CS4997

Honours Thesis

TRANSLATOR:
A TRANSlator from LAnguage TO Rules

David Z. Hirtle

April 12, 2006

Abstract

One explanation as to why the Semantic Web has not quite caught on yet is that the barrier to entry is too high. This paper describes TRANSLATOR, a free tool available as a Java Web Start application designed to allow anyone, even non-experts, to write facts and rules in formal representation for use on the Semantic Web. This is accomplished by automatically translating natural language sentences written in Attempto Controlled English into the Rule Markup Language, using the Attempto Parsing Engine Web service as a backend. Representation in RuleML has several advantages, not least of which is compatibility with existing Semantic Web standards.

Contents

Chapter 1

Introduction

The Semantic Web, often described as the next generation of the existing Web, is a highly collaborative effort with the ultimate goal of making it easier for humans and computers to work together [3]. The key is enriching the current Web, which is understandable by humans alone, with machine-interpretable (meta)data in the form of facts, rules and ontologies.

Enabling standards such as the Web Ontology Language (OWL) and Resource Description Framework (RDF) have already been defined by the World Wide Web Consortium (W3C), and work toward a Rule Interchange Format (RIF) is now in progress. Despite all of this activity, the Semantic Web has yet to see widespread adoption. There is some concern (e.g., [12]) that the barrier to entry is too high: understanding such formal languages is a challenge for the vast majority of individuals. The RIF Working Group charter[1], for example, openly acknowledges that users are unlikely to use it first-hand, but rather "are expected to work with tools or rule languages which are transformed to and from this format."

This stands in stark contrast to the original Web, where ordinary users could experiment with HTML (even simpler at the time) and get results relatively quickly and easily. The result is a cycle wherein each person who recognizes the benefits and decides to participate actually makes the Web more useful and therefore attractive to others. Similarly, tools (e.g., HTML editors) are created as participation increases, further lowering the barrier to entry.

One way of achieving this desirable cyclical effect in the Semantic Web is to lower the barrier to entry by offering a more user-friendly format. While the intuitiveness and expressiveness of natural language makes it a logical candidate, it is inherently ambiguous and has therefore largely been ignored for formal applications.

[1] http://www.w3.org/2005/rules/wg/charter

Fortunately, recent work has shown that principles from computational linguistics can be applied to translate between language and machine-interpretable logic. This is accomplished by using a subset of language that remains completely natural. One such *controlled* language is Attempto Controlled English (ACE) [7], whose development over the past ten years has also included its own Attempto Parsing Engine (APE) [9].

The focus of this work is using ACE as the means to author formal rules for the Semantic Web via a newly-developed application called TRANSLATOR (TRANSlator from LAnguage TO Rules). Specifically, rules written in ACE are taken as input, parsed (using the APE Web service) into a variant of first-order logic known as a Discourse Representation Structure (DRS) and then mapped into formal representation in the XML-based Rule Markup Language (RuleML) [4]. RuleML is the product of a long-standing effort toward standardizing rule markup on the Web and is compatible with other Semantic Web languages.

TRANSLATOR, available as a Java Web Start application[2], allows non-programmers to write their own rules on the Semantic Web via familiar natural language. As a trivial example, users may write:

> *If the customer is a student then he/she receives a discount.*

instead of something formal like the following (the corresponding DRS in logical notation):

$$\begin{aligned}
&\forall ABCDE : structure(D, atomic) \wedge \\
&quantity(D, cardinality, count_unit, E, eq, 1) \wedge \\
&object(D, customer, person) \wedge object(A, student, person) \wedge \\
&quantity(A, cardinality, count_unit, B, eq, 1) \wedge \\
&structure(A, atomic) \wedge predicate(C, state, be, D, A) \\
&\rightarrow \exists FGH : object(F, discount, object) \wedge \\
&quantity(F, cardinality, count_unit, G, eq, 1) \wedge \\
&structure(F, atomic) \wedge predicate(H, unspecified, receive, A, F)
\end{aligned}$$

The idea of controlled language dates back to the late 1920s when Basic English was developed as a universally accessible language. Decades later, industry (e.g., the European Association of Aerospace Manufacturers, Boeing and General Motors) began to realize the benefits of controlled languages: documentation that is more readable, consistent and (machine) translatable. Only relatively recently have controlled languages begun to focus primarily on computer processability.

The existing work most related to TRANSLATOR includes the following:

[2] http://www.ruleml.org/translator

Pseudo Natural Language (PNL)

PNL is an interface for Metalog, probably the first proposal for lowering the barrier to entry of the Semantic Web [15]. Metalog and PNL focus exclusively on RDF. Like ACE, PNL is unambiguous, but its grammar appears to be much less expressive (and therefore less natural).

Semantics of Business Vocabulary and Business Rules (SBVR)

SBVR includes a "Meaning and Representation Vocabulary" that is similar to RDF and can be mapped to OWL in addition to a Structured English notation [21]. A limitation, however, is its lack of support for anaphoric references.

Common Logic Controlled English (CLCE)

CLCE [20] is syntactically similar to but slightly less natural than ACE. While it includes an ontology for sets, sequences and integers, CLCE does not handle plurals.

Controlled English to Logic Translation (CELT)

CELT [16] was originally inspired by ACE, but its lexicon is imported from WordNet (including default word senses) and mapped to the Suggested Upper Merged Ontology (SUMO). The intent of CELT is to simplify ontology-based knowledge representation. It also does not support plurals.

Processable ENGlish (PENG)

Like CLCE and CELT, PENG is also quite similar to ACE though lacking plurals. Unlike them, however, work has been done on relating PENG to the existing Semantic Web standards of OWL and RDF (e.g., [19]). Also of interest is ECOLE [18], a look-ahead text editor that guides the author of PENG texts on-the-fly with syntactic hints, meaning he or she need not learn the rules explicitly.

Chapter 2

Input: Attempto Controlled English

The input accepted by TRANSLATOR has the appearance of plain English but is in fact a formal language in the same vein as the relatively esoteric OWL and RDF. Known as Attempto Controlled English (ACE), it is a tractable subset of full English that can be unambiguously translated into first-order logic. In this way, ACE combines the ease of use and familiarity of natural language with the ease of processing and precision of formal language.

2.1 Overview

The language of ACE remains as natural as possible while avoiding ambiguity. To this end, the grammar of ACE excludes certain imprecise phrasings, e.g.,

Students hate annoying professors.

(Do students hate their professors, who are annoying, or do they hate to annoy their professors?) Remaining ambiguity is handled by predefined rules favouring one interpretation over others. Consider the following sentence:

The student brings a friend who is an alumnus and receives a discount.

This is ambiguous in full natural language: does the student or the friend receive the discount? In ACE, on the other hand, the sentence has the interpretation that it is the student who receives the discount, i.e. the coordination is outside the relative phrase.

The alternative interpretation (the friend receives the discount) can be expressed in ACE by simply repeating the relative pronoun:

> *The student brings a friend who is an alumnus and* ***who*** *receives a discount.*

The vocabulary of ACE consists of predefined and user-defined elements. Function words such as articles (e.g., *a*, *the*), prepositions (e.g., *in*, *to*) and pronouns (e.g., *they*, *itself*, *everyone*) as well as fixed phrases (e.g., *it is not the case that...*) are all predefined, having syntactic roles to play. The default lexicon of nearly 100 000 content words (nouns, verbs, adjectives and adverbs) can be supplemented by user-defined (e.g., domain-specific) lexicons. Also, users can (temporarily) provide missing words using word class prefixes as follows:

> *p:Alexander is a a:self-proclaimed n:autodidact. He a:indubitably v:defenestrates himself during every lecture.*

This brief overview of ACE has not covered several significant features such as anaphoric references and plurals. Complete details on ACE, including further publications, are available from the Attempto project website[1].

2.2 Expressible Rules

While all ACE sentences are accepted English, the opposite is not true. Since TRANSLATOR focuses on translating language (i.e., ACE) to rules, a pertinent question becomes, "What kinds of rules are expressible in ACE?"

First, the notion of a rule is introduced. Knowledge is divisible into two categories: facts (e.g., *John is human*) and rules, which allow inferring new facts from existing ones (e.g., *All humans are mortal*, giving the new fact *John is mortal*). The typical logic programming (e.g., Prolog) representation of a rule is as a left-hand side (the head, i.e., conclusion) separated from a right-hand side (the body, i.e., one or more conditions) by a special "is implied" symbol, e.g.,

```
mortal(X) :- human(X).
```

This is roughly equivalent to the above rule; it can be translated to English as *X is mortal if X is human*, or, in the opposite direction, *if X is human then X is mortal.* The representation of a fact just has a left-hand side, e.g.,

[1] http://www.ifi.unizh.ch/attempto

```
human(John).
```

In other words, *John is human* (if nothing). The absence of any conditions in the body means that the conclusion is unconditionally true. Therefore facts are just a special kind of rule and rules can be considered as conditional facts. It should be no surprise, then, that ACE and RuleML (and thus TRANSLATOR) support both facts and rules; however, as implied by the name "TRANSlator from LAnguage TO Rules", (conditional) rules are the more interesting case and are therefore discussed in greater detail. Conversely, a query is essentially a rule with no conclusion (cf. a fact, which is somehow the opposite) and may likewise be considered a special kind of rule.

