CS6795 Semantic Web Techniques
Project Report

XSLT 2.0 Translation of Datalog$^+$ RuleML 1.01/XML to a Subset of the TPTP Language

Meng Luan and Changyang Liu

University of New Brunswick, Faculty of Computer Science

Dec 4, 2014

Professor: Dr. Harold Boley
Advisor: Dr. Tara Athan

1. Introduction

This project is aimed at implementing an XSLT 2.0 translator to convert Datalog$^+$ Deliberation RuleML 1.01 in XML format to an equivalent representation in a subset of the TPTP (Thousands of Problems for Theorem Provers) language.

1.1. Languages

Several languages are in the center of this project. RuleML and the TPTP language are respectively the source and the target languages of the translation. XSLT is the language that we use to implement the translator. Each language is introduced briefly in the following.

RuleML. RuleML (Rule Markup Language) is a markup language designed in an XML format. It is developed by an open non-profit organization of the same name, to provide a uniform representation for all kinds of relevant rule languages. As part of semantic technology efforts, the RuleML specification has become a de facto standard for Web rule knowledge representation. In both industry and academia, RuleML is used to bridge and exchange knowledge bases and queries among different systems[3], so the translation between RuleML and other rule-based languages is prevalent. The specification of RuleML pertinent to this project is Deliberation RuleML 1.01, which is the currently released Deliberation version. Deliberation RuleML 1.01 has broad coverage across various rule logics and has a modular schema system that allows fine-grained customization[5]. This project has its focus on Datalog$^+$ Deliberation RuleML 1.01, especially on the following three defining Datalog$^+$extensions[5], which can also be used in Hornlog$^+$:

- Existential Rules, where variables in rule conclusions are existentially quantified.
- Equality Rules, where the binary "Equal" predicate is applied in rule conclusions.
- Integrity Rules, which use the empty "Or" in rule conclusions to provide a convenient way for expressing falsity.

All these extensions have their equivalent representation in the target language of the translation in this project. The target language is a subset of the (full) first-order form (FOF) of the TPTP language.

The TPTP language. TPTP is a comprehensive library of automated theorem proving (ATP) test problems, which are available online. Its motivation is to provide support for the testing and evaluation of ATP systems[6]. The TPTP site also hosts online services for solving problems. These services are provided by the numerous individually developed ATP systems (or the TPTP systems) accessible from the TPTP portal. Standard input format is enforced by all the TPTP systems, whereby the TPTP language is defined. The input of a TPTP system in the form of the TPTP language is also called a TPTP problem, which consists of a number of TPTP formulae. A TPTP formula can be an "axiom", corresponding to the `<formula>` under `<Assert>` in RuleML, or a "conjecture", corresponding to the `<formula>` under `<Query>` in RuleML.

XSLT. XSLT (Extensible Stylesheet Language Transformations) Version 2.0 is a language to describe the ways to transform XML documents into other XML, HTML or plain text[7]. An XSLT document works as the input of an XSLT processor, together with the source XML file to be translated. The XSLT processor is typically an executable program. A new document is generated by the XSLT processor based on the content of the source XML and the definitions in the input XSLT document, which is called an XSLT translator in this report.

1.2. Objectives

As mentioned before, the task of this project is to implement an XSLT 2.0 translator to convert Datalog$^+$ Deliberation RuleML 1.01[1] knowledge bases[2] into the representation of the TPTP language, focusing on the three Datalog$^+$ extensions. As the main goal of this work, the translator should work correctly. In another sentence, the output of the translator should be accepted and solved by the TPTP systems without error, provided the input RuleML document is properly composed. The elements in the schema of Datalog$^+$ Deliberation RuleML 1.01 should be supported by the translation as much as possible, and any deviations should be documented.

Besides, we hope to render the output TPTP problems "pretty-printed", i.e., with proper

[1] In the following parts of this report the term "RuleML" is also used as a shortcut for "Datalog$^+$ Deliberation RuleML 1.01" when allowed by the context.

[2] Considered as the input of the translator, RuleML knowledge bases are called RuleML instances or RuleML documents in this report.

indentation, spacing and line breaking, to enhance the human readability. The following[3] is a sample formula in the TPTP language to illustrate the layout of pretty-printing:

```
fof(property_of_integer, axiom, (
    ! [X] :
      ( integer(X)
     => ( even(X)
        | odd(X) ) )
)).
```

The output TPTP formulae in this project always start with "`fof`". In the above TPTP formula, "`!`" stands for universal quantification, "`=>`" stands for implication, and "`|`" stands for disjunction. This formula means that if `X` is an integer, it is either even or odd (technically, because "`|`" is inclusive, `X` could be both even and odd).

