DAML Rules—An RDF Query, Inference, and Transformation Language

Abstract

DAML Rules is a semantic markup language for Web resources. It builds on earlier W3C standards such as XML and RDF and provides a layered rule language for RDF-based languages. DAML Rules provides modeling primitives found in rule languages like F-Logic, Datalog or Prolog. The layers of DAML Rules are designed to work together with DAML+OIL. The language has a clean and well defined semantics. This document gives a systematic, compact and informal description of syntax and semantics of DAML Rules.

1 Introduction

DAML Rules is a layered rule language, aiming to support applications in need of RDF reasoning and transformation. Inference systems for higher level–languages like RDF Schema and DAML+OIL can be implemented (as much as the underlying logical formalism permits) are higher level layers of DAML Rules . DAML Rules provides as (human readable) EBNF syntax as well as an RDF-based syntax. In the following section we describe the EBNF-based features.

1.1 Features of DAML Rules

In the following sections, the main features of DAML Rules are informally described. Note that not all the features are available in DAML Rules layer 0.

Namespaces and Resources DAML Rules has special support for namespaces and resources identifier. Namespaces are declared via clause-like constructs of the form "*nsabbrev* := *namespace*.", e.g.

rdf := "http://www.w3.org/1999/02/22-rdf-syntax-ns#".

Resources are written as *nsabbrev:name*, where *nsabbrev* is a namespace abbreviation and *name* is the local name of the resource.

Resource abbreviations can be introduced analogously to namespace abbreviations, e.g.

$$\text{subClassOf} := \text{rdfs:subClassOf}.$$

Statements and Molecules An RDF statement (triple) is—inspired by F-Logic object syntax—written as

$$subject[predicate \rightarrow object]$$

Several statements with the same subject can be abbreviated as "molecules":

$$pat[hasAge \rightarrow \text{"}42\text{"}; isMarried \rightarrow \text{"}yes\text{"}; \ldots]$$

One RDF statement (triple) can be written inside another eg.:

$$stefan[marriedTo \rightarrow birgit[hasAge \rightarrow \text{"}32\text{"}]]$$

Models RDF models, i.e., sets of statements, are made explicit in DAML Rules("first class citizens"). Statements from a specific model are written as *stmt@model*, where *stmt* is a statement (or molecule) and *model* is a model (i.e., a resource denoting a model), e.g.

$$michael[hasAge \rightarrow \text{"33"}]@factsAboutDFKI$$

DAML Rules also allows skolem functions as model specifications (which is useful for parameterized models). Skolem function can be used to derive a new model from a set of existing models when used in rules., e.g.

$$O[P \rightarrow Q]@sf(m1, X, Y)$$

If all (or many) statements/molecules in a formula (see section 1.1) are from one model, the following abbreviation can be used: @*model formula*. All statements/molecules in *formula* without an explicit model specification are implicitely suffixed with @*model*.

Instead of constants, variables and skolem functions also boolean combinations can be used, eg.: $(model_1 \cap model_2)$ specifying the intersection of two models, $(model_1 \cup model_2)$ specifying the union of two models, and $(model_1 \backslash model_2)$ specifying the set-difference of two models.

Reified Statements Reified statements are written as: $<statement>$ can be used inside other statements, eg.:

$$Stefan[beliefs \twoheadrightarrow <Ora[isAuthorOf \rightarrow homepage]>]$$

Path Expressions For navigation purposes path expressions have proven to be very useful in object oriented languages. DAML Rules allows the usable of path expressions instead of subject, predicate or object definitions. Path expressions are dot-delimited sequence of resources., e.g. eg.:

$$Stefan.spouse.mother$$

denotes Stefan's mother in law.

Logical Formulae DAML Rules uses the usual set of connectives and quantifiers for building formulae from RDF molecules and normal logic predicates, i.e., $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, etc.[1] All variables must be introduced via quantifiers, therefore marking them is not necessary (i.e., DAML Rules does not require variables to start with an uppercase letter as in PROLOG).

