January 16, 2012
Version 1.0

SWRL-IQ User Manual

Prepared by
Daniel Elenius, Susanne Riehemann

333 Ravenswood Avenue • Menlo Park, California 94025-3493 • 650.859.2000 • www.sri.com

Table of Contents

LIST OF TABLES

LIST OF FIGURES

1 OVERVIEW

SWRL-IQ (Semantic Web Rule Language Inference and Query tool) is a plugin for Protégé 3.x that allows users to edit, save, and submit queries to an underlying inference engine based on XSB Prolog. The tool has a number of features that distinguishes it from other query and reasoning tools, such as the Protégé SQWRLQueryTab supported by the Jess rule engine:

- Goal-oriented backward-chaining Prolog-style reasoning (as opposed to the forward-chaining paradigm used by Jess and the Protégé SWRL Bridge framework).
- Constraint-solving based on CLP(R) (Constraint Logic Programming with Reals). This allows for more declarative and powerful rules and queries.
- Saving queries to XML or CSV format (the SQWRLQueryTab saves to CSV only).
- Tracing and debugging inference results.
- Exporting query results in different formats.
- No dependency on proprietary or closed-source components. Uses XSB Prolog, which is freely available under the LGPL license.

SWRL-IQ also has powerful SWRL extensions and a Java procedural attachment mechanism, similar to what the Jess query tab supports.

1.1 Project Home Page and Mailing List

The SWRL-IQ home page is `http://www.onistt.org/display/swrliq/`.

The user mailing list is swrl-iq-users@lists.esd.sri.com. If you wish to subscribe, send a message to swrl-iq-users-request@lists.esd.sri.com with the word "subscribe" as subject. You will receive an auto response that includes a link to confirm the subscription. You will then receive an email welcoming you to the mailing list.

1.2 A Quick Tour

The rest of the Overview section provides a quick tour of the tool to illustrate some of its capabilities. The following sections provide a more thorough reference manual, covering all the tool's features.

All example queries can be seen and run in `SwrlIq/owl/food/food_planning.owl`.

Example query that uses only information directly expressed in OWL

`vin:Wine(?wine)` $\wedge$ `vin:hasSugar(?wine, ?sugar)`

returns all the wines and their `hasSugar` properties (Dry, OffDry, or Sweet), as shown in Figure 1.

Example query using a pseudo-builtin defined by a SWRL rule

`hasPrice(?wine, ?price)` $\wedge$ `withinRange(?price,10, 50)`

returns all the wines that have an average price between $10 and $50 according to `http://www.wine-searcher.com/` as in Figure 2. This query uses the "pseudo-builtin" (see Section 3.1.1) 3-argument predicate `withinRange` defined by the SWRL rule

```
swrlb:greaterThanOrEqual(?x, ?y) ∧
swrlb:lessThanOrEqual(?x, ?z)
→
withinRange(?x, ?y, ?z)
```

Example trace
You can see how the rule is used when you click the "Trace All" button, as shown in Figure 3. There is more information about tracing in Section 3.2.

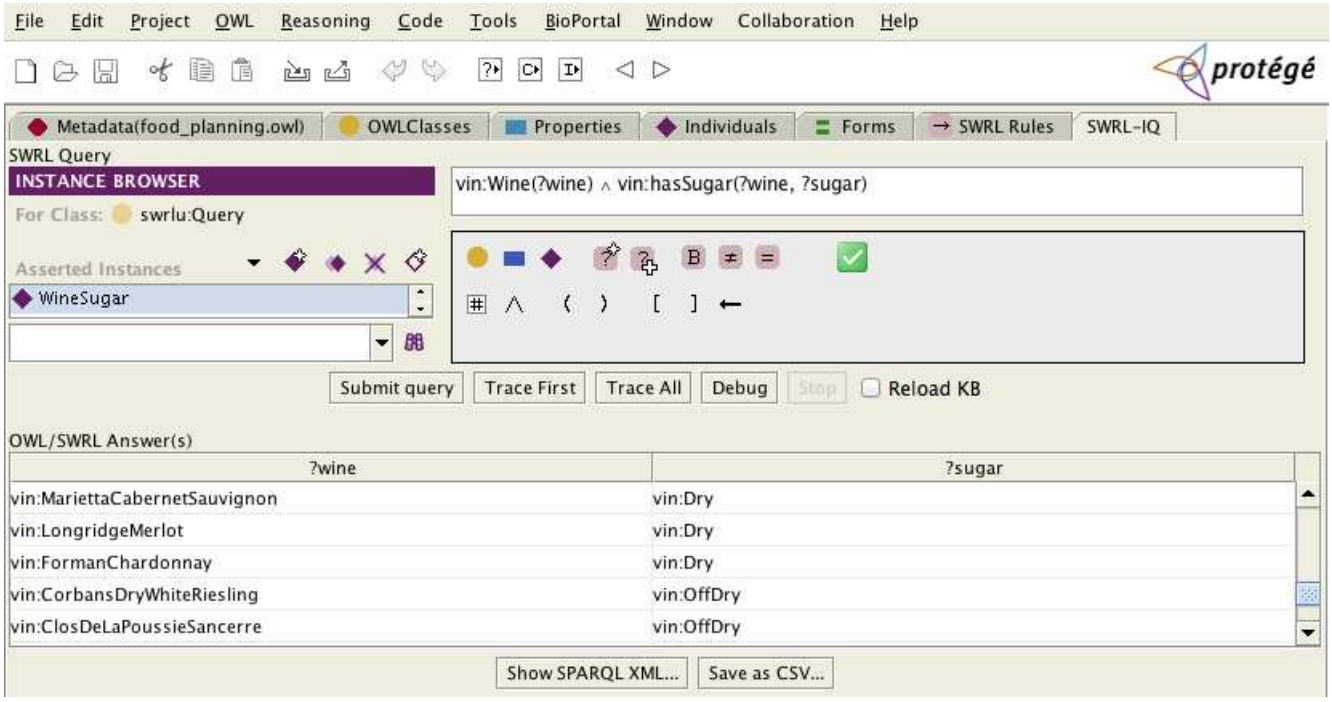

Fig. 1. Wine sugar query

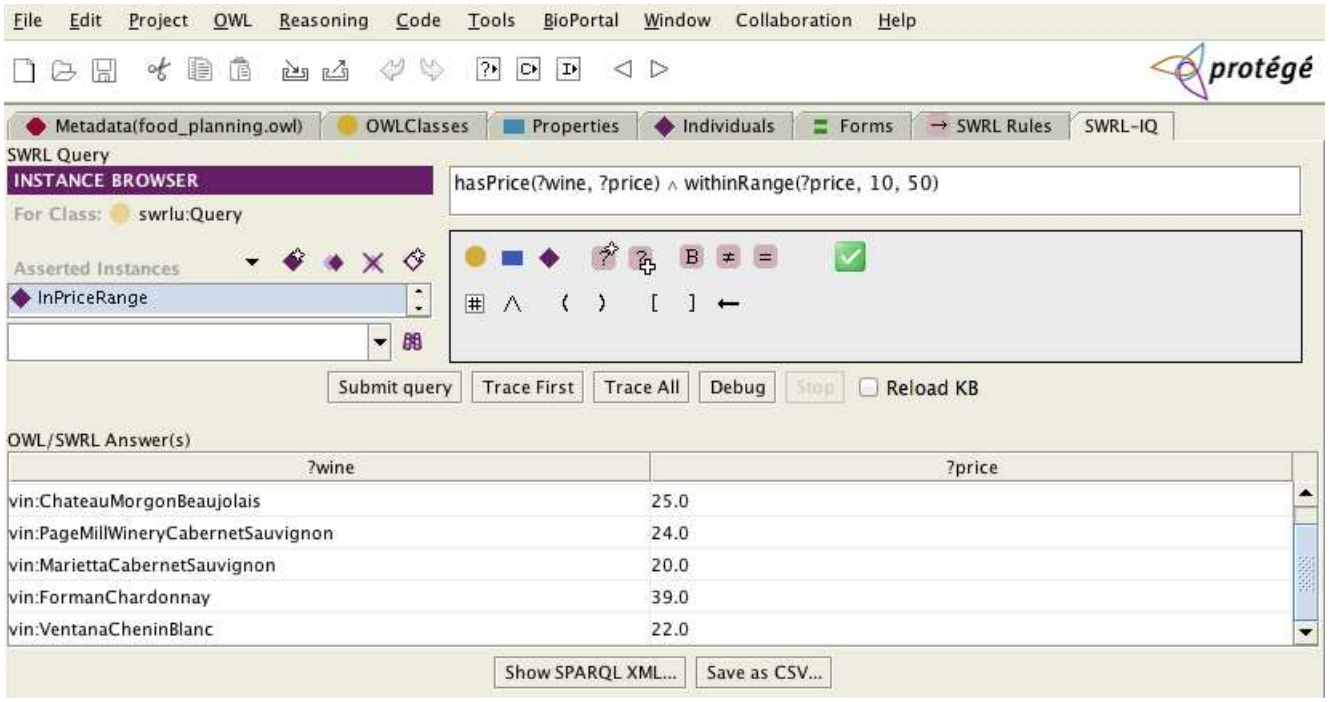

Fig. 2. Wine price range query

Fig. 3. Price range query trace

Example of saving results to XML and transforming to HTML

`vin:Wine(?wine)` ∧ `vin:hasMaker(?wine, ?maker)` ∧ `vin:hasColor(?wine, ?color)` ∧ `vin:madeFromGrape(?wine, ?grape)` ∧ `vin:hasSugar(?wine, ?sugar)` ∧ `hasPrice(?wine, ?price)` ∧ `priceReferenceURL(?wine, ?url)` ∧ `bottleImageURL(?wine, ?label)`

