,4))` | ?l = [1,2,3,4] |
| `swrlb:length(2,?l)` ∧ `swrlb:first(1,?l)` ∧ `swrlb:rest(?r,?l)` ∧ `swrlb:first(2,?r)` | ?l = [1,2] |
| `swrlb:length(2,?l)` ∧ `swrlb:member(1,?l)` | ?l = [1,?Var0],<br>?l = [?Var1,1] |
| `swrlb:first(1,?l)` | ?l = [1\|?Var0] |

**Table 2.** Examples of SWRL queries with lists

The fact that we can generate lists dynamically (unlike OWL individuals, which can *not* be generated by SWRL rules) means that we can use them as a general-purpose data structure. We have used this to implement matrix operations (e.g., for matrix multiplication) in SWRL, where matrices are represented as lists of lists. The flexibility of our built-ins (both for lists and for arithmetic) means that we can, for example, use the rule for adding matrices "backwards" to subtract matrices. The use of the list constructor `rdf:List` is an exception (the only one) to the function-free requirement of Datalog. It is possible to write rules using these list features that cause the reasoner to not terminate, but this is usually the result of a modeling error rather than an intended, useful rule.

### 3.4 SWRL Extensions

Sometimes SWRL itself is not enough. Perhaps the most significant limitation is that it is limited to unary and binary predicates (except for the fixed set of built-in predicates). When rules are used in a "programming" style or for complex or parameterized relationships, this is often not enough. One solution is to encode multiple arguments into an `rdf:List`, and then "unpack" the arguments inside the rule (using `swrlb:first` and `swrlb:rest`). However, this quickly becomes unwieldy, and the list-handling logic hides the real meaning of the rules. Thus, SWRL-IQ supports user-defined predicates with an arbitrary number of arguments. The way we define such predicates is to simply create new individuals of the `swrl:Builtin` class. These pseudo-builtins can then be used in SWRL rules and queries with any number of arguments. No actual "built-in" implementation in programmatic code is needed. Although our current approach to this is a bit of

a hack on the syntax level, in our experience n-ary predicates are indispensable for real-world use cases.

In general, our approach is to define all our predicates in SWRL, if possible. However, SWRL-IQ also provides a number of new built-in predicates that *are* implemented in code, because they go beyond the expressiveness of SWRL itself (even with n-ary predicates). We describe only some of these extensions here.

The **swrlex:ignore** predicate is deceptively simple – it takes any number of arguments and does nothing! This is needed because SWRL requires all variables that are in the rule head to also occur in the rule body. Sometimes such extra variables are unavoidable, for example in the "base case" rule of a recursively defined predicate[5].

The **swrlex:allKnown** predicate is an aggregation function. `swrlex:allKnown(?res,?x,?p,?x1,?x2,...)` returns a list `?res` containing all values for `?x` such that `p(?x1,?x2,...)` is satisfied, where `?x` is one of `?x1,?x2,...`. This is a non-monotonic extension, essentially because it depends on negation-as-failure (NAF)[6]. This is emphasized by the predicate's name: It returns all the values that are "known" (or can be inferred by the reasoner). Thus, `allKnown` provides a local closed-world assumption.

The **swrlex:apply** predicate gives us some "higher-order" expressiveness. `swrlex:apply(?p,?x1,?x2,...)` is satisfied when the atom with the predicate `?p` and arguments `?x1,?x2,...` is satisfied. This predicate lets us use predicates as arguments, and have variables that bind to predicates. This gives us a limited higher-order logic, similarly to lambda functions in functional programming. This allows us to write more generic rules instead of repeating almost-identical rules for different cases.

To illustrate how `ignore`, `allKnown` and `apply` might be used, consider a predicate **sort(?sl,?l,?p)** that sorts a list of any kind of entities, where `?sl` is the version of the list `?l` sorted in ascending order using the *ordering predicate* `?p`. For example, we can sort some numbers using `swrlb:lessThanOrEqual` as the ordering predicate:

```
swrlu:sort(?sl,rdf:List(4,17,3,20),swrlb:lessThanOrEqual)
```

returns

```
?x = [3,4,17,20]
```

We can also sort wines according to their price. Assuming that our wine ontology has a `hasPrice` predicate that links wines to their prices, we can define a predicate `wineCheaperThan` using a SWRL rule:

```
hasPrice(?w1, ?p1) ∧ hasPrice(?w2, ?p2) ∧
swrlb:lessThanOrEqual(?p1, ?p2) ⇒ wineCheaperThan(?w1, ?w2)
```

Then we can run the query

[5] It is possible to get around this requirement in other ways, e.g. using `owl:Thing(?x)`, but this does not work for data literals, and `ignore` also has the advantage that it can "swallow" any number of arguments using only one atom.