Clauses A DAML Rules clause is either a fact or a rule. Rule heads may only contain conjunctions and molecules and must not contain (explicitly or implicitly) any disjunctive or negated expressions.

1.2 Example: RDF Schema

The following example shows how rules axomatising (part of the) semantics of RDF Schema look like in DAML Rules. The rules can be used together with a horn logic based inference engine like XSB to derive additional knowledge from an RDF Schema specification.

The first lines define namespaces (for RDF and RDF Schema) and abbreviations (for type, subPropertyOf and subClassOf).

The rules are enclosed by a model specification block

```
FORALL Mdl @rdfschema(Mdl)
```

The skolem function $rdfschema(Mdl)$ is the model identifier of all facts derived by the rules enclosed by the model specification block. Mdl is variable, the parameter Mdl denotes the RDF Schema specification. The model

[1] For DAML Rules programs in plain ASCII syntax, the symbols AND, OR, NOT, FORALL, EXISTS, etc. are used; see the example.

$rdfschema(Mdl)$ contains the all statements from the model Mdl plus everything derived additionally by the rules. The first rule

```
FORALL O,P,V   O[P->V] <- O[P->V]@Mdl.
```

specifies that every triple contained in the model Mdl is also element of the model with the identifier $rdfschema(Mdl)$. The next rule defines the transitivity of the subclass relationship of RDF Schema. The remaining rules define the semantics of the subproperty and type property.

```
rdf := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
rdfs := 'http://www.w3.org/TR/1999/PR-rdf-schema-19990303#'.
type := rdf:type.
subPropertyOf := rdfs:subPropertyOf.
subClassOf := rdfs:subClassOf.

FORALL Mdl @rdfschema(Mdl) {
  FORALL O,P,V   O[P->V] <-
      O[P->V]@Mdl.

  FORALL O,P,V   O[subClassOf->V] <-
    EXISTS W   (O[subClassOf->W] AND W[subClassOf->V]).

  FORALL O,P,V   O[P->V] <-
    EXISTS S   S[subPropertyOf->P] AND O[S->V].

  FORALL O,P,V   O[subPropertyOf->V] <-
    EXISTS W   (O[subPropertyOf->W] AND W[subPropertyOf->V]).

  FORALL O,T   O[type->T] <-
    EXISTS S   (S[subClassOf->T] AND O[type->S]).
}
```

2 DAML Rules Core: Layer 0: RDF Syntax

In this section, we describe how to represent the DAML Rules_0 in RDF, where DAML Rules_0 denotes the subset of DAML Rules without quantifiers and negation.

2.1 Namespace for DAML Rules in RDF

In the following, 'damlrules' denotes the DAML Rules namespace (something like 'http://www.daml.org/2001/06/30/damlrules#').

2.2 Abbreviations

Abbreviations for namespaces and resources are not necessary: we simply use the XML namespace and entity declarations.

2.3 Triples, Molecules, Path Expresions

a[b->c] becomes an instance of damlrules:Triple which is a subclass of rdf:Statement:

```
<damlrules:Triple>
  <damlrules:subject rdf:resource="#a"/>
  <damlrules:predicate rdf:resource="#b"/>
  <damlrules:object rdf:resource="#c"/>
</damlrules:Triple>
```

There is no need for an RDF representation of abbreviations like a[b->c; p->q; ...] since such molecules are equivalent to the conjunction of single Triples. The same holds for path expressions (which can be split into separate Triples).