returns various properties of the wines, as show in in figure 4. The query results can be saved to an XML file and transformed to HTML (or other formats) using XSL transformations, as shown in Figure 5. For more details, see Section 4.1.

Example of saving results to CSV and graphing

Query results can be saved to a `.csv` (comma separated values) file and graphed in your favorite spreadsheet software, as shown in Figure 6. See Section 4.2 for more detail.

Example using reversible math

Many math builtins generate numerical constraints that are combined and if possible simplified away later. This makes them reversible and able to solve systems of equations. For example, instead of using `swrlb:pow(?x,3,2)` the usual way to determine what the result of 3^2 is, it can also be used to compute square roots by providing the 'output' argument and leaving one of the 'input' arguments as a variable. The query `swrlb:pow(9,3,?x)` returns `?x = 2.0`. See Section 5.1 for more detail.

Example using allKnown

`swrlex:allKnown(?regions, ?_region, vin:locatedIn, vin:BancroftChardonnay, ?_region)`

produces a list of all regions satisfying the transitive `vin:locatedIn` property, as shown in Figure 7. The `allKnown` predicate is used to aggregate query results into a list, and is described in more detail in Section 6. Also note the use of the anonymous variable `_region`, the values of which we are not interested in (see Section 3.1.2).

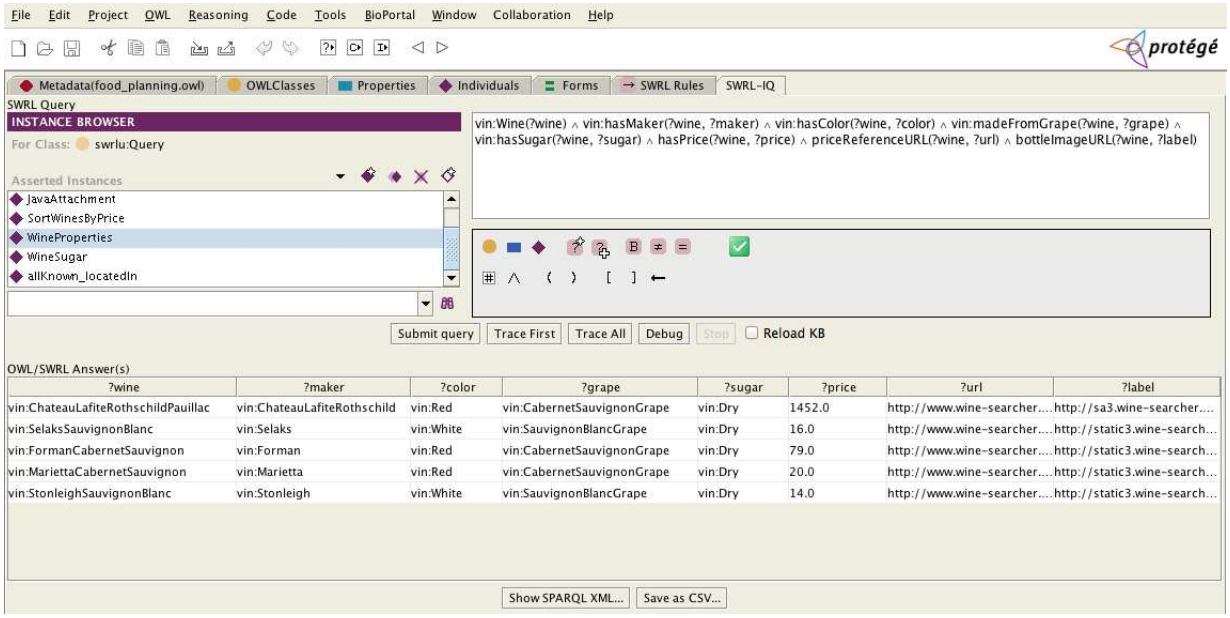

Fig. 4. Wine Properties Query result in Protege

Example using a predicate as an argument

`swrlex:allKnown(?_listOfWinePricePairs, rdf:List(?_wine, ?_price), hasPrice, ?_wine, ?_price) ∧ swrlu:sort(?sortedListOfWines, ?_listOfWinePricePairs, winePairCheaperThan)`

produces a list of the lists of all the wines and their prices sorted by price, as shown in Figure 8 (hover the mouse pointer over the result to see the blue box tool-tip). The sort predicate is discussed in Section 7. It takes a predicate (here `winePairCheaperThan`) as an argument, and sorts the list according to the ordering defined by that predicate.

Example using a Java attachment

`swrlex:callJavaStaticMethod(?r, "java.lang.Double", "toHexString", 14.432)`

returns the result of executing the given Java method, as shown in Figure 9. Any static method from the standard Java library can be used, and users can also call their own code. See Section 8 for more detail.

query_result.html

file:///Swrliq/html/query_result.html

Google

wine	maker	color	grape	sugar	url	price	label
ChateauLafiteRothschildPauillac	ChateauLafiteRothschild	Red	CabernetSauvignonGrape	Dry	Reference	$1452	
SelaksSauvignonBlanc	Selaks	White	SauvignonBlancGrape	Dry	Reference	$16	
FormanCabernetSauvignon	Forman	Red	CabernetSauvignonGrape	Dry	Reference	$79	
MariettaCabernetSauvignon	Marietta	Red	CabernetSauvignonGrape	Dry	Reference	$20	
StonleighSauvignonBlanc	Stonleigh	White	SauvignonBlancGrape	Dry	Reference	$14	

Fig. 5. Query result in html (using an xsl transformation)

	wine	maker	color	grape	sugar	price	url	label
1	vin:ChateauLa	vin:ChateauLa	vin:Red	vin:CabernetS	vin:Dry	1452	http://www.wi	http://sa3.wine-
2	vin:SelaksSau	vin:Selaks	vin:White	vin:Sauvignon	vin:Dry	16	http://www.wi	http://static3.wi
3	vin:FormanCa	vin:Forman	vin:Red	vin:CabernetS	vin:Dry	79	http://www.wi	http://static3.wi
4	vin:MariettaCa	vin:Marietta	vin:Red	vin:CabernetS	vin:Dry	20	http://www.wi	http://static3.wi
5	vin:StonleighS	vin:Stonleigh	vin:White	vin:Sauvignon	vin:Dry	14	http://www.wi	http://static3.wi

