SWRL2SPIN: A tool for transforming SWRL rule bases in OWL ontologies to object-oriented SPIN rules

Nick Bassiliades*
Department of Informatics, Aristotle University of Thessaloniki, Greece

Abstract. SWRL is a semantic web rule language that combines OWL ontologies with Horn Logic rules of the RuleML family of rule languages, extending the set of OWL axioms to include Horn-like rules. Being supported by the Protégé ontology editor as well as by popular rule engines and ontology reasoners, such as Jess, Drools and Pellet, SWRL has become a very popular choice for developing rule-based applications on top of ontologies. However, SWRL, being around for more than 10 years now, it is most probable that it will never become a W3C standard; therefore, its scope is difficult to reach out to the industrial world. On the other hand, SPIN has become a de-facto industry standard to represent SPARQL rules and constraints on Semantic Web models, building on the widespread acceptance of the SPARQL query language for querying and processing Linked Open Data. In this paper, we argue that the life of existing SWRL rule-based ontology applications can be prolonged by being transformed into SPIN. To this end, we have developed a prototype tool using SWI-Prolog that takes as input an OWL ontology with a SWRL rule base and transforms SWRL rules into SPIN rules in the same ontology, taking into consideration the object-oriented scent of SPIN, i.e. linking rules to the appropriate ontology classes as derived by analyzing the rule conditions.

Keywords: SWRL, SPIN, SPARQL, OWL, Rules, Ontologies, Semantic Web, Prolog, Transformation

1. Introduction

Rule-based systems have been extensively used in several applications and domains, such as e-commerce, personalization, games, businesses and academia. They offer a simplistic model for knowledge representation for both domain experts and programmers; experts usually find it easier to express knowledge in a rule-like format and programmers usually find rule-based programming easier to understand and manipulate, decoupling computation from control. The first is performed by the rules whereas the latter is determined by the rule engine itself, that is when and how to apply the rules.

The Semantic Web initiative [52] works on standards, technologies and tools in order to give to the information a well-defined meaning, enabling computers and people to work in better cooperation. Ontologies can be considered as a primary key towards this goal since they provide a controlled vocabulary of concepts, each with explicitly defined and machine processable semantics. The Web Ontology Language (OWL) [19] is the W3C recommendation for creating and sharing ontologies on the Web. It provides the means for ontology definition and specifies formal semantics on how to derive new information.

There are mainly two modeling paradigms for the Semantic Web [20]. The first paradigm is based on the notion of the Description Logics [1] on which the OWL is based. In this case, the semantics of OWL ontologies can be handled by DL reasoning systems, such as Pellet [45], RacerPro [15], Fact++ [51] and HermiT [12] that reuse existing DL algorithms, such as tableaux-based algorithms [2]. The other paradigm is based on Horn logic, whereas a subset of the OWL semantics is transformed into rules that are used by a rule engine in order to infer implicit knowledge. There are major differences between these two paradigms, including computational and expressiveness aspects. For example, the DL reasoning engines have a rather
inefficient instance reasoning performance, whereas rules are insufficient to model certain situations related to the open nature of the Semantic Web. The selection of the most suitable modeling paradigm depends on the domain and the needs of the application.

Since description logics and Horn logic are orthogonal in the sense that neither of them is a subset of the other [14], there are two interesting combinations of ontologies and rules, namely their intersection, which is OWL 2 RL, and their union, namely SWRL. OWL 2 RL [33] is an OWL 2 profile aiming at applications that require scalable reasoning without sacrificing too much expressive power. This is achieved by defining a syntactic subset of OWL 2 which is amenable to implementation using rule-based technologies, namely it is the largest syntactic fragment of OWL2 DL that is implementable using rules. The design of OWL 2 RL was inspired by Description Logic Programs [33] and pD* [49]. Obviously, OWL 2 RL is a decidable language, but one that is necessarily less expressive than either the description logic or rules language from which it is formed.

SWRL ([21], [22]) is a semantic web rule language that combines OWL ontologies with Horn Logic rules of the RuleML family of rule languages [43], extending the set of OWL axioms to include Horn-like rules. SWRL is considerably more powerful than either OWL DL or Horn rules alone; however, key inference problems for SWRL are undecidable [21]. Decidability can be regained by restricting the form of admissible rules, by imposing a suitable safety condition [35].

Being supported by the Protégé ontology editor [39] as well as by popular rule engines and ontology reasoners, such as Jess [10], Drools [10] and Pellet [45], SWRL has become a very popular choice for developing rule-based applications on top of ontologies ([5], [7], [17], [24], [30], [36], [38], [46]). However, SWRL being around for more than 10 years now, it is most probable that it will never become a W3C standard; therefore, its scope is difficult to reach out to the industrial world.

On the other hand, SPIN [26] has become a de-facto industry standard to represent SPARQL rules and constraints on Semantic Web models, building on the widespread acceptance of the SPARQL query language [16] for querying and processing Linked Open Data. SPARQL is well supported by numerous engines and databases. This means that SPIN rules can be directly executed on the databases and no intermediate engines with communication overhead need to be introduced. Also, SPIN is more expressive than SWRL, because SPARQL has various features such as UNIONS and FILTER expressions. SPIN has an object-oriented model that arguably leads to better maintainable models than SWRL’s flat rule lists. Finally, SPIN goes far beyond being just a rule language, and also provides means to express constraints and to define new functions and templates.

For all the above reasons, in this paper, we argue that the life of existing SWRL rule-based ontology applications can be prolonged by being transformed into SPIN. To this end, we have developed a tool called SWRL2SPIN, using SWI-Prolog [54] that takes as input an OWL ontology with an SWRL rule base and transforms SWRL rules into SPIN rules in the same ontology, taking into consideration the object-oriented scent of SPIN, i.e. linking rules to the appropriate ontology classes as derived by analyzing the rule conditions. Furthermore, conditions of transformed rules are optimized according to the hosting class by re-ordering condition elements. Our SWRL2SPIN tool is accompanied by a rich implementation of SWRL builtins ([41]); however, the way these builtins have been translated provides room for extensibility in the future to increase coverage.

