A Teaching Strategies Engine Using Translation from SWRL to Jess

Eric Wang and Yong Se Kim

Creative Design and Intelligent Tutoring Systems Research Center
Sungkyunkwan University, Suwon, Korea
{ewang, yskim}@skku.edu

Abstract. Within an intelligent tutoring system framework, the teaching strategy engine stores and executes teaching strategies. A teaching strategy is a kind of procedural knowledge, generically an if-then rule that queries the learner's state and performs teaching actions. We develop a concrete implementation of a teaching strategy engine based on an automatic conversion from SWRL to Jess. This conversion consists of four steps: (1) SWRL rules are written using Protégé's SWRLTab editor; (2) the SWRL rule portions of Protégé's OWL file format are converted to SWRLRDF format via an XSLT stylesheet; (3) SweetRules converts SWRLRDF to CLIPS/Jess format; (4) syntax-based transformations are applied using Jess meta-programming to provide certain extensions to SWRL syntax. The resulting rules are then added to the Jess run-time environment. We demonstrate this system by implementting a scenario with a set of learning contents and rules, and showing the run-time interaction with a learner.

1 Introduction

We are developing an intelligent tutoring system framework, in which learning contents and learner data are stored in ontologies. Within this framework, the teaching strategy engine stores and executes teaching strategies to determine the teaching actions. Teaching strategies are a kind of procedural knowledge, including assessment algorithms and decision procedures. Traditionally, such knowledge has been difficult to represent within an ontology in a format that supports automatic execution. However, recent advances in integrating rules and ontologies, and the development of tools to support them, present an opportunity to develop a teaching strategy engine using ontology-based rule representations. We have developed a conversion from SWRL rules written in Protégé, to a run-time engine in Jess, using several standard open-source tools. The benefit of representing teaching strategies as SWRL rules is that the strategies' computations would be explicitly represented in the ontology, and could be viewed and edited, as well as reasoned about by other applications.

2 Tools

Our conversion method uses the following standard tools.

M. Ikeda, K. Ashley, and T.-W. Chan (Eds.): ITS 2006, LNCS 4053, pp. 51 – 60, 2006.

2.1 SweetRules

SweetRules 236 is a toolkit that provides a suite of converters between several standard rule formats, including RuleML, Courteous Logic Programs, and SWRL, and several execution engines, including Jess, Jena, and XSB Prolog.

2.2 Semantic Web Rule Language

Semantic Web Rule Language (SWRL) 4 combines a rich OWL ontology + description logic (DL) with a subset of first-order logic (FOL) syntax. SWRL rules are always defined on top of an OWL ontology. Syntactically, a SWRL rule is function-free (no user-defined functions), Datalog (no functions within terms), Horn (no negation or disjunction), and has no explicit quantifiers, but with implicit universal quantification for all variables. SWRL atoms include class and property atoms whose names refer to classes and properties in the ontology, and a library of built-in function atoms that implement fundamental math, string, and date operations.

Class and property atoms in a SWRL rule body represent queries into the knowledge base. A class atom with a ground argument (individual name or bound variable) performs a class membership test. If the argument is an unbound variable, i.e. this atom is the first occurrence of the variable in the rule, then a class atom conceptually iterates over every individual of that class. (This iteration actually arises from the implicit universal quantification over all variables, which ensures that every rule will be applied to every individual in the knowledge base.) Property atoms are analogous, but perform property membership tests.

In a rule head, class and property atoms represent new conclusions. According to the SWRL semantics, an ontology with rules is consistent iff the head facts are true whenever the bodies are true. The standard implementation of this semantics is to *make* the head facts true, i.e. to add them to the knowledge base. SWRL has no intrinsic facilities for performing side-effects, including modifying existing knowledge.

The SWRL standard 4 defines two concrete XML-based formats for SWRL rule representation: an XML concrete syntax, and an RDF concrete syntax. SweetRules denotes these formats as SWRLXML and SWRLRDF, respectively, and provides some translation tools to and from these formats.

2.3 Protégé OWL + SWRLTab

Protégé 5 is a standard, open-source ontology editor, with support for Web Ontology Language (OWL) via an OWL plugin. Recent builds of Protégé OWL include a SWRLTab view, which provides convenient editing of SWRL rules. The SWRLTab editor uses SWRL's "human readable" syntax 4, which is comparable to Prolog in conciseness and readability. Another key benefit is that the SWRL rule editor is closely integrated with the OWL ontology.