Fig. 6. Query result saved to CSV and graphed

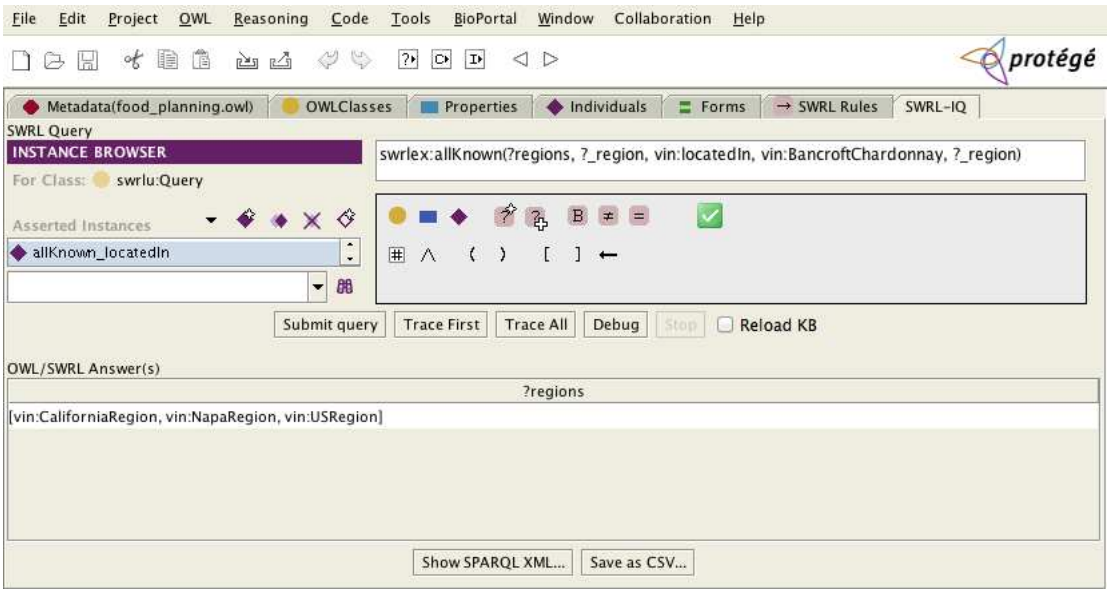

Fig. 7. All known regions query

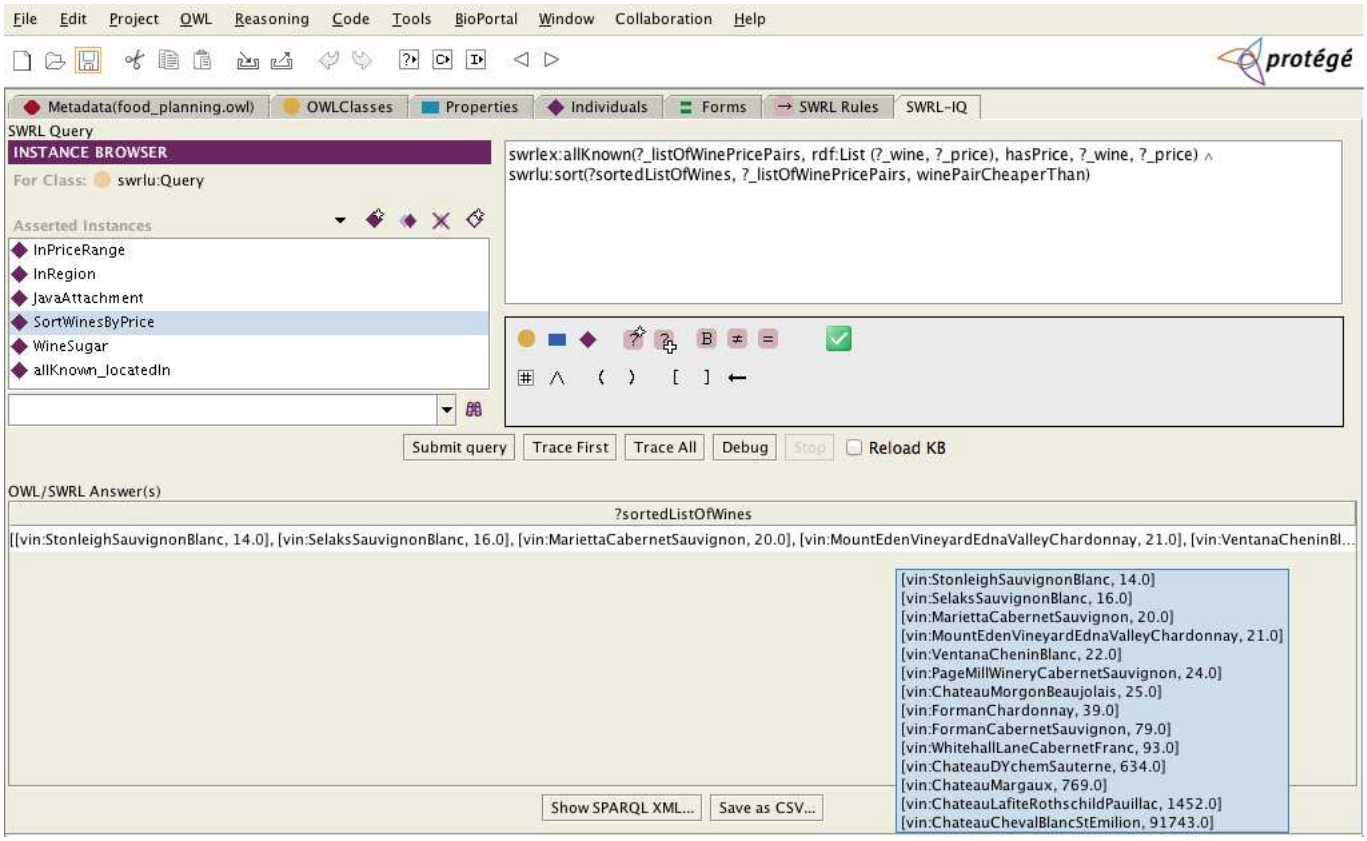

Fig. 8. Sort wines by price query

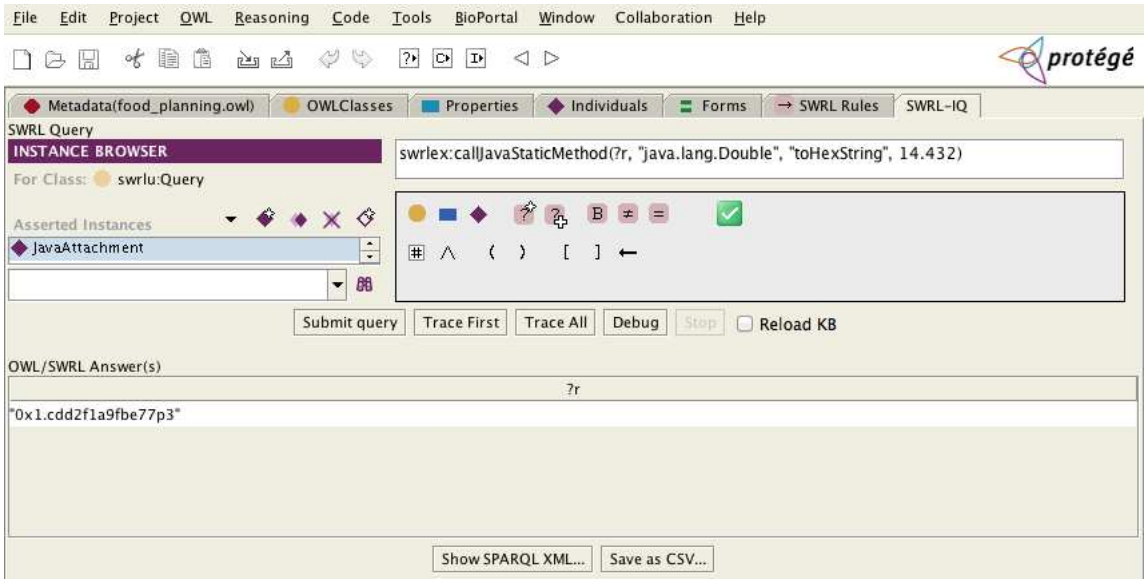

Fig. 9. Java attachment

2 INSTALLATION

2.1 Unzipping the plugin

SWRL-IQ is distributed in binary form, as a zipped archive. The first step of installing the tool is to unzip this archive. This will create a folder named `SwrlIq` which contains this manual, some sample ontologies, a sample java attachment, and a `swrliq-plugin.zip` file. Unzip `swrliq-plugin.zip` into the Protégé `plugins` directory. This creates a `com.sri.swrltab` directory with a few files in it.

2.2 Installing XSB

SWRL-IQ needs a working installation of XSB Prolog[1] to work. The main XSB site does not supply sufficiently up-to-date binaries, so we provide instructions below for building it from the source code. We also provide binaries for Windows at `http://www.onistt.org/display/SWRLIQ`. If you get those, you can skip sections 2.3 through 2.6.

2.3 Getting the Source Code

First you need to get the XSB source code, either the latest stable release at `http://xsb.sourceforge.net/downloads/downloads.html` or the latest version from CVS. To get it from CVS, enter the commands

```
cvs -d:pserver:anonymous@xsb.cvs.sourceforge.net:/cvsroot/xsb login
```

Hit enter when asked for password. Then,

```
cvs -z3 -d:pserver:anonymous@xsb.cvs.sourceforge.net:/cvsroot/xsb co -P XSB
```

This should give you a `XSB` directory.

2.4 Compiling XSB on Mac/Linux

In a shell window, go into the `XSB/build` directory, and enter

```
./configure
./makexsb
```

You should now be able to run `XSB/bin/xsb`. If there are errors, first make sure you have the latest versions of the build tools (e.g., Xcode on Mac).

