Lecture Notes in Computer Science 3791

Asaf Adi Suzette Stoutenburg
Said Tabet (Eds.)

Rules and Rule Markup Languages for the Semantic Web

First International Conference, RuleML 2005
Galway, Ireland, November 10-12, 2005
Proceedings

Springer

Volume Editors

Asaf Adi
Haifa University, IBM Research Lab in Haifa
Mount Carmel, 31905 Haifa, Israel
E-mail: adi@il.ibm.com

Suzette Stoutenburg
MITRE Corporation
1155 Academy Park Loop, Colorado Springs, CO 80910, USA
E-mail: suzette@mitre.org

Said Tabet
RuleML Initiative
24 Magnolia Road, Natick, MA 01760, USA
E-mail: stabet@ruleml.org

Library of Congress Control Number: 2005935452

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

ISSN	0302-9743
ISBN-10	3-540-29922-X Springer Berlin Heidelberg New York
ISBN-13	978-3-540-29922-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11580072 06/3142 5 4 3 2 1 0

Preface

RuleML 2005 was the first international conference on rules and rule markup languages for the Semantic Web, held in conjunction with the International Semantic Web Conference (ISWC) at Galway, Ireland. With the success of the RuleML workshop series came the need for extended research and applications topics organized in a conference format. RuleML 2005 also accommodated the first Workshop on OWL: Experiences and Directions.

Rules are widely recognized to be a major part of the frontier of the Semantic Web, and critical to the early adoption and applications of knowledge-based techniques in e-business, especially enterprise integration and B2B e-commerce. This includes knowledge representation (KR) theory and algorithms; markup languages based on such KR; engines, translators, and other tools; relationships to standardization efforts; and, not least, applications. Interest and activity in the area of rules for the Semantic Web has grown rapidly over the last five years. The RuleML 2005 Conference was aimed to be this year's premiere scientific conference on the topic. It continued in topic, leadership, and collaboration with the previous series of three highly successful annual international workshops (RuleML 2004, RuleML 2003 and RuleML 2002). The theme for RuleML 2005 was rule languages for reactive and proactive rules, complex event processing, and event-driven rules, to support the emergence of Semantic Web applications.

Special highlights of the RuleML 2005 conference included the keynote address by Sir Tim Berners- Lee, Director of W3C. His talk, titled "Web of Rules", discussed whether knowledge can be represented in a web-like way using rules, so as to derive serendipitous benefit from the unplanned reuse of such knowledge. The two other invited papers were by Dr. Opher Etzion of IBM Haifa Labs entitled "Towards an Event-Driven Architecture: An Infrastructure for Event Processing" and by Dr. Susie Stephens of Oracle USA on the application of Semantic Web technologies in life sciences entitled "Enabling Semantic Web Inferencing with Oracle Technology: Applications in Life Sciences".

We would like to thank our Steering and Program Committees as well as all colleagues who submitted papers to RuleML 2005. We would also like to thank the organizers of ISWC and the OWL Workshop for their cooperation and partnership. Special thanks go to IBM Haifa, Israel, for sponsoring RuleML 2005. Finally, we would like to thank GoWest for their conference planning services.

November 2005

Asaf Adi
Suzette Stoutenburg
Said Tabet

Organization

Conference Program Co-chairs

Asaf Adi, IBM, Israel
Suzette Stoutenburg, The MITRE Corporation, USA
Said Tabet, The RuleML Initiative, USA

Conference General Co-chairs

Harold Boley, National Research Council and University of New Brunswick, Canada
Benjamin Grosof, Massachusetts Institute of Technology, USA

Steering Committee

Asaf Adi, IBM, Israel
Grigoris Antoniou, University of Crete, FORTH, Greece
Harold Boley, National Research Council and University of New Brunswick, Canada
Benjamin Grosof, Massachusetts Institute of Technology, USA
Mike Dean, BBN Technologies, USA
Dieter Fensel, Digital Enterprise Research Institute (DERI),
National University of Ireland, Ireland
Michael Kifer, State University of New York at Stony Brook, USA
Steve Ross-Talbot, Pi4 Technologies, USA
Suzette Stoutenburg, The MITRE Corporation, USA
Said Tabet, The RuleML Initiative, USA
Gerd Wagner, Brandenburg University of Technology at Cottbus, Germany

Program Committee

Grigoris Antoniou, University of Crete, FORTH, Greece
Nick Bassiliades, Aristotle University of Thessaloniki, Greece
Harold Boley, National Research Council and University of New Brunswick, Canada
Mike Dean, BBN Technologies, USA
Dieter Fensel, Digital Enterprise Research Institute (DERI),
National University of Ireland, Ireland
Benjamin Grosof, Massachusetts Institute of Technology, USA
Guido Governatori, University of Queensland, Australia
Michael Kifer, State University of New York at Stony Brook, USA
Sandy Liu, National Research Council and University of New Brunswick, Canada
Jan Maluszynski, Linköping University, Sweden
Massimo Marchiori, W3C, MIT, USA and University of Venice, Italy

Donald Nute, University of Georgia, USA
Royi Ronen, Technion, Israel
Michael Sintek, DFKI, Germany
Bruce Spencer, National Research Council and University of New Brunswick, Canada
Said Tabet, RuleML Initiative, USA
Dmitry Tsarkov, University of Manchester, UK
Carlos Viegas Damasio, Universidade Nova de Lisboa, Portugal
Gerd Wagner, Brandenburg University of Technology at Cottbus, Germany
Kewen Wang, Griffith University, Australia

Sponsors

IBM Haifa Labs

Table of Contents

Towards an Event-Driven Architecture: An Infrastructure for Event Processing Position Paper

Opher Etzion

IBM EDA Initiative Leadership Team
opher@il.ibm.com

Abstract. Multiple business factors have emerged to accelerate the necessity of event-driven functionality and make it part of the main-stream computing, instead of a niche technology. Consequently, there is now focus on using high-level software constructs to build these applications. This paper presents a vision for such high-level features and architecture. This paper explains why "event-driven applications" becomes an emerging area, explains the basic terminology of EDA, explains the relationship to business rules, and sets some directions for the future of this discipline.

1 Introduction and Motivation

Event-driven applications are those which respond to the occurrence of events. This type pf processing is not new, and can be found over the history of computing, starting from exception handling in programming languages, passing through concepts and disciplines such as: active databases [1], publish/subscribe systems [2], network and system management [3] and business activity management [4]. Recently there is an increase in the interest in industry in this area, indicated from analysts' reports, from the sharp increase of start-ups in this area, and product announcements by application middleware and database vendors. This is an indication that event-driven programming moves from being used at some niches to the main stream of programming, and thus it is cost-effective to construct general tools that enable easy construction and maintenance of such applications. The contemporary business drivers for these directions are:

1. Enforcement of compliance with regulations inside the process (some times in "right-time" fashion)
2. The drive for expense reduction in back offices that increase the demand for more automation (e.g. automated exception handling)
3. Increasing complexity of inter process integration that require agility and flexibility;
4. Technology developments such as RFID that increase the scale and scope of event based data
5. Industry trends such as Business Activity Management, Real-Time Enterprise, Business Performance Management that place a demand on software infrastructure to deliver the event data to drive these high level objectives.

The rest of this position paper is structured in the following way. Section 2 explains the type of applications, and shows a case study. Section 3 explains the principles of EDA, Section 4 discusses relation to business rules technology, and Section 5 concludes the paper with some future predictions.

2 Event-Driven Applications

Event-driven functionality is an enabler for the IBM's vision of the "on demand" enterprise, it enables enterprises to make "just in time" reactions to eliminate "worst case" expenses, it enables enterprises to improve control over their operations, and eliminate getting to critical situations, it enables to save cost by providing automation to exception handling, and it is enables loosely coupled integration among processes and systems, improving the agility of application integrations.

Table 1 shows classification of these applications along with the associated business value.

Table 1. Classification of event-driven applications

Type	Examples	Business Value
1. Agile Process Integration	➢ Time Critical Targeting (Military) ➢ EAI Integration hub (Telco) ➢ Just-in-time car rental (Travel and Transportation) ➢ Trade processing (Financial Markets)	Providing integration between various systems based on event input. Enables to support applications that require dynamic composition of various business processes, based on event processing. Enables to perform operations just-in-time and not in advance, thus eliminates excessive cost.
2. Autonomic behavior in business cases	➢ Straight Through Processing (Financial Markets) ➢ Automatic policy setting (Operational Resilience) ➢ Loan and mortgages decision support (Banking) ➢ Automated shipping and receiving based on RFID (Distribution) ➢ London Congestion billing (Travel and Transportation)	Reducing expenses in back offices by automating exception handling processes. Improving business decision process, by linking decisions with business objectives.
3. Awareness to Business Situations	➢ Anti Money Laundering (Banking) ➢ Fraud Detection (multiple industries) ➢ e-Pedigree (Pharmaceutical) ➢ Promotion Evaluation (Retail)	Providing the ability of timely identification of Business situations that requires reaction, and avoid critical situations. This is an enabler for run-time enforcement of regulations.
4. Change Management/ Impact analysis	➢ Design collaboration (PLM / Automotive) ➢ Authorization management (Security) ➢ Compensation management (Insurance)	Provides the impact of a change, allows automatic propagation of system and ensuring consistency throughout systems.

Table 1. Continued

Type	Examples	Business Value
5. Delivery of information services	➢ Information services in mobile devices (Telco) ➢ Stock market information (Financial Markets) ➢ Customer notification system (Banking)	Enables highly personalized information services, extending the capabilities of publish/subscribe systems.
6. Management of services and processes	➢ Information service delivery management (Telco) ➢ Management of Billing and charging (Telco)	Management of service quality agreements and key performance indicators. Event-driven functions are enablers for this type of applications.
7. Proactive systems	➢ Check volume prediction and management (Banking) ➢ Design validity check (PLM/Automotive)	Enabler for proactive behavior in which a system can eliminate possible problems due to predictor's analysis.

Note, that this classification is not a partition; an application can combine application integration, awareness to business situations and automatic behavior in a single application. The idea is to build all the required event-driven functionality in a seamless fashion.

3 Toward an Event-Driven Architecture

Fig. 1 shows an example of event processing within a trade example. Table 2 explains the various event processing artifacts, we call event processing mediations.

Table 2. Event Processing mediations

Event Processing mediations ID	Explanation
EP1	Enrich: Add attributes to the order event (e1) based on a query in a database result in enriched order event (e3)
EP2	Validate: Perform validation test on the enriched order event (e3), may result in an alert event (e4).
EP3	Aggregate: Match the enriched order event (e3) with the appropriate allocation event (e2), creating an allocated order event (e5)
EP4	Validate: Check credit for the allocated order event (e5), possibly revise order to get confirmed order (e6)
EP5	Route: Make a decision to which exchange the confirmed order (e6) should be sent.
EP6	Route: Send time-out alert (e8) for not getting ack. From the exchange (e7) for the confirmed order (e6)
EP7	Compose: Match seller and buyer (both confirmed orders) according to fairness criteria and create a settlement event (e10)
EP8	Route: Retrieve historical order events (e9) as part of the compliance process.

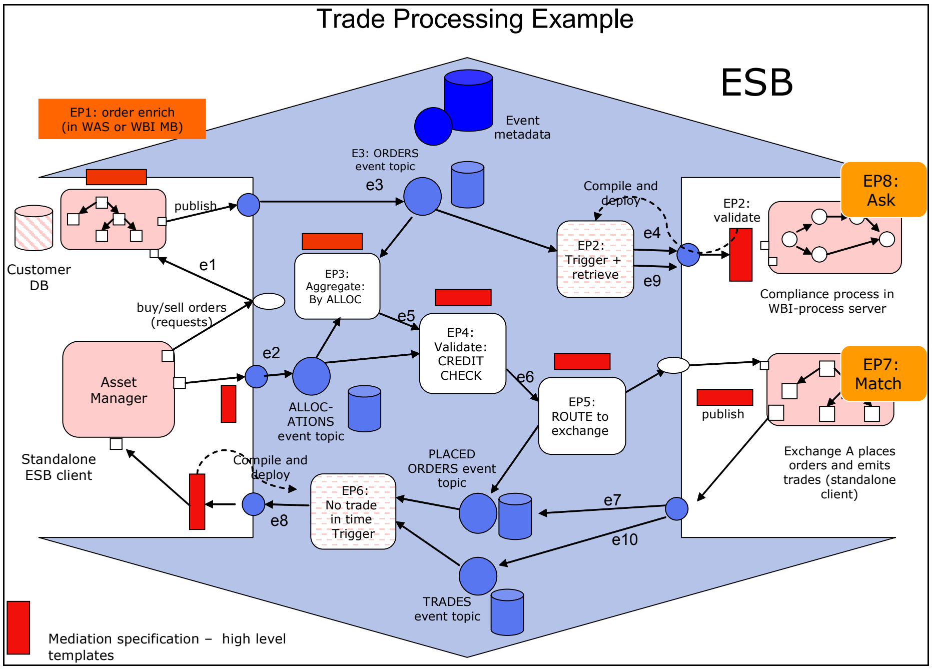


Fig. 1. An example of an event-driven application

The main challenges in setting up this type of applications are:

1. Providing architecture with standard interfaces so that:
   a. Event sources can emit events in various formats easily
   b. Event consumers can obtain events easily
   c. Different event processing components can plug in and interoperate
   d. On-line and historical events can be seamlessly processed
2. A standard event processing language will be used as a basis for all processing, and will enable to make it main-stream computing, independent in proprietary languages.
3. Develop tools that will enable to perform part of it as appropriate for user computing.

Our architecture composed of sources and consumers of events and to the event processing mediations. An event processing mediation is a triplet <selector, processor, emitter>, where:

1. Selector: selects the events that participate in the actual processing, usually by satisfying some pattern on multiple event streams (example: select a pair that consists of buy event and sell event that satisfy a complex pattern).
2. Processor: validate, enrich, compose etc…, creates new event.
3. Emitter: a decision process about the destination of the produced event.

Event processing is a descendent of concepts like “composite events” [5] and “situation detection” [6], but also has some partial overlap with the area of data stream management [7].

4 Relation to Business Rules

Some of the "frequently asked questions" about event processing refers to the relationship between event processing and business rules. This section will revisit the main features of business rules and position event processing in that context.

4.1 Introduction to Business Rules

The name "business rules" refers to a programming style that is intended to take some of the business logic out of the ordinary programming, to achieve more agility.

