Lecture Notes in Computer Science 2876
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Michael Schroeder Gerd Wagner (Eds.)

Rules and Rule Markup Languages for the Semantic Web

Second International Workshop, RuleML 2003
Sanibel Island, FL, USA, October 20, 2003
Proceedings

Springer

Volume Editors

Michael Schroeder
Technical University of Dresden, Faculty of Informatik
01062 Dresden, Germany
E-mail: msch@soi.city.ac.uk

Gerd Wagner
Eindhoven University of Technology
Faculty of Technology Management
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: G.Wagner@tm.tue.nl

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): H.4, H.3, C.2, I.2, H.5, K.4, F.3

ISSN 0302-9743
ISBN 3-540-20361-3 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10968123 06/3142 5 4 3 2 1 0

Preface

RuleML 2003 was the second international workshop on rules and rule markup languages for the Semantic Web, held in conjunction with the International Semantic Web Conference (ISWC). The aim of the RuleML workshop series is to stimulate research on all issues related to *web rule languages* and to provide an annual forum for presenting and discussing new research results.

The Semantic Web is a major world-wide endeavor to advance the Web by enriching its multimedia document content with propositional information that can be processed by inference-enabled Web applications. Rules and rule markup languages, such as RuleML, will play an important role in the success of the Semantic Web. Rules will act as a means to draw inferences, to express constraints, to specify policies for reacting to events, to transform data, etc. Rule markup languages will allow us to enrich Web ontologies by adding definitions of derived concepts, to publish rules on the Web, to exchange rules between different systems and tools, etc.

RuleML 2003 built on the success of RuleML 2002, which was held in conjunction with ISWC 2002, Sardinia, Italy. The proceedings of RuleML 2002 can be found at http://www.ceur-ws.org/Vol-60/.

Special highlights of the RuleML 2003 workshop were the two invited presentations given by Peter Chen on "Rules, XML, and the ER Model" and by Harold Boley on "Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms". This proceedings volume also contains an invited paper by François Bry and Sebastian Schaffert on "An Entailment Relation for Reasoning on the Web".

RuleML 2003 was held on Sanibel Island, Florida, USA, in conjunction with ISWC 2003. We would like to thank the organizers of ISWC 2003 for hosting our workshop. Special thanks go to Enigmatec Corporation, London, for sponsoring RuleML 2003. Finally, we thank all the colleagues that submitted their papers to RuleML 2003 and that contributed to the presentations and discussions.

August 2003

Michael Schroeder
Gerd Wagner

Organization

Steering Commitee

Harold Boley, CA
Mike Dean, USA
Benjamin Grosof, USA
Steve Ross-Talbot, UK
Michael Schroeder, UK
Bruce E. Spencer, CA
Said Tabet, USA
Gerd Wagner, NL

Program Commitee

Grigoris Antoniou, University of Bremen, DE
Harold Boley, University of New Brunswick, CA
Francois Bry, Ludwig-Maximilians-University Munich, DE
Carlos Damasio, New University of Lisbon, PT
Mike Dean, BBN Technologies / Verizon, USA
Stefan Decker, Information Science Institute, USA
Andreas Eberhart, International University, DE
Jerome Euzenat, INRIA Rhone-Alpes, Grenoble, FR
Benjamin Grosof, MIT, USA
Jan Maluszynski, Linkoeping University, SE
Massimo Marchiori, W3C, MIT, USA and University of Venice, IT
Donald Nute, University of Georgia, USA
Steve Ross-Talbot, Enigmatec, UK
Michael Schroeder, City University, London, UK
Bruce Spencer, University of New Brunswick, CA
Said Tabet, USA
Gerd Wagner, Eindhoven University of Technology, NL

Sponsors

Enigmatec Corporation, London, UK

Table of Contents

Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms 1
Harold Boley

An Entailment Relation for Reasoning on the Web 17
François Bry and Sebastian Schaffert

A Rule-Based XML Access Control Model 35
Chutiporn Anutariya, Somchai Chatvichienchai, Mizuho Iwiahara, Vilas Wuwongse, and Yahiko Kambayashi

Inference of Reactive Rules from Dependency Models 49
Asaf Adi, Opher Etzion, Dagan Gilat, and Guy Sharon

Value-Added Metatagging: Ontology and Rule Based Methods for Smarter Metadata 65
Marek Hatala and Griff Richards

Constructing RuleML-Based Domain Theories on Top of OWL Ontologies 81
Christopher J. Matheus, Mitch M. Kokar, Kenneth Baclawski, and Jerzy Letkowski

Inheritance and Rules in Object-Oriented Semantic Web Languages 95
Guizhen Yang and Michael Kifer

Rules and Defeasible Reasoning on the Semantic Web 111
Grigoris Antoniou and Gerd Wagner

Inference Queues for Communicating and Monitoring Declarative Information between Web Services 121
Bruce Spencer and Sandy Liu

Combining an Inference Engine with Databases: A Rule Server 136
Tanel Tammet and Vello Kadarpik

Extraction of Structured Rules from Web Pages and Maintenance of Mutual Consistency: XRML Approach 150
Juyoung Kang and Jae Kyu Lee

RuleML Annotation for Automatic Collection Levels on METS Metadata 164
Chieko Nakabasami

Author Index 173

Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms*

Harold Boley

Institute for Information Technology – e-Business
National Research Council of Canada
Fredericton, NB, E3B 9W4, Canada
Harold.Boley@nrc-cnrc.gc.ca

Abstract. This paper describes an Object-Oriented extension to RuleML as a modular combination of three sublanguages. (1) User-level roles provide frame-like slot representations as unordered argument collections in atoms and complex terms. (2) URI-grounded clauses allow for 'webizing' using URIs as object identifiers for facts and rules. (3) Order-sorted terms permit typed variables via Web links into taxonomies such as RDF Schema class hierarchies, thus reusing the Semantic Web's light-weight ontologies. Besides introducing the first sublanguage with the Positional-Roled (ASCII) syntax, all three sublanguages are introduced with the OO RuleML (XML) syntax. Their semantics are sketched and their implementation paths are discussed.

1 Introduction

RuleML started in 2000 with XML-encoded positional-argument rules, and in 2002 we have introduced frame-like knowledge representation (KR) with user-level role-filler slots as unordered arguments. Since 2001 RuleML has permitted a kind of 'webizing' to allow RDF-like [LS99] KR, and RuleML 0.8 has used URIs as optional additions to, or substitutes for, individual constants as well as relation and function symbols; in the following, URI grounding will also permit URIs within clause (`fact` and `imp`) and `rulebase` labels. Finally, since 2002,

* Thanks to Michael Schroeder and Gerd Wagner for inviting me to give this RuleML'03 presentation. I also want to express my gratitude to Michael Sintek and Said Tabet for valuable contributions on several topics of this paper. Said Tabet, Benjamin Grosof, and the RuleML Steering Committee have encouraged me early on regarding the OO RuleML design. Bruce Spencer has supported the development of OO RuleML, its implementations, as well as the PR rule language. Marcel Ball and Stephen Greene gave valuable hints and performed various OO RuleML and PR syntax implementations. OO RuleML has already been employed outside the RuleML team by Virendra Bhavsar (AgentMatcher project) and Daniel Lemire (RACOFI project); further applications are being planned, e.g. by Anna Maclachlan (Metaxtract project). This research was funded by NRC as part of the Sifter project.

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 1–16, 2003.

we have begun work on URI-based taxonomy access for order-sorted terms; this broadens the scope of webizing by typing, e.g., logic variables of RuleML via Semantic Web taxonomies such as RDF Schema (RDFS) class hierarchies.

This paper describes Object-Oriented RuleML (OO RuleML) conceived as the orthogonal combination of user-level roles, URI grounding, and order-sortedness. These orthogonal dimensions constitute three declarative OO sublanguages that can be visualized as the edges of an 'OO cube', i.e. they can be used independently from each other or can be freely combined:

The **OO contribution** of

1. **User-level roles** is to allow 'object-centered' sets of role-filler slots – much like the role-type slots of classes and role-value slots of their instances; because of the unorderedness of slot sets, the inheritance of slots will be easier than that of ordered argument sequences.
2. **URI grounding** is the provision of URIs as unique object identifiers (OIDs) for facts – much like instances – and for rules – much like methods.
3. **Order-sortedness** is making taxonomies available as declarative inheritance pathways for term typing – much like class hierarchies.

Since an ordered argument sequence can be augmented by user-level roles, the first dimension permits three choices. The other two dimensions just permit 'yes'/'no' distinctions for an entire rulebase, but even in the 'yes' case some of its clauses may be not URI-grounded or not order-sorted, since these are optional features. Because of the dimensions' mutual independence, $3 \times 2 \times 2 = 12$ modular combinations will thus be principally possible in the OO cube. Following the design rationale of RuleML 0.8 [BTW01,Bol02], these sublanguage combinations can be reflected by OO RuleML's lattice of XML DTDs/Schemas.

The sublanguages cover clauses that can be either facts or rules, where rules may be used for defining declarative methods. However, OO RuleML currently only captures the above declarative aspects of OO, not procedural aspects such as the updating of instance slots. We also omit here the possible fourth independent OO sublanguage of **signature-instantiated clauses**, which would allow the assertion of '`new`' instances[1].

2 Object Centering via User-Level Roles

Since the beginnings of knowledge representation (KR), there have been two paradigms in this field, which will be called here *position-keyed* and *role-keyed* KR. These differ in the two natural focus points and argument-access methods of elementary representations. In predicate-centered or position-keyed KR (pKR), one predicate or relation symbol is focused, and applied to positionally

[1] As part of intertranslating between positional arguments and user-level roles there is an experimental XSLT implementation of signatures [http://www.ruleml.org/indoo]. TRIPLE [SD02] has permitted the '`new`less' generation of URI-grounded facts in rule heads.

ordered objects as arguments. In object-centered or role-keyed KR (rKR), one object identifier is focused, and associated via property roles, unordered, with other objects as arguments. Elementary representations in pKR have directly employed atomic formulas (atoms) of first-order logic, or added some syntactic sugar. Elementary representations in rKR were inspired by (Lisp) property lists and directed labeled graphs (Semantic Nets), but later have also been developed as subsets of first-order logic (Description Logic, F-logic). The more expressive representations of pKR (e.g., derivation rules) and of rKR (e.g., method definitions) maintain the different focus points. Syntactic and semantic pKR-rKR blending has been attempted, and will be demonstrated here.

In the Web, versions of both paradigms surfaced again. A kind of pKR came back with XML, because its *parent elements* are focus points 'applied to' its ordered child elements. On the other hand, a kind of rKR came back with RDF, since its *descriptions* focus a resource that has properties associating it, unordered, with other objects. Finding a common data model as the basis of Web KR has thus become a foundational issue for the Semantic Web.

In RuleML 0.8, we have used a pKR-rKR-unifying data model that generalizes the data models of both XML and RDF to express clauses (facts and rules). It is based on differentiating *type* and *role* elements in XML, where role tags (distinguished by a leading underscore) accommodate RDF properties. However, in RuleML 0.8 only *system* roles are permitted, their names cannot be taken from the application domain, and the atoms within clauses are still predicate-oriented. A pKR example will illustrate this 'system-level' solution. For this and later examples of this section we will use the Positional-Roled (PR) syntax [http://www.ruleml.org/submission/ruleml-shortation.html]. Consider a ternary `offer` relation applied to ordered arguments for the offer name, category, and price. In the PR syntax an `offer` of an 'Ecobile' can be categorized as 'special' and priced at $20000 via the following fact (Prolog-like, except that a capitalized symbol like `Ecobile` denotes an individual constant, not a variable):

```
offer(Ecobile,special,20000).
```

In RuleML 0.8 this has been marked up as follows (the `_rlab` role provides clause labels as `ind` types here and later on):

```
<fact>
  <_rlab><ind>pKR fact 1</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>offer</rel></_opr>
      <ind>Ecobile</ind>
      <ind>special</ind>
      <ind>20000</ind>
    </atom>
  </_head>
</fact>
```

Notice that the `fact` type has a `_head` role associating it with an `atom` type. The `atom`, however, uses a role, `_opr`, only for its operator association with the `rel`(ation) type. The three arguments of type `ind`(ividual) are immediate `atom` children ordered in the spirit of XML and pKR. Thus, while the `_opr` role can be moved from the prefix position to a postfix position without changing its meaning, the `ind` types are semantically attached to their relative positions. This fact representation thus requires users and applications (e.g., XSLT) to 'store' the correct interpretation of the three arguments separately, and any extension by additional arguments requires changes to these positional interpretations, except when new arguments are always added at the (right) end only.

A 'user-level' solution in the spirit of RDF and rKR thus is to introduce (user-level) roles `name`, `category`, and `price` for the arguments. Our `offer` can then be represented in the PR syntax as follows (inspired by F-logic [KL89]):

```
offer(name->Ecobile;category->special;price->20000).
```

In OO RuleML this can be marked up thus:

```
<fact>
  <_rlab><ind>rKR fact 1</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>offer</rel></_opr>
      <_r n="name"><ind>Ecobile</ind></_r>
      <_r n="category"><ind>special</ind></_r>
      <_r n="price"><ind>20000</ind></_r>
    </atom>
  </_head>
</fact>
```

Actually, a single (system-level) metarole `_r` is employed here with different (user-level) values of the XML attribute `n`(ame). The XML DTDs/Schemas of RuleML thus only require a small change to introduce rKR for RuleML's atomic formulas [http://www.ruleml.org/indoo]. The correct interpretation of the three arguments is no longer position-dependent and additional arguments such as `expiry` and `region` can be added without affecting any existing interpretation.

Now suppose we wish to keep the earlier pKR for the first three (mandatory) arguments, but use rKR for two extra (optional) arguments. For this, we can employ a blend of one ordered sequence of arguments plus a role-keyed set of arguments before and/or after:

```
offer(Ecobile,
      special,
      20000;
      expiry->2003-12-31;
      region->North America).
```

```
<fact>
  <_rlab><ind>prKR fact 1</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>offer</rel></_opr>
      <ind>Ecobile</ind>
      <ind>special</ind>
      <ind>20000</ind>
      <_r n="expiry"><ind>2003-12-31</ind></_r>
      <_r n="region"><ind>North America</ind></_r>
    </atom>
  </_head>
</fact>
```

Since the XML DTDs/Schemas of OO RuleML naturally extend those of RuleML 0.8 by optional sets of roles around the possibly empty argument sequence, such pKR-rKR blends are implicitly taken care of.

As an example of a binary rKR relation, the PR syntax and OO RuleML can represent customer Peter Miller's gold status thus:

```
customer(name->Peter Miller;status->gold).

<fact>
  <_rlab><ind>rKR fact 2</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>customer</rel></_opr>
      <_r n="name"><ind>Peter Miller</ind></_r>
      <_r n="status"><ind>gold</ind></_r>
    </atom>
  </_head>
</fact>
```

Notice that the meaning of such unqualified roles is *local* to their atoms: within `offer` objects, `name` refers to offer names; within `customer` atoms, to customer names. Again, further roles could be easily added.

Variables can be introduced in all of these versions. In the PR syntax, variables are prefixed by a "`?`"; in OO RuleML, they use `var` type tags instead of `ind` markup. This, then, permits the representation of rules for both pKR and rKR.

As an example, here is an rKR version of a `discount` rule in PR syntax (the `price` role with the anonymous filler variable "`_`" requires the presence of price information but does not use it – without that requirement an anonymous rest-role variable could have been employed instead):

```
discount(offer name->?off;
         customer name->?cust;
         awarded amount->10)
   :-
offer(name->?off;
      category->special;
      price->_),
customer(name->?cust;
         status->gold).
```

In OO RuleML this can be marked up as follows:

```
<imp>
  <_rlab><ind>rKR rule 1</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>discount</rel></_opr>
      <_r n="offer name"><var>off</var></_r>
      <_r n="customer name"><var>cust</var></_r>
      <_r n="awarded amount"><ind>10</ind></_r>
    </atom>
  </_head>
  <_body>
    <and>
      <atom>
        <_opr><rel>offer</rel></_opr>
        <_r n="name"><var>off</var></_r>
        <_r n="category"><ind>special</ind></_r>
        <_r n="price"><var/></_r>
      </atom>
      <atom>
        <_opr><rel>customer</rel></_opr>
        <_r n="name"><var>cust</var></_r>
        <_r n="status"><ind>gold</ind></_r>
      </atom>
    </and>
  </_body>
</imp>
```

Notice that this `imp`(lication) rule derives a `discount`-centered `atom` in its head from the following conjunction in its body: An `offer`-centered query asks for `category` = `special` (but masks the specific `price` with an anonymous variable) and a `customer`-centered query asks for `status` = `gold`. The `off` and `cust` variable bindings obtained under `name` in the body are differentiated as `offer name` and `customer name` in the head and a 10% discount is awarded. Using rKR fact 1 and rKR fact 2, rKR rule 1 derives this amount for an Ecobile purchase by Peter Miller.

The **semantics** of rKR's clause sets with user-level roles can be defined by explaining how pKR's (here, LPs [Llo87]) notions of clause instantiation and ground equality (for the model-theoretic semantics) as well as unification (for the proof-theoretic semantics) should be extended.

Since rKR role names are assumed here to be non-variable symbols, *rKR instantiation* can recursively walk through the fillers of user-level roles, substituting dereferenced values from the substitution (environment) for any variables encountered.

Since OO RuleML uses explicit rest variables, *rKR ground equality* can recursively compare two clauses after lexicographic sorting – w.r.t. the role names – of the role-filler slots of atoms and complex terms.

Since OO RuleML uses at most one rest variable per atom or complex term, *rKR unification* can perform sorting as in the above ground equality, use the above rKR instantiation of variables, and otherwise proceed left-to-right as for pKR unification, pairing up identical roles before recursively unifying their fillers.

The **implementation** of OO RuleML for rKR has been done both via an XSLT translator to positional RuleML and via an extension of the Java-based jDREW interpreter [Spe02]. This has already been used for representing product-seaking/advertising trees in the tree-similarity-based AgentMatcher system [BBY03] and for expressing music filtering rules in the collaborative e-learning system RACOFI [ABB+03].

3 URI Grounding of Clauses

Our previous rKR clauses did not use object identifiers (OIDs) for providing each object with an (object) identity. RDF has introduced a new flavor of OIDs for describing resources via their URIs[2]. This style of KR, here called URI-grounded KR (gKR), is also possible in OO RuleML by permitting a `wid` (web `id`) attribute within the `ind` type of an `_rlab` (`r`ule `lab`el) or – not further detailed here – of an entire `_rbaselab` (`r`ule`base` `lab`el); this is complemented by a `widref` (web `id` `ref`erence) attribute within the `ind` type of a referring slot filler; so, `wid` and `widref` are dual like XML's `id` and `idref` and RDF's `about` and `resource`[3].

The previous rKR fact 1 can thus be URI-grounded such that it is specialized to grKR fact 1 for *the* Ecobile occurring as offer 37 in a certain catalog as shown below. Similarly, the `ind` type of `Ecobile` can have a `widref` to another grounded rKR fact.

[2] While RDF was originally mostly intended for 'metadata' describing 'data' at URI resources, it has meanwhile been taken up for general KR where – even besides RDF's "blank nodes" – none of these URIs need link to 'data'. But still RDF's OIDs are 'about' objects, not (URI) addresses of objects: Usually several RDF descriptions are collected in a document that itself can be located at a single URI unrelated to the URIs being described.

[3] With OIDs, circular references become possible in OO modeling, RDF, OO RuleML, and XML, but these can often be statically detected to avoid (indirectly) self-referential clauses and the well-known logical paradoxes. Our semantics will perform a graph search with loop detection for obtaining a closure of objects.

The other grounded rKR fact, grKR fact 3, uses a different kind of URI attribute within an `ind`: `href` refers to a 'home page' characterizing the individual (e.g., `gas`). Generally, while `widref` presupposes that a description about the URI exist but not that the URI (currently) exist, `href` presupposes that the URI (currently) exist but not that a description about the URI exist.

Finally, also *global* user roles can be constructed as Qualified Names (QNames) whose qualifier is a namespace prefix, e.g. `s` or `t`, which is associated with a URI in the namespace declaration of the `rulebase` type that surrounds all RuleML clauses. Fragment identifiers ("`#`") are employed to point into the URI-addressed document.

```
<ruleml:rulebase
 xmlns:ruleml="http://www.ruleml.org/dtd/0.83/ruleml-oodatalog.dtd"
 xmlns:s="http://offercore.org/offerproperties#"
 xmlns:t="http://productcore.org/productproperties#">
 <fact>
  <_rlab><ind wid="http://catalist.ca/37">grKR fact 1</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>offer</rel></_opr>
      <_r n="s:name"><ind widref="http://ecobile.com">Ecobile</ind></_r>
      <_r n="s:category"><ind>special</ind></_r>
      <_r n="s:price"><ind>20000</ind></_r>
    </atom>
  </_head>
 </fact>
 <fact>
  <_rlab><ind wid="http://ecobile.com">grKR fact 3</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>product</rel></_opr>
      <_r n="t:name"><ind">Ecobile SX</ind></_r>
      <_r n="t:fuel"><ind href="http://naturalgas.org">gas</ind></_r>
      <_r n="t:horsepower"><ind>90</ind></_r>
      <_r n="t:displacement"><ind>1550</ind></_r>
    </atom>
  </_head>
 </fact>
</ruleml:rulebase>
```

The use of QNames in an attribute value such as the above `s:name` in `n="s:name"` has been discussed in a recent W3C TAG Finding [http://www.w3.org/2001/tag/doc/qnameids-2002-06-04].

These OO RuleML facts – except for their optional labels (e.g., grKR fact 1), their explicit relation names (e.g., `offer`)[4], and their both named and grounded

[4] RDF's anonymous grounded descriptions can be represented by substituting a named relation markup such as `<rel>offer</rel>` with an anonymous one such as `<rel/>`, keeping the `_rlab`-embedded grounding `<ind wid="http://catalist.ca/37">`.

arguments (e.g., `Ecobile` on http://ecobile.com)[5] – correspond to the following RDF descriptions:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://offercore.org/offerproperties#"
  xmlns:t="http://productcore.org/productproperties#">
  <rdf:Description about="http://catalist.ca/37">
    <s:name rdf:resource="http://ecobile.com"/>
    <s:category>special</s:category>
    <s:price>20000</s:price>
  </rdf:Description>
  <rdf:Description about="http://ecobile.com">
    <t:name>Ecobile SX</t:name>
    <t:fuel>gas</t:fuel>
    <t:horsepower>90</t:horsepower>
    <t:displacement>1550</t:displacement>
  </rdf:Description>
</rdf:RDF>
```

Like RDF properties, OO RuleML roles can thus be based on RDF Schema definitions (e.g., `subPropertyOf`) at the documents pointed to by their qualifier URIs; the next section will further expand on this kind of RDF-RuleML interoperation (for OO RuleML sorts, using `subClassOf`). However, the following is also important when comparing RDF and OO RuleML:

1. RDF's domain-specific properties, e.g. `s:name`, `s:category`, and `s:price`, are used as XML elements, so RDF serializations cannot be given a generic DTD/Schema; OO RuleML's corresponding metarole `_r` contains the domain-specific roles only as XML attribute values, so is amenable to a generic DTD/Schema (actually, this is a simple extension to RuleML's generic DTD/Schema [http://www.ruleml.org/indoo]).
2. For (ground) facts, OO RuleML representations contain little more information than their RDF counterparts: RDF diagrams can be regarded as the minimal basic data model; OO RuleML could thus act as an alternate RDF serialization amenable to DTD/Schema validation, hence to embedding into other valid XML (including XHTML) documents.
3. Once RDF facts are captured in OO RuleML, RDF rules (and queries) over them are directly available as well: they can be taken from OO RuleML's system of sublanguages.
4. Issues and solutions can be transferred between RDF and RuleML already now (as started in [http://lists.w3.org/Archives/Public/www-rdf-rules]): some candidates are the semantic and implementation issues raised below.

[5] RDF's grounded-only objects used as slot fillers can again be represented by substituting a named, grounded argument markup such as `<ind widref="http://ecobile.com">Ecobile</ind>` with a grounded-only one such as `<ind widref="http://ecobile.com"/>`.

URI grounding is orthogonal to user-level roles. Actually, grounding has been used for pKR in RuleML 0.8. For example, our initial pKR fact 1 can be URI-grounded as follows:

```
<ruleml:rulebase
  xmlns:ruleml="http://www.ruleml.org/dtd/0.83/ruleml-oodatalog.dtd"
  <fact>
   <_rlab><ind wid="http://catalist.ca/37">gpKR fact 1</ind></_rlab>
   <_head>
     <atom>
       <_opr><rel>offer</rel></_opr>
       <ind widref="http://ecobile.com">Ecobile</ind>
       <ind>special</ind>
       <ind>20000</ind>
     </atom>
   </_head>
  </fact>
 </ruleml:rulebase>
```

The XML DTDs/Schemas of RuleML only require a small change to introduce gKR by allowing `wid`/`widref` attributes on RuleML's `ind` and `cterm` elements. Their validation could be defined similarly to XML's `id`/`idref` validation (after URI normalization), requiring unique `wid` values and one `wid` value to exist for every `widref` value of a local document[6].

The **semantics** of gKR's URI grounding in OO RuleML can be divided into three parts.

First, URI strings are often processed by rules for expansion, redirection, etc. (by a "canonicalization algorithm" [http://www.ietf.org/rfc/rfc2396.txt] [BLFM98]) before their referenced 'Web objects' (e.g., Web documents) can be retrieved or they turn out to be 'broken links'. Hence, it appears natural to check for semantic URI equality using a notion of string rewriting [http://www.loria.fr/~vigneron/RewritingHP]. We regard two URIs, which may be syntactically different, as semantically equal iff they are processed or rewritten to the same (normal form) URI just before both link to a (namely, the same) Web object or both lead to a broken link error. A *URI normal form* is a URI string that cannot be rewritten any further but either directly refers to a Web object or directly leads to a broken link error. For a gKR rulebase B we consider – at any given time t – a *URI rewriting system* $(s(B), R)$ over the finite set $s(B)$ of URIs used for the grounding of B. The rewriting relation R contains URI

[6] For *distributed* grKR, where several documents contribute to describe the same URI, the situation becomes similar to RDF's `about`/`resource`: `wid` values will not be globally unique and one or more facts with the same `wid` value can exist in separate documents for every `widref` value. The unorderedness of rKR would permit to 'materialize' all distributed '`wid`-equal' facts into a single virtual one on demand, restoring the original centralized situation. For distributed gpKR the orderedness of pKR would prevent this virtual 're-centralization'.

expansion rules such as for extending certain URIs by "/", "`index.html`", etc. R also contains redirection rules for replacing entire URIs by other URIs. For the grounding semantics the URI rewriting system must be *convergent* (terminating and confluent), i.e. unique normal forms must exist. Testing the syntactic equality of these normal forms can then be used to check for the semantic equality of any pair of URIs used in grounding. This semantic equality will be needed in the unification of URI-grounded terms as well as in the other parts below.

Second, a `wid` attribute within the `_rlab` of clauses or within the `_rbaselab` of rulebases semantically labels these elements with the normal form of the URI string of the attribute's value. On the other hand, `widref` attributes inside a clause semantically initiate the following graph search for the *clause closure* in the current document[7]: Retrieve the clause or rulebase having the same `wid` normal form label as exhibited by a `widref` attribute's value; recursively continue retrieval with all the `widref` attributes of all the clauses retrieved directly or within rulebases – just ignoring duplicates resulting from circular references – until a fixpoint is reached. The URI grounding semantics of the original `widref`-attributed clause then is the clause closure of all these retrieval results (conversely, for circular references, elements of this set can contain the current clause in the set of *their* grounding retrieval results). The URI grounding semantics of a rulebase is the union of the clause closures of all its clauses.

Third, an `href` attribute inside a clause semantically – at any given time t – makes the normal form of the attribute's URI value link to the semantics of the Web object or, for a broken link, causes it to denote an error object[8]. If the Web object linked to is another gKR rulebase, *its* semantics can be obtained as described in the current section; similarly, for the semantics of sections 2 and 4; further Semantic Web objects (e.g., in OWL [DS03]) could be covered as well.

Since the first part of this gKR semantics augments equality and unification, it can be used to extend the pKR and rKR semantics of section 2. The second part, constructing an additional clause closure, can be modularly added to the pKR/rKR semantics. Similarly, the semantics of the third part is only linked to, hence completely decoupled from, the pKR/rKR semantics.

An **implementation** of OO RuleML for gKR has not been attempted yet, mainly because it depends on an improved specification of semantic URI equality rules [http://www.w3.org/2001/tag/ilist#URIEquivalence-15] for the URI rewriting system.

[7] Although for non-distributed KR `wid` and `widref` act (internally) within a document only, the rewriting described in the first part is still performed (externally) using Web normal forms of the URIs. This will simplify the maintenance of large gKR documents and ensure later interoperability of gKR documents distributed over different namespaces.

[8] Currently, `href` and `type` (see next section) are the only OO RuleML attributes whose meaning links into the Web, but several related attributes of this kind could be distinguished.

4 Term Typing via Order-Sorted Taxonomies

Terms, in particular variables, in the previous pKR/rKR and gKR clauses are still untyped using *unsorted* KR. We proceed here to *order-sorted* KR (sKR) that is based on a special treatment of sort predicates and sorted individuals, variables, etc. in clauses. With sort restrictions directly attached to variables, hence usable during unification, proofs can be kept at a more abstract level, thus reducing the search space. An independently defined sort hierarchy, e.g. in RDFS (using `subClassOf`) or OWL, can be employed as the taxonomy that constitutes the partial order of the resulting order-sorted logic. We have developed a webized construct for linking RuleML variables to such externally defined sort hierarchies of the Semantic Web.

The way sorted RuleML variables link to RDFS classes is the following:

- Basically, the class hierarchy of an order-sorted logic – e.g. in RDFS – can be accessed from RuleML in a similar way as it can from RDF.
- RDF's use of `rdf:type` for taxonomic RDFS typing of individuals/resources is transferred to RuleML's typing of `ind`s.
- Additionally, we propose a new RDFS use: to access unchanged RDFS for typing RuleML variables, noting that the RDFS taxonomy must then be cycle-free (and, if we use an OWL taxonomy, it must also be consistent).

The technical construct is again based on namespace declarations using fragment identifiers ("`#`") to point into the RDFS document containing the *class* definition to be used as a type. This `#`*class* is assumed to exist there, usually with one (or more) `subClassOf` relations defining (multiple) inheritance. An `ind` or `var` is then typed via a `type` attribute augmenting the namespace prefix by the *class* name. Our earlier rKR rule 1 can thus be typed as follows (with *class* `Offer` and *class* `Customer`):

```
<ruleml:rulebase
 xmlns:ruleml="http://www.ruleml.org/dtd/0.83/ruleml-oodatalog.dtd"
 xmlns:t="http://distribcore.org/distribclasses#"
 xmlns:u="http://customercore.org/custclasses#">
 <imp>
  <_rlab><ind>rsKR rule 1</ind></_rlab>
  <_head>
    <atom>
      <_opr><rel>discount</rel></_opr>
      <_r n="offer name"><var type="t:Offer">off</var></_r>
      <_r n="customer name"><var type="u:Customer">cust</var></_r>
      <_r n="awarded amount"><ind>10</ind></_r>
    </atom>
  </_head>
```

```
  <_body>
    <and>
      <atom>
        <_opr><rel>offer</rel></_opr>
        <_r n="name"><var type="t:Offer">off</var></_r>
        <_r n="category"><ind>special</ind></_r>
        <_r n="price"><var/></_r>
      </atom>
      <atom>
        <_opr><rel>customer</rel></_opr>
        <_r n="name"><var type="u:Customer">cust</var></_r>
        <_r n="status"><ind>gold</ind></_r>
      </atom>
    </and>
  </_body>
 </imp>
</ruleml:rulebase>
```

Again, the use of QNames in an attribute value such as the above `t:Offer` in `type="t:Offer"` has been recently discussed in [http://www.w3.org/2001/tag/doc/qnameids-2002-06-04].

Here we assume that the URI `http://distribcore.org/distribclasses` links to an RDFS document containing a definition of `Offer`, e.g. specifying it as a subclass of `Distribution`. Similarly, for the URI `http://customercore.org/custclasses`. Sorted unification of two typed variables can then employ the RDFS sort hierarchy to find the greatest lower bound (glb) of their types, failing if it does not exist. For example, suppose `Sale` is defined as a subclass of both `Offer` and `Promotion`, another subclass of `Distribution`.

Based on this, the OO RuleML query

```
<ruleml:query
  xmlns:ruleml="http://www.ruleml.org/dtd/0.83/ruleml-oodatalog.dtd"
  xmlns:v="http://distribcore.org/distribclasses#">
  <_rlab><ind>rKR query 1</ind></_rlab>
  <_body>
    <atom>
      <_opr><rel>discount</rel></_opr>
      <_r n="offer name"><var type="v:Promotion">prom</var></_r>
      <_r n="customer name"><ind>Peter Miller</ind></_r>
      <_r n="awarded amount"><var>Rebate</var></_r>
    </atom>
  </_body>
</ruleml:query>
```

will unify with the head of the above defined rsKR rule 1 by binding `<var type="v:Sale">prom</var>` to `<var type="t:Sale">off</var>`, where

`Sale` is found in the RDFS document as the glb of `Offer` and `Promotion`, and also binding `<var type="u:Customer">cust</var>` to `<ind>Peter Miller</ind>` and `<var>Rebate</var>` to `<ind>10</ind>`.

Besides the above combination of sKR with rKR, sKR can also be combined with gKR, hence with grKR, and furthermore with the pKR versions.

The XML DTDs/Schemas of RuleML only require the following change for sKR: the introduction of a `type` attribute on `ind`, `var`, and `cterm` elements.

The **semantics** of sKR could be given directly but can also be reduced to unsorted KR in a well-known manner discussed here for sorted terms that are variables. All occurrences of a sorted variable are replaced by their unsorted counterparts plus a body-side application of a sort-corresponding unary predicate to that variable (sorted facts thus become unsorted rules). Moreover, the definition of the unary predicate reflects the subsumption relations of the sort taxonomy via rules.

Independently of using this reduction, the sKR semantics can be combined with the pKR/rKR and gKR semantics of sections 2 and 3 in a modular fashion.

The **implementation** of sKR has been performed directly (without the above reduction) for various sorted Prolog pKR systems before RDFS became available as a Web-based taxonomy language. We plan to adapt sorted indexing techniques for Prolog to RDFS and to the Java-based implementation of the Fredericton OO jDREW interpreter for OO RuleML.

5 Conclusions

Object-Orientation in OO RuleML currently comprises object-centered user-level roles, object identifiers for URI-grounded clauses, and class hierarchies over order-sorted terms. While these OO sublanguages can be used as three independent extensions to RuleML 0.8, they can also be modularly combined, pairwise or "all in one". A fourth conceivable OO sublanguage consists of signature declarations – possibly roled, grounded, and/or sorted – and their instantiation to '`new`' clauses (normally facts). However, the latter would cross the borderline between declarative KR, currently focussed in RuleML, and procedural KR; this borderline has already been touched with URI-grounded clauses.

The principal relationship between OO RuleML and OO Programming is as follows: Clauses that are (ground) facts correspond to instances, signatures can be viewed as classes, and rules may be used for defining methods. Again, in OO RuleML, those methods are declarative, for querying or deriving information, not procedural as in OOP, for updating a knowledge base or a program state.

There are several connections between OO RuleML and other RuleML extensions, of which we mention those related to some RuleML Technical Group (TG):

- The ongoing work on production and reaction rules often uses – as in Jess – instances/facts stored in the CLIPS format, which employs user-level roles. Moreover, much of the effort in the **Reaction Rules** TG utilizes object-oriented modeling in the style of OMG's UML, OCL, and MOF, whose integration has become easier with OO RuleML (similarly for events).

- The efforts in the **Ontology Combination** TG have led among other results to Description Logic Programs [GHVD03], which can be represented employing user-level roles and order-sorted terms.
- The TG on **Frames, Objects, and RUle Markup (FORUM)** [http://forum.semanticweb.org] – based mostly on F-logic [KL89] and TRIPLE [SD02] – has started studying rules for RDF and graph-based data, which can be mapped to roled, grounded, and possibly sorted OO RuleML.

OO RuleML for rKR has been first implemented via a positionalizing XSLT translator to jDREW for pKR [Spe02], and then directly in the form of the Java-based OO jDREW [http://www.ruleml.org/indoo]. Further tools for OO RuleML currently available from this home page include a positionalizing XSLT translator for OO RuleML's role-weighted extension as well as a pair of Java-based translators between the PR syntax and OO RuleML for rKR and pKR [http://www.ruleml.org/submission/ruleml-shortation.html].

OO RuleML has already served as an interchange format in two major applications. In the RACOFI system [http://racofi.elg.ca] OO RuleML rules are utilized in conjunction with collaborative-filtering techniques for querying a database of music objects rated in multiple dimensions [ABB+03]. For the Treesim algorithm [http://www.cs.unb.ca/~boley/treesimilarity] the role-weighted OO RuleML extension is utilized to represent all product-seaking/advertising trees of the AgentMatcher system [BBY03].

References

ABB+03. Michelle Anderson, Marcel Ball, Harold Boley, Stephen Greene, Nancy Howse, Daniel Lemire, and Sean McGrath. RACOFI: A Rule-Applying Collaborative Filtering System. Submitted for publication, August 2003.

BBY03. Virendra C. Bhavsar, Harold Boley, and Lu Yang. A Weighted-Tree Similarity Algorithm for Multi-Agent Systems in e-Business Environments. In *Proc. Business Agents and the Semantic Web (BASeWEB) Workshop*, pages 53–72. NRC 45836, June 2003.

BLFM98. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. Request for Comments 2396, Network Working Group, The Internet Society, August 1998.

Bol02. Harold Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *Proc. 14th International Conference on Applications of Prolog (INAP2001)*. The University of Tokyo, LNAI 2543, Springer-Verlag, October 2002.

BTW01. Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *Proc. Semantic Web Working Symposium (SWWS'01)*, pages 381–401. Stanford University, July/August 2001.

DS03. Mike Dean and Guus Schreiber. OWL Web Ontology Language – Reference. W3C Candidate Recommendation, W3C, August 2003.

GHVD03. Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. 12th Intl. Conf. on the World Wide Web (WWW-2003)*. Budapest, Hungary, May 2003.

KL89. Michael Kifer and Georg Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 134–146, Portland, Oregon, 31 May–2 June 1989.

Llo87. John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Heidelberg, New York, 1987.

LS99. Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Recommendation REC-rdf-syntax-19990222, W3C, February 1999.

SD02. Michael Sintek and Stefan Decker. TRIPLE – A Query, Inference, and Transformation Language for the Semantic Web. In *1st International Semantic Web Conference (ISWC2002)*. Sardinia, Italy, June 2002.

Spe02. Bruce Spencer. The Design of j-Drew: A Deductive Reasoning Engine for the Web. In *Joint CoLogNet Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems of LOPSTR '02*. Technical University of Madrid, Spain, September 2002.

An Entailment Relation for Reasoning on the Web

François Bry and Sebastian Schaffert

Institute for Computer Science, University of Munich, Germany
schaffer@informatik.uni-muenchen.de

Abstract. Reasoning on the Web is receiving an increasing attention because of emerging fields such as Web adaption and Semantic Web. Indeed, the advanced functionalities striven for in these fields call for reasoning capabilities. Reasoning on the Web, however, is usually done using existing techniques rarely fitting the Web. As a consequence, additional data processing like data conversion from Web formats (e.g. XML or HTML) into some other formats (e.g. classical logic terms and formulas) is often needed and aspects of the Web (e.g. its inherent inconsistency) are neglected. This article first gives requirements for an entailment tuned to reasoning on the Web. Then, it describes how classical logic's entailment can be modified so as to enforce these requirements. Finally, it discusses how the proposed entailment can be used in applying logic programming to reasoning on the Web.

1 Introduction

Emerging Web applications and issues primarily call for *reasoning* capabilities. E.g. Semantic Web applications rely on declarative meta-data specifying the content of Web pages and Web sites and on description logic and/or ontology reasoning for processing these meta-data. Web services similarly rely on declarative meta-data for describing software resources made available on the Web.

There exist languages for expressing such meta-data (e.g. RDF, OWL [1,2]) and also systems for reasoning on meta-data expressed in such languages (e.g. Triple, FaCT [3,4]). However, such languages and/or reasoning systems are restricted to working with meta-data in a specific format and are not capable of retrieving and reasoning with *any* kind of data on the Web, i.e. Web pages and databases as well as – but not limited to – semantic annotations and ontological data. This article investigates a language capable of this.

Such reasoning languages have a multitude of applications. For instance, contract negotiation often takes place when Web services are traded. Wrappers are tools for a structure aware data retrieval that often rely on declarative structure specifications such as grammars, DTD [5] and/or schemas [6]. Data mediators convert data structured according to one DTD or schema into data after another DTD or schema. Adaptive Web systems provide and/or render Web data depending on contexts specifying e.g. user preferences or rendering device characteristics. The adaption of data to context is often realized with rule-based expert systems, i.e. a form of reasoning systems.

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 17–34, 2003.

Reasoning on Web resources calls for methods better fitting the Web context than classical logic and conventional logic programming and automated reasoning. With conventional methods, additional data processing like data conversion from Web formats (e.g. XML [5] or HTML [7]) into some other formats (e.g. classical logic terms and formulas) is often needed and aspects of the Web (e.g. the need for partial queries on graph-shaped terms and the Web's inherent inconsistency) are neglected.

This article first gives a few requirements for an entailment tuned to Web applications. Instead of tree-shaped terms, terms possibly containing cyclic references are needed for such references often occur in XML [5] and HTML [7] data. Atomic formulas are needed using which it might be possible to express partial queries like with XPath [8] or XQuery [9]. A notion of formula satisfaction recursively defined on the formulas' structure is desirable since reasoning on the Web should be kept as "local" as possible. Also, meta-level, inconsistency tolerant, and nonmonotonic forms of reasoning are needed for Web applications. This article defines a notion of an entailment relation fulfilling these requirements. Finally, it discusses how this entailment relation can be used in applying logic programming to reasoning on the Web.

2 Semistructured Expressions

In contrast to the tuples of a relational database that are tree shaped, Web pages correspond to nested, possibly cyclic graphs. Abstracting from XML [5] and HTML [7], Web pages are conveniently formalised as "semistructured data" [10]. Semistructured data items are conveniently represented by "semistructured expressions" (short "sse") [10] that are defined by the following grammar:

```
                <sse> := ( oid "@" )? ( label | label <list>) .
               <list> := <ordered-list> | <unordered-list> .
   <ordered-list> := "[" <sse-str-ref> ( "," <sse-str-ref> )* "]" .
 <unordered-list> := "{" <sse-str-ref> ( "," <sse-str-ref> )* "}" .
    <sse-str-ref> := <sse> | """ string """ | "^" oid .
```

In this grammar, expressions between < and > are non-terminal symbols. "@", "[" , "{", "]", "}", """, and "^" are terminal symbols. "string", "oid" and "label" denote strings, tags and object identifiers, respectively. As usual in the formalisation of programming languages, these three symbols are terminal symbols to which values are assigned.

Slightly extending over XML and HTML, parentheses [ ] are used for expressing a compulsory order (e.g. f[a, b] and f[b, a] denote different data items), while parentheses { } serve to express that the order is irrelevant (e.g. both f{a, b} and f{b, a} denote the same data item). Ordered subterms are useful in representing (structured) texts and unordered subterms are useful in representing (structured) database items.

Semistructured expressions give rise to expressing tuples. E.g.

```
flight { number             [ "AF123"  ],
         departure-time     [ "1230" ],
         arrival-time       [ "1405" ],
         departure-airport  [ "Munich" ],
         arrival-airport    [ "Paris" ] }
```

is a possible representation of a tuple from a relation "flight" with attributes "number", "departure-time", "arrival-time", "departure-airport", and "arrival-airport". Semistructured expressions also give rise to representing cyclic XML documents like the following (where "key" is an attribute of type ID and "person-ref" is an attribute of type IDREF):

```
<persons>
     <person key="bry">
       <vorname>François</vorname>
       <nachname>Bry</nachname>
       <friend person-ref="schaffert"/>
     </person>
     <person key="schaffert" >
       <vorname>Sebastian</vorname>
       <nickname>Wastl</nickname>
       <nachname>Schaffert</nachname>
       <friend person-ref="bry"/>
       <friend person-ref="schaffert"/>
     </person>
</persons>
```

This XML document can be expressed as a semistructured expression as follows:

```
persons [ &1 @ person [ first-name [ "François" ],
                        last-name [ "Bry" ],
                        friend [ ^ &2 ]  ],
          &2 @ person [ first-name [ "Sebastian" ],
                        nickname [ "Wastl" ],
                        last-name [ "Schaffert" ],
                        friend [ ^ &1 ],
                        friend [ ^ &2 ]  ]  ]
```

Note that this semistructured expression contains two (directed) cycles.

Object identifiers in a semistructured expression are assumed to fulfil the following well-formedness conditions:

- Every object identifier &n referred to (i.e. occurring right of ^) in a semistructured expression, is also defined (i.e. occurs left of @) in this semistructured expression.
- An object identifier &n is defined (i.e. occurs left of @) at most once in a semistructured expression.

It might be convenient to also require that an object identifier &n defined in a semistructured expression is also referred to in this semistructured expression.

Without loss of generality, some features of XML and HTML (such as attributes, namespaces, DTD and/or schemas) that are not relevant to the present

study and redundancies of XML and HTML (such as the three referencing formalisms through ID-IDREF attributes, URIs, and Links) are not conveyed in semistructured expressions. Note that attributes can be expressed as semistructured expressions using an unordered collection of strings.

In the literature, slightly different formalisms are used for semistructured expressions. E.g. in [10] unordered children are not considered, one-element lists are represented without parentheses (e.g. instead of "first-name["François"]", [10] writes "first-name:"François""), object identifier definitions and object identifier references both appear to the right of an expression.

3 Requirements on a Logic for Reasoning on the Web

The paradigm considered in the following is to view a Web page as a ground atom and the Web as a (very large) set of Web pages. This paradigm translates into logic the view of a Web page as a data item and of the Web as a (large and highly distributed) database which is common in database research. A logic fulfilling the following requirements would make it possible to express Web-related reasoning problems directly, i.e. without translation of syntax and/or reasoning paradigms.

Terms Shaped as Rooted Graphs. Since Web pages (and therefore semistructured data) are shaped as rooted graphs possibly including cycles, a logical language with terms similarly shaped would ease reasoning with Web data.

Terms as Formulas. There is no natural way to classify Web page (and therefore semistructured) constructs into constructs interpreted as predicate and constructs interpreted as function symbols. Thus, there is no natural way to classify the subexpressions of a Web page (or semistructured expression) into terms and formulas. Therefore, a logical language not distinguishing between terms and formulas would be natural for reasoning on the Web.

Also, the *resource description framework* (RDF) proposed by the W3C [1] as a means to add semantics to web pages does not make such a distinction. In RDF, statements can both be interpreted as predicates and as objects.

Partial Queries. Queries on the Web as expressed in XQuery [9] and XPath [8] might specify Web pages partially. Therefore, a logical language for reasoning on the Web should make partial queries possible.

Satisfaction Recursively Defined. Classical logic satisfaction is defined by structural recursion: The truth value of a formula in an interpretation is defined in terms of the truth value of its subformulas in this interpretation. Recursive definitions of formula satisfaction give rise to evaluation procedures that are *local* to the formulas to evaluate and therefore can be efficiently processed against large data and/or knowledge bases. Therefore, a formula satisfaction recursively defined on the formulas' structures is desirable.

Meta-level Statements. Reasoning with meta-level assertions like database integrity constraints (i.e. statements that some Web data or sites should, or could, but not necessarily do fulfil) is desirable for advanced (e.g. adaptive or semantic Web) applications.

Inconsistency Tolerance. Because the Web is inherently heterogeneous, it is inherently inconsistent. Therefore, an inconsistency tolerant entailment relation is desirable for reasoning on the Web.

Nonmonotonic Negation. Nonmonotonic negation is needed on the Web as it is needed in relational databases: I.e. missing statements (e.g. a missing flight) allow to conclude the truth of the statement's negation (e.g. a missing flight does not exist).

4 Formulas for Reasoning on the Web

Building upon semistructured expressions (cf. above Section 2), a logical language is proposed for reasoning on the Web. In this language, semistructured expressions are a special kind of ground atomic formulas. Atomic formulas extend semistructured expressions with logical variables and a "descendant" construct used for "partial queries".

4.1 Atomic Formulas

Atomic formulas extend semistructured expressions with (logical) variables and with the descendant construct *desc*. These constructs can be informally understood as follows: Variables range over semistructured expressions and the descendant construct *desc* gives rise to specify a subexpression at any depth. Atomic formulas (or atoms) are defined by the following grammar:

```
        <atom> := ( oid "@" )? ( "desc" )?
                  (variable | label | ( variable | label ) <list>) .
        <list> := <ordered-list> | <unordered-list> .
<ordered-list> := "[" <atom-str-ref> ( "," <atom-str-ref> )* "]" .
<unordered-list> := "{" <atom-str-ref> ( "," <atom-str-ref> )* "}" .
<atom-str-ref> := <atom> | """ string """ | "^" oid .
```

Note that variables can occur everywhere except at the place of an object identifier. As in Prolog, identifiers beginning with an upper case letter will denote variables. The following atomic formula can be used for querying a Web page listing flights like an example of Section 2:

```
flight {
   number             [ Nb ],
   departure-time     [ Time1 ],
   arrival-time       [ Time2 ],
   departure-airport  [ "Munich" ],
   arrival-airport    [ "Paris" ]
}
```

The following atomic formula is a *partial query* to the same Web page returning the numbers (as bindings for the variable Nb) of the flights to Paris:

```
flight {
   number              [ Nb ],
   arrival-airport     [ "Paris" ]
}
```

Note the difference from classical logic and Prolog that require such a query to explicitely mention all the attributes of a flight tuple.

The descendant construct is a further means for expressing *partial queries*. The following three atomic formulas are partial queries to the persons Web page of Section 2. The first query evaluate to false. The second query evaluates to true returning the binding last-name/Label. The third evaluates to true (because of the references between the 'person' expressions).

