A Mechanism to Define and Execute SWRL Built-ins in Protégé-OWL

Martin J O'Connor , Amar Das
Stanford Medical Informatics, Stanford, CA 94305-5479

We have developed an extension to Protégé-OWL that provides mechanisms to define Java implementations of SWRL built-ins, to dynamically load these implementations, and to invoke them from a rule engine.

1. SWRL Built-Ins

SWRL[1] provides a very powerful extension mechanism that allows user-defined methods to be used in rules. These methods are called built-ins[2] and are predicates that accept one or more arguments. Built-ins are analogous to functions in production rule systems. A number of core built-ins are defined in the SWRL specification. This core set includes basic mathematical operators and built-ins for string and date manipulations. These built-ins can be used directly in SWRL rules. For example, the core SWRL mathematical built-in called `greaterThanOrEqual` can be used as follows to indicate that a person with an age of 18 or greater is an adult:

```
Person(?p) ^ hasAge(?p, ?age) ^ swrlb:greaterThanOrEqual(?age, 18) -> Adult(?p)
```

When executed, this rule would classify individuals of class `Person` with an `hasAge` property value of 18 or greater as members of the class `Adult`. The `swrlb` qualifier before the built-in name indicates the alias of the namespace containing the built-in definition. In this case, it indicates that the built-in comes from the core SWRL built-in ontology.

Users can also define their own built-in libraries. Example libraries could include built-ins for currency conversion, and statistical, temporal or spatial operations. Again, these user-defined built-ins can be used directly in SWRL rules.

An OWL definition for a SWRL built-in is provided by the class `Builtin`, which is contained in the files `swrl.owl` and `swrlb.owl`. These files have the namespace base `http://www.w3.org/2003/11/`. A new user-defined built-in is described in OWL as an instance of this class. The individual name is set to the name of the built-in. In general, a set of related built-ins are defined in a single OWL file. For example, a user-defined set of temporal built-ins could be defined in a file called `temporal.owl`. A specific built-in, such as, say, `before`, would then be defined in this file as an individual named `before`, which would be an instance of the class `Builtin`. The argument properties of each built-in can be used to specify the number arguments it is expecting.

To use these user-defined built-ins in SWRL rules, the file containing them must be imported. Sets of built-ins are usually given a user-friendly alias when they are imported. For example, the built-ins defined in `temporal.owl` could be give the alias `temporal` that can be used to qulaify their use in SWRL rules.

2. Defining Java Built-in Implementations

We have developed an extension to the Protégé-OWL SWRLTab[3],[4] called the *SWRL Built-in Bridge*[5] to provide support for defining and dynamically loading built-in implementation written in Java. Users wishing to provide implementations for a library of built-in methods must first define a Java class that contains definitions for all the built-ins in the library. The bridge is expecting this built-in implementation class to be called `SWRLBuiltInMethodsImpl`. This class must implement the

[1] http://www.w3.org/Submission/SWRL/
[2] http://www.daml.org/2004/04/swrl/builtins.html
[3] http://protege.stanford.edu/plugins/owl/swrl/
[4] M. J. O'Connor, H. Knublauch, S. W. Tu, B. Grossof, M. Dean, W. E. Grosso, M. A. Musen. Supporting Rule System Interoperability on the Semantic Web with SWRL. Fourth International Semantic Web Conference (ISWC2005), Galway, Ireland, 2005.
[5] http://www.w3.org/Submission/SWRL/BuiltInBridge.html

interface `SWRLBuiltInMethods`[6]. This interface acts as a typing or structuring mechanism - it does not define any methods itself.

The package name of the `SWRLBuiltInMethods` class should be the namespace qualifier of the built-ins appended to the Java package name `edu.stanford.smi.protegex.owl.swrl.bridge.builtins`. For example, the standard SWRL built-in `swrlb:greaterThan` should be defined as a method called `greaterThan` in the class `SWRLBuiltInMethodsImpl`, which should be located in the package `edu.stanford.smi.protegex.owl.swrl.bridge.builtins.swrlb`.