Barbara Von Hale [8], one of the industry business rules leaders, classifies rules according to the way they are activated. The two rule classes are:

- Service-oriented rules which are activated by explicit request. Business rule products such as: Blaze Advisor, and ILOG's Jrules are examples.
- Data-change-oriented rules which are activated by change in data. Business rule products such as: Versata and USOFT are examples.

Another classification is based on what the business rule does. This classification was done by the Business Rules Group:

- Derivation rules: a statement of knowledge that is derived from other knowledge in the business. This is further classified into:
  - Mathematical calculations
  - Inference of structural assertions
- Action assertion: a statement of a constraint or condition that limits or controls the actions of the enterprise.

4.2 What Is the Relationship Between Event Processing and Business Rules?

Event processing functionality can be expressed by the programming style of business rules.

Event processing can roughly be partitioned to two types of functionality:

- Event derivation: complex events are derived as a function of other events (selection + composition)
- Triggering: actions are triggered by (possibly complex) event.

While these type of processing can be expressed in the rule style, the major difference between event processing and other rule types is that the subject matter are events rather than data or structural assertions.

Table 3 compares traditional business rules and event processing.

Note that regular business rules can also be embedded in event processing, example: routing decisions can be done by decision trees, validation decision can employ rules.

5 Future Directions

The area of event processing now is in its early phases, there are some products, and some applications, but it is still climbing the hill towards prime time. Several developments will accelerate it use:

Table 3. Comparison between event processing and business rules

Dimension Name	Traditional business rules	Event Processing
Rule input	Facts - predicates in first order logic, typically relations among entities (e.g. John is Jim's manager) Data-elements – values of attributes (e.g. John's salary value).	Events – as defined in this document (e.g. John's salary promotion). The input is mostly event, but event processing may consult with facts and data-elements.
Rule output	Knowledge creation: Inference for facts (e.g. if John is Jim's manager, and John is Dan's manager, and peer is defined as two employees reporting to the same manager the system can infer that Jim and Dan are peers) Derivation for data (e.g. the account-balance = the sum of all deposits minus the sum of all withdrawals) Behavioral: Enacts an action (IF-THEN) example: IF there is a traffic jam than re-calculate the route Prevents an action example: if a transaction updates the salary of an employee to be higher than the salary of his manager, then REPAIR THE TRANSACTION to the maximal raise that does not violate the constraint	Complex Event Detection – (e.g. the complex event – at most two bike sells within the last hour) In event processing the created knowledge is in form of new event instances. Behavioral rules where the condition is a complex event. Both type of behavioral systems can be event-driven
Rule invocation	By Request: Rule is activated by specific request or as part of another request (e.g. query). Called: backward chaining (in inference rules, e.g. while answering the question about finding all the peers of Dan), lazy evaluation (in derivation rules, e.g. when calculating the final price). By Trigger: Rule is activated when there is some change in the universe in the rule's scope. Called: forward chaining (in inference rules, when a fact is added), eager evaluation (in derivation rules, when there is a withdrawal from the account).	Event processing functionality can be invoked both by trigger and by request. By trigger is detecting the complex event anytime that the condition is matched. By request --- process events in retrospect .
Rule processing type	Snapshot processing: The rule processes a single snapshot of the universe (regular SQL). Temporal processing: The rule processes a set of changes in the universe done over time (e.g. OLAP tools for data)	Temporal processing is one of the main characteristics of event processing.

1. Standardization: The ability to have standard event schemata, standard APIs, and standard event processing language will make events available and event processing services a major service.
2. Event extraction: There are many sources (such as: news streams, video streams, Blogs, audio files) that include events, but the extraction of these events to form in which events can be processed is very preliminary today. This area will be more mature in a few years.
3. Embedding event driven behavior in programming paradigms: being part of modeling and case tools, introduction into business process management tools.
4. Creation of highly scalable infrastructure that supports high rates of event processing.

References

1. Jennifer Widom, Stefano Ceri: Active Database Systems: Triggers and Rules For Advanced Database Processing. Morgan Kaufmann 1996
2. Roberto Baldoni, Mariangela Contenti, Antonino Virgillito: The Evolution of Publish/Subscribe Communication Systems. Future Directions in Distributed Computing 2003: 137-141
3. T. T. Mai Hoang: Network Management: Basic Notions and Frameworks. The Industrial Information Technology Handbook 2005: 1-15
4. Jun-Jang Jeng, David Flaxer, Shubir Kapoor: RuleBAM: A Rule-Based Framework for Business Activity Management. IEEE SCC 2004: 262-270
5. Shuang Yang, Sharma Chakravarthy: Formal Semantics of Composite Events for Distributed Environments. ICDE 1999: 400-407
6. Asaf Adi, Opher Etzion: Amit - the situation manager. VLDB J. 13(2): 177-203 (2004)
7. Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, Jennifer Widom: STREAM: The Stanford Stream Data Manager. SIGMOD Conference 2003: 665
8. Barbara Von Halle. Business Rules Applied: Building Better Systems Using the Business Rules Approach Wiley, 2002.

Enabling Semantic Web Inferencing with Oracle Technology: Applications in Life Sciences

Susie Stephens

Oracle, 10 van de Graaff Drive, Burlington, MA 01803, USA
susie.stephens@oracle.com

Abstract. The Semantic Web has reached a level of maturity that allows RDF and OWL to be adopted by commercial software vendors. Products that incorporate these standards are being used to help provide solutions to the increasingly complex IT challenges that many industries face. Standardization efforts for the Semantic Web have progressed to the point where efforts are starting in the integration of ontologies and rules. This paper showcases the implementation of a Semantic Web rulebase in Oracle Database 10g, and provides examples of its use within drug discovery and development. A more detailed paper is currently being prepared with Dr. Said Tabet of the RuleML initiative where a more detailed design and specification is provided explaining the

1 Introduction

The current Web is an environment developed primarily for human users. The Semantic Web intends to extend the use of the Web by making documents machine-accessible and machine-readable [3]. A number of standards have been recommended to achieve this goal, including Extensible Markup Language (XML), Resource Description Framework (RDF), and Web Ontology Language (OWL) [6, 20, 21]. As these standards reach maturity, commercial software vendors begin to incorporate the technologies into their products. This trend is illustrated by support for RDF and OWL in the Oracle Database and the integration of RDF metadata in Adobe images.

Having a language for sharing rules is often seen as the next step in promoting data exchange on the Web [2]. Semantic Web rules would allow the integration, transformation and derivation of data from numerous sources in a distributed, scalable, and transparent manner [10]. Rules would themselves be available as data on the Web, and therefore would be available for sharing.

RuleML was the first initiative to bring rules to the Web with support for XML and RDF [5]. It provides a modular lattice of rule sub-languages that have various levels of expressiveness. The Semantic Web Rule Language (SWRL), acknowledged by W3C as a member submission in 2004, is a result of continued efforts in this area [17]. SWRL is layered on existing W3C standards, and integrates OWL and RuleML [4].

The life sciences industry has been taking advantage of rules in drug discovery and development for many years. The use of rules covers areas as broad as the identification of siRNA for gene knockouts [30], the prediction of oral bioavailability of chemical compounds [18], the prediction of human drug metabolism and toxicity of novel compounds [11], decision support rules for patient diagnosis and clinical trial patient selection [26], the determination of the progression of cancer during treatment

[24], and meeting regulatory requirements from organizations such as the U.S. Food and Drug Administration (FDA) and the European Medicines Evaluation Agency (EMEA).

The life sciences industry has been experiencing productivity challenges over the last few years, with the cost of drug discovery soaring, and the number of new drugs to market declining rapidly [14]. As a consequence, the FDA has provided recommendations as to the actions drug discovery companies should take in order to improve product development [12].

Many of the recommendations require the effective sharing of data along the drug discovery and development pipeline. This has proven challenging for organizations to achieve. The difficulties stem from data being made available in many formats, for example, different tab-delimited files formats, XML schemas, and relational models. The task of data integration is made more challenging because the data models change as science progresses, and individuals learn that additional data is relevant to their studies. Further, there is acronym collision across the data sources, and data is provided in many different data types, for example graphs, images, text, and 2D and 3D formats.

The Semantic Web is of considerable interest to the life sciences industry as it promises the ability to overcome data integration challenges [29]. Many of the benefits of the Semantic Web are derived from its very flexible model for data aggregation and re-use [7, 23], along with semantic tagging of data [13, 19, 27]. Further, RDF can be used to integrate both structured and unstructured data. The development of unique identifier representations for both biological and chemical entities is also furthering interest in the Semantic Web [8, 9].

Recommendations from the FDA for improving productivity also include using biomarkers and modeling to determine the safety and efficacy profiles of drugs as early as possible within the drug development process [12]. Many organizations are consequently focusing on providing complex rules and models to help achieve these goals.

Several companies in the life sciences industry have developed proprietary rule-based systems for the identification of drug candidates. In drug discovery and development, the rules for decision support and the information input in the rules reflects the expertise and collective perspectives of senior researchers. Consequently, at present, most decision support tools that address the scientific aspects of drug discovery are the result of custom efforts. It would be highly beneficial if rules could be made available in an interoperable language that takes advantage of the Semantic Web to enable them to be shared across the life sciences community.

This paper discusses the impact of the Semantic Web on the life sciences. We describe the implementation of an RDF Data Model within Oracle Database 10*g* and its use within a drug development scenario.

2 Oracle RDF Data Model

In Oracle Database 10*g* Release 2, a new data model has been developed for storing RDF and OWL data. This functionality builds on the recent Oracle Spatial Network Data Model (NDM), which is the Oracle solution for managing graphs within the Relational Database Management System (RDBMS) [28]. The RDF Data Model supports three types of database objects: model (RDF graph consisting of a set of triples), rulebase (set of rules), and rule index (entailed RDF graph) [1, 22].

2.1 Model

There is one universe for all RDF data stored in the database. All RDF triples are parsed and stored in the system as entries in tables under the MDSYS schema. An RDF triple (subject, predicate, object) is treated as one database object. A single RDF document that contains multiple triples will, therefore, result in many database objects.

The subjects and objects of triples are mapped to nodes in a network, and predicates are mapped to network links that have their start node and end node as subject and object, respectively. The possible node types are blank nodes, Uniform Resource Identifiers (URIs), plain literals, and typed literals. The Oracle Database has a type named URIType that is used to store instances of any URI, and is used to store the names of the nodes and links in the RDF network.

2.2 Rulebase

Each RDF rulebase consists of a set of rules. Each rule is identified by a name, and consists of an 'IF' side pattern for the antecedents, an optional filter condition that further restricts the subgraphs, and a 'THEN' side pattern for the consequents.

A rule when applied to an RDF model may yield additional triples. An RDF model augmented with a rulebase is equivalent to the original set of triples plus the triples inferred by applying the rulebase to the model. Rules in a rulebase may be applied to the rulebase itself to generate additional triples.

Oracle supplies both an RDF rulebase that implements the RDF entailment rules, and an RDF Schema (RDFS) rulebase that implements the RDFS entailment rules. Both rulebases are automatically created when RDF support is added to the database. It is also possible to create a user-defined rulebase for additional specialized inferencing capabilities.

For each rulebase, a system table is created to hold rules in the rulebase, along with a system view of the rulebase. The view is used to insert, delete and modify rules in the rulebase.

2.3 Rules Index

A rules index is an object containing pre-computed triples that can be inferred from applying a specified set of rulebases to a specified set of models. If a graph query refers to any rulebases, a rule index must exist for each rulebase-model combination in the query.

When a rule index is created, a view is also created of the RDF triples associated with the index under the MDSYS schema. This view is visible only to the owner of the rules index and to users with suitable privileges. Information about all rule indexes is maintained in the rule index information view.

2.4 Query

Use of the SDO_RDF_MATCH table function allows a graph query to be embedded in a SQL query. It has the ability to search for an arbitrary pattern against the RDF data, including inferencing, based on RDF, RDFS, and user-defined rules. It can automatically resolve multiple representations of the same point in value space (e.g. “10” ^^xsd:Integer from “10” ^^xsd:PositiveInteger), and has The SDO_RDF_MATCH function has been designed to meet most of the requirements identified by W3C in SPARQL for graph querying [25].

The SDO_RDF_MATCH table function has the following attributes:

```
SDO_RDF_MATCH (
    query  VARCHAR2,
    models  SDO_RDF_MODELS,
    rulebases SDO_RDF_RULEBASES,
    aliases   SDO_RDF_ALIASES,
    filter  VARCHAR2
)  RETURN  ANYDATASET;
```

The ‘query’ attribute is a string literal with one or more triple patterns, usually containing variables. A triple pattern is a triple of atoms enclosed in parentheses. Each atom can be a variable, a qualified name that is expanded based on the default namespace and the value of the alias parameter, or a full URI. In addition, the third atom can be a numeric literal, a plain literal, a language-tagged plain literal, or a typed literal.

The ‘models’ attribute identifies the RDF model or models to use. The ‘rulebases’ attribute identifies one or more rulebases whose rules are to be applied to the query. The ‘models’ and ‘rulebases’ together constitute the RDF data to be queried.

The ‘aliases’ attribute identifies one or more namespaces, in addition to the default namespaces, to be used for expansion of qualified names in the query pattern. The following default namespaces are used by the SDO_RDF_MATCH table function.

(‘rdf’, ‘http://www.w3.org/1999/02/22-rdf-syntax-ns#’)
(‘rdfs’, ‘http://www.w3.org/2000/01/rdf-schema#’)
(‘xsd’, ‘http://www.w3.org/2001/XMLSchema#’)

The ‘filter’ attribute identifies any additional selection criteria. If this attribute is not null, it should be a string in the form of a WHERE clause without the WHERE keyword.

The SDO_RDF_MATCH table function returns an object of type ANYDATASET, with elements that depend upon the input variables.

For each variable *var* that may be a literal (that is, for each variable that appears only in the object position in the query pattern), the result elements have five attributes: *var*, *var*$RDFVTYP, *var*$RDFCLOB, *var*$RDFLTYP, and *var*$RDFLANG. For each variable *var* that cannot take a literal value, the result elements have two attributes: *var* and *var*$RDFVTYP. In both cases, *var* has the lexical value bound to the variable; *var*$RDFVTYP indicates the type of value bound to the variable (URI, LIT[literal], or BLN [blank node]), *var*$RDFCLOB has the lexical value bound to the variable if the value is a long literal, *var*$RDFLTYP indicates the type of literal bound if a literal is bound, and *var*$RDFLANG has the language tag of the bound literal if a literal with language tag is bound. *var*$RDFCLOB is of type CLOB, while all other attributes are of type VARCHAR2.

