SWRL2SPIN: Converting SWRL to SPIN

Nick Bassiliades

Department of Informatics, Aristotle University of Thessaloniki, Greece
nbassili@csd.auth.gr

Abstract. SWRL is a semantic web rule language that combines OWL ontologies with Horn Logic rules of the RuleML family of rule languages. Being supported by Protégé 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, being doubtful whether SWRL will become a W3C standard, it 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. 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, 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 [33] works on standards, technologies and tools 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.

There are mainly two modeling paradigms for the Semantic Web [14]. The first paradigm is based on the notion of the Description Logics [1] on which the Web Ontology Language (OWL) [13], the W3C recommendation for creating and sharing ontologies

on the Web, is based. The semantics of OWL ontologies can be handled by DL reasoning systems, such as Pellet [27], RacerPro [10], Fact++ [32] and HermiT [8] 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 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 [9], 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 [23] is an OWL 2 profile is 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 [23] and pD* [30]. 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 [15, 16] is a semantic web rule language that combines OWL ontologies with Horn Logic rules of the RuleML family of rule languages [26], 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 [15]. Decidability can be regained by restricting the form of admissible rules, by imposing a suitable safety condition [24]. Being supported by the Protégé ontology editor [25] as well as by popular rule engines and ontology reasoners, such as Jess [7], Drools [5] and Pellet [27], SWRL has become a very popular choice for developing rule-based applications on top of ontologies [3, 12, 22, 28]. 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 [18] 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 [11]. 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 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 the SWRL2SPIN tool, using SWI-Prolog [34] 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. To the best of our knowledge there is no other tool for transforming SWRL to SPIN.

In the rest of the paper, we overview SWRL and SPIN syntax and semantics, focusing on their RDF vocabularies, in sections 2 and 3, respectively. In section 4 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 5 we evaluate the tool and finally, in section 6, we conclude.

2 Semantic Web Rule Language

The Semantic Web Rule Language (SWRL) [16] 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 sublanguages 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 [24].

Rules are of the form of an implication between an antecedent (body) and consequent (head). The intended meaning can be read as: whenever the conditions 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 [21]) 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”.

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

Listing 1. Sample SWRL rule in human readable syntax

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 swr: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.

```
uni:s a swrl:Variable .
uni:c a swrl:Variable .
uni:f a swrl:Variable .
[ rdf:type swrl:Imp ;
   swrl:body [ a swrl:AtomList ;
      rdf:first [ a swrl:ClassAtom ;
                  swrl:classPredicate uni:Student ;
                  swrl:argument1 uni:z
                ] ;
            rdf:rest [ a swrl:AtomList ;
                       rdf:first [ a swrl:IndividualPropertyAtom ;
                                   swrl:propertyPredicate uni:attends ;
                                   swrl:argument1 uni:s ;
                                   swrl:argument2 uni:c
                                 ] ;
                       rdf:rest [ a swrl:AtomList ;
                                  rdf:first [ a swrl:IndividualPropertyAtom ;
                                              swrl:propertyPredicate uni:isTaughtBy ;
                                              swrl:argument1 uni:c ;
                                              swrl:argument2 uni:f
                                            ] ;
                                  rdf:rest rdf:nil
                                ]
                     ]
          ] ;
   swrl:head [ a swrl:AtomList ;
               rdf:first [ a swrl:IndividualPropertyAtom ;
                           swrl:propertyPredicate uni:knows ;
                           swrl:argument1 uni:s ;
                           swrl:argument2 uni:f
                         ] ;
               rdf:rest rdf:nil
             ]
 ] .
```

Listing 2. Sample rule in SWRL RDF concrete syntax

3 SPARQL Inferencing Notation

Modeling languages for the semantic web, such as RDF Schema [4] and OWL [13], 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) [18] 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 W3C standard [11] 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 [20].

The SPIN Modeling Vocabulary [19] 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.

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 superclasses 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 3a** 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 3b** shows how the same rule is represented using the SPIN modeling vocabulary.

```
uni:Student
  a rdfs:Class ;
  spin:rule
    [ a sp:Construct ;
      sp:text """
        CONSTRUCT {
          ?this uni:knows ?f .
        }
        WHERE {
        ?this uni:attends ?c .
        ?c uni:isTaughtBy ?f
        }"""
    ].
```

```
[ a  sp:Construct ;
  sp:templates ([
     sp:object spin:_this;
     sp:predicate uni:knows ;
     sp:subject sp:_f
  ]) ;
  sp:where (
     [ sp:object spin:_this ;
       sp:predicate uni:attends ;
       sp:subject sp:_c ]
     [ sp:object sp:_c;
       sp:predicate uni:knows ;
       sp:subject sp:_f ]
   )
]
```

Listing 3. Sample SPIN rule in (a) human-friendly notation and (b) 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 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 resresnted 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 4a**.