2.4 Associated Models, Model Expressions

Every Triple can have an *associated* model: a[b->c]@m becomes

```
<damlrules:Triple>
  <damlrules:subject rdf:resource="#a"/>
  <damlrules:predicate rdf:resource="#b"/>
  <damlrules:object rdf:resource="#c"/>
  <damlrules:model rdf:resource="#m"/>
</damlrules:Triple>
```

Note that triple:model is a property that can be used on all formulas and clauses, not only on Triples (see the section on @-Expressions below). Any term can function as a model; complex model expressions can be built with triple:ModelUnion, triple:ModelIntersection etc., e.g.:

```
<damlrules:ModelUnion>
  <damlrules:firstModel rdf:resource="#m"/>
  <damlrules:secondModel rdf:resource="#n"/>
</damlrules:ModelUnion>
```

Furhtermore, a triple model can be denoted by a Skolem function to allow parameterized models (triple:SkolemModel).

2.5 Terms

damlrules:Term comprises rdfs:Literal, damlrules:Variable, damlrules:Structure, damlrules:Resource, damlrules:ReifiedTriple, damlrules:Model etc.

2.6 Atoms and Formulas

We have two sorts of Atoms: damlrules:Triple and damlrules:HornAtom, where HornAtoms are the normal Horn atoms like p(a,X).

Since we do not support Lloyd-Topor transformations in DAML Rules_0, Atom and And/Or formulas are the only formulas.

2.7 Clauses

A triple:Clause simply consists of a head (with range triple:Atom) and a body (with range triple:Formula), both of which may be empty to form facts and queries. It can also have an associated model (see below).

2.8 @-Expressions

All forms of @-expressions from the ASCII representation of DAML Rules are mapped to usages of the triple:model property, even for the { } enclosed blocks, e.g.

```
@someModel {
  clause1.
  clause2.
}
```

becomes

```
<damlrules:Clause rdf:ID"clause1">
  ...
  <damlrules:model rdf:resource="#someModel"/>
</damlrules:Clause>

<damlrules:Clause rdf:ID"clause2">
  ...
  <damlrules:model rdf:resource="#someModel"/>
</damlrules:Clause>
```

3 Semantic Characterizations of DAML Rules

This section provides two semantic characterizations of DAML Rules.

1. An indirect semantic characterization by defining a mapping to Horn Logic. The mapping gives also hints how DAML Rules can be implemented using existing available inference engines.

2. A direct semantic characterization in terms of an alternate model theory.

3.1 An Indirect Semantic Characterization

$$
\begin{array}{rcll}
X & \longrightarrow & X \quad \text{for literals and symbols } X & (1) \\
A:N & \longrightarrow & \text{resource}(A,N) & (2) \\
O[P \rightarrow V] & \longrightarrow & \text{statement}(O,P,V) & (3) \\
S@M & \longrightarrow & \text{true}(S,M) \quad \text{for statements } S & (4) \\
<S> & \longrightarrow & S \quad \text{for statements } S & (5) \\
O[P_1 \rightarrow V_1; P_2 \rightarrow V_2; \ldots]@M & \longrightarrow & O[P_1 \rightarrow V_1]@M \ \wedge & (6) \\
 & & O[P_2 \rightarrow V_2]@M \ \wedge \ \ldots & \\
\text{true}(S, M_1 \cap M_2) & \longrightarrow & \text{true}(S,M_1) \ \wedge \ \text{true}(S,M_2) & (7) \\
\text{true}(S, M_1 \backslash M_2) & \longrightarrow & \text{true}(S,M_1) \ \wedge \ \neg\, \text{true}(S,M_2) & (8) \\
X \ := \ Y.\ S(X) & \longrightarrow & \forall\, X \quad (X = Y \ \wedge \ S(X)) & (9) \\
 & & \text{for clause sequences } S(X) &
\end{array}
$$

Figure 1: The RDF-specific rewrite rules

In this section, the semantics of DAML Rules is given—as a first approximation—by a mapping to Horn logic. This allows DAML Rules to be implemented on top of XSB (i.e., PROLOG with tabled resolution), analogously to the F-Logic Flora [Ludscher *et al.*, 1999].