2.5 Compiling XSB on Windows

First you need to install Microsoft Visual C++ Express, which is freely available at `http://www.microsoft.com/express/vc/`. Then install the Microsoft Windows SDK, from `http://msdn.microsoft.com/en-us/windows/bb980924.aspx`. Next, follow the instructions below, depending on the type of machine that you are compiling for.

2.5.1 32-bit Windows

In a command window, run

```
C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\BIN\vcvars32.bat
```

Then, in `XSB\build`, run

[1] `http://xsb.sourceforge.net/`

```
makexsb.bat
```

You should now be able to run

```
XSB\config\x86-pc-windows\bin\xsb
```

2.5.2 64-bit Windows

In a command window, run

```
C:\Program Files\Microsoft SDKs\Windows\v7.1\bin\SetEnv.cmd
```

Then, in `XSB\build`, run

```
makexsb64.bat
```

You should now be able to run

```
XSB\config\x64-pc-windows\bin\xsb
```

2.5.3 Cygwin

Compiling under Cygwin basically works the same as compiling under Linux/Mac, with one caveat: The space in the `C:\Program Files` directory which usually is part of the `JAVA_HOME` environment variable can cause some problems. To solve this, introduce extra (escaped) quotes before running `configure`, e.g.

```
export JAVA_HOME="\"C:\Program Files\Java\jdk1.6.0_24\""
```

Note that Cygwin only supports 32 bits.

2.6 Compiling CLPR

Go to `XSB/packages`
Start XSB (e.g. `..\bin\xsb` or `../bin/xsb`)
Enter the command
`[clpr].`
Exit XSB by using the command `halt.`
Go to `XSB/packages/clpqr`.
Start XSB again (e.g. `..\..\bin\xsb` or `../../bin/xsb`)
Enter these commands:
`[clpr_make].`
`make_clpr.`

2.7 Setting the XSB environment variable

SWRL-IQ needs to know where to find XSB. This is done by setting up an `XSB` environment variable pointing to the XSB executable. However, the variable must *not* point to the `XSB/bin/xsb` or `XSB/bin/xsb.bat` file. This file is a script that in turn calls the real executable, which is located in `XSB/config/ARCH/bin/xsb`, where `ARCH` is an architecture-dependent directory name. For example, on a 64-bit Linux system, the architecture directory is usually called `x86_64-unknown-linux-gnu`.

To set up the XSB environment variable on Windows 7,

1. Click on the Windows button in the lower left corner.
2. In the search field, type “environment”.
3. Click on “Edit the system environment variables”.

4. Click the "Environment Variables" button.
5. Click one the the "New" buttons, depending on whether you want to make this a system-wide setting or a user setting.
6. Enter "XSB" as the variable name and the appropriate executable (as discussed above) as the variable value.

On Linux and Mac, one simple way to set up the XSB variable is to edit the `run_protege.sh` script in the main Protégé folder. Add a line

```
export XSB=PATH
```

before the last line of the file, where `PATH` is the appropriate path to the executable. For example, the whole line might be

```
export XSB=/homes/elenius/XSB/config/x86_64-unknown-linux-gnu/bin/xsb
```

2.8 Importing SWRL Utilities

SWRL-IQ comes with a few `.owl` files. One of them is `swrl-utilities.owl`. This file needs to be imported in order to use SWRL-IQ. If it is not, the SWRL-IQ tab will show a text reminding the user of this. It contains the definition of the `swrlu:Query` class, which is used to save queries. It also provides a small library of useful predicates (see Section 7), and imports `swrl-extensions.owl`, which provides a number of very useful new SWRL "built-ins," (see Section 6).

2.9 Verifying the Installation

SWRL-IQ is a so-called Tab Widget. To use it, the widget must first be enabled under the Project->Configure menu item. Checking the box for "SwrlIQTab" should make a new tab with that title appear in Protégé. To test that the query engine works, create a new query individual by clicking the "create instance" button in the instance browser on the top left of the SWRL-IQ tab. Then enter a simple query into the query text field, e.g.,

```
swrlb:add(?x,3,2)
```

An answer to the query should appear on the bottom of the tab. Note that the *first* query always takes a little while. This is because the entire knowledge base has to be loaded into the Prolog back-end. Subsequent queries are much faster. This is discussed in more detail later in this manual.

If this does *not* work, look in the console window to see if there is any error message or Java exception. If you need help solving the problem, send email to swrl-iq-users@lists.esd.sri.com.

3 BASIC USAGE

SWRL-IQ is a plugin to Protégé, and uses some of the same user interface elements. We recommend that users who are new to Protégé first familiarize themselves with it. The Protégé web site contains a number of guides and tutorials[2]. In addition, learning how to use Protégé's SWRL rules tab[3] will help in understanding our SWRL query editor.

3.1 Queries

The SWRL-IQ user interface has three main parts: The query instance browser, the query editor, and the query results panel (see Figure 10). The query instance browser is a standard Protégé component. Most importantly, it allows you to create new query individuals, and to select among existing ones. Re-naming a query currently requires double-clicking on a query individual, which brings up the standard Protégé individual editor. For editing the actual query content, use the query editor to the right of the query instance browser.

Example: a simple query

`vin:Wine(?wine)` ∧ `vin:hasSugar(?wine, ?sugar)`

This query returns all the wines and their `hasSugar` properties (Dry, OffDry, Sweet), as shown in Figure 10.

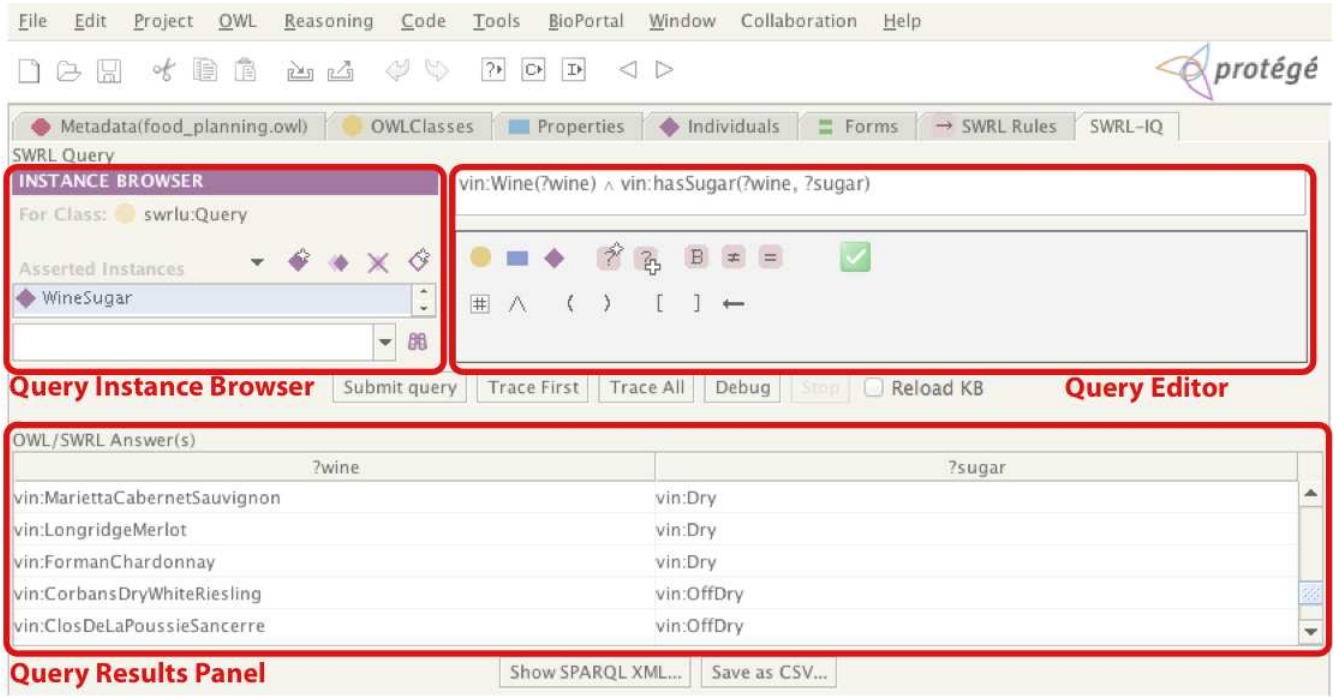

Fig. 10. Wine sugar query & Query UI

Note that you *have to* have a query individual to edit before you can change the content in the query editor. Sometimes you just want to try different queries without saving them. This is most easily done by just creating one "scratch pad" query individual for all such queries. The content of the scratch pad query

[2] `http://protege.stanford.edu/doc/users.html`
[3] `http://protege.cim3.net/cgi-bin/wiki.pl?SWRLEditorFAQ`

in the query editor can be deleted before saving the OWL file so that it does not get saved unintentionally (although saving it is not harmful, other than making the OWL file slightly larger).