In the rest of the paper, we briefly review related works on SWRL rule transformations for interchange and/or execution reasons in section 2, and then, we overview SWRL and SPIN syntax and semantics, focusing on their RDF vocabularies, in sections 3 and 0, respectively. In section 5 we present our tool, its transformation methodology, how rules are embedded into classes, how they are optimized and how builtins have been implemented. In section 6 we evaluate the tool and finally, in section 7, we conclude.

2. Related Work

To the best of our knowledge there is no other tool for transforming SWRL rule bases to SPIN rules. In this section, we briefly review existing approaches to transforming SWRL rules into another rule formalism, mainly for execution reasons, i.e. to be able to implement an SWRL rule engine by re-using another rule engine. The purpose of our work is rather to transform SWRL rules into SPIN rules so that SPIN-compliant ontology / rule editors can be used to maintain / extend these rule bases / applications. Thus, it is not exactly similar to just running SWRL rules, but it could also be viewed in this way, since TopSPIN [50] is a rule engine that after the translation of SWRL rules into SPIN rules it is able to execute them and store the results within the OWL ontology.
DL-reasoners do not support the full specification of SWRL because the reasoning becomes undecidable. So, there are different approaches of combining OWL-DL with SWRL reasoning.

**Translate SWRL into First Order Logic** and demonstrate reasoning tasks with a theorem prover. The best-known implementation of this paradigm is Hoolet [3], which is an implementation of an OWL-DL reasoner that uses the first order theorem prover Vampire [41] and supports SWRL [21].

**Translate OWL-DL axioms into rules** and give the axiom rules plus the translated SWRL rules to a rule engine. This approach cannot cover the full expressivity of OWL-DL due to many incompatibilities between Description Logics and Horn Logic. Examples of this paradigm are the implementations of the SWRLTab [37] of Protégé [39], such as the older SWRLJessTab plugin [13], which is available only for the v. 3.5 of Protégé and the newer SWRLDroolsTab plugin [48], which is available both for Protégé 5.2 and 3.5. These plugins translate the ontology axioms into facts of the Jess [10] and Drools [8] production rule engines, respectively, and SWRL rules into production rules that materialize the conclusions of the rules as derived facts. Then the production rule engines run the SWRL rules along with the entailment rules that implement part of the DL-reasoning process and the results (materialized inferred axioms of the SWRL rules) are copied back to Protégé.

Other systems that support a similar functionality, using forward chaining rule engines are Bossam [23], a forward chaining deductive (Datalog) rule engine that supports SWRL with minimal builtin support for math and string functions, and SWRL2COOL [42] a translator of SWRL rules into CLIPS production rules to accompany the O-Device production rule-based OWL reasoner [31], with also limited support for math and comparison builtins.

There are also approaches to integrate OWL axioms and SWRL rules into backward chaining rule engines, such as Prolog, namely [18] and [44], which also have limited builtin support. Furthermore, the SWRL-IQ plugin [9] for Protégé 3.x supported backward chaining querying of OWL ontologies and SWRL rule reasoning based on XSB Prolog. SWRL-IQ supports 47 SWRL builtins, including list builtins.

**Expand an existing OWL-DL reasoner** based on the tableaux algorithm. Most popular DL-reasoners, such as Pellet [45], HermiT [12], Racer [15], do support SWRL reasoning for DL-safe rules. Pellet supports all of the SWRL builtins except for Lists and provides support for only the first 5 Builtins for Date, Time, and Duration. Pellet will almost certainly never support the List builtins because of OWL DL restrictions. However, all date, time, and duration builtins could be provided in the future. HermiT also supports SWRL DL-safe rules, but with no builtin support [11]. Racer [15] supports processing of rules in a SWRL-based syntax by translating them into nRQL rules; there is no evidence as to whether SWRL builtins are supported. Protégé 5.2 [39] includes a Rules view in its Ontology Views that supports SWRL rules through the above DL reasoners.

KAON2 [34], a rather different DL-reasoner based on reducing a SHIQ(D) knowledge base to a disjunctive Datalog program also supports the so-called DL-safe subset [35] of SWRL, but again without any evidence for builtin support. KAON2 can be integrated to Protégé through a DIG interface [4].

Stardog [47] is an RDF database or triplestore that rewrites queries to answer questions using SWRL inferences. Stardog supports two different syntaxes for defining rules. The first is native Stardog Rules syntax based on SPARQL. The second is the de facto standard RDF/XML syntax for SWRL. It has the advantage of being supported in many tools; but it’s syntax is awkward. Stardog supports 55 SWRL builtins.

Finally, there are some works ([32], [53]) that translate SWRL rules into a rule meta-model for rule interchange reasons among various rules formats, such as R2ML (REWERSE Rule Markup Language) ([32], [40]) and RIF ([25], [53]). The purpose of these translations is merely the interchange of rules, preserving the basic SWRL atom semantics, but without covering the built-ins which are left to the implementer.

### 3. Semantic Web Rule Language

The Semantic Web Rule Language (SWRL) [22] is a proposed language for the Semantic Web that can be used to express rules, combining OWL DL or OWL Lite with the Unary/Binary Datalog RuleML sub-languages of the Rule Markup Language. SWRL extends the set of OWL axioms to include Horn-like rules. It thus enables Horn-like rules to be combined with an OWL knowledge base. SWRL has the full power of OWL DL, but at the price of decidability and practical implementations. However, decidability can be regained by restricting the form of admissible rules, typically by imposing a suitable safety condition [35].

