SWRLLanguageFAQ

What is SWRL? (6ZK)

SWRL is an acronym for Semantic Web Rule Language. It is intended to be the rule language of the Semantic Web. SWRL is based on a combination of the OWL DL and OWL Lite sublanguages of the OWL Web Ontology Language. It allows users to write Horn-like rules to reason about OWL individuals and to infer new knowledge about those individuals. These rules are expressed in terms of OWL concepts. SWRL is more expressive that OWL DL alone yet retains its formal semantics. It does so, however, at the expense of decidability. (8K5)

SWRL's model-theoretic semantics are defined in the SWRL Submission. This Submission does not supply a presentation syntax for the language. This FAQ describes a presentation syntax that is supported by the Protege-OWL SWRL Editor. Other SWRL implementations may chose an alternate presentation syntax. This FAQ also assumes that the reader is already familiar with OWL. A good user-level introduction to OWL can be found here. (8HX)

Basic Rules (8JL)

A SWRL rule contains an antecedent part, which is referred to as the *body*, and a consequent part, which is referred to as the *head*. Both the body and head consist of positive conjunctions of *atoms*. SWRL does not support negated atoms or disjunction. Informally, a SWRL rule may be read as meaning that if all the atoms in the antecedent are true, then the consequent must also be true. An empty antecedent is treated as trivially holding, and an empty consequent is treated as trivially not holding. Rules with an empty antecedent can thus be used to provide unconditional facts; however such unconditional facts are better stated in OWL itself. (8JW)

SWRL rules are written in terms of OWL classes, properties and individuals. For example, a SWRL rule expressing that a person with a male sibling has a brother would require capturing the concepts of 'person', 'male', 'sibling' and 'brother' in OWL. Intuitively, the concept of person and male can be captured using an OWL class called `Person` with

Your Visited Pages

SWRLLanguageFAQ
SWRLTab

View Backlinks

Search

a subclass `Man`; the sibling and brother relationships can be expressed using OWL object properties `hasSibling` and `hasBrother`, which are attached to `Person`. The rule in SWRL would then be: (71R)

- Person(?p) ^ hasSibling(?p,?s) ^ Man(?s) -> hasBrother(?p,?s) (721)

Executing this rule would have the effect of setting the `hasBrother` property of all OWL individuals with one or more male siblings to refer to those siblings. (71W)

Similarly, a rule that asserts that all persons that own a car should be classified as drivers can be written as follows: (72I)

- Person(?p) ^ hasCar(?p, true) -> Driver(?p) (8IF)

Again this rule would require that the datatype property `hasCar` and the class `Driver` exist in an OWL ontology. Executing this rule would have the effect of classifing all car-owner individuals of type `Person` to also be members of the class `Driver`. (72K)

Named individuals in an ontology can also be referred to directly in SWRL rules. For example, one could rewrite the above rule to classify an individual named Fred as a driver as follows: (8IG)

- Person(Fred) ^ hasCar(Fred, true) -> Driver(Fred) (8IH)

SWRL Built-Ins (8JQ)

One of the most powerful features of SWRL is its ability to support user-defined built-ins. A built-in is a predicate that takes one or more arguments and evaluates to true if the arguments satisfy the predicate. For example, an `equal` built-in can be defined to accept two arguments and return true if the arguments are the same. A number of core built-ins for common mathematical and string operations are contained in the SWRL Built-in Submission. (8JZ)

An example SWRL rule using a core SWRL built-in to indicate that a person with an age of greater than 17 is an adult is: (8D4)

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

Adult(?p) (8D5)

By convention, core SWRL built-ins are preceded by the namespace qualifier `swrlb`. When executed, this rule would classify individuals of class `Person` with an `hasAge` property value of greater than 17 as members of the class `Adult`. (8D6)

A rule that uses a core SWRL string built-in to determine if a person's telephone number starts with the international access code "+" can be written as follows: (8K0)

- Person(?p) ^ hasNumber(?p, ?number) ^ swrlb:startsWith(?number, "+") -> hasInternationalNumber(?p, true) (8K1)

Built-ins can take any number or combination of OWL datatype or object property values. Datatypes can be XML Schema Datatype or arbitrary user-defined types; object property arguments are effectively names of OWL individuals. The SWRL Submission does not provide a mechanism for specifying the number or types of built-in arguments - argument checking is the responsibility of the built-in implementor. The submission does specify that a built-in that is passed an incorrect number or arguments, or arguments of an incorrect type, should evaluate to `false`. (8K4)