The query editor works almost exactly like the Protégé SWRL rule editor. In terms of the user interface, the main difference is the lack of the implication symbol ($\rightarrow$). In other words, editing a query is much like editing the body of a SWRL rule.

The main purpose of a query is to find values for all the variables in the query such that the query is satisfied with respect to the current ontology, rules, and facts in the knowledge base.

Once a query is ready to run, click the "Submit query" button (the "Trace First," "Trace All," and "Debug" buttons are discussed in Section 3.2). There are three types of possible answers to the query: "SUCCEEDED", "FAILED", or a table of variable bindings. The first type can only happen when there are no variables in the query (or when all variables are anonymous, see section 3.1.2).

The variable bindings table shows one row in the table for each solution to the query. The table header shows the variable names, and the values in each column are the different bindings for the variable in question. Double-clicking on a value in a cell containing an RDF resource opens up the standard Protégé individual editor for that resource (double-clicking on numbers and strings does nothing). Clicking on a column header sorts the table alphabetically according to the entries in that column.

Sometimes a query (or trace/debug) attempt takes a very long time. In fact, with recursive SWRL rules, it is possible to create situations where the reasoning engine can run forever. In such cases, the "Stop" button can be used to abort what the reasoner is doing and make it available for new queries.

The "Show SPARQL XML" and "Save as CSV" buttons are discussed in Section 4.

3.1.1 Pseudo-Built-Ins

There is a "hidden" (and possibly unintentional) feature of Protégé that makes SWRL rules much more powerful. Regular SWRL supports only unary and binary predicates, except for the SWRL built-ins (see Section 5) which can take an arbitrary number of arguments. This is because OWL classes and properties are used for user-defined predicates. However, in Protégé there is a way to create new predicates with an arbitrary number of arguments! Here's how to do it:

1. In Project->Configure->Options, check "Display Hidden Frames".
2. In the Individuals tab, select the `swrlb:Builtin` class.
3. Create a new individual of this class and give it the name you want for your new predicate.

You can now use your new predicate in SWRL rules and queries. Note that the arity is not defined, and the SWRL rule and query editors will not display an error when you use the "wrong" number of arguments. Thus, you must make sure to consistently use the same number of arguments.

Of course, your new predicate isn't really "built-in," i.e. it is not implemented by the reasoning engine. You have to create the rules that define the semantics of the new predicate. Hence, we can refer to these predicates as pseudo-built-ins.

Being able to use more than two arguments is extremely useful. When you are limited to two arguments, one of which is often a "return value," you essentially can only create functions of one argument. The other solution is to use a list as an argument, and then pack the "real" arguments into this list. However, this becomes very cumbersome, as most rules need lots of additional content to construct and deconstruct these lists.

One word of caution: Using pseudo-builtins is a non-standard use of SWRL. Ontologies containing them may not load or work properly in other ontology tools.

3.1.2 Anonymous Variables

In many queries, there is a need to use "intermediate" variables, whose bindings are not interesting to the user. Normally the bindings for all variables are reported in the query result.

*Example: x=5*6+3*

```
swrlb:multiply(?z,5,6) ∧
swrlb:add(?x,?z,3)
```

returns

```
?x = 33.0
?z = 30.0
```

However, we're not interested in `?z`. This is also indicated by the fact that it does not even appear in the mathematical notation for the query. SWRL provides a relational encoding of mathematics and other functional expressions, and this always involves the introduction of new variables.

The solution to this is a feature called Anonymous Variables. By prefixing a variable name with `_` (underscore), it will not be reported in the query result. Thus,

*Example: x=5*6+3*

```
swrlb:multiply(?_z,5,6) ∧
swrlb:add(?x,?_z,3)
```

returns only

```
?x = 33.0
```

3.2 Tracing and Debugging

The tracing and debugging features of SWRL-IQ are used for explanation of reasoning results. They answer the questions "why?" or "why not?" did a result happen. The user interface for both is similar. The debug view adds a few elements that the trace view does not have, so we start by describing the trace view.

3.2.1 Tracing

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. There are two versions of the tracer - "Trace First" and "Trace All". The former returns the first proof of a given query. The latter returns all proofs of the query. There are often multiple ways to reach the same result. When many parts of a proof each have multiple proofs, the combination of possible proofs for the overall query can become very large, and cause the tracing to take a very long time. In such cases, the tracing can be aborted using the Stop button, and re-tried with the Trace First feature.

The trace window contains a tree showing one or more *proofs* of the given query, as was shown in Figure 3. Each node in the solutions tree has a **context menu** that can be accessed by right-clicking the node. The context menu allows you to expand all nodes below the current one, copy the current node to the clipboard, or copy the current node and all its children to the clipboard. The copy-to-clipboard options can be useful for doing text searches on the proof trees, when the proofs are very large or numerous. In the future, we intend to integrate search and filtering capabilities into the user interface in a more direct way.

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.

The Solution nodes have one or more child nodes, which we call *atom nodes* - One atom node for each atom in the query. The atom nodes consist of an *atom*, a *justification*, and possibly a number of child atom nodes. The atom nodes are color coded, as shown in Table 3.2.1.

Element	Color
Class	Yellow
Object Property	Blue
Datatype Property, Literal	Green
SWRL Builtin, Variable	Red
Individual	Purple
Other RDF Resource	Pink

Table 1. Color-coding in the trace and debug trees

In tracing mode, each atom will have all its variables instantiated. In debug mode, this is not always the case for failed sub-proofs (more on this below). The variable instantiations are different for different solutions. 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 possible justifications (inference rules) are shown in Table 3.2.1[4].

Note that the class expressions can occur wherever a 'C', 'C1', or 'Cn' argument is indicated. This also means that the justifications can be nested to an arbitrary depth (since class expressions can be nested in such a way).

3.2.2 Debugging

Debug trees show *attempted* proofs rather than actual proofs, and add two types of nodes to the proof trees: FAILED and ABORTED nodes. FAILED nodes occur as a child of an atom that was attempted, but could not be proved. ABORT nodes occur subsequent to FAIL nodes, because sub-proofs where one part has failed cannot succeed, so the sub-proof attempt is aborted.

Note that debug trees can have a very large number of "solutions", because there can be a very large number of ways to attempt to prove something. Often, many proof attempts will seem meaningless to the human user, and failures will occur in "uninteresting" places. If the debug output is overwhelming, we recommend breaking the query down into smaller parts to find the "interesting" failure.

3.3 Semantics and Limitations

The SWRL-IQ reasoner differs significantly from typical OWL reasoners such as Pellet and Fact++. Most OWL reasoners are *Description Logic* reasoners, which are intended primarily for so-called T-box, or terminological, reasoning. For example, they can be used to infer that class A is a subclass of class B. These reasoners have no, or limited, support for SWRL rules.

SWRL-IQ, on the other hand, focuses on so-called A-box, or assertional, reasoning, i.e. reasoning about facts in the knowledge base, while still taking into account terminological axioms in the ontology. SWRL-IQ supports complex built-ins (see Section 5), advanced SWRL extensions (see Section 6), and user-defined procedural attachments defined in Java (see Section 8).

[4] Some of these, e.g. disj and differentFrom, will currently not appear due to limitations of the reasoner. See Section 3.3

Justification entry	**Meaning**
Axioms	
assert	Asserted fact
comp	Computed as a built-in
rule(R)	SWRL rule with name R
subcls(C1,C2)	C1 is a sub-class of C2
eqvcls(C2,C2)	C1 and C2 are equivalent classes
disj(C2,C2)	C1 and C2 are disjoint classes
subprop(P1,P2)	P1 is a sub-property of P2
eqvprop(P1,P2)	P1 and P2 are equivalent properties
dom(P,C)	Property P has domain C
rng(P,C)	Property P has range C
invprop(P1,P2)	P1 and P2 are inverse properties
sameAs(I1,I2)	I1 is the same individual as I2
differentFrom(I1,I2)	I1 is a different individual than I2
funcprop(P)	P is a functional property
symprop(P)	P is a symmetric property
transprop(P)	P is a transitive property
invfuncprop(P)	P is an inverse functional property
Class expressions	
class name	the named class itself
union(C1,...,Cn)	Union class
int(C1,...,Cn)	Intersection class
enum(I1,...,In)	Enumerated class
compl(C)	Complement class
all(P,C)	allValuesFrom on property P, class C
some(P,C)	someValuesFrom on property P, class C
has(P,I)	hasValuesFrom on property P, individual I
maxCard(P,N)	maxCardinality on property P, value N
minCard(P,N)	minCardinality on property P, value N
card(P,N)	cardinality on property P, value N

Table 2. Justifications in the trace and debug trees

SWRL-IQ is a *sound* but not *complete* reasoner with regard to the OWL language. In other words, all query answers are correct, but there could be additional answers that the reasoner is not able to infer. This is because the reasoner sacrifices support for some OWL constructs in order to maintain computational efficiency. Indeed, SWRL-IQ could be considered a "pure SWRL" reasoner, in the sense that all of the OWL axioms that are supported could also be written as SWRL rules! For example[5], the OWL axiom

`SubClassOf(Wine, restriction(hasMaker, allValuesFrom(Winery)))`

can be written as a SWRL rule

`Wine(?x)` $\wedge$ `hasMaker(?x,?y)` $\rightarrow$ `Winery(?y)`

More precisely, the fragment of OWL that is supported is called DLP (Description Logic Programs)[6].