Rules are of the form of an implication between an antecedent (body) and consequent (head). The intended meaning can be read as: whenever the condi-
tions specified in the antecedent hold, then the conditions specified in the consequent must also hold. Both the antecedent (body) and consequent (head) consist of zero or more atoms. An empty antecedent is treated as trivially true (i.e., satisfied by every interpretation), so the consequent must also be satisfied by every interpretation; an empty consequent is treated as trivially false (i.e., not satisfied by any interpretation), so the antecedent must also not be satisfied by any interpretation. Multiple atoms are treated as a conjunction. Note that rules with conjunctive consequents could easily be transformed (via the Lloyd-Topor transformations [29]) into multiple rules each with an atomic consequent. Atoms in these rules can be of the form C(x), P(x,y), sameAs(x,y) or differentFrom(x,y), where C is an OWL description, P is an OWL property, and x, y are either variables, OWL individuals or OWL data values.

SWRL has various representation syntaxes: abstract, human readable, XML concrete and RDF concrete. Listing 1 shows an SWRL rule example in human readable syntax that states “when a student ?s attends a course ?c that is taught by a faculty member ?f, then the student ?s knows the faculty member ?f”.

Listing 1. Sample SWRL rule in human readable syntax

```swrl
uni:Student(?s)∧uni:attends(?s,?c)∧uni:isTaughtBy(?c,?f) → uni:knows(?s,?f)
```

Listing 2 shows how this rule is represented in the RDF concrete syntax. Rules are instances of the swrl:Imp class. The head and body of the rule are lists of atoms (swrl:AtomList); each atom can be one of classAtom, IndividualPropertyAtom, DatavaluedPropertyAtom, SameIndividualAtom, DifferentIndividualsAtom, or BuiltInAtom. All but the builtin atoms have one or two arguments (properties swrl:argumentNN); additionally classAtom has a classPredicate property, whereas the PropertyAtoms have a propertyPredicate property. BuiltInAtoms have a list of arguments instead and the name of the builtin function. Arguments can be variables, declared as instances of the swrl:Variable class, datatype constants, in the Value^^Datatype format, or individuals, i.e. instances of an OWL class.

Listing 2. Sample rule in SWRL RDF concrete syntax

```swrl
unis: a swrl:Variable .
unio: a swrl:Variable .
unif: a swrl:Variable .

[ rdf:type swrl:Imp ;
  swrl:body [ a swrl:AtomList ;
    rdf:first [ a swrl:ClassAtom ;
      swrl:classPredicate uni:Student ;
      swrl:argument1 unisz ] ;
  rdf:rest [ a swrl:AtomList ;
    rdf:first [ a swrl:IndividualPropertyAtom ;
      swrl:propertyPredicate uni:attends ;
      swrl:argument1 unisz ;
      swrl:argument2 unio ] ;
  rdf:rest [ a swrl:AtomList ;
    rdf:first [ a swrl:IndividualPropertyAtom ;
      swrl:propertyPredicate uni:isTaughtBy ;
      swrl:argument1 unio ;
      swrl:argument2 unif ] ;
  rdf:rest rdf:nil ]
];

swrl:head [ a swrl:AtomList ;
  rdf:first [ a swrl:IndividualPropertyAtom ;
    swrl:propertyPredicate uni:knows ;
    swrl:argument1 unisz ;
    swrl:argument2 unif ] ;
  rdf:rest rdf:nil ]
].
```
4. SPARQL Inferencing Notation

Modeling languages for the semantic web, such as RDF Schema [6] and OWL [19], provide mechanisms for capturing the static structure of data, i.e. they are used to define classes, properties and relationships between these conceptual entities. While they define axiomatic definitions of data structures, describing general computational behavior of objects is not within their scope. On the other hand, object oriented languages provide well-known mechanisms for defining object behavior by describing classes and associating methods with class members. Object oriented methods often formalize how the modification of one attribute implies changes to other attributes. Another common purpose of methods is to capture constraints to ensure that the state of the objects remains within the bounds that the class designer had intended.

The SPARQL Inferencing Notation (SPIN) [26] combines concepts from object oriented languages, query languages, and rule-based systems to describe object behavior on the semantic web. One of the basic ideas of SPIN is to link class definitions with SPARQL queries to capture constraints and rules that formalize the expected behavior of those classes. SPARQL is used because it is an existing WC3 standard [16] with well-formed query semantics across RDF data, has existing widespread use amongst most RDF query engines and graph stores, and provides sufficient expressivity for both queries and general computation of data. To facilitate storage and maintenance, SPARQL queries are represented in RDF triples, using the SPIN SPARQL Syntax [28].

The SPIN Modeling Vocabulary [27] defines a collection of properties and classes that can be used to link RDFS and OWL classes with SPARQL queries. For example, the class ex:Department can define a property spin:rule that points to a SPARQL CONSTRUCT query that computes the value of ex:studentProfessorRatio based on the values of ex:enrolledStudents and ex:numberOfFaculty. These properties follow existing SPARQL standards, and the execution of these constructs can be efficiently handled by any SPARQL processor. Since SPIN is entirely represented in RDF, rules and constraints can be shared on the web together with the class definitions they are associated with. The attachment of rules to classes also encourages a style in which rules are locally scoped and thus easier to maintain, avoiding the spaghetti code of "flat" rule languages, such as SWRL.

Other important features of SPIN include (a) SPIN templates, which are parameterized SPARQL queries, and (b) SPIN functions, i.e. user-defined SPARQL functions. Both are not further discussed in the scope of this paper, since SWRL2SPIN does not use them.

The SPIN class description vocabulary defines several RDF properties that can be used to attach SPARQL queries to classes. The property spin:rule can be used by SPIN reasoning engines to construct inferred RDF triples from the currently asserted information in the model. The SPARQL queries referenced by the SPIN properties are interpreted in the context of the associated class. At run-time, the SPARQL variable ?this is (by default) pre-bound with instances of the class and its sub-classes. Typically, the query itself does not need to bind ?this to any value in the WHERE clause. The execution context (e.g., inference engine) will do this before the query is executed.