Each implementation of a specific built-in in the `SWRLBuiltInMethodsImpl` class should have a signature of the form:

```
public static boolean <builtInName>(List arguments) throws BuiltInException
```

The single arguments parameter is a list that should contains one or more `Argument` objects[7]. The three possible types of argument objects expected by the bridge are `LiteralInfo`, `IndividualInfo` and `VariableInfo`. The `LiteralInfo` and `IndividualInfo` objects are used to pass information to built-ins: the `LiteralInfo` object contains OWL literals, such as integers or strings, and the `IndividualInfo` object specifies the name of an OWL individual. The `VariableInfo` class can be used by a built-in to assign a value to a variable. This value can be either an OWL individual name or a literal.

The three parameter classes have constructors to create them from their matching types. `LiteralInfo` objects can be constructed from `RDFFSLiteral` objects or from basic Java types. Accessor methods are provided to get these values. `IndividualInfo` and `VariableInfo` classes can be constructed from Java instances of `OWLIndividual` and `SWRLVariable` classes, respectively. Both of these classes also have a `getName` call to retrieve the variable or individual name.

For example, the SWRL rule atom `swrlb:add(?x, 2, 3)` could use the core SWRL `add` built-in to add two integer literals. When this built-in is invoked by a rule engine, the first argument should be an instance of the `VariableInfo` class with its name set to `x`; the second and third arguments should be instances of the `LiteralInfo` class that hold their respective values.

Each built-in class must declare the exception `BuiltInException`, which is defined in the `exceptions` subpackage of the standard bridge package. This abstract class has concrete subclasses for the four possible exceptions that can be thrown by a built-in implementation: (1) `InvalidBuiltInArgumentNumberException`, which is used indicate that an incorrect number of arguments have been passed to the built-in; (2) `InvalidBuiltInArgumentException`, which should be used used to indicate that an argument of the wrong type has been passed to the built-in; (3) `LiteralConversionException`, which can be used to indicated that literal argument is invalid in some way; and (4) `BuiltInNotImplementedException`, which can be used to indicate that a built-in (or variants of it for a particular argument type) has not been implemented.

Here is an example Java method defining a built-in called `stringEquals` from the core SWRL built-in library:

```
private static String STRING_EQUALS = "stringEquals";

public boolean stringEqualIgnoreCase(List arguments) throws BuiltInException {

  String argument1, argument2;

  SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(STRING_EQUALS, 2, arguments.size());

  argument1 = SWRLBuiltInUtil.getArgumentAsAString(STRING_EQUALS, 1, arguments);

  argument2 = SWRLBuiltInUtil.getArgumentAsAString(STRING_EQUALS, 2, arguments);

  return argument1.equals (argument2);

} // stringEquals
```

This method illustrates the use of a utility class called `SWRLBuiltInUtil` that can be used to process arguments and generate appropriate exceptions. In the above example, the `checkNumberOfArgumentsEqualTo` method will throw an `InvalidBuiltInArgumentNumberException` if two arguments are not passed to the built-in; the `getArgumentAsAString` method can be used to extract a string value from a supplied `LiteralInfo` object and will throw an `InvalidBuiltInArgumentException` if both supplied arguments are not strings. Most of the methods in this utility class take the name of the built-in as their first parameter so that the offending built-in name can be displayed if an error is thrown.

[6] This class is defined in the Protégé-OWL package `edu.stanford.smi.protegex.owl.swrl.bridge.builtins`.
[7] This class is defined in the Protégé-OWL package `edu.stanford.smi.protegex.owl.swrl.bridge.builtins`.