Our last goal is to retain comments in the translation from RuleML to the TPTP language. RuleML uses XML comments and the TPTP language has its own line commenting syntax starting with a "%" (TPTP systems also support another syntax for block comments, however, we prefer to use line comments in this project). A comment in the TPTP language extends from the first "%" to the end of the line. Thus, the following RuleML comment

```
<!-- A RuleML comment
is also an XML comment.
-->
```

will be converted into the following form in the TPTP language:

```
% A RuleML comment
% is also an XML comment.
```

[3]The RuleML input rule for producing this TPTP formula will be given in section 2.1.

2. Methodology

The procedure of the translation in this project consists of two phases as illustrated in Figure 2.1. The first phase is to normalize the input RuleML document by another XSLT translator, which is called the normalizer. The implementation of the normalizer itself is not part of this project. In the second phase, the XSLT translator[1] implemented in the project is invoked to generate the output in the form of the TPTP document from the normalized form of the input RuleML document.

2.1. Normalization

In Deliberation RuleML 1.01 some elements can be reconstructed from other ones and thus can be omitted for brevity. The omittable elements are edge elements (with first letters in lowercase). In contrast, Node elements (with first letters in uppercase) cannot be omitted. The following[2] rule in RuleML is an example of the stripe-skipped form, i.e. without edge elements but with only Node elements:

```
<RuleML xmlns="http://ruleml.org/spec">
  <Assert>
    <Forall>
      <Var>x</Var>
      <Implies>
        <Atom>
          <Rel>integer</Rel>
          <Var>x</Var>
        </Atom>
        <Or>
          <Atom>
            <Rel>even</Rel>
```

[1] The term "XSLT translator" used in this report here and later on will be dedicated to denote the specific translator that is implemented in this project. The XSLT normalizer, which is also an XSLT translator in the general sense will always be referred to as the "normalizer".

[2] This rule is the RuleML input producing the TPTP formula in section 1.2.

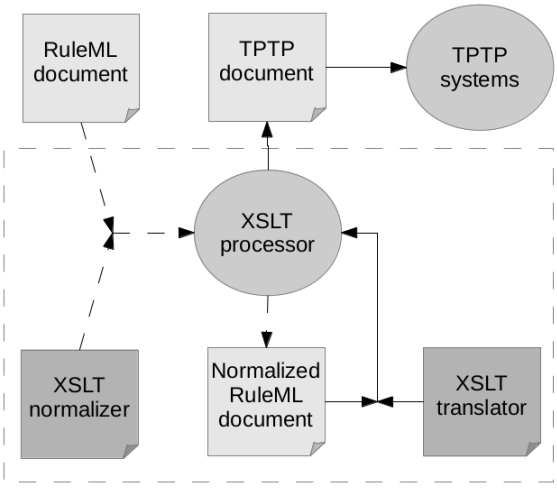

Figure 2.1.: The procedure of the translation. The dashed arrows indicate the process of normalization.

```
            <Var>x</Var>
          </Atom>
          <Atom>
            <Rel>odd</Rel>
            <Var>x</Var>
          </Atom>
        </Or>
      </Implies>
    </Forall>
  </Assert>
</RuleML>
```

For example, in the above rule the first subelement of `<Implies>`, i.e. `<Atom>`, constitutes the premise, and the next subelement, `<Or>`, is the conclusion. Their roles under `<Implies >` are determined by their positions. The fully-striped form of the above rule, i.e. with all edge elements in place (which will be the outcome of the normalizer used in this project) is as follows:

```
<RuleML xmlns="http://ruleml.org/spec">
  <act index="1">
    <Assert mapMaterial="yes" mapDirection="bidirectional">
      <formula>
        <Forall>
          <declare>
            <Var>x</Var>
          </declare>
          <formula>
            <Implies material="yes" direction="bidirectional">
              <if>
                <Atom>
                  <op><Rel>integer</Rel></op>
                  <arg index="1"><Var>x</Var></arg>
                </Atom>
              </if>
              <then>
                <Or>
                  <formula>
                    <Atom>
                      <op><Rel>even</Rel></op>
                      <arg index="1"><Var>x</Var></arg>
                    </Atom>
                  </formula>
                  <formula>
                    <Atom>
                      <op><Rel>odd</Rel></op>
                      <arg index="1"><Var>x</Var></arg>
                    </Atom>
                  </formula>
                </Or>
              </then>
            </Implies>
          </formula>
        </Forall>
      </formula>
```

```
    </Assert>
  </act>
</RuleML>
```

Note that the reconstructed elements, e.g. `<if>` and `<then>`, reveal roles explicitly without relying on their positions within `<Implies>`[3].

All in all, Datalog$^+$ allows freedom to some extent of skipping edges, defaulting attribute values, and reordering elements. When the translation from RuleML is performed, such freedom should be taken into account. However, a normalized RuleML document with all edge elements reconstructed and canonical ordering of elements established will significantly relieve the XSLT translator from dealing with positional semantics. This is the reason that the normalization phase performs preprocessing for the translation. Note that RuleML provides several XSLT normalizers which we could utilize. The normalizer that we adopt in this project is actually one[4] for a superset of Datalog$^+$ logic, however, it serves well for this project.