3 Rule-Based Drug Development Scenario

In clinical trials, information is typically gathered from hundreds of geographically dispersed data sources. In the Semantic Web environment, it would be expected that multiple ontologies would be utilized, thereby requiring support for mapping, integration, and translation. This objective cannot easily be achieved with ontology languages alone [15]. For example, ontology property chaining, dynamic information and complex transactions can be expressed more easily in rule languages [16].

The rule-based drug development scenario demonstrates the use of a rule in the critical task of identifying suitable patients for a clinical trial study. The identification of patients for trials has proven very challenging to the life sciences industry. With the move towards personalized medicine, it is becoming increasingly important that physicians are able to select appropriate patients for trials, in order that effective and safe new drugs can be released.

In this example, male patients over 40 with high-grade prostatic intraepithelial neoplasia (HGPIN) were selected for screening for chemoprevention of prostate cancer. The natural language representation of the rule is shown in Figure 1.

```
IF   Patient is a male over 40
     AND Patient never had Prostate Cancer
     AND Patient condition is HGPIN
THEN       Screen Patient for this Study
```

Fig. 1. Rule Example in Natural Language

Oracle has incorporated an RDF Data Model in Oracle Database 10*g*, which includes support for rules. The SDO_RDF_MATCH table function is invoked from within SQL, and provides support for graph querying. The addition of graph capabilities to SQL querying allows a powerful approach. For example, it enhances the rule language with negation, and takes advantage of the proven scalability of the Oracle platform. The encoding of the rule in Figure 1, is shown below in Figure 2, and the query in Figure 3.

```
Antecedent:
     (?x1 rdf:type :Patient)
     (?x2 :isStudy :SWOG-9917)
     (?x1 :hasSex :Male)
     (?x1 :hasCondition "HGPIN"^^xsd:string)
     (?x1 :hasAge ?age)

Filter:
     age >= 40

Consequent:
     (?x1 :isScreenedForStudy ?x2)
```

Fig. 2. Oracle Rule example

```
  SELECT patient
     FROM TABLE(SDO_RDF_MATCH(
              '(?patient :isScreenedForStudy ?study)
               (?study   :isStudy   :SWOG-9917)',
               SDO_RDF_Models(...), SDO_RDF_Rulebases(...),
               SDO_RDF_Aliases(...), null))
   MINUS
   SELECT patient
    FROM TABLE(SDO_RDF_MATCH(
               '(?patient :hasCondition "Prostate Cancer")',
                SDO_RDF_Models(...), SDO_RDF_Rulebases(...),
                SDO_RDF_Aliases(...), null));
```

Fig. 3. SDO_RDF_MATCH Table Function Query

In the life sciences, it is critical that rules can be exchanged between business partners. SWRL FOL (First Order Logic) RuleML was developed with these requirements in mind. To demonstrate the translation that would be required between the Oracle platform and SWRL FOL RuleML, the rule is implemented in Figure 4.

4 Discussion and Future Work

Rules are recognized as one of the most important components in the development of the Semantic Web and a requirement prior to its deployment and scalability to real-world systems. In this paper, we described the applicability of rules to the life sciences domain.

The life sciences domain is of interest due to the heterogeneous and distributed nature of the data. Information frequently needs to be integrated from many partner organizations, which can be challenging to achieve due to differing data models, missing data, acronym collision, and contradicting ontologies. The Semantic Web promises to help the life sciences industry overcome many of these challenges.

Rules are used for many purposes within the life sciences. Some are made publicly available (e.g. polices and regulation), while others are proprietary and are used to help companies gain and sustain competitive advantage. The published rulebases represent a good candidate for interchange within the Semantic Web. To advance knowledge sharing within the bioinformatics and cheminformatics communities, analytical workflow could also be made available using the interchange standard.

The implementation of an RDF Data Model by Oracle demonstrates the maturity of Semantic Web technologies. The provision of a rulebase and rule index enables pre-computed triples to be inferred from a specified set of rulebases and models. This forward chaining inferencing adds value by minimizing the steps required to identify entities of interest, and to capture complex relationships.

As part of our future work, we are planning to integrate the Oracle RDF inferencing engine with other reasoners, including the Oracle Application Server production rule system. We are also planning to experiment with the interoperability of Oracle technology with SWRL FOL RuleML rulebases. This work is presented in the upcoming paper with Dr. Said Tabet.

```
<ruleml:Assert owlx:name="#Example">
  <owlx:Annotation>
    <owlx:Documentation>
      IF Patient is a male over 40 and Patient never had Prostate
      Cancer and Patient condition is HGPIN THEN Screen Patient
      for this Study
    </owlx:Documentation>
  </owlx:Annotation>
  <ruleml:Forall>
    <ruleml:Var type="Patient">x1</ruleml:Var>
    <ruleml:Var type="Study">x2</ruleml:Var>
    <ruleml:Var type="xsd:int">age</ruleml:Var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:property="isPatient">
        <ruleml:var>x1</ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:property="isStudy">
        <ruleml:var>x2</ruleml:var>
        <ruleml:ind>SWOG-9917</ruleml:ind>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:property="hasSex">
        <ruleml:var>x1</ruleml:var>
        <owlx:Individual owlx:name="#male"/>
      </swrlx:individualPropertyAtom>
      <swrlx:datavaluedPropertyAtom
        swrlx:property="#hasCondition">
        <ruleml:var>x1</ruleml:var>
        <owlx:DataValue
owlx:datatype="http://www.w3.org/2001/XMLSchema#string">HGPIN
        </owlx:DataValue>
        </swrlx:datavaluedPropertyAtom>
        <swrlx:datavaluedPropertyAtom swrlx:property="#hasAge">
          <ruleml:var>x1</ruleml:var>
          <ruleml:var>age</ruleml:var>
        </swrlx:datavaluedPropertyAtom>
        <swrlx:builtinAtom
swrlx:builtin="http://www.w3.org/2003/11/swrlb#greaterThanOrEqual">
          <ruleml:var>age</ruleml:var>
          <owlx:DataValue
owlx:datatype="http://www.w3.org/2001/XMLSchema#int">40
          </owlx:DataValue>
        </swrlx:builtinAtom>
        <swrlx:individualPropertyAtom
    swrlx:property="isScreenedForStudy">
          <ruleml:var>x1</ruleml:var>
          <ruleml:var>x2</ruleml:var>
        </swrlx:individualPropertyAtom>
      </ruleml:And>
    </ruleml:Forall>
</ruleml:Assert>
```

Fig. 4. SWRL FOL RuleML Representation

Acknowledgements

We would like to acknowledge the Oracle Spatial group for the implementation of the RDF Data Model.

References

1. Alexander, N., Lopez, X., Ravada, S., Stephens, S., Wang, J. (2004) RDF Data Model in Oracle. http://lists.w3.org/Archives/Public/public-swls-ws/2004Sep/att-0054/W3C-RDF_Data_Model_in_Oracle.doc
2. Berners-Lee, T. (2005) Infrastructure Roadmap: Stack of Expressive Power. http://www.w3.org/2005/Talks/0517-boit-tbl/#[27].
3. Berners-Lee, T., Hendler, J. Lassila, O. (2001) The Semantic Web. Scientific American. 284, 5: 34-43.
4. Boley, H., Mei, J. (2005) Interpreting SWRL Rules in RDF Graphs, WLFM 2005. http://www.inf.fu-berlin.de/inst/ag-nbi/research/swrlengine/SWRLinRDF.pdf
5. Boley, H., Tabet, S., Wagner, G. (2001) Design Rationale of RuleML: A Markup Language for Semantic Web Rules. SWWS. http://islab.hanyang.ac.kr/bbs/data/cse995/DesignRationaleRuleML.pdf
6. Bray, T., Paoli, J., Sperberg-McQueen, Maler, E., Yergeau, F. (2004) Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation 04 February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/
7. Cheung, K., H., Yip, K., Y., Smith, A., deKnikker, R., Masiar A., Gerstein, M. (2005) YeastHub: a semantic web use case for integrating data in the life sciences domain. Bioinformatics 21, i85-i96.
8. Clark, T., Martin, S., and Liefeld, T. (2004) Globally Distributed Object Identification for Biological Knowledgebases. *Brief. Bioinformatics.* **5** (1), 59–70.
9. Coles, S. J., Day, N. E., Murray-Rust, P., Rzepa, H. S., Zhang, Y. (2005) Enhancement of the Chemial Semantic Web through the use of InCHI Identifiers. Org. Biomol. Chem., 3, 1832-1834.
10. Dean, M. (2004) Semantic Web Rules: Covering the Use Cases. Rules and Rule Markup Languages for the Semantic Web. 3rd Int. Workshop, RuleML 2004, Hiroshima, Japan, Nov. 2004. Antoniou G., Boley, H. (Eds.) pp. 1-5.
11. DeWitte, R. (2003) Product Focused Drug Delivery. Drug Discovery July/Aguust 2003. 30-33.
12. FDA. (2004) Innovation Stagnation. Challenge and Opportunity on the Critical Path to New Medical Products. http://www.fda.gov/oc/initiatives/criticalpath/whitepaper.html
13. Gardner, S. (2005) Ontologies and semantic data integration. Drug Discovery Today 10, 1001-1007.
14. Gilbert J, Henske, P., Singh, A. (2003) Rebuilding Big Pharma's Business Model. In Vivo, the Business & Medicine Report, Windhover Information, Vol. 21, No. 10, November 2003.
15. Golbreich, C. (2004) Combining Rule and Ontology Reasoners for the Semantic Web. Rules and Rule Markup Languages for the Semantic Web. 3rd Int. Workshop, RuleML 2004, Hiroshima, Japan, Nov. 2004. Antoniou G., Boley, H. (Eds.) pp. 6-22.
16. Hawke, S., Tabet, S., de Sainte Marie, C. (2005) Rule Language Standardization. Report from the W3C Workshop on Rule Languages for Interoperability. http://www.w3.org/2004/12/rules-ws/report/
17. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M. (2004) SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission 21 May 2004. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/
18. Lipinski, C.A., Lombardo, F., Dominy, B.W., Feeney, P.J. (1997) Experimental and computational approaches to estimate solubility and permeability in drug discovery and development settings. Adv. Drug Delivery Res. 23, 3-25.
19. Luciano, J. (2005) PAX of mind for pathway researchers. Drug Discovery Today 13, 938-942.

20. Manola, F., Miller, E. (2004) RDF Primer. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-primer/
21. McGuinness, D. L., van Harmelen, F. (2004) OWL Web Ontology Language Overview. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-features/
22. Murray, C. (2005) Oracle Spatial. Resource Description Framework (RDF) 10g Release 2 (10.2). http://download-west.oracle.com/otndocs/tech/semantic_web/pdf/rdfrm.pdf
23. Neumann, E., Miller E., Wilbanks, J. (2004) What the Semantic Web could do for the life sciences. Biosilico 2, 228-236.
24. Padhani, A. R., Ollivier, L. (2001) The RECIST criteria: implications for diagnostics radiologists. Br. J. Radiol. 74, 983-986
25. Prud'hommeaux, E., Seaborne, A. (2005) SPARQL Query Language for RDF. W3C Working Draft 21 July 2005. http://www.w3.org/TR/rdf-sparql-query
26. Salamone, S. (2005) Semantic Web Interest Grows. Bio-IT World (http://www.bio-itworld.com/archive/microscope/document.2005-06-16.8341855754) (2005).
27. Stephens, S., Musen, M. (2005) A novel ontology development environment for the life sciences. BioOntologies Special Interest Group, Intelligent Systems for Molecular Biology. http://bio-ontologies.man.ac.uk/download/stephens_bio_ontologies_submitted.doc
28. Stephens, S., Rung, J. Lopez, X. (2004) Graph Data Representation in Oracle Database 10*g*: Case studies in Life Sciences. IEEE Data Eng. Bull. 27, 61-67.
29. Weitzner, D. (2004) Summary Report – W3C Workshop on Semantic Web for Life Sciences. http://www.w3.org/2004/10/swls-workshop-report.html
30. Yuan, B., Latek, R., Hossbach, M., Tuschl, T., Lewitter, F. (2004) siRNA Selection Server: an automated siRNA oligonucleotide prediction server. Nucleic Acids Res. 32, W130-W134.

A Realistic Architecture for the Semantic Web

Michael Kifer[1], Jos de Bruijn[2], Harold Boley[3], and Dieter Fensel[2]

[1] Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794
kifer@cs.sunysb.edu

[2] Digital Enterprise Research Institute (DERI),
University of Innsbruck, Austria
National University of Ireland, Galway
{jos.debruijn,dieter.fensel}@deri.org
http://www.deri.org/

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

Abstract. In this paper we argue that a realistic architecture for the Semantic Web must be based on multiple independent, but interoperable, stacks of languages. In particular, we argue that there is a very important class of rule-based languages, with over thirty years of history and experience, which cannot be layered on top of OWL and must be included in the Semantic Web architecture alongside with the stack of OWL-based languages. The class of languages we are after includes rules in the Logic Programming style, which support default negation. We briefly survey the logical foundations of these languages and then discuss an interoperability framework in which such languages can co-exist with OWL and its extensions.

1 Introduction

An alternative architecture for the Semantic Web was recently proposed by several groups at the W3C Workshop on Rule Languages for Interoperability[1] and presented in the talk "Web for real people" by Tim Berners-Lee[2]. An older architecture, depicted in Figure 1, assumed that the main languages that comprise the Semantic Web should form a single stack and every new development in that area should build on top of the previous linguistic layers[3]. The older layers at the lower part of the stack are supposed to be upward compatible with the new developments, and in this way any investment made in the old technology will be preserved as the Semantic Web technology matures and expands.