[6] We have not added a `not` predicate for direct use of NAF, but it would be possible to do so.

```
swrlex:allKnown(?wines,?wine,hasPrice,?wine,?price) ∧
swrlu:sort(?sortedWines,?wines,wineCheaperThan)
```

which will return a list of wines (that have a price) sorted by price, in the `?sortedWines` variable. Note the higher-order style of passing predicates as arguments. We used `allKnown` to produce a list of all known wines to use as an input to the `sort` predicate. The definition of `sort` depends on `ignore` and `apply`, and is shown in Figure 3. The key point is that we have to create only *one* sort predicate, and we can use it to sort *anything*.

| | |
|---|---|
| `swrlb:empty(?x)` ∧<br>`swrlex:ignore(?pred)`<br>⇒ `sort(?x, ?x, ?pred)` | `min_list(?m, ?y, ?pred)` ∧<br>`swrlb:first(?m, ?x)` ∧<br>`makeList(?ml, ?m)` ∧<br>`swrlb:listSubtraction(?r, ?y, ?ml)` ∧<br>`sort(?sr, ?r, ?pred)` ∧<br>`swrlb:rest(?sr, ?x)`<br>⇒ `sort(?x, ?y, ?pred)` |
| `swrlb:length(1, ?args)` ∧<br>`swrlb:first(?x, ?args)` ∧<br>`swrlex:ignore(?pred)`<br>⇒ `min_list(?x, ?args, ?pred)` | `swrlb:length(?len, ?args)` ∧<br>`swrlb:greaterThanOrEqual(?len, 2)` ∧<br>`swrlb:first(?frst, ?args)` ∧<br>`swrlb:rest(?rest, ?args)` ∧<br>`min_list(?rmin, ?rest, ?pred)` ∧<br>`minimum(?min, ?rmin, ?frst, ?pred)`<br>⇒ `min_list(?min, ?args, ?pred)` |
| `swrlex:apply(?p, ?x, ?y)`<br>⇒ `minimum(?x, ?x, ?y, ?p)` | `swrlex:apply(?p, ?y, ?x)`<br>⇒ `minimum(?y, ?x, ?y, ?p)` |

**Fig. 3.** SWRL rules for the `sort` predicate (some auxiliary rules omitted)

Other extensions provide simple but useful utilities. For example, **swrlex:pi** returns the constant $\pi$, and **swrlex:string_number** allows us to convert between numbers and their string representations.

### 3.5 Java Attachments

Like many reasoning systems, SWRL-IQ has a mechanism for calling arbitrary programming code from within the reasoner. This is traditionally called procedural attachments. In our implementation, the code to be called has to be written in Java. We call our mechanism *Java Attachments*.

Most computations can be performed in SWRL, especially with the extensions described above. However, there are situations where it does not make sense to use SWRL rules. One such case is when a large and complicated function has already been implemented in programming code, and perhaps certified. Re-implementing and re-certifying in SWRL may not be feasible. Indeed, if the source code is not available and the algorithm is not known, there is no choice in the matter.

The hook to the Java attachments mechanism is the built-in predicate **callJavaStaticMethod(?ret,?cls,?mthd,?arg1,?arg2,...)** where `?ret` is the return value, `?cls` is a string representing the class, including the full package name, `?mthd` is a string for the static method to call, and `?arg1,?arg2,...` are the arguments to the method. For example,

```
swrlex:callJavaStaticMethod(?r,"java.lang.Math","random")
```

returns

```
?r = a random number from 0.0 to 1.0
```

As seen here, any static method of the standard Java library can be used. However, user-defined or third-party code can also be called as explained, by simply putting its jar file in the classpath. Note that only basic types can be used for the arguments (ints, strings, lists, and so on). There is currently no way to pass RDF resources (individuals, classes, and properties) to Java Attachments.

### 3.6 Tracing and Debugging

The tracing and debugging features of SWRL-IQ are used for the explanation of reasoning results. They answer the questions "why?" or "why not?" did a result happen. The user interface for both is similar.

Tracing is used to explain how and why a query succeeded. It can be used, for example, to learn why an unexpected result happened, or to check that some SWRL rules got used in the intended way. The trace window contains a tree showing one or more *proofs* of the given query. Figure 4 shows an example trace that uses some of our rules for virtual terrains to calculate an elevation error.