SWRL-IQ does not support OWL class descriptions inside SWRL rules (and Protégé does not even let you enter those, so most people will not miss this). However, defined classes can be used to achieve the same effect, as long as the class definition is within the DLP fragment. Other than this limitation, arbitrary SWRL rules are supported.

[5] This example is taken from the OWL Web Ontology Language Guide, `http://www.w3.org/TR/2004/REC-owl-guide-20040210`

[6] `http://www2003.org/cdrom/papers/refereed/p117/p117-grosof.html`

4 EXPORTING AND PROCESSING QUERY RESULTS

SWRL-IQ can export query results in two different formats. These results can be further processed or consumed by other tools. We describe the two export formats along with some example post-processing below.

4.1 SPARQL Query XML Results

After a query has been submitted and results appear in SWRL-IQ, clicking the "Show SPARQL XML" pops up a new window that shows the query result in the SPARQL XML Result format[7]. It may seem odd that we export using a SPARQL format as we do not support SPARQL as a query language. However, the motivation is that a) exporting to *some* XML format can come in handy, since it allows us to process the result using XML tools, and b) the SPARQL query result format already exists, and there is no reason to define a new format.

We make one addition to the SPARQL XML format. In order to allow lists in query results, we add a new `sri:list` element for the variable bindings, where `sri` is a prefix for the namespace `http://www.onistt.org/sparql`.

When the SPARQL XML result window is showing, the "Save..." button allows you to save the result to a file, as shown in Figure 11.

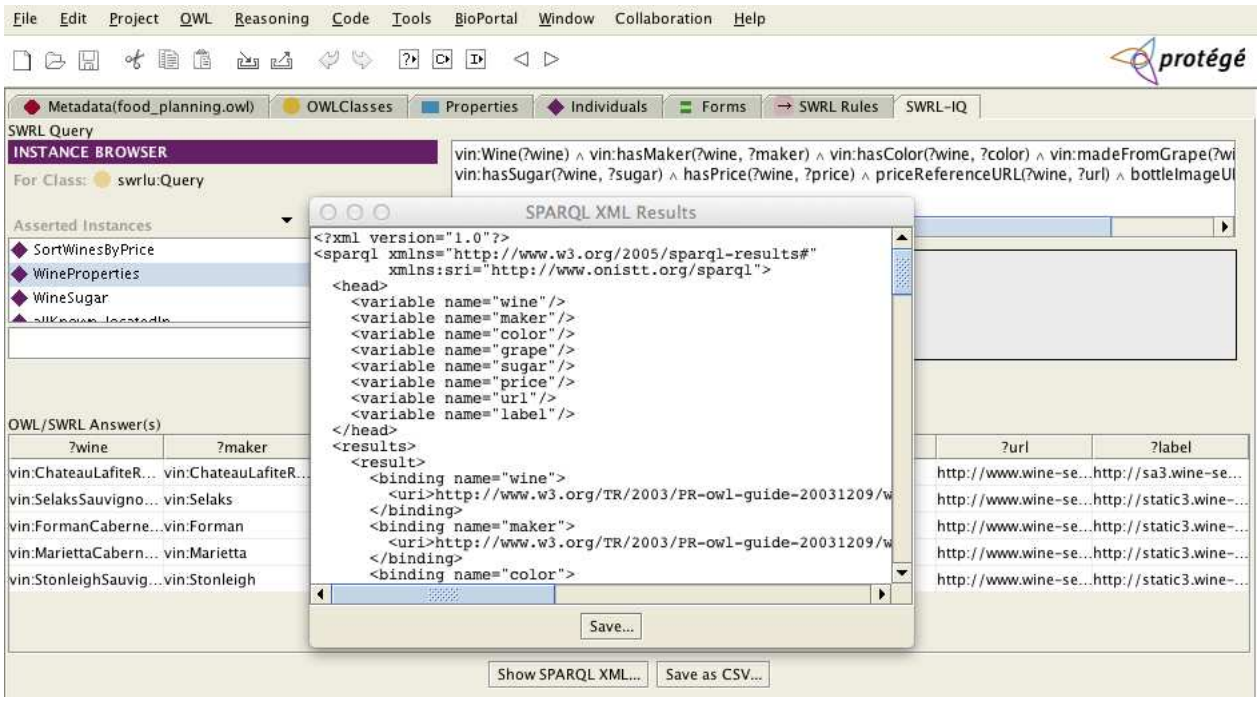

Fig. 11. Saving XML

[7] `http://www.w3.org/TR/rdf-sparql-XMLres/`

4.1.1 Post-Processing XML Query Results with XSLT

The XML query results can be post-processed with XSLT[8] to produce output in other formats, such as HTML. To try this out yourself, save the output of the `WineProperties` query from `food_planning.owl` into your `SwrlIq/html` directory and name it `query_result.xml` (this file name is hard coded into the `query_result.html` file provided)[9] Then look at the `query_result.html` page in an XSL-capable web browser, such as Firefox. It should look as in Figure 5 in the Overview Section. The `query_result.xsl` stylesheet takes the query output and formats it as a table, stripping prefixes like `vin:`, showing `.jpg` URLs as inline images, making other URLs clickable, and formatting decimals as prices in dollars. This is just a simple and not very robust example that can be modified for your own purposes.

4.2 CSV Query Results

The "Save as CSV" button allows you to save query results to a comma-separated values (CSV) file. The file will have the variable names in the first row, and the bindings in subsequent rows, similarly to the query results table in the main SWRL-IQ tab.

The main usefulness of CSV files is that they can be opened directly in Excel and other spreadsheet applications, and are automatically rendered into cells appropriately. This can be used to generate graphs, export to other formats, etc. An example of this was shown in Figure 6 in the Overview Section.

[8] XSL Transformations, `http://www.w3.org/TR/xslt`

[9] Alternatively you could create an html page that reads in path names dynamically and/or calls an XSLT Processor like SAXON instead of relying on the browser.

5 SWRL BUILT-INS

SWRL-IQ supports most of the standard SWRL Built-ins[10]. Table 3 summarizes our level of support for these built-ins, with more detailed notes and examples in the following sections. We do not repeat the standard definitions of all the predicates here.

5.1 Constraints

Many of the built-ins can generate numerical *constraints*, and are implemented to do so in a very flexible way. 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. One consequence of this is that these built-ins can be used "backwards," and in effect to solve systems of equations. This often makes rules that use these built-ins more declarative and powerful. The constraint mechanism is best understood through examples (see Table 4).

Note that the built-ins `integerDivide`, `mod`, `ceiling`, `floor`, and `round` cannot be used "backwards" for obvious reasons: The results are not uniquely defined in the reverse direction. These predicates do not generate constraints. If the input arguments to these predicates are not instantiated, the query will simply fail.

5.2 Lists

A few things are worth pointing out when it comes to our implementation of list operations:

- `rdf:List`s can be created on-the-fly in SWRL queries, using the syntax `rdf:List(m1,m2...)`, where `m1,m2...` are the list members.
- 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), i.e. `[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 5.

5.3 Strings

String literals can be typed directly in a query by enclosing them in double quotes. For example,

Example:

```
swrlb:upperCase(?s,"foo")
```

returns

```
?s = "FOO"
```

5.4 Dates, Times, and Durations

These can be used both to generate and to "disassemble" the corresponding units.