[1] http://www.w3.org/2004/12/rules-ws/
[2] http://www.w3.org/2005/Talks/0511-keynote-tbl/
[3] However, SparQL (http://www.w3.org/TR/rdf-sparql-query/) has recently joined as a language sitting outside of the stack

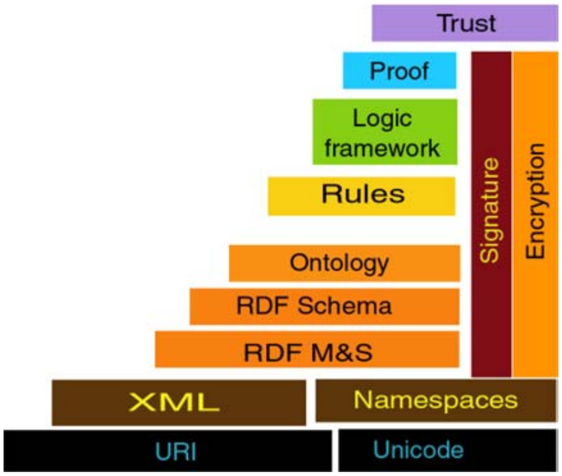


Fig. 1. Old Semantic Web Stack

While a single-stack architecture would hold aesthetic appeal and simplify interoperability, many workers in the field believe that such architecture is *unrealistic* and *unsustainable*. For one thing, it is presumptuous to assume that any technology will preserve its advantages forever and to require that any new development must be compatible with the old. If this were a law in the music industry (for example) then MP3 players would not be replacing compact disks, compact discs would have never displaced tapes, and we might still be using gramophones.

The Semantic Web has had a shorter history than music players, but it already saw its share of technological difficulties. RDF(S), the first layer in the Web architecture that was dubbed “semantic” [LS99], was proposed and standardized without ... a semantics. Years later a group of logicians was seen scrambling to define a formal semantics that would be reasonably compatible with the informal prose that was accompanying the official W3C’s RDF(S) recommendation. The resulting semantics for RDF(S) [Hay04] was certainly formal, but one might question whether the decision to follow the original *informal* RDF semantics (and even syntax) *to the letter* was justified. The difficulties that this created for OWL [DS04]—the subsequent and more expressive layer of the Semantic Web—are well-documented in [HPSvH03]. It is well-known, but rarely mentioned, that OWL-DL is not properly layered on top of RDF(S)[4], and that the single-stack architecture for the budding Semantic Web technology already has a small crack.

The alternative architecture proposed at the workshop recognizes the difficulties (even at the philosophical level) with the single-stack architecture (henceforth called *SSA*). The key idea is that a more realistic architecture must allow multiple technological stacks to exist side-by-side, as in Figure 2. Ideally, adja-

[4] For instance, there are statements that are valid RDF, but not OWL-DL

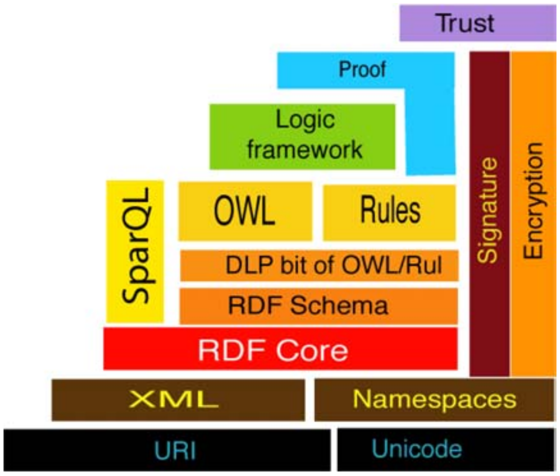


Fig. 2. New Semantic Web Stack

cent stacks should be interoperable to a high degree, but when this is not possible a loosely coupled integration will be acceptable in practice.

The new architecture is more realistic not only because, in the long run, a single stack architecture could saddle us with the Semantic Web equivalent of a gramophone, but also because it would make us use a gramophone to toast bread and mow grass. That is, it is a vain hope that a single upward-compatible language, developed in the Semantic Web's infancy, will suffice for all the future semantic chores to be done on the Web. This expectation is certainly not borne out of the fifty years of experience with programming languages.

To avoid any misinterpretation, it is *not* our intention to claim that the existing parts in the current stack of Semantic Web technologies are obsolete. However, every technology, including language design, eventually becomes obsolete, and no technology can address all problems.

We are therefore convinced that the multi-stack architecture (henceforth called *MSA*) is timely now because limitations of the currently standardized technology are already felt on the Semantic Web—especially in the subarea of rules. The need for rule-based processing on the Semantic Web was envisioned at the very early stage of the development [Bol96,FDES98,MS98,DBSA98], even before the very term "Semantic Web" was coined in, and the rule/ontology combination SHOE was already implemented and used in a web context to reason with annotations of web resources before the rise of XML [HHL03]. Now that OWL has standardized the base level of ontology specification and RuleML [BTW01] has provided a standard serialization layer, rules for the Web have become the focus of intense activity. SWRL [HPSB$^+$04], SWSL-Rules [BBB$^+$05], and WRL [ABdB$^+$05] are some of the languages in this domain that have been proposed recently.

SWRL is a new technology, which extends OWL-DL and permits the use of Description Logic with certain kinds of rules. However, Description Logic is not a technology that comes to mind when one hears about "rules." The use of rules for knowledge representation and intelligent information systems dates back over thirty years. By now it is a mature technology with decades of theoretical development and practical and commercial use. The accumulated experience in this area exceeds the experience gathered with the use of Description Logics and the field is arguably more mature when it comes to rules[5].

What does the SSA vs. MSA discussion have to do with rules? The problem is that the mature technology for rule-based applications mentioned in the previous paragraph is based on *logic programming* [Llo87] and *nonmonotonic reasoning* (*LPNMR*), which is not fully compatible with classical first-order logic (*FOL*) on which OWL and SWRL are built. The aforesaid Web rules proposals, WRL and SWSL-Rules, are based on LPNMR. Few people realize that SQL, arguably the most important rule-based language, has LPNMR as its foundation. Thus, while the OWL-based stack is undoubtedly important and SWRL will find its uses, the vast majority of the rule-based applications cannot be done *in principle* in SWRL or cannot be done *conveniently and efficiently.* This problem gives rise to the second main stack in the MSA diagram in Figure 2.

In the rest of this paper we briefly sketch the ideas underlying LPNMR and their motivation. We then describe interoperability frameworks for the rule-based stack and the OWL-based stack and also address the recent critique of MSA that appeared in [HPPSH05].

2 The Underpinnings of the Rules Stack

In a recent paper [HPPSH05], a number of arguments were laid out to suggest that the layers of the rules stack in Figure 2 are not properly extending each other. In particular, the paper claimed that the DLP layer is not properly extended by the Datalog layer. Unfortunately, it seems that [HPPSH05] mostly argues against a strawman that it itself constructed. To address this criticism, we need to clarify the relationship between the different layers of the rules stack in Figure 2.

A common feature of all the layers in the rules stack is that logical specifications are divided in two categories: *rule sets* and *queries.* A rule set is a set of statements—called *rules*—of the form[6]

$$\texttt{head} \; :- \; \texttt{body} \tag{1}$$

The head is an atomic formula[7] and the body is a conjunction of literals. A *literal* is either an atomic formula or a negation of such a formula. The most common

[5] Some say that there are many more OWL descriptions on the Web than there are rule-based programs, but this argument compares apples and oranges. In which column do we place online databases and SQL applications?

[6] The actual syntax varies. For instance, sometimes rules are written as `body => head`

[7] Typically of the form $\texttt{predicate}(\texttt{arg}_1, ..., \texttt{arg}_N)$, but can also have other forms, if extensions of predicate logic, such as HiLog [CKW93] or F-logic [KLW95] are used

form of negation used in the rule bodies is *default* negation (more on this later). However, extensions that permit weakened forms of classical negation (both in the rule heads and bodies) have been studied [GL91,Gro99]. Finally, we should note that all variables in a rule are assumed to be *universally quantified* outside of the rule.

Various other syntactic extensions of rules exist, which allow disjunctions and even explicit quantifiers in the rule body, and conjunctions in the rule head. However, we will not deal with these extensions here.

A *fact* is a special kind of a rule where the body part is an empty (hence, *true*) conjunction. Often, facts are also considered ground (variable-free), although sometimes universally quantified variables are allowed as well.

A *query* is a statement of the form

$$\exists\,\overline{\mathtt{X}}\,(\mathtt{atomicFormula}) \tag{2}$$

where $\overline{\mathtt{X}}$ is a list of all variables in `atomicFormula`. In general, queries can be much more general. For instance, instead of `atomicFormula`, conjunctions of atomic formulas and of their negation can be allowed. However, such queries can be reduced to the form (2) by introducing rules with `atomicFormula` in the head.

An *answer* to such a query with respect to a rule set **R** is a list of values $\bar{v}$ for the variables in $\overline{\mathtt{X}}$ such that **R** entails (according to the chosen semantics) `atomicFormula`$'$, denoted[8]

$$\mathbf{R} \;\models\!\!\!\approx\; \mathtt{atomicFormula}' \tag{3}$$

where `atomicFormula`$'$ is obtained from `atomicFormula` by consistently replacing each variable in $\overline{\mathtt{X}}$ with the corresponding value in $\bar{v}$.

Thus, in rule-based systems, entailment is limited to inferencing of sets of *facts* only (or their negation, if the language includes negation). This is quite different from first-order logic systems, such as Description Logic and OWL, where more general formulas can be inferred.

We now examine the layers of the rules stack in more detail. We start with *Description Logic Programs* (DLP) [GHVD03] and then clarify their relationship with the RDF layer below and the more expressive layers above.

Description Logic Programs Layer. The *rule-set part* of the DLP layer is a set of all statements in Description Logic that are translatable into Horn rules [GHVD03]. A *Horn rule* is a rule of the form (1) where the head and the body consist of only atomic formulas (no negation of any kind). For Horn rules, the entailment used in (3) is classical first-order.

The *query part* of DLP is of the form (2) above. Thus, even though DLP is a subset of Description Logic, the only entailments that are considered in the DLP layer are inferences of atomic formulas. (Note that [GHVD03] also defined

[8] We use $\models\!\!\!\approx$ instead of $\models$ to emphasize that the entailment relation used in rule languages is typically nonmonotonic and, therefore, non-classical

DHL—*Description Horn Logic*—which is like DLP, but arbitrary entailments are allowed from the rules.)

Since DLP is translated into Horn rules, the entailment in (3) is the *classical entailment* in first-order logic and, therefore, the *semantics of DLP in the OWL stack and in the rules stack are the same.*

RDF Layer. In the architecture diagrams, DLP (and the rest of OWL) is depicted as sitting on top of the RDF layer. This statement requires clarifications. From the logical point of view, RDF graphs are largely just sets of facts. However, RDF also includes two additional interesting features. The first feature, *reification*, cannot be modeled in DLP or even in more general description logics. In that sense, neither DLP nor OWL-DL truly reside on top of RDF. Second, RDF has so-called *blank nodes*. RDF graphs that contain blank nodes logically correspond to sets of atomic formulas that include existentially quantified variables. This is not a problem for description logics in general, but (at a first glance) seems to be a problem for DLP, since the latter does not allow existential variables in rule heads (and thus neither in facts).

However, it turns out that extending Horn rules to accommodate existentially quantified facts is not difficult as long as the queries are still of the form (2) above. Indeed, if $\mathbf{R}$ is a set of Horn rules plus facts, where some facts are existentially quantified, then the entailment (3) holds (where $\models\!\!\approx$ should be understood as classical first-order logic entailment) if and only if $\mathbf{R}' \models \mathtt{atomicFormula}'$, where $\mathbf{R}'$ is a skolemization of $\mathbf{R}$ (i.e., is obtained from $\mathbf{R}$ by consistently replacing the occurrences of existential variables with new constants).

Thus, skolemization appears to be the right way to deal with blank nodes and with embedding RDF in DLP, and this is how various implementations of the N3 rules language for RDF [BL04] treat blank nodes, anyway.

Reification can also be added to DLP (and to all the layers above it in the rules stack) along the lines of [YK03]. Therefore, an extension of DLP can be said to reside on top of RDF.

Datalog Layer. Datalog [MW88] is a subset of Horn logic that does not use function symbols. Since DLP is a subset of description logic, it does not use function symbols either and, therefore, the translation of DLP into rules yields a subset of Datalog.

Strictly speaking, Datalog cannot be said to reside on top of DLP, since the latter uses the syntax of description logics, which is different from the syntax of rules (1). However, Datalog certainly resides on top of the image of DLP under the translation described in [GHVD03]. Therefore, modulo such a translation, Datalog can be said to extend DLP. Later on, we will define this notion precisely.

Default Negation. Default negation is an inference rule associated with a negation operator, `not`, that derives new information based on the inability to derive some other information. More precisely, `not q` may be derived because `q` cannot be. This type of negation has been a distinguishing feature of rule-based languages for more than thirty years. With such an inference rule, given a rule-base with the single rule `p :- not q`, we can derive `p` because `not q` can be derived by default (since `q` is not derivable).

One of the main reasons for the emergence of default negation is that it is impractical, and often impossible, to write down all the negative facts that might be needed in a knowledge base in order to take advantage of the classical negation. It is a common practice in knowledge representation to specify only positive true facts and leave derivation of the negative facts to the default negation rule. Default negation is also often associated with common sense reasoning used by humans who tend to conclude non-existence of something because existence is not positively known.

Default negation is sometimes also referred to as *negation as failure*. This terminology is unfortunate, since negation as failure is the traditional name for a specific form of default negation [Cla78]—one that is used in the Prolog language. Negation as failure (as used in Prolog) is known to be problematic [ABW88] and modern logic programming languages use either the *well-founded* default negation [GRS91] or the one based on *stable models* [GL88].

It is well-known that the default negation layer is a semantic and syntactic extension of the Datalog layer in the sense defined below.

Default negation is not the only nonmonotonic inference rule that we deem to be important on the rules stack of MSA. A related inference rule, called *default inheritance*, is used in object-oriented knowledge representation. F-logic [KLW95] offers a comprehensive logical framework, which supports default inheritance, and this inference rule is implemented in most F-logic based systems, such as FLORA-2 [YK02,YKZ03,Kif05] and Ontobroker [Ont].

Constraints. Support for database-style constraints is another important feature of knowledge representation on the rules stack.

Logic languages that are based on pure first-order logic, like OWL, do not support constraints and have no notion of *violation* of a constraint. Instead, they provide *restrictions*, which are statements about the desired state of the world. Unlike constraints, restrictions may produce new inferences. For instance, if a person is said to have at most one spouse and the knowledge base records that John has two, Mary and Ann, then OWL would conclude that Mary and Ann is the same person. In contrast, a rulebase with nonmonotonic semantics will view such a knowledge base as inconsistent.