It is important to realize that rules are expressible in natural language in a variety of (possibly ambiguous) ways. Indeed, the same general rule can be expressed in many syntactically different forms, e.g.,

Everyone is mortal.
All humanity is mortal.
Every human being is mortal.
For each person the person is mortal.
If there is a member of the human race then he/she is mortal.

The answer to the question "what kinds of rules are expressible in ACE?" in this case is "all of the above"—all of the rules are valid ACE. Furthermore, each such pattern of rule can be further embellished with negation, relative clauses, etc. For example:

Every honest student who does not procrastinate receives a good mark and always passes the course easily.

TRANSLATOR also adds support for "infix" implication of the form:

```
Y ( and Y )* if X ( and X )*
```

The conclusions on the right-hand side are swapped with the conditions on the left-hand side and the word *then* is inserted, yielding the following equivalent (but now valid ACE) version:

```
If X ( and X )* then Y ( and Y )*
```

More concretely,

The student is happy if there is no class.

e.g., becomes,

If there is no class then the student is happy.

This preprocessing step is done before the input is sent to the APE Web service so that it never sees the original version. The rearrangement is not guaranteed to work (e.g., there may be anaphoric reference complications) but it is surely an improvement over the sentence being immediately rejected by APE.

The distinction between a fact and rule in natural language is sometimes a fine one. Consider this example business rule:

A gold customer is a customer who has more than 50 total purchases.

The above is indeed a rule but the Attempto Parsing Engine makes the (rather unlikely) interpretation that it is a fact. The rule can be more clearly stated as follows:

If a customer has more than 50 total purchases then he/she is a gold customer.

This "if . . . then . . . " construction seems to be the most natural and easily-recognizable way to express rules in natural language. Sometimes the *then* is omitted in favour of a comma in such rules; however, this is not valid ACE since the comma is reserved for specifying non-default precedence for sentence coordination.

There are actually several different kinds of rules; the examples so far have been derivation rules (also known as inference or deductive rules), which are primarily used for reasoning applications and do not invoke any actions. This category of rules is the focus for TRANSLATOR since RuleML is already well-developed in this area and the DRSs output by APE directly support this kind of implication.

Another type of rule is the integrity constraint, commonly found in databases. This type of rule simply expresses that something must always be true. For example:

The driver must be 16 years old.

Since ACE does not currently support modality (in this case, necessity), integrity constraints are not easily expressed, but extensions are being investigated.

Another common type is reaction rules, also known as ECA rules because they consist of an event, a condition and an action. The action is only performed when the event and condition are satisfied. A typical example is the following:

> *When a share price significantly drops and the company changes management then sell the stock.*

In ACE, sentences beginning with "when" are interpreted as questions, so this example is not valid. Even if the sentence is rephrased to

> *If a share price significantly drops and the company changes management then sell the stock.*

there is still the issue of the imperative verb *sell*. In fact, this makes expressing any sort of action in rules difficult in ACE. Production rules, perhaps the most common among businesses, also involve actions (and conditions, but not events). As with modality, extending ACE with imperatives is being looked into.

Unfortunately, there is currently a discrepancy introduced by the implementation of plurals as implications in ACE [17]. This results in apparent rules where there may not intuitively be any. However, the representation of plural nouns is currently being simplified, so this may no longer be an issue.

2.3 Discourse Representation Structures

The parsing engine APE, accessed as a Web service by TRANSLATOR, translates ACE texts into a syntactic variant of first-order logic called a Discourse Representation Structure (DRS). The utility of DRSs is a result of Discourse Representation Theory, a formal method of dealing with contextual meaning across multiple sentences [14]. In particular, DRSs provide a solution to the problem of anaphora resolution, e.g., whether *it* refers to the cat or the dog below:

> *The owner separates the cat from the dog. It growls.*

While only a brief introduction is given here, complete details are available in an Attempto report dedicated to DRSs [8].

A single DRS represents an entire discourse (group of sentences). Each entity (e.g., noun, verb) of the discourse is assigned a discourse referent (variable), which is listed at the top of the DRS. Below the referents is a list of related conditions (logical

atoms). For example, a DRS for the discourse above might have referents `F`, `B` and `D` and five conditions:

```
[F B D]
owner(F)
cat(B)
dog(D)
separate(F,B,D)
growl(D)
```

(Notice that it is the dog `D` that growls—the anaphor has been resolved.)

The conventional way to format a DRS is as a box:

F B D
owner(F) *cat(B)* *dog(D)* *separate(F,B,D)* *growl(D)*

In the Attempto system, the notation is a little different. Most importantly, predefined condition names are used so that, e.g., `cat(B)` becomes `object(B, cat, object)`:

A B C D E F G H
structure(F, atomic)-1 *quantity(F, cardinality, count_unit, G, eq, 1)-1* *object(F,* ***owner****, person)-1* *structure(B, atomic)-1* *quantity(B, cardinality, count_unit, C, eq, 1)-1* *object(B,* ***cat****, object)-1* *structure(D, atomic)-1* *quantity(D, cardinality, count_unit, E, eq, 1)-1* *object(D,* ***dog****, object)-1* *predicate(A, unspecified,* ***separate****, F, B)-1* *modifier(A, origin, from, D)-1* *predicate(H, unspecified,* ***growl****, D)-2*

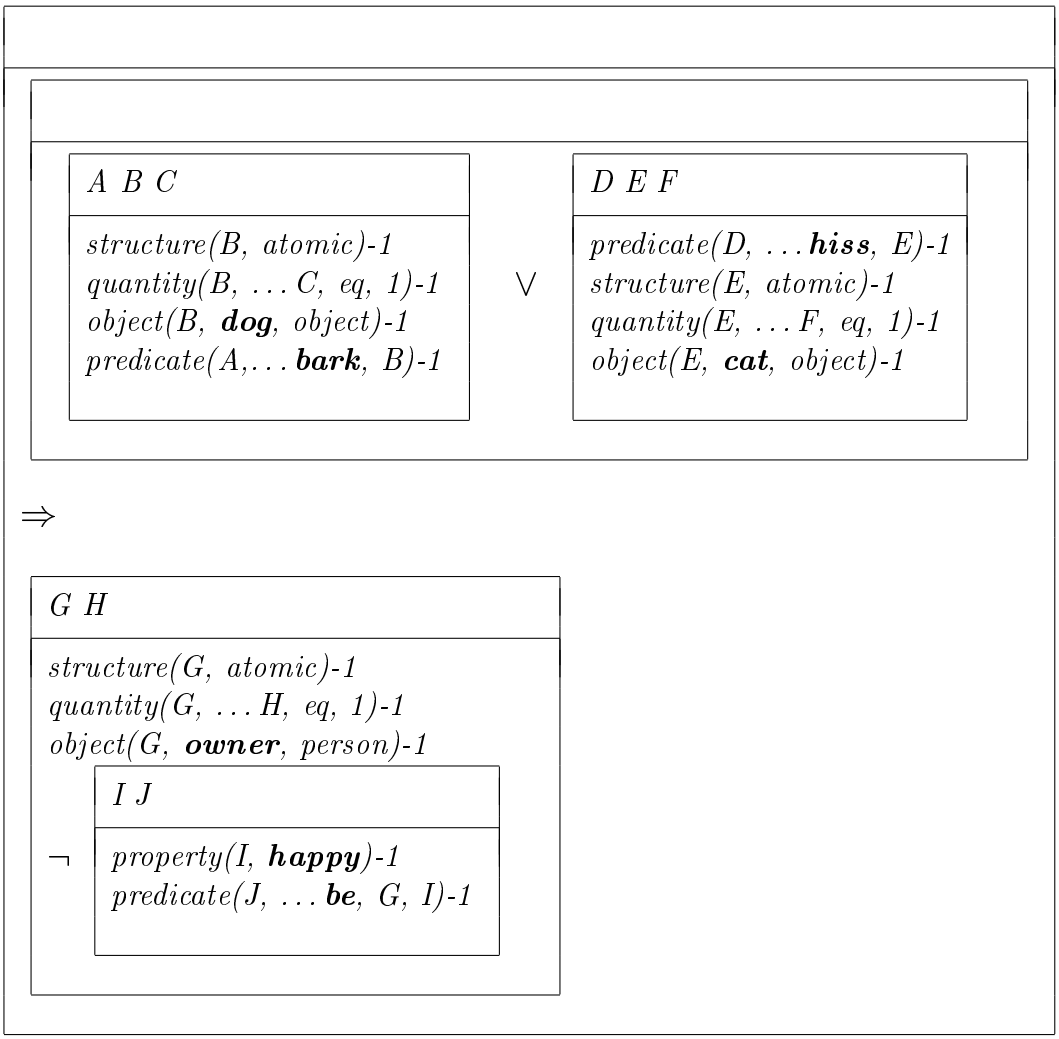
This "reified" representation allows using first-order logic where higher-order would normally be required. The Attempto representation also clearly introduces additional

referents and conditions to provide additional detail (e.g., quantity information). The integer at the end of each condition is just an index indicating the number of the corresponding sentence within the discourse.