SWRL allows new libraries of built-ins to be defined and used in rules. Users can define built-in libraries to perform a wide range of tasks. Such tasks could, for example, include currency conversion, temporal manipulations, and taxonomy searches. In general, the arguments to these built-ins should be OWL DL datatype values - that is, literals or individuals. However, class or property built-ins arguments may also be supported by some built-in libraries, though such built-ins should only be used in OWL Full ontologies. (98R)

Built-In Argument Binding (8JR)

Built-ins can also assign (or bind) values to arguments. For example, the SWRL atom `swrlb:add(?x, 2, 3)` uses the core SWRL built-in add method to add two integer literals. If `x` is unbound when this built-in is invoked, it will be assigned the the value 5 when the built-in returns. If `x` is already bound when the built-in is invoked, it will simply determine if its value is 5. (8CY)

A built-in method that successfully assigns a value to an argument should return `true`. If more than one unbound argument is present, all arguments must be bound. If the built-in returns `false` no assignments are expected. (8IJ)

An example rule that uses built-ins with assignment to calculate the area of a rectangle can be be written as follows: (8CT)

- Rectangle(?r) ^ hasWidthInMeters(?r, ?w) ^ hasHeightInMeters(?r, ?h) ^ swrlb:multiply(?areaInSquareMeters, ?w, ?h) -> hasAreaInSquareMeters(?r, ?areaInSquareMeters) (8CW)

Since the `areaInSquareMeters` variable is unbound when the multiply built-in is called, the built-in will assign a value so that the built-in predicate evaluates to true. If the variable is bound when the built-in is called, its value will be used directly in predicate evaluation. The use of a variable in any non built-in atom automatically ensures it is bound (assuming the rule fires). (8HQ)

We can use a similar rule to classify a rectangle with an area of over 100 square meters as a `BigRectangle`: (8CV)

- Rectangle(?r) ^ hasWidthInMetres(?r, ?w) ^ hasHeightInMetres(?r, ?h) ^ swrlb:multiply(?areaInSquareMeters, ?w, ?h) ^ swrlb:greaterThan(?100, ?areaInSquareMeters) -> hasAreaInSquareMetres(?r, ?areaInSquareMeters) ^ BigRectangle(?r) (8CX)

This rule illustrates the binding of a variable by one built-in and its subsequent use by another built-in in the same rule. The variable `areaInSquareMeters` is unbound when it is passed to the multiply built-in so a value is assigned to it; when the greater than built-in is called, it is passed this bound value. (8HR)

Binding precedence is from left to right. As mentioned above, the use of a variable in a non built-in atom automatically binds it - these atoms take precedence over all built-in atoms. (8HS)

Unbound variables may potentially occur in any argument position. For example, consider the following two rules, both of which convert a person's salary from pounds to dollars: (8JC)

- Person(?p) ^ hasSalaryInPounds(?p, ?pounds) ^ swrlb:divide(?pounds, ?dollars, ?1.9) -> hasSalaryInDollars(?p, ?dollars) (8JD)

- Person(?p) ^ hasSalaryInPounds(?p, ?pounds) ^ swrlb:multiply(1.9, ?pounds, ?dollars) -> hasSalaryInDollars(?p, ?dollars) (8JE)

In these cases, the unbound `dollars` variable is in the second and third argument positions, respectively. (8JF)

Unbound arguments may have more than one valid binding that satisfies the built-in. For example, the absolute value built-in invocation `swrlb:abs(7, ?x)` could be satisfied by both 7 and -7 as bindings for the variable x. (8V3)

Of course, not all built-ins will perform argument binding. For example, supporting argument binding in the core SWRL `greaterThan` built-in would be counterintuitive. The designer of a built-in must decide which - if any - arguments are to be bound. (8II)

A discussion of the implementation issues of defining SWRL built-ins that bind arguments can be found here. (8HV)

SameAs and DifferentFrom (8JM)

SWRL also supports `sameAs` and `differentFrom` clauses which can be used to determine if individuals refer to the same underlying individual or are distinct. For example, a rule that indicates that individuals are cooperators if they are co-authors on a paper can be written as follows: (8I6)