```
desc "Mary"
persons { desc Label [ "Schaffert" ] }
desc person [ desc "Sebastian", desc "François" ]
```

Ordered lists in queries are satisfied only by ordered lists in semistructured expressions, while unordered lists in queries are satisfied by both ordered and unordered lists in semistructured expressions. Thus, evaluated against $f[g[a,b],g\{c,d\}]$ the query $desc\ g[X,Y]$ returns the single answer $a/X, b/Y$ while $desc\ g\{X,Y\}$ returns the two answers $a/X, b/Y$ and $c/X, d/Y$. Unordered lists are not easily expressible in a DTD. A similar notion is achieved with *all* groupings of XML Schema [6].

4.2 Compound Formulas

Compound formulas are built up as usual from atomic formulas using the connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and $\neg$ and the quantifiers $\forall$ and $\exists$. E.g. the following is an open formula specifying one-stop connections from Munich to London:

```
flight {departure-airport ["Munich"], arrival-airport [Via]} ∧
   flight {departure-airport [Via], arrival-airport ["London"] }
```

The following is a closed formula expressing that every flight from A to B has a return flight:

```
∀ Nb1 ∀ D ∀ A
flight{number[Nb1], departure-airport[D], arrival-airport[A]}
  ⇒ ∃ Nb2
  flight{number[Nb2], departure-airport[A], arrival-airport[D]}
```

4.3 Propositional Formulas

As defined in 2, a semistructured expression, might be a label only (e.g. `a` or `flight`). Such semistructured expressions correspond to XML and HTML empty elements [5,7]. They might be seen as classical logic propositional variables or 0-ary predicate symbols, i.e. atomic formulas of propositional logic. Let us call such

semistructured expressions "propositional atoms". Formulas build up (as defined in Section 4.2) from propositional atoms have exactly the form of propositional logic formulas. In the following, they are called "propositional formulas".

5 Entailment: Positive Formulas

The meaning of positive formulas (i.e. formulas in which no negations explicitely occur) has been informally introduced in Section 4. It is now formalised.

5.1 Interpretations

In a first approximation, an interpretation is conveniently defined as a set of semistructured expressions. Since semistructured expressions represent Web pages, this definition is well suited to reasoning on the Web. This definition is refined below in Section 6.1 so as to accommodate negation as discussed in Section 3.

The notion of interpretation considered here and in Section 6.1 can be further developed assigning to every label (or tag) a "multi-arity relation" (i.e. a subset of $\bigcup_{n\in N} \mathcal{E}^n$ where $\mathcal{E}$ is the set of semistructured expressions). This leads to an novel and interesting notion of relation composition. For space reasons, this is not further developed here.

The definition of interpretations considered here might be unusual from the angle of classical logic. Arguably, it is rather natural from the angle of logic programming since it is quite close to Herbrand interpretations.

Because queries on the Web might be partial specifications, some subexpressions of a semistructured expression true in an interpretation $\mathcal{I}$ are also true in $\mathcal{I}$. Consider for example the following (singleton) interpretation $\mathcal{F}$:

```
flights{
          flight {
             number             [ AF1 ],
             departure-time     [ 1230 ],
             arrival-time       [ 1405 ],
             departure-airport  [ "Munich" ],
             arrival-airport    [ "Paris" ]
          },
          flight {
             number             [ AF2 ],
             departure-time     [ 1530 ],
             arrival-time       [ 1615 ],
             departure-airport  [ "Paris" ],
             arrival-airport    [ "Munich" ]
          }
}
```

So as to accommodate partial queries as described in Section 4.1, the following semistructured expressions (among others) must be satisfied (or true) in $\mathcal{F}$:

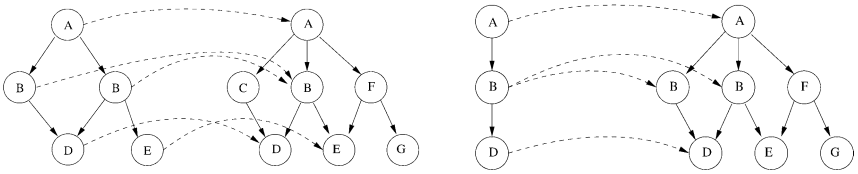

Fig. 1. Rooted Graph Simulations (with respect to vertex adornment equality)

```
flights
flights { flight }
flights { flight { number } }
flights { flight { number [ AF1 ] } }
flights { flight { number [ AF1 ] }, flight { number [ AF2 ] } }
```

5.2 Rooted Simulation

A notion of rooted simulation is used below in Section 5.3 in formalising the satisfaction of semistructured expressions and atomic formulas in an interpretation. The following definition is inspired from [11,12] and refines the simulation considered in [13]. Recall that a (directed) rooted graph $G = (V, E, r)$ consists in a set V of vertices, a set E of edges (i.e. ordered pairs of vertices), and a vertex r called the root of G such that there is in G a path from r to each vertex of G.

Definition 1 (Rooted Graph Simulation). *Let $G_1 = (V_1, E_1, r_1)$ and $G_2 = (V_2, E_2, r_2)$ be two rooted graphs and let $\preceq \subseteq V_1 \times V_2$ be an order relation. A relation $\mathcal{S} \subseteq V_1 \times V_2$ is a* rooted simulation *of G_1 in G_2 with respect to $\preceq$ if:*

1. *$r_1 \; \mathcal{S} \; r_2$.*
2. *If $v_1 \; \mathcal{S} \; v_2$, then $v_1 \preceq v_2$.*
3. *If $v_1 \; \mathcal{S} \; v_2$ and $(v_1, v_1') \in E_1$, then there exists $v_2' \in V_2$ such that $v_1' \; \mathcal{S} \; v_2'$ and $(v_2, v_2') \in E_2$*

A rooted simulation $\mathcal{S}$ of G_1 in G_2 with respect to $\preceq$ is minimal *if there are no rooted simulations $\mathcal{S}'$ of G_1 in G_2 with respect to $\preceq$ such that $\mathcal{S}' \subset \mathcal{S}$ (and $S \neq S'$).*

Definition 1 does not preclude that two distinct vertices v_1 and v_1' of G_1 are simulated by the same vertex v_2 of G_2, i.e. $v_1 \; \mathcal{S} \; v_2$ and $v_1' \; \mathcal{S} \; v_2$. Figure 1 gives examples of simulations with respect to the equality of vertex adornments. The simulation of the right example is not minimal.

In the following, the order relation $\preceq$ considered is "equality of labels" and "compatibility of grouping", i.e. if v_1 has label l_1 and unordered children and if v_2 has label l_2 and ordered children, then $v_2 \not\preceq v_1$ even if $l_1 = l_2$ and $v_1 \preceq v_2$ if $l_1 = l_2$.

A semistructured expression E induces as follows a graph G_E whose vertices are adorned with labels of E and their "kinds of grouping", i.e. ordered ([ ]) or unordered ({ }): The vertices of G_E are those subexpressions of E that are

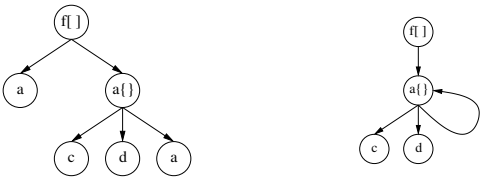

Fig. 2. $f[a, a[c, d, a]]$ and $f[\&1 \ @ \ a\{c, d, \hat{} \&1\}]$ as graphs

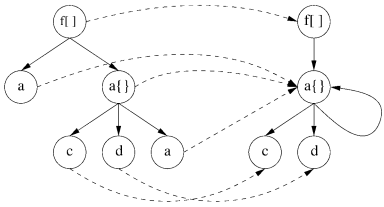

Fig. 3. Minimal simulation of $f[a, a\{c, d, a\}]$ in $f[\&1 \ @ \ a\{c, d, \hat{} \ \&1\}]$

semistructured expressions, a vertex of G_E is adorned with the leftmost label and kind of grouping of the semistructured expression it corresponds to, G_E has an edge (E_1, E_2) if E_2 is an immediate, proper subexpression of E_1 or if an immediate, proper subexpression of E_1 is a reference to E_2. Figure 2 illustrates this view of semistructured expressions as graphs.

Considering the view of semistructured expressions as graphs, the notion of rooted simulation immediately extends to semistructured expressions. Intuitively, there exists a simulation of a semistructured expression E_1 in a semistructured expression E_2 if the labels and the structure of E_1 can be found in E_2 (cf. Figure 3).

5.3 Satisfaction of Atomic Formulas

Recall that an interpretation is a set of semistructured expressions. A semistructured expression E is satisfied (i.e. true) in an interpretation $\mathcal{I}$ if for some semistructured expression $E' \in \mathcal{I}$ there exists a minimal rooted simulation of E in E'. In particular, E is satisfied in $\mathcal{I}$ if $E \in \mathcal{I}$ (since vertex identity is a rooted simulation of E in itself). Thus, interpretations must be *closed under rooted simulation:* if $\mathcal{I}$ is an interpretation, $E \in \mathcal{I}$, and E' is a semistructured expression simulated in E, then $E' \in \mathcal{I}$.

This definition conveys the notion of partial queries to those ground atomic formulas that are semistructured expressions. It is extended to atomic formulas with variables or descendant constructs by extending the notion of rooted simulation of Section 5.2 as follows (cf. Figure 4 for an illustration):

Definition 2 (Formula Satisfaction – Part 1).

- *There exists a minimal rooted simulation of an atomic formula desc A in a semistructured expression E if there exists a subexpression E' of E and a minimal rooted simulation of A in E'.*

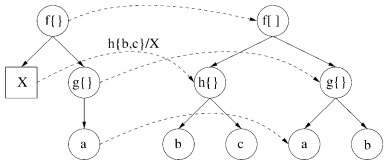

Fig. 4. Minimal simulation of the atomic formula $f\{X, g\{a\}\}$ in the semistructured expression $f[h\{b, c\}, g\{a, b\}]$

– *There exists a minimal rooted simulation of an atomic formula A in a semistructured expression E if there exists a grounding substitution of A (i.e. an assignment of semistructured expressions for variables in A) and a minimal rooted simulation of $A\sigma$ in E.*

5.4 Satisfaction of Compound Formulas

The satisfaction of compound formulas in which no negations explicitely occur is defined as usual recursively on the formulas' structures. The satisfaction of formulas with explicit negation is handled in a nonstandard manner in Section 6.1 below.

Definition 3 (Formula Satisfaction – Part 2). *Let $\mathcal{I}$ be an interpretation. Let F, F_1, and F_2 be formulas. Let X be a variable.*

– *$F_1 \wedge F_2$ is satisfied in $\mathcal{I}$ if both F_1 and F_2 are satisfied in $\mathcal{I}$.*
– *$F_1 \vee F_2$ is satisfied in $\mathcal{I}$ if at least one of F_1 and F_2 is satisfied in $\mathcal{I}$.*
– *$F_1 \Rightarrow F_2$ is satisfied in $\mathcal{I}$ if $\neg F_1 \vee F_2$ is satisfied in $\mathcal{I}$.*
– *$F_1 \Leftrightarrow F_2$ is satisfied in $\mathcal{I}$ if both $\neg F_1 \vee F_2$ and $F_1 \vee \neg F_2$ are satisfied in $\mathcal{I}$.*
– *$\forall X F$ is satisfied in $\mathcal{I}$ if for all semistructured expressions E $F[E/X]$ is satisfied in $\mathcal{I}$.*
– *$\exists X F$ is satisfied in $\mathcal{I}$ if for some semistructured expression E $F[E/X]$ is satisfied in $\mathcal{I}$.*

5.5 Entailment

Entailment between formulas or sets of formulas F and G is defined as usual: $F \models G$ if every interpretation satisfying F also satisfies G. A *(paraconsistent) model* of a set $\mathcal{F}$ of formulas is a (paraconsistent) interpretation satisfying each formula in $\mathcal{F}$. A model $\mathcal{M}$ of $\mathcal{F}$ is *minimal* if no strict subsets of $\mathcal{M}$ are (paraconsistent) models of F.

5.6 Satisfaction of Propositional Formulas

On propositional atoms, i.e. atomic formulas reduced to labels (cf. Section 4.3), Definition 2 coincides with classical logic satisfaction. Therefore, the satisfaction of a propositional positive formula resulting from Definitions 2 and 3 coincides with that of classical logic.

6 Entailment: Integrity Constraints and Nonmonotonic Reasoning

In this section, the notion of interpretation introduced in Section 5.1 is refined yielding a framework for defining the satisfaction of negated formulas in a (non-standard) manner.

6.1 Paraconsistent Interpretations

In the (paraconsistent) interpretations defined below both a negated formula $\neg F$ and its negation $\neg\neg F$ might be satisfied. In other words, classical logic's double negation elimination (i.e. $\neg\neg F \models F$) is dropped. Keeping with relational databases and logic programming, if an atomic formula A is true (false, resp.) in an interpretation $\mathcal{I}$, then $\neg A$ is false (true, resp.) in $\mathcal{I}$, and if $\neg\neg A$ is true (false, resp.) in $\mathcal{I}$, then $\neg\neg\neg A$ is false (true, resp.) in $\mathcal{I}$. Furthermore, if a (paraconsistent) interpretation satisfies a formula F, then $\mathcal{I}$ also satisfies $\neg\neg F$. Thus, an atomic formula A can be interpreted as follows (the first and last interpretations are like in classical logic):

A	$\neg A$	$\neg\neg A$	$\neg\neg\neg A$
true	false	true	false
false	true	true	false
false	true	false	true

Definition 4 (Interpretation). *A (paraconsistent) interpretation $\mathcal{I}$ is a set of semistructured expressions or doubly negated semistructured expressions such that:*

1. *$\mathcal{I}$ is closed under rooted simulation, i.e. if $E \in \mathcal{I}$ and if E' is a semistructured expression simulated in E, then $E' \in \mathcal{I}$.*
2. *If E is a semistructured expression and if $E \in \mathcal{I}$, then $\neg\neg E \in \mathcal{I}$.*

A (paraconsistent) interpretation $\mathcal{I}$ is consistent *if for all $\neg\neg E \in \mathcal{I}$, $E \in \mathcal{I}$. It is* inconsistent *otherwise.*

Dropping the elimination of double negation avoids problems found in classical logic. Consider for example a course database with integrity constraints as follows:

$$\begin{aligned} \text{DB: } & course[\text{"}CS1\text{"}] \\ & course[\text{"}CS2\text{"}] \\ & teaches[\text{"}Anna\text{"}, \text{"}CS1\text{"}] \\ \text{IC: } & \forall X.course[X] \;\Rightarrow\; \exists Y.teaches[Y, X] \end{aligned}$$

In classical logic, for which a so-called "open world assumption" holds, there exists a model for this database and integrity constraint (since it is not explicitly stated that there is no teacher for CS1), and thus it is consistent. Database

systems avoid this problem by relying on a so-called "closed world assumption" or Clark's completion [14] and by distinguishing between basic data (like course["CS1"]) and integrity constraints. Such a distinction is not necessary with paraconsistent interpretatrions as above because integraity constraints are prefixed with a double negation. The integrity constraint of the example above is thus expressed as:

$$\text{IC: } \neg\neg \left(\forall X.course[X] \;\Rightarrow\; \exists Y.teaches[Y, X]\right)$$

A paraconsistent model satisfying $course["CS1"]$, $course["CS2"]$, and the formula above does not have to satisfy $teaches[a, "CS2"]$ for some a. Instead, it suffices that $\neg\neg teaches[a, "CS2"]$ holds for some a, intuitively expressing that $teaches[a, "CS2"]$ *should* hold for some a.

6.2 Satisfaction of Negated Formulas

The satisfaction of negated formulas in a (paraconsistent) interpretation $\mathcal{I}$ is defined recursively on the formulas' structure by extending Definition 2 of Section 5.4 with the following rules:

Definition 5 (Formula Satisfaction – Part 3). *Let $\mathcal{I}$ be an interpretation. Let A be an atomic formula. Let F be a formula.*

- *$\neg A$ is satisfied in $\mathcal{I}$ if A is not satisfied in $\mathcal{I}$, i.e. there are no semistructured expressions $B \in \mathcal{I}$ with a minimal rooted simulation of A in B.*
- *$\neg\neg\neg A$ is satisfied in $\mathcal{I}$ if there are no $\neg\neg B \in \mathcal{I}$ with a minimal rooted simulation of A in B.*
- *$\neg\neg\neg\neg F$ is satisfied in $\mathcal{I}$ if $\neg\neg F$ is satisfied in $\mathcal{I}$.*

In a word, four and more than four nested negations are treated in the spirit of classical logic, double negations are not eliminated, and $F \models \neg\neg F$. Two kinds of positive literals are available, atoms and doubly negated atoms. Negated atoms and doubly negated atoms are treated in the relational database and logic programming style. From Definitions 4, 2, 3, and 5, it (easily) follows that for all formulas F, $F \models \neg\neg F$ (but $\neg\neg F \not\models F$).

Note that a consistent interpretation of propositional formulas induces a classical logic interpretations of these formulas. This is not the case of an inconsistent interpretation in which for some A, $\neg\neg A$ is true but A is not true.

Section 6.3 and Section 6.4 below point to the advantages of this unusual treatment of negation for reasoning on the Web.

6.3 Meta-level Reasoning: Integrity Constraints

Positive and general program clauses and programs are defined as usual but referring to the nonstandard atomic formulas defined in Section 4.1:

Definition 6 (Program Clauses and Programs).

- *A* program clause *is an expression of the form* $A \leftarrow B_1, \ldots B_n$ *where the* B_i *are atomic formulas or negated atomic formulas and* A *is an atomic formula in which no desc constructs occur. It denotes the formula* $\forall X_1 \ldots \forall X_m\ (B_1 \wedge \ldots \wedge B_n) \Rightarrow A$ *where* $X_1, \ldots, X_m$ *are the variables occurring in* $B_1, \ldots, B_n$, *or* A. *If all* B_i *are atomic formulas, then it is a* positive clause, *else a* general clause.
- *A* positive (general, resp.) program *is a finite set of positive (general, resp.) program clauses. A* propositional program *is a program in the clauses of which only propositional atomic formulas or propositional negated atomic formulas occur.*

Precluding *desc* constructs in clauses' heads ensure that positive programs have only one minimal model. Indeed, a clause like $a\{desc\ b\} \leftarrow B_1, \ldots B_n$ defines infinitely many atoms e.g. $a\{b\}$, $a\{a\{b\}\}$, $a\{a\{b\}, a\{b\}\}$, etc.

Integrity constraints to a program P are closed formulas that might (or, depending on the application, should, or could) logically follow from P. If C is a set of integrity constraints to a program P, the problem called "integrity verification" is to decide whether $P \models C$. If this is the case, P is said to satisfy the integrity constraints, otherwise to violate them. Note that the data specified by P, i.e. the model(s) of P, should not depend on C. Thus, integrity constraints are statements on P, i.e. meta-level statements.

If integrity constraints are represented as doubly negated (closed) formulas $\neg\neg F$, then the entailment relation of Section 6.1 suffices to solving integrity verification problems:

Proposition 1. *A positive program* P *satisfies a set of integrity constraints* C *represented as doubly negated formulas if and only if the minimal (paraconsistent) models of* $P \cup C$ *are consistent.*

Proof. Let P be a positive program and let C be a set of integrity constraints (of the form $\neg\neg F$) to P. Let $\mathcal{I}$ be a paraconsistent interpretation such that $\mathcal{I} \models P \cup C$, i.e. a paraconsistent model of $P \cup C$. Clearly, the subset of $\mathcal{I}$ consisting in all atoms of $\mathcal{I}$ is a standard Herbrand model of P (as defined in [15]). If $\mathcal{I}$ is minimal, then the subset of $\mathcal{I}$ consisting in all atoms in $\mathcal{I}$ is the minimal Herbrand model of P (in the standard sense). Now, observe that $P \models C$ if and only if $P \cup C$ has no minimal models that are inconsistent (i.e. models $\mathcal{M}$ with $\neg\neg A \in \mathcal{M}$ and $A \notin \mathcal{M}$ for some semistructured expression A).

6.4 Nonmonotonic Reasoning

With the entailment relation of Section 6.1, a ground program clause $A \leftarrow B_1, \ldots B_n, \neg C_1 \vee \ldots \vee \neg C_m$ is logically equivalent to $A \vee \neg B_1 \vee \ldots \vee \neg B_n \vee \neg\neg C_1 \vee \ldots \neg\neg C_m$. It is not only satisfied in interpretations in which A, or some $\neg B_i$, or some C_j is true, but also in interpretations in which some $\neg\neg C_j$ is true. Among such (paraconsistent) interpretations are inconsistent interpretations satisfying

$\neg\neg C_j$ but not satisfying C_j. Such interpretations "gives room" for interpreting so-called cycles of recursion through negation with an odd length [16] in a quite natural manner. E.g. the clause $p \leftarrow \neg p$ has a single minimal (inconsistent) paraconsistent model: $\{\neg\neg p\}$. Major advantages of the treatment of negation proposed above is that it extends the Stable Model Semantics [17] and gives it a "minimal model setting":

Proposition 2. *Let P be a propositional program. Every consistent model of P (in the sense of Section 6.1) is stable. Every stable model of P (in the sense of [17]) characterises a consistent model of P (in the sense of Section 6.1).*

If $\mathcal{M} = \{A_1, \ldots, A_k\}$ is a stable model, then the model in the sense of Section 6.1 $\mathcal{M}$ is said to characterise is $\{A_1, \neg\neg A_1, \ldots, A_k, \neg\neg A_k\}$.

Proof. Proposition 2 follows from the characterisation of minimal (paraconsistent) models given below in Proposition 3 which rephrases the program transformation of [17] and extend it to general formulas.

Proposition 3. *Let $\mathcal{M}$ be a (paraconsistent) model of a set of formulas S. Let $\overline{\mathcal{M}} = \{\neg E \mid E$ semistructured expression and $E \notin \mathcal{M}\} \cup \{\neg\neg\neg E \mid L$ semistructured expression and $\neg\neg E \notin \mathcal{M}\}$. $\mathcal{M}$ is a minimal model of S if and only if for all $K \in \mathcal{M}$, $S \cup \overline{\mathcal{M}} \models K$.*

Proof. **Necessary Condition.** Assume $\mathcal{M}$ is a minimal (paraconsistent) model of S. If $\mathcal{M} = \emptyset$, then the property trivially holds. Otherwise, let $K \in \mathcal{M}$. If $S \cup \overline{\mathcal{M}} \not\models K$, then by Definition 5 $S \cup \overline{\mathcal{M}} \cup \{\neg K\}$ has a minimal (paraconsistent) model $\mathcal{N}$. By definition of $\mathcal{N}$, $\mathcal{N} \models \neg K$, hence by Definition 5, $\mathcal{N} \not\models K$, i.e. $K \notin \mathcal{N}$. Since $\mathcal{N} \models \mathcal{S} \cup \overline{\mathcal{M}}$, $\mathcal{N} \subseteq \mathcal{M}$. Since $K \in \mathcal{M} \setminus \mathcal{N}$, $\mathcal{N} \neq \mathcal{M}$ contradicting that $\mathcal{M}$ is a minimal model of S. Thus, $S \cup \overline{\mathcal{M}} \models K$.
Sufficient Condition. Assume that for all $K \in \mathcal{M}$, $S \cup \overline{\mathcal{M}} \models K$. If $\mathcal{M}$ is not a minimal (paraconsistent) model of S, then there exists a strict subset $\mathcal{N}$ of $\mathcal{M}$ such that $\mathcal{N} \models S$. Let $\overline{\mathcal{N}} = \{\neg E \mid E$ semistructured expression and $E \notin \mathcal{N}\} \cup \{\neg\neg\neg E \mid E$ semistructured expression and $\neg\neg E \notin \mathcal{N}\}$. From the necessary condition it follows that for all $K \in \mathcal{N}$, $S \cup \overline{\mathcal{N}} \models K$. Since $\mathcal{N}$ is a strict subset of $\mathcal{M}$ there exists $K \in \mathcal{M} \setminus \mathcal{N}$, hence $\neg K \in \overline{\mathcal{N}}$ and $(\star)$ $S \cup \overline{\mathcal{N}} \models \neg K$. By hypothesis, $S \cup \overline{\mathcal{M}} \models K$. Since $\mathcal{N} \subset \mathcal{M}$, $\overline{\mathcal{M}} \subset \overline{\mathcal{N}}$. Therefore, $(\star\star)$ $S \cup \overline{\mathcal{N}} \models K$. $(\star)$ and $(\star\star)$ are contradictory, refuting that $\mathcal{M}$ is not minimal.

7 Logic Programming for Reasoning on the Web

Many Web applications are based on (1) selecting data from the Web and (2) processing the selected data in manners that are particularly amenable to declarative programming, especially to logic programming. The entailment relation introduced in the previous sections has been conceived so as to support such applications that are briefly described below. It provides with the semantics of a prototype Web query language called Xcerpt [13,18].

Dynamic Web Pages. Dynamic Web pages are texts or data that are dynamically generated when called. They make it possible for different pages to share data thus ensuring data consistency and to generate up-to-date Web pages from changeable data. Arguably, logic programming queries would be as convenient for the Web as views are in relational databases. Dynamic Web pages based on a logic query language would be more amenable to reasoning (e.g. for query optimisation purposes) than dynamic Web pages based on imperative scripts.

Adaptive Web. Most adaptive Web systems combine portions of text into Web pages depending on contexts specifying so-called user models (i.e. user preferences and/or rendering device characteristics) using rule-based systems. Arguably, a logic programming query language would be convenient to implement both, the retrieval of portions of text from the Web and their combination into context-dependent Web pages, thus considerably simplifying the implementation of adaptive Web systems.

Structure Transformations. Structure transformations are an essential application of languages such as XQuery [9] and XSLT [19]. Arguably, logic programming languages are especially convenient to express structure transformations because they are term (or pattern oriented) and because they are convenient to express recursion through term structures.

Styling. Styling, i.e. enriching an XML or HTML page with rendering parameters, is a special kind of transformation conveniently expressed in rule-based formalisms. CSS [20] is such a language that has interesting similarities with logic programming.

Semantically Aware Querying. Considering semantics annotations as expressed e.g. in ontologies while evaluating queries on the Web is an emerging research issue. To this aim, the reasoning system of ontologies is coupled with a programming language. Arguably, a logic programming query language could be used for both tasks.

8 Related Work

The work presented in this paper is related to query and transformation languages for XML [5] and semistructured data. XPath [8] and XQuery [9] are well known such languages. They are widely used Web standards. They are "navigational" in the sense that they express data retrieval in terms of root-to-node data item traversals, i.e. a rather procedural approach.

Other query languages for XML [5] and semistructured data do not build upon a navigational paradigm. UnQL [21] first proposed to express queries as terms (or patterns) as in logic and in the present paper. Such query languages can be called "positional" because the relative positions of the data to

retrieve, e.g. variables, are well conveyed in query terms (or patterns). A further positional query language is XMas [22]. Both UnQL and XMas are functional and inspired from the object database query language OQL [23]. Xcerpt (cf. `http://www.xcerpt.org`) is a further positional query and transformation language for XML and semistructured data. Xcerpt is based on the logic programming paradigm. The present paper is a contribution to Xcerpt's semantics. Xcerpt's operational semantics has been presented in [13].

Further query languages for XML and semistructured data are XSLT [19] and fxt [24]. XSLT is based on the matching of XML elements and on a built-in structural recursion of XML documents and it also offers a comprehensive collection of imperative programming constructs. A severe limitation of XSLT is that it has no real notion of procedure: An output of an XSLT subprogram cannot be further processed within the same program. fxt builds on tree grammars and an efficient matching of regular expressions (describing the children of a node) against semistructured expressions. fxt's processing is based on tree automata. In some projects, Prolog has been adapted to process and/or query XML and/or semistructured data, e.g. in [25,26,27]. Such Prolog extensions and/or adaptions are inspiring for query languages for XML [5] and semistructured data.

The approach to nonmonotonic reasoning described in the present paper is reminiscent of a widespread, often empirical approach consisting in "duplicating" every predicate p (cf. e.g. [16,28,29]). [30] describes in more detail this approach in the framework of classical logic. Proposition 3 is an adaptation to the nonstandard models of Definition 4 of a result given in [31] for classical logic.

9 Conclusion

This article has first given requirement for logics for reasoning on the Web as needed for emerging applications such as Semantic Web and adaptive Web systems. Then, it has defined an entailment relation for such a logic. A first salient aspect of this entailment relation is that it conveys a notion of partial queries reminding of the Web query languages XPath [8] and XQuery [9]. A second salient aspect of the entailment relation is that the satisfaction of a formula is recursively defined on the formulas' structures, like in classical logic and unlike most nonmonotonic logics. Arguably, such definitions of formula satisfaction give rise to efficient inference procedures local to the formulas considered, thus well-suited to the Web. A third salient aspect of the entailment relation proposed in this paper is that it extends the Stable Model Semantics [17] and gives it a "minimal model setting". Finally, the article has briefly discussed how the proposed entailment relation can be used in applying logic programming to reasoning on the Web.

The work reported about in this paper is part of a project aiming at defining a (prototype) logic programming language for reasoning on the Web called Xcerpt [13,18].

References

1. W3C: Resource Description Framework (RDF). (1999)
2. W3C: Web Ontology Language (OWL). (2003)
3. Decker, S.: TRIPLE – an RDF query, inference, and transformation language. (website) `http://triple.semanticweb.org/`.
4. Horrocks, I.: The FaCT System. (website) `http://www.cs.man.ac.uk/~horrocks/FaCT/`.
5. W3C: Extensible Markup Language (XML) 1.0, Second Edition. (2000)
6. W3C: XML Schema Part 0: Primer; Part 1: Structures, Part 2: Datatypes. (2001)
7. W3C: XHTML 1.0: The Extensible HyperText Markup Language. (2000)
8. W3C: XML Path Language (XPath). (1999)
9. W3C: XQuery: A Query Language for XML. (2001)
10. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web. From Relations to Semistructured Data and XML. Morgan Kaufmann (2000)
11. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing Simulations on Finite and Infinite Graphs. Technical report, Cornell Univ. (1996)
12. Milner, R.: An Algebraic Definition of Simulation between Programs. Memo aim-142, Stanford Univ. (1971)
13. Bry, F., Schaffert, S.: Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In: Proc. Int. Conf. on Logic Programming. LNCS, Springer-Verlag (2002)
14. Clark, K.L.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press (1978) 293–322
15. Lloyd, J.: Foundations of Logic Programming. Springer-Verlag (1987)
16. Apt, K.R., Bol, R.: Logic Programming and Negation: A Survey. J. Logic Programming **9** (1994)
17. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Proc. Int. Conf. on Logic Programming. (1988) 1070–1080
18. Bry, F., Schaffert, S.: A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In: Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web. (2002) (invited article).
19. W3C: Extensible Stylesheet Language (XSL). (2000)
20. W3C: Cascading Style Sheets, level 2. (1998)
21. Buneman, P., Fernandez, M., Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. VLDB Journal **9** (2000)
22. Baru, C., Ludöscher, B., Papakonstantinou, Y., Velikhov, P., Vianu, V.: Features and Requirements for an XML View Definition Language: Lessons from XML Information Mediation. In: Proc. QL'98 – The Query Languages Workshop. (1998)
23. Alashqur, A.M., Su, S.Y.W., Lam., H.: OQL: A Query Language for Manipulating Object-Oriented Databases. In: Proc. Int. Conf. on Very Large Data Bases. (1989)
24. Berlea, A., Seidl, H.: fxt – A Transformation Language for XML Documents. J. of Computing and Information Technology (2001)
25. Seipel, D.: Processing XML-Documents in Prolog. In: Proc. Workshop Logische Programmierung. (2002)
26. Heumesser, B., Seipel, D., Güntzer, U.: Flexible Processing of XML-Based Mathematical Knowledge in a Prolog-Environment. In: Proc. Int. Conf. on Mathematical Knowledge Management. LNCS (2003)
27. May, W.: A Logic-Based Approach to XML Data Integration. (2001) Habilitation Thesis.

28. Inoue, K., Sakama, C.: A Fixpoint Characterization of Abductive Logic Programming. J. Logic Programming (1996) 107–136
29. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Trans. Computational Logic **2** (2001) 526–541
30. Bry, F.: An Almost Classical Logic for Logic Programming and Nonmonotonic Reasoning. In: Proc. Paraconsistent Computational Logic. (2002)
31. Niemelä, I.: A Tableau Calculus For Minimal Model Reasoning. In: Proc. Workshop on Theorem Proving with Analytic Tableaux and Related Methods. LNAI, Springer-Verlag (1996)

A Rule-Based XML Access Control Model

Chutiporn Anutariya[1], Somchai Chatvichienchai[2], Mizuho Iwiahara[2], Vilas Wuwongse[3], and Yahiko Kambayashi[2]

[1] Computer Science Program, Shinawatra University, Pathumthani, Thailand
chutiporn@shinawatra.ac.th
[2] Department of Social Informatics, Graduate School of Informatics, Kyoto University, Kyoto, Japan
{somchai,iwaihara,kambayashi}@db.soc.i.kyoto-u.ac.jp
[3] Computer Science and Information Management Program, Asian Institute of Technology, Pathumthani, Thailand
vw@cs.ait.ac.th

Abstract. Due to a widely use of XML language in various application domains, a well-established mechanism for the definition and enforcement of security controls on specific accesses to XML documents is demanded, in order to ensure that only authorized entities can perform certain actions on the protected data. The proposed rule-based, declarative approach supports definition of (possibly implicit and complex) authorization rules on particular nodes within a document as well as enforcement of multiple user-defined policies, specifying selected mechanisms to resolve conflicts or to apply default authorization. Moreover, by founded on both RDF and XDD theory, the developed approach yields a simple yet flexible and interchangeable XML access control model with well-defined declarative semantics.

1 Introduction

Access control of XML documents provides security such that only eligible users can read or write specified portions of the documents. As XML is becoming a common language for information infrastructure, securing access to XML documents is ever becoming important. Typical areas where access control is vital include e-government, e-commerce and medical information systems, where protecting personal data for privacy is an important issue.

A number of models for access control of XML documents have been proposed [3,4,6,11], and OASIS recently announced *eXtensible Access Control Markup Language* (*XACML*) [7] as a standard for access control specification. Essential functionalities required for access control of XML documents are:

1. **Fine-grained access control:** Elements or attribute values within a document can be specified as secured. More generally, protected objects (elements/attribute values) may be specified by path expressions.
2. **User-defined policies for conflict resolution:** An object could have multiple authorization definitions or no definition. In such cases, the system needs to determine proper authorizations through user-defined policies. Typical examples of policies are denial-take-precedence and closed policy (denial as default).

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 35–48, 2003.

3. **Declarative semantics:** Authorization definitions may be exchanged between distributed sites on the Internet to facilitate flexible inter-access between them. In this situation, authorization definitions together with policies are required to be declarative, namely containing logical conditions only, to allow flexible checking of authorizations. Hard-coded programs for processing authorization definitions inhibit inference on those definitions.

The paper proposes an approach to realizing advanced access control of XML documents by employment of an XML-based rule language, namely *XML Declarative Description* (*XDD*) [2,14,15], for authorization and policy definitions. XDD's expressive mechanism for direct representation and manipulation of XML elements, their conditional relationships and constraints enables the developed approach to describe declaratively fine-grained access control and to define multiple, different policies to be enforced on different authorization definitions. In addition, with XDD's well-defined declarative semantics, the approach allows not only derivation of implicit, complex authorizations on the basis of other authorizations, but also facilitates application of appropriate policies to resolve conflicts and to yield an authorization decision.

Section 2 presents the proposed access control model; Section 3 elaborates an XDD approach to the formalization and evaluation of access authorizations; Section 4 discusses related work and Section 5 draws conclusions.

2 XML Access Control Model

Essential information necessary for deciding access authorization for a given access request includes *authorization rules* (authorizations, for short) and *conflict resolution and default policies*.

Definition 1 (Authorization) An *authorization* is a quintuple:

$$\langle subject, object, privilege, type, sign \rangle,$$

where

- *subject* is a user, a user group, a role or credentials describing properties (e.g., age, gender, domain) of the user to whom the authorization is granted;
- *object* is described by a triple $\langle level, target, path \rangle$, where $level \in \{instance, schema\}$ indicating whether the defined authorization is applied at an instance or a schema level, *target* denotes the identifier of an XML document or a DTD or a schema, and *path* denotes an XPath expression identifying an element or attribute within the XML document, DTD or schema;
- $privilege \in \{read, write\}$;
- $type \in \{cascade, no\_prop\}$; and
- $sign \in \{+, -\}$. ❑

Access of users to elements and attributes within an XML document (or all instances of a DTD or schema) is regulated on the basis of authorizations, which specify for each user the types of accesses that the user can/cannot exercise on each object. An authorization can be positive (granting access), +, or negative (denying access), -, to an element or attribute of an XML document (or all DTD instances). An authorization specified on an element can be propagated to all of its direct and indirect sub-

elements (by *cascade* option), or to only its attributes (by *no_prop* option). The *read* privilege allows a subject to view an element and its attributes. The *write* privilege allows a subject to write and delete information in an element (or in some of its parts).

The possibility of specifying both positive and negative authorizations introduces potential conflicts among authorizations. A user may have two authorizations for the same privilege on the same protected object but with different signs. These conflicting authorizations can be either explicit or derived through propagation. Example of conflict resolution policies include:

- **Instance-take-precedence:** Authorizations specified at the document level prevail over authorizations specified at the schema level;
- **Descendant-take-precedence:** Authorizations specified at a given level in the schema/document hierarchy prevail over authorization specified at higher levels;
- **Denial-take-precedence:** Negative authorizations take precedence over positive ones.

In addition, there might exist an element which has no associated authorization, either defined explicitly or propagated along the document structure by means of the cascade option. In such a case, a default authorization could be applied. Two main types of default policies are:

- **Open-Policy:** The access is allowed if there exists no negative authorization.
- **Closed-Policy [10]:** The access is denied if there exists no positive authorization.

```
<!DOCTYPE patient SYSTEM "Paients.dtd">

<patient code="p2305" project="X">
        <name>Jack Black</name>
        <address>10 Dolly, Berkeley, CA, 94720</address>
        <room>
            <number>2015 </number>
            <bed>1</bed>
        </room>
        <illness id="Angina">
            <therapy type="P.T.C.A.">
                <startdate>2003-05-16</startdate>
                <enddate>2003-05-21</enddate>
                <drug>
                    <name>heparin</name>
                    <daily_admin> 30 U/Kg </daily_admin>
                </drug>
            </therapy>
        </illness>
</patient>
```

Fig. 1. A sample XML document patients.xml.

Example 1 Consider the following authorizations for patients.xml of Fig.1.

A_1: ⟨*Staff*, ⟨*instance, patients.xml, /patient*⟩, *read, cascade,* +⟩
A_2: ⟨*Staff*, ⟨*instance, patients.xml, /patient/illness*⟩, *read, cascade,* -⟩
A_3: ⟨*ProjectXDoctor*, ⟨*instance, patients.xml, /patient*⟩, *read, cascade,* +⟩
A_4: ⟨*ProjectXDoctor*, ⟨*instance, patients.xml, /patient[@project="X"]/illness*⟩, *write, cascade,* +⟩

By the authorizations A_1 and A_2, a *Staff* is permitted to read names, addresses and rooms of all patients but is denied to read illness information. The authorization A_3

defines that a *ProjectXDoctor* is permitted to read all information of any patients. Moreover, A_4 authorizes a *ProjectXDoctor* to write the illness and treatment information of a patient whose project code value is equal to X. ❑

3 Authorization Rule Language

A new approach to the representation of access authorizations on XML document databases is proposed by means of XDD theory. The theory is reviewed first and followed by its application to formalize XML access authorizations.

3.1 XML Declarative Description

An Informal Review

XDD [2,14,15]—an XML-based rule language—extends ordinary XML elements by incorporation of variables for an enhancement of expressiveness and representation of implicit information into so-called *XML expressions*. Ordinary XML elements—XML expressions without variable—are called *ground XML expressions*. Every component of an XML expression can contain variables, e.g., its expression or a sequence of subexpressions (*E-variables*), tag names or attribute names (*N-variables*), strings or literal contents (*S-variables*), pairs of attributes and values (*P-variables*) and some partial structures (*I-variables*). Every variable is prefixed by '$*T*:', where *T* denotes its type; for example, $S:value and $E:expression are *S*- and *E*-variables, which can be specialized into a string or a sequence of XML expressions, respectively.

An *XDD description* is a set of *XML clauses* of the form:

$$H \leftarrow B_1, \ldots, B_n,$$

where $n \geq 0$, H is an XML expression, and B_i is an XML expression, an *XML constraint* or a *set-aggregation*. The XML expression H is called the *head*, the set $\{B_1, \ldots, B_n\}$ the *body* of the clause. When $n = 0$, i.e., the body is empty, such a clause is referred to as an *XML unit clause*. When clear from the context, a unit clause ($H \leftarrow$) is represented simply by H; hence, an XML element or document can be mapped directly onto a *ground XML unit clause*.

An *XML constraint*—useful for defining a restriction on XML expressions, their components or their sets—is a formula of the form $q(a_1, \ldots, a_n)$, where $n > 0$, q is a *constraint predicate* and a_i is an XML expression or a set of XML expressions. The satisfaction (truth or falsity) of a *ground constraint* is predetermined.

A *set-aggregation*—a concept employed in the description of complex queries/operations on sets of XML expressions—is an expression of the form:

```
<xdd:SetOf>
  <xdd:Set>S</xdd:Set>
  <xdd:Pattern>a</xdd:Pattern>
</xdd:SetOf>,
```

where S denotes a set of XML elements and a is an XML expression which specifies the pattern of XML elements to be included in the set S.

Intuitively, given an XDD description *P*, its meaning is the set of all XML elements which are directly described by or are derivable from the unit and non-unit clauses in *P*. Further explanations of XML expressions, clauses and XDD descriptions will be given from a practical point of view by means of examples. Paper [15] gives theoretical details of the theory.

Equivalent Transformation Framework

Since XDD focuses on information/knowledge representation, in order to provide a concise and expressive language with precise and well-defined semantics, its underlying representation scheme is separated from its computational mechanism. It achieves efficient manipulation of and reasoning with XDD descriptions by employment of *Equivalent Transformation* (*ET*) [1] computational paradigm, which solves a given problem, formalized as a particular description *P*, by simplifying it through repetitive application of (semantically-)equivalent transformation rules.

Founded on the XDD theory and the ET paradigm, *XML Equivalent Transformation* (*XET*) [2] engine has been developed for materializing the equivalent transformation of XDD descriptions, hence allowing more insight manipulation of and reasoning with XML expressions without a necessity for data conversion. Basically, an XET program comprises a set of *XET rules* – ET rules represented in XML format – and a set of XML elements/documents regarded as the program's data or *facts*. It takes as its input a query formalized as an XDD description and then computes the query's answers by applying the program's rules and facts.

3.2 Access Authorization Specification

With an emphasis on interoperability and information interchange among applications, the developed approach employs *RDF language* [5,12] as a foundation for semantic description of access control information and enhances RDF's expressive power with the facility to represent access control rules by means of the XDD theory. Fig. 2 illustrates the developed RDF-based authorization schema.

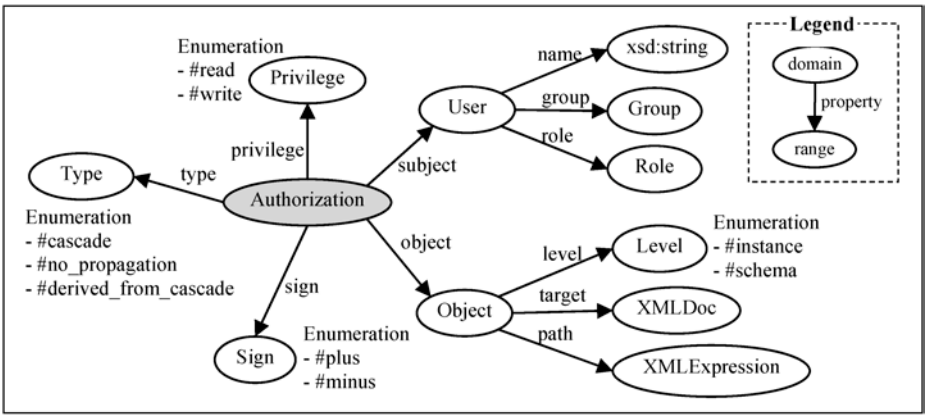

Fig. 2. Authorization schema.

Definition 2 (Authorization Clause) An *XML authorization clause* has the form

$$H \leftarrow B_1, \ldots, B_n,$$

where – H is a (possibly non-ground) Authorization-expression corresponding to the RDF model of Fig. 2,
– B_i is an XML expression, a constraint or a set-aggregation restricting a certain condition on the specified authorization. ❑

Intuitively, the definition states that an Authorization-element consists of five properties (subject, object, privilege, type and sign) corresponding to the access control model presented in Section 2. Simple and explicit authorizations are modeled directly as ground RDF elements while complex and implicit ones are specified as XML non-unit clauses. The heads of such clauses represent corresponding Authorization-elements to be obtained while their bodies comprise certain conditions on document instances, target objects, users or action privilege as well as other Authorization-elements. This representation style presents a compact form for expression of different types of axioms and conditional relationships among authorizations by not only enabling explicit specification of authorizations but also permitting their derivation.

Authorization Subject

Typically, an authorization subject specifies a user to whom the authorization is granted or denied. However, instead of defining authorizations for each specific user, a group and role mechanism can be used to facilitate specification of authorizations for a set of users belonging to a particular group or playing a certain role. In addition, appropriate ontologies defining hierarchical structure of groups and roles can also be expressed, hence permitting simple inferences through the formalized hierarchies. Fig. 3-a gives a simple example of a group ontology expressed in RDF and indicating that ProjectXDoctor is a subClassOf (subgroup of) Doctor. Fig. 3-b then describes that a user John belongs to the group ProjectXDoctor. Thus, according to the subClassOf axiomatic semantics, one can derive that John is also a member of the group Doctor.

```
G1: <Group  rdf:about="#Staff"/>

G2: <Group  rdf:about="#Doctor"/>

G3: <Group  rdf:about="#ProjectXDoctor">
      <rdfs:subClassOf rdf:about="#Doctor"/>
    </Group>
```

(a) A group ontology.

```
G4: <User  rdf:about="#john">
      <name>John Smith</name>
      <group rdf:resource=
             "#ProjectXDoctor"/>
    </User>
```

(b) A user instance.

Fig. 3. A simple group ontology and a user instance represented by RDF.

Authorization Object

An authorization object represents an element node or a set of element nodes within a target document where the specified authorization applies. It has three basic properties: level, target and path. Note that in order to allow reasoning about path expressions, a path is represented in the developed approach as a corresponding XML expression with variables. Fig. 4 illustrates an example.

Example 2 The XML clauses C_1 and C_2 of Fig. 5 formalize authorizations for the document patients.xml of Fig. 1. Moreover, since an authorization can be defined to be of the type cascade, the clause C_3 of Fig. 6 models its semantics which derives a new authorization of the type derived_from_cascade by propagating a given authorization on a particular element down to all of its (either direct or indirect) child nodes. Then, let an XDD description P comprise the subject ontology given by Fig. 3 and the clauses $C_1 - C_3$, i.e., $P = \{G_1, G_2, G_3, G_4, C_1, C_2, C_3,\}$. The meaning of P yields not only the explicitly defined authorizations but also the derived ones, such as those which specify that the user John is allowed to read the element /patient/room/number and to write the element /patient[project="X"]/illness/therapy/drug/name. ❑

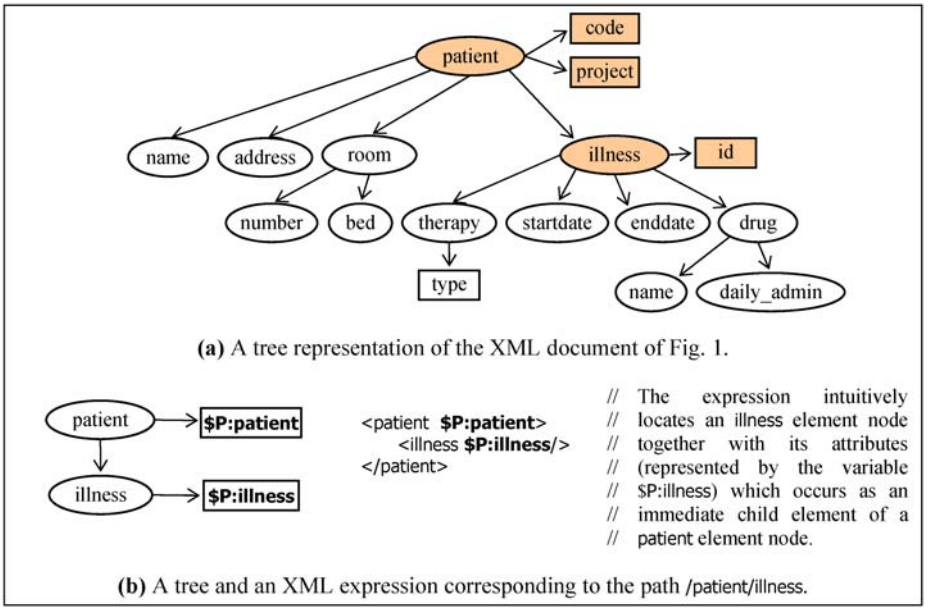

Fig. 4. Representing a path expression as an XML expression.

3.3 Conflict Resolution and Default Authorization Policy

Definition 3 (Policy Clause) A *policy* is represented as an XML clause of the form

```
<Policy>
  <privilege rdf:resource = privilege/>
  <subject rdf:resource = subject/>
  <conflict resolution rdf:resource = conflict/>
  <defaultAuthorization rdf:resource= default/>
</Policy>
        ← B1, … , Bn,
```

where – *privilege* is either "#read", "#write" or an *S*-variable (*String*-variable),
– *subject* gives a specific user identifier or an *S*-variable,

– *conflict* specifies the enforced conflict resolution mechanism,
– *default* indicates the applied default authorization which is either "#closed" or "#open", and
– B_i is an XML expression or an XML constraint restricting a certain condition on the specified policy. ❑

The proposed policy formalization permits multiple policies to be defined in a single application, thus enabling different policies and conflict resolutions to be enforced on different types of access, users or groups of users.

Example 3 Fig. 7-a depicts a policy specification example and Fig. 7-b models the semantics of the denial_take_precedence resolution method as well as the open and closed policies. ❑

```
C1: <Authorization>                                  // C1 states that users of the group
        <type rdf:resource="#cascade"/>              // ProjectXDoctor are granted a
        <privilege rdf:resource="#write"/>           // privilege on the document
        <sign rdf:resource="#plus"/>                 // patient.xml to write an illness-
        <subject rdf:resource=$S:user/>              // element occurring as a child of a
        <object>                                     // patient having the project code X. In
            <Object>                                 // addition, it states that the
                <level rdf:resource="#instance"/>    // authorization also cascades to all
                <target rdf:resource="patients.xml"/>// of the illness's child nodes.
                <path>                               //     In particular, C1's head
                    <patient project="X" $P:patient> // describes the Authorization-
                        <illness $P:illness/>        // element, while C1's body restricts
                    </patient>                       // it to only users of the group
                </path>                              // ProjectXDoctor.
            </Object>
        </object>
    </Authorization>
        ← <User rdf:about=$S:user>
            <group rdf:resource="#ProjectXDoctor"/>
            $E:userProperties
          </User>.

C2: <Authorization>                                  // C2 defines a read authorization for
        <type rdf:resource="#cascade"/>              // a patient-element of the document
        <privilege rdf:resource="#read"/>            // patients.xml to all users of the group
        <sign rdf:resource="#plus"/>                 // ProjectXDoctor. The cascade option
        <subject rdf:resource=$S:user/>              // expresses that the given
        <object>                                     // authorization must be propagated
            <Object>                                 // downward to all of the patient's
                <level rdf:resource="#instance"/>    // child nodes.
                <target rdf:resource="patients.xml"/>
                <path>
                    <patient $P:patient/>
                </path>
            </Object>
        </object>
    </Authorization>
        ← <User rdf:about=$S:user>
            <group rdf:resource="#ProjectXDoctor"/>
            $E:userProperties
          </User>.
```

Fig. 5. Examples of access authorizations represented by the XML clauses C_1 and C_2.

3.4 Authorization Database

Definition 4 An *Authorization Database* comprising a set of authorizations imposed on a particular XML document database is represented as an XDD description. It consists of XML unit and non-unit clauses formalizing an authorization schema and user ontology as well as access control, policies and conflict resolution rules. ❑

3.5 Access Query and Evaluation

Definition 5 An *authorization query*, which simply returns all (possibly conflicting) authorizations in the database satisfying the specified criteria for selecting target objects, subjects and privileges, is expressed as an XML clause. Its head represents the pattern of the resulting authorizations, while its body describes the authorizations to be selected as well as query constraints. ❑

```
C3: <Authorization>                                          // C3 formalizes propagation of
      <type rdf:resource="#derived_from_cascade"/>           // Authorization-elements of the type
      <object>                                               // cascade along the document
        <Object>                                             // structure. Its body represents an
          <path>                                             // Authorization specified for a nodeX
            <$I:any>                                         // of the type cascade or
              <$N:nodeX $P:Xattributes>                      // derived_from_cascade. Its head then
                <$N:nodeY $P:Yattributes/>                   // specifies that an Authorization of
              </$N:nodeX>                                    // the type derived_from_cascade can
            </$I:any>                                        // be obtained for any child of such
          </path>                                            // nodeX.
          $E:objectProperties                                //
        </Object>                                            //     Recall that an I-variable is
      </object>                                              // used to represent an element
      $E:authorizationProperties                             // when its structure, tag names,
    </Authorization>                                         // attribute lists and nesting patterns
  ←   <Authorization>                                        // are unknown. Thus, the $I:any-
        <type rdf:resource=$S:type/>                         // element in C3's body represents
        <object>                                             // any XML element with a sub-
          <Object>                                           // element $N:nodeX as one of its
            <path>                                           // leaf nodes, while that in C3's
              <$I:any>                                       // head specify an element with a
                <$N:nodeX $P:Xattributes/>                   // similar structure but contains a
              </$I:any>                                      // sub-element $N:nodeY as a child
            </path>                                          // of $N:nodeX and as a leaf node.
            $E:objectProperties
          </Object>
        </object>
        $E:authorizationProperties
      </Authorization>,
      member($S:type, ["#cascade", "#derived_from_cascade"]).
```

Fig. 6. Modeling the semantics of cascade authorizations.

Definition 6 An *access evaluation query* is a special type of query which answers a decision (grant or deny) to a given access request by retrieving authorizations specified for such an access from the authorization database and applying an appropriate policy to resolve conflicts or to yield a default permission. It is formalized as the clause Q of Fig. 8. ❑

```
C4: <Policy>                                          // C4 states that a
        <privilege rdf:resource="#write"/>            // denial_take_precedence
        <subject rdf:resource=$S:user/>               // conflict resolution and a
        <conflictResolution rdf:resource="#denial_take_precedence"/>
                                                      // closed policy will be
        <defaultAuthorization rdf:resource="#closed"/>// enforced on any write
    </Policy>                                         // access requested by a user
        ← <User rdf:about=$S:user>                    // of the group ProjectXDoctor.
              <group rdf:resource="#ProjectXDoctor"/>
              $E:userProperties
          </User>.
```

(a) A policy specification example represented by the XML clause C_4.

```
C5: <PolicyApplication>                               // C5 indicates that given a
        <conflictResolution rdf:resource="#denial_take_precedence"/>
                                                      // denial_take_precedence
        <defaultAuthorization rdf:resource=$S:any/>   // policy, if at least a
        <grantAuthorizations>$E:grantSet</grantAuthorizations>
                                                      // negative authorization for
        <denialAuthorizations>$E:denialSet</denialAuthorizations>
                                                      // a requested access can be
        <decision rdf:resource="#deny"/>              // derived, the decision is to
    </PolicyApplication>                              // forbid such an access.
        ← notEmptySet($E:denialSet).