From Protégé's perspective, the SWRL language syntax is represented as an ontology, and a SWRL rule is simply an OWL individual that is instantiated using classes from this ontology. The SWRLTab editor automates the laborious details of instantiating the tree-like structure for each SWRL rule. Protégé saves the OWL

ontology and SWRL rules into the same .owl file. (This causes some complications for the conversion to Jess, as will be discussed below.)

2.4 Jess Rule Engine

Jess (Java Expert System Shell) 1 is a rule engine written in Java. It provides a Lisp-like syntax and interpreter, with forward-chaining rules using the rete network algorithm. It is based on the earlier CLIPS rule language, and is still largely forward-compatible with CLIPS, in that most CLIPS programs are also valid Jess programs. It is free for research use.

Jess is able to use standard Java reflection to import Java libraries, call any Java code, and directly manipulate Java objects. This gives it great flexibility as a general-purpose execution environment. We exploit this capability to run XSLT and SweetRules from within Jess.

3 Conversion from OWL Ontology to Jess Facts

An OWL ontology consists of classes and properties, OWL restrictions to specify their semantics, and individuals defined using these elements. SweetRules provides a translation path (a sequence of translator tools) from OWL to Jess, which can be used to convert a knowledge base of facts. In particular, we can use this to convert the learning contents ontology to Jess facts.

An OWL ontology file is converted to Jess by executing the SweetRules command:

translate owl jess owl-input-path jess-output-path

This produces a Jess file in which OWL individuals are converted to Jess facts (simple assertions), and OWL properties and restrictions are converted to Jess rules.

Note that we must exclude SWRL rules from consideration here, since this translation path has expressiveness limitations that conflict with Protégé's handling of SWRL rules.

- SweetRules's OWL-to-Jess translation path is restricted to a subset of Protégé OWL's expressiveness. The OWL *functional* property attribute, which denotes that a property may have at most one value, must be avoided, as it is a kind of cardinality restriction, which this translation path can't support.
- However, to edit SWRL rules in Protégé, the user must "activate" SWRL, which imports the SWRL ontology definition. The SWRL ontology itself uses functional properties internally.

Hence, a Protégé .owl file that has "activated" the SWRL editing capabililty can no longer be translated via SweetRules' OWL-to-Jess path. It follows that to use the OWL-to-Jess translation requires two separate Protégé .owl files: (1) an "ontology-only" file that defines all individuals, but does not activate SWRL, which is converted as described here; and (2) a "SWRL-only" file that defines the SWRL rules, which is converted separately, as described in the next section. These two files must share the same class and property definitions, and declare the same xml:base URI, to ensure that individuals generated from one file will match rules generated from the other.

4 Conversion from SWRL Rules to Jess Rules

In this section, we describe our conversion from Protégé's SWRL rules to Jess code.

4.1 Conversion from Protégé OWL to SWRLRDF

Within Protégé's .owl file format, SWRL rules are saved in a syntax that is similar to SWRL's RDF concrete syntax (SWRLRDF format), but with some specific differences. We have developed an XSLT stylesheet to convert the SWRL rule subset of a Protégé .owl file to SWRLRDF. This conversion includes the following steps.

- **Convert lists to collections.** In Protégé OWL, swrl:body and swrl:head atoms are defined as lists, using a Lisp-like recursive list structure that provides explicit sequential ordering. Each swrl:body or swrl:head atom has a single swrl:AtomList child node, which itself has exactly two child nodes: an rdf:first node whose child is the first element of the (sub)list, and an rdf:rest node whose child is the remainder of the list, recursively represented as another swrl:AtomList, or by the special end-of-list symbol #nil. Since the rdf:first and rdf:rest nodes are explicitly named, their order within the file is arbitrary.

  SWRLRDF defines ruleml:body and ruleml:head atoms as collections of multiple nodes, with implicit sequential ordering between them. Specificially, these atoms have an rdf:parseType="Collection" attribute, and can have any number of child nodes.

  We convert the swrl:AtomList format to the rdf:parseType="Collection" format, preserving the sequence of the elements.
- **Lift variable declarations to top of file.** Every variable used in any SWRL rule must be declared once as a swrl:Variable atom. In SWRLRDF, it is preferred to list all such declarations *a priori*, in a block at the top of the file, before any SWRL rule definition. This convention simplifies subsequent conversion processes by automated tools.

```
<?xml version="1.0"?>
<rdf:RDF ...>     Root node
  <swrl:Variable rdf:ID="x" />  Variable declarations
  ...
  <swrl:Imp rdf:ID="rule-1">          Rule definitions
    ...
</rdf:RDF>
```