The semantics of database constraints is closely related to nonmonotonic reasoning, since it relies on the notion of canonical models—a subset of models that are considered to be "correct"—and focusing on canonical models is a standard way of defining the semantics for default negation [Sho87]. In contrast, pure first-order logic based semantics considers *all* models of a theory. Therefore, database constraints belong on the rules stack of MSA.

Additional Layers. The rules stack can be further extended with additional layers of which the more interesting ones include *classical negation*, *prioritized rules*, *object-orientation*, and *higher-order syntax*.

Extensions that permit classical negation alongside default negation have been proposed in [GL91,Gro99] and were implemented in a number of systems. Rule prioritization is part of Courteous Logic Programming [Gro99], and is supported by the Sweet Rules system[9]. Object-oriented extensions inspired by F-logic [KLW95] and HiLog higher-order extensions [CKW93] are part of the FLORA-2 system [YKZ03]. In fact, SWSL-Rules — a language that incorporates all of these layers have also been recently proposed [BBB$^+$05].

3 Scoped Inference

In (2), logical entailment happens with respect to an explicitly specified knowledge base, **R**. The assumption that the underlying knowledge base is known is a cornerstone of traditional knowledge representation. The Semantic Web challenges this assumption, since the boundaries of the Web cannot be clearly delineated. Therefore, the notion of inference on the Semantic Web needs to be revisited.

One idea that is beginning to take hold is the notion of *scoped inference*. The idea is that derivation of any literal, `q`, must be performed within the scope of an explicitly specified knowledge base. Different scopes can be used for different inferences, but the scope must always be declared. Scoped inference is an important feature of several knowledge representation systems for the Web. In FLORA-2 [YKZ03], the entire knowledge base is split into *modules* and inference is always made with respect to a particular module. In TRIPLE [SD02], the same idea goes under the name of a *context*.

Scoped inference can be realized using the notion of modules, as in FLORA-2 and TRIPLE, where the definition of a scope can be based on URIs, which dereference to concrete knowledge bases.

Related to the notion of scoped inference is an extension of the concept of default negation, called *scoped default negation*[10]. The idea is that the default negation inference rule must also be performed within the scope of an explicitly specified knowledge base. That is, `not q` is said to be true with respect to a knowledge base **K** if `q` is not derivable from **K**. A version of this rule is supported by some systems, such as FLORA-2, and is discussed in [Kif05].

While scoped inference is clearly useful even for deriving positive information, scope is imperative for deriving negative information from knowledge published on the Web. In fact, due to the open nature of the Web, it is not even meaningful to talk about the *inability* to derive something from a knowledge base whose bounds and the exact content are not known. On the other hand, with explicit scope, default negation becomes not only a meaningful derivation rule on the Web, but also as useful as in traditional knowledge bases.

4 The Relationship Between Layers

The layers of the rules stack are progressive syntactic and semantic extensions of each other (modulo the aforesaid caveats pertaining the RDF layer). Formally,

[9] http://sweetrules.projects.semwebcentral.org/

[10] This concept sometimes goes under the name *scoped negation as failure* or *SNAF*, which is unfortunate terminology for the reasons stated earlier

this means that each layer is a syntactic and semantic extension of the previous layer, as defined next.

Language Extensions. Let $L_1 \subseteq L_2$ be two logic languages and suppose their semantics are defined using the entailment relations $\models_1$ and $\models_2$. L_2 is said to be an *extension* of L_1 if for any pair of formulas $\phi, \psi \in L_1$, the entailment $\phi \models_1 \psi$ holds iff $\phi \models_2 \psi$ holds.

In case of a rules language, the set of formulas that can be used as premises is not the same as the formulas that can be used as consequents. Therefore, we should assume that $L_1 = Premises_1 \cup Consequents_1$ and $L_2 = Premises_2 \cup Consequents_2$. In addition, as in the case of DLP and Datalog, L_1 may not actually be a subset of L_2. Instead, it may be embedded in L_2 under a 1-1 transformation, ι. In our notation, this is expressed as $\iota(Premises_1) \subseteq Premises_2$ and $\iota(Consequents_1) \subseteq Consequents_2$.

We can now say that L_2 *extends* L_1 *under the embedding* ι if for every pair of formulas, $\phi \in Premises_1$ and $\psi \in Consequents_1$, the entailment $\phi \models_1 \psi$ holds iff $\iota(\phi) \models_2 \iota(\psi)$ holds.

With these definitions, we can now formally state (relying on the standard facts about Datalog and default negation) that Datalog extends DLP with respect to the DLP-to-Datalog embedding described in [GHVD03]. The default negation layer similarly extends Datalog with respect to the identity embedding.

Interoperability Through Language Extension. With a proper definition of language extensions, we can now address a recent criticism of the layered structure of the rules stack. It is claimed in [HPPSH05] that it is incorrect to say that Datalog is an extension of the DLP layer because, given a single fact, such as `knows(pat,jo)`, DLP and Datalog give different answers to the question of whether `pat` knows exactly one person.

The answer to this apparent paradox (relatively to our earlier discussion) is simple: the above question cannot be formulated in either DLP or Datalog! In the OWL stack, this query requires a more expressive description logic and on the rules side it requires default negation. Therefore, *as stated*, the above argument falls flat on its face. However, a restatement of this argument is worth debating:

> *Given a set of RDF facts and two "similar" queries—one expressed in the rules stack and the other in the OWL stack—does it matter that the two queries might return different answers?*

The word *similar* is in quotes because it is unclear whether—outside of Datalog—an acceptable systematic mapping exists to map OWL queries into rules, or vice versa. For instance, the aforesaid question about `pat` knowing exactly one person requires radically different expressions in OWL and in the default negation layer. Nevertheless, *intuitively* these two queries can be viewed as similar. Under the OWL semantics the answer will be "unknown" since it is not possible to either prove or disprove that `pat` knows exactly one person; under the rules semantics the answer will be a "yes." We argue, however, that both answers are right! A user who chooses to write an application using the rules stack does so because of a desire to use the language and semantics of that stack. Otherwise, a competent user should choose OWL and SWRL.

5 Interoperability Between Rules and OWL

It has often been observed that DLP, the *intersection* of Description Logic and Logic Programming, is rather minimalistic—a good reference point perhaps, but too small for realistic knowledge representation in the Semantic Web. On the other hand, the *union* of various classes of Description Logic and Logic Programming leads to supersets of first-order logic with default negation, which is not easily formalized model-theoretically and computationally. To achieve a usable level of interoperability between the two paradigms of knowledge representation, we need a "logical framework" that will be sitting *above* the OWL and rules stack and will enable inferences performed by OWL to be used by the rules stack, and vice versa.

As discussed in previous sections, OWL-based ontologies and rules are best viewed as complementary stacks in a *hybrid* Semantic Web architecture. Our interoperability framework derives from these observations. When we say "rules" here, we mean rule bases with nonmonotonic semantics. Pure first-order rules, as in SWRL, belong to the OWL stack, and we will include them under the rubric of "OWL-based ontologies."

The basic idea is that rules and OWL will view each other as "black boxes" with well-defined interfaces defined through *exported predicates*. OWL-based ontologies will export some of their classes and properties, while rule-based knowledge bases will export some of the predicates that they define. Each type of the knowledge base will be able to refer to the predicates defined in the other knowledge bases and treat them *extensionally*, as collections of facts.

One of the earliest integration frameworks in this spirit was AL-log [DLNS98]. AL-Log is a uni-directional approach where rules can refer to description logic based ontologies, but not vice versa. This approach is appropriate when OWL-based ontologies are used to classify objects into classes (analogously to database schema), and rules supply additional inferences.

Bi-directional integration is more powerful, but the semantics of an integrated knowledge base may not be clear if rules refer to ontologies and ontologies refer back to rules within the same knowledge base in a recursive manner. One example when such a semantics can be defined under certain restrictions was given in [ELST04]. However, we believe that recursive references across the rules/OWL boundary are unlikely, and this semantic complication will not arise (and can probably be disallowed in a practical language).

In sum, the hybrid architecture offers a way to combine expressive classes of nonmonotonic rulebases with OWL-based ontologies. The two kinds of knowledge bases can use inferences produced by each other or they can be used in a standalone mode. It is not hard to see that the interoperability framework discussed in this section can be implemented on top of the infrastructure provided by modules (or contexts) used in systems like FLORA-2 and TRIPLE, which was introduced in Section 3—the same infrastructure that can be used to solve the problem of scoped inference.

6 Conclusion

In this paper we provided an in-depth discussion of the multi-stack architecture (MSA) for the Semantic Web and argued that a stack of rule-based languages, complete with default negation, should exist side-by-side with the ontology stack based on OWL. We surveyed the theoretical underpinning of the rules stack and proposed a framework for interoperability between rules and ontologies. We also discussed the idea of scoped inference and highlighted its importance in the Web environment. We observed that both scoped inference and the interoperability framework can be implemented using the idea of modules.

We would like to further remark that the proposed multi-stack architecture is extensible and additional stacks can be added to it as long as they can interoperate according to the guidelines of Section 5. One candidate for such an additional stack is the SparQL language[11].

Acknowledgement

Michael Kifer was supported in part by NSF grant CCR-0311512 and by U.S. Army Medical Research Institute under a subcontract through Brookhaven National Lab. Jos de Bruijn was supported by the European Commission under the projects SEKT and ASG. Harold Boley was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). Dieter Fensel was supported in part by the Science Foundation Ireland (SFI).

References

[ABdB$^+$05] Jürgen Angele, Harold Boley, Jos de Bruijn, Dieter Fensel, Pascal Hitzler, Michael Kifer, Reto Krummenacher, Holger Lausen, Axel Polleres, and Rudi Studer. Web rule language (wrl), Juny 2005. Technical Report.

[ABW88] K. R. Apt, H. A. Blair, and A. Walker. *Foundations of Deductive Databases and Logic Programming*, chapter Towards a theory of declarative knowledge, pages 89–148. Morgan Kaufmann Publishers, Los Altos, CA, 1988.

[BBB$^+$05] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and S. Tabet. Swsl-rules: A rule language for the semantic web, July 2005. Technical Report.

[BL04] T. Berners-Lee. Primer: Getting into RDF & Semantic Web using N3, 2004. http://www.w3.org/2000/10/swap/Primer.html.

[11] http://www.w3.org/TR/rdf-sparql-query/

[Bol96] Harold Boley. Knowledge Bases in the World Wide Web: A Challenge for Logic Programming. In Paul Tarau, Andrew Davison, Koen De Bosschere, and Manuel Hermenegildo, editors, *Proc. JICSLP'96 Post-Conference Workshop on Logic Programming Tools for INTERNET Applications*, pages 139–147. COMPULOG-NET, Bonn, Sept. 1996. Revised versions in: International Workshop "Intelligent Information Integration", KI-97, Freiburg, Sept. 1997; DFKI Technical Memo TM-96-02, Oct. 1997.

[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.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. 15(3):187–230, February 1993.

[Cla78] K. L. Clark. *Logic and Data Bases*, chapter Negation as Failure, pages 293–322. Plenum Press, NY, USA, 1978.

[DBSA98] Stefan Decker, Dan Brickley, Janne Saarela, and Jürgen Angele. A query and inference service for RDF. In *QL'98 - The Query Languages Workshop, W3C Workshop*, 1998.

[DLNS98] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. AL-log: Integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.

[DS04] Mike Dean and Guus Schreiber, editors. *OWL Web Ontology Language Reference.* 2004. W3C Recommendation 10 February 2004.

[ELST04] Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. In *Proc. of the International Conference of Knowledge Representation and Reasoning (KR'04)*, 2004.

[FDES98] Dieter Fensel, Stefan Decker, Michael Erdmann, and Rudi Studer. Ontobroker: The very high idea. In *Proceedings of the 11th International FLAIRS Conference (FLAIRS-98)*, pages 131–135, Sanibel Island, Florida, USA, 1998.

[GHVD03] B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *12th International Conference on the World Wide Web (WWW-2003)*, May 2003.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[Gro99] B.N. Grosof. A courteous compiler from generalized courteous logic programs to ordinary logic programs. Technical Report RC 21472, IBM, July 1999.

[GRS91] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Hay04] Patrick Hayes. RDF semantics. Technical report, W3C, 2004. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-mt/.

[HHL03] J. Heflin, J. Hendler, and S Luke. *SHOE: A Blueprint for the Semantic Web.* MIT Press, Cambridge, MA, 2003.

[HPPSH05] Ian Horrocks, Bijan Parsia, Peter Patel-Schneider, and James Hendler. Semantic web architecture: Stack or two towers? In *Third Workshop on Principles and Practice of Semantic Web Reasoning*, Dagstuhl, Germany, September 2005.

[HPSB$^+$04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. Member submission 21 may 2004, W3C, 2004. Available from http://www.w3.org/Submission/SWRL//.

[HPSvH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[Kif05] M. Kifer. Nonmonotonic reasoning in FLORA-2. In *2005 Intl. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, Diamante, Italy, September 2005.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, July 1995.

[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.

[MS98] Massimo Marchiori and Janne Saarela. Query + metadata + logic = metalog. In *QL'98 - The Query Languages Workshop, W3C Workshop*, 1998.

[MW88] D. Maier and D.S. Warren. *Computing with Logic: Logic Programming with Prolog.* Benjamin-Cummings, Menlo Park, CA, 1988.

[Ont] Ontoprise, GmbH. Ontobroker. http://www.ontoprise.com/.

[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.

[Sho87] Y. Shoham. Nonmonotonic logics: meaning and utility. In *Proc. 10th International Joint Conference on Artificial Intelligence*, pages 388–393. Morgan Kaufmann, 1987.

[YK02] G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE)*, October 2002.

[YK03] Guizhen Yang and Michael Kifer. Reasoning about Anonymous Resources and Meta Statements on the Semantic Web. In Stefano Spaccapietra, Salvatore T. March, and Karl Aberer, editors, *J. Data Semantics I*, volume 2800 of *Lecture Notes in Computer Science*, pages 69–97. Springer, 2003.

[YKZ03] G. Yang, M. Kifer, and C. Zhao. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003)*, November 2003. The system is available at http://flora.sourceforge.net.