C7: <PolicyApplication>                               // C6 specifies that for a
        <conflictResolution rdf:resource="#denial_take_precedence"/>
                                                      // denial_take_precedence
        <defaultAuthorization rdf:resource=$S:any/>   // policy if there exists a
        <grantAuthorizations>$E:grantSet</grantAuthorizations>
                                                      // positive authorization but
        <denialAuthorizations>$E:denialSet</denialAuthorizations>
                                                      // not a negative one, the
        <decision rdf:resource="#grant"/>             // requested access is
    </PolicyApplication>                              // authorized.
        ← emptySet($E:denialSet), notEmptySet($E:grantSet).

C8: <PolicyApplication>                               // C7 describes that when no
        <conflictResolution rdf:resource=$S:any/>     // positive or negative
        <defaultAuthorization rdf:resource="#closed"/>// authorization is specified
        <grantAuthorizations>$E:grantSet</grantAuthorizations>
                                                      // for a requested access and
        <denialAuthorizations>$E:denialSet</denialAuthorizations>
                                                      // the closed policy is
        <decision rdf:resource="#deny"/>              // applied, the submitted
    </PolicyApplication>                              // request is denied.
        ← emptySet($E:denialSet), emptySet($E:grantSet).

C9: <PolicyApplication>                               // By contrast to C7, C8
        <conflictResolution rdf:resource=$S:any/>     // defines that if the open
        <defaultAuthorization rdf:resource="#open"/>  // policy is applied, the
        <grantAuthorizations>$E:grantSet</grantAuthorizations>
                                                      // submitted request is
        <denialAuthorizations>$E:denialSet</denialAuthorizations>
                                                      // permitted.
        <decision rdf:resource="#grant"/>
    </PolicyApplication>
        ← emptySet($E:denialSet), emptySet($E:grantSet).
```

(b) Modeling the semantics of the denial_take_precedence, open and closed policies given that $E:grantSet and $E:denialSet respectively represents the set of positive and negative authorizations derived from an authorization database for a requested access.

Fig. 7. Policy definitions.

Intuitively, the clause's head will be matched with a submitted AccessEvaluation-expression which uniquely identifies a target object, a subject and an access privilege. The clause's body which declaratively describes the evaluation of the access decision will then compute whether to grant or deny the requested access by appropriate instantiation of the variable $S:decision.

```
Q: <AccessEvaluation>
       <privilege rdf:resource=$S:privilege/>
       <subject rdf:resource=$S:user/>
       <object><Object>
              <target rdf:resource=$S:xdoc/>
              <path>$E:path</path>
       </Object></object>
       <decision rdf:resource=$S:decision/>
   </AccessEvaluation>
   ←  <xdd:SetOf>                                          // The variable $E:grantSet
          <xdd:Set>$E:grantSet</xdd:Set>                   // represents the set of
          <xdd:Pattern>                                    // Authorization-elements which
              <Authorization>                              // can be derived from a
                  <type rdf:resource=$S:any/>              // database and grant access to
                  <privilege rdf:resource=$S:privilege/>   // the given request.
                  <sign rdf:resource="#plus"/>
                  <subject rdf:resource=$S:user/>
                  <object><Object>
                         <target rdf:resource=$S:xdoc/>
                         <path>$E:path</path>
                  </Object></object>
              </Authorization>
          </xdd:Pattern>
       </xdd:SetOf>,
       <xdd:SetOf>                                         // By contrast to $E:grantSet,
          <xdd:Set>$E:denialSet</xdd:Set>                  // the variable $E:denialSet
          <xdd:Pattern>                                    // represents the set of
              <Authorization>                              // Authorization-elements which
                  <type rdf:resource=$S:any/>              // can be derived from a
                  <privilege rdf:resource=$S:privilege/>   // database and deny the given
                  <sign rdf:resource="#minus"/>            // access request.
                  <subject rdf:resource=$S:user/>
                  <object><Object>
                         <target rdf:resource=$S:xdoc/>
                         <path>$E:path</path>
                  </Object></object>
              </Authorization>
          </xdd:Pattern>
       </xdd:SetOf>,
       <Policy>                                            // An appropriate Policy
          <privilege rdf:resource=$S:privilege/>           // specified for the requested
          <subject rdf:resource=$S:user/>                  // privilege and user is queried,
          <conflictResolution rdf:resource=$S:conf/>       // where $S:conf represents the
          <defaultAuthorization rdf:resource=$S:default/>  // conflict resolution and
       </Policy>,                                          // $S:default the default policy
                                                           // to be applied.
       <PolicyApplication>                                 // PolicyApplication computes an
         <conflictResolution rdf:resource=$S:conf/>        // access decision ($S:decision)
         <defaultAuthorization rdf:resource=$S:default/>   // according to the obtained
         <grantAuthorizations>$E:grantSet</grantAuthorizations>   // $E:grantSet, $E:denialSet and
         <denialAuthorizations>$E:denialSet</denialAuthorizations>// the applied policy.
         <decision rdf:resource=$S:decision/>
       </PolicyApplication>.
```

Fig. 8. An XML clause Q representing an access evaluation query.

Example 4 With reference to Examples 2 and 3, let $ADB = P \cup \{C_4, \ldots, C_8, Q\}$ be an XDD description which models a simple authorization database of a hospital information application. Fig. 9-a illustrates an access evaluation request R_1 for the user

john to write the element /patient[code="p2305" project="X"]/address of the document patients.xml. Although the meaning of the database ADB does not include (either a positive or negative) authorization for writing the target element, by means of the default policy mechanism, Fig. 9-b derives that such a request is denied. Fig. 9-c briefly illustrates the computation process by successively and equivalently transforming the request R into the answer $R_.$. Due to page limitation, its detailed transformation steps are omitted. ❑

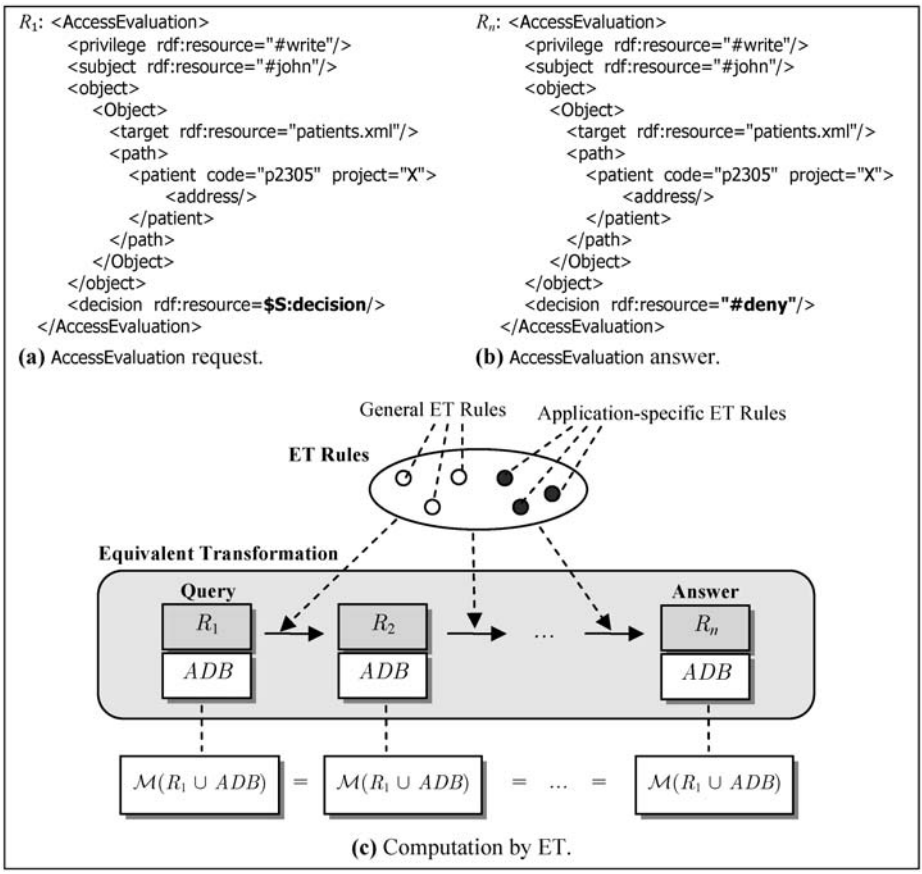

Fig. 9. An access evaluation example.

4 Related Work

Recent contributions to access control on XML documents focus on policies [3, 4,6,11] and cover different aspects of access control, such as user groups, document location in the web, override policies and access control for fragments of XML documents. In *XML Access Control Language* (*XACL*) [11], a *provisional authorization* is defined as a rule stating that a user request will be authorized provided that

certain security actions must be performed prior to the authorization. Bertino et al. [3] and Damiani et al. [6] have developed a behavioral model, while Bertino et al. [4] introduced *χ-Sec* as an XML-based language for specifying subject credentials and security policies and for organizing them into subject profiles and policy bases, respectively. The language is complemented by a set of subscription-based schemes for accessing distributed Web documents, which rely on defined XML subject profiles and XML policy bases.

The work by IBM on *Trust Policy Language* (*TPL*) [8] is devoted to the enforcement of an XML-based framework for specifying and managing role-based access control in a distributed context where the involved parties are characterized by credentials, and digital certificates are used for party authentication. It has also been extended for mapping subject certificates to a role based on certain predefined policies as well as the roles of the certificates' issuers [9].

References [3,4,6,11] underlie a recent standardization effort, organized under OASIS XACML Technical Committee [13], on the use of XML-based languages to express and interchange access control policies among Web applications. Recently, OASIS has announced XACML as a standard XML-based policy specification [7] for fine-grained control of authorized activities, characteristics of the access requestor as well as the protocol over which the request is made. It contains an access control policy language and a request/response language.

Important differences among these approaches to authorization definitions are:

1. XACML, Bertino's model and Damiani's model support both schema and instance level authorizations, while XACL supports only instance level authorizations.
2. In XACML, a subject is specified by a user identity (e.g. ID, group and role) together with some particular properties (e.g., age, gender, domain and request time), while in XACL and Bertino's model, it is represented by merely a user identity. On the other hand, Damiani's model represents a subject by user-id, group and location patterns (e.g., 151.100.*.*, *.it.com etc).
3. The conflict resolution mechanism of XACML and XACL is mainly based on denial-take-precedence and permission-take-precedence where order of decision rules appearing in the access policies is significant for policy decision. The mechanism employed by Bertino's model includes instance-take-precedence, descendant-take-precedence and denial-take-precedence, while that of Daminai's model is based on most-specific-take-precedence (e.g., smith.it.com is more specific than *.it.com) and denial-take-precedence.

5 Conclusions

This paper has developed a simple yet flexible approach to XML access control by means of an XML-based rule language. It covers and extends the significant functionalities supported by the existing approaches with the facilities to define application-specific subject ontologies, to constrain and derive authorizations at both schema and instance levels as well as to enforce multiple user-defined policies, while still possessing well-defined declarative semantics, flexibility and interchangeability. A prototype system is currently implemented using XET [2].

In addition to an access evaluation query which returns a decision whether the requested access should be authorized or not, various application domains demand also

an ability to compute a *view* of each user on a particular document. Basically, such a view presents only parts of the requested document that the user is authorized to read, while preserving the document structure. Extension of the model by a view computing mechanism is underway. Furthermore, since there might exist multiple conflict resolution policies that can be enforced on a particular access request, provision of a mechanism for policy prioritization and overriding is essential. This is also part of an ongoing research work.

References

1. Akama, K., Shimitsu, T., Miyamoto, E.: Solving Problems by Equivalent Transformation of Declarative Programs. *Journal of the Japanese Society of Artificial Intelligence*, Vol. 13, No. 6 (1998) 944–952 (in Japanese)
2. Anutariya, C., Wuwongse, V. and Wattanapailin, V.: An Equivalent-Transformation-Based XML Rule Language. *Proc. Int'l Workshop Rule Markup Languages for Business Rules in the Semantic Web*, Sardinia, Italy (2002)
3. Bertino, E., Castano, S., Ferrari, S. and Mesiti, M.: Specifying and Enforcing Access Control Policies for XML Document Sources. *World Wide Web*, Vol. 3, No. 3, Baltzer Science Publishers, Netherlands (2000)
4. Bertino, E., Castano, S. and Ferrari., E.: On specifying security policies for web documents with an XML-based language. *Proc. 6th ACM Symposium on Access control models and technologies*, ACM Press (2001) 57–65
5. Brickley, D. and Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. *W3C Working Draft* (Jan. 2003) http://www.w3.org/TR/rdf-schema/
6. Damiani, E., Vimercati, S., Paraboschi, S. and Samarati, P.: A Fine-Grained Access Control System for XML Documents. *ACM Transaction on Information and System Security*, Vol. 5, No. 2, (May 2002) 169–202
7. Godik, S. and Moses, T.: XACML 1.0, *OASIS Standard* (18 Feb. 2003) http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf
8. Herzberg, A., Mass. Y.: Relying Party Credentials Framework. *Proc. RSA Conference*, San Francisco, CA (Apr. 2001)
9. Herzberg, A., Mass, Y., Mihaeli. J.: Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. *Proc. IEEE Symposium Security and Privacy*, CA, (2000)
10. Jajodia, S., Samarati, P. Subrahmanian, V. S., Bertino. E.: A unified framework for enforcing multiple access control policies. *Proc. 1997 ACM SIGMOD: Int'l Conf. Management of data*, Arizona, (1997) 474–485
11. Kudo M. and Hada. S.: XML Document Security based on Provisional Authorization. *Proc. 7th ACM Conf. Computer and Communications Security*, Greece, (2000) 87–96
12. Lassila, O. and Swick, R.R.: Resource Description Framework (RDF) Model and Syntax Specification. *W3C Recommendation* (Feb. 1999) http://www.w3.org/TR/REC-rdf-syntax/
13. OASIS XACML Technical Committee. http://www.oasis-open.org/committees/xacml/
14. Wuwongse, W., Akama, K., Anutariya, C. and Nantajeewarawat, E.: A Data Model for XML Databases. *J. Intelligent Information Systems*, Vol.20, Issue 1, Kluwer Academic Publishers (2003) 63–80
15. Wuwongse, W., Anutariya, C., Akama, K. and Nantajeewarawat, E.: XML Declarative Description (XDD): A Language for the Semantic Web. *IEEE Intelligent Systems*, Vol. 16, No. 3 (2001) 54–65

Inference of Reactive Rules from Dependency Models

Asaf Adi, Opher Etzion, Dagan Gilat, and Guy Sharon

IBM, Haifa Research Laboratory
31905, Haifa, Israel
{adi,opher,dagang,guysh}@il.ibm.com

Abstract. Reactive rules are rules that specify reactions to events. In some cases it is easier and more intuitive for users to define a dependency model representing an ontology. In this paper, we introduce the ADI model and its inference capabilities to run-time rule execution. We introduce a case study on eTrade and define the model building blocks exemplified by this case study. Then we show the specific rule language that is being used as the execution infrastructure. We explain the inference mechanism and its dynamic nature. The paper concludes with related work and a discussion about its utilization.

1 Introduction

Enterprise systems contain many dependencies: value dependencies, causal dependencies, and business logic dependencies, to name a few. Different types of methodologies and tools handle each type of dependency [7].
Some examples of dependencies are:

1. The amount of communication bandwidth for a client is a function of the amount of available bandwidth and the type of service level agreement. Any change in one of these components requires the re-calculation of the allocated bandwidth for the specific client.
2. When a DB server space utilization exceeds the threshold of 90 percent, the database will most probably crash for every update transaction within 30 minutes during working hours, and within 90 minutes outside working hours.
3. When at least four power-generators should be active on site.
4. When a data-item that is described in inches needs to be translated to centimeters by multiplying the value by 2.54, and replacing the constant “in” with the constant “cm”.

Those dependencies are quite different from one another, but they are all common in business systems.

- Dependency 1 is a value dependency that uses some mathematical functions (like in a spreadsheet) [9].
- Dependency 2 is a temporal causal dependency based on probability. It has a predictive nature [11,12].
- Dependency 3 is a business logic dependency; it imposes a constraint that must be kept as a “law of nature”, to guarantee the safe functionality of the system [4,6].

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 49–64, 2003.

- Dependency 4 is a translation dependency; it defines a transformation rule between a data source (e.g., in the USA) and a data warehouse (e.g., in Europe).

Dependencies can also be mixed and transitive. For example, a database crash in one of the database replicas that is predicted by dependency 2 can impact the amount of available effective bandwidth and will require re-calculation of some of the users' allocation.

Furthermore, dependency 2 has three outstanding properties:

1. It is stochastic
2. It has temporal validity
3. It is context-sensitive: The dependency behaves differently in different contexts (working hours).

Currently, dependencies are dealt with using a variety of ad hoc tools, which makes it difficult to have a single seamless view of all the dependencies in a business system, and, consequently, to take the appropriate action. Today, a data integration system may receive information from data sources, may use a spreadsheet (periodically) to calculate functions, or invoke an ad-hoc program. Business logic languages are written as stand alone tool that is not easily integrated with the rest of the systems, and nobody can see the entire picture and estimate what will happen when a power supply goes down.

1.1 The Vision

Silbershatz et al [17] pointed out the problem of dependency-oriented correlation as one of the major challenges for the 21st century. The example used in that context was coordination among different contractors constructing a building together, making sure that a change in the specification of an elevator will be consistent with the electricity layout of the building. IT systems within a single organization have become very complex in terms of the number and types of dependencies.

The vision is to try to reduce this complexity by enabling the following properties:

1. Supporting a seamless, easy to use model of expressing context-sensitive dependencies and attaching to it dependency discovery modules where applicable.
2. Supporting a model that provides a multi-level (hierarchical) view of all the transitive dependencies.
3. Supplying an intelligent component that infers rules (including actions) from the dependencies graph.
4. Supplying a dependency visualization tool.

The idea (in a somewhat simplistic manner) is that dependencies are considered as universal laws. Whenever such a law is violated, by any transformation of the current state, the system should reflect upon its own knowledge about the nature of the dependency, and its own context, and determine an action that will restore the stability of the system with respect to its laws [8,10]. Such an action may be simple (e.g., sending an update operation to some database; however, the type of update may be context-sensitive), or complex (e.g., activating a load balancing algorithm, and reassigning all jobs according to its results). We are using AMIT (Active Middleware Technology) and its run-time rule engine ("the situation manager") as an execution mechanism. Rules are automatically and dynamically inferred from the model.

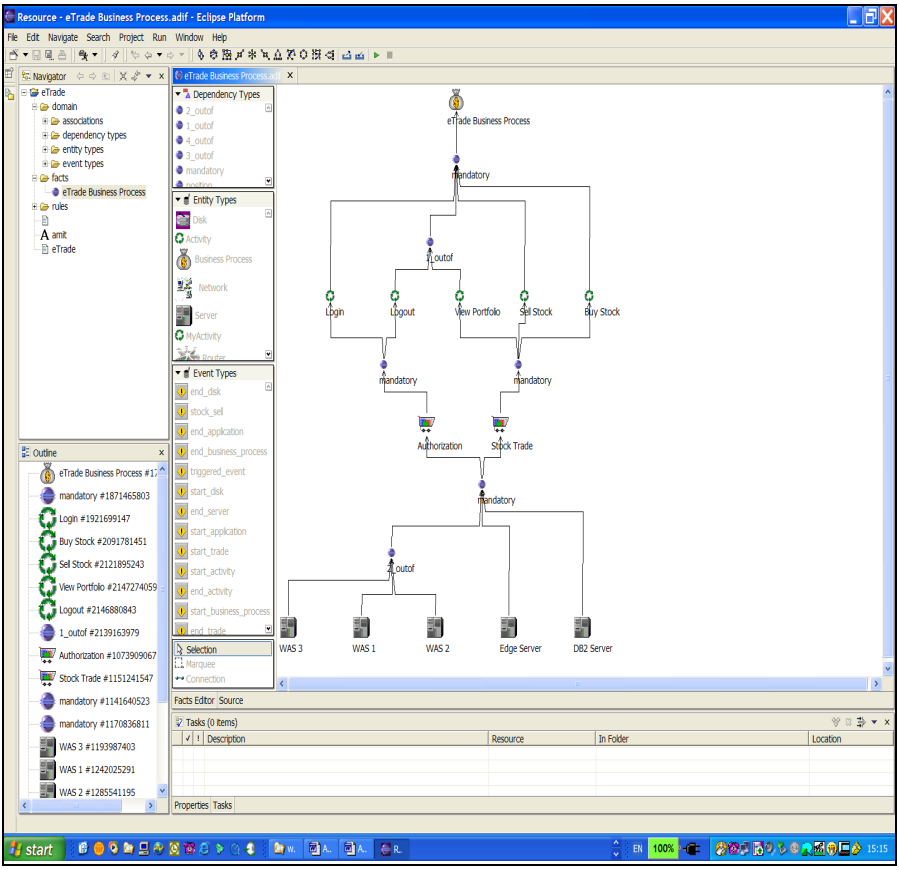

Fig. 1. The eTrade Business Process

1.2 The eTrade Case Study

An eTrading site, run by XInc, enables customers to buy and sell stocks. The customers pay a commission for each trading operation (buying or selling of stocks).
Figure 1 shows the eTrading site topology. The site has one business process – the eTrade business process. The eTrade business process is composed of 5 tasks: Login, Logout, View Portfolio, Buy Stock, and Sell Stock.

The most important tasks for the business process (Login, Buy Stock, Sell Stock) have a mandatory dependency with respect to the process, which means that if at least one of these tasks fail, the entire business process fails.

The View Portfolio and the Logout tasks are also important to the eTrade business process, but it will still function if one of these tasks fails.

We can state that the *eTrade business process will be OK if the Login task is OK and the Buy Stock task is OK and the Sell stock task is OK and at least one of the View portfolio task and the Logout task are OK.*

The tasks are dependent upon applications, which are the Authorization application and the Stock Trade application. The applications are in turn dependent upon servers in order to function.

1.3 Structure of This Paper

Section 2 describes the dependency model language (ADI), Section 3 describes the reaction rule languages to which it refers (AMIT), Section 4 describes the inference from the model language to the rule language, Section 5 discusses related work, and Section 6 concludes the paper.

2 ADI Model Language

The ADI model language models an ontology, its building blocks, and the dependencies among them. In the case study defined in Section 1, the building blocks are the enterprise systems, solutions, and services. It enables the modeling of various entities and business components (e.g. event, disk, application, activity, business process), the information that is associated with them (i.e. schema), and the semantic relationships among them (e.g. dependency between a business component and other business components and events).

2.1 Domain

ADI is domain independent framework and does not have predefined domain elements. Consequently, the first stage in the ADI deployment process is defining the domain elements according to the ontology it models. Three types of domain elements can be defined: event types, entity types and dependency types.

2.2 Event Type

An event type describes the common properties of a similar set of events. It defines the type of information that can be associated with the event (attributes) and relationships between events (an event model).

An event type has the following properties:

1. *An event type name* identifies the event type. It must be unique with respect to the set of all event type names.
2. *A set of super types* describes the type's direct ancestors. An event type inherits the attributes and methods of its super types.
3. *A set of attributes* describes the information that is associated with the event. An event attribute maps an event type to either an object class or to a collection of classes that are not necessarily events; the latter class describes the schema of the event type. Each attribute has a unique name, a type, a default value, and restrictions. These properties include:

a. *An attribute name* identifies the attribute. It must be unique with respect to the set of all attribute names defined in the event type.
b. *An attribute type* determines the attribute's possible values. It can be either a primitive attribute or a reference to an object. Primitive attributes include number, string, boolean, and chronon. The value of a reference attribute, in contrast to that of a primitive type, is a reference to an object that is the actual value of the attribute. This object can be an event instance or any other data structure.
c. *A default value* determines the attribute's value if it is not reported with the event. This is a complex expression that may involve other event attributes and external information (a database is an example). A default value can be used to define derived attributes
d. *Restriction*s define the attribute's possible values by limiting them to a specific range. An example is an event reporting on CPU utilization that has three attributes: an attribute of type integer that must be in the range 0 to 100 (utilization percent), an attribute of type string that reports and that has exactly three letters (CPU identifier), and an attribute of type string that can be either 'ok' or 'fail' (CPU status).

After instantiation, the information that is associated with the event is either reported by an event source, or determined by a default value, as immutable. This is because an event, and the information associated with it, describes something that happens instantaneously, or in an instantaneous step (the start of an activity). Such information does not change after the event's occurrence.

The event start activity is signaled when a user starts an activity (e.g. login, purchase, logout) within an e-trading session.

```
<eventType name="start_activity" >
        <attributeType name="Own_ID" type="integer" />
        <attributeType name="Session_ID" type="integer"/>
</eventType>
```

Fig. 2. Event type

2.3 Entity Type

An entity type describes the structure of a similar set of entities; its syntactic definition is similar to that of an event type. However, there are some differences between entity types and event types:

1. *An entity type has key attributes;* it is a set of one or more attributes, which identify uniquely an entity of the type.
2. *An entity type has a composes* property that designates a weak/strong entity relationship. A "weak" entity obtains the key attributes of a "strong" entity. The *composes* relationship is required when events refer to the strong entity type as the identifier of the weak entity type (e.g. in an event reporting on failure of disk d:/ of a specific server, the disk is a weak entity while the server is a strong entity). This

relationship's semantics is similar to the composition relationship semantics in data modeling.

3. *After instantiation, the information that is associated with the entity may change* (in contrast to that of an event). Entity information changes when an event reports on changes in the entity's state (e.g. disk utilization is 90%); the *effect* element expresses this phenomena.

Entities can model elements at any level of the enterprise: resource level (e.g. disk, database, and server), application level (e.g. login utility and SLA compliance monitoring utility), the task level (e.g. purchase activity), business process level (e.g. e-trading), and others.

```
<entityType name="resource" >
        <attributeType name="STATE" xsi:type="string">
                <enumeration value="state_ok" />
                <enumeration value="state_warning" />
                <enumeration value="state_problem" />
                <enumeration value="state_fail" />
        </attributeType>
</entityType>

<entityType name="server" extends="resource" >
        <attributeType name="serialNum" type="integer"
                isKey="true"/>
        <attributeType name="description" type="string"
                defaultValue="Server"  />
        </attributeType>
</entityType>

<entityType name="disk" extends="resource" clingTo="server" >
        <attributeType name="label" type="integer"
                isKey="true"/>
</entityType>
```

Fig. 3. Entity type

2.4 Dependency Type

A dependency is a semantic relationship among entities that designates in what manner an entity (the target entity) depends on other entities (the source entities). An example is a server with a RAID of three disks on which an application is running. In this case, the application depends on the server (the application is the target entity and the server is the source entity), which itself depends on the disks (the server is the target entity and the disks are the source entities). The semantics of the dependency (e.g. the server requires all the disks in the raid or only two out of the three disks) are defined by the dependency type.

A dependency type has the following properties:

1. *A dependency type name* identifies the dependency type. It must be unique with respect to the set of all dependency type names.

2. *A set of report attribute types* designates the information modified by the dependency at the target node.
3. *A set of receive attribute types* designates the information propagated by the dependency from the source nodes.
4. *A situation rule template* designates the semantics of the dependency.

```
<dependencyType name="mandatory">
        <reportAttributeType name="STATE" xsi:type="string" />
        <reportAttributeType name="path" xsi:type="string" />
        <receiveAttributeType name="STATE" xsi:type="string" />
</dependencyType>
```

Fig. 4. Dependency type

2.5 Effect

Effect definitions are used to describe how an event may have an effect on an entity, by creating or deleting and entity or by updating an entity's data, and how an entity may have an effect on an event, by triggering it.

An effect has the following properties:

1. There are five different effect types:
   a. *Update data* type — updates the entity's attribute values when the event is detected.
   b. *Create entity* type — creates a new entity when the event is detected.
   c. *Delete entity* type — deletes the entity when the event is detected.
   d. *Create/Update entity* type — creates a new entity if the entity does not exist and updates the entity's attribute values if it does exist.
   e. *Trigger event* type — triggers an event when at least one entity attribute value was updated.
2. *A condition* can be applied to the effect by creating an expression that can include the event and entity attributes. Only when the condition is satisfied will the effect be processed.
3. *Derivations* are used to set the affected attributes' values. This could be a simple mapping of two attributes one from the entity and the other from the event, meaning that the entity attribute will be set by the value of the event attribute. But it could be more complicated, by setting an attribute with the value of an expression that can include both event and entity attributes. For example, an entity has a *state* attribute of type integer [0..1] and the event has a *state* attribute of type string [fail..ok]. An effect describing how the event updates the entity will use a derivation for transforming from one form to the other.

```
<effect condition="update_activity.Session_ID=Activity.Session_ID"
      entityType="Activity" eventType="update_activity"
      effectType="UPDATE_DATA">
          <derivation name="state"
               expression="expression(update_activity.state)"/>
          <derivation name="time" expression="detectionTime"/>
</effect>
```

Fig. 5. Update data effect

2.6 Fact

A fact is the actual model being monitored by ADI. It consists of entities and dependencies between these entities. The entity part of a fact is a declaration of entity instances and each such entity has the following properties:

1. The entity type of which this entity is an instance.
2. Values for the entity's attributes. Attributes that are part of the entity's key attribute set must be given a value.

   A dependency has the following properties:

1. A Unique name.
2. The type of the dependency, i.e. its semantics.
3. A list of targets that reference either entities (by their key attribute value) or other dependencies (by their name). Enabling a dependency to reference another dependency increases the expressional power of the language and enables it to model more complex dependencies called compound dependencies.
   1. Derivations can be added to each target for mapping the report attribute values of the dependency to the entity attributes referenced by the target.
4. A list of sources that reference either entities or other dependencies. Each source must set the value for the receiving attributes of the dependency by using derivations.

3 Situation Manager Rule Language

This section briefly describes the situation manager, the rule infrastructure within which it carries out the execution that is defined by the ADI model.

```
<entity itemType="Activity" name="Buy Stock">
        <attribute name="Name" value="Buy Stock"/>
        <attribute name="Own_ID" value="10"/>
</entity>
<entity itemType="Application" name="Stock Trade">
        <attribute name="Name" value="Stock Trade"/>
        <attribute name="Own_ID" value="6"/>
</entity>
<factDependency dependencyType="mandatory" id="1">
         <factDependencyTarget entityType="Activity"
                              xsi:type="entityRefType">
                    <keyAttribute name="Own_ID" value="10"/>
         </factDependencyTarget>
        <factDependencySource entityType="Application"
                              xsi:type="entityRefType">
                    <keyAttribute name="Own_ID" value="6"/>
        </factDependencySource>
</factDependency>
```

Fig. 6. Fact: two entities and a dependency between them

The situation manager is a tool that includes both a language and an efficient runtime execution mechanism, aimed at reducing the complexity of active applications. It follows the observation that in many cases, there is a gap between current tools that enable an entity to react to a single event (following the ECA: Event-Condition-Action paradigm), and the reality, in which a single event may not require any reaction, but the reaction should be given to patterns over the event history. The concept of the situation presented in [2], extends the concept of a composite event [13, 5, 18], in its expressive power, flexibility, and usability.

A situation is a semantic concept in the customer's domain of discourse; its syntactic equivalent is a (possibly complex) pattern over the event history. Some examples, from various contexts, of situations that need to be handled are presented in figure 7. A situation manager rule definition consists of four parts: event collation, context, situation rule, and triggering expression; they are further discussed in the squeal.

- When client wishes to activate an automatic "buy or sell" program, and a security that is traded in two stock markets has a difference of more than five percent between its values in the markets, such that the time difference between the reported values is less than five minutes ("arbitrage").
- A customer relationship manager wishes to receive an alert, if a customer's request was reassigned at least three times.
- A groupware user wishes to start a session when there are ten members of the group logged in to the groupware server.
- A network manager whishes to receive an alert, if the probability that the network will be overloaded in the next hour is high.

Fig. 7. Situations

3.1 Event Collection

An event collection designates the events that are considered for situation detection. These events, denoted *candidates*, are partitioned into candidate lists. A candidate list is a collection of events (candidates) that:

- belong to the same event class
- have the same role in situation detection (e.g. in a situation that identifies arbitrage deals, events that report about quotes from one stock market have one role, and events that report about quotes from another stock market have another role)
- satisfy the same filtering conditions
- share the same decision possibilities about the reuse of events that participate in situation detection.

3.2 Context

A context [1] is a semantic notion that describes a composite perspective of the environment in which situations take place. It is a combination of subject-specific per-

spectives such as semantic perspective, temporal perspective, state perspective, and spatial perspective.

1. *Semantic* perspective designates environment information about a specific topic; an event group defines it. An event group is a collection of semantically associated event instances that refer to the same entity or concept (e.g. users who are members of the same group).
2. *Temporal* perspective designates environment information within a specific temporal element; it is defined by a lifespan. A lifespan is a temporal interval that is bounded by two events called *initiator* and *terminator*. The occurrence of an initiator event initiates the lifespan and an occurrence of a terminator event terminates it (e.g. network overload in one hour).
3. *State* perspective designates environment information within a specific state; it is defined by a predicate (e.g. low market volume).
4. *Spatial* perspective designates environment information within a specific location or area; it is defined by a space-span. A space-span is a physical region (e.g. a district, an intersection, a valley) in which active behavior is relevant. It designates a collection of events that occur in the same region (e.g. vehicles near a traffic problem).

3.3 A Situation Rule

A situation rule is an expression over event classes that determine situation occurrences as a function of an event collection instance and a context. A situation expression consists of a combination of an event algebra operator and qualifiers, a predicate (applicable for certain operators), and detection mode. The combination of an operator and qualifiers designates an event pattern; the predicate designates a condition over the events in the pattern and results in tuples of event instances that could have caused the situation; and the detection mode determines if a situation can be detected during the context (immediate) or at the end of it (deferred).

1. *Situation expression qualifiers* designate a selection strategy when several candidates exist in a candidate list of an event that is in the domain of a situation expression operator. We denote these events, operands of the situation expression. A qualifier is applied to every operand and has seven possible values: first, last, each, min, max, not, and never.
2. A *situation expression operator* designates an event pattern. The operators are classified into four groups:
   a. joining operators (conjunction, disjunction, sequence, strict sequence, simultaneous, and aggregation)
   b. selection operators (first, until, since, and cross)
   c. assertion operators (never, not, sometimes, last, min, max, and unless)
   d. temporal operators (at, every, and after).

3.4 A Triggering Expression

A triggering expression designated the information reported with the situation. It is a function from the set of events that caused the situation to occur to the situation's attributes. The function may involve event information, aggregation over several events, and context information.

4 Rule Inference from a Model

Each dependency type is translated to a situation rule template. The operands of the definition represent the status of the dependency sources according to their place in the dependency. Upon detection, the situation attribute values represent the status of the dependency targets. The context of the situation starts and ends with a corresponding event and all the definitions are keyed by the dependency id. This enables a single situation definition for all the dependencies of the same type; a rule template.

All definitions representing dependency types are prefixed by "ADI_" and as such, a situation definition looks like *ADI_*<dependency type> a context *ADI_*<dependency type>*_lifespan* and an operand in place *X* in the dependency looks like *ADI_*<dependency type>*_X*. This naming convention allows the rule inference from the dependencies' representation of the rules to a situation's rules using a single situation definition for each type of dependency.

In Figure 8, the dependency type is mandatory and the receiving attribute is *STATE,* which is used in influencing the max variable of the report situation. The reporting attribute is also *STATE* and its value is an expression on the max variable value of the situation when detected.

When ADI receives a model, first, it creates all the entities with their attribute values and then, for each dependency, it executes the following creation procedure:

1. ADI creates a new dependency with a unique identifier.
2. The lifespan (context) of the situation inferred by the new dependency is started. This occurs when ADI creates the event *ADI_*<dependency type>*_start* with the id of the dependency as the value for the event's attribute *dependency_id.*
3. ADI adds all the targets to the dependency by identifying the entities they represent using the values for the entities' key attributes. This way, when it detects the situation, it can update the entities as targets with the situation attributes.
4. ADI adds all the sources to the dependency in the same way as the targets are with the difference that it remembers their position in the dependency, i.e. 1 to n. For each source, the following event is created: *ADI_*<dependency type>*_X* where *X* stands for the position of the source. The attributes of the event are *dependency_id* and all the necessary attributes needed for the situation, which we call *receiving attributes*. The value for the first mentioned attribute is the dependency's id and the rest are calculated using the derivations declared in the dependency definition.
5. Finally, after acquiring all the operands' data, ADI can start to detect the situation by creating the event *ADI_*<dependency type>*_initiator* with the id of the dependency as the value for the event's attribute *dependency_id*.