SPIN takes an object-oriented world view on Semantic Web models, in which SPARQL queries play a similar role to functions and methods. Inheritance (expressed using rdfs:subClassOf) is treated in the sense that any query/rule defined for super-classes will also be applied to subclasses. In other words, SPIN class descriptors can only "narrow down" and further restrict what has been defined further up in the class hierarchy. In this spirit, global class descriptions are those that are attached to the root class rdfs:Resource or its OWL equivalent owl:Thing. Those global queries may not even mention ?this at all.

The property spin:rule links an rdfs:Class with a SPARQL CONSTRUCT query that defines an inference rule that determines how additional triples can be inferred from what is stated in the WHERE clause. For each binding of the pattern in the WHERE clause of the rule, the triple templates from the CONSTRUCT clause are instantiated and added as inferred triples to the underlying model. At query execution time, the SPARQL variable ?this is bound to the current instance of the class.

The example in Listing 3 defines a SPIN rule (in textual SPARQL format), attached to class uni:Student via the spin:rule property, that infers the value of the uni:knows property from values of uni:attends and uni:isTaughtBy. Listing 4 shows how the same rule is represented using the SPIN modeling vocabulary.

SPIN rules are instances of the sp:Construct class; the rule "head" is defined with the sp:templates property whereas the sp:where property defines the rule "body". The above properties contain lists of triple patterns (sp:subject, sp:predicate, sp:object). Other SPARQL query elements contained in rule "body" can be TriplePath, Filter, Bind, Optional, Union,
NamedGraph, SubQuery, NotExists, Minus, Service, and Values. In the following we only present the first three, since they are the only ones used in the SWRL2SPIN tool.

A TriplePath is similar to a triple pattern, but instead of an sp: predicate, has an sp:path property, whose value can be one of several types, sp: SeqPath being the most usual one. The sequential steps of the path are represented through consecutive sp:pathNN properties. The representation is more complex when an arbitrary length path matching is involved, i.e., when the * operator is used.

Filter elements are blank nodes, instances of sp:Filter that have property sp: expression, pointing to an expression that can be evaluated to true or false. Expressions are actually function calls which are represented as instances of the function’s URI. All other properties of expressions (or function calls) are interpreted as arguments, using consecutive sp:argNN properties. However, other property names can be used as well, depending in the function. Arguments can be either datatype constants or variables, which are blank nodes with an sp: varName property whose value is a string. E.g., the FILTER (?y > 30) expression is shown in Listing 5.

The BIND keyword assigns a computed value to a variable. Bind assignments in the rule “body” are represented as instances of the class sp:Bind, having an sp: variable property to point at the variable on the right side of the assignment. The property sp: expression points to the root of the expression tree that delivers the computed value, in much a similar way to filter expressions (i.e., function calls). E.g., the expression BIND ((?x * 2) AS ?y) is shown in Listing 6.

A more detailed presentation of the SPIN modelling vocabulary and syntax can be found at the respective references and are out of the scope of this paper.

5. SWRL2SPIN

The SWRL2SPIN tool accepts at its input an OWL ontology with SWRL rules embedded in the ontology using the RDF concrete syntax of SWRL, as exported by tools such as Protégé combined with the SWRLtab plugin. The tool produces at its output an OWL ontology (just copying the input one) extended by SPIN rules that have been created by translating the SWRL rules. SPIN rules are embedded inside their corresponding classes, following the OO nature of SPIN, instead of having a flat rule base as in SWRL. Furthermore, the ?this variable of SPIN is used to identify instances of the rule-embedding class, therefore SWRL condition elements that identify the class of the corresponding instances are removed, speeding-up, thus, rule execution. Finally, the same SWRL may involve instances of multiple classes, so our tool generates multiple versions / views of a rule, optimized for each of the classes, separately.
The main procedure for translating a SWRL rule into a SPIN rule involves mapping classes and properties of the RDF concrete syntax of SWRL into corresponding classes and properties of the SPIN modeling vocabulary, in a recursive way starting from swrl:Imp instances, following an almost one-to-one mapping scheme shown in Table 1. The only exception to the straightforward mapping is the SWRL built-ins whose translation is customized for each function. We will discuss translation of built-ins in section 5.3.

In the following, we give an example of translating a SWRL rule without built-ins to a SPIN rule. Consider the SWRL rule in Listing 1 that is translated into the SPIN rule in Listing 7. The actual translation is between the RDF representations of the SWRL and SPIN rules, shown in Listing 8 and Listing 9, respectively.