Figure 1 shows the rewrite rules for mapping RDF-specific features like resources and statements. All other mappings are wellknown (Lloyd-Topor transformations for handling of quantifier [Lloyd and Topor, 1984]) or straightforward (see the SiLRI system [Decker *et al.*, 1998]).

Example:

p:jdow[p:lastname$\rightarrow$doe]@$m1$. $\longrightarrow$

```
true(statement(resource(p, jdow), resource(p, lastname), doe),
m1).
```

3.2 An Direct Semantic Characterization

In a future document, a model-theoretic sematics based on minimal Herbrand models and fix-point operators will be provided.

References

[Decker *et al.*, 1998] S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for rdf. In *In QL'98 - The Query Languages Workshop*, Boston, USA, 1998. WorldWideWeb Consortium (W3C).

[Lloyd and Topor, 1984] J. Lloyd and R. Topor. Making prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[Ludscher *et al.*, 1999] B. Ludscher, G. Yang, and M. Kifer. FLORA: The secret of object-oriented logic programming. Technical report, SUNY at Stony Brook, 1999.

A BNF for DAML Rules

```
          TripleDeclaration::= ( DefaultModelDeclaration )+ <EOF>
    DefaultModelDeclaration::= ( Name ":=" ID_Term ";" | "{" (
                               DefaultModelDeclaration )+ "}" | (
                               <FORALL> VarNameList )? <AT>
                               SimpleModelExpression "{" (
                               DefaultModelDeclaration )+ "}" | Query ";"
                               | DoubleRule ";" | Rule ";" )
   IntersectModelExpression::= UnionModelExpression ( <INTERSECT>
                               IntersectModelExpression )*
       UnionModelExpression::= SetDifferenceModelExpression ( <UNION>
                               IntersectModelExpression )*
SetDifferenceModelExpression::= SimpleModelExpression ( <DIFF>
                               IntersectModelExpression )*
      SimpleModelExpression::= ( "(" IntersectModelExpression ")" |
                               ID_Term )
                      Query::= ( <FORALL> VarNameList )? ( <IMPLIEDBY> |
```

```
                              "?-" ) Formula
                   Rule::= ( <FORALL> VarNameList )? ConjExpression (
                           ( <IMPLIEDBY> | ":-" ) Formula )?
             DoubleRule::= ( <FORALL> VarNameList )? ConjExpression
                           <EQUIV> ConjExpression
         ConjExpression::= ( <NEG> )? ConjFinal ( ( <AND> | "," )
                           ConjExpression )*
              ConjFinal::= ( GraphMolecule | PredicateMolecule | (
                           "(" ConjExpression ")" ) )
                Formula::= ( ( <NOT> )? ( <EXISTS> | <FORALL> )
                           VarNameList Formula | <AT>
                           SimpleModelExpression Formula |
                           ImpliesFormula )
         ImpliesFormula::= ( ImpliedFormula ) ( <IMPLIES> Formula )*
         ImpliedFormula::= ( EquivFormula ) ( <IMPLIEDBY> Formula )*
           EquivFormula::= ( OrFormula ) ( <EQUIV> Formula )*
              OrFormula::= AndFormula ( <OR> Formula )*