Active Rules in the Semantic Web: Dealing with Language Heterogeneity

Wolfgang May[1], José Júlio Alferes[2], and Ricardo Amador[2]

[1] Institut für Informatik, Universität Göttingen
`may@informatik.uni-goettingen.de`
[2] Centro de Inteligência Artificial – CENTRIA, Universidade Nova de Lisboa
`{jja,ra}@di.fct.unl.pt`

Abstract. In the same way as the "static" Semantic Web deals with data model and language heterogeneity and semantics that lead to RDF and OWL, there is language heterogeneity and the need for a semantical account concerning Web dynamics. Thus, generic rule markup has to bridge these discrepancies, i.e., allow for *composition* of component languages, retaining their distinguished semantics and making them accessible e.g. for reasoning about rules.
In this paper we analyze the basic concepts for a general language for evolution and reactivity in the Semantic Web. We propose an ontology based on the paradigm of Event-Condition-Action (ECA) rules including an XML markup. In this framework, different languages for events (including languages for composite events), conditions (queries and tests) and actions (including complex actions) can be composed to define high-level rules for describing behavior in the Semantic Web.

1 Introduction

The goal of the *Semantic Web* is to bridge the heterogeneity of data formats, schemas, languages, and ontologies used in the Web to provide semantics-enabled unified view(s) on the Web, as an extension to today's *portals*. In this scenario, XML (as a format for storing and exchanging data), RDF (as an abstract data model for states), OWL (as an additional framework for state theories), and XML-based communication (Web Services, SOAP, WSDL) provide the natural underlying concepts. The *Semantic Web* should not only be able to support querying, but also to propagate knowledge and changes in a semantic way. This *evolution* and *behavior* depends on the cooperation of nodes. Here, also the heterogeneity of concepts for expressing behavior requires an appropriate handling on the semantic level. Since the contributing nodes are based on different concepts such as data models and languages, it is important that *frameworks* for the Semantic Web are modular, and that the *concepts* and the actual *languages* are independent. While for a data model and for querying, a "common" agreed standard evolves with RDF/RDFS, OWL and languages like RDF-QL etc., the concepts for describing and implementing behavior are much more different, due to different needs, and it is –in our opinion– unlikely that there will be a unique language for this throughout the Web.

Here, *reactivity* and its formalization as *Event-Condition-Action (ECA) rules* offer a suitable common model because they provide a modularization into clean concepts with a well-defined information flow. An important advantage of them is that the *content* of a rule (event, condition, and action specifications) is separated from the *generic semantics* of the ECA rules themselves that provides a well-understood formal meaning: when an event (atomic event or composite event) occurs, evaluate a condition (possibly after querying for some extra data), and if the condition is satisfied then execute an action (or a sequence of actions, a program, a transaction). ECA rules provide a generic uniform framework for specifying and implementing communication, local evolution, policies and strategies, and –altogether– global evolution in the Semantic Web.

In the present paper, we describe an ontology-based approach for specifying (reactive) behavior in the Web and evolution of the Web that follows the ECA paradigm. We propose a modular framework for *composing* languages for events, queries, conditions, and actions by separating the ECA semantics from the underlying semantics of events, queries, and actions. This modularity allows for high flexibility wrt. these sublanguages, while exploiting and supporting their meta-level *homogeneity* on the way to the Semantic Web.

Moreover, the ECA rules do not only operate on the Semantic Web, but are themselves also part of it. In general, especially if one wants to reason about evolution, ECA rules (and their components) must be communicated between different nodes, and may themselves be subject to being updated. For that, the ECA rules themselves must be represented as data in the (Semantic) Web. This need calls for an ontology and a (XML) Markup Language of ECA Rules. A markup proposal for active rules can be found already in RuleML [RML], but it does not tackle the complexity and language heterogeneity of events, actions, and the generality of rules, as described here.

Related Work – Concepts that Have to Be Covered and Integrated by this Approach. The importance of being able to update the Web has long been acknowledged, and several language proposals exist (e.g. XUpdate [XML00] and an extension to XQuery in [TIHW01]) for just that. More recently some reactive languages have been proposed that are also capable of dealing-with/reacting-to some forms of events, evaluate conditions, and upon that act by updating data. These are e.g. XML active rules in [BCP01,BBCC02], an ECA language for XML [BPW02], and RDFTL [PPW04], which is an ECA language on RDF data. These languages do not provide for more complex events, and they do not deal with heterogeneity at the level of the language. Relating to our approach, these rules will be used on a low abstraction level, dealing with data level events and actions. Active XML [ABM$^+$] embeds *service call* elements into XML documents that are executed when the element is accessed. The recent work on the language XChange [BP05] already aims at specifying more complex events and actions; nevertheless, it still presents a monolithic language for ECA rules in the Web, not dealing with the issue of language heterogeneity.

Structure of the Paper. In the next section, we analyse the abstraction levels of behavior (and ECA rules) in the Semantic Web. The modular structuring of our approach into different language families is presented in Section 3. Section 4 then analyzes the common structure and requirements for the languages for each of the parts in an ECA rule, i.e. languages for events, for querying static data and testing conditions, and for actions. Section 5 describes the global semantics of the rules, focussing on the handling of variables for communication between the rule components. Section 6 concludes the paper.

2 Behavior: Abstraction Levels

As described above, the *Semantic Web* can be seen as a network of autonomous (and autonomously evolving) nodes. Each node holds a *local* state consisting of extensional data (facts), metadata (schema, ontology information), optionally a knowledge base (intensional data), and, again optional, a behavior base. In our case, the latter is given by the ECA rules under discussion that specify which actions are to be taken upon which events under which conditions.

In the same way as the “Semantic Web Tower” distinguishes between the data level and the semantic (RDF/OWL) level, behavior can be distinguished wrt. different levels. There is local behavior of Web nodes, partially even “hidden” inside the database, local rules on the logical level, and *Business Rules* on the application level. The cooperation in the Semantic Web by global *Business Rules* is then based on local behavior. The proposed comprehensive framework for active rules in the Web integrates all these levels.

Physical Level: Database Triggers. The base level is provided by rules on the *programming language and data structure level* that react directly on changes of the underlying data. Usually they are implemented inside the database as *triggers*, e.g., in SQL, of the form ON database-update WHEN condition BEGIN pl/sql-fragment END. In the Semantic Web, the data model level is assumed to be in XML (or RDF, see below) format. While the SQL triggers in relational databases are only able to react on changes of a given tuple or an attribute of a tuple, the XML and RDF models call for more expressive event specifications according to the (tree or graph) structure. Work on triggers for XML data or the XQuery language has e.g. been described in [BBCC02,BPW02,PPW03,MAA05].

Logical Level: RDF Triggers. Triggers for RDF data have been described in [PPW04,MAA05]. Triggering events on the RDF level should usually bind variables Subject, Property, Object, Class, Resource, referring to the modified items (as URIs), respectively in the same form as SQL’s `OLD` and `NEW` values. In case that data is stored in an RDF database, these triggers can directly be implemented on the physical, storage level. RDF trigger events already use the terminology of the *application ontology* but are still based on the RDF structures of the logical level. Application-level events can be raised by such rules, e.g.,

```
ON INSERT OF has_professor OF department
  % (comes with parameters $subject=dept, $property=has_professor,
  % and $object=prof)
RAISE EVENT (professor_hired($object, $subject))
```

which is then actually an event professor_hired(*prof, dept*) of the application ontology on which *business rules* can react.

On the physical and logical levels, actions and events in general coincide, and consist of updates to data items.

Semantic Level: Active Rules. In rules on the semantic level, the events, conditions and actions refer to the ontology level:

```
ON professor_hired($prof, $dept)
LET $books := select relevant textbooks for subjects taught by $prof
IF enough money available
DO order(bookstore,$books)
```

Here, there is an important difference between *actions* and *events*: an event is a visible, possibly indirect or derived, consequence of an action. E.g., the action is to "debit 200E from Alice's bank account", and visible events are "a change of Alice's bank account" (that is immediately detectable from the update operation), or "the balance of Alice's bank account becomes below zero" (which has to be derived from an update).

More complex rules also use composite events and queries against the Web. Composite events in general consist of subevents at that are originally located at several different locations.

3 Language Heterogeneity and Structure: Rules, Rule Components and Languages

An ECA concept for supporting interoperability in the Semantic Web needs to be flexible and adapted to the "global" environment. Since the Semantic Web is a world-wide living organism, nodes "speaking different languages" should be able to interoperate. So, different "local" languages, be it the condition (query) languages, the action languages or the event languages/event algebras have to be integrated in a common framework. There is a more succinct separation between event, condition, and action part, which are possibly (i) given in separate languages, and (ii) possibly evaluated/executed in different places. For this, an (extendible) ontology for rules, events, and actions that allows for *interoperability* is needed, that can be combined with an infrastructure that turns the instances of these concepts into objects of the Semantic Web itself.

In the present paper, we will focus on the language and markup issues; a corresponding service-oriented architecture is discussed in [MAA05].

3.1 Components of Active Rules in the Semantic Web

A basic form of active rules is that of the well-known *database triggers*, e.g., in SQL, of the form ON *database-update* WHEN *condition* BEGIN *pl/sql-fragment* END. In SQL, the *condition* can only use very restricted information about the immediate database update. In case that an action should only be executed under certain conditions which involve a (local) database query, this is done in a procedural way in the *pl/sql-fragment*. This has the drawback of not being declarative; reasoning about the actual effects would require to analyze the program code of the *pl/sql-fragment*. Additionally, in the distributed environment of the Web, the query is probably (i) not local, and (ii) heterogeneous in the language – queries against different nodes may be expressed in different languages. For our framework, we prefer a *declarative* approach with a *clean, declarative* design as a "Normal Form": Detecting just the dynamic part of a situation (event), then check *if* something has to be done by first obtaining additional information by a query and then evaluating a *boolean* test, and, if "yes", then actually *do* something – as shown in Figure 1.

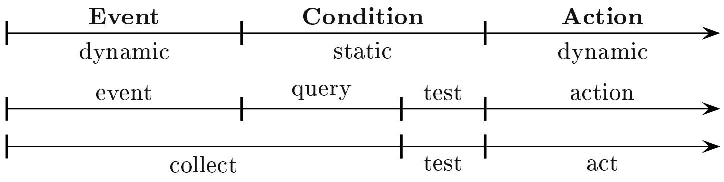


Fig. 1. Components and Phases of Evaluating an ECA Rule

With this further separation of tasks, we obtain the following structure:

– every rule uses an event language, one or more query languages, a test language, and an action language for the respective components,
– each of these languages and their constructs are described by metadata and an ontology of its semantics, and their nature as a language, e.g., associating them with a processor,
– there is a well-defined *interface* for communication between the E, Q&T, and A components by variables.

Sublanguages and Interoperability. For expressing and applying such rules in the Semantic Web, a uniform handling of the event, querying, testing, and action sublanguages is required. Rules and their components are objects of the Semantic Web, i.e., subject to a generic *rule ontology* as shown in the UML model in Figure 2. The ontology part then splits in a structural and application-oriented part, and an infrastructure part about the language itself. In this paper, we restrict ourselves to the issues of the ECA language structure and the markup itself. From the infrastructure part, we here only need to know that each language is identified by a URI with which information about the specific language (e.g, an XML Schema, an ontology of its constructs, a URL where an interpreter is available) is associated; details about a service-oriented architecture proposal that makes use of this information can be found in [MAA05].

3.2 Markup Proposal: ECA-ML

According to the above-mentioned structure of the rules, we propose the following XML markup. The connection with the language-oriented resources is provided via the namespace URIs:

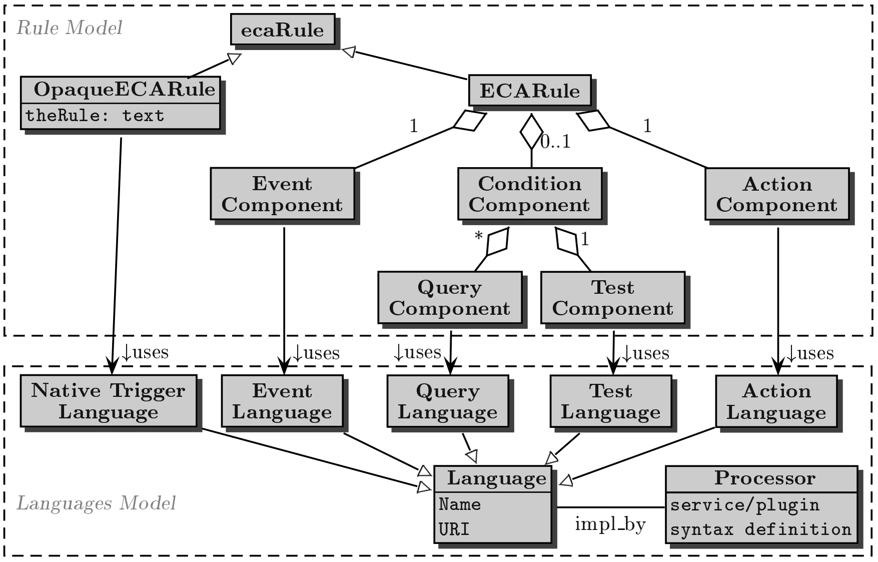


Fig. 2. ECA Rule Components and corresponding Languages II

```
<!ELEMENT rule (event, query*, test?, action+)>
<eca:rule declaration of namespaces e.g. xmlns:evlg="http://my.event-language.org" >
  rule-specific contents
  <eca:event> event specification as <evlg:sequence> ... </evlg:sequence> </eca:event>
  <eca:query> query specification </eca:query>
  <eca:test> test specification </eca:test>
  <eca:action> action specification </eca:action>
  <!– event, condition, and action specification use markup elements
    of their sublanguage's namespaces; see below –>
</eca:rule>
```

A similar markup for ECA rules has been used in [BCP01] with *fixed* languages (using a basic language for atomic events on XML data, XQuery as condition language and SOAP in the action part). This fixed approach falls short wrt. the language heterogeneity, and especially the use and integration of languages for composite events. In the same way, the XChange approach [BP05] uses a fixed language for specifying the event, condition, and action part. In contrast, the approach proposed here allows for using arbitrary languages. Thus, these other proposals are just *two* possible configurations. Our approach even allows to mix components of both these proposals.