### Table 1. Correspondence between SWRL and SPIN constructs

<table>
<thead>
<tr>
<th>SWRL</th>
<th>SPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>swrl:Imp</td>
<td>sp:Construct</td>
</tr>
<tr>
<td>swrl:head</td>
<td>sp:templates</td>
</tr>
<tr>
<td>swrl:body</td>
<td>sp:where</td>
</tr>
<tr>
<td>swrl:ClassAtom</td>
<td>sp:subject &lt;Arg&gt;</td>
</tr>
<tr>
<td>swrl:classPredicate</td>
<td>sp:predicate rdf:type</td>
</tr>
<tr>
<td>swrl:argument1 &lt;Arg&gt;</td>
<td>sp:object &lt;Class&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>swrl:IndividualPropertyAtom</td>
<td>sp:subject &lt;Arg1&gt;</td>
</tr>
<tr>
<td>swrl:propertyPredicate &lt;Prop&gt;</td>
<td>sp:predicate &lt;Prop&gt;</td>
</tr>
<tr>
<td>swrl:argument1 &lt;Arg1&gt;</td>
<td>sp:object &lt;Arg2&gt;</td>
</tr>
<tr>
<td>swrl:argument2 &lt;Arg2&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>swrl:SameIndividualAtom</td>
<td>sp:subject &lt;Arg1&gt;</td>
</tr>
<tr>
<td>swrl:argument1 &lt;Arg1&gt;</td>
<td>sp:predicate owl:sameAs</td>
</tr>
<tr>
<td>swrl:argument2 &lt;Arg2&gt;</td>
<td>sp:object &lt;Arg2&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>swrl:DifferentIndividualsAtom</td>
<td>sp:subject &lt;Arg1&gt;</td>
</tr>
<tr>
<td>swrl:argument1 &lt;Arg1&gt;</td>
<td>sp:predicate owl:differentFrom</td>
</tr>
<tr>
<td>swrl:argument2 &lt;Arg2&gt;</td>
<td>sp:object &lt;Arg2&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>swrl:DatavaluedPropertyAtom</td>
<td>sp:subject &lt;Arg1&gt;</td>
</tr>
<tr>
<td>swrl:propertyPredicate &lt;Prop&gt;</td>
<td>sp:predicate owl:hasValue</td>
</tr>
<tr>
<td>swrl:argument1 &lt;Arg1&gt;</td>
<td>sp:object &lt;Arg2&gt;</td>
</tr>
<tr>
<td>swrl:argument2 &lt;Arg2&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>swrl:BuiltInAtom</td>
<td>Customized translation</td>
</tr>
<tr>
<td>swrl:builtin &lt;Fun&gt;</td>
<td></td>
</tr>
<tr>
<td>swrl:arguments &lt;Args&gt;</td>
<td></td>
</tr>
<tr>
<td>swrl:Variable &lt;Var&gt;</td>
<td>sp:varName &quot;&lt;Var&gt;&quot;</td>
</tr>
<tr>
<td>&lt;Value&gt; ^ &lt;DataType&gt;</td>
<td>&lt;Value&gt; ^ &lt;DataType&gt;</td>
</tr>
<tr>
<td>&lt;Individual&gt;</td>
<td>&lt;Individual&gt;</td>
</tr>
</tbody>
</table>

Listing 7. Translation of SWRL rule of Listing 1 into SPIN

```
CONSTRUCT {
    ?x :knows ?z .
} WHERE {
    ?x rdf:type :Student .
    ?y :isTaughtBy ?z .
}
```

Listing 8. Example of an input SWRL rule in RDF concrete syntax

```
x rdf:type swrl:Variable .
y rdf:type swrl:Variable .
z rdf:type swrl:Variable .
[ rdf:type swrl:Imp ;
  swrl:body [ rdf:type swrl:AtomList ;
    rdf:first [ rdf:type swrl:ClassAtom ;
      swrl:classPredicate :Student ;
      swrl:argument1 :x ] ;
    rdf:rest [ rdf:type swrl:AtomList ;
      rdf:first [ rdf:type swrl:IndividualPropertyAtom ;
        swrl:propertyPredicate :attends ;
        swrl:argument1 :x ;
        swrl:argument2 :y ] ;
    rdf:rest [ rdf:type swrl:AtomList ;
      rdf:first [ rdf:type swrl:IndividualPropertyAtom ;
        swrl:propertyPredicate :isTaughtBy ;
        swrl:argument1 :y ;
        swrl:argument2 :z ] ;
swrl:head [ rdf:type swrl:AtomList ;
  rdf:first [ rdf:type swrl:IndividualPropertyAtom ;
    swrl:propertyPredicate :knows ;
    swrl:argument1 :x ;
    swrl:argument2 :z ] ;
```
Listing 9. Example of an output SPIN rule in SPIN modelling vocabulary

5.1. Embedding SPIN rules in Classes

One of the unique features of SPIN compared to SWRL is the ability to embed rules into classes and treat them in an OO way as inheritable behaviors (aka methods). By doing so, instances of the embedding class can be identified by variable ?this. In SWRL2SPIN we:

1. identify variables in the rule body that refer to class instances that play the role of the “subject” in the triple patterns;
2. identify the classes these variables refer to;
3. generate as many rules as the number of the different classes “discovered” in step 2;
4. rewrite each rule of step 3 so that:
   – corresponding variable names are replaced by ?this
   – rdf:type triple patterns that refer to ?this are removed from the rule body
   – triple patterns in the rule body are re-ordered so that the order of triple patterns is optimal.

For step 1, we collect all the variables in the rule body that are

1. arguments of a swrl:ClassAtom construct;
2. first arguments of a swrl:IndividualPropertyAtom or a swrl:DatavaluedPropertyAtom construct;

The rationale behind this is that subjects of triple patterns can only play the role of the “referenced object”, i.e. the object that exhibits the class behavior. For our example, the collected variables are:

1. variable ?x, due to Student(?x) class atom
2. variable ?y, due to isTaughtBy(?y,?z) individual property atom.

In step 2, we identify the class that the instantiations of the above variables belong to by:

1. checking if they are arguments of a swrl:ClassAtom construct;
2. retrieving the domain / range of arguments of swrl:IndividualPropertyAtom constructs;
3. retrieving the domain of arguments of swrl:DatavaluedPropertyAtom constructs.