             AndFormula::= NegOtherFormula ( ( <AND> | "," ) Formula
                           )*
        NegOtherFormula::= ( <NOT> )? OtherFormula
           OtherFormula::= ( EvaluableExpression | GraphMolecule |
                           PredicateMolecule | "(" Formula ")" )
    EvaluableExpression::= ( Name | IntegerNumber | FloatNumber )
                           <IS> ArithmeticExpression
   ArithmeticExpression::= MultiplicativeExpression ( ( <PLUS> |
                           <MINUS> ) MultiplicativeExpression )*
MultiplicativeExpression::= UnaryExpression ( ( <STAR> | <SLASH> |
                           <REM> ) UnaryExpression )*
        UnaryExpression::= ( ( <PLUS> | <MINUS> ) UnaryExpression |
                           UnaryExpressionNotPlusMinus )
UnaryExpressionNotPlusMinus::= ( UnaryFunction | BinaryFunction | Name |
                           IntegerNumber | FloatNumber |
                           StringLiteral | <LPAREN>
                           ArithmeticExpression <RPAREN> )
          UnaryFunction::= ( <UNOP> ) ArithmeticExpression <RPAREN>
         BinaryFunction::= <BINOP> ArithmeticExpression <COMMA>
                           ArithmeticExpression <RPAREN>
            VarNameList::= Name ( <COMMA> Name )*
  GraphMoleculeLookahead::= Reference "["
PredicateMoleculeLookahead::= Name ( "(" )?
```

```
       GraphMolecule::= Reference Specification
   PredicateMolecule::= Name ( "(" ListOfPaths ")" )?
 AdressableResource::= ( "(" Path ")" | "<" GraphMolecule ">" |
                      ID_Term )
           Reference::= AdressableResource ( ( Specification )? (
                      "." ) ArcApplication )*
                Path::= AdressableResource ( Specification )? ( (
                      "." ) ArcApplication ( Specification )? )*
      ArcApplication::= AdressableResource
           ArcResult::= "->" ( Path | "{" ( ListOfPaths )? "}" )
       Specification::= "[" ( ListOfArcs )? "]" ( <AT>
                      SimpleModelExpression )?
                 Arc::= ArcApplication ArcResult
          ListOfArcs::= Arc ( ";" Arc )*
         ListOfPaths::= Path ( "," Path )*
 NamespaceIdentifier::= Name ( ":" Name )?
             ID_Term::= ( ID_TermFunction | ID_TermList |
                      ID_TermSingle )
     ID_TermFunction::= ( ( Name <LPAREN> ( ListOfPaths )?
                      <RPAREN> ) | ( <BINOP> ( ListOfPaths )?
                      <RPAREN> ) | ( <UNOP> ( ListOfPaths )?
                      <RPAREN> ) )
         ID_TermList::= "[" ( ListOfPaths ( <BIT_OR> Path )? )?
                      "]"
       ID_TermSingle::= ( NamespaceIdentifier | IntegerNumber |
                      FloatNumber | StringLiteral )
                Name::= ( <IDENTIFIER> | <STRING_LITERAL> )
       IntegerNumber::= ( ( ( <PLUS> | <MINUS> ) <INTEGER_LITERAL>
                      ) | ( <INTEGER_LITERAL> ) )
         FloatNumber::= ( ( ( <PLUS> | <MINUS> )
                      <FLOATING_POINT_LITERAL> ) | (
                      <FLOATING_POINT_LITERAL> ) )
      IntegerLiteral::= <INTEGER_LITERAL>
        FloatLiteral::= <FLOATING_POINT_LITERAL>
       StringLiteral::= <STRING_LITERAL>
```