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 4b**.

```
[ rdf:type sp:Filter ;
   sp:expression [
      rdf:type sp:gt ;
      sp:arg1 [
        sp:varName "y" ;
      ] ;
      sp:arg2 "30"^^xsd:int ;
   ] ;
]
```

```
[ rdf:type sp:Bind ;
   sp:expression [
      rdf:type sp:mul ;
      sp:arg1 [ sp:varName "x" ;] ;
      sp:arg2 2 ;
   ] ;
   sp:variable [ sp:varName "y" ;] ;
]
```

Listing 4. Filter (a) and Bind (b) expressions in SPIN modeling vocabulary.

Table 1. Correspondence between SWRL and SPIN constructs

SWRL	**SPIN**
swrl:Imp	sp:Construct
swrl:head	sp:templates
swrl:body	sp:where
swrl:ClassAtom swrl:classPredicate <Class> swrl:argument1 <Arg>	sp:subject <Arg> sp:predicate rdf:type sp:object <Class>
swrl:IndividualPropertyAtom swrl:propertyPredicate <Prop> swrl:argument1 <Arg1> swrl:argument2 <Arg2>	sp:subject <Arg1> sp:predicate <Prop> sp:object <Arg2>
swrl:SameIndividualAtom swrl:argument1 <Arg1> swrl:argument2 <Arg2>	sp:subject <Arg1> sp:predicate owl:sameAs sp:object <Arg2>
swrl:DifferentIndividualsAtom swrl:argument1 <Arg1> swrl:argument2 <Arg2>	sp:subject <Arg1> sp:predicate owl:differentFrom sp:object <Arg2>
swrl:DatavaluedPropertyAtom swrl:propertyPredicate <Prop> swrl:argument1 <Arg1> swrl:argument2 <Arg2>	sp:subject <Arg1> sp:predicate <Prop> sp:object <Arg2>
swrl:BuiltinAtom swrl:builtin <Fun> swrl:arguments <Args>	*Customized translation*
swrl:Variable <Var>	sp:varName "<Var>"
<Value> ^^ <DataType>	<Value> ^^ <DataType>
<Individual>	<Individual>

4 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 4.3.

In the following, we give an example of translating a SWRL rule without built-ins to a SPIN rule. Consider the SWLR rule in **Listing 1** that is translated into the SPIN rule in **Listing 5**. The actual translation is between the RDF representations of the SWRL and SPIN rules, shown in **Listing 6** and **Listing 7**, respectively.

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

Listing 5. Translation of SRWL rule of **Listing 1** into SPIN

```
: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   ] ;
              rdf:rest rdf:nil
            ]]] ;
  swrl:head [ rdf:type swrl:AtomList ;
              rdf:first [ rdf:type swrl:IndividualPropertyAtom ;
                          swrl:propertyPredicate :knows ;
                          swrl:argument1 :x ;
                          swrl:argument2 :z   ] ;
              rdf:rest rdf:nil
]] .
```

Listing 6. Example of an input SWRL rule in RDF concrete syntax

```
spin:rule [
    rdf:type sp:Construct ;
    sp:templates (
        [ sp:object [ sp:varName "z" ;] ;
          sp:predicate :knows ;
          sp:subject [ sp:varName "x" ; ] ; ]   ) ;
    sp:where (
        [ sp:object :Student ;
          sp:predicate rdf:type ;
          sp:subject [ sp:varName "x" ; ] ; ]
        [ sp:object [ sp:varName "y" ; ] ;
          sp:predicate :attends ;
          sp:subject [ sp:varName "x" ; ] ; ]
        [ sp:object [ sp:varName "z" ; ] ;
          sp:predicate :isTaughtBy ;
          sp:subject [ sp:varName "y" ; ] ; ]  ) ;
] ;
```

Listing 7. Example of an output SPIN rule in SPIN modelling vocabulary

4.1 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 converted into two SPIN rules stored at classes Student (**Listing 8a**) and Course (**Listing 8b**), respectively:

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

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

Listing 8. SPIN rule embedded at class (a) Student, (b) Course

4.2 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 reorders 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 9**.