The conditions so far have all been simple, but complex ones can be formed with nested DRSs and logical implication ($\Rightarrow$), negation ($\neg$) and disjunction ($\vee$). For example, the ACE text

If the dog barks or the cat hisses then the owner is not happy.

is translated into the following DRS:



TRANSLATOR translates such DRSs into RuleML. The DRS-RuleML mapping is described in section 4.3.

Chapter 3

Translation

Translating English into RuleML is complicated. Fortunately, the APE Web service greatly simplifies the task. Since details on the Attempto parser are already published [9], the focus here is on the user interface of TRANSLATOR and the series of steps that comprise the translation process. The actual DRS-RuleML mapping is exemplified in the next chapter.

3.1 User Interface

The two primary design goals of TRANSLATOR were that it be

- widely accessible, and
- user friendly.

In fact, both goals are satisfied by its availability as a Java Web Start application: TRANSLATOR can be accessed (cross-platform) online and provides users with an easy to use graphical user interface depicted in Figure 3.1.

The user interface is divided into an input pane on the top and an output pane on the bottom. The input pane contains a scrolling textarea where the user may enter an ACE text as well as the translation button, which may also be activated by shortcuts `Ctrl-Enter` or `Alt-T`. While translation is in progress (i.e., the APE Web service is being accessed), the translation button is temporarily disabled, a message is displayed below and the wait cursor is visible. The interface is multi-threaded in order to remain responsive to the user during this time.

The output pane consists of four checkboxes, a non-editable scrolling textarea where the translation results are displayed, a help button and a save button. The checkboxes

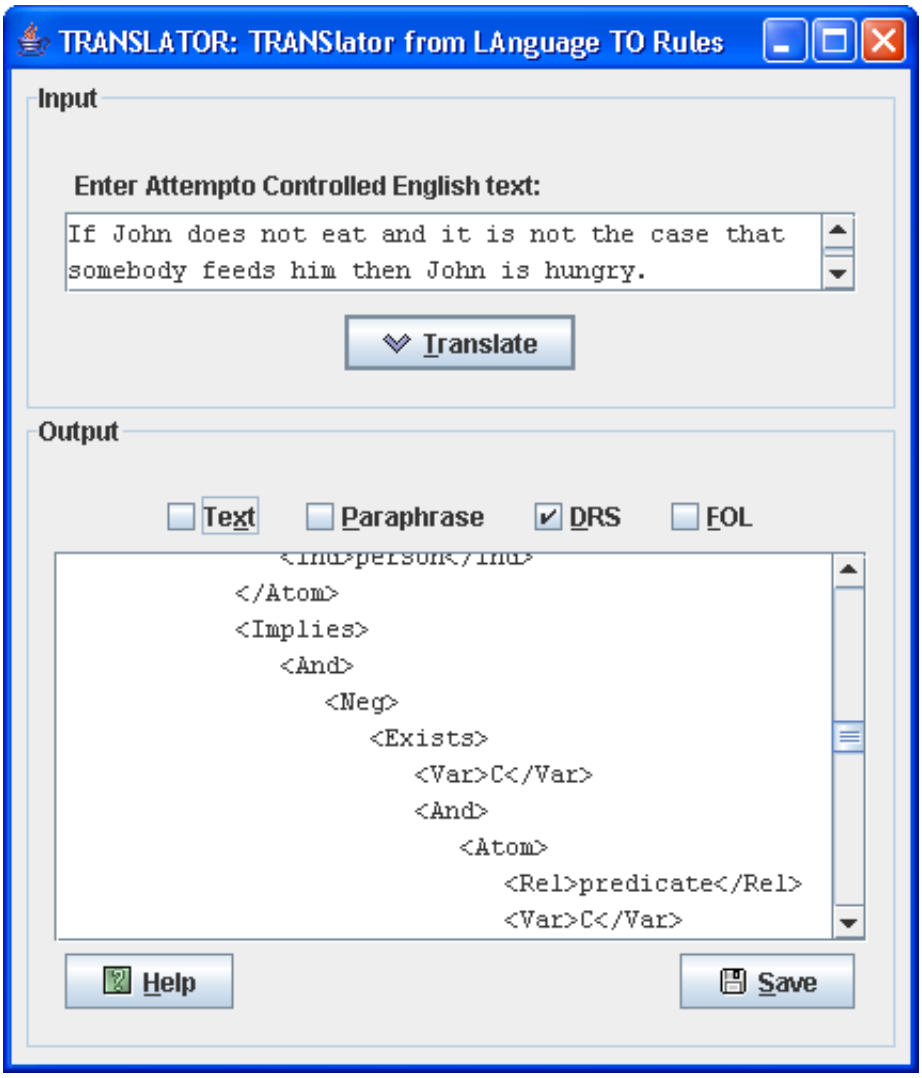


Figure 3.1: The graphical user interface of TRANSLATOR.

allow the user to customize what additional output (if any) is displayed within a comment at the top of the valid RuleML (e.g., the original ACE text, DRS, etc). Each can be independently checked or unchecked before translation or even after translation, in which case the output in the textbox is instantly updated accordingly. The help button (`Alt-H`) directs the user's browser to the TRANSLATOR webpage where additional information, helpful links and contact information are available. Finally, the save button (`Alt-S`) will save the current output to a file on the user's local machine for convenience. (Copying and pasting from the output textarea is another option, of course.) Note that the save button is disabled when there is no output to be saved.

3.2 Procedure

The translation process can be broken down into the following eight steps:

1. Accept the ACE text input by the user.
2. Preprocess the user input:
   - Reject any queries (ending with "?") since they are currently unsupported.
   - Add final "." if omitted. (Valid ACE sentences end with a period.)
   - If "infix implication" found, rearrange `Y if X` $\Rightarrow$ `If X then Y`.
3. Encode the input in a querystring and use it to access the APE Web service.
4. Retrieve XML-wrapped result from APE. (This may take several seconds.)
5. Parse result to get messages (if any), DRS, paraphrase, etc.
6. Display messages from APE, if any. End here if the result is otherwise empty (i.e. input was invalid ACE).
7. Traverse the DRS to build RuleML/XML.
8. Display the pretty-printed RuleML (and optional user-configurable extras) to the user.

An open source tree-based API called XOM is used for processing XML. The task of parsing the DRS in step 7 is simplified by JavaCC, another open source tool.

Chapter 4

Output: Rule Markup Language

The goal of the international Rule Markup Initiative is to define a canonical language for publishing and sharing rules on the Web. The result is the XML-based Rule Markup Language (RuleML), used for derivation, query, transformation, integrity checking and reactive behavior. RuleML is implemented with XML Schema, XSL Transformations (XSLT) and reasoning engines [2]. The RuleML Initiative collaborates with several standards bodies including W3C, OMG and OASIS.

4.1 Advantages

First of all, why RuleML as the target formalism? There are, in fact, several advantages to representation in RuleML.

RuleML inherits several benefits directly from XML, the open standard that forms the basis for its syntax. These include platform independence and interoperability, since the format is essentially structured text. Exchange is simplified because conformance to the RuleML specification (defined in XML Schema) can be confirmed using a validator such as the W3C's XSV. RuleML is also extensible, a prime example being its combination with OWL to form the Semantic Web Rule Language (SWRL) [13].

Another significant benefit to representation in RuleML is compatibility with other Semantic Web standards via XSL Transformations (XSLT). Translation from OWL, for example, is possible with OWLTrans[1]. Translators also exist for RuleML and RDF. Finally, RuleML is a major input to the W3C's upcoming RIF standard and compatibility is probable.

Other tools besides translators are available for RuleML and more are being de-

[1] http://www.ag-nbi.de/research/owltrans

veloped all the time. Reasoning engines, such as OO jDREW [1], Mandarax[2] and NxBRE[3], are a good example. RuleML is also a major part of an integrated toolkit for the Semantic Web called SweetRules[4]. Another useful tool is a bidirectional converter between RuleML and the more compact positional-slotted (POSL) syntax [5] combining Prolog and F-logic.