2.2. XSLT Translator

Taking advantage of RuleML normalization discussed in the previous section, sophisticated conditional branching between stripe-skipped and striped forms has been avoided in the implementation of the XSLT translator Hence we developed the translator following more closely to the "push style" of XSLT, as opposed to its "pull style". The pull style usually selects XML elements from the source document iteratively, and thus uses less XSLT templates. In contrast, the push style prefers to write more XSLT templates for the XML elements in the source document, and to apply the templates recursively. In general, the push style makes an XSLT document better structured, and it is the style that we adopted in this project.

In the Datalog$^+$ RuleML schema, most elements can be translated into the proper components of the TPTP language. The RuleML element for which the TPTP language has no equivalent counterpart is `<Retract>`, which will be ignored in the translation by not applying any XSLT template on it[5]. In addition, each TPTP formula has a unique name

[3]The reconstructed attributes, e.g. `@material` and `@direction`, will not be needed in our subsequent development.

[4]The normalizer document named 101_nafneghornlogeq_normalizer.xslt used in this project can be found at http://deliberation.ruleml.org/1.01/xslt/normalizer/, visited on December 3rd, 2014.

[5]It would possible to instead produce a warning or error message in the style of the XSLT-implemented Schematron [http://www.schematron.com], but this would cross the boundary between a translator

in the TPTP document, but rules in RuleML do not require names. We assume unnamed RuleML rules and use the following XSLT code to generate a unique name for every TPTP formula:

```
<!-- Formula name. -->
<xsl:value-of select="concat(
  $formula-source, '_in_act', $act-index, '_formula',
  count(preceding-sibling::r:formula) + 1)"/>
```

The above code is in the XSLT template matching the top level `<formula>` under `<Assert>` or `<Query>`. The string-type parameter `formula-source` of this template has the value of either "assert" or "query", depending on the parent of this `<formula>`. The parameter `act-index` has the same value as the attribute "index" (see the example of normalized RuleML in section 2.1) of the ancestor element `<act>`. The value (normally an integer) of this optional attribute should be unique for each `<act>` in the RuleML document, and can be reconstructed by the normalization phase if it is not provided in the original input RuleML document. The generated name takes the form of "assert_in_act<M>_formula<N>" or "query_in_act<M>_formula<N>", where <M> is substituted for the value of attribute "index" and <N> is substituted for by the sequential index of the current `<formula>`, starting from 1, among all the elements of `<formula>` at the same level.

Considering the scope of this project, any elements beyond Datalog+ expressivity will be ignored, and we will focus ourselves on RuleML elements `<Implies>`, `<Forall>`, `<Exists>`, and the falsity element `<Or/>`. Also only assertion rules, i.e., rules beginning with `<Assert>`, will be translated. `<Query>`s as well as other elements outside of the project scope of Datalog$^+$such as `<Fun>` and `<Expr>`, will be ignored in the translation by doing nothing on them. The only construct in the target TPTP language that cannot find a counterpart in the source RuleML instance is the name of each FOF formula. This can be worked around by generating unique IDs as formula names by employing the XSLT function `generate-id()`. However, the readability of the resulting TPTP formulae would be reduced in this way. In this project, we thus exploit the index property of the edge tag `<act>` to generate formula names, e.g., name "Assert2" for the rule with `<act index=''2''>`. Thus we can at least provide the correspondence between the source rules in normalized RuleML and the resulting formulae in the TPTP language.

In the development of the project, some RuleML instances will be dedicatedly composed as test cases to test the correctness of the translator, by running the results on a TPTP system. Examples from the Deliberation RuleML 1.01 website[5] will also be adapted to this project for testing purpose. These test cases should be kept well within the scope of this project as aforementioned. Meanwhile they should retain the integrity as a well-formed knowledge base, ready to be queried. A Relax NG validator will be employed

(our task) and a validator.

together with the Deliberation RuleML 1.01 schemas[5], to ensure that the RuleML test cases are syntactically valid.

In the next phase, we will consider the readability of the output document. White space will be properly reserved or inserted, e.g. lines will be properly broken, to generate a "pretty-printed" format for human readers. In addition, which was not required by the original project, we consider to incorporate XML comments of the source RuleML file into the output of the translation in proper commenting syntax. This will massively enhance the understandability of the output document and make the translator more practical.