```
<situation lifespan="ADI_mandatory_lifespan" name="ADI_mandatory">
  <report detectionMode="immediate" repeatMode="always"
          where="sum&amp;gt;=1000">
    <operandReport eventType="ADI_mandatory_1"
                   max=expression(STATE) override="true"
                   partMax="true" quantifier="last" retain="true"/>
    <operandReport eventType="ADI_mandatory_2"
                   max=expression(STATE) override="true"
                   partMax="true" quantifier="last" retain="true"/>
    <operandReport eventType="ADI_mandatory_3"
                   max=expression(STATE) override="true"
                   partMax="true" quantifier="last" retain="true"/>
    <operandReport addToSum="true" as="initiator"
                   eventType="ADI_mandatory_initiator"
                    override="true" quantifier="first" retain="true"
                    sum="1000"/>
  </report>
  <situationAttribute attributeName="dependency_id"
                   expression="key(dependency_id)"/>
  <situationAttribute attributeName="STATE"
                   expression=expression(max)/>
</situation>
```

Fig. 8. Mandatory dependency type semantics

When ADI detects a situation, it retrieves the inferring dependency using the dependency id reported by the situation. It updates all the dependency's targets with the situation's attributes, which we call *reporting attributes*, using the derivations declared in the dependency definition.

Dependencies can be part of a compound dependency by referencing other dependencies – source or target. In this case, it is important to add the dependencies according to their hierarchical position in the compound dependency, starting from root to leaf. The creation procedure is very similar to that which was previously mentioned except for step 5. This step needs to be performed by each dependency only after all the dependencies have performed step 4, but this time in the opposite hierarchical order – from leaf to root. When all the dependencies perform step 4, they are all still in acquisition mode, so when the leaves perform step 5 and situations get detected, the leaves' parents acquire their missing sources' status. Then the parents themselves can perform step 5 and so on until the root is reached. This order of creation will ensure the minimal stabilization time for the model when received by ADI.

Apart from receiving the model, ADI can also receive external events (i.e. those that do not have an "ADI_" prefix). All the effects in which the event can participate are gathered and processed one by one. Each effect is checked against all the entities of the same type for the possible effect of the event on the entity by evaluating the condition expression that can be a function of both the event and entity attribute values. If the result of the condition evaluation is true, the event does have an effect on the entity, resulting in a possible change to the entity's attribute values. The entity attributes are updated using the derivations declared in the effect definition. If there is

a value change in at least one attribute, all the effects of type *TRIGGER_EVENT* with the updated entity are gathered. Their conditions evaluated and for all true evaluations, the event in that effect definition is created with attribute values calculated using the derivations declared in that effect definition.

No matter if the entity attributes were updated or not according to the event, the current status (i.e., attribute values) of the entity is notified to all the dependencies it participates in as source. For each such case, an event is created *ADI_*<dependency type>*_X* where *X* is the place of the entity in that dependency, where the *dependency_id* attribute is the value of that dependency and where the receiving attributes are calculated using the derivations declared in that dependency definition. The rest of the flow of information is the same as the one mentioned for receiving a model in ADI.

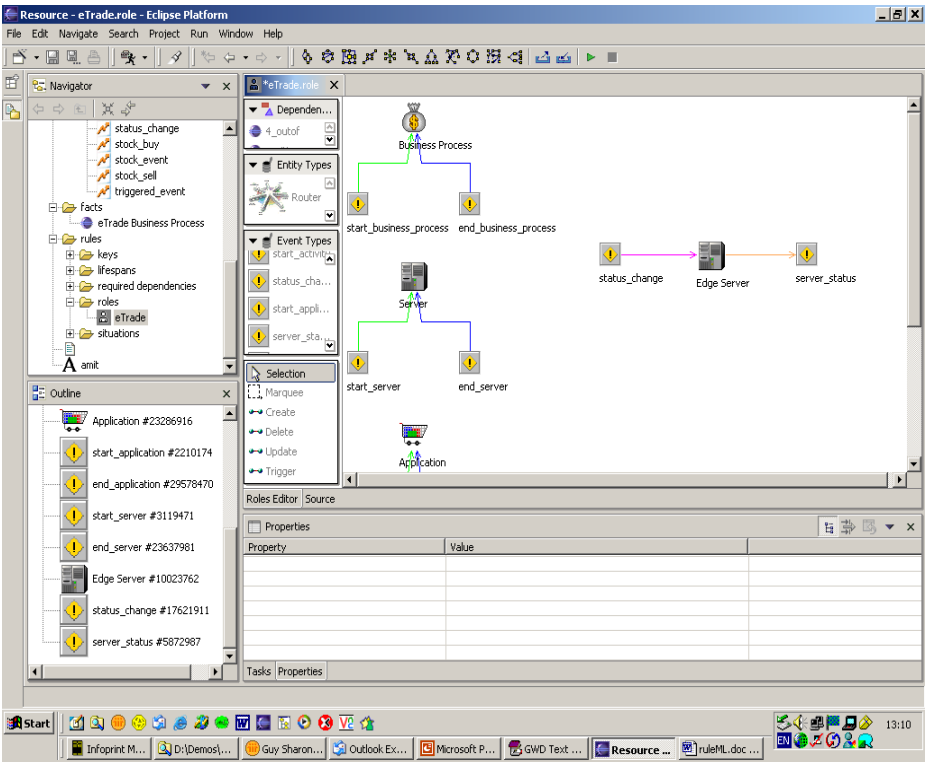

Fig. 9. Update data and trigger event effects

This example shows two effect definitions where, in one, an event can update an entity (purple edge) and, in the other, an entity can trigger an event (orange edge). The assumptions here are that the condition of both effects is **true** and that the derivation in both effects is simple, i.e. in the update effect entity attribute *STATE* equals event attribute *STATE* and in the trigger effect it is vise versa. The sequence of events is as follows:

1. ADI receives the event status_change STATE=fail
2. ADI updates the Edge server STATE attribute with the event and creates:
   a. *ADI_mandatory_2 dependency_id=1 STATE=fail* since it's a source in the mandatory dependency in the compound dependency example. The situation manager will detect a situation and report the event *ADI_mandatory dependency_id=1 STATE=fail* and ADI will update both application with the current state.
   b. *server_status STATE=fail* since one of the server's attributes was changed and there is a trigger effect for this entity.

5 Related Work

Related ADI work can be found both in research and in products available to the general public. The major difference and eventually advantage of ADI is its extensibility feature enabling the modeling in different domains with different dependencies.

Kar et al [14] develops algorithms and processes that can be used to discover dependencies in an e-business environment; represents dependencies using standard modeling tools; and uses dependency information structure to identify line of business views that relate business metrics to monitored resource metrics. There are several differences between Kar's work and ADI:

1. Kar focuses on dependency discovery, whereas discovering dependency is not a feature of ADI.
2. Kar uses standard modeling tools whereas ADI enables the definition of new dependency types and their semantics using situation manager rules. Standard modeling tools define some predetermined dependency types (if any) without specific semantics.
3. Kar does not perform runtime monitoring of the effect of events on business components, and how state change in one business component affects other business components, while ADI does.

One of the three products that are related to ADI is Appilog's PathFinder [3]—a Business Service Management solution that discovers IT assets, maps them to the business processes they support, and then visualizes the effects of infrastructure events. Its differences from ADI are that:

1. PathFinder supports a predetermined set of dependency types whereas ADI enables the definition of new dependency types and their semantics using situation manager rules.
2. PathFinder only monitors the effects of infrastructure events whereas ADI can monitor the effect of events at all levels, i.e. infrastructure events, applications events, business process events, etc.

The second product is Micromuse's Netcool [15], which provides high-level views of services available through its topology maps, allowing the management of the enterprise environment based on geography or other criteria. The differences between Netcool and ADI are that:

1. Netcool supports a predetermined set of relationships whereas ADI enables the definition of new dependency types and their semantics using situation manager rules.
2. Netcool focuses on the business management domain while ADI is aimed for any domain.
3. Netcool models are limited to the IT, applications and services; ADI can model entities in any levels.

The third product is the Managed Objects' Formula [16] that unifies management data by dynamically integrating and consolidating it into a Web-enabled object integration model. It differs from ADI in that:

1. Formula supports a predetermined set of dependency types whereas ADI enables the definition of new dependency types and their semantics using situation manager rules.
2. Formula only monitors the effects of infrastructure events whereas ADI can monitor the effect of events at all levels, i.e. infrastructure events, applications events, business process events, etc.

6 Conclusion

By modeling ontology, building blocks of a model, and the dependencies among those building blocks, it is possible to model the dependencies of a system and monitor the system accordingly. One of the implications of applying ADI in a business process management system is the ability to look at the "whole picture" linking resources to applications, tasks, and processes and monitoring the effects of changes on the entire system. Due to its extensibility, ADI can be applied to any domain where dependencies between entities in the domain need to be modeled, managed and monitored and where effects of changes across multiple hierarchical levels of the model may be too complicated to detect and understand without ADI.

At this time, an ADI model is represented through XML by using the ADI schema. In the near future we intend to supply the capability to model using UML diagrams. Future directions should lead to the modeling, managing, and monitoring of dynamic and temporal dependencies where dependencies between entities might change in time, be true only at particular time slots or can be created, modified, or removed to reflect the underlying system.

References

1. Asaf Adi, Ayelet Biger, David Botzer, Opher Etzion, and Ziva Sommer: Context Awareness in AMIT, Proceedings AMS'2003.
2. Asaf Adi, Opher Etzion: The situation manager rule language. RuleML 2002.
3. Appilog: http://www.appilog.com
4. Jacques Bouman, Jos Trienekens, Van der Zwan: Specification of Service Level Agreements, Clarifying Concepts on the Basis of Practical Research, Proceed. STEP'1999.

5. Sharma Chakravarthy, D. Mishra: Snoop - An Expressive Event Specification Language for Active Databases. DKE 14(1) 1994: 1-26.
6. Costas Courcoubetis, Vasilios A. Siris: Managing and Pricing Service Level Agreements for Differentiated Services, Proceed. IWQoS'1999.
7. Stephane Ducasse, Mireille Blay-Fornarino, Anne-Marie Pinna-Dery: A Reflective Model for First Class Dependencies. OOPSLA 1995: 265-280.
8. Opher Etzion: Active Interdatabase Dependencies Information Sciences Journal, 75, Dec 1993: 133-163.
9. Opher Etzion: The Reflective Approach for Data-Driven Rules. International Journal of Intelligent and Cooperative Information Systems, 2(4), Dec 1993: 399-424.
10. Opher Etzion, Boris Dahav: Patterns of Self-Stabilization in Database Consistency Maintenance. DKE 28(3) 1998: 299-319.
11. Opher Etzion, Avigdor Gal, Arie Segev: Extended Update Functionality in Temporal Databases. Temporal Databases, Springer-Verlag 1998: 56-95.
12. Avigdor Gal, Opher Etzion: A Multiagent Update Process in a Database with Temporal Data Dependencies and Schema Versioning. TKDE 10(1) 1998: 21-37.
13. Narain H. Gehani, H. V. Jagadish, Oded Shmueli: Event Specification in an Active Object-Oriented Database. SIGMOD Conference 1992: 81-90.
14. Gautam Kar, Andrzej Kochut: Managing Virtual Storage Systems: An Approach Using Dependency Analysis. Integrated Network Management 2003: 593-604.
15. Micromuse: http://www.micromuse.com/index.html
16. Managed Objects: http://www.managedobjects.com/
17. Abraham Silberschatz, Michael Stonebraker, Jeffrey D. Ullman: Database Research: Achievements and Opportunities Into the 21st Century. SIGMOD Record 25(1) 1996: 52-63.
18. Detlef Zimmer, Rainer Unland: On the Semantics of Complex Events in Active Database Management Systems. ICDE 1999: 392-399.

Value-Added Metatagging: Ontology and Rule Based Methods for Smarter Metadata

Marek Hatala and Griff Richards

Simon Fraser University Surrey
2400 Central City
Surrey, BC, Canada, V3T 2W1
{mhatala,griff}@sfu.ca

Abstract. In this paper we describe an ontology and rule based system that significantly increases the productivity of those who create metadata, and the quality of the metadata they produce. The system suggests values for metadata elements using a combination of four methods: inheritance, aggregation, content based similarity and ontology-based similarity. Instead of aiming for automated metadata generation we have developed a mechanism for suggesting the most relevant values for a particular metadata field. In addition to generating metadata from standard sources such as object content and user profiles, the system benefits from considering metadata record assemblies, metadata repositories, explicit domain ontologies and inference rules as prime sources for metadata generations. In this paper we first introduce the basic features of metadata systems and provide a typology of metadata records and metadata elements. Next we analyze the source of suggested values for metadata elements and discuss four methods of metadata generation. We discuss how the operations on objects the metadata are describing affect suggested metadata values and we present decision tables for the metadata generation scheduling algorithm. Finally, we discuss the use of our system in tools developed for creating e-learning material conformant with the SCORM reference model and the IEEE LTSC LOM standard.

1 Introduction

E-learning is a major industry with growing applications in many sectors, including industrial training, higher education, the military, individual training etceteras. This wide range of applications has given rise to a whole ecology of specialized systems for managing learning and learning resources [18]. Interoperability between systems and between content and systems has become a major issue and resulted in several standardization efforts. In e-learning, the first result has been a standard for learning object metadata (LOM) developed by the IEEE [12] and standardization of other aspects of e-learning, including content packaging, messaging between content and systems, or sequencing of learning resources. The Shareable Content Object Reference Model (SCORM) from the Advanced Distributed Learning Network (ADL) [19] pulls together these standards and specifications and defines how they work together. Standardized metadata descriptions enable content providers to describe different aspects of the learning resources and promote both interoperability and sharing of resources.

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 65–80, 2003.

The IEEE LTSC LOM is intended to be comprehensive and to cover most situations in which metadata are applied to learning resources. In actual implementations it is often useful to enrich the standard by applying taxonomies or ontologies to elements which are specific to the particular application. We refer to such implementations as application profiles [5].

One of the main obstacles to the widespread adoption of systems which make intensive use of metadata is the time and effort required to apply metadata to multiple resources and the inconsistencies and idiosyncrasies in interpretation that arise when this is a purely human activity. A typical three-hour course may be composed of several dozen content objects each of which may have several included media elements which can generate anywhere from 1,000 to 5,000 separate metadata values for a three hour course [15]. Some of these metadata values can change when the structure of the course changes, making upkeep an expensive proposition.

This problem is not limited to learning resources. Learning resources are increasingly being organized as learning objects [23] and in turn learning objects can be a part of a transformational set of content objects which includes knowledge objects (for use in knowledge management systems), performance objects (for use in performance support systems), collaboration objects (for use in collaboration systems) and so on. Learning objects, and content objects generally, tend to proliferate rapidly once introduced, and the amount of metadata that needs to be applied grows exponentially.

Recent studies by Greenberg et al [7] show increased awareness of the importance of creating metadata for objects even in industrial settings, despite worries about the quality of the metadata [21]. In a follow on study [8] Greenberg et al looked at collaborative metadata generation and found that of all the metadata fields, authors most frequently sought expert help when generating "subject" metadata. The study was conducted for a simple 12-element set from Dublin Core [4]. As the number of elements and requirements for richly structured metadata sets increase, the need for systems that support metadata generation and management will also grow.

There is an important relationship between ‘subject’ metadata, ontology construction and the goals of the semantic web. The semantic web initiative [2] is working towards a vision of having data on the web defined and linked in a way that can be used by machines for various applications and not just display purposes as is generally the case today [11]. Ontologies are a basic component of the semantic web. There is a direct connection between semantic web and metadata systems. The values for the metadata elements can be successfully represented using formal ontologies supporting computational reasoning [22].

Several systems for ontological metadata have been proposed. Weinstein and Birmingham [22] have proposed an organization of digital library content and services within formal ontologies. They focused on created ontological content metadata from existing metadata. In Jenkins et al. [13] an automatic classifier that classifies HTML documents according to Dewey Decimal Classification is described. An ontology-based metadata generation system for web-based information systems is described in Stuckenschmidt et al. [20]. The latter approach relies on the existence of a single ontology to which all the web pages can be related.

In this paper we present a system that helps metadata creators generate high quality metadata by suggesting values for the metadata fields. The main strength of the system is its ability to specify the rules for a particular metadata schema and application

domain ontologies based on the sound analysis of the domain. In addition to generating metadata from standard sources such as object content and user profiles, the system benefits from considering metadata record assemblies, metadata repositories, explicit domain ontologies and inference rules as prime sources for metadata generation.

The paper is organized as follows. In Section 2 we introduce the basic features of metadata systems and provide a typology of metadata records and metadata elements. In Section 3 we analyze the source of suggested values for metadata elements. Section 4 presents four methods of metadata generation and classifies operations on objects affecting the suggested metadata values. In Section 5 we discuss the implementation of the system using inference engine and discuss different types of rules based on the four methods presented in the previous section. We conclude with a discussion of our approach and provide direction for further projects.

2 Metadata Systems

Interest in metadata stems from a belief that these problems can be solved by focusing on the actual meaning of the words in the web document and providing a textual meaning for non-text-based documents. In this sense, metadata function in a manner similar to a card record in a library providing a controlled and structured description of resources through a searchable 'access points' such as title, author, date, location, description, etc. [6]. There are also compelling arguments against metadata which are based mainly on the nature of human behavior [3]. Interestingly enough, both metadata proponents and opponents agree that it is the *quality of metadata* that is essential for high quality retrieval.

The quality of the metadata is the underlying factor that affects the overall performance of the system. Standardization processes try to increase quality by providing sound technical solutions and best practices. However, it is the human factor in the process of metadata creation that affects a metadata quality and usefulness the most. Friesen et al. [6] make an excellent point that the metadata approach effectively inserts a layer of human intervention into the web-based search and retrieval process. Documents in this new approach are not determined as relevant to a specific subject or category as a direct result of their contents, but because of how a metadata creator or indexer has understood their relevance. By focusing on the metadata creation process and facilitating the metadata creator we aim to help overcome some of the well-known problems [3].

Metadata Standards and Record Types. Metadata can fulfill its purpose only when it complies with some standard that a group of people agreed on. Metadata standards can be generic and support broad range of purposes and business models, for example Dublin Core [4], or they can be specific for a particular community, for example IEEE LTSC Learning Object Metadata (IEEE LOM) [12]. A metadata standard provides a set of elements, defines their meaning, and provides guidelines and constraints on how to fill element values. An *individual* metadata record complying with the standard represents one object or document.

This approach can be extended to consider an *assembly* of documents or objects. For example, the SCORM initiative from the Advanced Distributed Learning Net-

work[1], which uses the IEEE LOM, describes a reference model for describing a collection of learning resources, their aggregations and their underlying assets.

Finally, a metadata *repository* is a collection of many metadata records, typically complying with the same standard, or with a way to map between standards. In addition to effective storage, a repository provides search and metadata creation tools and capabilities.

Typology of Metadata Elements. The elements of metadata schemas[2] can hold different types of values. The simplest element type is a *free text* field, for example a 'title' element. Typically, the standard provides a set of guidelines for what the values should represent. A second common element type has an associated *vocabulary* of possible values that metadata creator has to choose from. For example, the IEEE LOM element 'status' can have one of the four values {draft, final, revised, unavailable}. A third type of element uses an *external taxonomy* or classification schema. For example, Dublin Core enables the user to specify a classification schema for its 'subject' element, such as 'Library of Congress' and provide a term from the library's classification taxonomy. Finally, the element values can refer to concepts and objects in *ontologies* as they are defined in the semantic web initiative [2].

3 Source of Suggested Values

The number of elements can vary greatly between metadata schemas, for example, Dublin Core defines 15 elements, and IEEE LOM defines over 80 elements. There are two main obstacles in creating high quality metadata records: 1. the amount of work and time required for applying the metadata, and 2. the expertise required for this task – especially since in many situations, the metadata creator is either a subject-matter expert, a learning object designer/programmer, or a clerical member of a production team. With experience, creators can usually master the guidelines for the metadata standard in use, however references to external taxonomies and ontologies are more problematic as they are often beyond the creator's expertise [7]. Not only must the creator be familiar with the content of a particular ontology, s/he has to be able to navigate to the appropriate concepts quickly. It is therefore of great help to the creators if the system can suggest the most probable values to select from for as many elements as possible. This can improve both the speed with which metadata can be applied and the quality and consistency of the metadata itself. By constraining the values of the metadata elements using an ontology and by providing a system that can propose values, the individual biases described above can be moderated.

3.1 Sources Based on Record Types

The suggested values for metadata records can be computed from different sources. A different set of sources can be used for different record types.

---

[1] http://www.adlnet.org.

[2] As not all element sets are standardized, metadata schema provides for a more generic name for an element set.

Individual records. Individual metadata records (related or not related to other records) can have three main sources of suggested values for metadata elements:

- the user's profile,
- the application profile, and
- the object itself.

The metadata creation tool can enable the *user* to create a profile keeping information such as the user's name, affiliation, email, etc. This information can be used to automatically pre-fill the values for some metadata elements, for example element 'creator' in the Dublin Core standard.

Similarly, the *application* using the metadata is typically designed with some specific purpose, and in many cases several records are sequentially created for the same application. The application specific information can be preset in the application profile and used as suggested values for some elements. For example, the 'TypicalAgeRange' element in the IEEE LOM can be preset to value '18-24' for all the metadata if a creation tool is used for undergraduate post-secondary education.

The above sources are relatively static with regard to the individual metadata record. However, a quite significant amount of the metadata can be harvested from the *object or document* itself. The information retrieved directly from the object for which the metadata record is created can be size (in bytes), MIME type (e.g. text/html), location (e.g. URL), title (e.g. from the <TITLE> tag), etc.

Assemblies. Objects which are part of an *assembly* create together a whole and therefore it is possible that they share several element values. Although the objects in the assembly and their metadata records are distinct, a value set for one metadata element in one objects can propagate itself as a suggested value to other objects in the assembly. If the assembly is organized hierarchically some of the values can be inherited form the ancestor nodes or aggregated from the child nodes. For example, technical requirements to execute an assembly of objects are a union of requirements to execute each object which is a member of the assembly. A SCORM package is a good example of an assembly representing an e-learning course consisting of several learning objects.

Repositories. Metadata records have to be managed just as any other data type. The *repository* typically contains many records which are available as a source of suggested values for some elements. The basic idea uses a notion of 'similar objects' (we will define similarity later). If we can find an object similar to the object for which we are creating metadata we can assume that they will have the same or similar values for at least some of the metadata elements.

Two basic type of similarity can be used. First, two objects can be similar in their content. To be able to compute this similarity, some method of measuring the content similarity has to be used. Currently, the methods for the text-based documents are quite well documented [1][10][14], however methods for other media types are not generally available.

The second notion is similarity defined by some set of rules. For example, let's assume there is a dependence between the values in the element describing an 'eco-

nomic sector' in which a resource is used and 'prerequisite knowledge' for learning resource. A simple rule could capture following heuristic: if a user specifies a value for 'economic sector' in a new record then find other records with the same or 'similar'[3] values for this element and use values from found records from their 'prerequisite knowledge' element as suggested values for this element in the new record.

3.2 Sources Based on Types of Elements

With respect to the element typology presented in the previous section, values can be suggested for each type of element: free text elements, vocabulary elements, and external taxonomy/ontology elements. For example, the value for the 'title' element (free text) can be suggested from the object itself, the value for the 'TypicalEndUser' element can be pre-selected from the defined vocabulary based on the application or user profile, and the value for the 'EconomicSector' element can be suggested from the ontology as a result of an inference using all records in the repository.

It is also important to note that for some elements it is very problematic if not impossible to suggest a value. This is especially true when completely new content is being brought into a system.

4 Generation of Suggested Values

In the previous section we identified several sources which can be used to generate a set of suggested values (*SSV*) for metadata elements. This paper concentrates on generating suggested values for objects in assemblies and repositories, i.e. using values in metadata records for 'similar' objects. Although our implemented system makes use of all the techniques mentioned in the previous section we trust an interested reader can consult the available comprehensive literature and tools on generation of metadata from documents and profiles (e.g. available from http://dublincore.org).

By *assembly* we mean a collection of relatively independent units, each having its own metadata record. The assembly may have an associated metadata record of its own. These units are organized into some sort of structure, for example a hierarchical structure or a structure based on IMS Simple Sequencing. For example, an e-learning course can consist from several lessons which in turn can consist of several units. In the e-learning domain, the IMS Packaging schema used by SCORM makes it possible to define an organization of units (called Shareable Content Objects or SCOs) into a hierarchy of aggregations. Fig.1 displays a hierarchical organization of an e-learning course within an application for managing the structure and its metadata.

We define an *aggregation* as a content object which contains other resources called assets. The assets can be media resources which are reusable (i.e. it makes sense to apply metadata to them) but to demonstrate their value they have to be included into the context providing object. For example, a webpage including an animation and image can represent an aggregate which has its own metadata record. The animation and the image can each have their own metadata records as well. Fig.1 displays an aggregate with 2 objects included.

---

[3] In our example similarity for sectors can be defined either explicitly through related_to relation or implicitly for sectors within the subtree of primary sector in the sector ontology.

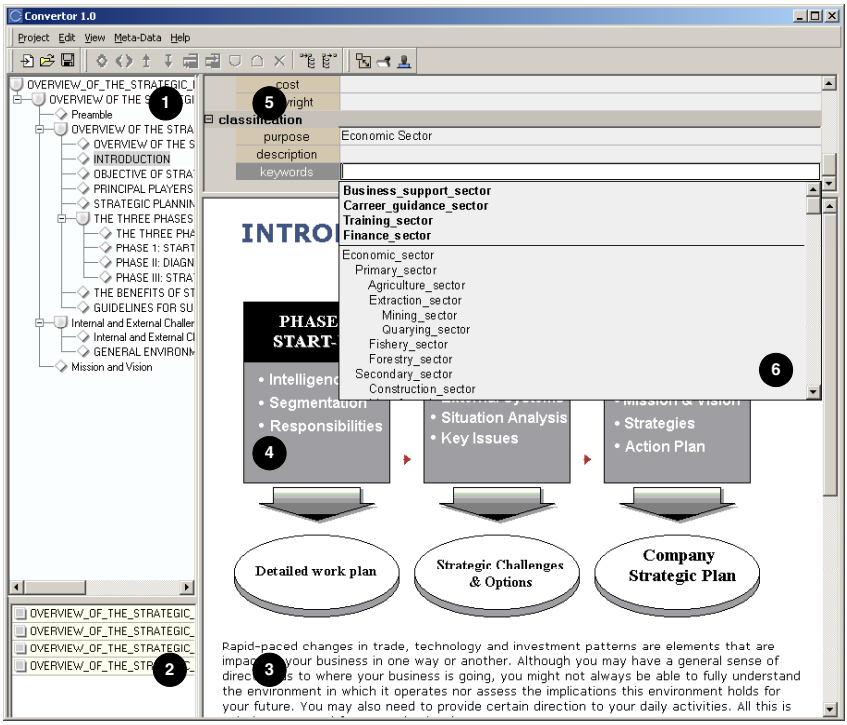

Fig. 1. Snapshot of the Recombo assembly and metadata creation tool containing: assembly hierarchy (1), list of assets in the selected object (2), object (3), asset included in the object representing an aggregation (4), metadata fields (5), and list of suggested values in front of the vocabulary (6).

Inheritance Method. This method is applicable to the metadata records of objects organized into assemblies and aggregations. The suggested values for an element exhibiting inheritance property are accumulated from the elements of records on the path to the top level record.

For assets in the aggregations, a set of suggested values generated through the inheritance is a union of SSV for their aggregate and metadata values of the particular element for the aggregate object.

It is interesting to note, that although both aggregates and assets can be viewed as parts of the same hierarchy in an assembly, the rules for the SSV can differ for the same element in each aggregate, SCO or asset. For example, the 'creator' element for the aggregates (i.e. course content) exhibits the inheritance property and we can include into *SSV* each value found in the 'creator' element in records on the path to the root record of the hierarchy. The situation is different for the 'creator' element for an asset: in addition to the values collected by inheritance from the root the second set of set of values is collected from all assets in the assembly with the 'similar' media type. This heuristics is based on an assumption that it is likely that a creator of a specific media type is a media developer specializing in a particular type of media object[4].

[4] The second set of values would be generated using semantically defined similarity.

Accumulation Method. This method works uniformly through assemblies and aggregations. The suggested values for metadata elements that exhibit an accumulation property are collected from same metadata elements in records forming a subtree of the current record.

An example of an element with an accumulation property is 'technical requirements' element containing a system specification required to execute an object. The technical requirements to execute an object representing a sub-hierarchy of objects and assets are a union of technical requirements of each object and asset in the hierarchy.

Content Similarity Method. The content similarity method makes use of all accessible metadata records in the repository. A set of suggested values for the elements exhibiting this property is calculated as the union of element values from metadata records of objects exhibiting content similarity with the object under consideration.

The method can employ different algorithms for calculating content similarity and different algorithms can be used for calculation of the similarity for different elements. For example, to generate *SSV* for the 'technical requirements' element for assets the algorithm to calculate content similarity can simply compare MIME types of the objects. However, to suggest values for the metadata element 'subject classification' a content analysis of the textual content of the object has to be performed as a statistical text analysis.

Semantically Defined Similarity Method. Semantically defined similarity is the most powerful and complex of the methods presented. This method operates purely on metadata records in the repository and takes into consideration both metadata element values of finalized records as well as the values in the record being created. Fig.2 demonstrates the main principle. Based on already filled values in the metadata elements in the current record one or more of the inference rules are triggered. The inference rules calculate the values for a set of metadata fields which characterize similar records. The similar records are retrieved and a set of suggested values for another field(s) is generated as a union of values from the similar records.

This method is most suitable for elements which use vocabularies derived form formal ontologies. The set of inference rules then operates on the ontologies and can use powerful inference techniques such as forward chaining or constraint satisfaction.

The customized metadata systems (metadata profiles) benefit from this method the most as standards typically provide vocabularies in only a limited form. Customized metadata systems can add elements significant for a particular domain and define domain ontologies and domain specific inference rules.

4.1 Effects of Operations on Assemblies

Creating metadata records is a highly interactive activity. Typically, users fill in metadata values in some form-type interface which can display several dozens of fields[5]. When creating an assembly, the total number of fields can easily reach hun-

[5]A field is a representation of the metadata element in the form interface.

dreds or thousands. A graphical interface, as the one shown in Fig.1, allows the user to quickly reorganize the structure of an assembly and change focus from one object to another. The suggested values generation module has to be able to respond to these changes in real time. Some of the methods presented in the previous sections can be computationally expensive, such as the semantic similarity method. Therefore, it is important that the system regenerates suggested values only for the elements affected by the change in the assembly structure or changes to the values of metadata elements.

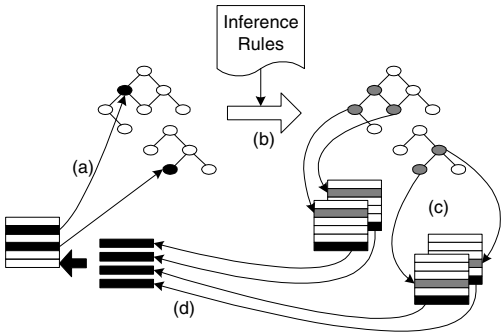

Fig. 2. Generating suggested value through semantic similarity: values mapped to ontology concepts (a) are consumed by inference rules (b) which define semantically similar concepts. The records with similar concepts are found in the repository (c) and their values are used as similar values for a specific metadata field.

We categorize possible operations into three main categories: changes to the object content, changes to the assembly structure, and changes to the metadata values. This categorization helps us analyze an influence of operations on the set of suggested values generated by methods presented above. The operations affecting suggested values are listed in Table 1.

There are three distinct significant places where changes affecting the set of suggested metadata values for a particular object can occur:

- In the object itself.
- In the path from the object to the root of the hierarchy.
- In the object's subtree in the hierarchy.

We have analyzed operations with respect to already generated suggested values and implemented an appropriate scheduling algorithm for re-calculation of suggested values affected by operations. For limited space here we will report these results elsewhere.

5 System for Metadata Generation

The module for generation of suggested metadata values is a component which can be plugged into a larger system for metadata management. The main purpose of the system is to support the user in his or her task of creating metadata descriptions for

sets of the objects. The system is generic and independent of the application domain and can work with various domain ontologies.

Table 1. Operations influencing suggested metadata values.

Operation	On	Description
Join	OC	Join operation concatenates content of two or more leaf objects in the hierarchy by replacing both objects with the final product. For non-leaf objects the object is replaced with concatenation of all its children.
Unjoin	OC	Unjoin operation restores the state before the Join operation was performed.
Split	OC	Split operation separates the content of a leaf node into two objects and by successive splits into multiple objects.
Group	AS	Group operation introduces and extra level into the object hierarchy in the assembly. It replaces a set of nodes with one node having an original set as its children.
Ungroup	AS	Ungroup operation restores the state before the Group operation was performed.
Promote	AS	Promote operation moves a node and all its subtree to the next level in the hierarchy. The new node is positioned after the node it belonged to.
Demote	AS	The node is moved into the first preceding group at the same level. The node will become the last child in the node.
Value filled	ME	Metadata value is filled or selected from the vocabulary for the metadata element.

OC – object content, AS assembly structure, ME – metadata element

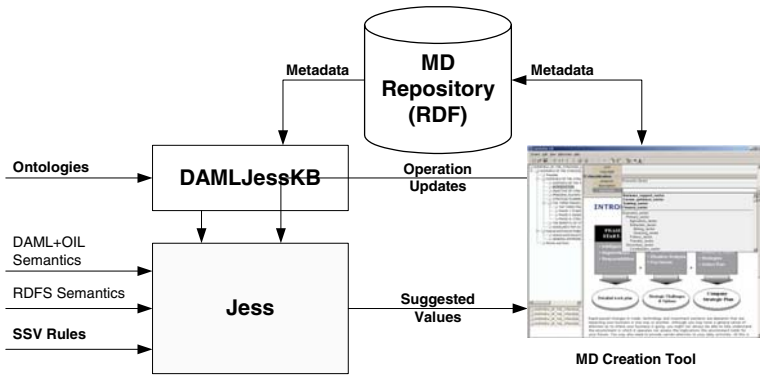

Fig.3. Architecture of the system for generation of suggested metadata values.

The architecture of the system is shown in Fig.3. The generation module responds to requests generated as a result of user's actions in the metadata creation tool such as the one shown in Fig.1. In the process of generation of suggested values the generation module communicates with other components to fulfill its goals. In this paper we concentrate only on the generation component and show how metadata standards, inference rules and ontologies are used to generate suggested values.

5.1 Metadata and Ontology Representation

As shown in Fig.3 the core of the generation module is a forward chaining inference engine (Jess). Jess operates on a knowledge base which consists of facts and rules. We have represented our input data (metadata schemas, metadata records, ontologies and currents state of the problem) as facts. To transform our input data expressed in RDF and DAML+OIL to Jess facts we have used DAMLJessKB[6]. DAMLJessKB transforms RDF triples into facts with a dummy prefix PropertyValue. Below are two examples of how IMS LOM metadata RDF statements are represented as Jess facts[7].

```
(PropertyValue lom_edu:intendededenduserrole
     http://www.sfu.ca/courses/some_lo/ lom_edu:learner)

(PropertyValue lom_edu:context
     http://www.sfu.ca/courses/some_lo/ lom_edu:HigherEducation)
```

The DAML+OIL statement

```
<daml:Class rdf:about="my_ont:Accounting_sector">
   <rdfs:subClassOf>
       <daml:Class rdf:about="my_ont:Business_support_sector" />
   </rdfs:subClassOf>
</daml:Class>
```

was converted into

```
(PropertyValue rdfs:subClassOf
     my_ont:Accounting_sector my_ont:Business_support_sector)
```

DAMLJessKB defines semantics of the RDFS and DAML+OIL as a set of rules which are loaded into the Jess engine and fired at the startup of the system. In addition to basic semantics we have also preloaded all the metadata schemas and ontologies used in the existing records in the repository. Finally, as the rules during inference process operate on metadata of existing records we needed to load metadata records into Jess. However, as not all metadata elements participate in the inference process, we have developed a filter which analyses the rules and loads only those metadata elements of existing records which are referred to by inference rules. Other record elements can be accessed by external subsystems directly from the repository as needed[8].

5.2 Inference Rules

The rules represent the knowledge in the system. The knowledge base contains several types of rules. As mentioned before, some of the rules representing semantics of RDFS and DAML+OIL are part of the DAMLJessKB. The rules we have developed

---

[6] http://plan.mcs.drexel.edu/projects/legorobots/design/software/DAMLJessKB/

[7] In shown Jess representation we have replaced full URIs with qualified notation for clarity.

[8] For example, the ‘description’ element is accessed directly from the repository by the algorithm evaluating content similarity which is implemented outside of Jess engine and is called when user modifies the text in the element exhibiting content similarity property and facts indicating content similarity are asserted into the Jess knowledge base.

generate suggested values for metadata elements and can be grouped according to the methods described in Section 4. Another set of rules is triggered by the operations described in Section 5 and maintains consistency of SSVs.

Rules for Inheritance and Accumulation Method. These simple rules collect values from one particular element from records belonging to the same hierarchical assembly as a currently created record. The rules use the information about the hierarchy to simply collect values. A rule collecting values from all objects in the path to the root of the hierarchy is shown below:

```
(defrule suggest-intededenduserrole-based-on-inheritance
  "Inherit values of intendededenduser from hierarchy above."
  (current_LO ?cur)
  (object-from-path-to-root ?cur ?obj)
  (PropertyValue lom_edu:intendededenduserrole ?obj ?val1)
=>
  (assert (SuggestedValue STRONG lom_edu:intendededenduserrole ?cur
?val1)))
```

Rules for Content Similarity Method. The content similarity is used in our system in two different ways. First, we use content similarity to compare values of some metadata elements such as 'description'. Secondly, we apply content similarity to the objects themselves. The algorithm to evaluate content similarity is external to the inference engine and only facts indicating content similarity and its level are used as the following rule shows:

```
(defrule suggest-subject-based-on-description-similarity
  "Object with similar descriptions can have same subject."
  (current_LO ?cur)
  (similar-content dc:description ?cur ?obj ?level)
  (PropertyValue dc:subject ?obj ?val1)
=>
  (assert (SuggestedValue ?level dc:subject ?cur ?val1)))
```

Rules for Semantic-Based Similarity Method. The rules in this category implement heuristics reflecting interdependencies between values in metadata records at several levels of specificity. The most generic rules capture the dependence between metadata elements regardless of the values they have. An example is a rule suggesting value for 'technical requirements' based on the 'media format':

```
(defrule suggest-requirements-based-on-formatr
  "Objects with same format can have same technical requirements."
  (current_LO ?cur)
  (PropertyValue dcq:medium ?cur ?med)
  (PropertyValue dcq:medium ?obj ?med1)
  (test (same-medium ?med ?med1))
  ;;look for specific subclass of TechnologyRequirement
  (PropertyValue rdfs:subClassOf ?trq lom_tech:technologyrequirement)
  (PropertyValue ?trq ?obj ?val1)
=>
  (assert (SuggestedValue STRONG ?trq ?cur ?val1)))
```

A second type of the rule makes use of the taxonomical information and captures the dependence between classes of values for the particular metadata elements and other elements. The notion of similarity in these rules is based on subClassOf proper-

ties between values which are expressed as taxonomies. For example in the IEEE LOM RDF binding the specific classifications are subclasses of the lom_cls:classification. The following rule collects suggested values for specific classifications from objects created by the same content provider:

```
(defrule suggest-classification-based-on-contentprovider
  "Same person is working within some domain so he can classify ob-
jects using the same classification as same objects within the same
package. "
  (current_LO ?cur)
  (PropertyValue lom_life:contentprovider ?cur ?val)
  (object-from-same-package ?cur ?prev)
  (PropertyValue lom_life:contentprovider ?prev ?val0)
  (test (same-entity ?val ?val0))
  ;;look for specific subproperty of classification
  (PropertyValue rdfs:subPropertyOf ?cls ims_cls:classification)
  ;;subject is used for purpose and idea
  (test (neq ?purpose dc:subject))
  (PropertyValue ?cls ?prev ?val1)
=>
  (assert (SuggestedValue STRONG ?cls ?cur ?val1)))
```

A third type of semantic similarity rules uses a generic notion of similarity. It relies on the use of ontologies as underlying conceptual structures for the values forming a metadata element vocabulary. The similarity is expressed by other set of rules and uses a full expressive power of ontologies. For example, the following rules suggest values for the classification element prerequisite based on the similarity between economic sectors. The sectors are considered to be 'similar' if they are related with each other. The relation is defined either by a property 'related_to' that is specified in the ontology directly for specific sectors or by the rule below covering a generic case where sectors are considered 'related' if they reside in the same subtree with the primary economic sector (level 2 and below):

```
(defrule related-within-main-sector
   "Industries within the main sector are related by default"
  (PropertyValue daml:subClassOf ?medSec es_ont:Economic_sector)
  (PropertyValue daml:subClassOf ?medSec2 ?medSec)
  (PropertyValue daml:subClassOf ?sec ?medSec2)
  (PropertyValue daml:subClassOf ?sec2 ?medSec2)
=>
  (assert (PropertyValue es_ont:related_to ?sec ?sec2)))

(defrule suggest-prerequisites-if-object-used-in-related-sectors
  "Objects used in related sectors can have same prerequisites."
  (current_LO ?cur)
  (PropertyValue md_sec:used_in ?cur ?sec)
  (PropertyValue md_sec:used_in ?obj ?sec1)
  (PropertyValue es_ont:related_to ?sec ?sec1)
  (PropertyValue lom_cls:prerequisite ?obj ?val1)
=>
  (assert (SuggestedValue MEDIUM lom_cls:prerequisite ?cur ?val1)))
```

It is interesting to note that the first of the above two rules depends purely on the ontology and all related sectors are inferred at startup time. The use of es_ont:related_to in the second rule is therefore straightforward pattern matching which is done very effectively in forward chaining rete algorithm.

5.3 Interaction between Knowledge Base and External System

As the creation tool is an end-user application the user's actions change the current state of the metadata records. When the user changes a metadata value in the creation tool, this information is inserted into the knowledge base and several rules can be activated and suggest new values for some fields. As these changes are caused by operations described in Section 4.1 another set of rules monitors the user's actions and keeps a set of suggested values consistent with the current status of the metadata record.

5.4 Implementation

The generation module was implemented within the content integration framework developed by Recombo Inc. in Vancouver, British Columbia (www.recombo.com). Recombo is developing a suite of applications to support emergent XML content standards. The initial implementation is in the application domain of eLearning. One of the applications, Convertor for Word, which imports an RTF file, chunks it into small pieces (learning objects), and provides tools to manipulate the assembly structure, in this case an IMS manifest, through a set of operations such as those described in Section 5.1. Tools for creating and managing the required IEEE LOM metadata and for exporting a SCORM package are also provided.

In the current implementation the rules operate on the IEEE LOM metadata expressed in its RDF binding[9] which is compatible with Dublin Core RDF specification. In addition to metadata schemas we have implemented some of the IEEE LOM vocabularies as lightweight DAML+OIL ontologies and used ontology concepts instead of the plain string values. The tool was tested on the set of local college training material. For that purpose, we have expanded the IMS metadata schema with a set of elements specific to the economic sectors and adapted or built ontologies for these elements (including 'economic sector', 'basic skills', and 'market demand' ontologies).

Fig.1 shows a snapshot of Convertor with the suggested values for the field 'Economic Sector' (marked with number 6). Although the selection looks simple, the suggested values listed above the single line have the potential to save the user from searching through the list of 84 concepts in the full Economic Sector ontology.

6 Conclusions

Metadata is widely recognized as the key to the more effective retrieval and use of information, but most efforts to date have foundered on the resistance of the average content creator to developing and applying accurate and consistent metadata [3]. Anyone who has tried to apply metadata to a large collection of content objects realizes just how time consuming this can be. We believe that the solution to this lies in three related developments: (i) the introduction of XML-based metadata standards, of which the IEEE LOM is one example, (ii) metadata profiles which customize these standards for specific groups of users and (iii) tools that support metadata generation

[9] http://www.imsglobal.org/rdf

and management. In this paper we have focused on an approach in which the system generates suggested values for the metadata with the user either selecting the value from the range of proposed choices or using the suggested values as a pointer to the potentially relevant part of larger ontologies. As this does not reduce the amount of fields the user has to fill in we are considering weighting suggested values and setting up a threshold for automatic value assignment. This will need more experimentation and testing.

Many current content object repositories, learning content management systems and content management systems assume a rich metadata environment. One reason for the delay in the widespread adoption of these systems is the difficulty, the cost, and in fact the reluctance of users to invest in the creation of accurate and consistent metadata. Systems that have been successful are typically those in which a very structured and controlled authoring environment can be created and enforced. This excludes many of the most interesting and creative sources of content, such as individual subject matter experts, the results of collaborative sessions, and the enormous wealth of legacy material that exists in the world today. We believe that the effective implementation of a system such as the one described here is the way forwards for widespread metadata creation and use.

Several approaches to the metadata generation have been tried so far. The metadata generation tools as SPLASH [9] or Aloha [16], or Recombo's Convertor can easily extract and fill values for the technical metadata fields. However, when we look at the systems generating values for other fields we find that they are either limited to a particular ontology domain [20] or they are specifically designed to work with some existing taxonomy [13]. In our approach we look at the object and its metadata record in its context – an assembly the object is part of and the repository the object's metadata record is stored in.

Considering repositories as source of the data is a basis of machine learning approaches. We have not done any comparison of our approach with these approaches yet and will explore some of these techniques in our future work.

Our research continues in several directions. Although anecdotal evidence gives us a positive feedback from the users our work is not complete until we complete an extensive empirical evaluation. By applying the datamining techniques for association rules mining we want to help the system designer in writing inference rules. As the suggested values are generated based on the values already filled in by the metadata creator the order in which values are filled determines how helpful the system can be. Based on the analysis of the rules we are investigating adaptable user interfaces for metadata creation and their usability acceptance. However, the most challenging task is to expand the system into distributed environment and specifically to integrate it with our infrastructure for connecting peer-to-peer repositories [9].

References

1. Aas, K., and Eikvil, L. Text categorisation: A survey. Technical report, Norwegian Computing Center, June 1999
2. Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. Scientific American, May. 2001.
3. Doctorow, C. Metacrap: Putting the torch to seven straw-men of the meta-utopia, Version 1.3: 26 August 2001, http://www.well.com/~doctorow/metacrap.htm

4. Dublin Core, www.dublincore.org
5. Friesen, N., Mason, J., and Wand, N. Building Educational Metadata Application Profiles. DC-2002: Metadata for e-Communities: Supporting Diversity and Convergence, Florence, October, 13-17, 2002. 63-69
6. Friesen, N., Roberts, A., and Fisher, S. CanCore: Learning Object Metadata, Canadian Journal of Learning and technology, Special issue on Learning Objects, Volume 28, Number 3, Fall 2002, 43-53
7. Greenberg, J., Pattuelli, M.C, Parsia, B., and Robertson, W.D. Author-generated Dublin Core Metadata for Web Resources: A Baseline Study in an Organization. Journal of Digital Information (JoDI), 2(2). 2001
8. Greenberg, J., and Robertson, W.D. Semantic Web Construction: An Inquiry of Authors' Views on Collaborative Metadata Generation. DC-2002: Metadata for e-Communities: Supporting Diversity and Convergence, Florence, October, 13-17, 2002. 45-52
9. Hatala, M.., and Richards, G. Global vs. Community Metadata Standards: Empowering Users for Knowledge Exchange, in: I. Horrocks and J. Hendler (Eds.): The Semantic Web – ISWC 2002, Springer, LNCS 2342, pp. 292-306, 2002.
10. Hearst, M.A. Untangling Text Data Mining. Proceedings of ACL'99: the 37th Annual Meeting of the Association for Computational Linguistics, University of Maryland, June 20-26, 1999
11. Horrocks, I., and Hendler, J. (eds.) The Semantic web – ISWC 2002 Springer, LNCS 2342, 2002..
12. IEEE P1484.12.2/D1, 2002-09-13 Draft Standard for Learning Technology - Learning Object Metadata - ISO/IEC 11404, http://ltsc.ieee.org/doc/wg12/LOM_1484_12_1_v1_Final_Draft.pdf
13. Jenkins, C., Jackson, M., Burden, P., and Wallis, J. Automatic RDF Metadata Generation for Resource Discovery, Proceedings of Eighth International WWW Conference, Toronto, Canada, May 1999.
14. Kan, M.Y., McKeown, K.R., and Klavans, J.L. Domain-specific informative and indicative summarization for information retrieval. Proc. Of the First Document Understanding Conference, New Orleans, USA, pp.19-26, 2001
15. Leacock, T., Farhangi, H., Mansell, A., and Belfer, K. Infinite Possibilities, Finite resources: The TechBC Course development Process. Proceedings of the 4th Conf. on Computers and Advanced Technology in Education (CATE 2001), June 27-29, 2001, Banff, Canada, pp.245-250.
16. Magee, M., Norman, D., Wood, J., Purdy, R, Iwing G. Building Digital Books with Dublin Core and IMS Content Packaging. DC-2002: Metadata for e-Communities: Supporting Diversity and Convergence, Florence, October, 13-17, 2002. pp.91-96
17. Qin, J., and Finneran, C. Ontological representation of learning objects. In: Proceedings of the Workshop on Document Search Interface Design and Intelligent Access in Large-Scale Collections, JCDL'02, July 18, 2002, Portland, OR
18. Richards, G. The challenges of the Learning Object Paradigm. Canadian Journal of Learning and technology, Special issue on Learning Objects, Volume 28, Number 3, Fall 2002, 3-9.
19. Shareable Content Object Reference Model (SCORM), www.adlnet.org
20. Stuckenschmidt, H., van Harmelen, F. Ontology-based Metadata Generation from Semi-Structured Information. Proceedings of the First Conference on Knowledge Capture (K-CAP'01), Victoria, Canada, October 2001
21. Weinheimer, J. How to Keep Practice of Librarianship Relevant in the Age of the Internet. Vine (Special Issue on Metadata, Part 1), 116: 14-27.
22. Weinstein, P., and Birmingham, W.P. Creating ontological metadata for digital library content and services, International Journal on Digital Libraries, 2:1 pp. 20-37. Springer-Verlag, 1998
23. Wiley, D. (ed.) The Instructional Use of Learning Objects, Association for Educational Communications and Technology, 2001

Constructing RuleML-Based Domain Theories on Top of OWL Ontologies

Christopher J. Matheus[1], Mitch M. Kokar[2],
Kenneth Baclawski[2], and Jerzy Letkowski[3]

[1] Versatile Information Systems, Inc., Framingham, MA, USA
cmatheus@vistology.com
http://www.vistology.com
[2] Northeastern University, Boston, MA, USA
kokar@coe.neu.edu, ken@baclawski.com
[3] Western New England College, Springfield, MA, USA
jerzy@letkowski.name

Abstract. Situation Awareness involves the comprehension of the state of a collection of objects in an evolving environment. This not only includes an understanding of the objects' characteristics but also an awareness of the significant relations that hold among the objects at any point in time. Systems for establishing situation awareness require a knowledge representation for these objects and relations. Traditional *ontologies,* as defined with a language like DAML/OWL, are commonly used for such purposes. Unfortunately, these languages are insufficient for describing the conditions under which specific relations might hold true, which requires the explicit representation of *implications,* as is provided by RuleML. This paper describes an approach to knowledge representation for situation awareness employing RuleML-based domain theories constructed over OWL ontologies, presented in the context of its implementation in a Situation Awareness Assistant under development by the authors. Suggestions are also made for additions to the RuleML specification.

1 Introduction

Maintaining a coherent *situation awareness* (SAW) with respect to all relevant entities residing in a region of interest is essential for achieving successful resolution of an evolving situation [1], [2], [3]. Examples of areas where this capability is critical include air traffic control, financial markets, military battlefields and disaster management. In any situation the primary basis for SAW is knowledge of the objects within the region of interest, typically provided by "sensors" (both mechanical and human) that perform object identification and characterization; this is known as Level 1 processing in military parlance [4]. Although knowledge of the existence and attributes of individual objects is essential, full awareness also requires knowing the relations among the objects that are relevant to the current operation. For example, simply knowing that there is a west-bound airline and an east-bound airline on the radar screen is not as important as knowing that the two planes are "dangerously close" to one another. In this case, "dangerously close" is a relation between two objects that must be derived from sensory data, although the data by itself says nothing about the concepts of "closeness" or "dangerous".

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 81–94, 2003.

Deriving relevant relations is at the core of what is referred to as Level 2 processing. Unfortunately, the problem of finding all relevant relations is a more difficult problem than merely determining the objects and their characteristics present in a situation. While the number of objects and their attributes may be large, the number scales linearly with the cardinality of the objects in the region. The same cannot be said for relations, whose possibilities increase exponentially as the number of objects increases. Furthermore, relations are abstract semantic concepts that can be constructed at will, unlike physical objects that are provided and fixed by the environment. Yet for any given situation only a small subset of all possible relations will be relevant and meaningful to the goals of the individuals who are analyzing and attempting to establish an awareness of what is occurring in the situation. It is therefore paramount that systems designed to perform Level 2 processing have a notion of the goals of the users and some knowledge about the relations that are relevant to those goals.

Systems that assist in SAW require the ability to represent objects and maintain information about their attributes and relationships with other objects as they evolve over time. This calls for the selection of some form of *data representation*. In addition, because the number of possible relations in a situation makes the problem intractable, some form of domain knowledge is required to help reduce the complexity. This necessitates the selection of some form of *knowledge representation* (KR). The domain knowledge that is required for SAW is of two types: 1) knowledge about what classes or objects, attributes and relations are possibly relevant and 2) what conditions must exist among the objects and their attributes for a given relation to hold true. Rather than selecting a single KR approach to achieve both of these requirements at once, we propose the use of OWL ontologies for the first requirement – a choice that also provides a data representation in terms of instance annotations – and RuleML rules constructed on top of these ontologies to satisfy the second requirement.

In earlier work we developed a formalization of SAW that required the generation of a SAW Core ontology [5], [6], [7], [8]. This ontology was originally developed in UML and then converted to DAML and Slang [9] for formal reasoning on SAW using SNARK [10]. We are now undertaking the development of a functional prototype Situation Awareness Assistant (SAWA) based on this formalization [11]. The SAW Core ontology at the heart of the system will be based in OWL [12]. RuleML [13] will serve as the language for defining domain theories (i.e., collections of axioms or rules). In this paper we describe how we use RuleML in conjunction with our OWL-based ontologies. We start out by discussing the role of ontologies in SAW. We then present our core SAW ontology and demonstration how it can be extended to a specific domain. This leads into a discussion of the need for rules and our selection of RuleML for this purpose. With a simple example we show how rules can be built on top of our core and domain specific ontologies using class URIs as name references. We conclude with several suggestions for additions to the RuleML specification.

2 Ontologies for Situation Awareness

SAW systems need to receive and represent situation-specific information in a computer-compatible form, which presumes the selection or creation of a descriptive language. Because we also intend to reason about this information we need a way to

represent the semantics of the language such that reasoning algorithms (e.g., theorem provers) can make use of the data and knowledge representations. An effective way to achieve this is with ontologies. An *ontology* is an explicit, machine-readable semantic model of a domain that defines classes of entities along with the possible inter-class relations, called properties, specific to that domain.

As part of its Semantic Web effort, the W3C has been engaging in the development of a new XML-based language called the Web Ontology Language (OWL) [12]. OWL is an emerging standard for ontologies and knowledge representations, based on the Resource Description Framework (RDF) [14] and the DARPA Agent Markup Language (DAML), which is the immediate predecessor of OWL. OWL is a declarative, formally defined language that fully supports specialization/generalization hierarchies as well as arbitrary many-to-many relationships. Both model theoretic and axiomatic semantics have been fully defined for the elements in OWL/DAML providing strong theoretical as well as practical benefits in terms of being able to precisely define what can and cannot be achieved with these languages [15]. The field is relatively young, yet several tools have been developed and many more are on the horizon for creating OWL ontologies and processing OWL documents (for a review of such tools see [16]). In our work, we create ontologies as UML diagrams and then programmatically convert them into DAML/OWL representations [17].

It should be noted that the job performed by ontology languages such as OWL cannot be accomplished with purely syntactic languages such as XML Schema. An XML Schema specification can define the structure of objects (*i.e.,* their composition) but it cannot capture the semantic meaning implicit in the relations that might exist between classes of objects. For example, it is not possible to state in XML Schema that the meaning of <address> in one part of a document is of the same class (and thus has the same meaning) as the tag <place> used in another part of the same or different documents. To do this requires the ability to represent knowledge about how classes of objects are related to one another, which is precisely what ontologies capture.

Once an ontology is constructed it can be used to create *instance annotations.* Instance annotations are descriptions of collections of objects marked up in terms of the classes and properties (which define inter-class relations) of an ontology. We will provide some examples of instance annotations after we described our SAW ontology and its extension to a specific domain.

2.1 A SAW Core Ontology

Our original work with SAW involved the development of a formalization of SAW [5,6,7,8] which consisted of a formal definition of SAW, a core SAW ontology, a methodology for reasoning about SAW and the realization of all three as sorts, ops and axioms in Specware [9]. For this paper we will concentrate on our SAW Core ontology, which is depicted in Fig. 1. Shown is a UML diagram of the ontology in which rectangles represent classes and connecting lines indicate inter-class relationships or properties. In the rest of this section we provide an overview of the most important classes and relationships; for a more detailed discussion of our SAW Ontology along with alternative design considerations see [7].

The *Situation* class (upper right corner) defines a situation to be a collection of Goals, SituationObjects and Relations. *SituationObjects* are entities in a situation – both

physical and abstract – that can have characteristics (*i.e.,* Attributes) and can participate in relationships with other objects (*i.e.,* Relations). *Attributes* define values of specific object characteristics, such as position, weight or color. A *PhysicalObject* is a special type of SituationObject that necessarily has the attributes of Volume, Position and Velocity. *Relations* define the relationships among ordered sets of SituationObjects.

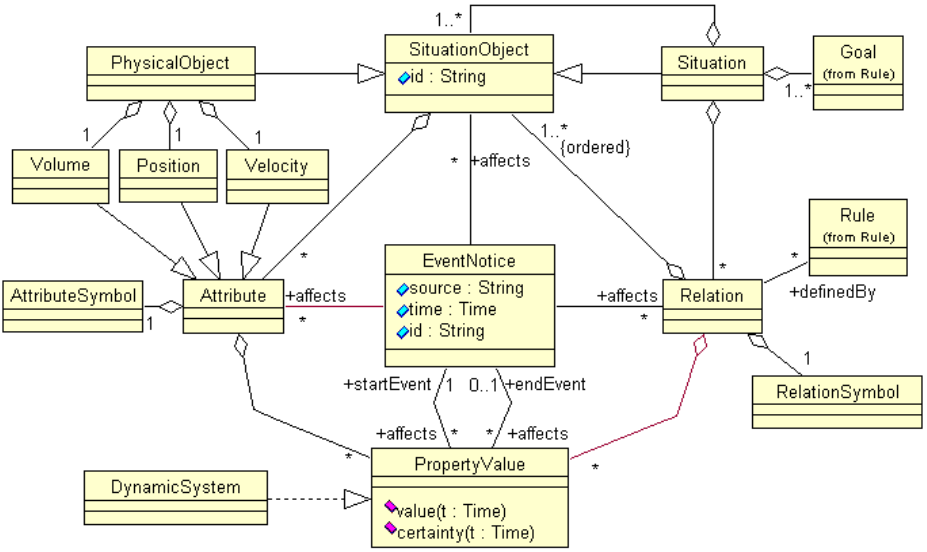

Fig. 1. Core SAW Ontology

An important aspect of Attributes and Relations is that they need to be associated with values that can change over time. To accomplish this Attributes/Relations are associated with zero or more *PropertyValues* each of which defines two time dependant functions, one for the actual *value* and the other for the *certainty* assigned to that value. A new PropertyValue is created for an Attribute/Relation whenever an EventNotice arrives that "affects" that Attribute/Relation. The value of an Attribute/Relation at a particular point in time (either current, past or future) can be determined by accessing the value function of the PropertyValue instance that is in effect at the prescribed time. This is illustrated in Fig. 2, but before explaining the illustration we need to introduce the EventNotice class.

EventNotices contain information about events in the real-world situation as observed by a sensor (the *source)* at a specific *time* that *affects* a specific Relation or Attribute (of a specific SituationObject*)* by defining or constraining its PropertyValue. These are the entities that indicate change in the situation and thus are the vehicles by which changes are affected in the Attributes and Relations of the situation representation.

Consider now the example depicted in Fig. 2. Some event happens at time *t1* resulting in the generation of *eventnotice-t1* by some sensor. This EventNotice affects *attribute1* or *object1* by assigning it a value and certainty instantiated as *property-value1*. At time *t2* a second event occurs generating *eventnotice-t2*, which in turn

affects *attribute1*, in this case by assigning it a new value and certainty in the form of *propertyvalue2*. *Eventnotice-t2* also becomes associated with *propertyvalue1* as it effectively marks the end of *propertyvalue1's* period of being in effect. A similar process occurs with the onset of the third event at time *t3*.

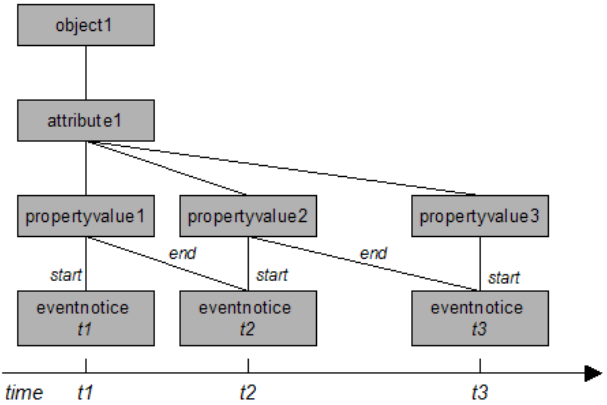

Fig. 2. *PropertyValues* delineated by *EventNotices*

The ontology permits a PropertyValue to be implemented as a DynamicSystem. What this means is the value and certainty functions are dynamically modeled and therefore they cause the PropertyValue to change even in the absence of new EventNotices. To illustrate the need for a DynamicSystem implementation of PropertyValues, consider the Position attribute of a PhysicalObject. The object's Position attribute's value at time t+1 is related to the object's Velocity (a vector providing speed and direction) at time t. Even if no new EventNotice affecting the position is received at time t+1, it is reasonable to assume that the object's position has changed. In the absence of additional information (*e.g.,* acceleration, trajectory) it might be reasonable to assume that the object continues to move with its last noted speed and direction until informed otherwise, albeit with increasing uncertainty as time goes on.

To be able to make such projections in the absence of explicit sensory information requires predictive models. It is for this reason that the SAW ontology shows DynamicSystems as a way of *implementing* PropertyValues. Certain attributes, such as Position, would be modeled by dynamic systems that might themselves generate internal EventNotices to update the attribute values, with some lesser degree of certainty, until new external sensory information arrives. It might also become desirable to fuse multiple model-predicted values or to combine model-generated values with sensory information in cases where the certainty of the external information is less than perfect.

2.2 Extending the SAW Core Ontology to a Specific Domain

The SAW Core ontology defines the fundamental classes and properties needed to support the representation of a large class of situations, specifically those that can be

described by objects, attributes and relations that evolve over time. The core ontology is not very useful, however, until it is extended to a specific domain. There are two aspects to extending the SAW Core ontology to a given domain. The first involves sub-classing the *SituationObject* and *Attribute* classes, which define what kinds of objects are permitted as well as how they are to be described. The second involves sub-classing the *Relation* class to define the types of higher-order relations (i.e., Level 2 information) that are possibly relevant to the domain. These two processes are described in the following sub-sections. Two domain-specific sub-ontologies are used in these discussions; each is an example of a part of what might be a larger, all inclusive ontology for annotating military battlefield situations.

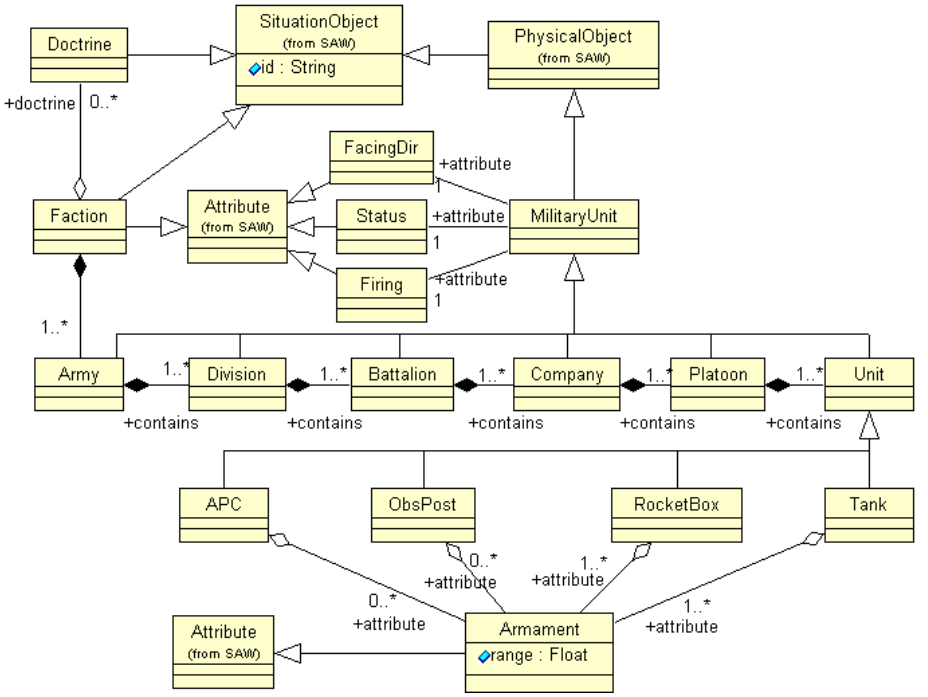

Fig. 3. Military Unit Ontology (partial realization)

Defining SituationObject and Attribute Sub-classes. The natural place to begin extending the SAW Core ontology to a specific domain is with the definition of the SituationObjects pertinent to the domain. Fig. 3 shows a UML diagram of a Military Unit ontology intended to describe individual military units, their attributes and the ways they can be aggregated. Note that all of the defined classes are descendants of either the *SituationObject* class or the *Attribute* class defined in the SAW Core ontology, as indicated by the "(from SAW)" notes. The MilitaryUnit class is the super class for all other unit classes and it defines the Attributes shared by all of them: FacingDir, Status, and Firing. Note that this definition of MilitaryUnit is not meant to be complete but rather to illustrate the concept.

Defining Relation Sub-classes. After defining the classes of objects pertinent to the domain comes the task of defining the relations among these objects that might be of interest. This process is accomplished by sub-classing the SAW Core ontology's *Relation* class. Fig. 4 depicts in UML a Battlefield Relation ontology that partially achieves this for the battlefield domain. While most of the relation sub-classes are partially defined, one, *FiringAt*, is fully defined by specifying its subject and object. Each relation sub-class must specify the classes of its operands (which may be more than two) in this way and the cardinality on these properties must be one; here, only *FiringAt* is shown completely so as to reduce the complexity of the diagram.

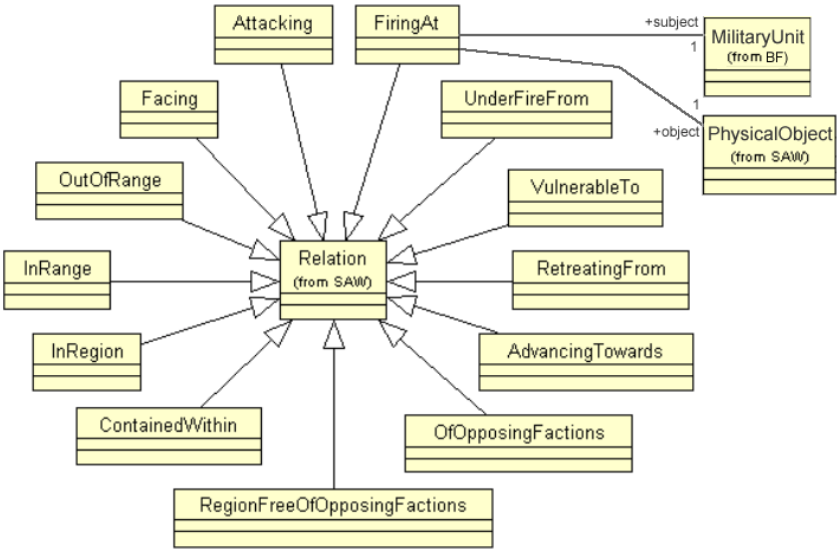

Fig. 4. Battlefield Relations

The reason the subject and object properties are needed has to do with the type of reasoning that we ultimately want to do concerning the relations occurring in a situation. This reasoning requires that we can identify the set of objects in the situation that can possibly be used in the satisfaction of a rule. By defining the permissible classes of operands for a relation we are effectively defining "sorts" which can be used to identify which objects the relations can be applied to. This becomes important when we develop rules for these relations that define when they hold true in a situation. Since RuleML, which we use to represent the rules for relations, does not have the capability of limiting the assignment of variables used in predicates to classes of objects, it is necessary for us to represent this information in the domain-specific relation ontologies.

The following OWL code fragment provides a concrete example of how these relation sub-classes are defined when we convert from UML to OWL; this fragment shows a portion of the code for the battlefield relation ontology highlighting the *FiringAt* relation, which we will be discussing further in the next section on RuleML:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY saw
     'http://www.vistology.com/onts/SAW/saw-core#'>
  <!ENTITY mu
     'http://www.vistology.com/onts/SAW/BF/units#'>]>
<rdf:RDF
  xmlns:owl ="http://www.w3.org/2002/07/owl"
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:saw ="http://www.vistology.com/onts/SAW/saw-core#"
  xmlns:mu=
         "http://www.vistology.com/onts/SAW/BF/units#">
 …
  <owl:Class rdf:ID="FiringAt">
   <rdfs:subClassOf resource="&saw;Relation"/>
   <rdfs:subClassOf>
    <owl:Restriction>
     <owl:onProperty rdf:resource="#subject"/>
     <owl:allValuesFrom
          rdf:resource="&mu;MilitaryUnit"/>
    </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
    <owl:Restriction>
     <owl:onProperty rdf:resource="#object"/>
     <owl:allValuesFrom
          rdf:resource="&saw;PhysicalObject"/>
    </owl:Restriction>
   </rdfs:subClassOf>
  </owl:Class>
 …
</rdf:RDF>
```

3 Domain Theory Rules and RuleML

For a system to be able to detect when a relation exists in a situation it needs to know the specific conditions that must be present for the relation to hold true. This knowledge is most readily represented using *implication*. With implication we can specify that Y (the consequence) can be taken to hold true when $X_1 \ldots X_n$ (the antecedents) are all true. One of the simplest approaches for representing such implication is through Horn clause statements [18], which is what we have selected for our system. Unfortunately, general Horn clause statements are not explicitly representable using the primitives in OWL. OWL can represent simple implication such as subsumption, but it has no mechanism for defining arbitrary, multi-element antecedents.