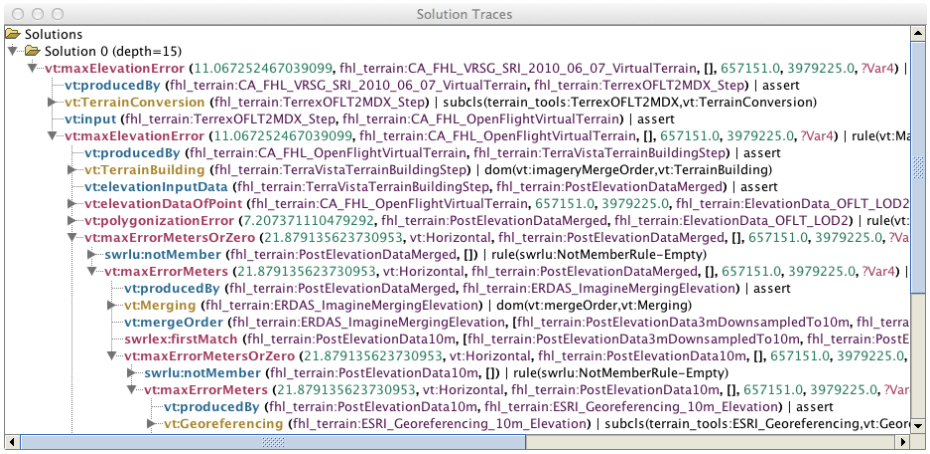


**Fig. 4.** Partially expanded trace of a query about virtual terrain elevation error

Each "solution" node shows the *depth* of its proof, which can sometimes be interesting. This corresponds to the depth of the tree starting at the solution

node. Each solution node has one or more child nodes, which we call *atom nodes* - one atom node for each atom in the query. An atom node consists of an *atom* followed by a *justification* after the vertical bar, and possibly a number of child atom nodes. The atom nodes are color coded according to the type of the atom (class, object property, etc.) and its arguments (individual, literal, etc.).

The meaning of an atom node, along with its justification and its child atom nodes, is that the atom was proved, using the inference rule given in the justification, and (if applicable) the child atoms, which in turn have their own justifications, and so on. The trace in Figure 4 uses SWRL rules (`rule`), subclass (`subcls`) and domain (`dom`) axioms, asserted facts (`assert`), and computed properties (`comp`, i.e. from SWRL built-ins) as its justifications. The tree structure directly follows the operational semantics of the Prolog reasoning. Debug trees show all *attempted* proofs rather than actual proofs, and add two types of nodes to the proof trees: FAILED and ABORTED nodes.

The tracer and debugger are both implemented in a straightforward way as Prolog "meta-interpreters"; they come almost for free with our Prolog-based approach. The elements of the proof trees are relatively easy to understand, but the user can get overwhelmed by the size of the proofs. Explanation of inference results is an area of ongoing research. We consider the existing tracing feature to be a rudimentary form of explanation, and we plan to do more work in this area.

### 3.7 Performance

SWRL-IQ supports reasoning on a fairly expressive language fragment. This means that scalability is sacrificed to some extent. The type of reasoning goes beyond what can be achieved with disk-based methods [4], i.e. it is all in-memory and therefore limited by the amount of memory available. Our own use cases are fairly small – our largest knowledge base contains about 40,000 triples. One bottleneck of the system is the time it takes for XSB to build its tables. For our 40,000-triple KB, this is about 10 seconds on a typical PC. However, this has to be done only once. Further changes to the KB are tabled incrementally and very quickly. Query answering time on this scale of KB is typically less than one second. Comparisons with other reasoners is not meaningful due to the vastly different expressivity of SWRL-IQ compared to most RDF reasoners.

## 4 Related Work

Semantic Web querying usually means using the SPARQL language. We have chosen to use SWRL syntax for our queries for several reasons. First, SPARQL is closely tied to the triple model. As we explained in Section 3, our reasoner does not operate on the triple level. The SPARQL syntax would not allow for predicates with arbitrary arity, which are critical to the expressiveness of SWRL-IQ. On the other hand, SPARQL allows for queries about schema (TBox) information – e.g., by using `rdfs:subclassOf` in the predicate position of a triple pattern. SWRL-IQ cannot handle such queries because the schema is implicit in the Prolog translation and not explicitly represented. Furthermore, it is natural to

use the same language for rules and queries. SWRL rules relate to SWRL queries in the same way that Prolog rules relate to Prolog queries. More generally, one might say that SPARQL is a query language for RDF triple models, whereas SWRL queries operate on a higher semantic level of OWL+SWRL ontologies.

Protégé provides a query language (and interface) called SQWRL [5]. This requires the Jess reasoning engine[7] (which is not free for non-academic users). This solution differs in many ways from ours: It uses forward chaining instead of backward chaining, it does not support the SWRL built-ins as flexibly as SWRL-IQ does, does not return constraints, and so on.

There are many other OWL and RDF reasoning engines. We discussed the main differences between our approach and other engines in Section 3. Jena[8] includes a rule engine[9] that combines forward chaining with tabled backward chaining (like XSB). It is closely tied to the triple model, has its own proprietary rule language, and does not support arbitrary arities and many other features of SWRL-IQ. In previous work [6], we developed an OWL+SWRL reasoner based on narrowing and rewriting logic, which allowed even more expressive handling of constraints, as well as equality reasoning, but at the price of efficiency.