3. Invoking a Built-in Method from a Rule Engine

A built-in methods can be invoked by a rule engine through the `SWRLRuleEngineBridge` class. The constructor for this class[8] takes an instance of an `OWLModel` class that contains the relevant knowledge base for the rules being executed, in addition to the rules themselves. To support built-in invocation, this class has a method called `invokeSWRLBuiltIn` that takes the name of a built-in and a list of `Argument` objects for that built-in. It returns a boolean value that holds the result of the built-in invocation. The built-name passed to the invoke method must be of the form `<builtInLibraryAlias>:<builtInName>`. For example, the `add` method in the core SWRL built-in library would be referred to as `swrlb:add`.

The invoke method itself can directly throw three possible exceptions: (1) `InvalidBuiltInNameException`, which is used to indicate that a supplied built-in name is not a valid name for a SWRL built-in, i.e., no OWL individual of class `BuiltIn` with the name of the built-in exists in the OWL model supplied to the bridge; (2) `UnresolvedBuiltInException`, which indicates that no Java implementation method for the built-in could be found in any of the dynamically loaded built-in libraries; and (3) `InvalidBuiltInMethodsImplementationClass`, which indicates that the Java implementation class found for the built-in did not correctly implement the interface `SWRLBuiltInMethods`.

If an implementation is found for the supplied built-in, it is invoked and is supplied with the argument list passed to the invoke method. As discussed above, a built-in implementation can throws four possible exceptions (which are also subclasses of the `BuiltInException` class). If an exception is throws it is passed directly back to the caller. If the method executes successfully, its (boolean) return value is passed back from the invoke method.

Rule engines that wish to access this invocation mechanism are responsible for creating an instance of the `SWRLRuleEngineBridge` class[9] and calling the invoke method at run-time as rules are executed. The mechanism that connects invocations of built-ins from inside a rule engine to the bridge's invoke method is going to be rule engine specific.

It is worth noting that there is going to be considerable overhead to invoking built-in implementations from inside a rule engine. In addition to marshalling built-in arguments, the process of invoking external methods from inside a rule engine during rule evaluation can dramatically slow down its execution. This overhead may not be dramatic if the rule engine is Java-based, but for non Java-based engines the overhead could be significant. In general, if a particular built-in is expected to be used a lot, a native rule engine implementation of the method should be developed. Many engines have in libraries of in-built methods available that can be used in this implementation process.

4. Loading a Built-in Implementation Class at Runtime

Sets of related built-ins are contained in the same `SWRLBuiltInMethodsImpl` class. For example, the implementation class containing the above `stringEquals` method can then be defined as follows:

```
package edu.stanford.smi.protegex.owl.swrl.bridge.builtins.swrlb;

import edu.stanford.smi.protegex.owl.swrl.bridge.builtins.*;

import edu.stanford.smi.protegex.owl.swrl.bridge.exceptions.*;

public class SWRLBuiltInMethodsImpl implements SWRLBuiltInMethods {

  public boolean stringEquals(List arguments) throws BuiltInException { ... }

  ....

} // SWRLBuiltInMethodsImpl
```

To allow Protege-OWL to find this built-in implementation classes at run time it must first be placed in a JAR file. This JAR file should then be placed in the Protege-OWL plugins directory. Protege will automatically add this JAR file to the applications class path so that a class loader will be able to find this class at run time. The bridge employs a lazy loading mechanism: When a built-in from a particular implementation class is invoked for the first time, the bridge loads the implementation class using Java's class loader. Any subsequent invocations of built-ins from the class will be routed directly to the loaded class. The Java package name of the built-in implementation classes

An example `SWRLBuiltInMethodsImpl` class that implements most of the core SWRL built-ins can be found in the `edu.stanford.smi.protegex.owl.swrl.bridge.builtins.swrlb` package in the standard Protege-OWL distribution.

---

[8] This class is defined in the Protégé-OWL package `edu.stanford.smi.protegex.owl.swrl.bridge`.
[9] Described here: http://smi-web.stanford.edu/people/moconnor/swrl/RuleEngineBridge.html.