Another strength of RuleML is that its modular design can accommodate the diverse needs of its users. For example, rule subcommunities requiring a certain feature (e.g., negation-as-failure) are not burdened by others (e.g., complex terms) that they do not want. Some features, however, are built-in (but always optional) in RuleML, e.g., slots, datatypes and weights.

4.2 Modularization

The Rule Markup Language is actually a family of sublanguages realized with modular XML Schemas, following the general software engineering principle of modularity. A refined view of the modularization[5] of RuleML is shown in Figure 4.1.

Each sublanguage, represented as an unshaded rectangle, corresponds to a well-known rule system (e.g., datalog and hornlog), allowing users to pick and choose according to their specific needs. TRANSLATOR, for instance, uses the first-order logic sublanguage "folog". The shaded rectangles in the figure represent groups of related sublanguages. If ACE is later extended to handle negation-as-failure, the sublanguage "naffolog" (part of "FOL+") would be the best choice for representation in RuleML. As in the Unified Modeling Language (UML), a diamond-headed arrow indicates an aggregation association (e.g., "datalog is *part of* hornlog") while a regular-headed arrow indicates generalization.

4.3 DRS-RuleML Mapping

There are many possible mappings between DRSs and RuleML. An important criterion when selecting the mapping was reversibility in order to accommodate the future possibility of translation from RuleML back to ACE. Therefore, it is important that there be no loss of information; in particular, all details of the extended DRS notation used in the Attempto system should be preserved.

A direct mapping is therefore best. For example, the ACE text

[2] http://mandarax.sourceforge.net
[3] http://www.agilepartner.net/oss/nxbre
[4] http://sweetrules.projects.semwebcentral.org
[5] For the complete picture, see http://www.ruleml.org/modularization

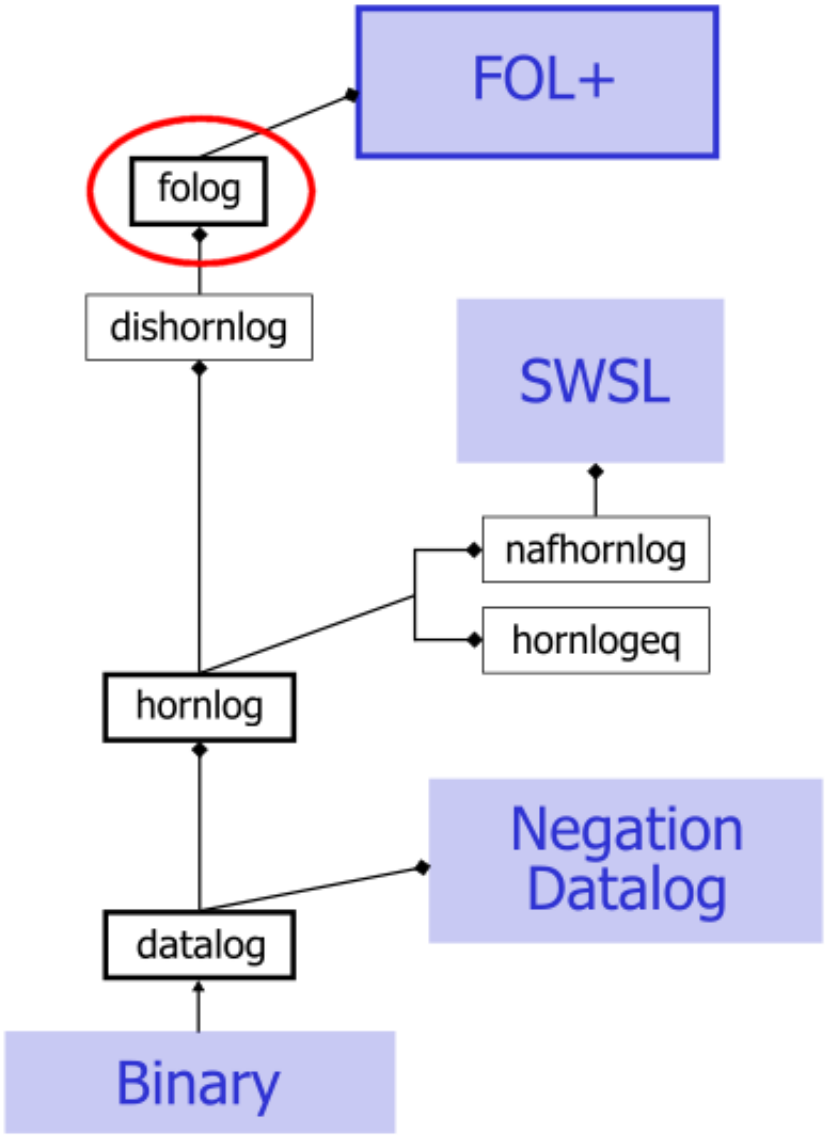


Figure 4.1: The modularization of RuleML.

The professor is mortal.

with the following DRS

A B C D
structure(C, atomic)-1 *quantity(C, ... 1)-1* *object(C,* ***professor*** *...)-1* *property(A,* ***mortal****)-1* *predicate(...* ***be****, C, A)-1*

is straightforwardly mapped into RuleML as follows:

```
<RuleML ... xmlns="http://www.ruleml.org/0.9/xsd">
 <Assert>
  <Exists>
   <Var>A</Var><Var>B</Var><Var>C</Var><Var>D</Var>
   <And>
    <Atom><Rel>structure</Rel><Var>C</Var><Ind>atomic</Ind></Atom>
    <Atom><Rel>quantity</Rel><Var>C</Var>...<Data>1</Data></Atom>
    <Atom><Rel>object</Rel><Var>C</Var><Ind>professor</Ind>...</Atom>
    <Atom><Rel>property</Rel><Var>A</Var><Ind>mortal</Ind></Atom>
    <Atom>
     <Rel>predicate</Rel>...<Ind>be</Ind><Var>C</Var><Var>A</Var>
    </Atom>
   </And>
  </Exists>
 </Assert>
</RuleML>
```

RuleML has a so-called "striped syntax" of class-like type tags and method-like role tags. The mapping by default uses the compact type-only ("role-skipped") syntax (as above), but an XSLT stylesheet exists to automatically reconstruct skipped role tags to arrive at the expanded form, which has the advantage of increased compatibility with RDF. As an illustration, the above `property` atom in expanded form becomes:

```
<formula>
 <Atom>
  <op><Rel>property</Rel></op>
  <arg index="1"><Var>A</Var></arg>
  <arg index="2"><Ind>mortal</Ind></arg>
 </Atom>
</formula>
```

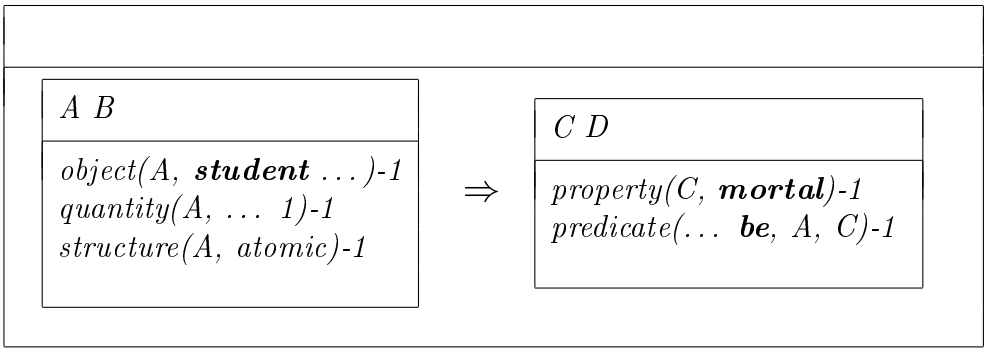
RuleML also supports non-positional "slotted" notation where each position in an atom is explicitly given a name. However, the positional form is instead used in order to be consistent with the DRSs. The content of each position of all the predefined predicates used in the extended DRS notation of Attempto is fully documented in the appendix of the DRS report [8].

Finally, the quantification of variables is made explicit in RuleML for clarity. It is implicit in a DRS that all variables are existentially quantified; RuleML has the `<Exists>` tag for this purpose. The only exceptions are variables on the condition side of an implication, which are universally quantified. In RuleML, these are placed within a `<Forall>` tag.