Triggers as Rules. The above mentioned database triggers, where a rule is given in an internal syntax, are just wrapped as *opaque* rules. In such a case, the eca:rule element contains only an eca:opaque element with text contents (program code of some rule language) and attributes lang (text) and ref (URI where an interpreter is found, similar to the namespace); a similar mechanism to XML's NOTATION can also be applied.

```
<eca:rule>
  <eca:opaque name="SQL trigger" ref= "uri of the trigger language">
    ON database-update WHEN condition BEGIN action END
  </eca:opaque>
</eca:rule>
```

Since opaque rules are ontologically "atomic" objects, their event, condition, and action parts cannot be accessed by Semantic Web concepts. Note that there are canonic mappings between such triggers and their components and the general ECA ontology, where then the components still end up as opaque native code segments (see e.g. the analysis of the components structure in Section 4; especially Fig. 5).

3.3 Hierarchical Structure of Languages

The framework defines a hierarchical structure of language families (wrt. embedding of language expressions) as shown in Figure 3: As described until now, there is an ECA language, and there are (heterogeneous) event, query, test, and action languages. Rules will combine one or more languages of each of the families. In general, each such language consists of an own, application-independent, syntax and semantics (e.g., event algebras, query languages, boolean tests, process algebras or programming languages) that is then applied to a domain (e.g. travelling, banking, universities, etc.). The domain ontologies define the static and dynamic notions of the application domain, i.e., predicates or literals (for queries and conditions), and events and actions (e.g. events of train schedule changes, actions of reserving tickets, etc.). Additionally, there are domain-independent languages that provide primitives (with arguments), like general communication, e.g. received_message(M) (where M in turn contains domain-specific content), or transactional languages with an action commit(A) and an event committed(A) where A is a domain-specific action.

In the next section, we discuss common aspects of the languages on the "middle" level (that immediately lead to the tree-style markup of the respective components, thus yielding a straightforward XML markup). Section 5 then deals with the interplay between the rule components.

4 Common Structure of Component Languages

The four types of rule components use specialized types of languages that, although dealing with different notions, share the same algebraic language structure:

- event languages: every expression gives a description of a (possibly composite) event. Expressions are built by composers of an event algebra, where the leaves are atomic events of the underlying application;
- query languages: expressions of an algebraic query-language;

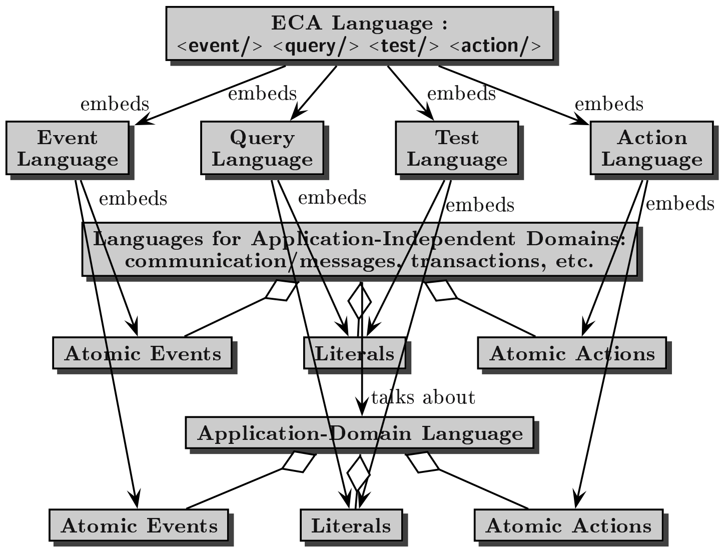

ECA Language :
<event/> <query/> <test/> <action/>
embeds
Event Language
Query Language
Test Language
Action Language
Languages for Application-Independent Domains:
communication/messages, transactions, etc.
Atomic Events
Literals
Atomic Actions
talks about
Application-Domain Language

Fig. 3. Hierarchy of Languages

– test languages: they are in fact formulas of some logic over literals (of that logic) and an underlying domain (that determines the predicate and function symbols, or class symbols etc., depending on the logic);
– action languages: every expression describes an (possible composite) activity. Here, algebraic languages (like process algebras) or "classical" programming languages (that nevertheless consist of expressions) can be used. Again, the atomic items are actions of the application domain.

Algebraic Languages. As shown in Figure 4, all component languages consist of an algebraic language defining a set of *composers*, and embedding *atomic* elements (events, literals, actions) that are contributed by *domain languages*, either for specific applications or application-independent (e.g., messaging). Expressions of the language are then (i) atomic elements, or (ii) composite expressions recursively obtained by applying composers to expressions. Due to their structure, these languages are called *algebraic languages*, e.g. used in *event algebras*, *algebraic query languages*, and *process algebras*. Each composer has a given *cardinality* that denotes the number of expressions (of the same type of language, e.g., events) it can compose, and (optionally) a sequence of parameters (that come from another ontology, e.g., time intervals) that determines its *arity* (see Figures 4 and 5). For instance, "E_1 followed_by E_2 within t" is a binary composer to recognize the occurrence of two events (atomic or not) in a particular order within a time interval, where t is a parameter.

Thus, language expressions are in fact trees which are marked up accordingly. The markup elements are provided by the definition of the individual languages, "residing" in, and distinguished by, the appropriate namespaces: the expression "structure" inside each component is built from elements of the algebraic lan-

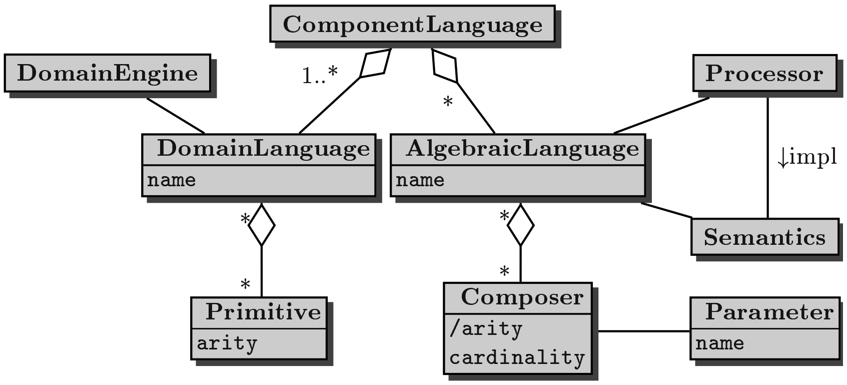


Fig. 4. Notions of an Algebraic Language

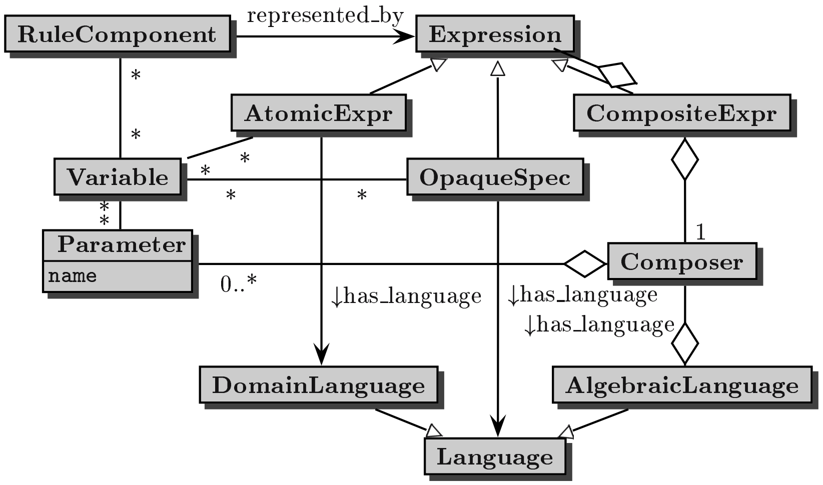


Fig. 5. Syntactical Structure of Expressions of an Algebraic Language

guage. An expression is either an atomic one (atomic event, literal, action) that belongs to a domain language, or an opaque one that is a code fragment of some event/query/logic/action language, or a composite expression that consists of a composer (that belongs to a language) and several subexpressions (where each recursively also belongs to a language). The leaves of the expression trees are the atomic events, literals, or actions, contributed by the application domains (and residing in the domain's ontology and namespace); they may again have an internal structure in the domain's namespace.

Note that it is also possible to nest composers and expressions from different languages of the same kind (e.g., an event sequence where the first part is described in event algebra A and the second in another algebra B), distinguishing them by the namespaces they use. Thus, languages are not only associated once on the *rule component* level, but this can also be done on the *expression* level.

5 Semantics of Rule Execution

For classical deductive rules, there is a *bottom-up* evaluation where the body is evaluated and produces a set of tuples of variable bindings. Then, the rule head is "executed" by *iterating* over all bindings, for *each* binding instantiating the structure described in the head (in some languages also executing actions in the head). The semantics of ECA rules should be as close as possible to this semantics, adapted to the temporal aspect of an event:

ON event AND additional knowledge, IF condition then DO something.

To support communication between heterogeneous languages at the rule component level, there must be a precise convention between all these languages on how the different components of a rule can exchange information and interact with each other. In the following, we state some requirements on the contributing sublanguages and provide technical means to integrate these languages with our framework.

5.1 Logical Variables

We propose to use *logical variables* in the same way as in Logic Programming. For each instance of a rule, a variable must *bound* only once. In case that a variable *occurs* more than once, it acts then as a join variable. While in LP rules, variables must be bound by a positive literal in the body to serve as join variables in the body and to be used in the head, in ECA rules we have four components: A variable must be bound in the rule, in an "earlier" (E<Q<T<A) or at least the same component as where it is used. Usage can be as a join variable in case of the E, Q, or T component, or to execute ("derive") an action in the action component (that in ECA corresponds to the rule head). This leads to a definition of safety of ECA rules that is similar to that of LP rules. Variables can be bound to several things: values/literals, references (URIs), XML or RDF fragments, or events (marked up as XML or RDF fragments). Expressions can also use local variables, e.g., in first-order logic conditions scoped by a quantifier.

Variable Handling on the Rule Level. As in Logic Programming, the semantics of rules is based on sets of tuples of (answer) variable bindings. We propose to use a simple tuple-based representation for interchange of bindings:

```
<variable-bindings>
  <tuple>
    <variable name="name" ref="rdf-uri" />
    <variable name="name"> contents </variable>
      :
  </tuple>
</variable-bindings>
```

Variable Handling in E, Q, T, and A Sublanguages. While the semantics of the ECA *rules* provides the infrastructure for these variables, the markup of specific languages must provide the actual handling of variables in its expressions. Currently languages mainly use variables in two ways:

- Languages that bind variables by matching free variables (e.g. query languages like Datalog, F-Logic [KL89], XPathLog [May04]). Here, the matches can be *literals* (Datalog) or literals and structures (e.g., in F-Logic, XPathLog, Xcerpt [BS02]). Similar techniques can also be applied to design languages for the event component.
- Functional-style languages: the sublanguages for the query and event components can be designed as functions over a database or an event stream. In the XML world, such languages return a (nameless) data fragment (e.g. XQuery; also the expressions of the above-mentioned F-Logic, XPathLog and Xcerpt can be used in this way). For event languages, the "result" of an expression can be considered the sequence of detected events that "matched" the event expression in an event stream (e.g., as in XChange [BP05]).

Variables: Syntax. We propose constructs for handling variables borrowed from XSLT: use variables by {$*var-name*} and by <variable name="..."> elements:

- <eca:variable name="*name*">*content*</eca:variable>
  where *content* can be any expression whose value is bound to the variable (i.e., an event specification or a query).
- <eca:variable name="*name*" select="*ql-expr*"/>
  Such expressions can be used for navigational access/comparison of values, or for defining a new variable based on already bound ones in *expr*, and are to be understood as a shorthand for

```
<eca:variable name="name">
  <eca:query xmlns:ql="ql-url">
    <eca:opaque> expr </eca:opaque>
  </eca:query>
</eca:variable>
```

  where *ql* is a (simple!) query language, e.g. XPath.

Both constructs can be used on the rule level (e.g., for binding the result of the event component; see later example), and we recommend also to consider them when designing component languages.

5.2 Firing ECA Rules: The Event Component

Formally, detection of an event results in an occurrence indication, together with information that has been collected. An ECA rule is fired for each successful detection of the specified event, and initial variable bindings are produced by the event component. The event component consists, as shown above, of an event algebra term whose leaves are atomic events. The pattern is "matched" against the stream of detected events. Inside of <eca:atomic-event> elements, the domain namespaces are used for specifying *event patterns* to be matched. In the event component, variables can be bound as described above pattern-based with <eca:variable name= "*var-name*"> ... </eca:variable> , or navigation-based: inside the atomic event itself (as an XML fragment) is available as $event. Then,

<eca:variable name= "*var-name*" select= "$event/*path*..."/>

can be used to match and access data within the event.

In many approaches (including the SNOOP event algebra [CKAK94]), the "result" of event detection is the sequence of the events that "materialized" the event pattern to be detected. In this case, an appropriate way is to bind this result to a variable (in the present case, using the XML representations of the events). Further variable bindings can then be extracted by subsequent <eca:query> or <eca:variable> elements.

Example 1. Consider the following scenario: "when registration for an exam of subject S is opened, students X register, and registration for S closes, then ... do something". This cumulative event *can be specified in SNOOP as*

$$\mathcal{A}^*(\text{reg\_open(Subj)}, \text{register(Subj, Stud)}, \text{reg\_close(Subj)}) \ .$$

The incoming domain-level events are e.g. of the form

<uni:register subject="Databases" name="John Doe"/>.