Example:

```
swrlb:dateTime(?dt,2011,11,11,11,11,11,?tz)
```

[10] `http://www.w3.org/Submission/SWRL/#8`

Built-in predicate	Support	Constr.	Notes
swrlb:equal, swrlb:notEqual	Yes	Yes	Only for numerical datatypes
swrlb:lessThan, swrlb:lessThanOrEqual, swrlb:greaterThan, swrlb:greaterThanOrEqual	Yes	Yes	
swrlb:add, swrlb:subtract, swrlb:multiply, swrlb:divide, swrlb:pow, swrlb:unaryPlus, swrlb:unaryPlus, swrlb:unaryMinus, swrlb:abs, swrlb:sin, swrlb:cos, swrlb:tan	Yes	Yes	
swrlm:sqrt, swrlm:log	Yes	Yes	
swrlb:integerDivide, swrlb:mod, swrlb:ceiling, swrlb:floor, swrlb:round	Yes	No	
swrlb:roundHalfToEven	No	No	
swrlb:booleanNot	Yes	No	
swrlb:stringConcat, swrlb:stringLength, swrlb:startsWith, swrlb:endsWith, swrlb:upperCase, swrlb:lowerCase, swrlb:contains, swrlb:containsIgnoreCase, swrlb:stringEqualIgnoreCase, swrlb:matches, swrlb:replace	Yes	No	
swrlb:normalizeSpace, swrlb:translate, swrlb:substringBefore, swrlb:substringAfter, swrlb:tokenize	No	No	
swrlb:date, swrlb:time, and swrlb:dateTime	Yes	No	date/time/dateTime types cannot be entered directly in a query
Other SWRL time built-ins	No	No	
swrlb:resolveURI, swrlb:anyURI	No	No	
swrlb:listConcat, swrlb:listIntersection, swrlb:listSubtraction, swrlb:member, swrlb:length, swrlb:first, swrlb:rest, swrlb:sublist, swrlb:empty	Yes	No	

Table 3. Summary of support for SWRL built-ins

Query	Bindings returned	Constraints returned
`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, ?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, ?x = -1.4142135623730951	none
`swrlb:pow(?_sq,?x,2)` ∧ `swrlb:multiply(10,?_sq,5)` ∧ `swrlb:greaterThanOrEqual(?x,0)`	?x = 1.4142135623730951	none

Table 4. Examples of SWRL queries with constraints

Query	Bindings returned
`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], ?l = [?Var1,1]
`swrlb:first(1,?l)`	?l = [1|?Var0]

Table 5. Examples of SWRL queries with lists

returns

```
?dt = 2011-11-11T11:11:11
```

Note that Protégé does not have a way to store timezone information (the last argument). Also note that there is no way to directly write a `xsd:date`, `dateTime`, or `xsd:time` value in a SWRL query or rule. E.g. the query

```
swrlb:dateTime(2011-11-11T11:11:11,?y,?m,?d,?h,?m,?s,?tz)
```

is not syntactically correct (you will get an "invalid literal" error in the SWRL query editor). If you put quotes around the first argument, the query is syntactically correct, but the first argument is treated as a plain string, and therefore the query fails. However, a date/time term can be created dynamically using these predicates as shown above. If a value of one of these types exists in the ontology (e.g., as the value of a datatype property assertion), it can be retrieved by an appropriate query, and the different parts of the date/time value can then be extracted using these predicates.

The "time arithmetic" built-ins are not implemented, mainly because they operate on non-standard XSD data types that cannot actually be used in Protégé.

5.5 Protégé Extensions

Protégé adds a number of SWRL built-ins[11]. SWRL-IQ does not support most of these. The ones we do support are described below. Note that you need to import the relevant ontology (i.e. `rdfb.owl` or `swrlm.owl`, respectively) to use these built-ins.

rdfb:hasLabel(?s,?l)
Satisfied iff `rdfs:label(?s,?l)` is satisfied. This predicate is needed because SWRL does not accept RDF predicates in queries. This lets us "call" the `rdfs:label` predicate from a SWRL rule or query.

[11] `http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTabBuiltInLibraries`

swrlm:log(?s,?l)
Satisfied iff the first argument is equal to the natural logarithm (base e) of the second argument.

swrlm:sqrt(?s,?l)
Satisfied iff the first argument is equal to the square root of the second argument. This can also be achieved with the `swrlb:pow` built-in.

6 SWRL EXTENSIONS

SWRL-IQ comes with an `swrl-extensions.owl` ontology, which contains a number of new SWRL "built-ins". These built-ins all have dedicated implementations in the Prolog reasoner core. Remember to use the prefix for the SWRL Extensions ontology when using these predicates in queries outside of that file, e.g. `swrlex:pi` rather than just `pi`. Keep in mind that these extensions are non-standard, and will not work in other reasoning engines besides SWRL-IQ (unless others decide to implements the same extensions). The examples below are all kept very simple, and can be executed with no ontology loaded other than `swrl-extensions.owl`.

The first several predicates (`ignore`, `pi`, `string_number`, `string_resource`, and `value`) are relatively simple utility predicates. The last few predicates (`apply`, `allKnown`, `firstMatch` and `callJavaStaticMethod`) offer more radical extensions to SWRL. They go beyond the semantics of SWRL, and should be used sparingly if a pristine model theory is considered critical.

ignore(?x1,?x2,...)
Takes an arbitrary number of arguments and is always satisfied. This is useful when creating queries, because SWRL requires all variables that are in the rule head to also be in the rule body. However, there is often a need to have "extra variables" in the rule head, e.g., when there are several rules that define a predicate, and some of the arguments are not used in some of the rules. Inserting an `ignore` atom into the rule body accomplishes this without unwanted side-effects.

Example: A predicate that adds a number to all numbers in a list

```
swrlb:empty(?l2) ∧
swrlb:empty(?l1) ∧
swrlex:ignore(?n)
→
addN(?l1,?l2,?n)

swrlb:first(?f2,?l2) ∧
swrlb:add(?f1,?f2,?n) ∧
swrlb:first(?f1,?l1) ∧
swrlb:rest(?r2,?l2) ∧
swrlb:rest(?r1,?l1) ∧
addN(?r1,?r2,?n)
→
addN(?l1,?l2,?n)
```

In the first rule, `ignore` is needed since the variable `?n` would otherwise appear in the rule head but not in the rule body.

The `ignore` predicate is not useful in queries.

pi(?x)
Takes one argument, which is unified with the constant π (as a decimal up to the precision that the underlying Prolog engine supports).

Example:

```
swrlex:pi(?pi) ∧
swrlb:divide(?_cf,?pi,180) ∧
swrlb:multiply(?radians,90,?_cf)
```

returns

```
?pi = 3.141592653589793
?radians = 1.5707963267948966
```

string_number(?str,?num)
Takes two arguments, a string and a number. Satisfied when the string is a recognized string representation of the number. Can be used in both directions to convert between numbers and strings. Note that the exact representation may be changed e.g., between scientific and decimal notations.

Example:

```
swrlex:string_number("5.0e-3",?x)
```

returns

```
?x = 0.005
```

Example:

```
swrlex:string_number(?x,0.00000001)
```

returns

```
?x = 1.000000e-08
```

string_resource(?str,?res)
Takes two argument, a string and an RDF resource (i.e., an OWL individual, class, etc). Satisfied when the string is the full name of the resource (including the full URI). Like `string_number`, this predicate can be used in both directions.

Example:

```
swrlex:string_resource(?s,owl:Thing)
```

returns

```
?s = "http://www.w3.org/2002/07/owl#Thing"
```

value(?s,?o)
Satisfied when `rdf:value(?s,?o)` is satisfied. This predicate is needed because SWRL does not accept RDF predicates in queries. This lets us "call" the `rdf:value` predicate from a SWRL rule or query.

apply(?p,?x1,?x2,...)
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, which can be quite useful sometimes. Note that queries using this predicate may take a very long time if `?p` is not instantiated, i.e., if you ask for all predicates that satisfy given arguments.

As an example, consider a predicate `minimum(?min,?x,?y,?p)` which returns the smallest value `?min` out of two values `?x` and `?y`, according to some ordering predicate `p`. Think of the ordering predicate as "less than or equal". The `minimum` predicate is defined by the following two rules in `swrl-utilities.owl` and also documented in Section 7.

```
swrlex:apply(?p,?x,?y)
→
minimum(?x,?x,?y,?p)

swrlex:apply(?p,?y,?x)
→
minimum(?y,?x,?y,?p)
```

The minimum predicate can now be used to determine the minimum of any types of numbers or individuals, as long as the appropriate ordering predicate can be supplied. This in itself may not seem that useful, but the value of this predicate becomes clear when we consider that it is the basis for the more

complex predicates `min_list` and `sort`. The latter can be used to sort a list of arbitrary types of RDF resources, for example to sort a list of wines according to their price by passing a properly defined predicate `wineCheaperThan` as the parameter `?p`. Note that if the two values are "equal" according to `?p`, i.e. both `p(?x,?y)` and `p(?y,?x)` are true, then we get two solutions. This is useful - for example the `sort` predicate produces all valid sortings of the "equal" elements.