For the ongoing SWRL rule example, the collected variables ?x and ?y belong to classes Student and Course, respectively. The former is discovered from the Student(?x) class atom, while the latter is discovered from the domain of the isTaughtBy(?y,?z) individual property atom and / or the range of the attends(?x,?y) atom. Thus, the SWRL rule is actually converted into two SPIN rules stored at classes Student (Listing 10) and Course (Listing 11), respectively:

```
CONSTRUCT {    # @Student
   ?this :knows ?z .
}
WHERE {
   ?this :attends ?y .
   ?y :isTaughtBy ?z .
}
```

Listing 10. SPIN rule embedded at class Student

```
CONSTRUCT {    # @Course
   ?x :knows ?z .
}
WHERE {
   ?x rdf:type :Student .
   ?x :attends ?this .
   ?this :isTaughtBy ?z .
}
```

Listing 11. SPIN rule embedded at class Course
5.2. Optimizing SPIN rules

In the previous example, the body of the SPIN rule at class Course has two triple patterns that contain variable ?this and one triple pattern for variable ?x ranging over all instances of class Student, following the initial ordering of the atoms at the body of the SWRL rule. However, it is evident that this ordering leads to a very inefficient SPARQL query execution, since variable ?x can be instantiated with many values, whereas variable ?this instantiates each time only with one value. So, SWRL2SPIN re-orders the triple patterns in the body of converted/ embedded SPIN rules using the following heuristics:

1. Triple patterns that contain variable ?this at the subject of the triple pattern are placed first;
2. Triple patterns that contain variable ?this at the object of the triple pattern are placed second;
3. Triple patterns that contain the properties owl:sameAs or owl:differentFrom are placed after the triple patterns that instantiate the variables of their subject and object;
4. The order of all other triple patterns remains unchanged.

According to the above, the triple patterns of the body of the SPIN rule at class Course are re-ordered as shown in Listing 12.

Listing 12. Optimized SPIN rule at class Course

5.3. Implementing SWRL builtins

The translation of the SWRL builtins does not follow the straightforward approach for the rest of the SWRL atoms and it depends on the nature of each function and the existence of equivalent SPIN or SPARQL functions. More specifically, SWRL specification [22] has defined 78 builtin functions classified across the categories: Comparisons, Mathematics, Boolean Values, Strings, Date, Time and Duration, URIs, and Lists. Currently, SWRL2SPIN implements more than half of the SWRL builtins (41), mostly in the categories: Comparisons, Mathematics, Strings, and Lists. For the Date, Time and Duration category, we implemented only the swrlb:date function.

Table 2 contains all the supported builtins, the category they belong to and how their conversion to SPIN/SPARQL was achieved. As it can be observed, the conversion of the builtins falls into ten categories: binary filter, associative infix assign, binary infix assign, unary assign, assign function, filter function, magic property, complex assign, complex filter, and complex expression. Filter-type conversions lead to SPARQL FILTER Boolean expressions, whereas assign-type conversions lead to BIND expressions. Simple mathematical comparisons and operations are treated as binary infix mathematical operations, such as >= or -. Addition and multiplication in SWRL builtins can have an arbitrary number of arguments, so they are treated as associative binary infix operators. Finally, there are also simple unary operators, e.g. minus.

Another large category is SWRL builtin functions with an exact equivalent SPIN / SPARQL function, as e.g. round, replace, and contains. The conversion of these functions is straightforward, as in the FILTER case all arguments of the SWRL builtin become arguments of the SPIN / SPARQL function, whereas in the BIND case the first argument of the SWRL builtin becomes the variable to be bound in the SPIN / SPARQL BIND expression, whereas the rest of the arguments of the SWRL builtin become the arguments of the SPIN / SPARQL function.

As discussed in Section 0, FILTER and BIND expressions both have an sp:expression property that contains the mathematical or functional SPARQL expression; BIND also has an sp:variable for the assigned variable. All expressions belong to a type, which is the name of the main SPARQL function in the expression, e.g. sp:gt, sp:case, etc. In the case of the complex functional expressions, the outer function is the type of the FILTER expression, e.g. sp:contains in the case of the containsIgnoreCase SWRL builtin. The argument list of the SWRL builtin (property swrl:arguments) is treated as explained above, generating sp:argNN properties of the SPARQL expression / function. The only exception is the spif:cast function, whose second argument is represented by an arg:datatype property. The values of the sp:argNN properties can be SPIN variables, datatype constants, individuals or nested SPARQL functions / expressions.

The rest of the SWRL builtins are treated as Complex cases, meaning that their translation involves the combination of more than one simple functions, as discussed above. Complex cases can be filters, assignments or general SPARQL expressions (graph patterns) and they are treated in an ad-hoc manner. For example, the integerDivide builtin is translated as a division and a cast to integer, whereas the pow builtin is
translated as repetitive multiplication using recursion. List builtins are of special interest because their translation cannot be performed using SPIN/SPARQL functions, but can be treated using SPARQL path expressions. For example, the member builtin is translated into a recursive path expression combining RDF:first and RDF:rest. The translation of the length builtin is the most complicated one because it requires a SPARQL subquery that counts all the elements in the list, i.e., all possible iterations of the RDF:rest property in the RDF:rest* recursive path.