  In contrast, OWL supports an enhanced syntax with "just-in-time" declarations, in which the first occurrence of an identifier in the .owl file is written as a child node below the node where it is first used, with an rdf:id=*name* attribute, which acts as a declaration. All subsequent occurrences of the same identifier in this file use an rdf:resource=#*name* attribute on the node that uses it, which acts as a reference to the previous declaration. This means that swrl:Variable declarations may appear anywhere in a Protégé OWL file, nested at any depth within a SWRL rule's

definition, and that every usage of a swrl:Variable has two alternative syntactic forms, which complicates any processing. The following excerpt shows an OWL "just-in-time" declaration for `swrl:argument2`'s "U" variable, and a reference for `swrl:argument1`'s "H" variable.

```
<swrl:IndividualPropertyAtom>
  <swrl:argument2><swrl:Variable rdf:ID="U"/>
  </swrl:argument2>
  <swrl:propertyPredicate
rdf:resource="#hasLearner"/>
  <swrl:argument1 rdf:resource="#H"/>
</swrl:IndividualPropertyAtom>
```

We convert swrl:Variable atoms in two steps. (1) When converting the `rdf:RDF` root node (which is always the first node in the file, by definition), we "look ahead" and extract all swrl:Variable atoms, at any depth in the file, and copy them as immediate children of the root node. (2) All "just-in-time" declarations are rewritten by copying the child node's rdf:ID attribute as an rdf:resource reference.

- **Filter out OWL ontology atoms.** All class, property, and individual atoms that pertain to the OWL ontology are not used by SWRLRDF. These are filtered out by simply not copying them.

  In a sense, the conversion from OWL to SWRLRDF "loses" all information about the OWL ontology. The OWL ontology and individuals are assumed to be converted separately, as described in Section 0. It is the user's burden to ensure that the results of these two separate translations remain consistent with each other.

4.2 Conversion from SWRLRDF to CLIPS

The SweetJess component of SweetRules includes a translation path from SWRLRDF to CLIPS. This conversion is achieved by executing the SweetRules command:

translate swrlrdf clips swrlrdf-input-path clips-output-path

As Jess uses CLIPS syntax, the result is also valid as a Jess file. We distinguish the "ontology-only" Jess file produced in Section 0, from the "rules-only" Jess file produced here, by assigning them distinct extensions ".jess" and ".clp", respectively.

By default, a SWRL rule represents new conclusions only, i.e. new facts that are "made true" by adding them to the knowledge base. This is typically implemented by converting every SWRL rule head to a Jess `assert` statement. Consider a simple SWRL rule, written in the Protégé SWRLTab:

Student(?S) → **Person(?S)** "A student is a person"

SweetRules conversion from SWRLRDF to CLIPS produces the following Jess rule. (Here and hereafter, Jess rules and facts are shown in a terse format, with namespace prefixes omitted, for clarity.)

(defrule rule-1 (triple type ?S **Student**) => (assert (triple type ?S **Person**)))

This is a valid Jess rule, which is triggered by existing facts about `Students`, and responds by `asserting` new facts about `Persons`.

4.3 Extending SWRL Via Rule Transformations in Jess

SWRL provides a restricted expressiveness to ensure decidability. Jess rules are often more concisely expressed using constructs not available in SWRL. We have devised a generic mechanism to extend SWRL rules with additional constructs based on syntactic rule transformations in Jess. Syntactically, a SWRL rule must be a flat list of atoms, but it does guarantee that sequence is preserved. We find that judicious insertion of new, reserved class and property atoms is sufficient to add new keywords and even block structure. We define the following extension keywords:

SWRLx Atom	Where	Why	Effect
__name(*str*)	body	SweetRules workaround	Sets the Jess rule name to *str*. (SweetRules ignores Protégé's encoding of a SWRL rule's name, and generates default names.)
__naf(?)	body	Expressive	Converts to a Jess (**not** …) block.
__all(?A) __end(?A)	body	Expressive	Expands to a "guarded not" block 7, which handles the implicit iteration for an "if all" test.
__bind(?R)	body	Jess	Expands to a Jess pattern binding.
__modify(?R)	head	Jess	Changes the immediately following clause from an (assert …) to (modify ?R …).
__call(?)	head	Effecting	Changes the immediately following clause from an (assert (P …)) to a Jess function call (P …).

Our extended keywords are edited normally using the SWRL rule editor, and are converted verbatim by SweetJess. We treat the resulting CLIPS file as an intermediate quasi-rule format, convert each quasi-rule to a nested-Vector format that supports text processing, and apply the above rule transformations as a set of functions written in Jess, thereby achieving our extended semantics. The transformed rules are then evaluated in Jess, which updates Jess's working memory.