3. Results

In the current version of RuleML2TPTP translator, the source RuleML will be normalized through 101_nafneghornlogeq_normalizer.xslt in the first step; then the translator will convert the standard RuleML into TPTP-FOF format and generate an output document. Here is an introductory example, showing the translation of an existential rule:

Original RueML Input:

```
<?xml version="1.0"?>
<RuleML xmlns="http://ruleml.org/spec">
 <Assert>
<Forall>
      <Var>H</Var>
      <Implies>
        <Atom>
          <Rel>human</Rel>
          <Var>H</Var>
        </Atom>
        <Exists>
          <Var>M</Var>
          <Atom>
            <Rel>hasMother</Rel>
            <Var>H</Var>
            <Var>M</Var>
          </Atom>
        </Exists>
      </Implies>
    </Forall>
  </Assert>
```

Normalized RuleML:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec">
  <act index="1">
    <Assert mapMaterial="yes" mapDirection="bidirectional">
      <formula>
        <Forall>
          <declare>
            <Var>H</Var>
          </declare>
          <formula>
            <Implies material="yes" direction="bidirectional">
              <Exists>
                <declare>
                  <Var>M</Var>
                </declare>
                <formula>
                  <Atom>
                    <op><Rel>hasMother</Rel></op>
                    <arg index="1"> <Var>H</Var> </arg>
                    <arg index="2"> <Var>M</Var> </arg>
                  </Atom>
                </formula>
              </Exists>
              <if>
                <Atom>
                  <op> <Rel>human</Rel> </op>
                  <arg index="1"> <Var>H</Var> </arg>
                </Atom>
              </if>
            </Implies>
          </formula>
        </Forall>
      </formula>
    </Assert>
  </act>

TPTP-FOF Output:

fof (exampleExistential, axiom, (
    ! [H] : ( human(H) => ? [M] : hasMother (H, M))
  )).
```

Another, complex example showing all kinds of Datalog+ rules can be found in the appendix.

4. Conclusion

In this project, we use XSLT 2.0 to develop a translator, RuleML2TPTP, which takes Datalog+ RuleML presentation syntax documents as input, and outputs equivalent TPTP-FOF documents. By using the translator, the input RuleML/XML format will be successfully translated into TPTP format documents, which can be executed by TPTP-accepting (FOL) provers such as Vampire. The TPTP output is well-formed and pretty-printed, as can be validated with Vampire. Major work we have done in this project includes:

1. Select a normalizer to normalize all types of Datalog+ RuleML input, and produce a standard format for the translation.

2. Use XSLT 2.0 to set up traslation templates in push style. The templates which we build collect information from the source Datalog+ RuleML, and output in TPTP format.

3. In order to make the TPTP output pretty-printed, add some clauses in the translator to improve the readability and appearance.

4. Output the result by generating a document and execute it through the online[1] Vampire reasoner.

5. Publish and maintain[2] the project on GitHub.

4.1. Future Work

Future work on the topics of this project includes:

- Compose RuleML2TPTP with translators targeting RuleML/XML.

[1] http://www.cs.miami.edu/ tptp/cgi-bin/SystemOnTPTP
[2] https://github.com/EdmonL/RuleML2TPTP

- Make it a "Save as ..." option of other RuleML tools (e.g., editors and engines).
- Extend the expressivity of RuleML input sublanguage from Datalog+ to Hornlog+.
- Further extensions could allow FOL RuleML input and even HOL RuleML.
- Consider an inverse TPTP2RuleML translator (with increasing expressivity subsets for the TPTP-to-RuleML direction).

Bibliography

[1] *GitHub*, https://github.com/

[2] *Jing*, http://www.thaiopensource.com/relaxng/jing.html

[3] *RuleML Home*, http://wiki.ruleml.org/index.php/RuleML_Home

[4] *Saxon*, http://saxon.sourceforge.net/

[5] *Specification of Deliberation RuleML 1.01*,
http://wiki.ruleml.org/index.php/Specification_of_Deliberation_RuleML_1.01

[6] *The TPTP Problem Library for Automated Theorem Proving*,
http://www.cs.miami.edu/~tptp/

[7] *XSL Transformations (XSLT) Version 2.0*, http://www.w3.org/TR/xslt20/

A. Translating the Business Scenario Example

The knowledge base of the instructive example accompanying[1] the Deliberation RuleML 1.01 release is translated here from RuleML/XML to TPTP. Since the RuleML input is partially striped, we do not show the fully striped intermediate form, but immediately give the TPTP output.

Original RuleML Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="http://deliberation.ruleml.org/1.01/relaxng/
   datalogplus_min_relaxed.rnc"?>
<RuleML xmlns="http://ruleml.org/spec">
  <!-- Some of these examples are from
    "A general Datalog-based framework for tractable query
       answering over
     ontologies",
    Andrea Calì,
    Georg Gottlob
    Thomas Lukasiewicz
    http://dx.doi.org/10.1016/j.websem.2012.03.001
        (preprint at http://www.websemanticsjournal.org/index.
           php/ps/article/view/289)
    -->

  <!-- This RuleML Document incrementally asserts into,
     retracts from and
       queries a rulebase (within the <RuleML> element) for a
          total of
       28 transactions: 13 Asserts, 2 Retracts and 13 Queries.
       Each Query demonstrates the semantics of the previous
          Assert or Retract
```