Table 2. SWRL2SPIN support for SWRL builtins

<table>
<thead>
<tr>
<th>SWRL Builtin</th>
<th>Conversion Category</th>
<th>SPIN/SPARQL Function</th>
<th>op</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>greaterThan(?x, ?y)</td>
<td>Compare</td>
<td>binary filter</td>
<td>&gt;</td>
<td>FILTER (?x &gt; ?y)</td>
</tr>
<tr>
<td>lessThan(?x, ?y)</td>
<td>Compare</td>
<td>binary filter</td>
<td>&lt;=</td>
<td>FILTER (?x &lt;= ?y)</td>
</tr>
<tr>
<td>lessThanOrEqual(?x, ?y)</td>
<td>Compare</td>
<td>binary filter</td>
<td>&lt;=</td>
<td>FILTER (?x &lt;= ?y)</td>
</tr>
<tr>
<td>equal(?x, ?y)</td>
<td>Compare</td>
<td>binary filter</td>
<td>=</td>
<td>FILTER (?x = ?y)</td>
</tr>
<tr>
<td>notEqual(?x, ?y)</td>
<td>Compare</td>
<td>binary filter</td>
<td>/=</td>
<td>FILTER (?x /= ?y)</td>
</tr>
<tr>
<td>add(?x, ?y)</td>
<td>Math</td>
<td>associative infix assign</td>
<td>+</td>
<td>BIND ((?x + ?y) AS ?z)</td>
</tr>
<tr>
<td>subtract(?x, ?y)</td>
<td>Math</td>
<td>binary infix assign</td>
<td>-</td>
<td>BIND ((?x - ?y) AS ?z)</td>
</tr>
<tr>
<td>divide(?x, ?y)</td>
<td>Math</td>
<td>binary infix assign</td>
<td>/</td>
<td>FILTER (?x / ?y AS ?z)</td>
</tr>
<tr>
<td>unaryPlus(?x)</td>
<td>Math</td>
<td>unary assign</td>
<td>+</td>
<td>BIND (+(?x) AS ?y)</td>
</tr>
<tr>
<td>unaryMinus(?x)</td>
<td>Math</td>
<td>unary assign</td>
<td>-</td>
<td>BIND (-(?x) AS ?y)</td>
</tr>
<tr>
<td>abs(?x)</td>
<td>Math</td>
<td>assign function</td>
<td></td>
<td>BIND(abs(?x) AS ?y)</td>
</tr>
<tr>
<td>ceiling(?x)</td>
<td>Math</td>
<td>assign function</td>
<td></td>
<td>BIND(ceiling(?x) AS ?y)</td>
</tr>
<tr>
<td>floor(?x)</td>
<td>Math</td>
<td>assign function</td>
<td></td>
<td>BIND(floor(?x) AS ?y)</td>
</tr>
<tr>
<td>round(?x)</td>
<td>Math</td>
<td>assign function</td>
<td></td>
<td>BIND(round(?x) AS ?y)</td>
</tr>
<tr>
<td>stringConcat(?x, ?y, , ?z)</td>
<td>Strings</td>
<td>assign function</td>
<td></td>
<td>BIND(CONCAT(?x, ?, ?y, ?z) AS ?y)</td>
</tr>
<tr>
<td>stringLength(?x)</td>
<td>Strings</td>
<td>assign function</td>
<td></td>
<td>BIND(STRINGLEN(?x) AS ?y)</td>
</tr>
<tr>
<td>upperCase(?x)</td>
<td>Strings</td>
<td>assign function</td>
<td></td>
<td>BIND(UCASE(?x) AS ?y)</td>
</tr>
<tr>
<td>lowerCase(?x)</td>
<td>Strings</td>
<td>assign function</td>
<td></td>
<td>BIND(LCASE(?x) AS ?y)</td>
</tr>
<tr>
<td>substrBefore(?x, ?y, ?z)</td>
<td>Strings</td>
<td>assign function</td>
<td></td>
<td>BIND(STRBEFORE(?x, ?y, ?z) AS ?y)</td>
</tr>
<tr>
<td>substrAfter(?x, ?y, ?z)</td>
<td>Strings</td>
<td>assign function</td>
<td></td>
<td>BIND(STRAFTER(?x, ?y, ?z) AS ?y)</td>
</tr>
<tr>
<td>replace(?x, ?y, ?z)</td>
<td>Strings</td>
<td>assign function</td>
<td></td>
<td>BIND(REPLACE(?x, ?y, ?z) AS ?y)</td>
</tr>
<tr>
<td>endsWith(?x, ?y)</td>
<td>Strings</td>
<td>filter function</td>
<td></td>
<td>FILTER STRENDS(?x, ?y)</td>
</tr>
<tr>
<td>startsWith(?x, ?y)</td>
<td>Strings</td>
<td>filter function</td>
<td></td>
<td>FILTER STRSTARTS(?x, ?y)</td>
</tr>
<tr>
<td>contains(?x, ?y)</td>
<td>Strings</td>
<td>filter function</td>
<td></td>
<td>FILTER CONTAINS(?x, ?y)</td>
</tr>
<tr>
<td>match(?x, ?y)</td>
<td>Strings</td>
<td>filter function</td>
<td></td>
<td>FILTER REGEX(?x, ?y)</td>
</tr>
<tr>
<td>tokenize(?x, ?y)</td>
<td>Strings</td>
<td>magic property</td>
<td></td>
<td>BIND(spf:split(?x, ?y) AS ?z)</td>
</tr>
<tr>
<td>integerDivide(?x, ?y)</td>
<td>Math</td>
<td>complex assign</td>
<td></td>
<td>BIND(spf:divide(?x, ?y, AS ?z)</td>
</tr>
<tr>
<td>pow(?x, ?y)</td>
<td>Math</td>
<td>complex assign</td>
<td></td>
<td>BIND(spf:pow(?x, ?y) AS ?z)</td>
</tr>
<tr>
<td>normalizeSpace(?x)</td>
<td>Strings</td>
<td>complex assign</td>
<td></td>
<td>BIND(REPLACE(REPLACE(REPLACE(?x, &quot;\s+&quot;, &quot;\s&quot;), &quot;\s+&quot;, &quot;\s&quot;), &quot;\s+&quot;, &quot;\s&quot;) AS ?y)</td>
</tr>
<tr>
<td>date(?x, ?y, ?z)</td>
<td>Date, Time, Duration</td>
<td>complex assign</td>
<td></td>
<td>BIND(spf:date(?x, ?y, ?z) AS ?y)</td>
</tr>
<tr>
<td>containsIgnoreCase(?x, ?y)</td>
<td>Strings</td>
<td>complex filter</td>
<td></td>
<td>FILTER CONTAINS(LCASE(?x), LCASE(?y))</td>
</tr>
<tr>
<td>stringEqualIgnoreCase(?x, ?y)</td>
<td>Strings</td>
<td>complex filter</td>
<td></td>
<td>FILTER(LCASE(?x) = LCASE(?y))</td>
</tr>
<tr>
<td>empty(?x)</td>
<td>Lists</td>
<td>complex filter</td>
<td></td>
<td>FILTER(?x == nil)</td>
</tr>
<tr>
<td>first(?x)</td>
<td>Lists</td>
<td>complex expr</td>
<td></td>
<td>?x rdf:first ?z</td>
</tr>
<tr>
<td>rest(?x)</td>
<td>Lists</td>
<td>complex expr</td>
<td></td>
<td>?x rdf:rest ?y</td>
</tr>
<tr>
<td>member(?x, ?y)</td>
<td>Lists</td>
<td>complex expr</td>
<td></td>
<td>?y rdf:rest* rdf:first ?z</td>
</tr>
<tr>
<td>length(?x)</td>
<td>Lists</td>
<td>complex expr</td>
<td></td>
<td>SELECT COUNT(?x) AS ?y</td>
</tr>
<tr>
<td>WHERE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As an example, consider the SWRL rule in Listing 13 which is translated in the SPIN rule at class Person (Listing 14). Specifically, the RDF concrete syntax for the SWRL builtin atom is shown in Listing 15, whereas the converted SPIN / SPARQL expression is shown in Listing 16.