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

Listing 9. Optimized SPIN rule at class Course

4.3 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 [16] 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.

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[1]. 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 3, 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:lcase, 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.

[1] Due to space limitations, details can be found at https://github.com/nbassili/SWRL2SPIN

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. As an example, consider the SWRL rule in **Listing 10** which is translated in the SPIN rule at class Person (**Listing 11**). Specifically, the RDF concrete syntax for the SWRL builtin atom is shown in **Listing 12**, whereas the converted SPIN / SPARQL expression is shown in **Listing 13**.

```
Person(?x) ∧ firstName(?x, ?y) ∧ lastName(?x, ?z) ∧
             swrlb:stringConcat(?a, ?y, " ", ?z) →  fullName(?x, ?a)
```

Listing 10. Sample SWRL rule with builtin

```
CONSTRUCT {    # @Person
    ?this  :fullName ?a . }
WHERE {
    ?this  :firstName ?y .
    ?this  :lastName ?z .
    BIND (CONCAT(?y, " ", ?z) AS ?a) . }
```

Listing 11. Sample SWRL builtin translated to SPIN/SPARQL

```
[ rdf:type swrl:BuiltinAtom ;
  swrl:builtin swrlb:stringConcat ;
  swrl:arguments [ rdf:type rdf:List ;
            rdf:first  :a ;
            rdf:rest [ rdf:type rdf:List ;
            rdf:first  :y ;
            rdf:rest [ rdf:type rdf:List ;
            rdf:first " "^^xsd:string ;
            rdf:rest (  :z ) ] ] ]
] ;
```

Listing 12. RDF syntax for the SWRL builtin example

```
[  rdf:type sp:Bind ;
   sp:expression [  rdf:type sp:concat ;
           sp:arg1 [ sp:varName "y" ; ] ;
           sp:arg2 " " ;
           sp:arg3 [ sp:varName "z" ; ];];
   sp:variable [ sp:varName "a" ; ] ;
]
```

Listing 13. RDF syntax for the converted example of Listing 12

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 [6] for Protégé 3.x.

5 Evaluation

To evaluate SWRL2SPIN we have initially generated use cases of a University ontology with various SWRL rules in Protégé[2] [25], including all supported SWRL-builtins. Then we have used the SWRLDroolsTab [29] 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 [31] for equivalent inferences. The results were found identical for all use cases, except the ones that could not be run in SWRLDrools.

Finally, we have evaluated the optimized SPIN rules (section 4.2) of SWRL2SPIN against their unoptimized version. For this we have used the unoptimized rule at Listing 8b against the optimized rule at Listing 9 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 unoptimized 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.

6 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

[2] We have used the SWRLTab editor of both Protégé 3.5 and 5.2.

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 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 evaluate it for converting larger SWRL rule bases, to support more SWRL builtins and to possibly provide this tool as an add-on to some SPIN rule engine. Finally, a transition of the tool to SHACL SPARQL rules [17] is underway[4].

7 References

1. Baader F. 2003. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge Univ Press.
2. Baader F., Sattler U. 2001. An overview of tableau algorithms for description logics. Studia Logica, 69(1), 5-40.
3. Billet Y.-G., Gravier C., Fayolle J.: SWRL-Based Context Awareness for Application Servers Hosting Digital Services. RuleML America 2011: 222-229
4. Brickley D., Guha R.V. (Eds.), RDF Schema 1.1, W3C Rec, 25 Feb 2014, http://www.w3.org/TR/rdf-schema/
5. Drools, http://www.drools.org/
6. Elenius D.: SWRL-IQ: A Prolog-based Query Tool for OWL and SWRL. OWLED 2012
7. Friedman-Hill E. 2003. Jess in Action: Rule Based Systems in Java. Manning Publications. ISBN 1-930110-89-8
8. Glimm B., Horrocks I., Motik B., Stoilos G., Wang Z., HermiT: An OWL 2 Reasoner, J Automated Reasoning (2014) 53: 245.
9. Grosof B. N., Horrocks I., Volz R., Decker S. 2003. Description Logic Programs: Combining Logic Programs with Description Logic. In Proceedings of the International Conference on World Wide Web (pp. 48-57). ACM Press.
10. Haarslev V., Hidde K., Möller R., Wessel M. The RacerPro knowledge representation and reasoning system. Semantic Web Journal, 3(3):267–277, 2012.