As a concrete example, the ACE text

> *Every student is mortal.*

results in the DRS



whose quantification is made explicit in the mapping to RuleML as follows (where the quantification tags are boxed for clarity):

```
<RuleML ... xmlns="http://www.ruleml.org/0.9/xsd">
 <Assert>
  <Forall>
   <Var>A</Var><Var>B</Var>
   <Implies>
    <And>
     <Atom><Rel>object</Rel><Var>A</Var><Ind>student</Ind>...</Atom>
     <Atom><Rel>quantity</Rel><Var>A</Var>...<Data>1</Data></Atom>
     <Atom><Rel>structure</Rel><Var>A</Var><Ind>atomic</Ind></Atom>
    </And>
    <Exists>
     <Var>C</Var><Var>D</Var>
     <And>
```

```
          <Atom><Rel>property</Rel><Var>C</Var><Ind>mortal</Ind></Atom>
          <Atom>
           <Rel>predicate</Rel>...<Ind>be</Ind><Var>A</Var><Var>C</Var>
          </Atom>
         </And>
        </Exists>
       </Implies>
      </Forall>
     </Assert>
    </RuleML>
```

A complete colour-coded example is given in Appendix A. The grammar for TRANSLATOR's DRS parser, which implements this DRS-RuleML mapping, is included as Appendix B.

A more standard non-reified representation in RuleML (e.g., with nouns and verbs as relations) could be derived from the version above using XSLT translation. One possibility is as follows:

```
<RuleML ... xmlns="http://www.ruleml.org/0.9/xsd">
 <Assert>
  <Implies>
   <Atom><Rel>student</Rel><Var>A</Var></Atom>
   <And>
    <Atom><Rel>mortal</Rel><Var>C</Var></Atom>
    <Equal><Var>A</Var><Var>C</Var></Equal>
   </And>
  </Implies>
 </Assert>
</RuleML>
```

Note that the explicit quantification and extra atoms are gone, and that the remaining atoms have been un-reified (to have a more meaningful relation name). Finally, an `<Equal>` element has replaced the following atom:

```
<Atom><Rel>predicate</Rel>...<Ind>be</Ind><Var>A</Var><Var>C</Var></Atom>
```

For regular verbs, however, this extra measure would not be necessary.

Chapter 5

Future Work

There are several possible avenues of future development for TRANSLATOR, most of which involve continued collaboration with the Attempto project.

5.1 Queries

The only aspect of ACE not currently supported by TRANSLATOR is questions. Normally these would be mapped to `<Query>` in RuleML, but currently the DRS for the query

> *Does the man enter a card?*

is identical to the DRS for

> *The man enters a card.*

and it would be impossible to identify when to use `<Query>` versus `<Assert>`. If the DRS included a `query` predicate as the DRS for, e.g., "who", "what", "when" and "where" questions already does, this would not be an issue.

5.2 Translators

Once represented in RuleML, the rules can be automatically translated to other XML-based languages using XSLT (e.g., OWL and RDF). There are also useful opportu-

nities for translation within RuleML, e.g., simplifying the "enriched" representation as sketched at the end of section 4.3.

5.3 Verbalization

A prototype tool called DRACE is already available for translating a DRS into ACE (subject to some limitations) [10]. If this tool were accessible (e.g., as a Web service), TRANSLATOR could be extended to be bidirectional, supporting translation from RuleML back to ACE by reversing the DRS-RuleML mapping.

5.4 ACE Extensions

The Attempto team is extending ACE in several ways as part of the EU Network of Excellence Reasoning on the Web with Rules and Semantics (REWERSE). Planned extensions include:

- negation-as-failure to supplement the existing logical negation [6]
- modality, for expressing, e.g., possibility (*can*, *may*), necessity (*must*, *always*), obligation (*should*, *shall*), etc. [22]
- imperatives, useful, e.g., for dialogue systems and expressing action rules [11]

Once implemented, these new features should, of course, also be supported by TRANSLATOR.

Chapter 6

Conclusion

TRANSLATOR is an open source tool that allows anyone to write facts and rules in formal representation for use on the Semantic Web. It accomplishes this by taking input written in ACE, accessing the APE Web service and then translating the generated DRS into RuleML. The hope is that this user-friendly front-end will help lower the barrier to entry of the Semantic Web and encourage non-experts to get involved; this seems to have been a critical factor in the success of the original Web.

One potential use of TRANSLATOR is to construct real-world use cases, e.g., for company policies. One open issue is that the output of TRANSLATOR, as first-order logic, is undecidable and therefore not supported by reasoning engines. However, reorganization into a decidable subset should be possible, e.g., by factoring out multiple conclusions of a rule.

While TRANSLATOR is already useful, in its present form it is mostly restricted to derivation rules (and, of course, facts). This limitation is inherited from ACE and RuleML, but extensions already in progress will grant the increased expressivity necessary for integrity constraints, reaction rules, etc. Extending TRANSLATOR to be bidirectional, i.e., also capable of translating RuleML into ACE, is also planned.

Meanwhile, development of the RIF standard continues. Fortunately, the future challenge of relating TRANSLATOR to RIF (once defined) should be straightforward since RuleML will most likely be translatable to RIF. TRANSLATOR would therefore be one of the tools referred to in the Working Group charter that obviate the need for users to work directly with RIF.

TRANSLATOR is available as a Java Web Start application at http://www.ruleml.org/translator and is also linked to from the Attempto website[1]. Further collaboration with the Attempto team is anticipated.

[1] http://www.ifi.unizh.ch/attempto/tools

Bibliography

[1] Marcel Ball. OO jDREW: Design and Implementation of a Reasoning Engine for the Semantic Web. Honours Thesis, Faculty of Computer Science, University of New Brunswick, 2005. http://www.jdrew.org/oojdrew/docs.html.

[2] Marcel Ball, Harold Boley, David Hirtle, Jing Mei, and Bruce Spencer. Implementing RuleML Using Schemas, Translators, and Bidirectional Interpreters. W3C Workshop on Rule Languages for Interoperability Position Paper, April 2005. http://www.w3.org/2004/12/rules-ws/paper/49.

[3] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[4] Harold Boley. Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms. In *RuleML*, pages 1–16, 2003.

[5] Harold Boley. POSL: An Integrated Positional-Slotted Language for Semantic Web Knowledge, May 2004. http://www.ruleml.org/submission/ruleml-shortation.html.

[6] Piero A. Bonatti et al. Deliverable I2-D1. Rule-based Policy Specification: State of the Art and Future Work. Technical report, REWERSE, 2004. http://rewerse.net/deliverables.html.

[7] Norbert E. Fuchs, Stefan Höfler, Kaarel Kaljurand, Fabio Rinaldi, and Gerold Schneider. Attempto Controlled English: A Knowledge Representation Language Readable by Humans and Machines. In *Reasoning Web*, pages 213–250, 2005.

[8] Norbert E. Fuchs, Stefan Höfler, Gerold Schneider, and Uta Schwertel. Extended Discourse Representation Structures in Attempto Controlled English. Technical Report ifi-2005.08, Department of Informatics, University of Zurich, 2005. http://www.ifi.unizh.ch/attempto/publications.

[9] Norbert E. Fuchs, Kaarel Kaljurand, Fabio Rinaldi, and Gerold Schneider. Deliverable I2-D3. A Parser for Attempto Controlled English. Technical report, REWERSE, 2005. http://rewerse.net/deliverables.html.

[10] Norbert E. Fuchs, Kaarel Kaljurand, and Gerold Schneider. Deliverable I2-D5. Verbalising Formal Languages in Attempto Controlled English I. Technical report, REWERSE, 2005. http://rewerse.net/deliverables.html.

[11] Norbert E. Fuchs, Kaarel Kaljurand, and Gerold Schneider. Attempto Controlled English Meets the Challenges of Knowledge Representation, Reasoning, Interoperability and User Interfaces. In *FLAIRS 2006*, 2006. To be published.

[12] Stefan Haustein and Jörg Pleumann. Is Participation in the Semantic Web Too Difficult? In *International Semantic Web Conference*, pages 448–453, 2002.

[13] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, May 2004. http://www.w3.org/Submission/SWRL.

[14] Hans Kamp and Uwe Reyle. *From Discourse to Logic: Introduction to Model-theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Kluwer, Dordrecht, 1993.

[15] Massimo Marchiori. Towards a People's Web: Metalog. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 320–326, 2004.

[16] Adam Pease and William Murray. An English to Logic Translator for Ontology-based Knowledge Representation Languages. In *Proceedings of the 2003 IEEE International Conference on Natural Language Processing and Knowledge Engineering*, pages 777–783, 2003.

[17] Uta Schwertel. Plural Semantics for Natural Language Understanding – A Computational Proof-Theoretic Approach. Technical report, Department of Informatics, University of Zurich, 2004. http://www.ifi.unizh.ch/attempto/publications.