- Publication(?a) ^ hasAuthor(?x, ?y) ^ hasAuthor(?x, ?z) ^ differentFrom(?y, ?z) -> cooperatedWith(?y, ?z) (8I7)

There is a common temptation to write the above rule without the `differentFrom` atom and assume that variables `y` and `z` will only refer to distinct individuals. However, due to normal rule pattern matching, both variables can refer to the same individual, leading to useless (and possibly erroneous) assertions. (8JN)

More importantly, OWL does not have a unique name assumption. Just because two individual names are different does not mean the names

refer to different individuals. So an `owl:allDifferents` restriction should be defined for all individuals of class `Author` to get correct operation of the above rule. (8JS)

Similarly, SWRL will only deduce that two individuals are the same as each other if there is an explicit `owl:sameAs` OWL restriction defined for the indivuduals in an ontology. (8JT)

OWL Restrictions (8JP)

SWRL also supports the use of OWL restrictions in rules. For example, a rule that classifies an individual as a parent if it is a member of a class with a `hasChild` property defined on it with a minimum cardinality is one can be written as follows: (8HZ)

- (hasChild >= 1)(?x) -> Parent(?x) (8I0)

It is important to note that OWL restrictions used in SWRL rules have identical semantics to OWL restrictions used elsewhere in an OWL ontology. That is, they are logical statements about classes and their associated properties. The rule above matches all individuals for which *it can be proven* (directly or indirectly) that they are members of a class that has the specified cardinality constraint on a `hasChild` property. It does *not* match all individuals in an OWL ontology that have one or more values for a `hasChild` property. Because of OWL's (and SWRL's) open world assumption, this rule may actually match individuals that have no values for the `hasChild` property in the current ontology but for which the existence of such values can be deduced from restrictions. It is not possible to express this type of match in SWRL. (8ID)

The remainder of this FAQ describes the various SWRL language constructs defined in the SWRL Ontology and the presentation syntax for each of them that is supported by the Protege-OWL SWRL Editor. Example rules are provided throughout. There is also a simple example ontology containing a set of SWRL rules in our ontology library at family.swrl.owl. (72Q)

How are conjunction and implication symbols represented? (6ZP)

The conjunction symbol is represented by the "^" character. The implication symbol can be entered using the "->" character pair. (71X)

How are rule variables represented? (6ZR)

A variable is written as an identifier preceded by a "?" character. Example variables are: (6ZS)

- ?x (6ZT)
- ?a-variable (6ZU)
- ?x2 (6ZV)

A variable name may not be the same as an OWL class, property or individual in the same knowledge base. (722)

Only variables that occur in the antecedent of a rule may occur in the consequent (a condition usually referred to as "safety"). (73T)

How are individuals represented? (6ZW)

Individuals are referred to using their OWL name. Example individual names include: (6ZX)

- Fred (6ZY)
- Mary (6ZZ)

An example SWRL rule that uses individuals is: (724)

- Person(Mary) ^ hasSibling(Mary, Fred) ^ Male(Fred) -> hasBrother (Mary, Fred) (725)

This rule will set the `hasBrother` property of Mary to contain Fred. (73U)

How are literals represented? (700)

Numeric literals are simply typed as is. Integers, longs, floats and doubles are supported. The Boolean values true and false are represented using the values "true" and "false". Case is not significant for Boolean values. All other literals must be enclosed in quotation marks. Example literals include: (701)

- 34 (702)
- 4.45 (703)
- -34 (704)
- True (705)

- false (706)
- "" (707)
- "a literal" (708)

Literals may also be qualified by their XML Schema datatypes. All such datatypes must be preceded by the namespace qualifier `xsd:`. (709)

Example datatypes include: (70A)

- xsd:unsignedInt (70B)
- xsd:string (70C)

A literal can be qualified by a datatype by following the literal by the the character "#" and the datatype name. Example qualified literals include: (70E)

- 34#xsd:unsignedInt (70F)
- "a literal"#xsd:string (70G)

XML Schema datatype qualification is not supported in the SWRL Editor at the moment, but will be available shortly. (70H)

How are class atoms represented? (70I)

Class atoms are constructed from an OWL class name followed by one variable or individual name in parenthesis. Example class atoms are: (70J)

- Man(?x) (70K)
- Woman(Mary) (70L)