[1] http://deliberation.ruleml.org/1.01/exa/DatalogPlus/datalogplus_min.ruleml

```
     by providing (in the XML comments) the expected answer
        to the Query.
  -->
<Assert>
  <!--
       Equations are allowed as facts:
       William is an employee.
       "Bill" is an alias for "William".
       Sublanguage: datagroundfacteq
    -->
  <Atom>
    <Rel>employee</Rel>
    <Ind>William</Ind>
  </Atom>
  <Equal>
    <Ind>Bill</Ind>
    <Ind>William</Ind>
  </Equal>
</Assert>
<Query>
  <!--
       Who are the employees?
       Answers:
       x: <Ind>Bill</Ind>
       x: <Ind>William</Ind>
       Sublanguage: datalogeq
    -->
  <Atom>
    <Rel>employee</Rel>
    <Var>x</Var>
  </Atom>
</Query>

<Assert>
  <!--
       Equations may be universally quantified.
       This fact is a degenerate case, corresponding to the
          body of an implication being empty.
       The following makes the reflexive property of
          equality explicit (as built into most systems):
       Everything is equal to itself.
       Sublanguage: datalogeq
```

```
    -->
  <Forall>
    <Var>x</Var>
    <Equal>
      <Var>x</Var>
      <Var>x</Var>
    </Equal>
  </Forall>
</Assert>
<Query>
  <!--
       What is equal to itself?
       Answers:
       x: <Ind>Bill</Ind>
       x: <Ind>William</Ind>
       Sublanguage: datalogeq
    -->
  <Equal>
    <Var>x</Var>
    <Var>x</Var>
  </Equal>
</Query>

<Assert>
  <!--
       Non-ground facts are allowed.
       John is the CEO.
       John is responsible for everything.
       Sublanguage: datalogeq
    -->
  <Atom>
    <Rel>CEO</Rel>
    <Ind>John</Ind>
  </Atom>
  <Forall>
    <Var>x</Var>
    <Atom>
      <Rel>responsible_for</Rel>
      <Ind>John</Ind>
      <Var>x</Var>
    </Atom>
  </Forall>
```

```
</Assert>
<Query>
  <!--
       Is John responsible for Bill?
       Answers:
       Succeeds.
       Sublanguage: datalogeq
    -->
  <Atom>
    <Rel>responsible_for</Rel>
    <Ind>John</Ind>
    <Ind>Bill</Ind>
  </Atom>
</Query>

<Assert>
  <!--
       Rules are expressed as universally-quantified
          implications.
       A simple rule can be used to assert a subclass
          relationship:
       Every manager is an employee.
       Sublanguage: datalog
    -->
  <Forall>
    <Var>x</Var>
    <Implies>
      <if>
        <Atom>
          <Rel>manager</Rel>
          <Var>x</Var>
        </Atom>
      </if>
      <then>
        <Atom>
          <Rel>employee</Rel>
          <Var>x</Var>
        </Atom>
      </then>
    </Implies>
  </Forall>
  <Atom>
```

```
    <Rel>manger</Rel>
    <Ind>John</Ind>
  </Atom>
</Assert>
<Query>
  <!--
       Who are the employees?
       Answers:
       x: <Ind>Bill</Ind>
       x: <Ind>William</Ind>
       x: <Ind>John</Ind>
       Sublanguage: datalog
    -->
  <Atom>
    <Rel>employee</Rel>
    <Var>x</Var>
  </Atom>
</Query>

<Assert>
  <!--
       Equations are allowed in the body of (existential)
          implications.
       If anyone is the same as Margaret, then they
          supervise someone.
       (Note that this rule could be semantically simplified
       to an existential fact relying on axioms of equality
          .)
       Sublanguage: datalogexeq
    -->
  <Forall>
    <Var>x</Var>
    <Implies>
      <if>
        <Equal>
          <Ind>Margaret</Ind>
          <Var>x</Var>
        </Equal>
      </if>
      <then>
        <Exists>
          <Var>y</Var>
```

```
          <Atom>
            <Rel>supervises</Rel>
            <Var>x</Var>
            <Var>y</Var>
          </Atom>
        </Exists>
      </then>
    </Implies>
  </Forall>
</Assert>
<Query>
  <!--
       Does Margaret supervise someone?
       Answer:
       Succeeds
       Sublanguage: datalogeq
    -->
  <Exists>
    <Var>y</Var>
    <Atom>
      <Rel>supervises</Rel>
      <Ind>Margaret</Ind>
      <Var>y</Var>
    </Atom>
  </Exists>
</Query>

<Assert>
  <!--
       Pairwise Disjoint Classes
       Nothing is both an employee and a department.
       Sublanguage: ncdatalog
    -->
  <Forall>
    <Var>x</Var>
    <Implies>
      <if>
        <And>
          <Atom>
            <Rel>employee</Rel>
            <Var>x</Var>
          </Atom>
```

```
          <Atom>
            <Rel>department</Rel>
            <Var>x</Var>
          </Atom>
        </And>
      </if>
      <then>
        <Or/>
      </then>
    </Implies>
  </Forall>
</Assert>
<Assert>
  <!-- HR is an employee.
       HR is a department.
       These facts together with the previous rule create an
           inconsistency.
       Sublanguage: datalog
    -->
  <Atom>
    <Rel>employee</Rel>
    <Ind>HR</Ind>
  </Atom>
  <Atom>
    <Rel>department</Rel>
    <Ind>HR</Ind>
  </Atom>
</Assert>
<Query>
  <!--
       Is there any inconsistency?
       Succeeds, indicating inconsistency.
       Sublanguage: ncdatalog
    -->
  <Or/>
</Query>
<Retract>
  <!--
       Remove that HR is an employee.
       Sublanguage: datalog
  -->
  <Atom>
```

```
    <Rel>employee</Rel>
    <Ind>HR</Ind>
  </Atom>
</Retract>
<Query>
  <!--
       Is there any inconsistency?
       Fails, indicating consistency.
       Sublanguage: ncdatalog
    -->
  <Or/>
</Query>
<Assert>
  <!--