[18] Rolf Schwitter, Anna Ljungberg, and David Hood. ECOLE - A Look-ahead Editor for a Controlled Language. In *Controlled Translation, Proceedings of EAMT-CLAW03*, pages 141–150, 2003.

[19] Rolf Schwitter and Marc Tilbrook. Controlled Natural Language meets the Semantic Web. In *Proceedings of the Australasian Language Technology Workshop*, pages 55–62, 2004.

[20] John F. Sowa. Common Logic Controlled English, 2004. http://www.jfsowa.com/clce/specs.htm.

[21] Business Rules Team. Semantics of Business Vocabulary & Business Rules (SBVR). W3C Workshop on Rule Languages for Interoperability Position Paper, March 2005. http://www.w3.org/2004/12/rules-ws/paper/85.

[22] Gerd Wagner, Sergey Lukichev, Norbert E. Fuchs, and Silvie Spreeuwenberg. Deliverable I1-D2. First-Version Controlled English Rule Language. Technical report, REWERSE, 2005. http://rewerse.net/deliverables.html.

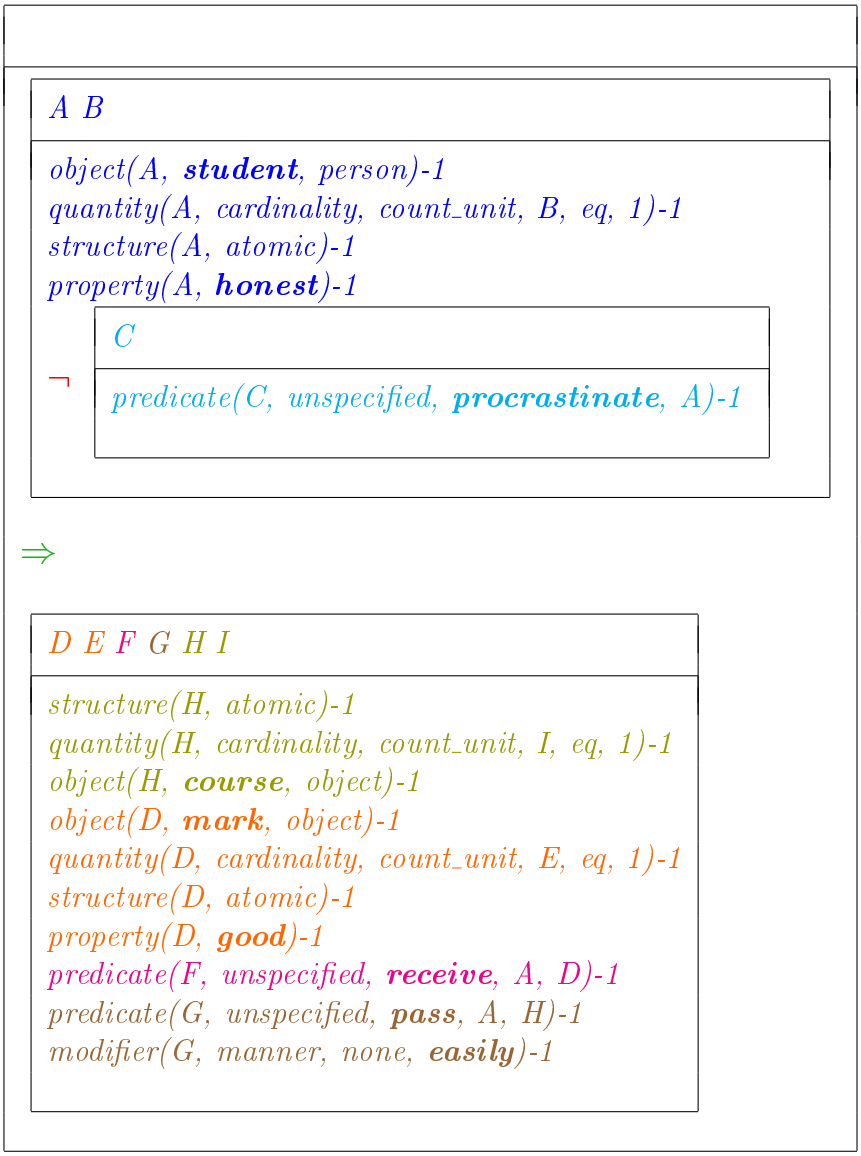
Appendix A

Colour-coded ACE-DRS-RuleML Example

ACE sentence:

Every ***honest*** ***student*** *who does not* ***procrastinate*** ***receives*** *a* ***good*** ***mark*** *and* ***easily*** ***passes*** *the* ***course****.*

DRS:



RuleML:

```
<RuleML
 xsi:schemaLocation="http://www.ruleml.org/0.9/xsd
                     http://www.ruleml.org/0.9/xsd/folog.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.ruleml.org/0.9/xsd">
   <Assert>
      <Forall>
         <Var>A</Var>
         <Var>B</Var>
         <Implies>
            <And>
               <Atom>
                  <Rel>object</Rel>
                  <Var>A</Var>
                  <Ind>student</Ind>
                  <Ind>person</Ind>
               </Atom>
               <Atom>
                  <Rel>quantity</Rel>
                  <Var>A</Var>
                  <Ind>cardinality</Ind>
                  <Ind>count_unit</Ind>
                  <Var>B</Var>
                  <Ind>eq</Ind>
                  <Data>1</Data>
               </Atom>
               <Atom>
                  <Rel>structure</Rel>
                  <Var>A</Var>
                  <Ind>atomic</Ind>
               </Atom>
               <Atom>
                  <Rel>property</Rel>
                  <Var>A</Var>
                  <Ind>honest</Ind>
               </Atom>
               <Neg>
                  <Exists>
                     <Var>C</Var>
                     <And>
                        <Atom>
                           <Rel>predicate</Rel>
                           <Var>C</Var>
                           <Ind>unspecified</Ind>
                           <Ind>procrastinate</Ind>
```

```
                <Var>A</Var>
              </Atom>
            </And>
          </Exists>
        </Neg>
      </And>
      <Exists>
        <Var>D</Var>
        <Var>E</Var>
        <Var>F</Var>
        <Var>G</Var>
        <Var>H</Var>
        <Var>I</Var>
        <And>
          <Atom>
            <Rel>structure</Rel>
            <Var>H</Var>
            <Ind>atomic</Ind>
          </Atom>
          <Atom>
            <Rel>quantity</Rel>
            <Var>H</Var>
            <Ind>cardinality</Ind>
            <Ind>count_unit</Ind>
            <Var>I</Var>
            <Ind>eq</Ind>
            <Data>1</Data>
          </Atom>
          <Atom>
            <Rel>object</Rel>
            <Var>H</Var>
            <Ind>course</Ind>
            <Ind>object</Ind>
          </Atom>
          <Atom>
            <Rel>object</Rel>
            <Var>D</Var>
            <Ind>mark</Ind>
            <Ind>object</Ind>
          </Atom>
          <Atom>
            <Rel>quantity</Rel>
            <Var>D</Var>
            <Ind>cardinality</Ind>
            <Ind>count_unit</Ind>
            <Var>E</Var>
            <Ind>eq</Ind>
```

```
                    <Data>1</Data>
                </Atom>
                <Atom>
                    <Rel>structure</Rel>
                    <Var>D</Var>
                    <Ind>atomic</Ind>
                </Atom>
                <Atom>
                    <Rel>property</Rel>
                    <Var>D</Var>
                    <Ind>good</Ind>
                </Atom>
                <Atom>
                    <Rel>predicate</Rel>
                    <Var>F</Var>
                    <Ind>unspecified</Ind>
                    <Ind>receive</Ind>
                    <Var>A</Var>
                    <Var>D</Var>
                </Atom>
                <Atom>
                    <Rel>predicate</Rel>
                    <Var>G</Var>
                    <Ind>unspecified</Ind>
                    <Ind>pass</Ind>
                    <Var>A</Var>
                    <Var>H</Var>
                </Atom>
                <Atom>
                    <Rel>modifier</Rel>
                    <Var>G</Var>
                    <Ind>manner</Ind>
                    <Ind>none</Ind>
                    <Var>easily</Var>
                </Atom>
            </And>
        </Exists>
    </Implies>
</Forall>
</Assert>
</RuleML>
```

Appendix B

Grammar for DRS Parser

The following file, `DRSParser.jj`, is a grammar for DRSs used by the open source tool JavaCC to generate the actual DRS parser used by TRANSLATOR. The notation used is similar to standard BNF except that actions are also embedded in the grammar[1]. These actions are Java statements surrounded by braces (e.g., `{ System.out.println("start"); }`) that are executed when traversed in order to simultaneously build the RuleML representation and parse the DRS.

```
options { JAVA_UNICODE_ESCAPE = true; }

PARSER_BEGIN(DRSParser)

  package org.ruleml.translator.parser;
  import nu.xom.*;
  import java.util.*;

  public class DRSParser
  {
    final static String RULEML_NAMESPACE =
      "http://www.ruleml.org/0.9/xsd";
  }

PARSER_END(DRSParser)

SKIP : /* white space */
{
  " "
| "\t"
| "\n"
| "\r"
```