B RDF Schema for DAML Rules

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE rdf:RDF [
 <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
 <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
 <!ENTITY damlrules 'http://www.daml.org/2001/06/30/rules#'>
]>
<rdf:RDF xmlns:rdf="&rdf;"
 xmlns:rdfs="&rdfs;"
 xmlns:damlrules="&damlrules;"
 xmlns="&damlrules;">


<!-- Proposal for damlrules0 in RDF       Michael Sintek    2001-07-03 -->


<!-- damlrules -->

<rdfs:Class rdf:ID="Triple">
  <rdfs:comment>
     a Triple is a statement with an (optionally) associated model expression
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="&rdf;Statement"/>
  <rdfs:subClassOf rdf:resource="&damlrules;Atom"/>
</rdfs:Class>

<rdf:Property rdf:ID="subject">
  <rdfs:subPropertyOf rdf:resource="&rdf;subject"/>
  <rdfs:domain rdf:resource="&damlrules;Triple"/>
  <rdfs:range rdf:resource="&damlrules;Term"/>
</rdf:Property>

<rdf:Property rdf:ID="predicate">
  <rdfs:subPropertyOf rdf:resource="&rdf;predicate"/>
  <rdfs:domain rdf:resource="&damlrules;Triple"/>
  <rdfs:range rdf:resource="&damlrules;Term"/>
</rdf:Property>

<rdf:Property rdf:ID="object">
  <rdfs:subPropertyOf rdf:resource="&rdf;object"/>
  <rdfs:domain rdf:resource="&damlrules;Triple"/>
  <rdfs:range rdf:resource="&damlrules;Term"/>
</rdf:Property>

<!-- a Triple also can have an associated model via damlrules:model -->
```

```
<!-- Model -->

<rdfs:Class rdf:ID="Model">
  <rdfs:subClassOf rdf:resource="&damlrules;Term"/>
</rdfs:Class>

<rdfs:Class rdf:ID="SimpelModel">
  <rdfs:subClassOf rdf:resource="&damlrules;Model"/>
  <!-- rdf:label is used to name a model -->
</rdfs:Class>

<rdfs:Class rdf:ID="SkolemModel">
  <rdfs:subClassOf rdf:resource="&damlrules;Model"/>
</rdfs:Class>

<rdf:Property rdf:ID="skolemFunction">
  <rdfs:domain rdf:resource="&damlrules;SkolemModel"/>
  <rdfs:range rdf:resource="&damlrules;Structure"/>
</rdf:Property>

<rdfs:Class rdf:ID="BinaryModelExpression">
  <rdfs:subClassOf rdf:resource="&damlrules;Model"/>
</rdfs:Class>

<rdf:Property rdf:ID="firstModel">
  <rdfs:domain rdf:resource="&damlrules;BinaryModelExpression"/>
  <rdfs:range rdf:resource="&damlrules;Model"/>  <!-- Term ? -->
</rdf:Property>

<rdf:Property rdf:ID="secondModel">
  <rdfs:domain rdf:resource="&damlrules;BinaryModelExpression"/>
  <rdfs:range rdf:resource="&damlrules;Model"/>
</rdf:Property>

<rdfs:Class rdf:ID="ModelIntersection">
  <rdfs:subClassOf rdf:resource="&damlrules;BinaryModelExpression"/>
</rdfs:Class>

<rdfs:Class rdf:ID="ModelUnion">
  <rdfs:subClassOf rdf:resource="&damlrules;BinaryModelExpression"/>
</rdfs:Class>

<rdfs:Class rdf:ID="ModelDifference">
  <rdfs:subClassOf rdf:resource="&damlrules;BinaryModelExpression"/>
```

```
</rdfs:Class>


<!-- Term -->

<rdfs:Class rdf:ID="Term"/>

<rdfs:Class rdf:ID="Variable">
  <rdfs:subClassOf rdf:resource="&damlrules;Term"/>
  <!-- rdf:label is used to name a variable -->
  <!-- or: damlrules:label as subPropertyOf rdf:label ??? -->
</rdfs:Class>

<Description rdf:about="&rdfs;Literal">
  <!-- can we simply make rdfs:Literal a subClassOf Term; or do
       we need something like damlrules:Symbol or damlrules:Literal ??? -->
  <rdfs:subClassOf rdf:resource="&damlrules;Term"/>
</Description>

<rdfs:Class rdf:ID="Resource">
  <rdfs:subClassOf rdf:resource="&damlrules;Term"/>
  <!-- needed to allow any resource to be a term -->
</rdfs:Class>

<rdfs:Class rdf:ID="ReifiedTriple">
  <rdfs:subClassOf rdf:resource="&damlrules;Term"/>
</rdfs:Class>

<rdf:Property rdf:ID="triple">
  <rdfs:domain rdf:resource="&damlrules;ReifiedTriple"/>
  <rdfs:range rdf:resource="&damlrules;Triple"/>
</rdf:Property>