       Functionality Constraint:
       Everyone (or everything) has at most one supervisor.
       Sublanguage: datalogeq
    -->
  <Forall>
    <Var>x</Var>
    <Var>y</Var>
    <Var>z</Var>
    <Implies>
      <if>
        <And>
          <Atom>
            <Rel>supervises</Rel>
            <Var>x</Var>
            <Var>z</Var>
          </Atom>
          <Atom>
            <Rel>supervises</Rel>
            <Var>y</Var>
            <Var>z</Var>
          </Atom>
        </And>
      </if>
      <then>
        <Equal>
          <Var>x</Var>
          <Var>y</Var>
        </Equal>
```

```
      </then>
    </Implies>
  </Forall>
  <Atom>
    <Rel>supervises</Rel>
    <Ind>Margaret</Ind>
    <Ind>Bill</Ind>
  </Atom>
  <Atom>
    <Rel>supervises</Rel>
    <Ind>Peggy</Ind>
    <Ind>Bill</Ind>
  </Atom>
</Assert>
<Query>
  <!--
        Is Peggy the same as Margaret?
        Answer:
        Succeeds.
        Sublanguage: datalogeq
   -->
  <Equal>
    <Ind>Peggy</Ind>
    <Ind>Margaret</Ind>
  </Equal>
</Query>

<Assert>
  <!--
        Negative Constraints are allowed.
        No one (or no thing) is their own supervisor.
        Sublanguage: ncdatalog
    -->
  <Forall>
    <Var>x</Var>
    <Implies>
      <if>
        <Atom>
          <Rel>supervises</Rel>
          <Var>x</Var>
          <Var>x</Var>
        </Atom>
```

```
      </if>
      <then>
        <Or/>
      </then>
    </Implies>
  </Forall>
</Assert>
<Assert>
  <!--
        Margaret supervises Peggy.
        Sublanguage: datalog
  -->
  <Atom>
    <Rel>supervises</Rel>
    <Ind>Margaret</Ind>
    <Ind>Peggy</Ind>
  </Atom>
</Assert>
<Query>
  <!--
        Is there any inconsistency?
        Succeeds, indicating inconsistency.
        Sublanguage: ncdatalog
  -->
  <Or/>
</Query>
<Retract>
  <!--
        Remove that Margaret supervises Peggy.
        Sublanguage: datalog
  -->
  <Atom>
    <Rel>supervises</Rel>
    <Ind>Margaret</Ind>
    <Ind>Peggy</Ind>
  </Atom>
</Retract>
<Query>
  <!--
        Is there any inconsistency?
        Fails, indicating consistency.
        Sublanguage: ncdatalog
```

```
  -->
  <Or/>
</Query>
<Assert>
  <!--
        Equations may appear in the body of negative
           constraints.
        The simplest case is the assertion that two
           individuals
        are different (as built into systems making the
           unique name assumption).
        Sublanguage: ncdatalogeq

    -->
  <Implies>
    <if>
      <Equal>
        <Ind>Sue</Ind>
        <Ind>Maria</Ind>
      </Equal>
    </if>
    <then>
      <Or/>
    </then>
  </Implies>
</Assert>
<Query>
  <!--
        Is Sue the same as Maria?
        Answer:
        Fails.
        Sublanguage: datalogeq
    -->
  <Equal>
    <Ind>Sue</Ind>
    <Ind>Maria</Ind>
  </Equal>
</Query>

<Assert>
  <!--
        Existential (Head) Rules
```

```
        Every manager directs at least one department.
        Maria is a manager.
        Sublanguage: datalogex
    -->
  <Forall>
    <Var>M</Var>
    <Implies>
      <if>
        <Atom>
          <Rel>manager</Rel>
          <Var>M</Var>
        </Atom>
      </if>
      <then>
        <Exists>
          <Var>P</Var>
          <Atom>
            <Rel>directs</Rel>
            <Var>M</Var>
            <Var>P</Var>
          </Atom>
        </Exists>
      </then>
    </Implies>
  </Forall>
  <Atom>
    <Rel>manager</Rel>
    <Ind>Maria</Ind>
  </Atom>
</Assert>
<Query>
  <!--
        Does Maria direct a department?
        Answer:
        Succeeds.
        Sublanguage: datalog
    -->
  <Exists>
    <Var>P</Var>
    <Atom>
      <Rel>directs</Rel>
      <Ind>Maria</Ind>
```

```
      <Var>P</Var>
    </Atom>
  </Exists>
</Query>

<Assert>
  <!--
        The heads and bodies of existential rules can
           contain conjunctions.
        Every employee who directs a department is a manager
           , and supervises at
        least another employee who works in the same
           department.
        Sublanguage: datalogexcon
    -->
  <Forall>
    <Var>E</Var>
    <Var>P</Var>
    <Implies>
      <if>
        <And>
          <Atom>
            <Rel>employee</Rel>
            <Var>E</Var>
          </Atom>
          <Atom>
            <Rel>directs</Rel>
            <Var>E</Var>
            <Var>P</Var>
          </Atom>
        </And>
      </if>
      <then>
        <Exists>
          <Var>E'</Var>
          <And>
            <Atom>
              <Rel>manager</Rel>
              <Var>E</Var>
            </Atom>
            <Atom>
              <Rel>supervises</Rel>
```

```
                <Var>E</Var>
                <Var>E'</Var>
              </Atom>
              <Atom>
                <Rel>works_in</Rel>
                <Var>E'</Var>
                <Var>P</Var>
              </Atom>
            </And>
          </Exists>
        </then>
      </Implies>
    </Forall>
  </Assert>
  <Query>
    <!--
          Does Maria supervise someone?
          Answer:
          Succeeds.
          Sublanguage: datalog
      -->
    <Exists>
      <Var>E'</Var>
      <Atom>
        <Rel>supervises</Rel>
        <Ind>Maria</Ind>
        <Var>E'</Var>
      </Atom>
    </Exists>
  </Query>
</RuleML>
```