[1]The notation is documented at https://javacc.dev.java.net/doc/javaccgrm.html

```
| "\f"
}

TOKEN : /* punctuation and logical connectors */
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACK: "[" >
| < RBRACK: "]" >
| < QUOTE: "'" >
| < COMMA: "," >
| < IMPLIES: "=>" >
| < DASH: "-" >
| < OR: "v" >
}

TOKEN : /* keywords */
{
  < DRS: "drs" >
| < NEG_DRS: "-drs" >
| < KG: "kg" >
| < CM: "cm" >
| < EQ: "eq" >
| < OF: "of" >
| < DOM: "dom" >
| < LEQ: "leq" >
| < GEQ: "geq" >
| < SIZE: "size" >
| < TIME: "time" >
| < MASS: "mass" >
| < LESS: "less" >
| < UNIT: "unit" >
| < STATE: "state" >
| < GROUP: "group" >
| < LITER: "liter" >
| < EVENT: "event" >
| < ATOMIC: "atomic" >
| < VOLUME: "volume" >
| < LENGTH: "length" >
| < PERSON: "person" >
| < WEIGHT: "weight" >
| < GREATER: "greater" >
| < DIMENSION: "dimension" >
| < COUNT: "count_unit" >
| < CARDINALITY: "cardinality" >
| < UNSPECIFIED: "unspecified" >
}
```

```
TOKEN : /* predicates (note that query is not supported) */
{
  < OBJECT: "object" >
| < NAMED: "named" >
| < STRUCTURE: "structure" >
| < QUANTITY: "quantity" >
| < PREDICATE: "predicate" >
| < PROPERTY: "property" >
| < QUOTED: "quoted_string" >
| < MODIFIER: "modifier" >
| < PROPER: "proper_part_of" >
| < PART: "part_of" >
| < RELATION: "relation" >
| < SUM: "sum_of" >
}

TOKEN :
{
  < REFERENT: ["A"-"Z"] (<INTEGER>)? >
| < INTEGER: ( ["1"-"9"] (<DIGIT>)* | "0" ) >
| < QUOTED_STRING: <QUOTE> <STRING> <QUOTE> >
| < STRING: (<LETTER> | "-")+ >
| < #DIGIT: ["0"-"9"] >
| < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
}

TOKEN :
{
 <
  QUOTED_SPACED_STRING:
```

```
  <QUOTE> (<LETTER> | "-") ( (" ")* (<LETTER> | "-") )* <QUOTE>
 >
}

Element Start() :
{
  Element elem;
  Element as = new Element("Assert", RULEML_NAMESPACE);
}
{
  { System.out.println("DRS parser started"); }
  elem = DRS()
  <EOF>
  {
    as.appendChild(elem);
    return as;
  }
}

Element DRS() :
{
  List vars = null;
  Element exists = new Element("Exists", RULEML_NAMESPACE);
  Element and, child;
}
{
  <DRS>
  <LPAREN>
  <LBRACK>
  ( vars = Referents()
    {
      if (vars != null)
      {
        for (int i=0; i < vars.size(); i++)
          exists.appendChild((Element)vars.get(i));
      }
    }
  )?
  <RBRACK>
  <COMMA>
  <LBRACK>
    and = Conditions()
  <RBRACK>
  <RPAREN>
  {
    // skip over empty exists elements
    if ((vars == null) && (and.getChildCount() == 1))
```

```
    {
      child = (Element)and.getChild(0);
      child.detach();
      return child;
    }
    else
    {
      exists.appendChild(and);
      return exists;
    }
  }
}

Element NegDRS() :
{
  List vars = null;
  Element exists = new Element("Exists", RULEML_NAMESPACE);
  Element neg = new Element("Neg", RULEML_NAMESPACE);
  Element and, child;
}
{
  <NEG_DRS>
  <LPAREN>
  <LBRACK>
  ( vars = Referents()
    {
      if (vars != null)
      {
        for (int i=0; i < vars.size(); i++)
          exists.appendChild((Element)vars.get(i));
      }
    }
  )?
  <RBRACK>
  <COMMA>
  <LBRACK>
    and = Conditions()
  <RBRACK>
  <RPAREN>
  {
    // skip over empty exists elements
    if ((vars == null) && (and.getChildCount() == 1))
    {
      child = (Element)and.getChild(0);
      child.detach();
      neg.appendChild(child);
      return neg;
```

```
    }
    else
    {
      exists.appendChild(and);
      neg.appendChild(exists);
      return neg;
    }
  }
}

// for universally quantified DRSs in the body of an implication
Element RDRS() :
{
  List vars = null;
  Element forall = new Element("Forall", RULEML_NAMESPACE);
  Element and;
  Element implies = new Element("Implies", RULEML_NAMESPACE);
}
{
  <DRS>
  <LPAREN>
  <LBRACK>
  ( vars = Referents()
    {
      if (vars != null)
      {
        for (int i=0; i < vars.size(); i++)
          forall.appendChild((Element)vars.get(i));
      }
    }
  )?
  <RBRACK>
  <COMMA>
  <LBRACK>
    and = Conditions()
  <RBRACK>
  <RPAREN>
  {
    implies.appendChild(and);
    if (vars != null)
    {
      forall.appendChild(implies);
      return forall;
    }
    else return implies;
  }
}
```

```
List Referents() :
{
  List refs = new ArrayList();
  Token t;
  Element var;
}
{
  t = <REFERENT>
    {
      var = new Element("Var", RULEML_NAMESPACE);
      var.appendChild(t.image);
      refs.add(var);
    }
  (
    <COMMA> t = <REFERENT>
    {
      var = new Element("Var", RULEML_NAMESPACE);
      var.appendChild(t.image);
      refs.add(var);
    }
  )*
  {
    return refs;
  }
}

Element Conditions() :
{
  List conditions = new ArrayList();
  Element elem = null;
  Element and = new Element("And", RULEML_NAMESPACE);
}
{
  elem = Condition() { conditions.add(elem); }
  ( <COMMA> elem = Condition() { conditions.add(elem); } )*
  {
    for (int i=0; i < conditions.size(); i++)
      and.appendChild((Element)conditions.get(i));
    return and;
  }
}

Element Condition() :
{ Element atom = null, elem = null; }
{
  atom=SimpleCondition() { return atom; }
```

```
|
  elem=ComplexCondition() { return elem; }
}

Element SimpleCondition() :
{ Element atom; }
{
  atom = Predicate()
  "-"<INTEGER>
  { return atom; }
}

Element ComplexCondition() :
{
  Element neg, element, exists, exists2, implies, or;
}
{
  neg=NegDRS()
  { return neg; }
| LOOKAHEAD( DRS() <OR> ) // syntactic lookahead
  exists=DRS() <OR> exists2=DRS()
  {
    or = new Element("Or", RULEML_NAMESPACE);
    or.appendChild(exists);
    or.appendChild(exists2);
    return or;
  }
|
  element=RDRS() <IMPLIES> exists=DRS()
  {
    if (element.getLocalName().equals("Implies"))
    {
      element.appendChild(exists);
    }
    else // element is "forall"
    {
      implies =
       element.getChildElements("Implies",RULEML_NAMESPACE).get(0);
      implies.appendChild(exists);
    }
    return element;
  }
}

Element Predicate() :
{ Element atom; }
{
```

```
  ( atom = NamedPred()
  | atom = StructurePred()
  | atom = QuantityPred()
  | atom = ObjectPred()
  | atom = PredicatePred()
  | atom = PropertyPred()
  | atom = QuotedPred()
  | atom = ModifierPred()
  | atom = ProperPred()
  | atom = PartPred()
  | atom = RelationPred()
  | atom = SumPred()
  )
  { return atom; }
}

Element NamedPred() :
{ Element atom, rel, var, ind; Token r,s,t; }
{
  r = <NAMED> <LPAREN>
  s = <REFERENT> <COMMA>
  ( t = <QUOTED_STRING> | t=<STRING> )
  <RPAREN>
  {
    System.out.println("Recognized named");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    return atom;
  }
}

Element StructurePred() :
{ Element atom, rel, var, ind; Token r,s,t; }
{
  r = <STRUCTURE> <LPAREN>
  s = <REFERENT> <COMMA>
  ( t=<ATOMIC> | t=<GROUP> | t=<MASS> | t=<DOM> )
  <RPAREN>
  {
```

```
    System.out.println("Recognized structure");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    return atom;
  }
}

Element QuantityPred() :
{ Element atom, rel, var, ind, data; Token r,s,t,u,v,w,x; }
{
  r = <QUANTITY> <LPAREN>
  s = <REFERENT> <COMMA>