![RDF syntax for the SWRL builtin example](image)

Listing 15. RDF syntax for the SWRL builtin example

A special case is magic properties which are supported by many SPARQL engines to dynamically compute values at query time. A magic property usually is implemented by a calculation function that determines bindings of the variables on the left or right side of the predicate. SPIN enables users to define such magic properties, in a very similar way as SPIN Functions, but providing greater flexibility. In contrast to BIND/FILTER functions, magic properties can return multiple values. Furthermore, any input or output variable may be unbound; it is the task of the magic property to find their potential bindings. The magic property spif:split is used in SWRL2SPIN to translate the swrlb:tokenize SWRL builtin. The first variable of the SWRL builtin generates multiple bindings. When the spif:split magic property is used, the subject of the “triple pattern” generates multiple alternative bindings. Magic properties are treated in an ad-hoc manner in SWRL2SPIN, since their definition and behavior does not follow a regular pattern.

The rest of the SWRL builtins will be implemented as a future work, most probably as complex conversion cases or as user-defined magic properties. We notice here that the only other SWRL related tool supporting functions for RDF lists is the SWRL-IQ plugin [9] for Protégé 3.x.

6. Evaluation

To evaluate SWRL2SPIN we have initially generated use cases of a University ontology with various SWRL rules in Protégé [39], including all the SWRL builtins of Table 2. Then we have used the SWRL-DroolsTab [48] to run SWRL rules and identify all the inferences. Consequently, we have converted the SWRL use cases through SWRL2SPIN and we have tested the generated SPIN rules using TopSPIN in TopBraib Composer FE [50] for equivalent inferences. The results were found identical for all use cases, except the ones that could not be run in SWRLDrools.

We have also performed a scalability test for the translation time of SWRL2SPIN tool. Results shown in Fig. 1 indicate that the translation time is linear to the number of rules, which was expected since each rule is translated individually, even if there are common variables or other constructs among SWRL rules.

1 We have used the SWRLTab editor of both Protégé 3.5 and 5.2.

2 Except for builtins not supported by Protégé SWRLTab [37].
Rule 1 is the one in Listing 1, with 3 atoms in the rule body, while Rule 2 (Listing 17) has 6 atoms in the body, including one builtin. The translation time per rule also depends on the number of atoms and the number and type of builtins. The average translation time per rule is about 0.72 msec for Rule 1 and 2.82 msec for Rule 2.

![Graph showing translation time scalability](image)

**Fig. 1. Rule translation time scalability.**

Listing 17. Sample SWRL rule in human readable syntax

Finally, we have evaluated the optimized SPIN rules (section 5.2) of SWRL2SPIN against their unoptimized version. For this we have used the unoptimized rule at Listing 11 against the optimized rule at Listing 12 in an ontology with 100K student instances that all attend the same course with one teacher. The inference took 1256.93 msec (on average) for the optimized rule version at TopBraid against 1414.61 msec for the un-optimized rule. Results are statistically significant with a p-value equal to 0.0208 < 0.05. All tests were performed on a Windows 10 PC with Intel i7-4770 @ 3.40GHz, 8 GB RAM and SSD.

7. Conclusions

In this paper we have argued that SPIN is a more promising de-facto industrial standard for the future of combining ontologies and rules, because it builds upon the widespread use of SPARQL. Furthermore, SWRL has been around for quite a while, not being able to achieve a W3C recommendation status. SPIN also offers more expressivity than SWRL due to constructs like FILTER and UNION, and also offers object-orientation by being able to store rules to classes as behaviors to be inherited through the class hierarchy. Thus, we believe that existing large SWRL projects can benefit from being translated into SPIN rules.

To this end we have developed in Prolog and presented the SWRL2SPIN prototype tool\(^3\) that translates ontologies with SWRL rules into ontologies with SPIN rules. We have tested the tool using ontologies and SWRL rule bases edited (and tested for reasoning) by Protégé and we have successfully imported the translated ontologies and SPIN rules into the TopBraid Composer, having exactly the same inference results. We have also evaluated the scalability of the tool and the effectiveness of some optimization of the generated SPIN rules. Our tool currently supports 41 SWRL builtins, including builtins for lists which are usually not supported, but we have provided a structured methodology for supporting more in the future.

Notice that our translation methodology is based on direct RDF-to-RDF translation between the SWRL and SPIN RDF vocabularies; therefore, it is not dependent on the implementation language we have choose for SWRL2SPIN. As for future work, we plan to make our tool available for public testing, to evaluate it for converting large SWRL rule bases, to support more SWRL builtins and to be able to automate the translation process, which currently has to be run within the Prolog environment, possibly as an add-on to some SPIN rule engine.

**References**


---

\(^3\) Available at http://users.auth.gr/nbassili/swrl2spin/