If used in the antecedent of a rule, a class atom is used to determine if the variable or individual parameter is a member of the specified OWL class. If used in a rule consequent, it is used to classify individuals to be members of the specified class. (73V)

The SWRL Proposal also allows arbitrary OWL expressions to be used instead of a class name in class atoms. The SWRL Editor does not currently support use of these expressions. (81K)

How are SWRL individual property atoms represented? (70N)

View other revisions RSS

Individual property atoms are constructed from an OWL object property name followed by two arguments in parenthesis. Each parameter is either a variable or individual name. An example individual property atom is: (73W)

- hasChild(?x, ?y) (73X)

Other example individual property atoms are: (70O)

- hasParent(Fred, ?x2) (70P)
- hasNephew(?x1, ?x3) (70Q)

How are SWRL datavalued property atoms represented? (70S)

Datavalued property atoms are represented in the same way as individual property atoms. They are constructed from a OWL datatype property name followed by two arguments in parenthesis. The second argument is either a variable representing a data value or a literal value itself. Example data valued property atoms are: (70T)

- hasAge(Mary, 10) (70U)
- hasName(?x1, ?x2) (70V)
- hasName(?x1, "Freddy") (70W)

How are SWRL different individuals and same individual atoms represented? (70X)

These atoms are constructed from the tokens `differentFrom` and `sameAs` followed by two arguments in parenthesis. Example atoms are: (70Y)

- sameAs(?x, ?y) (70Z)
- differentFrom(?x, Mary) (710)

For example, a rule that indicates that individuals are cooperators if they are co-authors on a paper can be written as follows: (8I8)

- Publication(?a) ^ hasAuthor(?x, ?y) ^ hasAuthor(?x, ?z) ^ differentFrom(?y, ?z) -> cooperatedWith(?y, ?z) (8I9)

How are SWRL built-ins represented? (711)

SWRL built-ins are predicates that accept several arguments. A number
Last edited April 9, 2007 13:57 (diff)

of core built-ins for mathematical and string operations are contained in the SWRL Built-in Proposal. These built-ins are defined in the file `swrlb.owl`, which has the namespace `http://www.w3.org/2003/11/swrlb`. (726)

By convention, all core SWRL built-ins are preceded by the namespace qualifier `swrlb`. (714)

Example built-in atoms include: (715)

- swrlb:lessThan(?x, 10) (716)
- swrlb:equals(?x, ?y) (717)

An example SWRL rule using a built-in to indicate that a person with an age of greater than 17 is an adult is: (8D4)

- Person(?p) ^ hasAge(?p, ?age) ^ swrlb:greaterThan(?age, 17) -> Adult(?p) (8D5)

When executed, this rule would classify individuals of class `Person` with an `hasAge` property value of greater than 17 as members of the class `Adult`. (8D6)

The SWRLTab has a subcomponent called the SWRLBuiltInBridge that provides a mechanism to define Java implementations of SWRL built-ins. (8CZ)

How are OWL Restrictons Represented? (8I1)

OWL restrictions as represented by an OWL restriction in parenthesis followed by a single argument. (8I2)

For example, a rule that classifies an individuals as a parent if it is a member of a class with a `hasChild` property defined on it with a minimum cardinality is one can be written as follows: (8HZ)

- (hasChild >= 1)(?x) -> Parent(?x) (8I3)

In the current release of the SWRLTab, restrictions are not supported, but will be shortly. (8I4)

How are data range atom represented? (71B)

Data range atoms have two forms: (1) a datatype name followed by a variable or literal in parenthesis, and (2) a list of brackets-enclosed comma-separated literals followed by a variable or literal in parenthesis. (71C)

An example of the first form would be: (71D)

- xsd:unsignedInt(?x) (71E)

which asserts that variable x must be an unsigned integer. (71F)

Other examples are: (71G)

- xsd:unsignedInt(234) (71H)
- xsd:string(?y) (71I)
- xsd:normalizedString("a string") (71J)

An example of the second form of a data range atom is: (71K)

- [3, 4, 5](?x) (71L)

which asserts that variable x must be one of 3, 4, or 5. (8D0)

Another example of this form would be: (8D1)

- ["red", "green", "blue"](?x) (8D2)

which asserts that variable x must be one of the strings "red", "green" or "blue". (8D3)

In the current release of the SWRLTab, the second form of data range atoms is not supported, but will be shortly. (71N)