  // {cardinality, weight, size, ...}
  ( t=<STRING> | t=AnyReservedWord() ) <COMMA>

  // {count_unit, kg, cm, ...}
  ( u=<STRING> | u=AnyReservedWord() ) <COMMA>

  v = <REFERENT> <COMMA>
  ( w=<EQ> | w=<LEQ> | w=<GEQ> | w=<GREATER> | w=<LESS> ) <COMMA>
  ( x=<INTEGER> | x=<UNSPECIFIED> )
  <RPAREN>
  {
    System.out.println("Recognized quantity");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(u.image);
    atom.appendChild(ind);
    var = new Element("Var", RULEML_NAMESPACE);
```

```
      var.appendChild(v.image);
      atom.appendChild(var);
      ind = new Element("Ind", RULEML_NAMESPACE);
      ind.appendChild(w.image);
      atom.appendChild(ind);
      data = new Element("Data", RULEML_NAMESPACE);
      data.appendChild(x.image);
      atom.appendChild(data);
      return atom;
    }
}

Element ObjectPred() :
{ Element atom, rel, var, ind; Token r,s,t,u; }
{
  r = <OBJECT> <LPAREN>
  s = <REFERENT> <COMMA>

  // noun
  ( t=<STRING> | t=<QUOTED_STRING> | t=AnyReservedWord() ) <COMMA>

  ( u=<PERSON> | u=<OBJECT> | u=<TIME> | u=<UNSPECIFIED> )
  <RPAREN>
  {
    System.out.println("Recognized object");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(u.image);
    atom.appendChild(ind);
    return atom;
  }
}

Element PropertyPred() :
{ Element atom, rel, var, ind; Token r,s,t,u=null,v=null; }
{
  r = <PROPERTY> <LPAREN>
  s = <REFERENT> <COMMA>
```

```
  // adjective/comparative
  ( t=<STRING> | t=<QUOTED_STRING> | t=AnyReservedWord() )

  ( <COMMA> u = <REFERENT> (<COMMA> v = <REFERENT>)? )?
  <RPAREN>
  {
    System.out.println("Recognized property");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    if (u != null)
    {
      var = new Element("Var", RULEML_NAMESPACE);
      var.appendChild(u.image);
      atom.appendChild(var);
    }
    if (v != null)
    {
      var = new Element("Var", RULEML_NAMESPACE);
      var.appendChild(v.image);
      atom.appendChild(var);
    }
    return atom;
  }
}

Element PredicatePred() :
{ Element atom, rel, var, ind; Token r,s,t,u,v,w=null,x=null; }
{
  r = <PREDICATE> <LPAREN>
  s = <REFERENT> <COMMA>
  ( t=<EVENT> | t=<STATE> | t=<UNSPECIFIED> ) <COMMA>

  // verb
  ( u=<STRING> | u=<QUOTED_STRING> | u=AnyReservedWord() ) <COMMA>

  v = <REFERENT>
  ( <COMMA> w=<REFERENT> (<COMMA> x=<REFERENT>)? )?
  <RPAREN>
```

```
  {
    System.out.println("Recognized predicate");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(u.image);
    atom.appendChild(ind);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(v.image);
    atom.appendChild(var);
    if (w != null)
    {
      var = new Element("Var", RULEML_NAMESPACE);
      var.appendChild(w.image);
      atom.appendChild(var);
    }
    if (x != null)
    {
      var = new Element("Var", RULEML_NAMESPACE);
      var.appendChild(x.image);
      atom.appendChild(var);
    }
    return atom;
  }
}

Element QuotedPred() :
{ Element atom, rel, var, ind; Token r,s,t; }
{
  r = <QUOTED> <LPAREN>
  s = <REFERENT> <COMMA>
  ( t=<QUOTED_SPACED_STRING> | t=AnyReservedWord() )
  <RPAREN>
  {
    System.out.println("Recognized quoted_string");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
```

```
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    return atom;
  }
}

Element ModifierPred() :
{ Element atom, rel, var, ind; Token r,s,t,u,v; }
{
  r = <MODIFIER> <LPAREN>
  s = <REFERENT> <COMMA>

  // { location, origin, ...}
  ( t=<STRING> | t=AnyReservedWord() ) <COMMA>

  // preposiion
  ( u=<STRING> | u=<QUOTED_STRING> | u=AnyReservedWord() ) <COMMA>

  // referent or adverb
  (
    v=<REFERENT> | v=<STRING> |
    v=<QUOTED_STRING> | v=AnyReservedWord()
  )

  <RPAREN>
  {
    System.out.println("Recognized modifier");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(u.image);
    atom.appendChild(ind);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(v.image);
    atom.appendChild(var);
```

```
    return atom;
  }
}

Element ProperPred() :
{ Element atom, rel, var; Token r,s,t; }
{
  r = <PROPER> <LPAREN>
  s = <REFERENT> <COMMA>
  t = <REFERENT>
  <RPAREN>
  {
    System.out.println("Recognized proper_part_of");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(t.image);
    atom.appendChild(var);
    return atom;
  }
}

Element PartPred() :
{ Element atom, rel, var; Token r,s,t; }
{
  r = <PART> <LPAREN>
  s = <REFERENT> <COMMA>
  t = <REFERENT>
  <RPAREN>
  {
    System.out.println("Recognized part_of");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(t.image);
    atom.appendChild(var);
    return atom;
```

```
  }
}

Element RelationPred() :
{ Element atom, rel, ind, var; Token r,s,t,u,v; }
{
  r = <RELATION> <LPAREN>
  s = <REFERENT> <COMMA>
  ( t=<STRING> | t=AnyReservedWord() ) <COMMA>
  u = <OF> <COMMA>
  v = <REFERENT>
  <RPAREN>
  {
    System.out.println("Recognized relation");
    atom = new Element("Atom", RULEML_NAMESPACE);
    rel = new Element("Rel", RULEML_NAMESPACE);
    rel.appendChild(r.image);
    atom.appendChild(rel);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(s.image);
    atom.appendChild(var);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(t.image);
    atom.appendChild(ind);
    ind = new Element("Ind", RULEML_NAMESPACE);
    ind.appendChild(u.image);
    atom.appendChild(ind);
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(v.image);
    atom.appendChild(var);
    return atom;
  }
}

Element SumPred() :
{
  Element atom, rel, plex, var; Token r,s,t;
  List refs = new ArrayList();
}
{
  r = <SUM> <LPAREN>
  s = <REFERENT> <COMMA>
  <LBRACK> t = <REFERENT>
  {
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(t.image);
    refs.add(var);
```

```
  }
  ( <COMMA> t = <REFERENT>
   {
    var = new Element("Var", RULEML_NAMESPACE);
    var.appendChild(t.image);
    refs.add(var);
   }
  )* <RBRACK>
 <RPAREN>
 {
   System.out.println("Recognized sum_of");
   atom = new Element("Atom", RULEML_NAMESPACE);
   rel = new Element("Rel", RULEML_NAMESPACE);
   rel.appendChild(r.image);
   atom.appendChild(rel);
   var = new Element("Var", RULEML_NAMESPACE);
   var.appendChild(s.image);
   atom.appendChild(var);
   plex = new Element("Plex", RULEML_NAMESPACE);
   atom.appendChild(plex);
   if (refs != null)
   {
     for (int i=0; i < refs.size(); i++)
     plex.appendChild((Element)refs.get(i));
   }
   return atom;
 }
}

Token AnyReservedWord() :
{ Token t; }
{
 (
   t=<OBJECT> | t=<NAMED> | t=<STRUCTURE> | t=<QUANTITY> |
   t=<PREDICATE> | t=<PROPERTY> | t=<QUOTED> | t=<MODIFIER> |
   t=<PROPER> | t=<PART> | t=<RELATION> | t=<SUM> | t=<DRS> |
   t=<KG> | t=<CM> | t=<EQ> | t=<OF> | t=<DOM> | t=<LEQ> |
   t=<GEQ> | t=<SIZE> | t=<TIME> | t=<MASS> | t=<LESS> |
   t=<UNIT> | t=<STATE> | t=<GROUP> | t=<LITER> | t=<EVENT> |
   t=<ATOMIC> | t=<VOLUME> | t=<LENGTH> | t=<PERSON> |
   t=<WEIGHT> | t=<GREATER> | t=<DIMENSION> | t=<COUNT> |
   t=<CARDINALITY> | t=<UNSPECIFIED>
 )
 { return t; }
}
```