TPTP-FOF Output:

```
% Some of these examples are from
% "A general Datalog-based framework for tractable query
   answering
% over ontologies",
% Andrea Calì,
% Georg Gottlob
% Thomas Lukasiewicz
% http://dx.doi.org/10.1016/j.websem.2012.03.001
```

```
% (preprint at http://www.websemanticsjournal.org/index.php/ps
   /article/view/289)
% This RuleML Document incrementally asserts into, retracts
   from and
% queries a rulebase (within the <RuleML> element) for a total
    of
% 28 transactions: 13 Asserts, 2 Retracts and 13 Queries.
% Each Query demonstrates the semantics of the previous Assert
    or Retract
% by providing (in the XML comments) the expected answer to
   the Query.
% Equations are allowed as facts:
% William is an employee.
% "Bill" is an alias for "William".
% Sublanguage: datagroundfacteq
fof(assert_in_act1_formula1, axiom, (
    employee(william)
)).
fof(assert_in_act1_formula2, axiom, (
    bill = william
)).
% Who are the employees?
% Answers:
% x: <Ind>Bill</Ind>
% x: <Ind>William</Ind>
% Sublanguage: datalogeq
fof(query_in_act2_formula1, conjecture, (
    employee(X)
)).
% Equations may be universally quantified.
% This fact is a degenerate case, corresponding to the body of
    an implication being empty.
% The following makes the reflexive property of equality
   explicit (as built into most systems):
% Everything is equal to itself.
% Sublanguage: datalogeq
fof(assert_in_act3_formula1, axiom, (
    ! [X] : X = X
)).
% What is equal to itself?
% Answers:
% x: <Ind>Bill</Ind>
```

```
% x: <Ind>William</Ind>
% Sublanguage: datalogeq
fof(query_in_act4_formula1, conjecture, (
    X = X
)).
% Non-ground facts are allowed.
% John is the CEO.
% John is responsible for everything.
% Sublanguage: datalogeq
fof(assert_in_act5_formula1, axiom, (
    cEO(john)
)).
fof(assert_in_act5_formula2, axiom, (
    ! [X] : responsible_for(john, X)
)).
% Is John responsible for Bill?
% Answers:
% Succeeds.
% Sublanguage: datalogeq
fof(query_in_act6_formula1, conjecture, (
    responsible_for(john, bill)
)).
% Rules are expressed as universally-quantified implications.
% A simple rule can be used to assert a subclass relationship:
% Every manager is an employee.
% Sublanguage: datalog
fof(assert_in_act7_formula1, axiom, (
    ! [X] :
      ( manager(X)
     => employee(X) )
)).
fof(assert_in_act7_formula2, axiom, (
    manger(john)
)).
% Who are the employees?
% Answers:
% x: <Ind>Bill</Ind>
% x: <Ind>William</Ind>
% x: <Ind>John</Ind>
% Sublanguage: datalog
fof(query_in_act8_formula1, conjecture, (
    employee(X)
```

```
)).
% Equations are allowed in the body of (existential)
   implications.
% If anyone is the same as Margaret, then they supervise
   someone.
% (Note that this rule could be semantically simplified
% to an existential fact relying on axioms of equality.)
% Sublanguage: datalogexeq
fof(assert_in_act9_formula1, axiom, (
    ! [X] :
      ( margaret = X
     => ? [Y] : supervises(X, Y) )
)).
% Does Margaret supervise someone?
% Answer:
% Succeeds
% Sublanguage: datalogeq
fof(query_in_act10_formula1, conjecture, (
    ? [Y] : supervises(margaret, Y)
)).
% Pairwise Disjoint Classes
% Nothing is both an employee and a department.
% Sublanguage: ncdatalog
fof(assert_in_act11_formula1, axiom, (
    ! [X] :
      ( ( employee(X)