<rdfs:Class rdf:ID="Structure">
  <rdfs:subClassOf rdf:resource="&damlrules;Term"/>
</rdfs:Class>

<rdf:Property rdf:ID="functor">
  <rdfs:domain rdf:resource="&damlrules;Structure"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>

<rdf:Property rdf:ID="args">
  <rdfs:domain rdf:resource="&damlrules;Structure"/>
  <rdfs:range rdf:resource="&damlrules;TermSeq"/>
```

```
</rdf:Property>

<rdfs:Class rdf:ID="TermSeq">
  <!-- I'm not comfortable with untyped sequences, so we should
       decide how to represent a term sequence; rdf:Seq or
       daml:List do not seem to be appropriate
  -->
  <rdfs:subClassOf rdf:resource="&rdf;Seq"/>
</rdfs:Class>

<!-- other Terms ... -->


<!-- Formula -->

<rdfs:Class rdf:ID="Formula"/>

<!-- Atom is a Formula, see its definition -->

<rdf:Property rdf:ID="model">
  <rdfs:domain rdf:resource="&damlrules;Clause"/>
  <rdfs:domain rdf:resource="&damlrules;Formula"/>
  <rdfs:range rdf:resource="&damlrules;Term"/>
    <!-- or damlrules:Model plus damlrules:Variable? -->
</rdf:Property>

<rdfs:Class rdf:ID="BinaryFormula">
  <rdfs:subClassOf rdf:resource="&damlrules;Formula"/>
</rdfs:Class>

<rdf:Property rdf:ID="firstFormula">
  <rdfs:domain rdf:resource="&damlrules;BinaryFormula"/>
  <rdfs:range rdf:resource="&damlrules;Formula"/>
</rdf:Property>

<rdf:Property rdf:ID="secondFormula">
  <rdfs:domain rdf:resource="&damlrules;BinaryFormula"/>
  <rdfs:range rdf:resource="&damlrules;Formula"/>
</rdf:Property>

<rdfs:Class rdf:ID="And">
  <rdfs:subClassOf rdf:resource="&damlrules;BinaryFormula"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Or">
```

```
  <rdfs:subClassOf rdf:resource="&damlrules;BinaryFormula"/>
</rdfs:Class>

<rdfs:Class rdf:ID="UnaryFormula">
  <rdfs:subClassOf rdf:resource="&damlrules;Formula"/>
</rdfs:Class>

<rdf:Property rdf:ID="formula">
  <rdfs:domain rdf:resource="&damlrules;UnaryFormula"/>
  <rdfs:range rdf:resource="&damlrules;Formula"/>
</rdf:Property>

<!-- Atom -->

<rdfs:Class rdf:ID="Atom">
  <rdfs:subClassOf rdf:resource="&damlrules;Formula"/>
</rdfs:Class>

<!-- Triple is an Atom, see its definition -->

<rdfs:Class rdf:ID="HornAtom">
  <rdfs:subClassOf rdf:resource="&damlrules;Atom"/>
</rdfs:Class>

<rdf:Property rdf:ID="predicateSymbol">
  <rdfs:domain rdf:resource="&damlrules;HornAtom"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>

<rdf:Property rdf:about="#args">
  <rdfs:domain rdf:resource="&damlrules;HornAtom"/>
</rdf:Property>


<!-- Clause -->

<rdfs:Class rdf:ID="Clause"/>

<rdf:Property rdf:ID="head">
  <rdfs:domain rdf:resource="&damlrules;Clause"/>
  <rdfs:range rdf:resource="&damlrules;Atom"/>
</rdf:Property>

<rdf:Property rdf:ID="body">
  <rdfs:domain rdf:resource="&damlrules;Clause"/>
```

```
  <rdfs:range rdf:resource="&damlrules;Formula"/>
</rdf:Property>

<!-- a Clause can (optionally) have an associated model, see
     the model Property definition
-->


</rdf:RDF>
}
```