## 5 Future Work

With SWRL-IQ, we have a very powerful and flexible query tool for OWL and SWRL. However, it is still very difficult to create, edit, understand, and validate SWRL rules and queries, even for ontology experts. Existing rule editors and visualization tools like GrOWL [7] and Axiomé [8] offer partial solutions, but fail to scale to the complexity and size of our rule bases. We are concerned with projects that involve large amounts of detailed information from subject-matter experts (SMEs). That knowledge has to be encoded in OWL and SWRL. Furthermore, it must be possible to convince the SME that the encoding is correct, i.e., to validate the formal knowledge. Our goal is to have an integrated system for all knowledge engineering tasks. In a follow-up project, we are planning to address some shortcomings of current tools. Features in the scope of this work include rule testing, provenance, type checking, rule templates, search, natural language, rule dependency analysis, improved debugging and tracing, rule syntax improvements, views and perspectives, and rule analysis.

We are also considering supporting Protégé 4 and OWL 2. Due to its flexible implementation, it would be easy to add support for the OWL 2 constructs that still fall within the expressiveness of the reasoning engine. However, a significant drawback of Protégé 4 is that it, at the time of writing, has less support than Protégé 3 for creating and editing SWRL rules. In addition, the combination of SWRL with OWL 2 has not been defined (SWRL is an extension to OWL 1). Another possibility is OWL 2 + RIF, but again, there is no existing editing tool for RIF rules, and RIF also has the disadvantage of being less tightly integrated with OWL, although the combined OWL 2 + RIF semantics have

[7] `http://www.jessrules.com`
[8] `http://incubator.apache.org/jena`
[9] `http://incubator.apache.org/jena/documentation/inference/`

been specified[10] and an RDF embedding of RIF is in the works[11]. In any case, these language changes would not fundamentally change the reasoning power of SWRL-IQ, only the pragmatics of syntax and tools.

## Acknowledgments

The work described in this paper was carried out at the SRI International facilities in Menlo Park, California, and was funded in part by the U.S. Department of Defense, TRMC (Test Resource Management Center) T&E/S&T (Test and Evaluation/Science and Technology) Program under NST Test Technology Area prime contract N68936-07-C-0013. The authors are grateful for this support and sthank Gil Torres, NAVAIR, for his leadership of the Netcentric System Test (NST) technology area, to which the ANSC project belongs. We also acknowledge ODUSD/R/RTPP (Training Transformation) for its sponsorship of the associated ONISTT project. Reg Ford and Susanne Riehemann collaborated on SWRL-IQ. Finally, David Warren and Terrance Swift provided expert guidance and bug fixes for XSB Prolog, and Miguel Calejo did the same for InterProlog.

## References

1. Elenius, D., Martin, D., Ford, R., Denker, G.: Reasoning about Resources and Hierarchical Tasks Using OWL and SWRL. In A. Bernstein et al., ed.: 8th International Semantic Web Conference, ISWC 2009. Lecture Notes in Computer Science, Springer (2009) 795–810
2. Riehemann, S., Elenius, D.: Ontological Analysis of Terrain Data. In Liao, L., ed.: COM.Geo. ACM International Conference Proceeding Series, ACM (2011)
3. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description Logic Programs: Combining Logic Programs with Description Logic. In: Proc. 2nd International Semantic Web Conference (ISWC2003). (2003)
4. Yahya, M., Theobald, M.: D2R2: Disk-Oriented Deductive Reasoning in a RISC-Style RDF Engine. In Olken, F., Palmirani, M., Sottara, D., eds.: RuleML America. Volume 7018 of Lecture Notes in Computer Science. (2011) 81–96
5. O'Connor, M.J., Das, A.K.: SQWRL: A Query Language for OWL. In Hoekstra, R., Patel-Schneider, P.F., eds.: Proc. 5th International Workshop on OWL: Experiences and Directions (OWLED 2009). CEUR Workshop Proceedings (2009)
6. Elenius, D., Denker, G., Stehr, M.O.: A semantic web reasoner for rules, equations and constraints. In Calvanese, D., Lausen, G., eds.: RR. Volume 5341 of Lecture Notes in Computer Science., Springer (2008) 135–149
7. Krivov, S., Williams, R., Villa, F.: GrOWL: A Tool for Visualization and Editing of OWL Ontologies. Web Semantics **5** (2007) 54–57
8. Hassanpour, S., O'Connor, M.J., Das, A.K.: Axiomé: A Tool for the Elicitation and Management of SWRL Rules. In Hoekstra, R., Patel-Schneider, P.F., eds.: Proc. 5th International Workshop on OWL: Experiences and Directions (OWLED 2009). CEUR Workshop Proceedings (2009)

[10] `http://www.w3.org/TR/rif-rdf-owl/`
[11] `http://www.w3.org/TR/rif-in-rdf/`