        & department(X) )
     => $false )
)).
% HR is an employee.
% HR is a department.
% These facts together with the previous rule create an
   inconsistency.
% Sublanguage: datalog
fof(assert_in_act12_formula1, axiom, (
    employee(hR)
)).
fof(assert_in_act12_formula2, axiom, (
    department(hR)
)).
% Is there any inconsistency?
% Succeeds, indicating inconsistency.
```

```
% Sublanguage: ncdatalog
fof(query_in_act13_formula1, conjecture, (
    $false
)).
% Is there any inconsistency?
% Fails, indicating consistency.
% Sublanguage: ncdatalog
fof(query_in_act15_formula1, conjecture, (
    $false
)).
% Functionality Constraint:
% Everyone (or everything) has at most one supervisor.
% Sublanguage: datalogeq
fof(assert_in_act16_formula1, axiom, (
    ! [X, Y, Z] :
      ( ( supervises(X, Z)
        & supervises(Y, Z) )
     => X = Y )
)).
fof(assert_in_act16_formula2, axiom, (
    supervises(margaret, bill)
)).
fof(assert_in_act16_formula3, axiom, (
    supervises(peggy, bill)
)).
% Is Peggy the same as Margaret?
% Answer:
% Succeeds.
% Sublanguage: datalogeq
fof(query_in_act17_formula1, conjecture, (
    peggy = margaret
)).
% Negative Constraints are allowed.
% No one (or no thing) is their own supervisor.
% Sublanguage: ncdatalog
fof(assert_in_act18_formula1, axiom, (
    ! [X] :
      ( supervises(X, X)
     => $false )
)).
% Margaret supervises Peggy.
% Sublanguage: datalog
```

```
fof(assert_in_act19_formula1, axiom, (
    supervises(margaret, peggy)
)).
% Is there any inconsistency?
% Succeeds, indicating inconsistency.
% Sublanguage: ncdatalog
fof(query_in_act20_formula1, conjecture, (
    $false
)).
% Is there any inconsistency?
% Fails, indicating consistency.
% Sublanguage: ncdatalog
fof(query_in_act22_formula1, conjecture, (
    $false
)).
% Equations may appear in the body of negative constraints.
% The simplest case is the assertion that two individuals
% are different (as built into systems making the unique name
   assumption).
% Sublanguage: ncdatalogeq

fof(assert_in_act23_formula1, axiom, (
    ( sue = maria
   => $false )
)).
% Is Sue the same as Maria?
% Answer:
% Fails.
% Sublanguage: datalogeq
fof(query_in_act24_formula1, conjecture, (
    sue = maria
)).
% Existential (Head) Rules
% Every manager directs at least one department.
% Maria is a manager.
% Sublanguage: datalogex
fof(assert_in_act25_formula1, axiom, (
    ! [M] :
      ( manager(M)
     => ? [P] : directs(M, P) )
)).
fof(assert_in_act25_formula2, axiom, (
```

```
    manager(maria)
)).
% Does Maria direct a department?
% Answer:
% Succeeds.
% Sublanguage: datalog
fof(query_in_act26_formula1, conjecture, (
    ? [P] : directs(maria, P)
)).
% The heads and bodies of existential rules can contain
   conjunctions.
% Every employee who directs a department is a manager, and
   supervises at
% least another employee who works in the same department.
% Sublanguage: datalogexcon
fof(assert_in_act27_formula1, axiom, (
    ! [E, P] :
      ( ( employee(E)
        & directs(E, P) )
     => ? [E'] :
          ( manager(E)
          & supervises(E, E')
          & works_in(E', P) ) )
)).
% Does Maria supervise someone?
% Answer:
% Succeeds.
% Sublanguage: datalog
fof(query_in_act28_formula1, conjecture, (
    ? [E'] : supervises(maria, E')
)).
```