5 Implementation Issues for Automatic Conversion

We now discuss some pragmatic issues in implementing the conversion capability.

5.1 SweetRules Installation

SweetRules's considerable power is offset by its intimidating installation requirements. It depends on many third-party open-source software components, all of which must be installed concurrently. We note the following installation pitfalls, which could deter a typical user.

- **Version dependencies.** Some parts of SweetRules suffer from hard-coded dependencies to specific versions of other components. In our experience, some of

these components can be upgraded to the latest versions without harm[1]. Other components *must* use the stated versions, else SweetRules quickly fails and throws many Java exceptions. More recent versions of these components have typically renamed some internal Java .jar files, which breaks SweetRules' hard-coded dependencies.

The following table summarizes SweetRules' software components, their stated version requirements, and the allowed and forbidden upgrades, based on our empirical testing. All version requirements are as stated by SweetRules 2.1's own installation program, which is the latest version of SweetRules.

Table 1. SweetRules software components, and allowed version upgrades

Source/ Vendor	Component	Stated Version	Can Upgrade To
---	SweetRules	2.1	---
Sun	Java SDK	1.4.2	5.0 Update 5, 6
Sandia	Jess	6.1p7	6.1p8
IBM	CommonRules	3.3	---
Sun	Java Web Services Development Pack (JWSDP)	1.5	**Must use 1.5**
SourceForge	dom4j	1.5	1.6.1
Apache	log4j	1.2.8	1.2.12
SourceForge	Junit	3.8.1	---
Apache	Xalan	2.6.0	**Must use 2.6.0**
Declarativa	InterProlog	2.1.1	**Must use 2.1.1**
SourceForge	XSB Prolog	2.6	2.6-fixed
SourceForge	KAON DLP	0.5	---
SourceForge	Saxon	8.1.1	**Must use 8.1.1**
HP	Jena	2.2	**Must use 2.2**

- **Installer error.** The final step of the SweetRules 2.1 installer should produce a Windows command script, "runsr.cmd", which is the most convenient way to launch SweetRules, as it properly fills in the extremely long[2] Java classpath. But it fails due to an unfortunate bug. The installer's final input screen prompts the user to enter the SweetRules installation directory path, which it uses to compose a Java command line that will produce the script. However, it assumes that the user included double-quotes around the SweetRules path, and blindly strips the first and last characters (without checking to see whether they actually are double-quotes!). This results in an invalid directory path, which causes the Java command to fail.

  The workaround is straightforward: Edit the installation log file, copy the final Java command line, manually fix all occurrences of the SweetRules directory path, and manually execute it from a Windows command prompt.

[1] More precisely, we can say that these upgrades have *not yet* caused any observed problems for the OWL-to-Jess and SWRLRDF-to-CLIPS translation paths.

[2] For the author's runsr.cmd file, the classpath argument is 4,006 characters long.

5.2 Managing SweetRules as a Child Process

SweetRules presents a simple command-line interface. We run SweetRules as a child process under Jess, using the standard Java ProcessBuilder class, and spawn worker threads to read SweetRules' output buffers asynchronously as a work-around for its tendency to block during translations. This reduces the entire SweetRules conversion step to a single function call.

6 Example Scenario

We demonstrate the teaching strategy engine using the following simple scenario.

- The learning contents includes a Problem1, which has a Solution1a.
- Among all learners, there is a learner named Alan.
- Alan has scored 58.0 on Problem1, which is recorded in a HistoryDatum h1.

We define the base ontology separately, then use OWL import statements to ensure that the "individuals-only" and "rules-only" ontology files share it. The base ontology is shown in Figure 1.

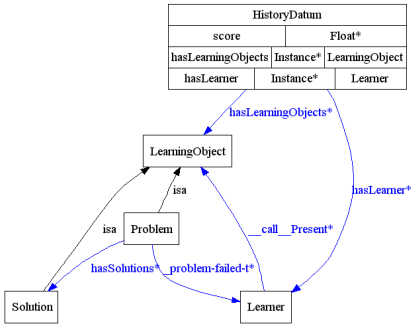

Fig. 1. Ontology classes and properties for scenario 1

The "individuals-only" ontology file adds individual definitions corresponding to the scenario data described above. This file is converted via the OWL-to-Jess translation path to a set of Jess statements, including the following fact:

```
(assert (triple score h1 58.0))                    (a)
```

This scenario uses one teaching macro-strategy: "If a learner fails a problem, show a solution immediately" 8, where a problem is considered to be failed if its score is less than 75%. This strategy is decomposed into three SWRL rules, organized in a bottom-up manner:

r-score-low	"A score less than 75% is a *low score*."
r-problem-failed	"A learner fails a problem if the learner has a *low score*."
s-strategy1	"If a learner fails a problem, show a solution immediately."