It is possible to construct an OWL ontology for representing Horn clause statements that could be used to implement rules as instance annotations, however, the rules would be rather verbose, extremely difficult to read and write, and would not be based on an accepted standard. An alternative approach is to use something outside of OWL. There are many languages one could choose for this purpose, but with the benefits afforded by XML (which our system exploits extensively) and the fact that

OWL is XML-based, an XML-based language such as RuleML is really the only approach we would seriously consider. Since the DAML/OWL community decided last year to more closely align its rule effort with that of RuleML, RuleML currently appears as the most logical choice for representing rules in the context of OWL ontologies.

Our approach to the rule construction process is that of building RuleML rules on top of OWL ontologies. We assume that the ontologies are pre-defined and that all their classes are available to be used as elements in the rules. The question is which of the classes from the ontologies are to be used and in what way? Since the purpose of the rules is to permit definition of the conditions that must hold for relations to exist, the heads of the rules clearly must make reference to the domain-specific subclasses of the SAW Core class *Relation*. To do this we simply provide the URI of the desired relation class as the content of the <rel> tag in the head atom operator of a RuleML rule. For example:

```
<_head><atom><_opr><rel>bf:FiringAt</rel>…
```

The bodies of the rules contain additional atoms with <rel> tags that contain URIs pointing either to relation classes (just as is done in the head) or to domain-specific subclasses of the SAW Core class *Attribute*.

3.1 An Example of a Domain Theory in RuleML

A domain theory can be viewed as a network that defines the inter-relationships among the theory's axioms or rules. If we select a single node representing the head of a rule and consider only its descendants, we can create a view that forms a tree (with possibly replicated nodes). Such a tree representing a subset of a hypothetical domain theory for battlefield scenarios is depicted in Fig. 5. The nodes labeled with predicates represent relations or attributes from the domain-specific ontologies. Relation predicates that have child nodes represent heads of rules and their children represent the antecedents or bodies for individual rules; these are denoted with solid dots and the links projecting from them are combined by arcs to indicate that they are to be taken conjunctively.

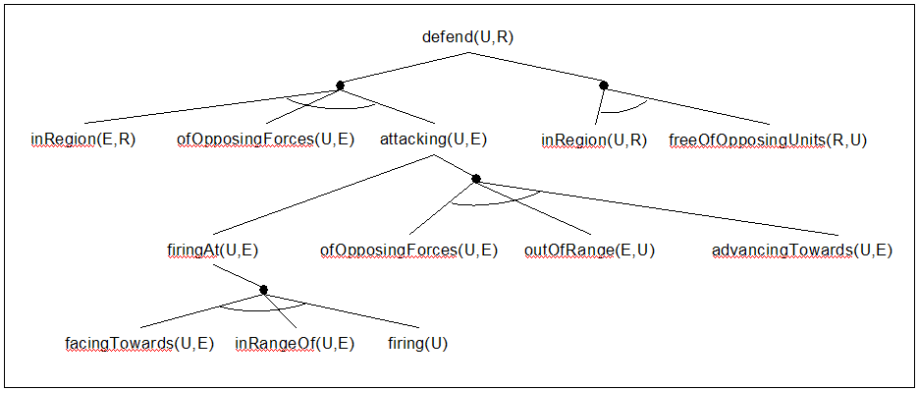

Fig. 5. Domain Theory Tree for defend(U,R)

The partial sub-theory shown in Fig. 5 (intended for illustrative purposes only) defines what it means for a unit U to ***defend*** a region R. The root of the tree is the relation *defend(U,R)* and its two children represent two different rule bodies, either of which, if satisfied, would imply the satisfaction of the root node relation. The first rule body contains three relations, the third one of which – *attacking(U,E)* – has two child nodes (i.e., rule bodies) of its own. Considering the first of these we see that one way to satisfy the relation *attacking(U,E)* is to satisfy the relation *firingAt(U,E)*. This relation in turn is satisfied by conjunctively satisfying the two relations *facing(U,E)* and *inRange(U,E)* as well as satisfying the attribute predicate, *firing(U)*. Note that in our approach, both relations and attributes are represented as predicates. The only difference between the two is that attribute predicates can only pertain to one *SituationObject* at a time whereas relation predicates can and usually do deal with multiple *SituationObjects*.

Here are RuleML rules for *attacking(U,E)* and *firingAt(U,E*:

```
<?xml version="1.0"?>
<!DOCTYPE rulebase SYSTEM
  "http://www.dfki.uni-kl.de/ruleml/dtd/0.8/ruleml-datalog-
monolith.dtd">

<rulebase
  xmlns:saw="http://vistology.com/onts/SAW/saw-core#"
 xmlns:bf="http://vistology.com/onts/SAW/BF/battefield#">
…
<imp name="Attacking">
 <_head>
  <atom>
   <_opr>
    <rel>bf:Attacking</rel>
   </_opr>
   <var>X</var>
   <var>Y</var>
  </atom>
 </_head>
 <_body>
  <and>
   <atom>
    <_opr>
     <rel>bf:FiringAt</rel>
    </_opr>
    <var>X</var>
    <var>Y</var>
   </atom>
  </and>
 </_body>
</imp>

<imp name="Attacking">
 <_head>
  <atom>
   <_opr>
    <rel>bf:Attacking</rel>
```

```
    </_opr>
    <var>X</var>
    <var>Y</var>
   </atom>
  </_head>
  <_body>
   <and>
    <atom>
     <_opr>
      <rel>bf:OfOpposingFactions</rel>
     </_opr>
     <var>X</var>
     <var>Y</var>
    </atom>
    <atom>
     <_opr>
      <rel>bf:OutOfRange</rel>
     </_opr>
     <var>X</var>
     <var>Y</var>
    </atom>
    <atom>
     <_opr>
      <rel>bf:AdvancingTowards</rel>
     </_opr>
      <var>X</var>
      <var>Y</var>
    </atom>
   </and>
  </_body>
</imp>

<imp name="Firing At">
 <_head>
  <atom>
   <_opr>
    <rel>bf:firingAt</rel>
   </_opr>
   <var>X</var>
   <var>Y</var>
  </atom>
 </_head>
 <_body>
  <and>
   <atom>
    <_opr>
     <rel>bf:Facing</rel>
    </_opr>
    <var>X</var>
    <var>Y</var>
   </atom>
   <atom>
    <_opr>
     <rel>bf:InRangeOf</rel>
```

```
      </_opr>
      <var>X</var>
      <var>Y</var>
     </atom>
     <atom>
      <_opr>
       <rel>bf:Firing</rel>
      </_opr>
      <var>X</var>
     </atom>
    </and>
  </_body>
</imp>
…
</rulebase>
```

The URI references to relation and attribute sub-classes from the ontologies are underlined and in bold to highlight the location of their use. Note that there is no distinction made in the rules to indicate whether a predicate is a relation or an attribute. Our system actually needs this information for a process called *relevance determination* in which it derives relevant relations and attributes. Although the information is not in the rules it is in the ontologies and it is possible, using XSLT scripts, to determine the type of a predicate by looking up its URI in the ontology and climbing the inheritance hierarchy to determine whether the URI class is a descendant of *Relation* or *Attribute*.

4 Suggestions for RuleML

Our experience using RuleML in conjunction with OWL ontologies in the context of SAW reasoning has identified some possible additions to the evolving RuleML specification:

- We recommend the adding the option of being able to indicate the permitted classes of values that the variables in RuleML operators may take on. We understand that this may pose some issues for the underlying semantics of RuleML as it is defined currently. However, given the development of three versions of OWL– Full, DL and Lite – we believe a similar approach could be taken with RuleML that would permit applications to take advantage of the convenience of having sort-restricted variables at the possible expense of foregoing certain semantically desirable characteristics. Our work around for the absence of this capability in RuleML was to incorporate argument type information in the definitions of the relations in the domain specific ontologies. This approach has the disadvantage that both the rules and the ontologies need to be present in order to determine the permissible types for arguments in a rule.
- For our purposes it is necessary to be able to uniquely identify specific rules so we can pull a subset of relevant rules out of a larger domain theory (see [5,8]). As shown in Code Fragment 1, we have introduced a “name” attribute to the <imp> tag for this purpose. We considered using the rdf:ID attribute but it is more convenient for us to be able to have multiple rules with the same name, which rdf:ID

does not permit. Having an rdf:ID attribute in addition to a name attribute would likely be useful for situations requiring uniquely identified rules even if they pertain to the same head.

- It seemed contrary to the spirit of XML to have to write the string representation of a URI as the text node of a <rel> tag. It would be more natural to be able to use an rdf:IDREF attribute on the <rel> tag, but the XML Schema for RuleML 0.8 (monolith version) [19] does not permit use of this attribute. Of all our recommendations accepting this one would seem to provide the most benefit as far as making it easier for RuleML to work more closely with OWL.
- The syntax of RuleML rules seems excessively verbose for writing basic rules. The authors have not been a part of the design discussions for RuleML and we accept that there are good reasons for the current syntax. Even so, it is difficult to see why a reduced syntax such as

```
<rule>
  <head>
    <rel/><var/>…
  </head>
  <body>
    <rel/><var/>…
    <rel/><var/>…
    …
  </body>
</rule>
```

could not be part of some human-user friendly dialect of RuleML, similar in spirit to the proposed OWL Presentation Syntax [20].

5 Summary

We presented a case for using RuleML to construct domain theory rules on top of OWL ontologies. The context for this work is that of reasoning about situation awareness, for which we are currently developing a Situation Awareness Assistant (SAWA). We presented a SAW Core ontology and showed how it can be extended to handle domain-specific situations. Since our system needs to be able to derive higher-order relations that exist among the objects in a situation we need to be able to encode domain theories that specify the conditions under which specific relations come into being. This need requires the representation of general implication, a capability that goes beyond the simple implications available in OWL. We argued that RuleML not only provides general implication in the form of Horn clauses but also that its XML representation makes it the ideal chose for use with OWL. We then showed how we use RuleML to define rules that reference relation subclasses in our core and domain-specific ontologies. We concluded with four recommendations for additions to the RuleML specification: 1) permit constrains to be imposed on the classes of values that variables can assume, 2) add “name” and/or “rdf:ID” as possible attributes to the <imp> tag, 3) add “rdf:IDREF” as a possible attribute of the <rel> tag, and 4) create an alternative representation that is easier for humans to process.

Acknowledgements

This work was partially funded by the Air Force Research Laboratory, Rome, NY under contract numbers F30602-02-C-0039 and F30601-03-C-0076.

References

1. M. Endsley and D. Garland, *Situation Awareness, Analysis and Measurement,* Lawrence Erlbaum Associates, Publishers, Mahway, New Jersey, 2000.
2. J. Barwise, "Scenes and other situations", *J. Philosophy* 77, 369-397, 1981.
3. J. Barwise, *The Situation In Logic*, CSLI Lecture Notes 17, 1989.
4. Steinberg, C. Bowman, and F. White, Revisions to the JDL data fusion model, In Proceedings of SPIE Conf. Sensor Fusion: Architectures, Algorithms and Applications III, volume 3719, pages 430-441, April 1999.
5. Matheus, M. Kokar and K. Baclawski, Phase I Final Report: A Formal Framework for Situation Awareness. AFRL Funding Number: F30602-02-C-0039, January 2003.
6. K. Baclawski, M. Kokar, J. Letkowski, C. Matheus and M. Malczewski, Formalization of Situation Awareness, In Proceedings of the Eleventh OOPSLA Workshop on Behavioral Semantics, pp. 1-15, November, 2002.
7. C. Matheus, M. Kokar and K. Baclawski, A Core Ontology for Situation Awareness. Proceedings of FUSION'03, Cairns, Queensland, Australia, July 2003.
8. C. J. Matheus, K. Baclawski and M. M. Kokar, Derivation of ontological relations using formal methods in a situation awareness scenario, In Proceedings of SPIE Conference on Mulitsensor, Multisource Information Fusion, pages 298-309, April 2003.
9. Specware: Language manual, version 2.0.3. Technical report, Kestrel Institute, 1998.
10. SNARK: SRI's new automated reasoning kit, http://www.ai.sri.com/stickel/snark.html, 2002.
11. C. Matheus, M. Kokar and K. Baclawski, Phase II Proposal: A Formal Framework for s
12. OWL Web Ontology Language XML Presentation Syntax. http://www.w3.org/TR/owl-xmlsyntax/.
13. The RuleML Initiative, http://www.ruleml.org/.
14. Resource Description Framework (RDF). http://www.w3.org/RDF/.
15. Ian Horrocks and Peter F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. In Proc. of the 2003 International Semantic Web Conference (ISWC 2003), 2003.
16. OntoWeb Consortium, OntoWeb Deliverable 1.3: A survey on ontology tools, May 2002. http://www.aifb.uni-karlsruhe.de/WBS/ysu/publications/OntoWeb_Del_1-3.pdf.
17. Kogut, P. A., Cranefield, S., Hart, L., Dutra, M., Baclawski, K. and Kokar, M. M. and Smith, J. E. UML for Ontology Development, *The Knowledge Engineering Review*, Vol. 17 (1), pp. 61 - 64, 2002.
18. Horn, "On sentences which are true of direct unions of algebras", *Journal of Symbolic Logic*, 16, 14-21, 1951.
19. RuleML DTDs, http://www.dfki.uni-kl.de/ruleml/indtd0.8.html.

Inheritance and Rules in Object-Oriented Semantic Web Languages*

Guizhen Yang[1] and Michael Kifer[2]

[1] Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260, USA
gzyang@CSE.Buffalo.edu
[2] Department of Computer Science
Stony Brook University, Stony Brook, NY 11794, USA
kifer@CS.StonyBrook.edu

Abstract. Rule-based and object-oriented techniques are rapidly making their way into the infrastructure for representing and reasoning about semantic information on the Web. Combining these two paradigms has been an important objective and F-logic is a widely adopted formalism that achieves this goal. However, the original F-logic was lacking the notion of *instance methods* — one of the most common object-oriented modeling tools. Extending F-logic with instance methods poses new, nontrivial problems. It requires a different kind of nonmonotonic inheritance and impacts much of the semantics of the logic. In this paper we incorporate instance methods into F-logic and develop a complete model theory as well as a computation framework for the extended language.

1 Introduction

From its humble beginning in RDF, the Semantic Web effort has progressed to Description Logic based languages such as DAML+OIL and OWL. More recent initiatives, such as RuleML [19] and DAML Rules [7], testify to the intense interest in extending the Semantic Web with rule-based capabilities. Object-oriented modeling of semantic information on the Web has had strong appeal since very early on. This is perhaps most pronounced in the influential OntoBroker project [2] and its subsequent commercialization by ontoprise.com, which utilizes F-logic [12] — a logic which extends the classical predicate calculus with object-oriented concepts and meta-data programming facilities, and provides a foundation for modeling ontologies and rules in one natural framework. The object-oriented approach was also prominent in the design of the OIL language [1], although this particular aspect received lower priority in the combined DAML+OIL language [8, 9]. Finally, object-oriented approaches, specifically, F-logic based systems, are very popular with the efforts that combine the Web with rule-based reasoning (cf. $\mathcal{F}$LORA-2 [23, 24], TRIPLE [20], XPathLog [14], and FLORID [3]).

* This work was supported in part by NSF grant IIS-0072927.

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 95–110, 2003.

One major difficulty with rule-based object-oriented languages is the semantics of inheritance, especially the issues related to overriding and conflict resolution [12, 25]. A recent study [13] shows that inheritance — especially multiple inheritance — permeates RDF schemas developed by various communities over the past few years. However, neither RDF nor OWL supports inheritance overriding or conflict resolution (in fact, they support no default reasoning at all, which many feel is a serious limitation). As shown in [12, 25], the difficulty in defining a semantics for inheritance is due to the intricate interaction between inferences made by inheritance and via rules (Section 3 shows a simple yet practical example). Unfortunately, this aspect received no satisfactory solution in the original work on F-logic [12], and subsequent works tried to either justify the original solution or impose unreasonable restrictions on the language [16, 11, 15].

Our earlier work [25] proposed a solution to the above problem by developing a semantics that is both theoretically sound and computationally feasible. However, this semantics (like the one in [12] and most other related works) is restricted to the so called *class methods* [18] (or *static methods* in Java terminology). The notion of *instance methods* — a much more important object-oriented modeling tool — was not supported in the language or its semantics. In this paper we rectify this drawback by introducing instance methods and a new kind of inheritance, called *code inheritance*, into F-logic. Our main contribution is the development of a complete model theory and computation framework for nonmonotonic multiple inheritance. Limitation of space does not allow us to define the semantics in full generality, but the reader is referred to [22] for details.

This paper is organized as follows. Section 2 introduces the basic F-logic syntax that is used throughout the paper. Section 3 motivates the research problems concerning inheritance and rules by presenting a motivating example. The new three-valued semantics for F-logic is introduced in Section 4. Section 5 defines the inheritance postulates and Section 6 formalizes the associated notion of object models. Section 7 develops the optimistic object model semantics and discusses its properties and implementation. Finally, Section 8 concludes the paper.

2 Preliminaries

F-logic provides natural and powerful syntax as well as semantics for modeling and reasoning about semantic data. In particular, classes, methods, and values are all treated as objects. For instance, the same object, *e.g.*, ostrich, can be viewed as a class in one context (with members such as tweety and fred) and as a member of another (second-order) class (*e.g.*, species) in another context.

To save space, we use a subset of the F-logic syntax in this paper, which includes only three kinds of *atomic* formulas: A formula of the form $\mathsf{o}:\mathsf{c}$ says that object o is a member of class c; $\mathsf{s}::\mathsf{c}$ says that class s is a (not necessarily immediate) subclass of class c; and $\mathsf{e}[\mathsf{m} \twoheadrightarrow \mathsf{v}]$ says that object e has an inheritable set-valued method, m, whose result is a set that contains object v. The symbols o, c, s, e, m, and v here are the usual first-order terms.

Traditional object-oriented languages normally distinguish between *instance methods* and *class methods*. The former characterize all instances of a class while

the latter characterize classes themselves as objects [18]. Class methods are analogous to "static" methods in the Java language and instance methods correspond to the other nonstatic (instance) methods. In object-oriented data modeling especially in the case of semistructured objects on the Web it is also convenient to *explicitly* define *object methods* for individual objects. These explicitly specified methods override the methods inherited from superclasses. Object methods are similar to class methods except that they are not intended for inheritance. In F-logic both instance and class/object methods are specified using rules.

Let A be an atom. A literal of the form A is called a *positive* literal and $\neg\,\mathsf{A}$ is called a *negative* literal. An F-logic program is a finite set of rules where all variables are *universally* quantified. There are two kinds of rules: value-rules and code-rules. Value-rules were introduced in the original F-logic [12] while introduction of code-rules is one of the contributions of this paper. Generally, value-rules define class membership, subclass relationship, and class/object methods. Code-rules represent pieces of *code* that define instance methods.

A value-rule has the form $\mathsf{H} \Longleftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n$, where $\mathsf{n} \geq 0$, H is a positive literal, called the rule *head*, and each $\mathsf{L_i}$ is a positive or a negative literal. The conjunction of $\mathsf{L_i}$'s is called the *body* of the rule. A code-rule has the following form: $\mathsf{code\ c[m} \twoheadrightarrow \mathsf{v]} \Longleftarrow \mathsf{L}_1, \ldots, \mathsf{L}_n$, which is similar to a value-rule except that it is prefixed with the special keyword code and its head must specify a method (*i.e.*, it cannot be $\mathsf{o:c}$ or $\mathsf{s::c}$). We will also assume that c is a constant[1] and will say that such a code-rule defines the instance method m for the class c.

In the rest of this paper, we will use uppercase names to denote variables and lowercase names to denote constants. A rule with an empty body is called a *fact*. We will call value-rules and code-rules with an empty body value-facts and code-facts, respectively; we will omit " $\Longleftarrow$ " when writing down the facts.

3 A Motivating Example: Building a Pricing Agent

We will now use the simplied F-logic language introduced above to specify a pricing agent that helps an ficticious product vender, Acme International, to stay ahead of competition. The job of this pricing agent is to propose discount prices for sale items based on the following attributes: (i) compPrice: competitor's price; (ii) cost: Acme's cost of procuring a sales item; (iii) discPrice: proposed discount price; (iv) approved: a boolean attribute indicating whether a discount offer is valid (yes) or not (no).

The agent has two modules: one includes discount offer rules (Figure 1) and the other contains loss control rules (Figure 2). The first module calculates discount prices while the second audits if those prices make economic sense. Let us first consider these two modules separately.

In Figure 1, the name coItem represents the class of commodity items and code-rule (1) encodes the pricing policy for all commodity items. It states that *the discount price of a commodity item is 10% off the competitor's price if this discount offer is approved.* Value-rule (2) says that *a sales item is a commodity*

[1] The semantics does not require this, but this assumption is appropriate in practice.

code colItem[discPrice $\twoheadrightarrow$ P] $\longleftarrow$ colItem[approved $\twoheadrightarrow$ yes], colItem[compPrice $\twoheadrightarrow$ C], P = C $*$ (1-10%).	(1)
X : colItem $\longleftarrow$ X[compPrice $\twoheadrightarrow$ C], C $<$ 50.	(2)
colItem[approved $\twoheadrightarrow$ yes].	(3)

Fig. 1. Discount Offer Rules.

X : loItem $\longleftarrow$ X[discPrice $\twoheadrightarrow$ P], X[cost $\twoheadrightarrow$ C], P $<$ C.	(4)
loItem[approved $\twoheadrightarrow$ no] $\longleftarrow$ loItem[totalLoss $\twoheadrightarrow$ T], T $>$ 10000.	(5)

Fig. 2. Loss Control Rules.

item if the competitor's price for it drops below 50 dollars. Finally, value-fact (3) assigns the value yes to the class method approved of colItem. In light of inheritance, this implies that *discount offers of commodity items are valid by default.*

Suppose our knowledge base also contains the fact item101[compPrice $\twoheadrightarrow$ 30]. How do we determine the discount price (the value of the method discPrice) for item101? First, since value-rule (2) is enabled, we can derive item101 : colItem. Now that item101 is known to be an instance of the class colItem, it can inherit the definition of the instance method discPrice (code-rule (1) in Figure 1) from colItem. When inherited, this code-rule is instantiated for item101 as follows:

item101[discPrice $\twoheadrightarrow$ P] $\longleftarrow$
item101[approved $\twoheadrightarrow$ yes], item101[compPrice $\twoheadrightarrow$ C], P = C $*$ (1-10%).

i.e., the object ID, item101, is substituted for the class ID, colItem, which is essentially a *placeholder* that stands for all instances of this class. This substitution corresponds to the so called *late binding* in traditional object-oriented languages like C^{++} and Java. To determine the value of the method approved on item101, observe that approved is defined as a class method of colItem in (3) and hence item101 can inherit its value. Therefore, we can derive item101[approved $\twoheadrightarrow$ yes] and then item101[discPrice $\twoheadrightarrow$ 27].

What if our knowledge base contains the fact item101[approved $\twoheadrightarrow$ no] in addition to all the previous information? Since now item101[approved $\twoheadrightarrow$ no] *explicitly* assigns the value no to the method approved of the object item101, it *overrides* the inherited value yes (from colItem). Consequently the body of the above instantiated rule is no longer satisfied and the method discPrice is not defined on item101. This scenario illustrates the *nonmonotonic* aspect of inheritance — addition of a new fact invalidates a previous deduction.

Now let us look at the rules in Figure 2, which are intended to control possible losses due to price cuts. Here loItem represents the class of loss items whose membership is defined by value-rule (4). It states that *an item is a loss item if its discount price is lower than its cost.* Value-rule (5) defines a *conditional* class method. It implies that *by default loss items are not subject to discount if the aggregate loss on the sales of loss items exceeds 10000.*

Suppose our knowledge base contains rules (1)–(5) and the following facts, item101[compPrice $\twoheadrightarrow$ 30], item101[cost $\twoheadrightarrow$ 28], and loItem[totalLoss $\twoheadrightarrow$ 20000].

It is instructive to see how one can determine the value of the method discPrice for item101. As before, value-rule (2) is fired immediately and yields item101 : coItem. The body of value-rule (5) is also satisfied, which derives loItem[approved $\twoheadrightarrow$ no]. At this point, no other deduction via rules is possible and item101 is known to belong to the class coItem but no other classes. Therefore, it *appears* that inheritance from coItem to item101 should take place, *i.e.*, the discount offer of item101 should be approved (item101[approved $\twoheadrightarrow$ yes]).

As before inheritance from coItem to item101 enables deduction of the fact item101[discPrice $\twoheadrightarrow$ 27]. Now this newly derived fact enables value-rule (4), which in turn yields item101 : loItem. However, this leads to a contradiction: item101[approved $\twoheadrightarrow$ yes] was derived via inheritance assuming item101 is a member of coItem and no other inheritance contradicts this derivation; but based on this assumption we have inferred that item101 also belongs to loItem, which offers a different value, no, for the same method approved through inheritance!

The above scenario illustrates a very interesting problem: deduction via inheritance and via rules can enable each other; moreover, the original reasons for inheritance may be invalidated by subsequent deductions via rules which were enabled by inheritance. In the example above, our solution is to "withdraw" the inheritance from coItem to item101 and make the method discPrice *undefined* on item101. This is analogous to reasoning by contradiction in which a reasoner rejects assumptions that lead to contradictions.

Observations

Nonmonotonic Inheritance. Overriding in inheritance leads to nonmonotonic reasoning, since more specific definitions take precedence over more general ones. However, overriding is not the only source of nonmonotonicity here. When an object belongs to multiple incomparable classes, inheritance conflicts can arise and their "canceling" effects can lead to nonmonotonic inheritance as well.

Dynamic Class Hierarchy. A class hierarchy becomes *dynamic* when class membership and/or subclass relationship is defined using rules (*e.g.*, value-rules (2) and (4) in Figures 1 and 2, respectively), because it can only be decided at runtime but not compile time due to satisfiability of these rules. In such cases complex interactions can come into play between deduction via inheritance and via rules. In particular, a "bad" inheritance decision may trigger a chain of deductions via rules which eventually violate the assumptions based on the principles of overriding and multiple inheritance conflict.

Value Inheritance vs. Code Inheritance. Inheritance of instance method definitions (*e.g.*, inheritance of code-rule (1) in Figure 1) is called *code inheritance*, while inheritance of class method definitions (*e.g.*, inheritance of value-fact (3) in Figure 1) is called *value inheritance*. A fundamental difference between value and code inheritance is that the former is *data-dependent* whereas the latter is not. This difference becomes even more apparent when class method definitions are specified using rules. For instance, inheritability of the method approved defined by value-rule (5) in Figure 2 hinges on satisfiability of its rule body. Thus, if we had loItem[totalLoss $\twoheadrightarrow$ 5000] instead of loItem[totalLoss $\twoheadrightarrow$ 20000] then

the body of value-rule (5) would not be satisfied, $\mathsf{loItem}[\mathsf{approved} \twoheadrightarrow \mathsf{no}]$ would not be derived, and multiple inheritance conflict would not arise. In contrast, inheritability of instance method definitions is independent of whether a rule body is satisfied or not: it is the code itself that is inherited, not the facts derived through this code. Therefore, if value-rule (5) were a code-rule, multiple inheritance conflict would persist regardless of the actual value of totalLoss.

4 Three-Valued Semantics

The motivating example in Section 3 has illustrated the complicated interactions between deduction via inheritance and rules. This suggests the use of the stable model semantics [6] or the well-founded semantics [5] as the basis of a reasoning mechanism. In this paper we adopt the latter. Since well-founded models are three-valued and the original F-logic models were two-valued [12], we define a suitable three-valued semantics for F-logic programs first.

Let P be an F-logic program. The *Herbrand universe* of P, denoted $\mathcal{HU}_\mathrm{P}$, consists of all the *ground* (*i.e.*, variable-free) terms constructed using the function symbols and constants found in the program. The *Herbrand instantiation* of P, denoted *ground*(P), is the set of rules obtained by consistently substituting all the terms in $\mathcal{HU}_\mathrm{P}$ for all variables in every rule of P. The *Herbrand base* of P, denoted $\mathcal{HB}_\mathrm{P}$, consists of the following sorts of atoms: $\mathsf{o}:\mathsf{c}$, $\mathsf{s}::\mathsf{c}$, $\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{s}}_{\mathrm{ex}}$, $\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}$, and $\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{co}}$, where o, c, s, m, and v are terms from $\mathcal{HU}_\mathrm{P}$.

A *three-valued interpretation* $\mathcal{I}$ of an F-logic program P is a pair $\langle\, \mathrm{T}; \mathrm{U} \,\rangle$, where T and U are *disjoint* subsets of $\mathcal{HB}_\mathrm{P}$. The set T contains all atoms that are *true* whereas U contains all atoms that are *undefined*. The set F of all atoms that are *false* is defined as $\mathrm{F} = \mathcal{HB}_\mathrm{P} - (\mathrm{T} \cup \mathrm{U})$.

Following [17], we will define the truth valuation functions for atoms, literals, and value-rules. The atoms in $\mathcal{HB}_\mathrm{P}$ can take one of the three values: **t**, **f**, and **u**. Intuitively the truth value **u** means possibly true or possible false and so carries more "truth" than the truth value **f**. Therefore, the ordering among truth values is defined as follows: $\mathbf{f} < \mathbf{u} < \mathbf{t}$. An interpretation $\mathcal{I} = \langle\, \mathrm{T}; \mathrm{U} \,\rangle$ can be defined as a truth valuation function on any atom A from $\mathcal{HB}_\mathrm{P}$ as follows: (i) $\mathcal{I}(\mathsf{A}) = \mathbf{t}$, if $\mathsf{A} \in \mathrm{T}$; (ii) $\mathcal{I}(\mathsf{A}) = \mathbf{u}$, if $\mathsf{A} \in \mathrm{U}$; (iii) Otherwise, $\mathcal{I}(\mathsf{A}) = \mathbf{f}$. Moreover, for any $\mathsf{A}_\mathsf{i} \in \mathcal{HB}_\mathrm{P}$, $1 \le \mathsf{i} \le \mathsf{n}$: $\mathcal{I}(\mathsf{A}_1 \wedge \ldots \wedge \mathsf{A}_\mathsf{n}) = min\{\mathcal{I}(\mathsf{A}_\mathsf{i}) \,|\, 1 \le \mathsf{i} \le \mathsf{n}\}$.

We can extend the truth valuation function $\mathcal{I}$ to all value-rules in *ground*(P). In an interpretation of an F-logic program, atoms of the form $\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{s}}_{\mathrm{ex}}$ capture the idea that $\mathsf{m} \twoheadrightarrow \mathsf{v}$ is *explicitly defined* at s via a value-rule, while atoms of the forms $\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}$ and $\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{co}}$, where $\mathsf{o} \neq \mathsf{c}$, capture the idea that object o inherits $\mathsf{m} \twoheadrightarrow \mathsf{v}$ from class c by value and code inheritance, respectively.

The intuitive reading of a value-rule is as follows: its rule head acts as an *explicit definition* while its rule body as a *query*. In particular, if $\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]$ is in the head of a value-rule and the body of this rule is satisfied, then $\mathsf{m} \twoheadrightarrow \mathsf{v}$ is explicitly defined for s. However, in the body of a value-rule the literal $\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]$ tests whether s has an explicit definition of $\mathsf{m} \twoheadrightarrow \mathsf{v}$, or s inherits $\mathsf{m} \twoheadrightarrow \mathsf{v}$ from some superclass by either value or code inheritance. Therefore, the truth valua-

tion of a ground F-logic literal depends on whether it appears a rule head or in a rule body. The above discussion is formalized as follows.

Definition 1. *Given an interpretation $\mathcal{I}$ of an F-logic program* P*, the truth valuation functions, $\mathcal{V}^{\mathbf{h}}_{\mathcal{I}}$ and $\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}$ (**h** and **b** stand for head and body, respectively), on ground F-logic literals are defined as follows: (i) $\mathcal{V}^{\mathbf{h}}_{\mathcal{I}}(\mathsf{o}:\mathsf{c}) = \mathcal{I}(\mathsf{o}:\mathsf{c})$; (ii) $\mathcal{V}^{\mathbf{h}}_{\mathcal{I}}(\mathsf{s}::\mathsf{c}) = \mathcal{I}(\mathsf{s}::\mathsf{c})$; (iii) $\mathcal{V}^{\mathbf{h}}_{\mathcal{I}}(\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]) = \mathcal{I}(\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{s}}_{\mathrm{ex}})$; (iv) $\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{o}:\mathsf{c}) = \mathcal{I}(\mathsf{o}:\mathsf{c})$; (v) $\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{s}::\mathsf{c}) = \mathcal{I}(\mathsf{s}::\mathsf{c})$; (vi) $\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]) = max\{\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{o}}_{\mathrm{ex}}), \mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}), \mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{co}}) \mid \mathsf{c} \in \mathcal{HU}_{\mathrm{P}}\}$. Let L and L_{i} ($1 \leq \mathsf{i} \leq \mathsf{n}$) be ground literals. Then: (i) $\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\neg \mathsf{L}) = \neg\, \mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{L})$; (ii) $\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{L}_1 \wedge \ldots \wedge \mathsf{L}_{\mathsf{n}}) = min\{\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{L}_{\mathsf{i}}) \mid 1 \leq \mathsf{i} \leq \mathsf{n}\}$; where $\neg\, \mathbf{f} = \mathbf{t}$, $\neg\, \mathbf{u} = \mathbf{u}$, and $\neg\, \mathbf{t} = \mathbf{f}$.*

With the truth valuation functions $\mathcal{V}^{\mathbf{h}}_{\mathcal{I}}$ and $\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}$ for ground literals, we can define the truth valuation function $\mathcal{I}$ on ground value-rules. We should point out that although $\mathcal{I}$ is three-valued when applied to ground atoms, it becomes *two-valued* when applied to ground value-rules. Intuitively, a ground value-rule is evaluated to be true if and only if the truth value of its rule head is greater than or equal to that of its rule body. Formally, we have the following definition.

Definition 2. *Given an interpretation $\mathcal{I}$ of an F-logic program* P*, the truth valuation function $\mathcal{I}$ on a ground value-rule, $\mathsf{H} \longleftarrow \mathsf{B}$, in ground*(P)*, is defined as follows: $\mathcal{I}(\mathsf{H} \longleftarrow \mathsf{B}) = \mathbf{t}$, if $\mathcal{V}^{\mathbf{h}}_{\mathcal{I}}(\mathsf{H}) \geq \mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{B})$; otherwise, $\mathcal{I}(\mathsf{H} \longleftarrow \mathsf{B}) = \mathbf{f}$. Similarly, given a ground value-fact, H, in ground*(P)*: $\mathcal{I}(\mathsf{H}) = \mathbf{t}$ iff $\mathcal{V}^{\mathbf{h}}_{\mathcal{I}}(\mathsf{H}) = \mathbf{t}$.*

We will say that a three-valued interpretation satisfies the value-rules of an F-logic program, if it satisfies all the ground value-rules of this program.

Definition 3 (Value-Rule Satisfaction). *A three-valued interpretation $\mathcal{I}$* satisfies *the value-rules of an F-logic program* P*, if $\mathcal{I}(\mathsf{R}) = \mathbf{t}$ for every value-rule R in ground*(P)*.*

5 Inheritance Postulates

Even if an interpretation $\mathcal{I}$ satisfies all the value-rules of an F-logic program P, it does not necessarily mean that $\mathcal{I}$ is an intended *object model* of P, because $\mathcal{I}$ must also include facts that are derived via inheritance. F-logic programs only specify class hierarchies and method definitions — what needs to be inherited is not explicitly stated. In fact, as we saw in Section 3, defining exactly what should be inherited is a subtle issue. In our framework, it is the job of the *inheritance postulates*, which embody the common intuition behind inheritance.

First we will introduce the notion of inheritance candidacy. The various concepts to be defined in this section come with two flavors: *strong* or *weak*. The "strong" flavor of a concept requires that all relevant facts be positively established while the "weak" flavor allows some or all facts to be undefined.

Definition 4 (Explicit Definition). *Given an interpretation $\mathcal{I}$ of an F-logic program* P*, $\mathsf{s}[\mathsf{m}]$ is a strong explicit definition, if $max\{\mathcal{I}(\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{s}}_{\mathrm{ex}}) | \mathsf{v} \in \mathcal{HU}_{\mathrm{P}}\} = \mathbf{t}$; $\mathsf{s}[\mathsf{m}]$ is a weak explicit definition if $max\{\mathcal{I}(\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{s}}_{\mathrm{ex}}) | \mathsf{v} \in \mathcal{HU}_{\mathrm{P}}\} = \mathbf{u}$.*

Definition 5 (Value Inheritance Context). *Given an interpretation $\mathcal{I}$ of an F-logic program* P, $\mathsf{c}[\mathsf{m}]$ *is a strong value inheritance context for* o, *if* $\mathsf{c} \neq \mathsf{o}$ [2] *and* $min\{\mathcal{I}(\mathsf{o}:\mathsf{c}), max\{\mathsf{c}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{ex}} | \mathsf{v} \in \mathcal{HU}_{\mathrm{P}}\}\} = \mathbf{t}$*;* $\mathsf{c}[\mathsf{m}]$ *is a weak value inheritance context for* o, *if* $\mathsf{c} \neq \mathsf{o}$ *and* $min\{\mathcal{I}(\mathsf{o}:\mathsf{c}), max\{\mathsf{c}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{ex}} | \mathsf{v} \in \mathcal{HU}_{\mathrm{P}}\}\} = \mathbf{u}$ *(roughly speaking, if* o *is a proper member of* c *and* $\mathsf{c}[\mathsf{m}]$ *is an explicit definition).*

Definition 6 (Code Inheritance Context). *Given an interpretation $\mathcal{I}$ of an F-logic program* P, $\mathsf{c}[\mathsf{m}]$ *is a strong (weak) code inheritance context for* o, *if* $\mathsf{c} \neq \mathsf{o}$, $\mathcal{I}(\mathsf{o}:\mathsf{c}) = \mathbf{t}$ *($\mathcal{I}(\mathsf{o}:\mathsf{c}) = \mathbf{u}$), and there is a code-rule in* P *which specifies the instance method* m *for the class* c.

Note that explicit definitions can only be established via value-rules but not code-rules. The difference between a value and a code inheritance context is that the former requires at least one value be established for its class method via a value-rule, whereas the latter only requires the presence of at least one code-rule which specifies its instance method. Generally we will use the term *inheritance context* to refer to either a value or a code inheritance context. In the following definitions we will see that value and code inheritance contexts are treated equally as far as overriding is concerned.

Definition 7 (Overriding). *Given an interpretation $\mathcal{I}$ of an F-logic program* P, *the class* s *strongly overrides* $\mathsf{c}[\mathsf{m}]$ *for* o, *if* $\mathsf{s} \neq \mathsf{c}$, $\mathcal{I}(\mathsf{s}::\mathsf{c}) = \mathbf{t}$, *and* $\mathsf{s}[\mathsf{m}]$ *is either a strong value or a strong code inheritance context for* o.

The class s *weakly overrides* $\mathsf{c}[\mathsf{m}]$ *for* o *if the above conditions are relaxed by allowing* $\mathsf{s}::\mathsf{c}$ *to be undefined and/or allowing* $\mathsf{s}[\mathsf{m}]$ *to be a weak inheritance context. Formally this means that either: (i)* $\mathcal{I}(\mathsf{s}::\mathsf{c}) = \mathbf{t}$ *and* $\mathsf{s}[\mathsf{m}]$ *is a weak inheritance context for* o*; or (ii)* $\mathcal{I}(\mathsf{s}::\mathsf{c}) = \mathbf{u}$ *and* $\mathsf{s}[\mathsf{m}]$ *is either a weak or a strong inheritance context for* o.

Definition 8 (Value Inheritance Candidate). *Given an interpretation $\mathcal{I}$ of an F-logic program* P, $\mathsf{c}[\mathsf{m}]$ *is a strong value inheritance candidate for* o, *denoted* $\mathsf{c}[\mathsf{m}] \overset{sv}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$, *if* $\mathsf{c}[\mathsf{m}]$ *is a strong value inheritance context for* o *and there is no* s *that strongly or weakly overrides* $\mathsf{c}[\mathsf{m}]$ *for* o.

$\mathsf{c}[\mathsf{m}]$ *is a weak value inheritance candidate for* o, *denoted* $\mathsf{c}[\mathsf{m}] \overset{wv}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$, *if the above conditions are relaxed by allowing* $\mathsf{c}[\mathsf{m}]$ *to be a weak value inheritance context and/or allowing weak overriding. Formally, this means that there is no* s *that strongly overrides* $\mathsf{c}[\mathsf{m}]$ *for* o *and either: (i)* $\mathsf{c}[\mathsf{m}]$ *is a weak value inheritance context for* o*; or (ii)* $\mathsf{c}[\mathsf{m}]$ *is a strong value inheritance context for* o *and there is* s *that weakly overrides* $\mathsf{c}[\mathsf{m}]$ *for* o.

Definition 9 (Code Inheritance Candidate). *Given an interpretation $\mathcal{I}$ of an F-logic program* P, $\mathsf{c}[\mathsf{m}]$ *is a strong code inheritance candidate for* o, *denoted* $\mathsf{c}[\mathsf{m}] \overset{sc}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$, *if* $\mathsf{c}[\mathsf{m}]$ *is a strong code inheritance context for* o *and there is no* s *that strongly or weakly overrides* $\mathsf{c}[\mathsf{m}]$ *for* o.

$\mathsf{c}[\mathsf{m}]$ *is a weak code inheritance candidate for* o, *denoted* $\mathsf{c}[\mathsf{m}] \overset{wc}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$, *if the above conditions are relaxed by allowing* $\mathsf{c}[\mathsf{m}]$ *to be a weak code inheritance context and/or allowing weak overriding. Formally, this means that there is no* s

[2] $\mathsf{c} \neq \mathsf{o}$ means that c and o are distinct terms.

that strongly overrides $\mathsf{c}[\mathsf{m}]$ *for* o *and either: (i)* $\mathsf{c}[\mathsf{m}]$ *is a weak code inheritance context for* o*; or (ii)* $\mathsf{c}[\mathsf{m}]$ *is a strong code inheritance context for* o *and there is* s *that weakly overrides* $\mathsf{c}[\mathsf{m}]$ *for* o.

Example 1. As an example, consider an interpretation $\mathcal{I} = \langle \mathrm{T}; \mathrm{U} \rangle$ of an F-logic program P, where $\mathrm{T} = \{\mathsf{c}_1 : \mathsf{c}_2, \mathsf{c}_1 : \mathsf{c}_4, \mathsf{c}_1 : \mathsf{c}_5, \mathsf{c}_2 :: \mathsf{c}_4, \mathsf{c}_3 :: \mathsf{c}_5\} \cup \{\mathsf{c}_2[\mathsf{m} \twoheadrightarrow \mathsf{a}]^{\mathsf{c}_2}_{\mathrm{ex}}, \mathsf{c}_3[\mathsf{m} \twoheadrightarrow \mathsf{b}]^{\mathsf{c}_3}_{\mathrm{ex}}, \mathsf{c}_4[\mathsf{m} \twoheadrightarrow \mathsf{c}]^{\mathsf{c}_4}_{\mathrm{ex}}\}$ and $\mathrm{U} = \{\mathsf{c}_1 : \mathsf{c}_3\}$. $\mathcal{I}$ and P are shown in Figure 3, where solid and dashed arrows represent true and undefined values, respectively.

In the interpretation $\mathcal{I}$, $\mathsf{c}_2[\mathsf{m}]$ and $\mathsf{c}_4[\mathsf{m}]$ are strong value inheritance contexts for c_1. $\mathsf{c}_5[\mathsf{m}]$ is a strong code inheritance context for c_1. $\mathsf{c}_3[\mathsf{m}]$ is a weak value inheritance context for c_1. The class c_2 strongly overrides $\mathsf{c}_4[\mathsf{m}]$ for c_1, while c_3 weakly overrides $\mathsf{c}_5[\mathsf{m}]$ for c_1. The context $\mathsf{c}_2[\mathsf{m}]$ is a strong value inheritance candidate for c_1, while $\mathsf{c}_3[\mathsf{m}]$ is a weak value inheritance candidate and $\mathsf{c}_5[\mathsf{m}]$ is a weak code inheritance candidate for c_1. Finally, $\mathsf{c}_4[\mathsf{m}]$ is neither a strong nor a weak value inheritance candidate for c_1.

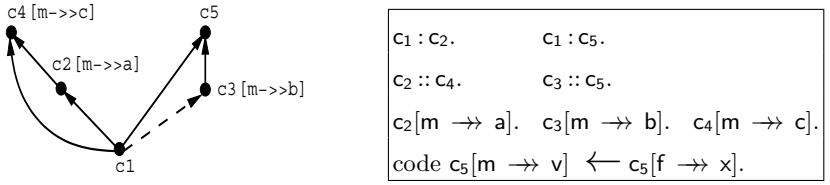

Fig. 3. Inheritance Context, Overriding, and Inheritance Candidate.

For convenience, we will simply write $\mathsf{c}[\mathsf{m}] \rightsquigarrow_{\mathcal{I}} \mathsf{o}$ when it does not matter whether $\mathsf{c}[\mathsf{m}]$ is a strong or a weak value or code inheritance candidate. Now we are ready to introduce the postulates for nonmonotonic multiple value and code inheritance. The inheritance postulates consist of two parts: core inheritance postulates and optimistic inheritance postulates. We formalize the core inheritance postulates first.

Definition 10 (Positive ISA Transitivity). *An interpretation* $\mathcal{I}$ *of an F-logic program* P *satisfies the positive ISA transitivity constraint if the positive part of the class hierarchy is transitively closed, formally, if the following two conditions hold: (1) for all* s, c*: if there is* x *such that* $\mathcal{I}(\mathsf{s} :: \mathsf{x}) = \mathbf{t}$ *and* $\mathcal{I}(\mathsf{x} :: \mathsf{c}) = \mathbf{t}$*, then* $\mathcal{I}(\mathsf{s} :: \mathsf{c}) = \mathbf{t}$*; (2) for all* o, c*: if there is* x *such that* $\mathcal{I}(\mathsf{o} : \mathsf{x}) = \mathbf{t}$ *and* $\mathcal{I}(\mathsf{x} :: \mathsf{c}) = \mathbf{t}$*, then* $\mathcal{I}(\mathsf{o} : \mathsf{c}) = \mathbf{t}$.

Definition 11 (Context Consistency). *An interpretation* $\mathcal{I}$ *of an F-logic program* P *satisfies the context consistency constraint, if the following conditions hold: (1) for all* $\mathsf{o}, \mathsf{m}, \mathsf{v}$*:* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{o}}_{\mathrm{va}}) = \mathbf{f}$ *and* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{o}}_{\mathrm{co}}) = \mathbf{f}$*; (2) for all* $\mathsf{c}, \mathsf{m}, \mathsf{v}$*: if* $\mathcal{I}(\mathsf{c}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{ex}}) = \mathbf{f}$*, then* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}) = \mathbf{f}$ *for all* o*; (3) for all* c, m*: if there is no code-rule in ground*(P) *which specifies the instance method* m *for the class* c*, then* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{co}}) = \mathbf{f}$ *for all* o, v*; (4) for all* o, m*: if* $\mathsf{o}[\mathsf{m}]$ *is a strong explicit definition, then* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}) = \mathbf{f}$ *and* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{co}}) = \mathbf{f}$ *for all* v, c.

The context consistency constraint captures the implications of specificity. The first condition rules out self inheritance. The second condition states that if $\mathsf{m} \twoheadrightarrow \mathsf{v}$ is not explicitly defined at c, then no class should inherit $\mathsf{m} \twoheadrightarrow \mathsf{v}$ from c by value inheritance. The third condition states that if a class c does not specify an instance method m, then no object should inherit any value of m from c by code inheritance. The fourth condition states that if $\mathsf{m} \twoheadrightarrow \mathsf{v}$ is explicitly defined at o, then this definition should prevent o from inheriting any value of m from other classes (by either value or code inheritance).

The following constraint captures the semantics of nonmonotonic multiple inheritance.

Definition 12 (Unique Source Inheritance). *An interpretation $\mathcal{I}$ of an F-logic program* P *satisfies the unique source inheritance constraint, if all of the following three conditions hold: (1) for all* $\mathsf{o}, \mathsf{m}, \mathsf{v}, \mathsf{c}$*: if* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}) = \mathbf{t}$ *or* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{co}}) = \mathbf{t}$*, then* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{z}]^{\mathsf{x}}_{\mathrm{va}}) = \mathbf{f}$ *and* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{z}]^{\mathsf{x}}_{\mathrm{co}}) = \mathbf{f}$ *for all* z, x *where* $\mathsf{x} \neq \mathsf{c}$*; (2) for all* $\mathsf{c}, \mathsf{m}, \mathsf{o}$*: if* $\mathsf{c}[\mathsf{m}] \overset{sv}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$ *or* $\mathsf{c}[\mathsf{m}] \overset{sc}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$*, then* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{x}}_{\mathrm{va}}) = \mathbf{f}$ *and* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{x}}_{\mathrm{co}}) = \mathbf{f}$ *for all* v*,* x *such that* $\mathsf{x} \neq \mathsf{c}$*; (3) for all* $\mathsf{o}, \mathsf{m}, \mathsf{v}, \mathsf{c}$*:* $\mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}) = \mathbf{t}$ *iff (i)* $\mathsf{o}[\mathsf{m}]$ *is neither a strong nor a weak explicit definition; and (ii)* $\mathsf{c}[\mathsf{m}] \overset{sv}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$*; and (iii)* $\mathcal{I}(\mathsf{c}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{ex}}) = \mathbf{t}$*; and (iv) there is no* x *such that* $\mathsf{x} \neq \mathsf{c}$ *and* $\mathsf{x}[\mathsf{m}] \rightsquigarrow_{\mathcal{I}} \mathsf{o}$*.*

Intuitively, we want our semantics for inheritance to have such a property that if inheritance is allowed, then it should take place from a *unique* source. The first condition above implies that positive value or code inheritance from a class should prevent any inheritance from other classes. The second condition states that if a strong value or code inheritance candidate, $\mathsf{c}[\mathsf{m}]$, exists, then inheritance of the method m cannot take place from any other source (because there would be a multiple inheritance conflict). The third condition specifies when "positive" value inheritance takes place. An object o inherits $\mathsf{m} \twoheadrightarrow \mathsf{v}$ from a class c by value inheritance iff: (i) no value is explicitly defined for the method m at o; (ii) $\mathsf{c}[\mathsf{m}]$ is a strong value inheritance candidate for o; (iii) $\mathsf{m} \twoheadrightarrow \mathsf{v}$ is explicitly defined at c and is positive; and (iv) there are no other inheritance candidates — weak or strong — from which o could inherit the method m.

We should point out that the constraints introduced so far only capture the intuition behind the "definite" part of an object model (the notion of an object model is formalized in Section 6), *i.e.*, the true and the false components. We view them as *core inheritance postulates* that any reasonable object model must obey. However, we still need to assign a meaning to the undefined part of an object model. Since "undefined" means possibly true or possibly false, intuitively we want the conclusions drawn from undefined facts to remain undefined, *i.e.*, the semantics should be "closed" with respect to undefined facts. Therefore, although it might seem tempting to "jump" to negative conclusions from undefined facts in some cases (*e.g.*, if there are multiple weak inheritance candidates), our semantics is biased towards undefined conclusions, which is why we call it "optimistic". Due to want of space, we will omit the optimistic inheritance postulates here. Their definitions can be found in [22].