All of these predicates can be used on plain numbers by passing `swrlb:lessThanOrEqual` as the parameter `?p`. Section 7 contains examples of this.

allKnown(?res,?x,?p,?x1,?x2,...)
Returns a list `?res` containing all values for `?x` such that `p(?x1,?x2,...)` is satisfied.

Example: All solutions to $x^2 = 4$

```
swrlex:allKnown(?res,?x,swrlb:pow,4,?x,2)
```

returns

```
?x = [-2.0, 2.0]
```

Note the difference with the first example for `swrlb:pow` in Table 3. In that case we got two different solutions. When we use `allKnown`, we get all the solutions gathered into a list.

The `?x` parameter can also be a list of variables, as in the wine example from Section 1:

```
swrlex:allKnown(?_listOfWinePricePairs,
   rdf:List(?_wine, ?_price), hasPrice, ?_wine, ?_price)
```

When used in this way, the result is a list of lists of bindings of all the variables in the variable list.

If the goal contains additional variables that are not in the second parameter, they are treated existentially. For example

```
swrlex:allKnown(?wines,?wine,hasPrice,?wine,?price)
```

in the `food-planning.owl` ontology returns one list containing all the wines that have some value for the `hasPrice` property.

This built-in significantly extends the power of the language/reasoner. It provides a kind of "local closed-world assumption". As the name of this predicate implies, it returns all "known" solutions to a query. The open-world assumption of OWL and SWRL means that there could be additional solutions.

firstMatch(?res,?list,?p,?x1,?x2,..)
Returns the first element in the list that satisfies `p(?x1,?x2,...)`. The variable `?res` is normally one of the arguments `?x1,?x2,...`.

Example:

```
swrlex:firstMatch(?x,rdf:List(1,2,3,4),swrlb:greaterThan,?x,2)
```

returns

```
?x = 3
```

callJavaStaticMethod(?ret,?cls,?mthd,?arg1,?arg2,...)
This predicate is the hook to the Java Attachment mechanism. This is described in detail in Section 8.

7 SWRL UTILITIES

Besides containing the ontological elements for saving queries, as discussed in Section 3, `swrl-utilities.owl` also contains a number of predicates, defined in SWRL rules, that are generally useful. The rules for these predicates can also serve as nice, small examples of how to write SWRL rules.

Remember to use the prefix for the SWRL Utilities ontology when using these predicates in queries outside of that file, e.g. `swrlu:nth` rather than just `nth`.

disjointWith(?l1,?l2)
Satisfied if the two lists `?l1` and `?l1` do not have any common elements.

intersectsWith(?l1,?l2)
Satisfied iff the two lists have at least one element in common.

subsetOf(?l1,?l2)
Satisfied iff the list `?l1` is a subset of the list `?l2`.

nth(?n,?l,?res)
Satisfied iff `?res` is the `?n`th element of the list `?l`, starting at `?n` = 0 for the first element.

first(?x,?l), second(?x,?l), third(?x,?l), fourth(?x,?l)
Satisfied iff `?x` is the first/second/third/fourth element in the list `?l`. Use `nth` for higher indices, or define additional short-cut predicates like these.

logn(?x,?y,?n)
Satisfied iff `?x` is the base `?n`-logarithm of `?y`.

log10(?x,?y)
Satisfied iff `?x` is the base 10 logarithm of `?y` (i.e., just a short-cut for `logn`).

dB(?x,?y)
Satisfied iff `?x` is the decibel value of `?y`.

notMember(?el,?l)
Satisfied iff `?el` is not a member of the list `?l`.

minimum(?min,?x,?y,?p)
Satisfied iff `?min` is the smaller of the elements `?x` and `?y` according to the ordering predicate `?p`.

Example:

```
swrlu:minimum(?min,4,3,swrlb:lessThanOrEqual)
```

returns

```
?x = 3
```

The rules defining this predicate are discussed in Section 6.

min_list(?x,?l,?p)
Satisfied iff `?x` is the smallest element in the non-empty list `?l` according to the ordering predicate `?p`.

Example:

```
swrlu:min_list(?x,rdf:List(4,17,3,20),swrlb:lessThanOrEqual)
```

returns

```
?x = 3
```

There are also the corresponding predicates:
maximum(?max,?x,?y,?p)

max_list(?x,?l,?p)
and these convenient shortcuts for numerical values:
min_num(?min,?x,?y)
min_num_list(?x,?l)
max_num(?max,?x,?y)
max_num_list(?x,?l)

sort(?sl,?l,?p)
Satisfied iff `?sl` is the version of the list `?l` sorted in ascending order using the ordering predicate `?p`.

Example: Sorting numbers

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

returns

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

Example: Sorting wines by their price
First we 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

```
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.

8 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 our extensions (see Section 6). One might wonder why an external procedural attachment mechanism is needed. Indeed, SWRL rules are in many ways more powerful and flexible (see Section 8.3 below). 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 following built-in predicate (defined in `swrl-extensions.owl`).

callJavaStaticMethod(?ret,?cls,?mthd,?arg1,?arg2,...)
`?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.

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 below.

8.1 Argument Conversions

Java and OWL/SWRL have vastly different type systems. Java Attachments support neither all Java types nor all OWL "types". In fact, we support what might be considered the intersection of the two, viz. the basic types of both, with some obvious conversions. Table 8.1 summarizes the supported types for the two sides, and their conversions:

SWRL/OWL type	Java type
`xsd:int`, `xsd:integer`, etc	`int`
`xsd:float`, `xsd:double`, `xsd:decimal`, etc	`double`
`xsd:boolean`	`boolean`
`xsd:string`	`String`
`rdf:List`	`Object[]`

Table 6. OWL-Java type conversions

If integers, floats, etc, are inside a `rdf:List`, they are mapped to the corresponding "boxed" Java type, i.e. `Integer`, `Double`, or `Boolean`.

Note that there is no way to pass RDF resources (individuals, classes, and properties) to Java Attachments.

8.2 Writing and Installing a Java Attachment

Creating a new Java Attachment involves creating a new class with the new method to call, compiling it, and putting the `.jar` file in the `com.sri.swrltab` directory under Protégé's `plugins` directory.

As an example, see the `java/test/BuiltinTest.java` file distributed with SWRL-IQ. Its full contents are shown below.

```
package test.builtins;

public class BuiltinTest {

  public static String helloWorld(String firstname, String lastname){
    return "Hello " + firstname + " " + lastname + "!";
  }

}
```

To install this Java attachment, first compile it into a jar file. On Windows, enter the following commands in the main `SwrlIq/java` directory:

```
javac test\*.java
jar cf hello.jar test\*.class
```

On Linux or Mac, use these commands:

```
javac test/*.java
jar -cf hello.jar test/*.class
```

Next, copy `hello.jar` to the SwrlIq plugin directory under Protégé. Restart Protégé. Now enter the query

```
swrlex:callJavaStaticMethod(?res,"test.BuiltinTest",
        "helloWorld","John","Doe")
```

This should return one result:

```
?res = "Hello John Doe!"
```

This is all it takes to create your own Java Attachments.

8.3 Limitations and Caveats

Java Attachments have a number of limitations when compared to predicates defined using SWRL rules. Generally speaking, Java Attachments do not have a well-defined semantics, and make things more procedural and less declarative. They cannot be used "backwards," and all the input arguments must be fully instantiated before the attachment is called. The user must understand the operational semantics of the reasoner to make sure this happens, and rules containing Java Attachment calls must be written with the atoms in a certain order. Java Attachments will not work with OWL/SWRL reasoners other than SWRL-IQ, since they are our own extension. Despite all this, for some purposes they can be quite useful.

9 ACKNOWLEDGMENTS

The work described in this paper was carried out at the SRI facilities in Menlo Park, CA and was funded by the U.S. Department of Defense, TRMC (Test and Evaluation/Science and Technology) 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 would like to thank Gil Torres, NAVAIR, for his leadership of the Netcentric System Test (NST) technology area, to which the ANSC project belongs. We would also like to acknowledge ODUSD/R/RTPP (Training Transformation) for its sponsorship of the associated ONISTT project. Reg Ford, Mark Johnson, and Stijn Heymans at SRI provided invaluable feedback on this manual and query-testing. Finally, David Warren and Terrance Swift provided expert guidance and bug fixes for XSB Prolog, and Miguel Calejo did the same for InterProlog.