Editing of these rules in Protégé SWRL is shown in Figure 2.

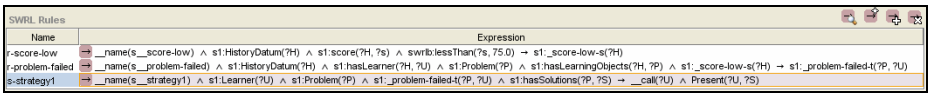

Fig. 2. Protégé SWRL rules for scenario 1

Having edited the rules, the teacher converts the rules to Jess by opening the "rules-only" .owl file. This automatically invokes XSLT, SweetRules, and rule transformations in Jess to achieve the conversion, detects which rules have changed, and updates only those rules in Jess's working memory, as shown in Figure 3.

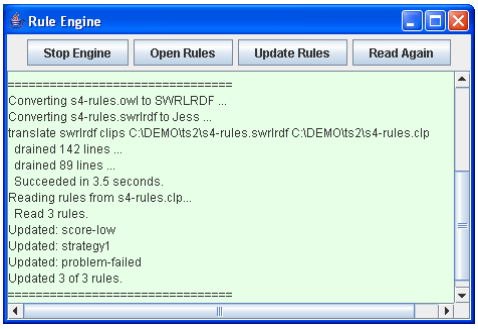

Fig. 3. Rule engine interface for automatic conversion from Protégé SWRL to Jess

With the scenario facts and rules loaded, the running Jess rule engine produces the following results:

- Fact (a), with h1's score of 58.0, satisfies the bottommost rule **r-score-low**. This rule fires, and asserts a new fact:

  (triple type h1 _score-low-s) (b)

- Fact (b) satisfies the intermediate rule **r-problem-failed**, which fires and asserts a new fact

  (triple _problem-failed-t Problem2 Alan) (c)

- Fact (c) satisfies the topmost rule **s-strategy1**, which fires and executes the Jess function call

  `(Present Alan Solution2b)`

We may assume that the `Present` function displays the solution in some manner that is integrated with the system's visual interface.

7 Summary

We have developed a teaching strategy engine in Jess, based on an automatic conversion mechanism from OWL individuals and SWRL rules in Protégé to Jess facts and rules. Our conversion mechanism leverages existing tools for rule language conversions, and extends their applicability to include Protégé's OWL format. More broadly, we have established a framework for representing teaching strategy knowledge as rules in a standard ontology editor. This supports a pedagogical development environment where a teacher can incrementally edit rules in the ontology, and quickly reload them into the teaching strategy engine for testing. We have also demonstrated a syntax-based extension to SWRL that supports more flexible and expressive rules, which supports the development of practical rules that can encode interactions with a human learner.

Acknowledgements

This research was supported by the Korean Ministry of Science & Technology through the Creative Research Initiative Program.

References

1. Friedmann-Hill, E. (2003). Jess in Action. Manning, Greenwich.
2. Grosof, B. N. (1997a). Building Commercial Agents: An IBM Research Perspective. IBM Research Report RC20835.
3. Grosof, B. N., Gandhe, M. D., and Finin, T. W. (2003). SweetJess: Inferencing in Situated Courteous RuleML via Translation to and from Jess Rules, *Unpublished working paper*, http://ebusiness.mit.edu/bgrosof/#sweetjess-basic.
4. Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., and Dean, M. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, http://www.w3.org/Submission/SWRL/.
5. Protégé, http://protege.stanford.edu/.
6. SweetRules, http://sweetrules.projects.semwebcentral.org/.
7. Wang, E., Kim, S. A., and Kim, Y. S. (2004). A Rule Editing Tool with Support for Non-Programmers in an Ontology-Based Intelligent Tutoring System, Workshop on Semantic Web for E-Learning (SW-EL), 3rd Int'l. Semantic Web Conf. (ISWC), Hiroshima, Japan.
8. Wang, E., and Kim, Y. S. (2006). Teaching Strategies Using SWRL. Submitted for publication in *Journal of Intelligent Information Systems*. Revised version of: Wang, E., Kashani, L., and Kim, Y. S. (2005), Teaching Strategies Ontology Using SWRL Rules, Int'l. Conf. on Computers in Education (ICCE), Singapore.