6 Object Models

A model of an F-logic program should satisfy all the rules in it. In Section 4 we have formalized the notion of value-rule satisfaction. Here we will extend this notion to code-rules. First note that when an object inherits an instance method definition, *i.e.*, a code-rule, it will be evaluated in the context of this object. This corresponds to the idea of *late binding* in imperative object-oriented languages like C^{++} and Java.

Definition 13 (Binding). *Let* $\mathsf{R} \equiv (\mathsf{code\ c[m} \twoheadrightarrow \mathsf{v]} \longleftarrow \mathsf{B})$ *be a ground code-rule which specifies the instance method* m *for the class* c. *The* binding *of* R *with respect to an object* o, *denoted* $\mathsf{R||o}$, *is obtained from* R *by substituting* o *for every occurrence of* c *in* R. *We will use* $\mathsf{X}_{\mathsf{c}\backslash\mathsf{o}}$ *to represent the term that is obtained from* X *by substituting* o *for every occurrence of* c *in* X.

Therefore, the truth valuation function will be defined on *bindings* of ground code-rules instead of on code-rules directly. When an object inherits code-rules from a class, the bindings of these code-rules with respect to this object should be satisfied similarly to value-rules. However, because only those code-rules which are inherited need to be satisfied, satisfaction of code-rules depends on how they are inherited: strongly or weakly.

Definition 14 (Strong Code Inheritance). *Let* $\mathcal{I}$ *be an interpretation of an F-logic program* P *and* $\mathsf{R} \equiv (\mathsf{code\ c[m} \twoheadrightarrow \mathsf{v]} \longleftarrow \mathsf{B})$ *a code-rule in ground*(P). *An object* o *strongly inherits* R, *if the following conditions hold: (1)* $\mathsf{c[m]} \overset{sc}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$; *(2)* o[m] *is neither a strong nor a weak explicit definition; (3) there is no* $\mathsf{x} \neq \mathsf{c}$ *such that* $\mathsf{x[m]} \rightsquigarrow_{\mathcal{I}} \mathsf{o}$.

Definition 15 (Weak Code Inheritance). *Let* $\mathcal{I}$ *be an interpretation of an F-logic program* P *and* $\mathsf{R} \equiv (\mathsf{code\ c[m} \twoheadrightarrow \mathsf{v]} \longleftarrow \mathsf{B})$ *a code-rule in ground*(P). *An object* o *weakly inherits* R, *if the following conditions hold: (1)* $\mathsf{c[m]} \overset{sc}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$ *or* $\mathsf{c[m]} \overset{wc}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$; *(2)* o[m] *is not a strong explicit definition; (3) there is no* $\mathsf{x} \neq \mathsf{c}$ *such that* $\mathsf{x[m]} \overset{sv}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$ *or* $\mathsf{x[m]} \overset{sc}{\rightsquigarrow}_{\mathcal{I}} \mathsf{o}$; *(4)* o *does not strongly inherit* R.

We can define a function, $imode_{\mathcal{I}}$, on bindings of ground code-rules, which returns the "inheritance mode" of a binding: (i) $imode_{\mathcal{I}}(\mathsf{R||o}) = \mathbf{t}$, if o strongly inherits R; (ii) $imode_{\mathcal{I}}(\mathsf{R||o}) = \mathbf{u}$, if o weakly inherits R; (iii) $imode_{\mathcal{I}}(\mathsf{R||o}) = \mathbf{f}$, otherwise. Now we can extend the truth valuation function to ground code-rules as follows.

Definition 16. *Let* $\mathcal{I}$ *be an interpretation,* $\mathsf{R} \equiv (\mathsf{code\ c[m} \twoheadrightarrow \mathsf{v]} \longleftarrow \mathsf{B})$ *be a ground code-rule and* $\mathsf{F} \equiv (\mathsf{code\ c[m} \twoheadrightarrow \mathsf{v]})$ *a ground C-fact. The truth valuation function* $\mathcal{I}$ *on* $\mathsf{R||o}$ *and* $\mathsf{F||o}$ *(the bindings of* R *and* F *with respect to* o, *respectively) is defined as follows:*

$$\mathcal{I}(\mathsf{R||o}) = \begin{cases} \mathbf{t}, \textit{ if } imode_{\mathcal{I}}(\mathsf{R||o}) \geq \mathbf{u} \textit{ and} \\ \quad \mathcal{I}(\mathsf{o[m} \twoheadrightarrow \mathsf{v]}^{\,\mathsf{c}}_{\,\mathsf{co}}) \geq min\{\mathcal{V}^{\mathbf{b}}_{\mathcal{I}}(\mathsf{B}_{\mathsf{c}\backslash\mathsf{o}}), imode_{\mathcal{I}}(\mathsf{R||o})\}; \\ \mathbf{t}, \textit{ if } imode_{\mathcal{I}}(\mathsf{R||o}) = \mathbf{f} \textit{ and } \mathcal{I}(\mathsf{o[m} \twoheadrightarrow \mathsf{v]}^{\,\mathsf{c}}_{\,\mathsf{co}}) = \mathbf{f}; \\ \mathbf{f}, \textit{ otherwise.} \end{cases}$$

$$\mathcal{I}(\mathsf{F}||\mathsf{o}) = \begin{cases} \mathbf{t}, \textit{ if } imode_{\mathcal{I}}(\mathsf{R}||\mathsf{o}) \geq \mathbf{u} \textit{ and } \mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\,\mathsf{c}}_{\mathrm{co}}) \geq imode_{\mathcal{I}}(\mathsf{R}||\mathsf{o}); \\ \mathbf{t}, \textit{ if } imode_{\mathcal{I}}(\mathsf{R}||\mathsf{o}) = \mathbf{f} \textit{ and } \mathcal{I}(\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\,\mathsf{c}}_{\mathrm{co}}) = \mathbf{f}; \\ \mathbf{f}, \textit{ otherwise.} \end{cases}$$

Recall that atoms of the form $\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\,\mathsf{c}}_{\mathrm{co}}$ represent those facts that are derived via code inheritance. Moreover, observe that in the case of strong code inheritance, the truth valuation function on code-rules will be defined essentially the same way as on value-rules. Now the idea of code-rule satisfaction and the notion of an *object model* can be formalized as follows.

Definition 17 (Code-Rule Satisfaction). *A three-valued interpretation $\mathcal{I}$ satisfies the code-rules of an F-logic program* P*, if $\mathcal{I}(\mathsf{R}||\mathsf{o}) = \mathbf{t}$ for all code-rule $\mathsf{R} \in ground(\mathrm{P})$ and all $\mathsf{o} \in \mathcal{HU}_{\mathrm{P}}$.*

Definition 18 (Object Model). *An interpretation $\mathcal{I}$ is called an object model of an F-logic program* P*, if $\mathcal{I}$ satisfies both the value-rules and the code-rules in* P*, plus the core inheritance postulates: the positive ISA transitivity constraint, the context consistency constraint, and the unique source inheritance constraint.*

7 Optimistic Object Models

In this section we introduce a particular object model, called *optimistic object model*, which is *uniquely* defined for *any* F-logic program and satisfies all the inheritance postulates defined in Section 5, including the optimistic inheritance postulates. First we will present a *procedural* characterization of optimistic object models. Due to space limitation, here we will only sketch the main concepts. The reader is referred to [22] for more details.

Computation of optimistic object models extends the alternating fixpoint process in [4]. The new element here is the book-keeping mechanism for recording inheritance information. This book-keeping information will be projected out when the final object model is generated. The main component of the computation is an *antimonotonic* operator, Ψ_{P}, which takes an F-logic program P as input and performs implicit deduction via value and code inheritance as well as explicit deduction via rules. Based on the operator Ψ_{P} we define another operator, $\mathbf{F}_{\mathrm{P}} \stackrel{\text{def}}{=} \Psi_{\mathrm{P}} \cdot \Psi_{\mathrm{P}}$, which is *monotonic* and hence has a unique least fixpoint, denoted $\mathrm{lfp}(\mathbf{F}_{\mathrm{P}})$. The definition of optimistic object models is stated as follows.

Definition 19 (Optimistic Object Model). *The optimistic object model, $\mathcal{M}$, of an F-logic program* P *is defined as follows: $\mathcal{M} = \langle\, \mathrm{T}; \mathrm{U} \,\rangle$, where $\mathrm{T} = lfp(\mathbf{F}_{\mathrm{P}})$ and $\mathrm{U} = \Psi_{\mathrm{P}}(lfp(\mathbf{F}_{\mathrm{P}})) - lfp(\mathbf{F}_{\mathrm{P}})$.*

Theorem 1. *The optimistic object model $\mathcal{M}$ of an F-logic program* P *is an object model of* P*. Moreover, it satisfies the optimistic inheritance postulates.*

It turns out that the (unique) optimistic object model of an F-logic program P can be computed as the well-founded model of a certain general logic program with negation, which is obtained from P by rewriting. Before describing the rewriting procedure we first define a rewriting function that applies to all value-rules and code-rules.

Definition 20. *Given an F-logic program* P *and a literal* L *in* P*, the functions* ρ^h *and* ρ^b *that rewrite head and body literals in* P *are defined as follows:*

$$\rho^h(\mathsf{L}) = \begin{cases} isa(\mathsf{o},\mathsf{c}), & \text{if } \mathsf{L} = \mathsf{o}:\mathsf{c} \\ sub(\mathsf{s},\mathsf{c}), & \text{if } \mathsf{L} = \mathsf{s}::\mathsf{c} \\ exmv(\mathsf{s},\mathsf{m},\mathsf{v}), & \text{if } \mathsf{L} = \mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}] \end{cases}$$

$$\rho^b(\mathsf{L}) = \begin{cases} isa(\mathsf{o},\mathsf{c}), & \text{if } \mathsf{L} = \mathsf{o}:\mathsf{c} \\ sub(\mathsf{s},\mathsf{c}), & \text{if } \mathsf{L} = \mathsf{s}::\mathsf{c} \\ mv(\mathsf{o},\mathsf{m},\mathsf{v}), & \text{if } \mathsf{L} = \mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}] \\ \neg(\rho^b(\mathrm{G})), & \text{if } \mathsf{L} = \neg\,\mathsf{G} \end{cases}$$

The rewriting function ρ *on value-rules and code-rules in* P *is defined as follows:*

$$\rho(\mathsf{H} \longleftarrow \mathsf{L}_1,\ldots,\mathsf{L}_n) = \rho^h(\mathsf{H}) \longleftarrow \rho^b(\mathsf{L}_1),\ldots,\rho^b(\mathsf{L}_n)$$

$$\rho(\mathsf{code}\quad \mathsf{c}[\mathsf{m} \twoheadrightarrow \mathsf{v}] \longleftarrow \mathsf{L}_1,\ldots,\mathsf{L}_n) = ins(\mathsf{O},\mathsf{m},\mathsf{v},\mathsf{c}) \longleftarrow \rho^b(\mathsf{B}_1),\ldots,\rho^b(\mathsf{B}_n)$$

where O *is a new variable that does not appear in* P *and each* $\mathsf{B_i} = (\mathsf{L_i})_{\mathsf{c}\backslash\mathsf{O}}$ $\mathsf{B_i}$ *is obtained from* $\mathsf{L_i}$ *by substituting* O *for all occurrences of* c*. The predicates isa, sub, exmv, mv, and ins are auxiliary predicates introduced by the rewriting.*

Note that because literals in rule heads and bodies have different meanings, they are rewritten differently. Moreover, literals in the heads of value-rules and in the heads of code-rules are also rewritten differently. The rewriting procedure that transforms F-logic programs into general logic programs is defined below.

Definition 21 (Well-Founded Rewriting). *The well-founded rewriting of an F-logic program* P*, denoted* P^{wf}*, is a general logic program constructed by the following steps: (1) For every value-rule* R *in* P*, add its rewriting* $\rho(\mathsf{R})$ *into* P^{wf}*; (2) For every code-rule* R *in* P*, which specifies an instance method* m *for a class* c*, add its rewriting* $\rho(\mathsf{R})$ *into* P^{wf}*. Moreover, add a fact* $codedef(\mathsf{c},\mathsf{m})$ *into* P^{wf}*; (3) Include the trailer rules shown in Figure 4 to* P^{wf} *(note that uppercase letters denote variables in these trailer rules).*

Note that while rewriting an F-logic program into a general logic program, we need to output facts of the form $codedef(\mathsf{c},\mathsf{m})$ to remember that there is a code-rule specifying the instance method m for the class c. Such facts are used to derive overriding and code inheritance candidacy information.

There is a unique *well-founded model* for any general logic program [5]. Let P^{wf} be the well-founded rewriting of an F-logic program P. We can define an *isomorphism* between the well-founded model of P^{wf} and the optimistic object model of P based on an one-to-one mapping between the following atoms: (i) $isa(\mathsf{o},\mathsf{c})$ and $\mathsf{o}:\mathsf{c}$; (ii) $sub(\mathsf{s},\mathsf{c})$ and $\mathsf{s}::\mathsf{c}$; (iii) $exmv(\mathsf{s},\mathsf{m},\mathsf{v})$ and $\mathsf{s}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{s}}_{\mathrm{ex}}$; (iv) $vamv(\mathsf{o},\mathsf{m},\mathsf{v},\mathsf{c})$ and $\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{va}}$; (v) $comv(\mathsf{o},\mathsf{m},\mathsf{v},\mathsf{c})$ and $\mathsf{o}[\mathsf{m} \twoheadrightarrow \mathsf{v}]^{\mathsf{c}}_{\mathrm{co}}$. Note that while defining the isomorphism we have projected out other bookkeeping information for inheritance. As stated in the following theorem, We have proved that these two models are indeed isomorphic.

Theorem 2. *Given the well-founded rewriting* P^{wf} *of an F-logic program* P*, the well-founded model of* P^{wf} *is isomorphic to the optimistic object model of* P*.*

$$
\begin{array}{rl}
mv(\mathsf{O},\mathsf{M},\mathsf{V}) & \longleftarrow exmv(\mathsf{O},\mathsf{M},\mathsf{V}).\\
mv(\mathsf{O},\mathsf{M},\mathsf{V}) & \longleftarrow vamv(\mathsf{O},\mathsf{M},\mathsf{V},\mathsf{C}).\\
mv(\mathsf{O},\mathsf{M},\mathsf{V}) & \longleftarrow comv(\mathsf{O},\mathsf{M},\mathsf{V},\mathsf{C}).\\
sub(\mathsf{S},\mathsf{C}) & \longleftarrow sub(\mathsf{S},\mathsf{X}),\ sub(\mathsf{X},\mathsf{C}).\\
isa(\mathsf{O},\mathsf{C}) & \longleftarrow isa(\mathsf{O},\mathsf{S}),\ sub(\mathsf{S},\mathsf{C}).\\
vamv(\mathsf{O},\mathsf{M},\mathsf{V},\mathsf{C}) & \longleftarrow vacan(\mathsf{C},\mathsf{M},\mathsf{O}),\ exmv(\mathsf{C},\mathsf{M},\mathsf{V}),\ \neg ex(\mathsf{O},\mathsf{M}),\ \neg multi(\mathsf{C},\mathsf{M},\mathsf{O}).\\
comv(\mathsf{O},\mathsf{M},\mathsf{V},\mathsf{C}) & \longleftarrow cocan(\mathsf{C},\mathsf{M},\mathsf{O}),\ ins(\mathsf{O},\mathsf{M},\mathsf{V},\mathsf{C}),\ \neg ex(\mathsf{O},\mathsf{M}),\ \neg multi(\mathsf{C},\mathsf{M},\mathsf{O}).\\
vacan(\mathsf{C},\mathsf{M},\mathsf{O}) & \longleftarrow isa(\mathsf{O},\mathsf{C}),\ exmv(\mathsf{C},\mathsf{M},\mathsf{V}),\ \mathsf{C}\neq\mathsf{O},\ \neg override(\mathsf{C},\mathsf{M},\mathsf{O}).\\
cocan(\mathsf{C},\mathsf{M},\mathsf{O}) & \longleftarrow isa(\mathsf{O},\mathsf{C}),\ codedef(\mathsf{C},\mathsf{M}),\ \mathsf{C}\neq\mathsf{O},\ \neg override(\mathsf{C},\mathsf{M},\mathsf{O}).\\
ex(\mathsf{O},\mathsf{M}) & \longleftarrow exmv(\mathsf{O},\mathsf{M},\mathsf{V}).\\
multi(\mathsf{C},\mathsf{M},\mathsf{O}) & \longleftarrow vacan(\mathsf{X},\mathsf{M},\mathsf{O}),\ \mathsf{X}\neq\mathsf{C}.\\
multi(\mathsf{C},\mathsf{M},\mathsf{O}) & \longleftarrow cocan(\mathsf{X},\mathsf{M},\mathsf{O}),\ \mathsf{X}\neq\mathsf{C}.\\
override(\mathsf{C},\mathsf{M},\mathsf{O}) & \longleftarrow sub(\mathsf{X},\mathsf{C}),\ isa(\mathsf{O},\mathsf{X}),\ exmv(\mathsf{X},\mathsf{M},\mathsf{V}),\ \mathsf{X}\neq\mathsf{C},\ \mathsf{X}\neq\mathsf{O}.\\
override(\mathsf{C},\mathsf{M},\mathsf{O}) & \longleftarrow sub(\mathsf{X},\mathsf{C}),\ isa(\mathsf{O},\mathsf{X}),\ codedef(\mathsf{X},\mathsf{M}),\ \mathsf{X}\neq\mathsf{C},\ \mathsf{X}\neq\mathsf{O}.
\end{array}
$$

Fig. 4. Trailer Rules for Well-Founded Rewriting.

Clearly, given an F-logic program P, generation of P^{wf} takes time linear in the size of P. Note that the trailer rules in Figure 4 are fixed for an given F-logic program. Therefore, the size of P^{wf} is also linear in the size of the original F-logic program P. This observation combined with Theorem 2 essentially leads to the following claim about the data complexity [21] of our inheritance semantics.

Corollary 1. *The data complexity of the optimistic object model semantics for function-free F-logic programs is polynomial time.*

Theorems 1 and 2 give procedural characterization of the optimistic object model as the least fixpoint of extended alternating fixpoint computation. Next we will present a different characterization of the optimistic object model semantics based on the so called *truth ordering* among object models[3].

It is common to compare different models of a program based on the amount of "truth" contained in the models. Typically, the true component of a model is minimized and the false component maximized. However, in F-logic we also need to deal with inheritance, which complicates the matters somewhat, because a fact may be derived via inheritance. As a consequence, there exist object models that look similar but actually are incomparable. This leads to the following definition of truth ordering among object models, which minimizes not only the set of true atoms of an object model, but also the amount of positive inheritance information implied by the object model.

Definition 22 (Truth Ordering). *Let* $\mathcal{I}_1 = \langle \mathrm{P}_1; \mathrm{Q}_1 \rangle$ *and* $\mathcal{I}_2 = \langle \mathrm{P}_2; \mathrm{Q}_2 \rangle$ *be two object models of an F-logic program* P*. We write* $\mathcal{I}_1 \leq \mathcal{I}_2$ *iff (i)* $\mathrm{P}_1 \subseteq \mathrm{P}_2$*; and (ii)* $\mathrm{P}_1 \cup \mathrm{Q}_1 \subseteq \mathrm{P}_2 \cup \mathrm{Q}_2$*; and (iii) for all* $\mathsf{c},\mathsf{m},\mathsf{o}$*:* $\mathsf{c}[\mathsf{m}] \overset{sv}{\rightsquigarrow}_{\mathcal{I}_1} \mathsf{o}$ *implies* $\mathsf{c}[\mathsf{m}] \overset{sv}{\rightsquigarrow}_{\mathcal{I}_2} \mathsf{o}$*; and (iv) for all* $\mathsf{c},\mathsf{m},\mathsf{o}$*:* $\mathsf{c}[\mathsf{m}] \overset{sc}{\rightsquigarrow}_{\mathcal{I}_1} \mathsf{o}$ *implies* $\mathsf{c}[\mathsf{m}] \overset{sc}{\rightsquigarrow}_{\mathcal{I}_2} \mathsf{o}$.

[3] We can also define *stable* object models and a different ordering, called *information ordering*, among object models. We have also shown that the optimistic object model of an F-logic program is the *least* stable object model of this program with respect to information ordering. See [22] for more details.

Definition 23 (Minimal Object Model). *An object model $\mathcal{I}$ is minimal iff there exists no object model $\mathcal{J}$ such that $\mathcal{J} \leq \mathcal{I}$ and $\mathcal{J} \neq \mathcal{I}$.*

The above definitions minimize the number of strong inheritance candidates implied by an object model *in addition to* the usual minimization of truth and maximization of falsehood. This is needed because increasing the number of false facts might inflate the number of strong inheritance candidates (due to nonmonotonic inheritance), which in turn might inflate the number of facts that are derived by inheritance. Nevertheless it turns out that the optimistic object models are minimal.

Theorem 3. *The optimistic object model of an F-logic program* P *is minimal among the object models of* P *that satisfy the optimistic inheritance constraints.*

8 Conclusion and Future Work

We presented a new and natural model theory for nonmonotonic multiple value and code inheritance and its implementation using a deductive engine that supports well-founded semantics [5]. The problem described in this paper also arises in the design of rule-based object-oriented languages for the Semantic Web. We illustrated the practical nature of the problem as well as the difficulties in finding a proper solution.

Our model-theoretic approach points to several interesting research directions. First, the proposed semantics for inheritance is *source-based*, since in determining multiple inheritance conflict the semantics only considers whether the same method is *defined* at different inheritance sources. A conflict is declared even if the *set* of return values (*i.e.*, *content*) of this method is equivalent at these sources. Intuitively content-based inheritance has a "higher order" flavor and we plan to study its impact on computational efficiency in the future.

Second, it has been argued that inheritance-like phenomena arise in many applications such as discretionary access control and trust management [10], but they cannot be formalized using a single semantics. We are working on extensions to our framework, which allow users to specify their own *ad hoc* inheritance policies in a programmable, yet declarative, way.

References

1. S. Decker, S. Melnik, F. V. Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks. The Semantic Web: The roles of XML and RDF. *IEEE Internet Computing*, 15(3):63–74, October 2000.
2. D. Fensel, S. Decker, M. Erdmann, and R. Studer. Ontobroker: Or how to enable intelligent access to the WWW. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1998.
3. J. Frohn, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing semistructured data with FLORID: A deductive object-oriented perspective. *Information Systems*, 23(8):589–613, 1998.
4. A. V. Gelder. The alternating fixpoint of logic programs with negation. In *ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.

5. A. V. Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, July 1991.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
7. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programms: Combining logic programs with description logic. In *International World Wide Web Conference*, 2003.
8. I. Horrocks. DAML+OIL: A description logic for the Semantic Web. *IEEE Bulletin of the Technical Committee on Data Engineering*, 25(1), March 2002.
9. I. Horrocks and S. Tessaris. Querying the semantic web: A formal approach. In *International Semantic Web Conference*, 2002.
10. S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, June 2001.
11. H. M. Jamil. Implementing abstract objects with inheritance in Datalog^{neg}. In *International Conference on Very Large Data Bases*, pages 56–65, 1997.
12. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.
13. A. Magkanaraki, S. Alexaki, V. Christophides, and D. Plexousakis. Benchmarking RDF schemas for the semantic web. In *International Semantic Web Conference*, 2002.
14. W. May. A rule-based querying and updating language for XML. In *International Workshop on Database Programming Languages (DBPL)*, pages 165–181, 2001.
15. W. May and P. Kandzia. Nonmonotonic inheritance in object-oriented deductive database languages. *Journal of Logic and Computation*, 11(4), 2001.
16. W. May, B. Ludäscher, and G. Lausen. Well-founded semantics for deductive object-oriented database languages. In *International Conference on Deductive and Object-Oriented Databases*, pages 320–336. Springer Verlag LNCS, 1997.
17. T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM Symposium on Principles of Database Systems*, pages 11–21, 1989.
18. F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, March 1991.
19. The rule markup initiative. http://www.dfki.uni-kl.de/ruleml/.
20. M. Sintek and S. Decker. TRIPLE – a query, inference, and transformation language for the semantic web. In *International Semantic Web Conference*, 2002.
21. M. Vardi. The complexity of relational query languages. In *ACM Symposium on Theory of Computing*, pages 137–145, 1982.
22. G. Yang. *A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases*. PhD thesis, SUNY at Stony Brook, December 2002. http://www.cs.sunysb.edu/~guizyang/.
23. G. Yang and M. Kifer. Implementing an efficient DOOD system using a tabling logic engine. In *First International Conference on Computational Logic, DOOD'2000 Stream*, July 2000.
24. G. Yang and M. Kifer. $\mathcal{F}$LORA-2: User's Manual. http://flora.sourceforge.net/, June 2002.
25. G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *International Conference on Ontologies, DataBases, and Applications of Semantics*, October 2002.

Rules and Defeasible Reasoning on the Semantic Web

Grigoris Antoniou[1] and Gerd Wagner[2]

[1] Institute of Computer Science, FORTH, Greece
ga@csd.uoc.gr
[2] Faculty of Technology Management
Eindhoven University of Technology, The Netherlands
G.Wagner@tm.tue.nl

Abstract. This paper discusses some issues related to the use of rules for the Semantic Web. We argue that rule formalisms and rule-based technologies have to offer a lot for the Semantic Web. In particular, they allow a simple treatment of defeasible reasoning, which is essential for being able to capture many forms of commonsense policies and specifications.

1 Introduction

The Semantic Web initiative [23] aims at advancing the Web by enriching its content with propositional information that can be processed by inference-enabled Web applications. It is expected that the deployment of Semantic Web technologies will have significant impact on the way information is exchanged and business is conducted.

The development of the Semantic Web proceeds in layers, each layer being built on top of lower layers. The first "real" Semantic Web layer (above XML), is the meta-data language RDF. At present, the highest layer that has reached W3C standardization is the *web ontology language* OWL [7] based on the DAML+OIL language proposal [6]. These languages were designed to be sufficiently rich to be useful in applications, while being simple enough to allow for efficient inference procedures. Since they correspond to certain extensions of well-known *description logics* [5, 14, 13], they come with a clear semantics and with efficient inference algorithms.

One of the next steps in the development of the Semantic Web core will be the realization of more advanced *reasoning* capabilities for web applications, preferably built on top of RDF and OWL. A key to this will be *rules*. *Rule technologies* are by now well-established and no longer restricted to AI systems. The logic corresponding to the (limited) rule concept of a definite Horn clause is called *definite Horn logic*. Description logics and definite Horn logic are not reducible to each other: while both of them are subsystems of first order logic, each of them can express something the other one cannot express (see Section 2.2).

It seems natural to add rules "on top" of web ontologies. However putting rules and description logics together poses many problems, and may be an overkill, both computationally and linguistically. Another possibility is to start with RDF/RDFS, and extend it by adding rules. Reasons for the use of rules in this setting include the following: (a) there exist powerful implementations of rule engines; (b) rule-based technology has been in use for many years and is well understood by practitioners; (c) many phenomena can be modeled naturally with the help of rules.

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 111–120, 2003.

In addition to Horn rules, we also discuss *defeasible rules*, which support

- certain forms of *inconsistency tolerance* expected to be useful in Web applications of the future;
- *default inheritance* which may be useful for Web ontologies.

While most nonmonotonic logics have high computational complexity [18], we will focus on defeasible reasoning systems [24, 20, 3, 2, 9], which have low complexity [17, 9]. Their use in various application domains has been advocated, including the modeling of regulations and business rules [19, 9, 1], modeling of contracts [9], legal reasoning [21] and agent negotiations [8].

2 Monotonic Rules for the Semantic Web

2.1 Basics of Horn Logic Programming

A *Horn rule* has the form

$$B_1, \ldots, B_n \rightarrow A$$

where $A, B_1, \ldots, B_n$ are atomic formulas. A is the *head* of the rule, and $B_1, \ldots, B_n$ are the *premises* of the rule. The set $\{B_1, \ldots, B_n\}$ is referred to as the *body* of the rule.

A *fact* is an atomic sentence, such as $loyalCustomer(345678)$; it says that the customer with ID 345678 is loyal. A *definite logic program* P is a finite set of facts and Horn rules.

A Horn rule $B_1, \ldots, B_n \rightarrow A$ corresponds to the following formula of first order predicate logic:

$$\forall X_1 \ldots \forall X_k((B_1 \wedge \ldots \wedge B_n) \supset A)$$

where $X_1, \ldots, X_k$ are all variables occurring in $A, B_1, \ldots, B_n$. A fact A corresponds to $\forall(A)$, i.e. the universal closure of A. Thus, a definite logic program corresponds to a set of definite Horn clauses. Consequently, its semantics is given by its least Herbrand model and the SLD-resolution-based proof theory).

2.2 Rules versus Description Logics

Definite Horn logic and description logics are not reducible to each other. The following examples, taken from [11], illustrate this property.

Horn rules lack the possibility to express existential quantification. For example, they do not allow to state that every person has a father, which can be easily done in description logics:

$$Person \sqsubseteq \exists father.T$$

Also, negation in the head is not allowed in Horn rules. Therefore it is impossible to state that a person is either a male or a female, while it can be easily expressed in description logics:

$Person \sqsubseteq Man \sqcup Woman$
$Man \sqsubseteq \neg Woman$

On the other hand, in description logics one cannot state that individuals who work and live at the same location are home workers, while it can be easily stated with a Horn rule:

$$homeWorker(X) \leftarrow works(X, Y), lives(X, Z), loc(Y, L), loc(Z, L)$$

More generally, description logics are unable to express chains of joins across different predicates.

In a more elaborated comparison of rules and description logics, which we cannot undertake here, one should also include

- normal logic programs (rules with negation-as-failure)
- disjunctive rules
- extended logic programs (rules with strong negation and negation-as-failure)

2.3 Logic Programs as the Basis of Ontology Languages

We propose to use rules as a uniform formal basis for the ontology, logic and proof layers. The departing point for this approach is, of course, RDF(S). In this section we show how the basic constructs of RDF(S) can be modeled using rules. We proceed to show that rules are capable of representing many of the features of OWL. The material of this section is based on [11].

RDF(S) Statements.

$Statement(a, P, b)$:	$P(a, b)$
$type(a, C)$:	$C(a)$
$C\ subClassOf\ D$:	$C(X) \rightarrow D(X)$
$P\ subPropertyOf\ Q$:	$P(X, Y) \rightarrow Q(X, Y)$
$domain(P, C)$:	$P(X, Y) \rightarrow C(X)$
$range(P, C)$:	$P(X, Y) \rightarrow C(Y)$

OWL Statements.

$C\ sameClassAs\ D$:	$C(X) \rightarrow D(X)$
	$D(X) \rightarrow C(X)$
$P\ samePropertyAs\ Q$:	$P(X, Y) \rightarrow Q(X, Y)$
	$Q(X, Y) \rightarrow P(X, Y)$
$transitiveProperty(P)$:	$P(X, Y), P(Y, Z) \rightarrow P(X, Z)$
$inverseProperty(P, Q)$:	$P(X, Y) \rightarrow Q(Y, X)$
	$Q(Y, X) \rightarrow P(X, Y)$
$functionalProperty(P)$:	$P(X, Y), P(X, Z) \rightarrow Y = Z$

Mapping OWL Constructors.

Union
When a union constructor occurs on the left hand side of a subclass axiom, then it can simulated by rules as follows:

$(C_1\ Union\ C_2)\ subClassOf\ D$: $C_1(X) \rightarrow D(X)$, $C_2(X) \rightarrow D(X)$

But a union on the right hand side of a subclass axiom cannot be simulated (because we would need disjunction in the head of rules).

Intersection

$(C_1 Intersection\ C_2)\ subClassOf\ D$: $C_1(X), C_2(X) \rightarrow D(X)$
$D\ subClassOf\ (C_1 Intersection\ C_2)$: $D(X) \rightarrow C_1(X)$ $D(X) \rightarrow C_2(X)$

Universal restriction

$C\ subClassOf\ \forall P.D$: $P(X,Y), C(X) \rightarrow D(Y)$
$\forall P.D\ subClassOf\ C$: Simulation not possible.

Existential restriction

$C\ subClassOf \exists\ P.D$: Simulation not possible.
$\exists P.D\ subClassOf\ C$: $P(X,Y), C(Y) \rightarrow D(X)$

Negation
Simulation is not possible, in general.

Cardinality restrictions
To express cardinality restrictions one needs rules with equality, with the drawback of decreased efficiency.

3 Defeasible Rules for the Semantic Web

3.1 Nonmonotonic Reasoning and the Semantic Web: Why

We illustrate the use of nonmonotonic reasoning for the Semantic Web by presenting three ideas/application scenarios. The first two illustrate applications *on top of ontologies*. The third one concerns the use of defeasible rules *within ontology languages*.

Modeling Incomplete Information: An Example. We believe that defeasible rules will play an important role in the areas of (Semantic-Web-enabled) Electronic Commerce and Knowledge Management. In these areas situations arise naturally where the available information is incomplete. Let us briefly illustrate a scenario relevant to the Semantic Web.

Suppose that I have had so positive experiences with my Semantic Web personal agent that I trust it fully. One morning I wake up thinking of my girlfriend who is in Greece, while I am in Germany, and decide to send her flowers because it is her birthday. I ask the agent to do the job and leave home, being unavailable for further contact. The agent sets out to locate several relevant service companies and to compare their price, reputation, delivery policies etc.

Now suppose a company has a policy that it will grant a special 5% discount if the recipient happens to have birthday on that day. Further suppose the company is wise enough to represent its pricing policy in a declarative way. Now how could it represent the discounting rule? It cannot be sure to receive the information about birthday (unaware of the discounting possibility I failed to tell my personal agent; nor can I be contacted). This is a typical situation where reasoning must be made in the presence of *incomplete information*. Obviously the pricing policy needs something like the following:

R_1: If a birthday is provided and corresponds with the current date then give a 5% discount.
R_2: If the birthday is not provided then use the standard price.

The solution using defeasible rules is simple:

R_1: If a birthday is provided and corresponds with the current date then give a 5% discount.
R_2: Usually use the standard price.
$R_1 > R_2$

Here the priority $R_1 > R_2$ decides the conflict that arises in case both rules are applicable.

Potential Applications of Defeasible Rules.

- *Modelling Business Rules and Policies*: For example pricing policies, refund policies, security policies, privacy policies, delivery options etc.
- *Personalization*: e.g. recommender systems.
- *Brokering*: declarative descriptions of capabilities/products/services, declarative descriptions of user needs/requirements/preferences, matching between the two.
- *Bargaining*: declarative descriptions of capabilities/products/services, declarative descriptions of user needs/requirements/preferences, declarative negotiation strategies, formal negotiation.

Default Inheritance. Default inheritance is a feature supported by many object-oriented systems, both for knowledge representation and for OO programming. It allows a property to be inherited from superclasses, but also to override the property by more specific information in the subclass.

For example, we might wish to say that birds typically fly, but penguins, which form a subclass of birds, don't fly. This information can be easily expressed in OWL.

Unfortunately we would get a logical inconsistency if we were to declare an instance of the class penguin.

This observation is not surprising since OWL, as all description logics, are basically a subset of predicate logic and, as such, monotonic. Clearly ontology languages must be extended by a nonmonotonic component if they are to implement default inheritance. Such an extension of the description logic underlying OWL by adding a preference order on axioms and defining a preferential entailment relation is proposed in [12]. We define such a mechanism in the framework of rules in the following subsections.

3.2 Defeasible Reasoning: Ideas

The approach we present uses two kinds of knowledge:

- Strict (monotonic) knowledge: monotonic rules (Horn logic) will be used.
- Defeasible (nonmonotonic) knowledge: defeasible rules will be used, which may be overridden by other rules.

The key ideas of the approach are listed below:

- The approach is skeptical: If rules with contradicting heads can fire, none of them is applied, to avoid conflicting conclusions. This feature supports inconsistency tolerance.
- Conflict resolution: A priority relation may be used to resolve some conflicts among rules. Interestingly, priorities can be compiled into the rule system [3].
- The approach we propose is efficient. In the best case the computational complexity is linear [17].
- Full integration between strict and defeasible rules is achieved (which makes the approach different from previous work [10, 4]). Thus, defeasible rules can be used in rule-based ontology languages (for default inheritance), or in application layers on top of ontological knowledge.

3.3 Defeasible Reasoning: The Language

A *defeasible rule* has the form

$$L_1, \ldots, L_n \Rightarrow L$$

such that all L and L_i are literals $p(a_1, \ldots, a_m)$ or $\neg p(a_1, \ldots, a_m)$, with constants $a_1, \ldots, a_m$ and a predicate p. $\{L_1, \ldots, L_n\}$ is the set of *antecedents* of the rule r, denoted $A(r)$. And L is called the *head* (or *consequent*) of r, denoted $C(r)$.

Defeasible rules with variables are interpreted as schemas: they represent the set of their ground instances. This interpretation is standard in many nonmonotonic reasoning approaches, among others in default logic [22] and defeasible logics [20].

A *superiority relation* on R is an acyclic relation $>$ on R (that is, the transitive closure of $>$ is irreflexive). When $r_1 > r_2$, then r_1 is called *superior* to r_2, and r_2

inferior to r_1. This expresses that r_1 may override r_2. For example, given the defeasible rules

$$r: \quad bird(X) \Rightarrow flies(X)$$
$$r': brokenWing(X) \Rightarrow \neg flies(X)$$

which contradict one another, no conclusive decision can be made about whether a bird with a broken wing can fly. But if we introduce a superiority relation $>$ with $r' > r$, then we can indeed conclude that it cannot fly.

A *knowledge base* is a pair $(R, >)$, where R is a finite set of strict (monotonic) and defeasible rules, and $>$ an acyclic relation on R.

3.4 The Defeasible Logic Meta-program

In this section we introduce a meta-program $\mathcal{M}$ in a logic programming form that expresses the essence of the defeasible reasoning; [16] showed that it is equivalent to the standard defeasible logic presentation [3].

$\mathcal{M}$ consists of the following clauses. We first introduce the predicates defining classes of rules, namely

$\texttt{supportive\_rule}(Name, Head, Body)$:-
  $\texttt{strict}(Name, Head, Body)$.

$\texttt{supportive\_rule}(Name, Head, Body)$:-
  $\texttt{defeasible}(Name, Head, Body)$.

Now we present clauses which define provability of literals. Initially we distinguish between two levels of proof: definite provability which uses only the strict rules, and defeasible provability.

$c1$ $\texttt{definitely}(X)$:-
  $\texttt{strict}(R, X, [Y_1, \ldots, Y_n])$,
  $\texttt{definitely}(Y_1)$,...,$\texttt{definitely}(Y_n)$.

Now we turn to defeasible provability. If a literal X is definitely provable it is also defeasibly provable.

$c2$ $\texttt{defeasibly}(X)$:-
  $\texttt{definitely}(X)$.

Otherwise the negation of X must not be strictly provable, and we need a rule R with head X which fires (that is, its antecedents are defeasibly provable) and is not overruled.

$c3$ $\texttt{defeasibly}(X)$:-
  not $\texttt{definitely}(\neg X)$,
  $\texttt{supportive\_rule}(R, X, [Y_1, \ldots, Y_n])$,
  $\texttt{defeasibly}(Y_1)$,...,$\texttt{defeasibly}(Y_n)$,
  not $\texttt{overruled}(R, X)$.

A rule R with head X is overruled if there is a rule S with head $\neg X$ which fires and is not defeated.

$c4$ `overruled`(R, X):-
`supportive_rule`$(S, \neg X, [U_1, \ldots, U_n])$,
`defeasibly`(U_1),...,`defeasibly`(U_n),
`not defeated`$(S, \neg X)$.

And a rule S with head $\neg X$ is defeated if there is a rule T with head X which fires and is superior to S.

$c5$ `defeated`$(S, \neg X)$:-
`sup`(T, S),
`supportive_rule`$(T, X, [V_1, \ldots, V_n])$,
`defeasibly`(V_1),...,`defeasibly`(V_n).

Given a defeasible theory $D = (R, >)$, the corresponding program $\mathcal{D}$ is obtained from $\mathcal{M}$ by adding facts according to the following guidelines:

1. `strict`$(r_i, p, [q_1, \ldots, q_n])$. for each rule $r_i : q_1, \ldots, q_n \rightarrow p \in R$
2. `defeasible`$(r_i, p, [q_1, \ldots, q_n])$. for each rule $r_i : q_1, \ldots, q_n \Rightarrow p \in R$
3. `sup`(r_i, r_j). for each pair of rules such that $r_i > r_j$

We still need to say which logic programming semantics to use for the negation operator. In [16] it was shown that $\mathcal{D}$, under the Kunen semantics [15], is equivalent to the defeasible logic of [3].

The following consistency preservation result can be shown.

Theorem 1. *Let K be a knowledge base and L a literal. If L and $\neg L$ are both defeasibly provable, then they are both strictly provable. That is, an inconsistency in the set of defeasible conclusions is only possible if the certain part of K is inconsistent.*

3.5 Rules with Two Kinds of Negation

Unlike normal and extended logic programs, the defeasible knowledge bases we have considered do not use negation-as-failure in their object language (of course, negation-as-failure lies at the heart of the proof theory). If desired, negation-as-failure can be added to our rule language without requiring a new proof theory, since it can be eliminated by transforming a rule with both negations into a set of rules with a suitable superiority relation. Consider the rule with negation-as-failure:

$$A_1, \ldots, A_n, not\ B_1, \ldots, not\ B_m \Rightarrow C.$$

Its meaning is captured by the following program (fragment):

$r :\ A_1, \ldots, A_n, p \Rightarrow C$
$r_0 : \Rightarrow p$
$r_1 :\ B_1 \Rightarrow \neg p$
$\ldots$
$r_m :\ B_m \Rightarrow \neg p$
$r_1 > r_0, \ldots, r_m > r_0$

where p is a new atom.

4 Conclusion

This paper proposes to use rules as a fundamental concept of the Semantic Web. Rules are well-understood, efficient, supported by mature tools, and well-known among practitioners. Also we discussed the integration of monotonic and nonmonotonic rules.

Future work includes, among others, further studies of the interplay between strict and defeasible rules. In this paper we have concentrated on one particular idea, where strict rules are considered as defeasible if their antecedents are only defeasibly, but not strictly, provable. This approach is computationally simple, but alternatives do exist and need further exploration.

Further future work includes theoretical work on the combination of description logics and logic programming, the development of rule-based tools for the Semantic Web, and evaluation of the rule-based approach from various application areas.

References

1. G. Antoniou, D. Billington and M.J. Maher. On the analysis of regulations using defeasible rules. In *Proc. 32nd Hawaii International Conference on Systems Sciences*, 1999.
2. G. Antoniou, D. Billington, G. Governatori and M.J. Maher. A flexible framework for defeasible logics. In *Proc. 17th American National Conference on Artificial Intelligence (AAAI-2000)*, 405–410.
3. G. Antoniou, D. Billington, G. Governatori and M.J. Maher. Representation Results for Defeasible Logic. *ACM Transactions on Computational Logic* 2,2 (2001): 255–287.
4. G. Antoniou. Nonmonotonic Rule Systems on top of Ontology Layers. In *Proc. 1st International Semantic Web Conference*, LNCS 2342, Springer 2002, 394–398.
5. M. Buchheit, F. Donini and A. Schaerf. Decidable Reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research* 1 (1993): 109138.
6. D. Connolly et al. *DAML+OIL (March 2001) Reference Description*. http://www.w3.org/TR/daml+oil-reference.
7. M. Dean et al. *OWL Web Ontology Language Reference 1.0*. http://www.w3.org/TR/owl-ref/
8. M. Dumas, G. Governatori, A. ter Hofstede, and P. Oaks. A formal approach to negotiating agents development. *Electronic Commerce Research and Applications*,1,2 (2002).
9. B. Grosof, Y. Labrou and H. Chan. A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In *Proc. 1st ACM Conference on Electronic Commerce*, ACM 1999.
10. B. Grosof and T. Poon. Representing Agent Contracts with Exceptions using XML Rules, Ontologies, and Process. In *Proc. Intern. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, held 14 June 2002, Sardinia (Italy) in conjunction with the First International Semantic Web Conference.
11. B. Grosof and I. Horrocks. *Description Logic Programs: Combining Logic Programs with Description Logic*. Unpublished manuscript.
12. S. Heymanns and D. Vermeir. A Defeasible Ontology Language. In R. Meersman and Z. Tari (Eds.), *CoopIS/DOA/ODBASE 2002*, LNCS 2519, 1033–1046.
13. I. Horrocks and U. Sattler. Ontology reasoning in the SHOQ(D) description logic. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI-01)*, Morgan Kaufmann 2001, 199–204.
14. A. Levy and M-C. Rousset. CARIN: A Representation Language Combining Horn rules and Description Logics. *Artificial Intelligence* 104(1-2), 1998, 165 209.

15. K. Kunen. Negation in Logic Programming. *Journal of Logic Programming* 4,4 (1987): 289–308.
16. M.J. Maher and G. Governatori. A Semantic Decomposition of Defeasible Logics. *Proc. American National Conference on Artificial Intelligence (AAAI-99),* 299–306.
17. M.J. Maher. Propositional Defeasible Logic has Linear Complexity. *Theory and Practice of Logic Programming*, 1,6 (2001): 691–711.
18. V. Marek and M. Truszczynski. *Nonmonotonic Reasoning – Context-Dependent Reasoning*. Springer 1993.
19. L. Morgenstern. Inheritance Comes of Age: Applying Nonmonotonic Techniques to Problems in Industry. *Artificial Intelligence*, 103(1998): 1–34.
20. D. Nute. Defeasible Logic. In D.M. Gabbay, C.J. Hogger and J.A. Robinson (eds.): *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 3*, Oxford University Press 1994, 353–395.
21. H. Prakken. *Logical Tools for Modelling Legal Argument: A Study of Defeasible Reasoning in Law.* Kluwer Academic Publishers 1997.
22. R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence* 13(1980): 81–132.
23. www.w3.org/2001/sw/
24. G. Wagner. Ex contradictione nihil sequitur. In Proceedings of International Joint Conference on Artificial Intelligence IJCAI'91, Morgan Kaufmann, 1991.

Inference Queues for Communicating and Monitoring Declarative Information between Web Services

Bruce Spencer and Sandy Liu

Institute for Information Technology – e-Business
National Research Council of Canada
46 Dineen Drive, Fredericton, New Brunswick, Canada E3B 9W4
Faculty of Computer Science, University of New Brunswick
P.O Box 4400, Fredericton, New Brunswick, Canada E3B 5A6
{Bruce.Spencer, Sandy.Liu}@nrc.gc.ca
http://iit-iti.nrc-cnrc.gc.ca/groups/il_e.trx

Abstract. We introduce the inference queue as a mechanism for communicating and transforming data in Web services choreography. The insert operation provides definite clauses to the inference queue, and the remove operation generates output that is sound, complete, fair, and irredundant. Both operations are thread safe and responsive. The inference queue can form part of a highly configurable data transformation system. Rules can also monitor for events of interest based on the occurrence of certain conditions. Our suggestion of system wide monitoring of communication is complementary to existing Web services proposals.

1 Introduction

Communication from one web service to another is the most basic task of two important recent research efforts in e-business: Web Service Choreography[16] and Semantic Web Services[15]. This information should be meaningful to both parties; it is common in B2B e-business today that both parties will have a common format, perhaps a variant of XML, and will have a common understanding of the intended meaning of each portion of the message, of each XML tag. But for the future we want to consider doing business with automatically discovered Web services, not previously known. Thus the information should be declarative in nature, with meanings assigned from a standard ontology in the general subject area, as suggested by the Semantic Web research effort[14].

RuleML[2] provides the basic syntactic structures from first order logic so that the relations and functions/complex structures are apparent to both parties. RuleML is also used for expressing data transformation rules, that convert syntactic conventions used by one party to syntax understandable by the other party. Such transformations will preserve meaning because the inferences are sound. Beyond transformations, rules are suited to express preconditions that declarative information must meet before its communication can be accepted

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 121–135, 2003.

by the intended recipient. Rule engines that mediate the communication serve as monitors of communication within the overall system. They can report any exceptional conditions to high level control systems, or intelligently transform the message so that it becomes acceptable. Thus a rule engine can serve as both a data transformation system and a monitor.

In this paper we introduce the inference queue as a mechanism for communicating and transforming data. We assume that we are reasoning about XML messages sent from one web service to another. It is convenient to use RuleML specifically for these messages as a single RuleML fact can contain the entire message. The inference queue has several characteristics required in the situation we have described so far: it has asynchronous insert, remove and isEmpty methods; it buffers information, allowing the producing process on the insert side to occasionally exceed the capacity of the consuming process on the delete side, so that the producer wastes less time waiting; it is an inference engine that is responsive and fair in the way it generates the complete set of sound conclusions. The inference system also eliminates reporting any results that are duplicates or specializations of previous results.

The inference queue can form part of a highly configurable monitoring system; rules can conclude that some exceptionally interesting event is taking place based on the occurrence of certain conditions. As the evidence of that event passes through the queue, the rule fires. The inference queue possibly has several output ports so these can be configured to transmit reactions to the interesting event. The inference queue also has possibly several input ports, so that data from mixed sources can be combined. A rule may be added with multiple conditions, and facts for these conditions are expected to arrive at distinct input ports. The conclusion of this rule would be produced for the output port(s) when all of its conditions are inserted at the input ports. Thus events arising in separate locations can be monitored, arbitrary conditions on their combination can be expressed and reactions taken when those conditions arise. This allows us to trace information relevant to one business process through a network of Web services, where the links in the network are inference queues.

The rest of the paper is organized as follows: We cover the necessary background on Web services and Web service choreography frameworks. We then introduce the inference queue for transporting data from one web service to another, we define the six properties they need for this task, and establish that they are met by our architecture, which is described in some detail. To make the discussion more concrete, we give an example of a Web service connected to a client with two one-way inference queues, and illustrate an intelligent monitoring system.

2 Web Services and Choreography Frameworks

Web services allow businesses to describe, publish, and invoke self-contained modular software components over the Internet and therefore make distributed computing available Internet-wide. However, a single Web service is often not

sufficient to accomplish a business process that involves multiple parties and/or multiple activities. There is a need to compose a series of Web services together to achieve a new or more useful service. For instance, in order to plan a trip, a user may require the cooperation from a set of services including a flight-scheduling service, a hotel booking service, a car-rental service, and a credit card payment service. This set of services can be considered as a business process. The Business Process Management (BPM) community has long been looking for solutions to standardize the description, modeling, deploying, and monitoring of business processes that contain services provided by heterogeneous vendors. There are several specifications that can serve as candidates to streamline this process including BPML[1], BPEL4WS[7], ebXML BPSS[9], WSCI[6], and DAML-S[4].

3 Inference Queues

An inference queue is a priority queue data structure into which we enqueue (insert) input fomulas and from which we dequeue (remove) inferred output formulas. Syntactic restrictions on both formulas are often imposed to simplify the reasoning task. In this paper we restrict the input formulas to definite clauses and the output to single literal positive clauses (facts) implied by those definite clauses.

Inference queues have standard queue operations (insert, remove and isEmpty) but adapt the standard operational semantics. In response to insert requests, facts and rules are inserted at one end, the *upstream end*; and in response to a remove request, these facts and any implied facts are removed from the other *downstream* end. Unlike normal queues, the size of the output of an inference queue is not necessarily equal to the size of the input.

We define the inference queue also to have four important properties relating to logic and deduction: soundness, completeness, irredundancy and fairness, and to have two properties relating to its simultaneous usage by separate threads: thread safe (assessible by separately running thread or processes) and responsive (no infinite computations are allowed between requests). In this section we describe these six properties of the inference queue in detail, tell why these properties are relevant to communication among Web services, and explain how these properties are guaranteed by our implementation.

3.1 Definitions of the Properties We Seek

Suppose definite clauses $C_1, \ldots, C_n$ have been inserted into the queue, where $\{C_1, \ldots C_n\} \models F_1, \ldots, F_m$, for facts $F_1, \ldots, F_m$. If the queue is capable of delivering, in response to m remove requests, all of these facts (or a covering set of more general facts), then it is **complete**. If it is capable of delivering only entailed facts, it is **sound**.

Suppose that the facts are removed from the queue in the order $F_1, \ldots, F_m$. For every $i \geq 0$ and every $j > 0$ fact F_{i+j} is delivered after fact F_i, and it is not the case for any i that $F_i \models F_{i+j}$. In other words, no fact is more specific than a previous fact. Then the output queue is **irredundant**.

By **fair** we mean that no given conclusion will be infinitely deferred from being produced for a dequeue. Every conclusion will eventually be found. We use fairness to express priority among facts. Exceptional conditions can be given priority, so they are guaranteed to be reported first.