[3] Available at https://github.com/nbassili/SWRL2SPIN
[4] https://github.com/nbassili/SWRL2SHACL

11. Harris S., Seaborne A., SPARQL 1.1 Query Language, W3C Rec, 21 Mar 2013. http://www.w3.org/TR/sparql11-query/
12. Herrero-Zazo M., Segura-Bedmar I., Hastings J., Martínez P.: DINTO: Using OWL Ontologies and SWRL Rules to Infer Drug-Drug Interactions and their Mechanisms. J. of Chemical Information and Modeling 55(8): 1698-1707 (2015)
13. Hitzler P., Krötzsch M., Parsia B., Patel-Schneider P. F., Rudolph S., OWL 2 Web Ontology Language Primer (2nd Edition), W3C Rec, 11 Dec 2012. http://www.w3.org/TR/owl-primer
14. Horrocks I., Parsia B., Patel-Schneider P., Hendler J. 2005. Semantic Web Architecture: Stack or Two Towers?. Principles and Practice of Semantic Web Reasoning. PPSWR 2005. LNCS 3703. Springer.
15. Horrocks I., Patel-Schneider P. F., Bechhofer S., Tsarkov D., OWL rules: A proposal and prototype implementation, Journal of Web Semantics, 3(1), 2005, pp. 23-40.
16. Horrocks I., Patel-Schneider P. F., Boley H., Tabet S., Grosof B., Dean M. 2004. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission. 21 May 2004. http://www.w3.org/Submission/SWRL/
17. Knublauch H., Allemang D., Steyskal S., SHACL Advanced Features, W3C Working Group Note 08 June 2017, https://www.w3.org/TR/shacl-af/
18. Knublauch H., Hendler J. A., Idehen K., SPIN - Overview and Motivation, W3C Member Submission, 22 Feb 2011. http://www.w3.org/Submission/spin-overview/
19. Knublauch H., SPIN - Modeling Vocabulary, W3C Member Submission, 22 Feb 2011. http://www.w3.org/Submission/spin-modeling/
20. Knublauch H., SPIN - SPARQL Syntax, W3C Member Submission, 22 Feb 2011. http://www.w3.org/Submission/spin-sparql/
21. Lloyd J. W. Foundations of logic programming (2nd edition). Springer series in symbolic computation. Springer, 1987.
22. Matheus C. J., Baclawski K., Kokar M. M., Letkowski J. Using SWRL and OWL to Capture Domain Knowledge for a Situation Awareness Application Applied to a Supply Logistics Scenario. RuleML 2005: 130-144
23. Motik B., Cuenca Grau B., Horrocks I., Wu Z., Fokoue A., Lutz C., OWL 2 Web Ontology Language Profiles (2nd Edition), W3C Recommendation 11 Dec 2012. https://www.w3.org/TR/owl2-profiles/
24. Motik B., Sattler U., Studer R. 2005. Query Answering for OWL-DL with Rules. J. of Web Semantics. 3(1): 41–60.
25. Protégé ontology editor. http://protege.stanford.edu/.
26. RuleML, http://wiki.ruleml.org/index.php/RuleML_Home
27. Sirin E., Parsia B., Grau B. C., Kalyanpur A., Katz Y. 2007. Pellet: A Practical OWL-DL Reasoner. Journal of Web Semantics, 5(2), 51-53.
28. Somodevilla M. J., Mena I., Pineda Torres I. H., de Célis Herrero C. P.: Deducting Lifestyle Patterns by Ontologies' SWRL Rules. DEXA Workshops 2015: 9-13
29. SWRL Drools Tab, 2012. http://protege.cim3.net/cgi-bin/wiki.pl?SWRLDroolsTab.
30. ter Horst H. J. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. J. of Web Semantics 3(2–3):79–115, 2005
31. TopQuadrant, TopBraid Composer, https://www.topquadrant.com/tools/IDE-topbraid-composer-maestro-edition/
32. Tsarkov D., Horrocks I. 2006. Fact++ description logic reasoner: System description. In Proceedings of Automated Reasoning (pp. 292-297). Springer.
33. W3C (2013). The Semantic Web Activity. http://www.w3.org/2001/sw/.
34. Wielemaker J., Schrijvers T., Triska M., Lager T.: SWI-Prolog. TPLP 12(1-2): 67-96 (2012)