We expect the inference queue to be used by several concurrent programs, or threads, as opposed to a single program thread. A program that can interact with other concurrent programs is **thread safe.** This means two threads cannot interfere with each other's tasks, and usually is done by defining a critical section in the code where only one thread at a time is permitted to run. It also means we have the option to force threads requesting data to wait. In our case a remove request is defined with blocking semantics. This means the thread invoking the remove request will be blocked if there is no element in the queue. The expectation is that another thread will eventually call the insert method, making an element available, and then the removing thread can be notified (re-activated) and the remove request granted. The other requests, to insert a clause into the queue and to ask if the queue is empty, are non-blocking operations.

The queue is also defined to be **responsive** which means that between any pair of requests, no infinite computations are allowed. From a finite set of input clauses, an infinite set of output clauses can arise:

$$\{p(X) \rightarrow p(f(X)), \quad p(a)\} \models p(a), p(f(a)), p(f(f(a))), \ldots.$$

Inference queues address this infinite list by requiring that an infinite sequence of calls to dequeue be made to produce the infinitely many answers. Since the operations are responsive, there will be no infinite operations between these successive calls.

The combined properties of being complete, responsive and fair mean that not only are all conclusions generated (completeness) with no infinite delays (responsiveness), but also that this remains true as new information is being dynamically added or inferred. Thus if a specific fact has been inferred but has not yet been delivered to a remove request, and a sequence of facts is being inferred, such as the infinite computation in the previous paragraph, then eventually that undelivered fact will be delivered, before the infinitely many other facts are all delivered. To accomplish this we depend on the existence of a partial order $\preceq$ (preceeds) between facts. Suppose the inference queue delivers the facts $F_1, \ldots, F_m$ in this order. Then this order must be consistent with the given partial order. In other words $F_{i+j} \not\preceq F_i$.

Beyond the fixed set of clauses, if new facts are being inserted or inferred between remove operations, the partial order is still observed. For example, suppose the fact F_i has been recently removed and F_{i+1} is next in line to go, and a new fact G is inserted or inferred before F_{i+1} is removed, such that $G \preceq F_{i+1}$. Then G will be removed before F_{i+1}. It may also be the case that $G \preceq F_i$, but clearly G cannot be removed before F_i because G was not available at the time F_i was removed. While this appears to disrupt the precedence ordering of the output facts, it is not contrary to the definition. Among the facts available when the remove request is made, a smallest fact is always removed.

Given $Clause_1 C_1, \ldots C_m \rightarrow A$ and $Clause_2 A_1, \ldots, A_n \rightarrow B$ such that there exists i and a substitution θ that unifies A with A_i,
Produce $(A_1, \ldots, A_{i-1}, C_1, \ldots, C_m, A_{i+1}, \ldots, A_n \rightarrow B)\theta$

Note that Robinson's resolution[12] would also allow multiple A_j to be resolved at once, but this is not necessary for completeness of resolution with definite clauses. Here we restrict the application so that $m0$, to enforce unit clause resolutions, and so that $i1$ to select goals in textual order.

Fig. 1. Robinson's resolution applied to definite clauses

We must also assume that the partial order has no infinite strata. In other words, for any given element there is a finite number of elements that are greater than it. This property of no infinite strata is easy to guarantee. For instance if the $\preceq$ relation is based on the number of symbols in the atom, i.e. its length, and there is a finite set of symbols to draw from, then there will be only a finite number of atoms shorter than any given atom.

3.2 A Theorem Prover with the Logic Properties We Seek

The above six properties, as far as the authors know, have not been combined into one theorem prover before, but we have found it is entirely feasible to do so. The ideas are these: (1) allow the theorem prover to incrementally accept new clauses, (2) turn the basic theorem proving loop from an autonomous producer of formula into a demand-driven, or lazy, producer of one fact at a time, and (3) after either operation (a new clause is inserted or a new fact is selected), run part of the theorem prover's machinery to infer all the conclusions that are needed to restore the system to a state ready to service the next request.

The two essential features of any theorem prover are its rule(s) of inference and its strategy for applying those rules, examples of which are shown in Figures 1 and 2, respectively. For the inference queue, we use Robinson's resolution applied to definite clauses. Since this is the only rule of inference and since it is well-known to be sound, **soundness** of the inference queue follows immediately.

Negative literals are selected in clauses in the order in which they appear, which is sufficient since it is well known that resolution's literal selection within a clause is don't-care nondeterministic. In other words, if a proof exists with a specific condition selected first, then the same proof exists with any of the conditions selected first. Thus the general rule of Figure 1 is specialized to select the first condition.

The application strategy is related to those found in theorem provers such as Otter [10], in that the most important strategies and design decisions are represented: The *set of support* restriction [8] is reflected in the fact that the positive clauses are resolved against mixed ones, so the facts form the set of support. Forward subsumption is applied when the a new fact is selected, and it is applied lazily on selection rather then eagerly as facts are generated. This decision requires storing potentially many more new facts, but far fewer subsumption

Three data structures are defined: *NewFacts, OldFacts* and *Rules*.

NewFacts: This is a priority queue for storing single literal definite clauses (facts), ordered by $\preceq$, for that have not yet been processed. Initially it is populated with the facts from the input clases.
OldFacts: This is a list for storing single literal definite clauses that have already been processed.
Rules: This is a list for storing definite clauses with at least one negative literal (condition). These are indexed by one of these negative literals, called the *selected goal*. Assume that the first negative literal is the index. Initially it is populated with the rules from the input clauses.

```
main loop
   select and remove a new fact f_new from NewFacts
   while f_new is subsumed by some member of OldFacts
      select and remove another f_new from NewFacts
   end while
   for each rule r whose first condition unifies with f_new
      resolve r against f_new producing r_1
      process(r_1)
   end for each
   add f_new to OldFacts
end main loop

process(c)
   if c is a rule
      for each old fact f_old unifying with the selected goal of c
         resolve c with f_old to produce new result n
         process(n)
         add c to Rules
      end for each
   else
      add c to NewFacts
   end if
end process
```

Fig. 2. An application strategy for definite clause reasoning

tests are attempted. Backward subsumption is not applied for similar reasons; the list of old facts would potentially be less redundant if it were applied, in that old facts could not be more specific than new ones, but a far greater number of subsumptions tests would be attempted. Furthermore, forward subsumption guarantees that the list of old facts is absolutely irredundant when the facts are ground. We expect that the inference queue will often be in this situation. The stored facts are separated into two lists: *OldFacts* and *NewFacts* so that we can ensure unification attempts are never tried more than once for any given pair of literal occurrences. The reason is this: For every potentially unifiable pair consisting of a fact and a condition that occurs as the selected condition (the first

condition) in a rule, either the fact is selected from *NewFacts* before the rule is created, or the rule is produced before the fact is selected. There are two cases to consider: Suppose the fact is selected before the rule is created. Since the rule does not yet exist, the fact will be put into **OldFacts** without the resolution being performed. Later when the rule is produced, **process** is called and the resolution against the old fact will be performed. Alternately, suppose the rule is created before the fact is selected. Later, when the fact is selected, it is resolved in the main loop against all existing rules including the one of interest. Then it is put into the old facts and is never resolved against any existing rules, only against new rules. Thus, there is exactly one point in time when any given fact is attempted to be resolved against any given rule condition. No redundant searching is done. This argument also serves to convince the reader of the completeness of the search precedure: all proofs will be built because all possible resolutions are tried. Combined with the notion of completeness of the resolution rule of inference for generating single literal conclusions, we can now conclude that the system is **complete** for generating facts. Note that resolution is not complete in the sense of generating all logical consequences, because it does not generate weaker conclusions like $A \vee B$, which are logical consequences of single literal conclusions like A.

There is one point of caution about the theorem prover in Figure 2. The number of stored clauses, both rules and facts, will grow. Since a ground rule with n conditions in its body may give rise to n stored clauses (with $n-1$, $n-2$, ..., to 0 conditions) there is a quadratic effect on the size of stored clauses. For non-ground clauses the affect, naturally, may be worse.

The set **NewFacts** is a priority queue, ordered on $\preceq$ defined on atomic formulas. Of all of the new facts that can be selected a minimal one is chosen next. When control in the procedure is at the select statement of the main loop, the set **NewFacts** contains all of the facts that have been inferred but not yet reported. It does not necesarily contain all of the facts that are implied by the set of clauses. So it is important to point out that the sequence of selected facts is not necessarily arranged in $\preceq$ order. What is true and important for our purposes, is that no single fact will be left unselected indefinitely long, and that the $\preceq$ order allows us to express heuristically the order in which we would prefer to receive our conclusions – in effect, our order of importance. Thus the system displays **fairness.**

3.3 An Inference Queue with All the Properties We Seek

While the theorem prover in Figure 2 has some of the properties we need, it cannot be used in a multithreaded environment in this form. We need to convert it so that explicit insert and remove operations are defined that are thread safe and responsive. By thread safe, we mean that its internal state will not be disturbed by any set of simultaneous external requests from client processes. By responsive we mean that the system cannot go into an infinite loop in response to requests, even though the clause set may imply an infinite set of atoms.

The main inference queue operations are shown in Figure 3. This program has many features in common with Figure 2. The main difference is the subsystem for background processing. Background processing performs the inferences needed after a new fact is selected, and corresponds to the second half of the code in the **main loop** of Figure 2. Before any other insert and remove operations are done, the status variables are checked to see if this background processing is yet undone. The background processing is separated for two reasons. First the inference queue should react to remove requests as quickly as possible, so it returns a result as soon as one is found, deferring the background processing. Second, since requests are made asynchronously from client threads, there will sometimes be opportunities when there are no requests pending, so the inference queue can do this processing on its own time. Thus there is expected to be a background process, or *daemon*, that monitors the requests and the state of *backgroundProcessingIsComplete*. It does the background processing while the queue is otherwise idle.

To establish that the system is thread safe, one needs to ensure that the mutual exclusion lock is in force anytime the remove operation is running. This lock prevents any other thread from entering code that can affect the internal data structures. Likewise the thread performing the insert operation claims the mutual exclusion lock, preventing another thread from entering either the insert or the remove operation. We use the convention that any procedure that claims ownership of the mutual exclusion lock is declared *synchronized.*

We need the system to be **responsive,** which means there is no chance that any step will take infinitely long. While **process** may appear to contain a potentially infinite loop because of the inner call to itself, each call to it is guaranteed to terminate. Note that for a given parameter in a call to **process**, the argument provided to each inner call is smaller by one literal. Since there will be a finite number of inner calls, limited by the finite number of stored rules, there is no chance for an infinite loop. Similarly background processing will never take infinitely long. Neither insert nor remove has any infinite loops, either.

Although we have not shown it, the operation to tell if the inference queue is empty first ensures that all background processing is done, and then returns true if and only if the *NewFacts* priority queue is empty.

3.4 Variations of the Inference Queue

We allow several input ports to be opened to an inference queue. There is no state information associated with the insert process, so there is no essential difference between having one or many input ports.

It may be of interest for an inference queue client to receive consequences that match a certain pattern only, instead of receiving all consequences. This is easily handled by invoking a unification step as part of the output port's operation; facts that do not meet the pattern are not transferred out of that port.

There may also be several output ports receiving consequences from an inference queue, and the queue services remove requests from each. New output ports can be opened at any time. Suppose that a recently initialized inference

In addition to the three data structures from Figure 2, *NewFacts*, *OldFacts* and *Rules*, there are two non-local variables:

- A boolean state variable *backgroundProcessingIsComplete* recalls whether processing has been done since the most recent **remove**. It is initially true.
- A fact *mostRecentlyRemovedFact* has a valid value when *backgroundProcessingIsComplete* is false.

```
synchronized insert(C)
    if not backgroundProcessingIsComplete
        performBackgroundProcessing(mostRecentlyRemovedFact)
    endif
    process(C)
    notify
end insert

synchronized Fact remove
    if not backgroundProcessingIsComplete
        performBackgroundProcessing(mostRecentlyRemovedFact)
    endif
    while NewFacts is empty
        wait
    end while
    select and remove a new fact f_new from NewFacts
    while f_new is subsumed by some member of OldFacts
        select and remove another f_new from NewFacts
    end while
    set backgroundProcessingIsComplete to false
    set mostRecentlyRemovedFact to f_new
    return f_new
end remove

synchronized performBackgroundProcessing(f_new)
    for each rule r whose first condition unifies with f_new
        resolve r against f_new producing r_1
        process(r_1)
    end for each
    set backgroundProcessingIsComplete to true
    add f_new to OldFacts
end performBackgroundProcessing

process(c)
    if c is a rule
        for each old fact f_old unifying with the selected goal of c
            resolve c with f_old to produce new result n
            process(n)
            add c to Rules
        end for each
    else
        add c to NewFacts
    end if
end process
```

Fig. 3. Inference Queue Operations: insert and remove

queue has delivered several facts to its single output port. Then a second output port is opened. The first several requests from the second port will be serviced by replaying the *OldFacts* list, until it catches up with the first port. If the newer port then makes more requests, the inference machinery will be used to calculate the consequences. Then requests from the original output port will be serviced by replaying these recently inferred facts, which were lately added to *OldFacts*.

This mechanism allows at any time an inference queue to service a request for a full disclosing of what has transpired, and in what order. Thus the *OldFacts* lists doubles as a log file.

We may need a monitoring service whose main interest in an inference queue is not what can be concluded, but what has been sent to other output ports. This need can be met by opening an *observation port*, which is an output port that is not allowed to advance the state of the inference engine. Thus once the remove requests from an observation port are given all of the old facts, further remove requests are blocked and these threads are made to wait. When a dequeue request is made from another (non-observation) port it is serviced with a new fact, and the thread waiting on the observation port is notified and delivered the same fact.

4 Use Cases

The inference queue complements existing Web service choreography proposals, such as WSCI, DAML-S and BPEL4WS. It offers services not currently implemented in those proposals, but mentioned as future work[5](p. 3). We view Web services choreography as the configuration of communicating networks where nodes are instances of Web services; we see the role of the inference queues as the links between these nodes. By using an inference queue for communication among Web services, data transformation and format conversion are possible. In the following sections we offer a small example to illustrate potential advantages of using inference queues for this communication. We further the example to illustrate that an inference queue can provide a monitoring service for tracing the steps of a specific business process in the Web services network. Since the inference queue stores in *OldFacts* all of the messages that have been transmitted, the data is available later for forensic analysis.

4.1 Mediating Communication between Web Services

Suppose a travel agent Web service mediates between a traveller and various agents for flights, events, and hotel reservations for example. This is illustrated in Figure 4.

In this use case, the human client (1) requests a Travel Agent Web service (2) to help him/her purchase a ticket that meets his needs. The Travel Agent service uses Canadian conventions for encoding dates, exclusively. It deals with various Ticket Agent Web services, some of which use different local conventions for encoding data, such as dates. Suppose there is little flexibility in the interfaces of

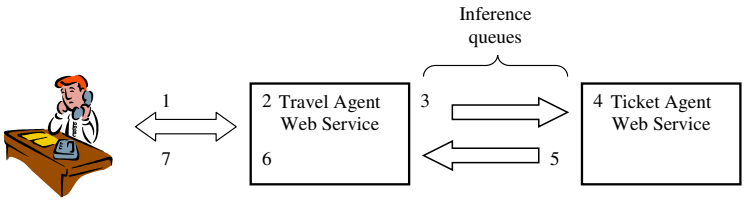

Fig. 4. Web services mediation in two directions

these Ticket Agent services – and they do not support Canadian date encoding, day-month-year. Then the inference queue (3) connecting the Travel Agent to the Ticket Agent can have some rules for date transformation: (Here we use the Prolog convention for displaying rules.)

requestOut(Flight, LocalDate, CustomerNumber) ←
    requestIn(Flight, CanadianDate, CustomerNumber),
    dateConversion(CanadianDate, LocalDate).

When the inference queue receives a request, such as the fact

requestIn('Delta861', '23-06-2003', 'Traveler229'),

following rule is added to *Rules*:

requestOut('Delta861', LocalDate, 'Traveler229') ←
    dateConversion('23-06-2003', LocalDate).

This is eventually used to conclude the fact: requestOut('Delta861', '06-23-2003', 'Traveler229'). Note that the date is now month-day-year. This fact is delivered at the next remove request from the downstream service (4), or to an agent that has the initiative to pass such facts to the input of the Ticket Agent service. Web services may need such an active process since they are usually reactive and not initiators.

When the Ticket agent is ready to issue a purchase order number, which will tell the customer what he needs to know to actually purchase the ticket, it issues the response fact to the return inference queue (5): responseOut('Delta861', 'Traveller229', '06-23-2003', 'PO1242'). That inference queue has the task of insulating the Travel Agent from non-Canadian dates, so it contains the rules.

responseOut(Flight, CanadianDate, CustomerNumber, PurchaseOrderNumber)
    ←
    responseIn(Flight, LocalDate, CustomerNumber, PurchaseOrderNumber),
    dateConversion(CanadianDate, LocalDate).

The fact responseOut('Delta861', 'Traveller229', '23-06-2003', 'PO1242') is eventually issued to the Travel Agent (6) and some appropriate response is given to the Customer (7).

4.2 Monitoring and Exception Handling

Of all of the Web service choreography proposals, BPEL4WS deals most with exceptions; an exception can be thrown by a Web service and caught by a BPEL fault handler. The error handling framework can compensate by attempting to roll back the latest step in the business process. Complementary to this, the inference queue allows faults to be defined and monitored outside of the Web service, so it is possible to have robust processing using Web services that do not incorporate fault handling. In Figure 5 we have extended the example from Figure 4 so that several Web services are being invoked. Each Web service is connected to another Web service by an inference queue, perhaps with a return queue. Each inference queue links to a monitor via an observation output port, represented in the figure by thin lines and described in Section 3.4. In principle there could be a network of Web services linked by inference queues that send information back to a common monitor, which would have access to all communicated facts across the system.

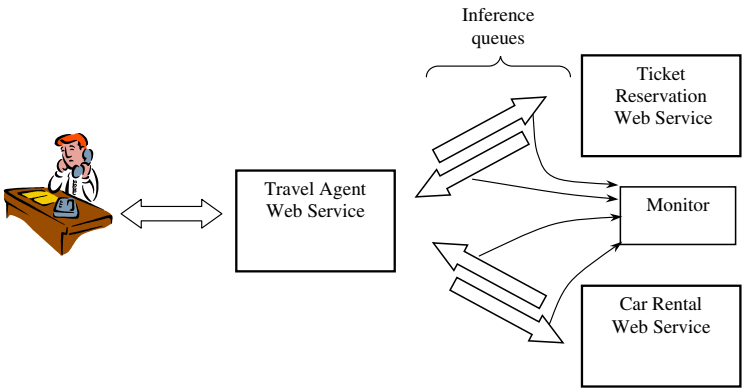

Fig. 5. Web services fault detection and forensic analysis

5 Related Work

The existing Web Service Description Language (WSDL) only defines operations in terms of incoming/outgoing messages and binding details at the syntactic level without outlining the relationship of the messages in a service that encompasses multiple operations. The Web Service Choreography Interface (WSCI) describes how WSDL operations are choreographed and which properties these choreographies expose, such as transaction and correlation. As a result, WSCI complements the shortfalls of WSDL by providing a detailed description of the behaviours in different states for a given Web service described by WSDL. WSCI also describes the collective message exchange among interacting Web services, providing a global, message-oriented view of a process involving multiple Web services.

Note that WSCI only describes a one-sided interface for a Web services. In other words, the message exchange is described from the point of view of one Web service[11]. To compose a set of Web services, we need standards to model the workflow of services. A few emerging standards taking different approaches are attempting to fill in this gap.

The Web Services Modeling Framework (WSMF) propose a comprehensive conceptual model for developing, describing, and composing Web services by appointing the principles of maximal de-coupling and a scalable mediation service. The WSMF defines four main elements: ontologies, goal repositories that define the problems to be solved by Web services, Web services descriptions, and mediators to overcome the interoperability problem.

While WSMF provides a very high-level model for integrating Web services, others are aiming to offer grounded models to realize the vision described by WSMF for Web services orchestration and choreography. Early endeavors in this path include IBM's Web Services Flow Language (WSFL) evolved from Petri Nets and Microsoft's XLANG based on the Pi-Calculus model[3]. These two proposals have since converged into a single specification called BPEL4WS (Business Process Execution Language for Web Services), which unifies the essence of graph-oriented processes from WSFL and structural constructs from XLANG to enable the aggregation of Web services into a process execution model. "As an executable process implementation language, the role of BPEL4WS is to define a new Web service by composing a set of existing services.[17]." Hence, the composition (called the *process* in BPEL4WS) indicates how the service interface fits into the overall execution of the composition.

As BPEL4WS uses WSDL portType information for service description, it inherits the limitation of WSDL, which does not describe side effects, pre- or post-conditions of services, and the expressiveness of service behavior and inputs/outputs are constrained by XML and XML Schema. In contrast, DAML-S, a semantic markup specification for Web services, employs a complementary approach to describe Web services. Indeed, DAML-S is a DAML-OIL ontology for describing Web services. It aims to enable automated Web services discovery, invocation, composition, and execution monitoring by providing sufficient semantic description. A DAML-S document comprises a ServiceProfile, a ServiceModel, and a Service Grounding. The ServiceProfile describes the properties of a service such as input and output types, pre-conditions and post-conditions, and binding patterns to facilitate automatic service discovery where the ServiceModel together with ServiceGrounding provide information for an agent to make use of a service.

One of the important aspects of modeling a business process is to have a mechanism for exception handling, such that when an exception is raised during the course of a business process, the model allow appropriate recovery actions (such as roll-back) to be performed. Web services provide a basis for passing messages between participants in collaboration-based processes. Nevertheless, most of the current proposals do not provide this much-needed monitoring service. DAML-S has the notion of execution monitoring, but has not yet been defined

in the current release; BPEL4WS claims to have exceptions (faults) built into the language via the <throw> and <catch> constructs, the fault concept on BPEL4WS is directly related to the fault concept on WSDL and builds on it. As a result, the fault has to be defined explicitly when describing the messages in a Web service. In WSDL, the optional fault elements specify the abstract message format for any error messages that may be output as the result of the operation. To bridge the gap, the inference queue can be placed between two Web services. Process rules can be stored in the queue, such that the outputs of one service can be input to the queue as facts before they are fed into the next Web service. The inference mechanism can then determine if a Web service has performed the task and has generated results in expected fashion, based on the facts and a set of pre-defined rules.

6 Current Status, Future Work, and Conclusions

The current inference queue system is implement using the jDREW tool kit [13] in Java. We have deployed it as a Web service as well. It accepts input and generates output either in a Prolog-like syntax, or RuleML. We plan to offer it as open source.

In the future we plan to study how declarative descriptions of Web services and the links between them, as in BPEL4WS, may be transformed into running networks of Web services with inference queues proving the communication between them. Each inference queue could be loaded with rules that check the preconditions, in the sense of DAML-S, of the downstream service and monitor the effects of the upstream service.

This paper introduces the inference queue as a means of communicating meaningful information among Web services. It is based on a thread-safe priority queue data structure. It contains first order definite clauses and it inserts inferred facts into the queue, so that the output is sound, complete, fair, and irredundant. The insert and remove operations are responsive; even if there is an infinite set of inferred facts, no operation will need to wait forever. We propose to use the inference queue for transporting information between pairs of Web services; data from one service to the next can be mediated and transformed by the rules in the queue. Most Web services choreography specifications do not support comprehensive monitoring of systems built up from individual Web services, although a need for this is identified in WSMF and DAML-S. We suggest how the inference queues can use their rules to detect exceptional conditions, and pass these exceptions to a monitor receiving information from all inference queues in the system. Proposals for handling exceptions in Web services, such as in BPEL4WS, depend on the Web service detecting problems and throwing exceptions. But if a system architect seeks to incorporate a third-party Web service, s/he cannot depend on that Web service to throw faults appropriate for the rest of the system.

References

1. A. Arkin. Business Process Modeling Language. "http://www.bpmi.org/bpml-spec.esp", 2002.
2. Harold Boley. The rule markup initiative. http://www.ruleml.org, 2003.
3. BPMI.org. BPML‖BPEL4WS: A Convergence Path toward a Standard BPM Stack. "www.bpmi.org/downloads/BPML-BPEL4WS.pdf", 2002.
4. DAML-S Coalition. DAML Services. "http://www.daml.org/services/", 2003.
5. The DAML Services Coalition. DAML-S: Semantic Markup for Web Services. In *Proceedings of SWWS'01 The First Semantic Web Working Symposium*, pages 404–411, 2003.
6. Assaf Arkin et al. Web Services Choreography Interface (WSCI)1.0. "http://www.w3.org/TR/wsci", 2002.
7. T. Andrews et al. Business Process Execution Language for Web Services version 1.1. "ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf", 2003.
8. L. Wos, D. Carson and G. Robinson. Efficiency, completeness and the set of support strategy in theorem proving. *J. ACM*, 12:536–541, 1965.
9. Paul Levine. ebXML business Process Specification Schema Version 1.01. "http://www.ebxml.org/specs/ebBPSS.pdf", 2001.
10. W. W. McCune. Otter 3.0 users guide. Technical Report ANL-94/6, Mathematics and Computer Science Division, Argonne National Laboratories, Argonne, IL, 1994.
11. Sun Microsystems. WSCI 1.0 Specification - FAQs. "http://wwws.sun.com/software/xml/developers/wsci/faq.html", 2003.
12. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.
13. Bruce Spencer. The design of j-drew: a deductive reasoning engine for the web. In Kung-Kiu Lau Manuel Carro, Claudio Vaucheret, editor, *First Colognet Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems*, pages 155–166. Universidad Politécnica de Madrid, September 2002. CLIP4/02.0.
14. W3C. Semantic Web. "http://www.w3.org/2001/sw/", 2001.
15. W3C. Semantic Web Web Services. "http://www.w3.org/2001/11/11-semweb-webservices", 2001.
16. W3C. Web Services Choreography Working Group. "http://www.w3.org/2002/ws/chor/", 2003.
17. S. Weerawarana and F. Curbera. Business Process with BPEL4WS: Understanding BPEL4WS, Part 1. "http://www-106.ibm.com/developerworks/library/ws-bpelcol1", 2002.

Combining an Inference Engine with Databases: A Rule Server

Tanel Tammet and Vello Kadarpik

Tallinn Technical University
tammet@staff.ttu.ee,vello@softshark.ee

Abstract. Complex Semantic Web applications require easy-to-use tools for data and rule storage along with query mechanisms. We describe a prototype server software RLS which is used similarly to the ordinary usage of SQL RDBMS software in application programming. The server combines first-order theorem provers with several query and rule language layers for application development in the Semantic Web context.

1 Introduction

The goal of the paper is to describe experiences while implementing an actual industrial system, using Semantic Web technologies as a basis. We will give an overview of the task, approach, concrete problems tackled and the solutions we have chosen.

Special attention is paid to the problem of integrating resolution-based theorem provers into the system where a significant number of the rules are expected to stem from large ontologies and most of the data is kept in legacy databases.

Our actual implementation is still in a pilot phase and is likely to undergo several changes and extensions until its actual use by a client.

The system we are building consists of two main layers, one being a concrete software application aimed at the end users, another being a tool aimed at application developers in the Semantic Web context.

- A fund management software.
- Underlying rule server software RLS.

Our main target is the rule server software RLS, which is used as a tool for building actual implementations for clients. The fund management software, although a goal in itself, is at the same time considered to be a testbed for the practical usability of the rule server.

We note that the fund management software is not a perfect example for a Semantic Web application. The facts and rules used originate from different sources, but due to the nature of the application, they could, in principle, be easily collected into a common database.

The rule server software, on the other hand, is aimed to be used in a Semantic Web setting, where different knowledge items are regularly obtained from various sources over a network.

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 136–149, 2003.

The fund management software uses a restricted subset of the features of RLS. However, we claim that the capabilities used in this particular project are actually critical for typical Semantic Web applications, hence the application is highly relevant as a testbed.

We will avoid the issues of namespaces, XML etc. in the paper. Instead, we concentrate on the practical usability issues of the underlying machinery for logical derivations.

We are planning to release RLS along with source code, under a licence allowing free use for educational and research purposes as well as free use during the commercial software development process. The initial release is currently planned for the end of October, 2003.

2 Fund Management Software: Context and Requirements

We will first describe the context and requirements posed by the fund management software.

The client ordering the software is a fund management company, owned by a branch of an international bank. The client operates a number of funds: on the one hand, buying and selling shares of the funds to the clients, on the other hand buying and selling shares, foreign currencies and a number of different financial instruments, comprising the property of funds.

The actual problem the software is designed to alleviate is checking the rules posed by various funds, during the day-to-day operations of fund managers. A fund manager is obviously not allowed to operate with financial instruments at will. It must be guaranteed that after every transaction (for example, buying some shares) the rules for the fund are still satisfied.

The rules for the different funds are numerous and change relatively often. Hence it is hard for the manager to quickly check (before or after a transaction) that the rules are not violated after the transaction. In particular, in case the rules *are* violated, the manager wants to know the reason of violation.

Some typical examples of rules:

- The amount of shares owned by a fund X should contain at least P_o percent of shares of companies from OECD countries.
- The fund should contain no more than P_d percent of deposits, P_i percent of interest investments, P_s percent of stocks.
- The maximal percentage of financial instruments emitted by a single emitent X should be less than P_e.

Pension funds are a good example showing different origins of rules. Since - in our case - the central bank of the country imposes several rules for pension funds, the rules for pension funds originate from the following sources:

- Rules imposed by the central bank of the country.
- Rules for all the funds of our client.
- Specific rules for a pension fund X.

Of course, it would be possible to design software which checks the rules, using ordinary program code. However, since the rules change, the code should be modified fairly often, thus making the maintenance of the system costly. Ideally we would like to have a situation where a manager is in principle able to change the rules himself and to experiment with the effects of these changes, without competence in programming.

However, we note that it would be highly impractical to encode all the basic facts about shares, transactions made, etc. in a rule server. These facts are already kept in a database system, maintained separately from our system.

Our implementation of the system contains three large building blocks:

- Ordinary program code. In this particular system we use Python. Ordinary program code is used for creating the user interface, for registering transactions in a database, listing contents of the database, querying the rule server.
- An ordinary SQL database, storing basic facts and answering basic queries about these.
- A rule server, storing the rules for the funds along with the ontology of financial terms as used in the rules. The rule server answers complex questions, using automated theorem proving (ATP) techniques, augmented as necessary (described in later sections). A rule server is called from the ordinary program code and it calls the SQL database if necessary.

3 Architecture and Targets of the Rule Server

The rule server is the core of our system. While building the rule server we have taken into consideration the experiences from building the financial system along with the various well-known needs arising when building distributed, web-based systems.

The core of the rule server itself is our high-performance ATP system based on the ideas and algorithms from Gandalf, a theorem prover built and maintained by one of the authors of this article. See [Tammet97], [Tammet98], [Tammet02].

Similarly to Gandalf, our ATP system is targeted at proving theorems of full first order predicate calculus (FOL).

Hence the input language of the rule server is not restricted to RDF, RDFS or OWL. Although it accepts these languages, it also accepts full FOL.

Our work on the fund management application has immediately shown that even full FOL is inconvenient for application development. It has been necessary to build significant extensions to the FOL prover.

The rules we handle are basically formulas of first order predicate logic (FOL), extended by additional, non-FOL features in the rule language.

The rule server is aimed at being used in a manner similar to the ordinary usage of SQL servers: like an SQL server, so is the rule server called from the ordinary program code of the application programmer. The rule server is not optimised for storing large amounts of factual data. Instead, an SQL server is used for that.

Since typical software systems maintaining large amounts of data do use ordinary database servers, it would be nonproductive to attempt to "take over" the role of a database server.

We target our rule server to be used in the scenario analogous to the scenario outlined in the previous chapter. Architecturally, a typical application using the rule server consists of three main components:

- Ordinary program code.
- An ordinary SQL database engine.
- The rule server engine.

3.1 Basic Architectural Considerations

Importance of Being a Server. Observe that SQL database engines are typically server applications. Application programmers contact the database servers, send queries and get answers. One database engine instance is often used by many different application programs (and vice versa: an application program often uses a number of different database servers).

Hence we have implemented the rule server in a similar manner to a typical SQL database engine. It is a server program, waiting for the application programs to take contact, send queries and get answers. It is capable of simultaneously serving a number of different application programs and different queries.

Rules and Rule Sets. All the rules kept in the server have a unique name, given by the user storing the rule. Names are necessary, since:

- When queries are asked, it must be specified (either in a query itself or the context of a query) which rules are used during answering the query. See also "rule sets" in the following.
- Rules are sometimes changed or deleted from the server altogether.
- Names are used for grouping several rules into a rule set.

Rules can be grouped into rule sets. A rule set is used when asking a query, in order to specify which rules are used for answering the query.

For example, in case a set of rules is obtained from a web page, it is natural to use the URI of the web page as a name of the rule set.

A Language and Queries Matching SQL. Since typical application programmers are used to using the SQL servers and do have competence of the SQL language, we have designed the language of our rule server to be as similar to SQL as possible. This facilitates relatively easy takeup of the technology by the programmers, who are typically not well versed in FOL or the Semantic Web.

The structure of the "answer" to a query is a particularly important question. Typical FOL provers give a yes/no answer (or fail to terminate, or terminate due to internal limits) indicating whether an input formula is satisfiable (or valid)

or not. Some, but not all the provers are capable of additionally returning the proof, if found.

Such capabilities are clearly not enough for programmers used to obtaining a table of results from the prover.

We are using the well-known "answer predicate" machinery of ATP systems to find answers to queries (essentially, objects substituted for existentially quantified variables in a proof). Since a large number of different objects may be valid answers, the main part of the answer to any query is a table containing rows of suitable objects, analogously to the answers of SQL queries. We will describe the structure of answers in the following chapters.

Strings-Based Interface to the Application Program Code. The core of the rule server - application program interface is written in the C language and is given as a library to the application programmer. In order to perform a query from a C program the programmer constructs a query string and calls a suitable C function in the library, with the pointer to the string as an argument to the function.

The result of the query is, again, a string following specific conventions of presenting the various parts of the answer. The answer can be either parsed by the application program or the programmer can use suitable library functions to access various parts of the result string.

Data Types. SQL programmers are used to a large variety of data types. At the very least, programmers expect to have integers, strings and dates in the query language they use.

We have therefore incorporated several basic datatypes into the core of our language. From these it is easy to construct more complex types.

From the FOL standpoint the datatypes are encoded as terms led by the datatype constructor function and containing ordinary constants (or other datatypes) as arguments. For example, an integer 15 is encoded as $int(15)$ in the FOL language used internally in the rule server. A date 2003-05-31 is encoded as $date(int(2003), int(5), int(31))$. The primitive type constructor functions are hidden from the user of the system.

Built-in Functions. Despite the fact that computable functions can, in principle, be defined in our FOL-based rule language, such definitions are inevitably slow when they are run by the theorem prover.

Hence the system contains a mechanism for adding user-defined functions. The functions are programmed in C. The theorem prover uses the functions as a "black box", essentially computing the value of a function each time a user-defined function occurs in a term handled by the prover.

Since some of the user-defined functions may be highly costly, it is possible to indicate that a specific function should be cached by the prover. For such functions the prover keeps an discrimination-tree indexed table of recently handled argument tuples along with the computed results.

Compilation of Rule Sets. Consider the typical scenario where an application program asks a query Q, indicating that rule sets A, B and C have to be used for answering the query. It is likely that the program will soon pose other queries Q_1, Q_2, etc, each indicating that the same rule sets A, B and C have to be used. In order to achieve acceptable efficiency it is necessary to compile the conjuction of A, B and C before answering the query: the same compilation can be used over and over for the following queries Q_1, Q_2, etc.

Obviously, since not all queries use exactly the same rule sets (and the rule sets themselves occasionally change) the server has to create and keep several compilations of sets of rule sets. It automatically discards a compilation whenever some rule set on which the compilation is based on changes.

We will give an outline of the compilation and query answering process used in the rule server. The details and explanations of various parts of the compilation process are explained in the following chapter introducing the concept of *chain resolution*.

- *Analysis.* Analyse the rule base R to determine whether R or a large subset R_d of R falls into a known decidable class. Determine a suitable ordering-based strategy *os* for R or R_d.
- *Terminating strategy run.* Run resolution with the strategy *os* combined with the chain strategy on R or R_d, for N seconds, where N is predetermined limit. This will build both a chain box C_b and a set of new clauses R_1. In case inconsistency was found, stop, otherwise continue.
- *First filtering.* If R was not in a decidable class, then the previous step probably did not terminate. Even if R was decidable, N may have been too small for the previous step to terminate in time. Hence, in case the strategy did not terminate, the set of new clauses R_1 is likely to be very large. It is sensible to employ heuristics to keep only some part R_f of the derived clauses R_1 for the future.
- *Chain box building.* Regardless of whether the terminating strategy run terminated in time or not, it will be advantageous to increase the size of a chain box by performing another run of resolution on $R \cup R_f$, this time using a different strategy, optimised for fast derivation of chain clauses. After a predetermined time of M seconds the run will be stopped (or it may terminate by itself earlier).
- *Final filtering and storage.* The set of clauses R_2 obtained after the previous run will be heuristically filtered, again storing a subset R_{f2} of R_2. Finally, both the chain box constructed so far as well as the new clauses R_{f2} along with the (possibly) simplified initial clause set R will be stored as a *compilation* of the initial clause set R.

4 Chain Resolution

In practical applications it is fairly common to arrive to a hard situation where a proof task contains both an "ordinary" FOL part well suited for resolution and a large special theory not well suited for resolution. One approach in such

situations is to extend the resolution-based methods with a specialised "black box" system. Unfortunately, for complex "black boxes" it is often either very hard or impossible to prove completeness of the combined system.

We propose a simple "black box" system called "chain resolution" extending standard resolution for theories appearing in both contexts:

- verifying state transition systems
- answering queries in case large ontologies are present in the theory, as is common in the applications of Semantic Web and terminological reasoning

In a yet unpublished paper [Tammet03] we have shown that chain resolution is sound and complete. More importantly, it can be combined with several crucial strategies of resolution: set of support and ordering strategies.

An important feature of chain resolution is that it is easy to implement and easy to incorporate into existing theorem provers. The presence of a chain resolution component should not seriously degrade the performance of a theorem prover for tasks where chain resolution is useless, while significantly improving performance for tasks where it can be employed.

4.1 Informal Introduction to Chain Resolution

We assume familiarity with standard concepts of resolution, see [Handbook01]. We will use the following notions in our presentation:

Definition 1. *A* signed predicate symbol *is either a predicate symbol or a predicate symbol prefixed by a negation.*

Definition 2. *A* chain clause *is a clause of the form*

$$A(x_1, \ldots, x_n) \vee B(x_1, \ldots, x_n)$$

where $x_1, \ldots, x_n$ are different variables, A and B are signed predicate symbols.

Definition 3.
A clause C' is a propositional variation *of a clause C iff C' is derivable by a sequence of binary resolution steps from C and a set of chain clauses.*

Definition 4.
A search space *of a resolution prover is a set of clauses kept by the prover at some point during proof search.*

Definition 5.
A passive list *of a resolution prover is a subset of a search space containing exactly these clauses which have not been resolved upon yet.*

Throughout the paper we will only consider chain clauses of a simpler form $A(x) \vee B(x)$ where A and B are unary predicates. However, the presented properties of chain resolution obviously hold also for the general chain clauses as defined above.

We note that the concept of the chain clause can be sometimes extended to cases where the literals $A(x_1, \ldots, x_n) \vee B(x_1, \ldots, x_n)$ contain constant arguments in addition to the distinct variables. In order for this to be possible it must be shown that the constants occur in positions where there are either no variable occurrences in the corresponding literals in non-chain clauses or such clauses can be finitely saturated to the point that the offending variable occurrences disappear. In such cases we can essentially treat a predicate symbol together with some constant occurrences as a new, specialised version of the predicate symbol.

The basic idea of chain resolution is simple. Instead of keeping chain clauses in the search space of the prover and handling them similarly to all the other clauses, the information content of the chain clauses is kept in a special data structure which we will call *the chain box* in the paper.

Chain clauses are never kept or used similarly to regular clauses. At each stage of the proof search each signed unary predicate symbol P is associated with a set S of signed unary predicate symbols $P'_1, P'_2, \ldots, P'_m$ so that the set S contains all these and exactly these signed unary predicate symbols P'_i such that an implication $\neg P(x) \vee P'_i(x)$ is derivable using only the chain clauses in the initial clause set plus all chain clauses derived so far during proof search.

The "black" chain box of chain resolution encodes implications of the form $A(x) \Rightarrow B(x)$ where A and B may be positive or negative signed predicates. The chain box is used during ordinary resolution, factorisation and subsumption checks to perform these operations modulo chain clauses as encoded in the box. For example, literals $A(t)$ and $B(t')$ are resolvable upon in case $A(t)$ and $\neg A(t')$ are unifiable and $\neg B$ is a member of the set S associated with the signed predicate symbol $\neg A$ inside the chain box. Due to the way the chain box is constructed, no search is necessary during the lookup of whether $\neg B$ is contained in a corresponding chain: this operation can be performed in constant, logarithmic or linear time, depending on the way the chain box is implemented.

A suggested way to implement the chain box is using a bit matrix of size $(n * 2) * (n * 2)$ where n is a number of unary predicate symbols in the problem at hand. Even for a large n, say, 10000, the bit matrix will use up only a small part of a memory of an ordinary computer.

The main effect of chain resolution comes from the possibility to keep only one propositional variation of each clause in the search space. In order to achieve this the subsumption algorithm of the prover has to be modified to subsume modulo information in the chain box, analogously to the way the resolution is modified.

Since the main modification of an "ordinary" resolution prover for implementing chain resolution consists of modifying the equality check of predicate symbols during unification and matching operations of literals, it would be easy to modify most existing provers to take advantage of chain resolution.

4.2 Motivation for Chain Resolution

During experimental resolution-based proof searches for query answering in large ontologies is has become clear to us that:

- New chain clauses are produced during proof search.
- Some chain clauses are typically present in the proof, making a proof larger and hence harder to find.
- Chain clauses produce a large number of propositional variations of non-chain clauses, making the search space larger.

We will elaborate upon the last, critical item. Suppose we have a clause C of the form $A_1(t_1) \vee \ldots \vee A_n(t_n) \vee \Gamma$ in the search space. Suppose search space also contains m_i chain clauses of the form $\neg A_i(x) \vee A'_1(x), \ldots \neg A_i(x) \vee A'_{m_i}(x)$ for each i such that $1 \leq i \leq n$. Then the resolution method will derive $m_1 * m_2 * \ldots * m_n$ new clauses (propositional variations) from C using chain clauses alone.

Although the propositional variations of a clause are easy to derive, the fact that they significantly increase the search space will slow down the proof search. Hence we decided to develop a special "black box" implementation module we present in this paper. Chain resolution module allows the prover to keep only one propositional variation of a clause in the search space and allows the prover to never use chain clauses in ordinary resolution steps.

An important aspect of the Semantic Web context is that a crucial part of the prospective formalisations of knowledge consist of formalisations of ontologies. Taxonomies, ie hierarchies of concepts, typically form a large part of an ontology. Most prospective applications of the Semantic Web envision formalisations where the ontologies are not simple taxonomies, but rather more general, ordered acyclic graphs.

We will bring a small toy example: "a mammal is an animal", "a person is a mammal", "a person thinks", "a man is a person", "a woman is a person". The following clause set encodes this knowledge as implications in FOL: $\{\neg mammal(x) \vee animal(x), \neg person(x) \vee mammal(x), \neg person(x) \vee thinks(x), \neg man(x) \vee person(x), \neg woman(x) \vee person(x)\}$.

Observe that all the clauses in the previous clause sets are chain clauses. Chain clauses can also capture negated implications like "a man is not a woman".

A full ontology may also contain knowledge which is not representable by simple implications. Prospective applications and typical queries formalise knowledge which is also not representable by chain clauses. However, ontologies and simple implications are currently seen as a large, if not the largest part of prospective Semantic Web applications.

Since the Semantic Web context is strongly focused on the speed and (whenever possible) decidability of query answering, ordinary resolution methods are not well-suited for the applications. Instead, a number of efficient, specialised decription logic languages and derivation mechanisms have been devised by the terminological reasoning community, see [Handbook03].

However, most of these derivation mechanisms are incomplete in the context of full FOL: they require knowledge to be encoded in a severely restricted subset of FOL.

Chain resolution, although a very simple mechanism, makes efficient query answering possible also for the resolution-type methods.

5 Extensions to FOL

During the implementation of the fund management software we noticed that the majority of rules for funds cannot be easily encoded in FOL.

Consider an example rule: the maximal percentage of financial instruments emitted by a single emitent X should be less than P_e.

In order to check this rule the prover has to filter out the value of instruments in the fund for each emitent. The data about all the instruments owned by the fund is kept in the SQL database. In SQL it is straightforward to find all instruments owned for a particular emitent, and then calculate their summed value.

However, in the FOL approach there is no guarantee that the instruments found from the SQL base are all there is. Hence we cannot calculate their amount: we only know the lower limit, but not the upper limit.

There are several ways to overcome the problem. We could explicitly axiomatise (as a huge disjunction) that any instrument owned is actually present in the SQL base. However, such a solution is

- highly inconvenient,
- very hard for the prover to handle efficiently,
- hard to maintain in case new instruments are added to the SQL base.

The second approach is to introduce a special closed-world negation similarly to Prolog. We have experimentally introduced such a negation, but do not use it for the financial application.

The third approach - the one actually used in our system - is to use special higher-order-looking constructions in the language, allowing to express the sums, means, etc. of data obtained from derivable facts.

The problem with the second and third approach is that in the general case there is no guarantee a prover will manage to derive all the facts we want to count or sum. However, in the settings like the application we are working with, it is possible to show that the prover is always capable of doing this. In other words, we can show that the whole set of rules falls into a decidable class where we do have such a guarantee, hence the counts and sums are actually computable.

6 Language Layers of the Rule Server

The rule server uses and understands a number of different (extended) FOL languages. Some of the languages are internal, never shown to the application programmer using the rule server. Most of the languages, however, can be used by the application programmer.

- *Internal layer.* This layer is not shown to any users and developers except prover engine developers. Every data item is encoded as an integer plus

additional data structures for information. Facts and rules are clauses with internal decorated pointer structure.

- *Raw input language layer.* This layer could be accessed by application developers, but is rarely necessary. Raw input layer is used internally by the prover as a result of parsing an external file given in some specific language. Raw input language uses scheme lists for representing clauses and additional information. Prover engine converts raw input language into the internal layer before starting proof search.
- *Programmer layer.* This is the main language layer used by the application programmer for writing rules and facts and asking queries from the system. It is basically a "syntactic sugar" on top of the raw input layer. A special module parses that language into the raw input layer language for the prover engine.

  During actual usage of the system (ie calling the RLS from ordinary programs) the sentences in the programmer layer are typically surrounded by the data definition, manipulation and query language layer.

  Programmer layer presents first order logic similarly to a programming language and (somewhat) to a natural language. Programmer layer is designed to be used by programmers and sophisticated users. It can be extended and modified as need arises.

  The programmer layer syntax is somewhat similar to the triples syntax of RDF or the language N3 used in the paper [Berners-Lee,Connolly,Hawke03]. The language allows to write predicates with arity one and two in an infix notation, while predicates with a higher arity are written in the prefix notation.

  The following are some example sentences in the programmer language layer:

```
"John" has father "Michael".
"John" has age 10.
"Michael" is a father of "Chris".
if Person1 has father Person2 and
   Person2 has father Person3 then
   Person3 is grandfather of Person1.
```

  We will not bring the full syntax and semantics of the programmer language in the paper. Basically, the language is transformed to the extended FOL.

  As said, in addition to ordinary FOL the language contains special higher-order query functions not directly representable in first order logic. Special functions `$sumforallfacts` and `$maxforallfacts` are two typical examples. Both functions (internally) loop over all the given or derivable facts which correspond to the criterias given in the function term and compute sum (or max) of all the integers in a given place.

  An example of using $sumforallfacts:

```
$sumforallfacts
    (Percentage,
     tickerpercentage(Percentage,Ticker),
     deposit(Ticker)).
```

loops over all given and/or derivable atoms of the form

`tickerpercentage`(..., ...)

where the first argument of the `tickerpercentage(...,...)` can be proved to be a deposit.
An alternative sugared syntax:

```
$sumforallfacts
    (Percentage,
     Percentage is tickerpercentage of Ticker,
     Ticker is deposit).
```

– *Data definition, data manipulation and query language layer.*
This layer has a syntax similar to SQL. The purpose of the layer is to:
  - Allow users to add facts and rules to the database, delete or replace existing rules and facts.
  - Declare the relations between SQL tables (or queries) and the predicates/functions in the programmer language layer. Basically, an RLS user can tell the RLS system that certain tables or queries in some SQL database should be automatically mapped to the facts usable by the FOL prover component as a part of some rule set.
  - Ask queries, where the contents of the query are presented in the programmer language layer.

This is the language layer normally used when interacting with the RLS system.
The following example demonstrates actual usage of the data language layer with the previous fact/rule examples from the programmer language layer.

```
Insert into rules family fact f1 as
   "John" has father "Michael".
Insert into rules family fact f2 as
   "John" has age 10.
Insert into rules family fact f3 as
   "Michael" is a father of "Chris".
Insert into rules family rule r1 as
  if Person1 has father Person2 and
     Person2 has father Person3 then
     Person3 is grandfather of Person1.
Select using rules family all X1,X2 where
  X1 is grandfather of X2.
```

– *SCL layer.* SCL (simple common logic) is a FOL-based language currently under development by the SCL working group, see [SCL]. It has similar goals, yet smaller scope, than CL (common logic). SCL has different concrete syntaxes: a KIF-like syntax, a "traditional FOL" syntax, an XML-based syntax.

We use SCL both as a legitimate input language and a "glue" language between different Semantic Web languages and the raw internal language layer.
We have built converters from the current version of SCL to the raw internal language, but since SCL is not fully finished, the SCL converters are likely to change.

- *RDF/RDFS layer*. We have built the first versions of RDF/RDFS converters to SCL, based on the semantics of RDF from [RDF]. From SCL we finally obtain the raw internal language layer.
- *OWL layer*. We are currently not reading the OWL language. However, we plan to add the OWL-to-SCL converter in the near future, based on the semantics of OWL from [OWL].

7 Queries and Answers to Queries

We will now describe the structure of queries and answers. There are, essentially, two kinds of queries:

- Queries asking for a yes/no/unknown answer.
- Queries asking for specific data items a la Prolog or SQL.

In order to pose a query of the second type, a query string has to explicitly indicate the variables in the query for which we want to get the actual values. Additionally it is possible to add either a qualifier "all" or an integer qualifier N, indicating that we want to get all possible value tuples or just N value tuples. If no qualifiers are given, RLS will attempt to find a single value tuple.

The result string of a query consists of the following parts:

- Result code. Successful/not successful/error code for a specific error
- Result string: a human-readable string explaining the previous code.
- Comma-separated lines, each line corresponding to one tuple of values found.
- A list of strings, each starting with `EXPLANATION STARTS` and ending with `EXPLANATION ENDS`. Each element of the list is a proof of the derivation of one of the value tuples.

8 Implementation Details

The server core and protocol implementations are written in C. The extended FOL prover component of RLS, as well as the converters to the raw input language are written in Scheme and compiled to C. The converters for the SQL glue language are created using YACC. Several utilities and extensions are written in Python.

As a rule database engine either PostgreSQL or Oracle can be used. The SQL RDMBS interface to RLS is written in C.

References

Berners-Lee,Connolly,Hawke03. Berners-Lee, T. Connolly, D. Hawke, S. Semantic Web tutorial using N3. 2003.
http://www.w3.org/2000/10/swap/doc/tutorial-1.pdf

Handbook01. Robinson, A.J., Voronkov, A., editors. Handbook of Automated Reasoning. MIT Press, 2001.

Handbook03. Baader, F., Calvanese, V., McGuinness, V., Nardi, D., Patel-Schneider, P., editors. The Description Logic Handbook. Cambridge University Press, 2003.

Tammet97. Tammet, T. Gandalf. Journal of Automated Reasoning vol 18 No 2 (1997) 199–204.

Tammet98. Tammet, T. Towards Efficient Subsumption. In CADE-15, pages 427-441, Lecture Notes in Computer Science vol. 1421, Springer Verlag, 1998.

Tammet02. Tammet, T. Gandalf. http://www.ttu.ee/it/gandalf/, Tallinn Technical University.

Tammet03. Tammet, T. Extending classical theorem proving for the Semantic Web. Draft paper.

SCL. Simple Common Logic. Mailing list archive of the working group.
http://philebus.tamu.edu/mailman/listinfo/scl

RDF. RDF Semantics. W3C Working Draft 23 January 2003. W3C. Hayes, P. ed.
http://www.w3.org/TR/rdf-mt/

OWL. OWL Web Ontology Language. Semantics and Abstract Syntax. W3C Working Draft 31 March 2003. W3C. Patel-Schneider, P.F, Hayes, P., Horrocks, I., eds.
http://www.w3.org/TR/owl-semantics/

Extraction of Structured Rules from Web Pages and Maintenance of Mutual Consistency: XRML Approach

Juyoung Kang and Jae Kyu Lee

Graduate School of Management
Korea Advanced Institute of Science and Technology
207-43 Cheongryang, Seoul 130-012, Korea
{jykang,jklee}@kgsm.kaist.ac.kr
http://kgsmweb.kaist.ac.kr

Abstract. Web pages provide valuable knowledge for human comprehension in text, tables, and mathematical notations. However, the extraction and maintenance of structured rules from the Web pages are not easy tasks. To tackle these problems, we adopt the eXtensible Rule Markup Language framework. The RIML (Rule Identification Markup Language) and RSML (Rule Structure Markup Language) are two compliant representations in XRML for this purpose. RIML identifies the implicit rules in the Web pages possibly using multiple pages to make a rule or rule group. RSML specifies the complete rule structure to be processed by software agents or expert systems.
In this study, we cover the natural text, tables, and implicit numeric functions in the texts. In order to fulfill the research goal, we define the necessary tags for the rule extraction and maintenance in XRML. Typical ones include tags for rule grouping, tabular rules, numeric operators, and functions. The rule acquisition process consists of rule base design, rule identification with RIML, and rule structuring with RSML. The maintenance process for the revisions that may occur either in Web pages and structured rules is also described. The approach is demonstrated with the shipping cost comparison on the electronic book stores.

1 Introduction

The World Wide Web could be a valuable resource supplying knowledge covering many subject areas. We can exploit the implicit knowledge in Web pages to construct a rule base which provides a specialized reasoning service. However, the implicit knowledge in Web pages should be transformed into structured rules to be utilized in rule based systems. In consideration of cost and time, constructing a rule base by transforming knowledge in Web pages is more advantageous than starting from the beginning with traditional development methodology.

However, we have to overcome several difficulties when acquiring rules from Web pages. First, it is difficult to extract rules that can be used for inference directly from Web pages. Also, rules in the rule base should be consistent with the knowledge

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 150–163, 2003.

in Web pages. In addition, knowledge to construct the rule base is dispersed throughout multiple Web pages.

Several methods and tools to acquire knowledge from the Web have been developed such as natural language processing [2, 16], machine learning [1, 3, 8], Web mining [9], and ontology based learning [4, 7]. However, acquired knowledge using those methods consists of mostly concepts, information or simple knowledge [2, 14] rather than inference rules. Research related with the Semantic Web [5, 17] also focuses on providing knowledge acquisition methods. But, these efforts of knowledge acquisition are limited to ontology building [6, 7] and there are few methods for rule acquisition in this area.

eXtensible Rule Markup Language (XRML) version 1.0 gives some of the answers to the above problems [10]. XRML is a language that represents the implicit rules in such a way as to allow software agents to process them as well as to browse them for human comprehension, and provides consistency between knowledge for humans and rules for software agents [10]. XRML consists of two components – RIML for humans and RSML for software agents. RIML identifies the existence of implicit rules in the Web pages and RSML represents a formal structure to be processed with inference engines [10].

Figure 1 shows the procedure of rule acquisition from the Web pages into the rule base using RIML and RSML. Let us briefly overview the rule acquisition procedure using XRML. First, we identify basic rule components from the Web pages and build RIML documents. Second, we structure those basic components to structured rule formats and build RSML documents. Last, RSML document is translated into rule base automatically. Yet, there are more detailed problems of rule acquisition and consistency maintenance which can't be resolved with XRML 1.0.

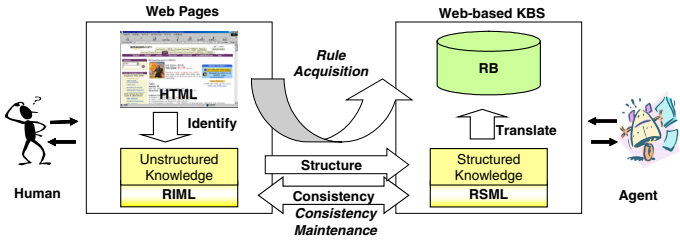

Fig. 1. Rule Acquisition and Consistency Maintenance with XRML

In this paper, we propose the methodology of rule acquisition from multiple Web pages and consistency maintenance between Web pages and the rule base by extending XRML 1.0. First, in order to cover the natural text, tables, and implicit numeric functions in Web pages, we need to extend the tags of XRML 1.0. So we define XRML 2.0. Second, we propose the rule acquisition procedure utilizing those extended tags. We also describe an answer for how to acquire rules from multiple Web pages and how to merge those rules into a rule base. A knowledge engineer can get the aid of an intelligent rule editor in the process of identifying and structuring rules. Moreover, we present extended tags and acquisition procedures of table, operator,

and function. Third, we describe the consistency maintenance procedure between multiple Web pages and the rule base by proposing detailed update, add, and delete procedures with XRML tags. Finally, we show the usefulness of our approach with an application we call ComparePrice. The objective of our application is to acquire rules about delivery cost from different online bookstores and compare the price of books including exact delivery cost of selected books.

The content of this paper is organized as follows. In section 2, we define and explain extended tags for rule acquisition and consistency maintenance. In section 3 and 4, we propose the detailed procedure of rule acquisition and mutual consistency maintenance with extended XRML tags. The application constructed with our approach is illustrated in section 5 and finally we conclude and discuss issues for further study.

2 XRML 2.0 for Rule Acquisition and Consistency Maintenance

Since we choose our target rule format as the backward inference rule, we exploit the syntax of UNIK-BWD [12]. Figure 2 shows the rule syntax of UNIK-BWD, which describes the components to be acquired. In this section, we address the detailed issues of rule acquisition and mutual consistency maintenance between Web pages and the rule base. Based on these issues, we propose newly defined tags of XRML.

```
(BWD-RULE rule-name
  [RULE-GROUP group-name]
  [PRIORITY integer_type]
IF
   predicate1
   predicate2
   …
   predicaten
THEN
   action1
   action2
   …
   actionn)

predicate = (operand1 operator operand2 … operandn),
              User-defined Fuction
action = (operand1 operator operand2 … operandn),
            User-defined Fuction
operator = NOT, AND, OR, NOR, XOR, IS(=), ISNOT(<>), <=, >=, >, <, +, -,
     *, /
operand = predicate(action), variable, value
```

Fig. 2. Rule Syntax of UNIK-BWD

2.1 Issues and Extended Tags for Rule Acquisition

In general, a rule base is composed of several rule groups and these rule groups are scattered over multiple Web pages. Also, different rule groups may exist in one Web page. So, we added <RuleGroup> and <RuleGroupTitle> tags in XRML 2.0 to represent such rule groups. Although most IF and THEN symbols are omitted and not clearly identified in the identification step, we added <IF> and <THEN> tags to RIML to automate structuring as much as possible.

Connectives such as AND, OR, and NOT are mostly omitted in Web pages. So, we didn't consider identifying connectives in the identification step in XRML 1.0. However, to complete connectives in the structuring step, the knowledge engineer should refer to natural text in the Web pages again. If we identify connectives in the identification step, it is very helpful for the knowledge engineer to structure connectives because the knowledge engineer doesn't need to refer to the Web pages again. So, we added tags for connectives like <AND>, <OR>, and <NOT> in RIML.

The most fundamental components of a rule are variables and values and we identify these components using <variable> and <value> tags. Yet, it is hard to distinguish which symbols in Web pages can be variables or values. In response to this problem, we can exploit the ontology from similar existing rules to identify variables and values. Sometimes, variables or values are omitted in HTML. But, we can guess the variable or the value using the context and common sense. We added the 'name' attribute to variable and value entities to represent any omitted variable or value. A very common issue of rule acquisition is a synonym problem. In XRML 2.0, we added the 'standard' attribute to variable and value entities to represent the standard word of synonyms.

Some components of rules can be acquired by a different method. We added tags for such special components like table, operator, and function. We will describe these tags and the acquisition procedures of these components in section 3.2.

2.2 Issues and Extended Tags for Consistency Maintenance

Since the content in the rule base and the content in Web pages consist of the same knowledge, the change in Web pages or the rule base should reflect each other. But, this reflection of the change is not easy because the representation of the same rule is different in each format and there is no information to link rules each other. The linkage between the Web page and the rule base is required to reflect this change. In our research, we use RIML and RSML tags to link Web pages and the rule base.

To link a symbol of RIML to the corresponding symbol of RSML, we propose multiple connection levels starting from RIML and RSML. In the first level, an RIML document and an RSML document can be connected to each other by having the other's URL using <URL> tag. In the second level, the same rule group of both RIML and RSML documents are connected to each other by <RuleGroupTitle>. In the third level, each rule within the rule group is linked by an "id" which is given to the rule as unique attribute. In the final level, we gave an "id" to the variable as a unique attribute. All the variables, values, and operators have the "id" attribute in RIML, but only the variables in RSML have the "id" attribute. In RSML, a value and an operator are surrounded by the variable tag and consequently don't require the "id" attribute. For example, the sentence of RIML as follows:

<variable1> items </variable1> <operator1 type="GT"> greater than </operator1> <value1>2</value1>

is changed in RSML like as follows:

<items operator="GT" id=1>2</items>

Table 1 shows the comparison between the tags in XRML 1.0 and the extended tags in XRML 2.0 [11].

Table 1. XRML 1.0 vs. XRML 2.0

■ **RIML 1.0 vs. RIML 2.0**

Element - Attributes of RIML 1.0	Element - Attributes of RIML 2.0
RIML - version	RIML - version
URL	***RuleGroup*** ***RuleGroupTitle*** URL – ***rsml*** ***RuleTable*** ***RuleTableTitle***
Rule RuleTitle	Rule - ***id*** RuleTitle ***IF/THEN*** ***AND/OR/NOT***
variable - id value - id	variable – id, ***name, standard*** value – id, ***name, standard*** ***function – type*** ***arg – id*** ***operator – type, id***

■ **RSML 1.0 vs. RSML 2.0**

Element - Attributes of RSML 1.0	Element - Attributes of RSML 2.0
RSML - version	RSML - version
URL	***RuleGroup*** ***RuleGroupTitle*** URL – ***riml*** ***RuleTable*** ***RuleTableTitle***
Rule RuleTitle IF/THEN AND/OR	Rule - ***id*** RuleTitle IF/THEN AND/OR/***NOT***
variable value	variable – id, ***operator*** value ***function – type*** ***arg – id***

3 Rule Acquisition with XRML

In this section, we propose the rule acquisition procedure utilizing extended tags which were described above. Also, we describe extended tags and acquisition procedures of table, operator, and function.

3.1 Rule Acquisition Procedure

There are three phases of rule acquisition. The first phase is the design phase. The knowledge engineer should design the application which will use the extracted rules. The structure of the rule base depends on the design of the application.

The second phase is the identification phase. Since the rules in the rule base are dispersed over multiple Web pages, the knowledge engineer should identify corresponding rule groups in Web pages and associate them with rule groups in the rule base. After associating Web pages and the rule base, the knowledge engineer identifies and associates basic components of rule and generates RIML documents.

The third phase is the structuring phase. In this phase, the knowledge engineer should link and structure the basic components which were identified in the previous phase. To achieve this, the knowledge engineer merges rule groups which are related to one rule base and structures them into one RSML document.

The knowledge engineer can get helps from the XRML editor during the identification phase and the structuring phase. The XRML editor is an intelligent editor which is developed to support rule acquisition and consistency maintenance. The rule

acquisition procedure is divided into two parts. While one is executed by the knowledge engineer only, the other is supported by the XRML editor. Figure 3 shows the whole procedure of rule acquisition conducted by the knowledge engineer and the XRML editor.

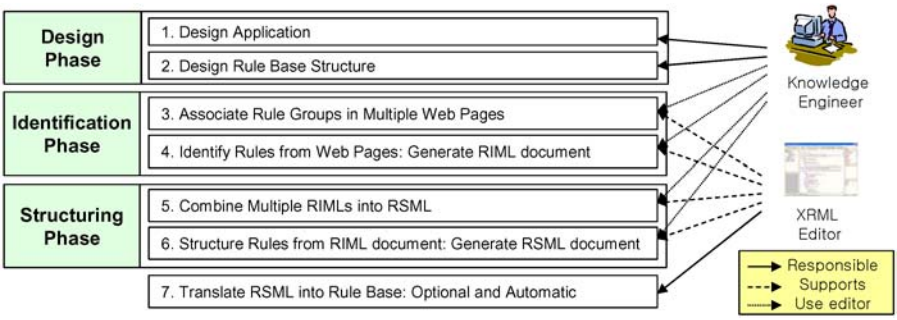

Fig. 3. Overall Rule Acquisition Procedure with XRML

Association of Rule Base with Multiple URLs. The purpose of this step is to identify rule groups from Web pages which constitute one rule base and to associate them. The rule groups which constitute the rule base are scattered over multiple Web pages. Each rule group has <RuleGroupTitle> to identify itself and <URL> to represent the location of RIML or RSML which contains the corresponding rule group.

Rule Identification Procedure. The knowledge engineer identifies basic components of rules from Web pages and associates those components in this step. First, the knowledge engineer identifies the scope of a rule. Second, the knowledge engineer identifies IF, THEN, and connectives such as AND, OR, and NOT. Third, the knowledge engineer identifies most fundamental components – symbols like variable, value, and operator. Finally, the symbols are associated by identifying a relevant variable, value, and operator and giving the same id to the variable, value, and operator. It is better to associate IF, THEN, and connectives in the structuring phase because most of these components aren't well structured in Web pages. When the symbols are omitted in the Web page, they will be complemented in the structuring step. While identifying these components, attaching the corresponding RIML tags to the original Web page can be automated by the XRML editor. Table 2 shows the detailed identification steps by comparing the role of the knowledge engineer to the role of the XRML editor.

Combining Multiple Rule Groups in Web Pages into RSML. In this step, the XRML editor merges rule groups from several RIML documents into one RSML document and makes a template for structuring. The XRML editor extracts rule groups which have the same <URL> of RSML. This merging step can be fully automated. Synonym problems for symbols may occur in this step. Accordingly, the knowledge engineer should unify names of symbols with ontology in thesaurus type.

Rule Structuring Procedure. The object of this procedure is to link identified rule components and make complete rules. The RSML template is constructed by extracting rule components from Web pages automatically. The knowledge engineer decides the name of the omitted variable and completes the association of the omitted variable, value, and operator. The associated pairs constitute IF or THEN parts by linking pairs with proper connectives. In addition, the knowledge engineer should consider omitted IF or THEN parts. Finally, the knowledge engineer can make complete rules by linking IF and THEN parts. After the completion of structuring rules, the rule refinement procedure which detects incompleteness or inconsistency of the rule base is required [13, 15, 18]. Table 3 shows the detailed structuring steps by comparing the role of the knowledge engineer to the role of the XRML editor. Figure 4 shows the example of rule acquisition phases from HTML.

Table 2. Rule Identification Procedure

Step	Knowledge Engineer	XRML Editor
A. Initialize RIML	1. Select HTML document which includes implicit rules	2. Assign markup <RIML> and </RIML>
B. Identify RuleGroup and Rule	3. Identify rule group scope which consists of same type of rules	4. Assign markup <RuleGroup> and </RuleGroup> to each scope
	5. Identify rule group title of rule scope	6. Assign markup <RuleGroupTitle> and </RuleGroupTitle>
	7. Identify URL of related Rule Base (RSML)	8. Assign markup <URL> and </URL> to each rule group
	9. Identify rule scope	10. Assign markup <Rule> and </Rule> and assign Rule id
	11. Identify rule title	12. Assign <RuleTitle> and </RuleTitle>
C. Identify IF/THEN and Connectives	13. Identify IF/THEN and Connectives and their scope	14. Assign <IF>, </IF>, <THEN>,</THEN> and corresponding tags of Connectives(<AND>, </AND> or <OR>, </OR> or <NOT>, </NOT>
D. Identify Symbols	15. Identify the words that will be used as variable, value and simple operator	16. Assign <variable>, </variable>, <value>, </value>, <operator>, </operator>, <function>,</function>
	17. Determine appropriate synonym using a thesaurus	18. Assign standard word to synonym with "standard" attribute
E. Associate Symbols	19. Associate <variable>, <value> and <operator> considering omission	20. Assign variable id as <variable#> and assign this id to <value#> and <operator#> and <function#>

3.2 Acquisition of Rule Components

Some components in Web pages can be acquired by different manners. In this section, we propose tags and procedures for acquiring table, operator, and function.

Table. Generally, a table implies a set of rules in the Web page. For example, an online bookstore represents many shipping rate rules with tables. Since a table in the Web has a semi-structured format, we can easily identify and structure the rules in it. Let us describe this procedure. The knowledge engineer identifies the scope and the name of a rule table. Then, the knowledge engineer assigns <RuleTable> and <RuleTableTitle> respectively. A table is divided into a head and contents. Since the head part consists of variables, this is identified with <variable>. Also, the content parts consist of values and they are identified with <value>. Since one row usually becomes one rule, one row is identified with <Rule>. Some columns of a table may

become conditions of rules, and other columns of a table may become conclusions. The knowledge engineer assigns <IF> to the condition columns and <THEN> to the conclusion columns. If the knowledge engineer identifies each part, the XRML editor assigns the tags automatically.

Table 3. Rule Structuring Procedure

Step	Knowledge Engineer	XRML Editor
A. Combine RIMLs into RSML and make RSML template	1. Select RIMLs to be structured	2. Generate RSML template with selected RIMLs and assign <RSML> and </RSML>
		3. Extract <RuleGroup>,</RuleGroup> and <RuleGroupTitle>
		4. Assign <URL>*url of RIML*</URL> to each RuleGroup
		5. Extract <Rule>, </Rule> and <RuleTitle>
		6. Extract <AND>, </AND>, <OR>,</OR>, <NOT>,</NOT>, <IF>,</IF> and <THEN>, </THEN>
		7. Convert <variable>, <value> to the form of <data of variable>data of value</data of variable>
B. Resolve Synonym		8. If variable or value has "standard" attribute, then replace its value to the value of "standard" attribute
C. Complete Omitted Symbols		9. Identify omitted symbols
	10. Determine appropriate name to omitted symbols	11. Complete the form of <data of variable>data of value</data of variable>
D. Associate Connectives	12. Complement omitted connectives	13. Assign <AND>, <OR>, <NOT>
	14. Associate connectives to make IF and THEN parts	15. Assign <IF>, </IF> and <THEN>, </THEN>
E. Associate IF and THEN	16. Complement omitted IF and THEN parts	17. Assign <IF>, </IF> and <THEN>, </THEN>
	18. Associate IF and THEN part	19. Arrange IF and THEN and complete <Rule>

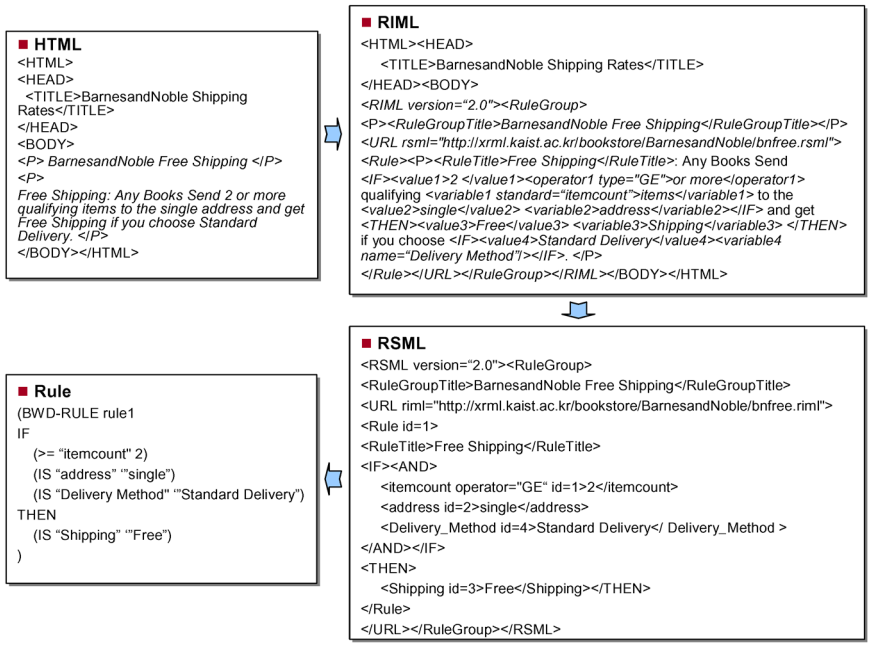

Fig. 4. Example of Rule Acquisition Phases from HTML

Since most components of rule from a table can be acquired in the identification step, its structuring can be automated by the XRML editor. Each <Rule> part is translated into one rule. The <IF> part of each row is translated into the <IF> part in RSML, the <THEN> part to <THEN>. If the <IF> part or <THEN> part is composed of several columns, then it should be connected to the columns with the <AND> connective. The knowledge engineer assigns an "id" to complete each rule. <RuleTable> and <RuleTableTitle> in RIML documents are the same as those in RSML documents. Figure 5 shows an example of table acquisition.

Table

Delivery Method	Ship Time	Total Shipping Price	
		Per Order	Per Item
Standard Delivery	3 to 8 business days	$3.00 per order	$0.99 per item
UPS Select	2 to 5 business days	$3.99 per order	$1.99 per item

RIML

```
<RuleTable>
<Table><tr>
   <IF><td><variable1>Delivery Method</variable1></td></IF>
   <THEN><td><variable2>Ship Time</variable2> Total Shipping
   Price </td>
   <td><variable3>Per Order</variable3></td>
   <td><variable4>Per Item</variable4></td></THEN></tr><tr>
<Rule id=1>
   <IF><td><value1>Standard Delivery</value1></td></IF>
   <THEN><td><value2>3 to 8 business days</value2></td>
      <td><value3>$3.00</value3> per order</td>
   <td><value4>$0.99</value4> per
   item</td></THEN></Rule></tr><tr>
<Rule id=2>
   <IF><td><value1>UPS Select</value1></td></IF>
   <THEN><td><value2>2 to 5 business days</value2></td>
   <td><value3>$3.99</value3> per order</td>
   <td><value4>$1.99</value4> per item</td></THEN>
</Rule></tr></Table></RuleTable>
```

RSML

```
<RuleTable>
<Rule id=1>
<IF>
      <Delivery_Method>Standard Delivery</Delivery_Method></IF>
<THEN>
      <Ship_Time>3 to 8 business days</Ship_Time>
      <Per_Order>$3.00</Per_Order>
      <Per_Item>$0.99</Per_Item></THEN></Rule>
<Rule id=2>
<IF>
      <Delivery_Method>UPS Select</Delivery_Method></IF>
<THEN>
      <Ship_Time>2 to 5 business days</Ship_Time>
      <Per_Order>$3.99</Per_Order>
      <Per_Item>$1.99</Per_Item>
</THEN></Rule></RuleTable>
```

Fig. 5. Example of Table Acquisition

Operator and Function. Operators are represented with natural language in Web pages. So those operators are identified with predefined operator types. For example, "GT" is a type of operator for "Greater Than", "LT", "LE", "GE", and "NOT" are other examples of type attributes. Structuring of operators is accomplished by translating the type attribute of <operator> in RIML to the operator attribute of variable tag in RSML. The numeric expression in Web pages should be translated into a form that can be utilized in the rule base. The knowledge engineer identifies this numeric expression and determines the type attribute of <function>. After that, the knowledge engineer identifies arguments of the function and assigns <arg> tags. The structuring step also can be automated by the XRML editor. Table 4 shows an example of operator and function acquisition.

Table 4. Example of Operator and Function Acquisition

	HTML	RIML	RSML
operator	Products are more than $150	<variable1> *Products* </variable1> are <operator type="GT">more than</operator> <value1>$150</value1>	<*Products* operator="GT"> $150 </*Products* >
function	Total Delivery Time = Available-to-Ship Time + Delivery-Method-Ship-Time	<variable1> Total Delivery Time </variable1> <value1> <function type="+"> <arg1> Available-to-Ship Time </arg1> <arg2> Delivery-Method-Ship-Time </arg2> </function></value1>	<Total_Delivery_Time> <function type="+"> <arg> Available-to-Ship Time </arg> <arg> Delivery-Method-Ship-Time </arg> </function> </Total_Delivery_Time>

4 Mutual Consistency Maintenance with XRML

So far, we have described rule acquisition procedures from Web pages to the rule base by employing XRML. However, after acquiring rules from Web pages and transforming them into the rule base, there is another critical issue on how to maintain consistency between rules in Web pages and rules in the rule base. When the same rules exist in both Web pages and the rule base, the modification of rules in Web pages should be reflected in the rule base and vice versa. However, it is hard to find exact matching symbols between them without additional meta-knowledge and well-defined procedures. As described in section 2.2, the tags of RIML and RSML which are composed of four connection levels of symbol linking between Web pages and the rule base can represent the meta-knowledge for consistency maintenance. By utilizing these tags, let us briefly describe the procedures of consistency maintenance in this section.

The modification of Web pages or the rule base can be categorized by three actions: update, add, and delete. We have developed the XRML editor to perform these procedures with XRML tags. When there is a change in Web pages or the rule base, the knowledge engineer follows the predefined procedures incorporated into the XRML editor.

Figure 6 shows update, add, and delete procedures from RIML to RSML. All of these actions require finding a matching symbol of RSML. A finding procedure consists of four steps as we already mentioned in section 2.2. First, find a matching RSML with <URL>; second, find a matching rule group with <RuleGroupTitle>; third, find a matching rule with an “id” attribute of <Rule>; and finally, find a symbol with the “id” attribute of a variable. The update procedure is simple. If the matching symbol is found, update the matching symbol.

Since adding only one symbol doesn’t make sense, the unit of add procedure is a predicate in the condition part or an action in the conclusion part. The add procedure is more complicated than the update procedure. When the knowledge engineer wants to add a predicate or an action in RIML, the knowledge engineer writes proper text including the predicate or the action and assigns proper tags to that text. Then, the knowledge engineer finds a proper position to add the predicate or the action in RSML. After the knowledge engineer determines an appropriate position to add the new part, he assigns relevant RSML tags for the new part.

In the delete procedure, the unit of deletion is a predicate in the condition part or an action in the conclusion part because it is hard to delete only one symbol. First step of the delete procedure is to find associated symbols of the predicate or action which consists of a variable, value and operator in RIML. The knowledge engineer deletes all of these symbols in RIML and finds matching symbols in RSML. Since these symbols consist of a predicate or an action, the knowledge engineer should check whether there is only one predicate or action in the condition or conclusion part. If so, deletion of that predicate or action causes deletion of the entire rule. If there is another predicate or action in that rule, the knowledge engineer can delete only the matching predicate or action. After deletion of the rule in RSML, the knowledge engineer should reflect this deletion in RIML.

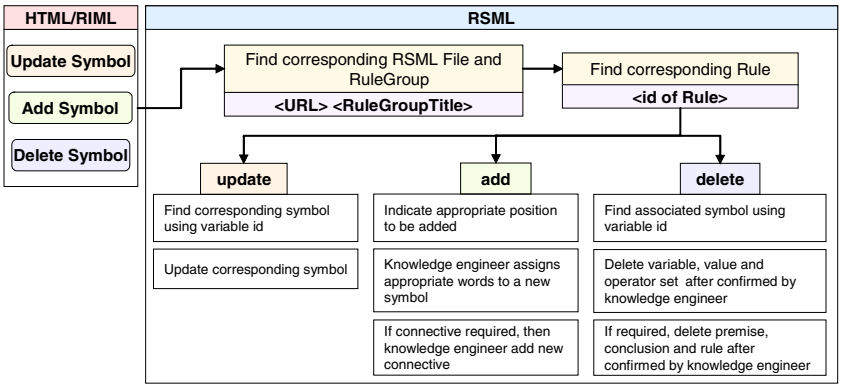

Fig. 6. Consistency Maintenance from RIML to RSML

While the most part of the update procedure can be automated, the add procedure or the delete procedure is hardly automated as described above. Modifying the rule base may induce incompleteness or inconsistency of the rule base. We can adopt rule base refinement methods in the final step of modification [13, 15, 18].

5 Application Using XRML: ComparePrice Including Shipping Cost

In this section, we show the usefulness of our approach with the design and construction of an application we call ComparePrice which applies our rule acquisition methodology.

There are many comparison sites on the Web. These sites compare the prices of several shopping malls for a customer's selected goods, and they provide chances to choose the lowest price based on one fixed delivery cost for all goods. But, delivery costs vary depending on each customer's address, the delivery method, and the category of selected goods. Thus, the price with a single fixed delivery cost is not accurate and comparing these inaccurate prices induces customer to make the wrong choice. To calculate this delivery cost, specific rules related with delivery cost are required. In general, shopping malls post this delivery description on their Web pages. In our application, we acquired delivery rules from these Web pages of each shopping mall and developed a ComparePrice system which provides the comparison of the price including delivery cost based on acquired rules. We focused on online bookstores to narrow the scope.

The objective of ComparePrice is to compare the exact total cost of books in the shopping cart by including the delivery cost in addition to the price of books over multiple online bookstores. To fulfill this objective, we acquire rules to calculate the delivery costs from Web pages of online bookstores. Let us illustrate the rule acquisition procedure. First, we chose three online bookstores – Powells, Barnes&Nobles, and Amazon. From these online bookstores, we found Web pages explaining rules

related with delivery cost. We acquired rules from those Web pages and generated RIML and RSML. After that, we developed a web-based inference system using a rule base translated from RSML.

Consistency between Web pages of online bookstores and the rule base of ComparePrice can be maintained by reflecting changes of Web pages in the rule base using the XRML editor and predefined procedures.

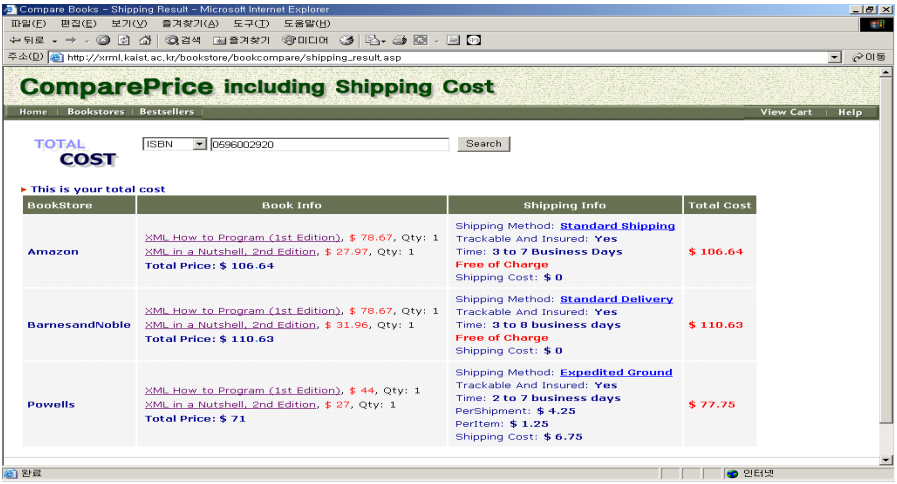

Fig. 7. Example of Comparison Results using the ComparePrice System

The maximum portion of the difference between real delivery cost and estimated average delivery cost of another comparison site was 88% of the book price (delivery cost difference $39.1/total book price $44.28). This means that delivery cost plays an important role in the comparison of a book's price. In our application, the difference of delivery cost between online bookstores averages $40. On the other hand, general comparison sites can't reflect the difference of the delivery cost among those bookstores in price comparison. Figure 7 shows the example of comparison results using our application.

6 Conclusion

In this paper, we have designed XRML version 2.0. In addition, we have presented the stepwise rule acquisition procedure to identify and structure rules implied in multiple Web pages and the consistency maintenance procedure to reflect updates, adds, and deletes of any side of Web pages and the rule base using XRML. We cover the natural text, tables, and implicit numeric functions in Web pages. Also, we developed an application whose name is ComparePrice to show the usefulness of our approach.

We expect that our method can be utilized in various applications of knowledge management system such as tax service and legal service, and electronic commerce

such as price comparison, insurance comparison, loan comparison, and savings comparison. For example, a third party broker can exploit our method to acquire and exchange not only data, but also rules from related Web sites while maintaining consistency. Consequently, they can provide more reliable comparison information to customers based on rules as well as data.

We believe that the semi-automated procedure proposed in this paper can be improved to reduce the effort of the knowledge engineer. For example, ontology can increase automated procedures by providing the information about components of rules and the relationship between them. Also, graphical representation of rules can help the knowledge engineer to structure rules more intuitively. As the further study, we are in the progress of developing a rule acquisition method using ontology.

References

1. Babowal, D., Joerg, W.: From Information to Knowledge: Introducing WebStract's Knowledge Engineering Approach. Proceedings of the 1999 IEEE Canadian Conference on Electrical and Computer Engineering (1999)
2. Chan, K., Low, B.T., Lam, W., Lam, K.P.: Extracting Causation Knowledge from Natural Language Texts. Lecture Notes in Artificial Intelligence, Vol. 2336. Springer-Verlag, Berlin Heidelberg New York (2002) 555-560
3. Cravan, Mark., DiPasco, Dan., McCallum, Andrew., Mitchell, Tom., Nigamm, Kamal., Quek, Choon Yang.: Learning to Construct Knowledge Bases from the World Wide Web. Artificial Intelligence, Vol. 118(1-2) (1999) 69-113
4. Crow, L.R., Shadbolt, N. R.: Extracting focused knowledge from the semantic web. International Journal of Human-Computer Studies, Vol. 54 (2001) 155-184
5. Devedzic, Vladan.: The Semantic Web. Tutorial of PAIS Conference (2001)
6. Fensel, D., Horrocks, I., van Harmelen, F., Decker, S., Erdmann, M., Klein, M.: OIL in a nutshell. Knowledge Acquisition, Modeling, and Management, Proceedings of the European Knowledge Acquisition Conference (2000)
7. Heijst, Van., Wielinga, Schreiber.: Using explicit ontologies in KBS development. International Journal of Human-Computer Studies, Vol. 45 (1997) 183-292
8. Hemnani, A., Bressan, S.: Extracting Information from Semi-Structured Web Documents. Lecture Notes in Computer Science, Vol. 2426. Springer-Verlag, Berlin Heidelberg New York (2002) 166-175
9. Jicheng, W., Yuan, H., Gangshan, W., Fuyan, Z.: Web Mining: Knowledge Discovery on the Web. IEEE SMC '99 Conference Proceedings, Vol. 2 (1999)
10. Lee, J.K., Sohn, M.: Extensible Rule Markup Language – toward Intelligent Web Platform. Communications of the ACM, May, Vol. 46 (2003) 59-64
11. Lee, J.K., Sohn, M.: Extensible Rule Markup Language Version 1.0 specification. http://xrml.kaist.ac.kr (2002)
12. Lee, J.K., Song, Y.U., Kwon, S.B., Kim, W.J., Kim, M.Y.: Rule Syntax of UNIK-BWD. Development of Expert System with UNIK, Bup Young, Ltd (1996) 99
13. Liebowitz, J.: The Handbook of Applied Expert Systems. CRC Press LLC (1998)
14. Moulin, B., Rousseau, D.: Automated Knowledge Acquisition from Regulatory Texts. IEEE Expert (1992)
15. Nguyen, Tin A., Perkins, Walton A.: Knowledge Base Verification. AI Magazine, Vol. 8. No. 2 (1987) 69-75

16. Schmidt, G., Wetter, T.: Using natural language sources in model-based knowledge acquisition. Data & Knowledge Engineering, Vol.26 (1998) 327-356
17. Semantic Web: Semantic Web Introduction, Specifications and Related Works. http://www.w3.org/2001/sw/ (2001)
18. Torsun, I.S.: Foundations of Intelligent Knowledge-Based Systems. Academic Press (1995)

RuleML Annotation for Automatic Collection Levels on METS Metadata

Chieko Nakabasami

Toyo University, 1-1-1 Izumino Itakura Oura Gunma 374-0193 Japan
chiekon@toyonet.toyo.ac.jp

Abstract. In this paper, we propose a rule annotation method using RuleML in the Metadata Encoding and Transmission Standard, called METS. By putting rules into METS, the collection levels of digital contents can be set automatically, and their re-organization can be performed easily. Applying RuleML to the rule description is one of the promising methods for re-organizing digital contents dynamically without human effort. RuleML descriptions are implemented by inserting them into the behavior section of METS, which is prepared for defining behaviors of digital contents. A set of rules embedded into the behavior section may designate how collection levels for target digital contents are set.

1 Introduction

In this paper, we propose a rule annotation method using RuleML[1] in the Metadata Encoding and Transmission Standard, called METS[2], a type of metadata for both the management of digital library objects within a repository and the exchange of such objects between repositories. By putting rules into METS, the collection levels of digital contents can be set automatically, and their re-organization can be performed easily.

In recent years, as next-generation web technologies such as the semantic web and web services have emerged, information sharing and reuse on the web have become indispensable. Many proposals for metadata for various uses are published by such groups as WWW Consortium[3]. Information distributed at different locations should not be arranged based on methods that are understandable only to the owner but should be organized and published according to guidelines for metadata encoding and interface based on common agreement among users.

METS is a metadata encoding standard for digital content developed by the U.S. Library of Congress and also introduced by the *Journal of Artificial Intelligence* in Japan[4]. To archive digital contents, bibliographic records must be collected, stored, and organized, and their contents must be preserved. To this end, publishing must meet users' needs. Digitalized bibliographic contents as well as their originals have potential value from the perspective of preserving contents from physical decay and the possibility for their more flexible use.

Users of digital archives require sophisticated searches for multimedia contents and the sharing of contents among digital libraries. To meet users' needs, collections in which digital contents are grouped according to physical or logical levels are needed. Some collection levels are proposed by the Berkeley Digital Library[5]. Applying

M. Schroeder and G. Wagner (Eds.): RuleML 2003, LNCS 2876, pp. 164–171, 2003.

RuleML to the rule description for automatically generating such collections is one of the promising methods for re-organizing digital contents dynamically without human effort. It is important to enhance common rules because RuleML is a prominent candidate for a standard rule description of the rule section of the seven layers of the semantic web[6].

RuleML descriptions are implemented by inserting them into the behavior section of METS, which is prepared for defining behaviors of digital contents. The Fedra project[7] proposes a method for digital contents management and access to services by designating a target URI of WSDL in the behavior section of Fedra's METS metadata. For related works applying RuleML to web service clients, the WSDF (Web Service Description Framework) has been proposed [8], into which RuleML is embedded for clarifying the semantics of web service methods.

This paper is organized as follows: In Sections 2 and 3, METS is introduced in more detail, and the collection levels of the Berkeley Digital Library[5] are explained. In Section 4, we show a metadata description based on METS in terms of sentence collections of Chinese traditional fixed phrases written in the Japanese language[9], which is part of our ongoing work. We also discuss the realization of methods for setting collection levels to digital objects dynamically using RuleML.

2 METS Metadata

METS is a successor to MOA2[10], a Digital Library Federation (DLF)-funded initiative started in 1997. METS is an XML-based standard for encoding 'hub' documents for materials whose contents are digital. METS has three uses: 1) It is standardized for transmitting and exchanging digital entities; 2) it provides end users with the ability to view and navigate digital content and associated metadata; and 3) it is standardized for archiving digital entities. In a presentation on METS, some example applications are shown[11][12][13], e.g., 'Patrick Breen's Diary of the Cancroft Library, San Francisco 15-Minute Quadrangle, and Kuan-chung pa ching tu of Xian in China. These are digital images by page, and users can browse them via an HTML browser as if they were turning over the pages of a book. METS is to be used as metadata not only for digital library objects but also for digital contents in a wide array of applications for arranging digital contents.

The digital contents of our research target are composed of text and audio data extracted from an ancient Chinese fixed-phrase collection book written in Japanese[9]. These digital contents are considered to be a set of secondary products in the sense that they have been re-organized by us; however, they are thought to be a one-way manifestation of their original contents. These digital contents are not of the type stored as digital image objects for each page but are XML-ized text files and audio files in wav format managed by METS metadata. At present, we are constructing an educational application using these contents and METS. These contents can be applied to a wide variety of web-based applications in the future.

In general, METS metadata consist of the following six sections.

(1) Header

The METS header records administrative metadata about the METS document itself, such as the author (agent) and its role, and the creation and update dates and times of target digital contents.

(2) Descriptive Metadata
METS can record all of the units of metadata pertaining to the digital object represented by the METS document.

(3) Administrative Metadata
METS can record administrative metadata pertinent to the METS object or its four parts (technical, source, rights, and digital provenance).

(4) File Groups
METS records all of the files that together comprise the contents of the digital entity represented by the METS document. Files are organized into file groups based on physical format.

(5) Structural Map
METS specifies the hierarchical structure of the digital entity represented by the METS document and specifies how the content files fit into its structure.

(6) Behavior
METS can record all of the dissemination behaviors pertaining to a digital entity or its parts.

The behavior section may be a good place to describe rules for automatic collection-level setting. In the behavior section, there is a data unit which contains a reference to an external interface definition in terms of a set of related behaviors and a reference to an external executable implementing these behaviors. A set of rules embedded into the behavior section may designate how collection levels for target digital contents are set. The rules are represented in RuleML. The concrete representation is shown in Section 4.

3 Collection Levels for Digital Contents

In the collection levels in a digital contents repository, as noted in [4], a library's intent regarding the breadth and depth of the material it will collect within certain subject areas, genres, or physical formats is publicly declared. Researchers can use the contents collected in one level to determine the relative utility of an entire collection for their purposes as well as to promote cooperative collection development among several libraries. In [5], five collection levels are prepared for print-based materials; in addition, another four levels are proposed for digital contents. Both levels are orthogonal. In our study, a collection is designed in layers similar to those of [5]; however, the digital contents do not move between the levels. They are regarded as a means for interfacing the classification of the digital contents and are used when the contents are published on the web.

In our study, the digital contents can be set for each collection level dynamically by means of adding rules to them. For instance, in sound formats with an 'audio' mimetype, Web applications can recognize that the selected contents can be played by several installed sound players so that the digital sound contents can be selected automatically. In another case, the applications can find that some add-on software

must be prepared to play the target multimedia contents. In addition, we are investigating the classification of the digital content collections into more logical levels. In the example of the [9], the four tones of vowels in the Chinese language are used to classify the contents. Relating the collection levels of digital contents to their semantics would enable the classification of the contents in a more meaningful and flexible manner.

4 RuleML Representation in Behavior Section of METS

In Figure 1, a fragment of the XML representation in the METS metadata description and a sample of the contents in [9] are shown. Adding to the XMLized file, sound files in wav format are provided where sounds in the Chinese language are recorded. In Figure 2, a sample RuleML representation in the behavior section in METS is shown.

```
<?xml version="1.0" encoding="shift_jis" ?>
<METS:mets xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:METS="http://www.loc.gov/METS/"
xmlns:xlink="http://www.w3.org/TR/xlink"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:audioData="http://www.loc.gov/AMD"
xmlns:ruleml="http://www.dfki.de/ruleml"
xsi:schemaLocation="http://www.loc.gov/METS/ http://www.loc.gov/standards/mets/mets.xsd
http://www.loc.gov/AMD http://lcweb2.loc.gov/mets/Schemas/AMD.xsd">
<METS:metsHdr CREATEDATE="2003-05-21" LASTMODDATE="2003-05-21">
<METS:agent ROLE="CREATOR">
<METS:name>Chieko Nakabasami & Lu Tong</METS:name>
</METS:agent>
</METS:metsHdr>
<METS:dmdSec ID="DM1">
<METS:mdWrap MIMETYPE="text" MDTYPE="DC" LABEL="Dublin Core Metadata">
<METS:xmlData>
<dc:title>Oboete Okitai Chuugokugo Xiehouyu 300</dc:title>
<dc:creator>Jun'ichi Kousaka</dc:creator>
<dc:date>1995-04-20</dc:date>
<dc:publisher>Kouseikan</dc:publisher>
<dc:type>text</dc:type>
</METS:xmlData>
</METS:mdWrap>
</METS:dmdSec>
<METS:dmdSec ID="DM2">
<METS:mdWrap MIMETYPE="audio/wav" MDTYPE="DC" LABEL="Dublin Core Metadata">
<METS:xmlData>
<dc:creator>Lu Tong</dc:creator>
<dc:format>audio/wav</dc:format>
</METS:xmlData>
</METS:mdWrap>
</METS:dmdSec>
<METS:amdSec>
```

```
<METS:techMD ID="ADM1">
<METS:mdWrap MDTYPE="LC-AV" LABEL="Audio File Extension Schema">
<METS:xmldata>
<audioData:AUDIOMD ID="AD1">
<audioData:file_data ID="ADF1">
<audioData:format_name>Wave Format</audioData:format_name>
</audioData:file_data>
</audioData:AUDIOMD>
</METS:xmldata>
</METS:mdWrap>
</METS:techMD>
</METS:amdSec>
<METS:fileSec>
<METS:fileGrp VERSDATE="2003-5-26">
<METS:fileGrp ID="TEXT1">
<METS:file ID="XFILE001" MIMETYPE="text/xml">
<METS:FLocate LOCTYPE="URL"
xlink:href="http://teniwoha.itakura.toyo.ac.jp/mets/xiehouyu/xiehouyu001.xml"/>
</METS:file>
</METS:fileGrp>
<METS:fileGrp ID="AUDIO1">
<METS:file ID="FFILE001" MIMETYPE="audio/wav">
<METS:FLocate LOCTYPE="URL"
xlink:href="http://teniwoha.itakura.toyo.ac.jp/mets/sound/f001.wav"/>
</METS:file>
<METS:file ID="BFILE001" MIMETYPE="audio/wav">
<METS:FLocate LOCTYPE="URL"
xlink:href="http://teniwoha.itakura.toyo.ac.jp/mets/sound/b001.wav"/>
</METS:file>
</METS:fileGrp>
<METS:fileGrp ID="TEXT2">
<METS:file ID="XFILE002" MIMETYPE="text/xml">
<METS:FLocate LOCTYPE="URL"
xlink:href="http://teniwoha.itakura.toyo.ac.jp/mets/xiehouyu/xiehouyu002.xml"/>
</METS:file>
</METS:fileGrp>
<METS:fileGrp ID="AUDIO2">
<METS:file ID="FFILE002" MIMETYPE="audio/wav">
<METS:FLocate LOCTYPE="URL"
xlink:href="http://teniwoha.itakura.toyo.ac.jp/mets/sound/f002.wav"/>
</METS:file>
<METS:file ID="BFILE002" MIMETYPE="audio/wav">
<METS:FLocate LOCTYPE="URL"
xlink:href="http://teniwoha.itakura.toyo.ac.jp/mets/sound/b002.wav"/>
</METS:file>
</METS:fileGrp>
</METS:fileGrp>
</METS:fileSec>
<METS:structMap TYPE="logical">
<METS:div ID="div1" LABEL="kitune no shippo">
<METS:div ID="div1.1" LABEL="1 tume no kotoba xml">
<METS:fptr FILEID="XFILE001"/>
</METS:div>
<METS:div ID="div1.2" LABEL="1 tume no mae kotoba">
```

```
<METS:fptr FILEID="FFILE001"/>
</METS:div>
<METS:div ID="div1.3" LABEL="1 tume no usiro kotoba">
<METS:fptr FILEID="BFILE001"/>
</METS:div>
</METS:div>
<METS:div ID="div2" LABEL="kaitei kara hari wo sukuu">
<METS:div ID="div2.1" LABEL="2 tume no kotoba xml">
<METS:fptr FILEID="XFILE002"/>
</METS:div>
<METS:div ID="div2.2" LABEL="2 tume no mae kotoba">
<METS:fptr FILEID="FFILE002"/>
</METS:div>
<METS:div ID="div2.3" LABEL="2 tume no usiro kotoba">
<METS:fptr FILEID="BFILE002"/>
</METS:div>
</METS:div>
</METS:structMap>
<METS:behaviorSec ID="CoMS1.0" BTYPE="bdef-da:1" CREATED="2003-05-31"
LABEL="Collection Management Behaviors">

<!-- RuleML description -->

<METS:interfaceDef LABEL="Collection Management Behavior Definition" LOCTYPE="URN"
xlink:href="bdef-da:1"/>
<METS:mechanism LABEL="Collection Management Behavior Mechanism"
LOCTYPE="URN" xlink:href="bdef-da:2"/>
</METS:behaviorSec>
</METS:mets>
```

Fig. 1. A fragment of the XML representation in the METS on the collections of Chinese traditional fixed phrases

In Figure 2, the first rule says that a digital object in a collection has an audio format if the MIMETYPE[14] of that digital object is of type “audio/wav”.

```
<ruleml:if>
<ruleml:atom>
<ruleml:rel>collection</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>audio</ruleml:ind>
</ruleml:atom>
<ruleml:atom>
<ruleml:rel>isMIMETYPE</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>audio/wav</ruleml:ind>
</ruleml:atom>
</ruleml:if>
<ruleml:if>
<ruleml:atom>
<ruleml:rel>collection</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>photo</ruleml:ind>
```

```
</ruleml:atom>
<ruleml:atom>
<ruleml:rel>isMIMETYPE</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>image/jpg</ruleml:ind>
</ruleml:atom>
</ruleml:if>
<ruleml:if>
<ruleml:atom>
<ruleml:rel>collection</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>photo</ruleml:ind>
</ruleml:atom>
<ruleml:atom>
<ruleml:rel>isMIMETYPE</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>image/gif</ruleml:ind>
</ruleml:atom>
</ruleml:if>
<ruleml:if>
<ruleml:atom>
<ruleml:rel>collection</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>video</ruleml:ind>
</ruleml:atom>
<ruleml:atom>
<ruleml:rel>isMIMETYPE</ruleml:rel>
<ruleml:var>digital_object</ruleml:var>
<ruleml:ind>video/mpeg</ruleml:ind>
</ruleml:atom>
</ruleml:if>
```

Fig. 2. A sample RuleML representation in the behavior section in METS

5 Future Work

This paper discusses a promising methodology for automatically setting the collection level in digital contents using METS metadata. In the future, the digital contents in Chinese fixed phrases should be re-organized suitably for their users, and many bibliographical records in the world of literature should be treated using METS so that they can be managed properly. The effectiveness of RuleML implementation for automatic collection level setting should be validated, and its advantages over systems other than bibliographical contents should be shown.

References

1. The RuleML Markup Initiative. RuleML Homepage. http://www.dfki.uni-kl.de/ruleml/
2. The U.S. Library of Congress. Metadata Encoding and Transmission Standard (METS).http://www.loc.gov/standards/mets/
3. The World Wide Web Consortium. http://www.w3.org/

4. Shigeo Sugimoto and Maria Kuisa Calanag. Archive and Metadata for Digital Contents. *the Journal of Artificial Intelligence in Japan.* Vol.18, No. 3, pp.217-223 (2003)
5. Berkley Digital Library SunSITE. Digital Library SunSITE Collection and Preservation Policy. http://sunsite.berkeley.edu/Admin/collection.html
6. Tim Berners-Lee. Enabling Standards & Technologies - Layer Cake. http://www.w3.org/2002/Talks/04-sweb/Overview.html
7. The University of Virginia and Cornell University. The Fedra Project. http://www.fedora.info/index.shtml
8. Andreas Eberhart. Towards Universal Web Service Clients. EuroWeb 2002 Conference. http://www.i-u.de/schools/eberhart/wsdf/eberhart_euroweb_2002.html
9. Jun'ichi Kousaka and Shi Yi Xin.*Oboete Okitai Chuugokugo Xiehouyu 300 (in Japanese).* Kouseikan (1995)
10. The Making of America II project. http://sunsite.berkeley.edu/moa2/
11. Rick Beaubien. METS: An Introduction. http://www.loc.gov/standards/mets/presentations/METSIntro1.ppt
12. Rick Beaubien. METS: An Introduction Part II. http://www.loc.gov/standards/mets/presentations/METSIntro1.ppt
13. Rick Beaubien. METS: An Introduction - Towards a Digital Object Standard. http://www.loc.gov/standards/mets/presentations/METSUCCSC.ppt
14. Internet Assigned Numbers Authority. MIME Media Types. http://www.iana.org/assignments/media-types/index.html

Author Index

Adi, Asaf 49
Antoniou, Grigoris 111
Anutariya, Chutiporn 35

Baclawski, Kenneth 81
Boley, Harold 1
Bry, François 17

Chatvichienchai, Somchai 35

Etzion, Opher 49

Gilat, Dagan 49

Hatala, Marek 65

Iwiahara, Mizuho 35

Kadarpik, Vello 136
Kambayashi, Yahiko 35
Kang, Juyoung 150
Kifer, Michael 95
Kokar, Mitch M. 81

Lee, Jae Kyu 150
Letkowski, Jerzy 81
Liu, Sandy 121

Matheus, Christopher J. 81

Nakabasami, Chieko 164

Richards, Griff 65

Schaffert, Sebastian 17
Sharon, Guy 49
Spencer, Bruce 121

Tammet, Tanel 136

Wagner, Gerd 111
Wuwongse, Vilas 35

Yang, Guizhen 95