The following markup of the event component binds the complete sequence to regseq *and the subject to* Subj *(note that* Subj *is used as a join variable):*

```
<eca:rule ... >
  <eca:variable name="regseq">
    <eca:event xmlns:xmlsnoop="http://xmlsnoop.nop">
      <xmlsnoop:cumulative>
        <xmlsnoop:atomic>
          <uni:reg_open>
          <xmlsnoop:variable name="Subj" select="$event/@Subject" />
          </uni:reg_open>
        </xmlsnoop:atomic>
        <xmlsnoop:atomic> <uni:register subject="$Subj"/> </xmlsnoop:atomic>
        <xmlsnoop:atomic> <uni:reg_close subject="$Subj"/> </xmlsnoop:atomic>
      </xmlsnoop:cumulative>
    </eca:event>
  </eca:variable>
    :
</eca:rule>
```

Note the namespaces: eca *for the rule level,* xmlsnoop *for the event algebra level (which also supports the variables) and* uni *for the domain level.*

The event component collects the relevant events, and returns them as a sequence, e.g. resulting in the following variable bindings:

```
<variable-bindings>
  <tuple>
    <variable name="regseq">
      <reg_open subject="Databases"/>
      <register subject="Databases" name="John Doe"/>
      <register subject="Databases" name="Scott Tiger"/>
        :
      <reg_close subject="Databases"/>
    </variable>
    <variable name="Subj">Databases</variable>
  </tuple>
</variable-bindings>
```

5.3 The Query Component

The query component is concerned with *static* information that is obtained and restructured from analyzing the data that has been collected by the event component (in the variable bindings) and, based on this data, stating queries against databases and the Web. Whereas the firing of the rule due to a successful detection of an event results in exactly one tuple of variable bindings, the query component is very similar to the evaluation of database queries and rule bodies in Logic Programming: in general, it results in a set of tuples of variable bindings. We follow again the Logic Programming specification that every answer produces a variable binding. For variable binding by matching (as in Datalog, F-Logic, XPathLog, Xcerpt etc.), this is obvious. Since we also allow variable bindings in the functional XSLT style, the semantics is adapted accordingly:

– each answer node of an XPath expression yields a variable binding;
– each node that is *return*ed by an XQuery query yields a variable binding; if the XQuery query is of the form
  <*name*>{ for ... where ... return ...} </*name*> ,
  then the whole result yields a single variable binding.

Example 2. Consider again Example 1 where the resulting event contained several registrations of students. The names of the students can be extracted as multiple string-valued variables:

```
<eca:rule ... >
  ... same as above, binding variables "Subj" and "regseq" ...
  <eca:variable name="Student">
    <eca:query>
      <eca:opaque lang='xpath'>
        $regseq//register[@subject=$Subj]/@name/string()
      </eca:opaque>
    </eca:query>
  </eca:variable>
    :
</eca:rule>
```

The above query generates the extended variable bindings
$\beta_1 = \{\mathit{Subj} \rightarrow \text{'Databases'},\ \mathit{regseq} \rightarrow \text{(as above)},\ \mathit{Student} \rightarrow \text{'John Doe'}\}$,
$\beta_2 = \{\mathit{Subj} \rightarrow \text{'Databases'},\ \mathit{regseq} \rightarrow \text{(as above)},\ \mathit{Student} \rightarrow \text{'Scott Tiger'}\}$.

5.4 The Test Component

In general, the evaluation of conditions is based on a logic over literals with boolean combinators and quantifiers. A Markup Language exists with FOL-RuleML [BDG$^+$]. Instead of first-order atoms, also "atoms" of other data models can be used. Note that XPath expressions are also literals that result in a true/false (true if the result set is non-empty) value. The result of the test component is the set of tuples of variable bindings that satisfy the condition (for further propagation to the action part).

5.5 The Action Component

The action component is the one where *actually* something is done in the ECA rule: for each tuple of variable bindings, the action component is executed. The action component may consist of several <eca:action> elements which contain action specifications, possibly in different action languages, e.g., the CCS process algebra [Mil83]. This can be updates on the database level, explicit message sending (e.g. to Web Service calls), or actions on the ontology level (that must then be implemented appropriately). The semantics is that all actions are executed.

6 Conclusion

We described the concepts and proposed an XML markup for a general ECA-rule framework for the Semantic Web, taking into account the heterogeneity of (existing) languages. By using the namespace/URI mechanism for identifying the languages, also appropriate services can be located. Such an architecture based on this framework is described in [MAA05]. Rules can e.g. be used both for defining rule-based Web Services and for combining the functionality of Web Services (that provide events and execute actions) by rules.

Although the above examples all used "syntactical" languages in XML term markup for the components, also languages using a semantical, e.g., OWL-based representation (which have to be developed) can be used if they support the variable-based communication mechanisms described in Section 5.

There are several issues that are explicitly not dealt with in our approach – because they are encapsulated inside (and "bought with") the concepts to be integrated: The detection of complex events is done and provided by the individual event languages and their engines – the framework provides an environment for embedding them. In the same way, query evaluation itself is left to the original languages and processors to be embedded into the global approach; also actual execution of actions (and transactions) is left with the individual solutions.

Acknowledgements

This research has been funded by the European Commission within the 6th Framework Programme project REWERSE, no. 506779.

References

[ABM$^+$] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration. In *VLDB*, 2002.

[BBCC02] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Intl. Conference on Data Engineering (ICDE)*, pp. 403–418, 2002.

[BCP01] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing Reactive Services to XML Repositories Using Active Rules. In *WWW Conf. (WWW 2001)*, 2001.

[BDG$^+$] H. Boley, M. Dean, B. Grosof, M. Sintek, B. Spencer, S. Tabet, and G. Wagner. FOL RuleML: The First-Order Logic Web Language. `http://www.ruleml.org/fol/`.

[BP05] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *ACM Symp. Applied Computing*, 2005.

[BPW02] J. Bailey, A. Poulovassilis, and P. T. Wood. An Event-Condition-Action Language for XML. In *WWW Conf.*, 2002.

[BS02] F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation Unification. In *Intl. Conf. on Logic Programming (ICLP)*, Springer LNCS 2401, 2002.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*, 1994.

[KL89] M. Kifer and G. Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. In *ACM SIGMOD*, pp. 134–146, 1989.

[MAA05] W. May, J. J. Alferes, and R. Amador. An Ontology- and Resources-Based Approach to Evolution and Reactivity in the Semantic Web. In *Ontologies, Databases and Semantics (ODBASE)*, to appear in Springer LNCS, 2005.

[May04] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 4(3), 2004.

[Mil83] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, pp. 267–310, 1983.

[PPW03] G. Papamarkos, A. Poulovassilis, and P. T. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *Workshop on Semantic Web and Databases (SWDB'03)*, 2003.

[PPW04] G. Papamarkos, A. Poulovassilis, and P. T. Wood. RDFTL: An Event-Condition-Action Rule Languages for RDF. In *Hellenic Data Management Symposium (HDMS'04)*, 2004.

[RML] Rule Markup Language (RuleML). `http://www.ruleml.org/`.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Halevy, and D. Weld. Updating XML. In *ACM SIGMOD*, pp. 133–154, 2001.

[XML00] XML:DB. XUpdate - XML Update Language. `http://xmldb-org.sourceforge.net/xupdate/`, 2000.

Towards an Abstract Syntax and Direct-Model Theoretic Semantics for RuleML

Adrian Giurca and Gerd Wagner

Institute of Informatics, Brandenburg University of Technology at Cottbus
{Giurca,G.Wagner}@tu-cottbus.de

Abstract. This paper contains a proposal of an abstract syntax and a model theoretic semantics for NafNegDatalog, sublanguage of RuleML [9]. The model-theoretic semantics use the partial logic ([7], [10]) to provide an interpretation and a satisfaction relation, and provide a formal meaning for RuleML knowledge bases written in the abstract syntax.

Keywords: rule markup languages, RuleML, abstract syntax, semantics, partial logic.

1 Introduction

The RuleML Initiative [9] started in August 2000 during the Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000). It has brought together expert teams from several countries, including leaders in Knowledge Representation and Markup Languages, from both academia and industry.

The RuleML Initiative is developing an open, vendor neutral XML/RDF-based rule language. This will allow for the exchange of rules between various systems including distributed software components on the Web, heterogeneous client-server systems found within large corporations, etc. The RuleML language offers XML syntax for rules Knowledge Representation, interoperable among major commercial and non-commercial rules systems.

During this period 12 sublanguages was developed (see [2], Figure 2). Each sublanguage has a proposed XML syntax and DTD for validation and from version 0.85 we have also an XML Schema for validation.

In this paper we propose an abstract syntax and a model theoretic semantics for the NafNeg-Ur-Datalog sublanguage of RuleML.

2 The Abstract Syntax

The abstract syntax is specified here by means of a version of Extended BNF, very similar to the EBNF (based on Wirth's definition) notation used for XML. Nonterminals are enclosed between angle brackets (< and >) and terminals are in boldface. Alternatives are either separated by vertical bars (|) or are given in different productions. Components that can occur at most once are enclosed in square brackets ([...]); components that can occur any number of times (including zero) are enclosed in braces ({...}). Whitespace is ignored in all the productions.

Definition 1 (Relations). *A* RuleML relation *is a named predicate or a named builtin predicate:*

$\langle Rel \rangle$ ::= **Rel(**$\langle relID \rangle$**).**

$\langle relID \rangle$ ::= **URIreference.**

In this paper we follow the Datalog (constructor-function-free) sublanguage, the foundation for the kernel of RuleML, i.e. we don't not consider functions in atom construction (Datalog model).

Definition 2 (Terms). *A RuleML* term *is a* variables *or an* individual constant*:*

$\langle Term \rangle$ ::= $\langle Variable \rangle$ | $\langle Individual \rangle$**.**

$\langle Variable \rangle$ ::= **Var(**$\langle varName \rangle$**).**

$\langle Individual \rangle$ ::= **Ind(**$\langle individualID \rangle$**).**

$\langle individualID \rangle$ ::= **URIreference.**

$\langle varName \rangle$ ::= **UnicodeString.**

Slots are basic RuleML constructs which are used in slotted atoms (see below) for capturing object descriptions.

Definition 3 (Slots). *The abstract syntax of* RuleML slots *is:*

$\langle Slot \rangle$ ::= **Slot(Property(**$\langle slID \rangle$**),** $\langle slotValue \rangle$**).**

$\langle slotValue \rangle$ ::= $\langle Individual \rangle$ | $\langle Data \rangle$

$\langle Data \rangle$ ::= **Data(**$\langle dataValue \rangle$**).**

$\langle dataValue \rangle$ ::= **UnicodeString.**

$\langle slID \rangle$ ::= **URIreference.**

RuleML allow two kinds of atoms: *positional atoms* and *slotted atoms.* Positional atoms are used for representing Prolog like atoms while the slotted atoms capture object descriptions.

Definition 4 (Atoms). *The abstract syntax of* RuleML atoms *is:*

$\langle Atom \rangle$ ::= $\langle PositionalAtom \rangle$ | $\langle SlottedAtom \rangle$**.**

$\langle PositionalAtom \rangle$ ::= **PosAtom(**$\langle Rel \rangle$**, {**$\langle Term \rangle$**}).**

$\langle SlotAtom \rangle$ ::= **SlotAtom(**$\langle Term \rangle$**, {**$\langle Slot \rangle$**}).**

Below we provide the abstract syntax of literals (negated atoms) in RuleML. Intuitively speaking, *weak negation* captures the absence of positive information, while *strong negation* captures the presence of explicit negative information (in the sense of Kleene's 3-valued logic).

Definition 5 (Negations). *The abstract syntax of weak negation and strong negations is:*

$\langle \mathit{WeakNegation} \rangle$ ::= **Naf(**$\langle \mathit{Atom} \rangle$**).**

$\langle \mathit{StrongNegation} \rangle$ ::= **Neg(**$\langle \mathit{Atom} \rangle$**).**

Definition 6 (AndOrNafNegFormula). *The abstract syntax of* RuleML AndOrNafNegFormula *is:*

$\langle \mathit{AndOrNafNegFormula} \rangle$ ::= $\langle \mathit{Atom} \rangle$ | $\langle \mathit{WeakNegation} \rangle$ | $\langle \mathit{StrongNegation} \rangle$
| $\langle \mathit{Conjunction} \rangle$ | $\langle \mathit{Disjunction} \rangle$

$\langle \mathit{Conjunction} \rangle$ ::= **And({**$\langle \mathit{AndOrNafNegFormula} \rangle$**}).**

$\langle \mathit{Disjunction} \rangle$ ::= **Or({**$\langle \mathit{AndOrNafNegFormula} \rangle$**}).**

A RuleML *rule* has a *body* which is represented by an and-or-naf-neg-formula and a *head* which is an atom or a strong negated atom. Rules with an empty body are *facts* and rules with an empty head are *queries.*

Definition 7 (Rule). *The abstract syntax of the rule in RuleML is:*

$\langle \mathit{Rule} \rangle$ ::= **Imp(**[$\langle \mathit{ruleID} \rangle$,][$\langle \mathit{Body} \rangle$,][$\langle \mathit{Head} \rangle$]**).**

$\langle \mathit{Body} \rangle$ ::= **Body(**[$\langle \mathit{AndOrNafNegFormula} \rangle$]**).**

$\langle \mathit{Head} \rangle$ ::= **Head(**$\langle \mathit{Atom} \rangle$**)** | **Head(**$\langle \mathit{StrongNegation} \rangle$**).**

$\langle \mathit{ruleID} \rangle$ ::= **URIreference.**

In the following example presents the abstract syntax form of a rule from the RDF/RuleML interoperability position paper [3].

Example 1 (A rule). The following is an example of a rule using both kinds of negation. Here, the relation requiresService does not allow Closed-World inference, while the relation isAssignedToRentalContract does allow it, and, hence, the former is negated with Neg and the latter with Naf.

```
Imp("R1",
 Body(
  And(
   PosAtom(Rel(ex:RentalCar),Var(Car))
   Neg(
    PosAtom(Rel(ex:requiresService),Var(Car))
   )
   Naf(
    PosAtom(Rel(isAssignedToRentalContract ),Var(Car))
   )
  )
 )
 Head(
  PosAtom(Rel(ex:isAvailable),Var(Car))
 )
)
```

Definition 8 (Knowledge Base). *Any* RuleML **Knowledge Base** *is a set of rules.*

$\langle KB \rangle$ ::= **{**$\langle Rule \rangle$**}.**

3 RuleML Semantics in Partial Logic

In business and administrative domains, most of the information to be processed is assumed to be complete (this tacit assumption is called Closed-World Assumption, [8] in AI). But in other domains, such as in medicine and criminology, most information is incomplete. RuleML is a rule language for applications so it should allow expressing rules for both types of information.

3.1 Why Partial Logic?

Unlike negation in classical logic, real-world negation is not a simple two-valued truth function. The simplest generalization of classical logic that is able to account for two kinds of negation is partial logic giving up the classical bivalence principle and subsuming a number of 3-valued and 4-valued logics (see [7]).

Because the web does not operate under de CWA hypothesis (see [1]), RuleML should distinguish between relations (or slots) that are totaly represented (a total assumption corresponds to a predicate-specific Closed-World Assumption) or are partial represented.

In the case of a completely represented predicate, negation-as-failure reflects falsity, and negation-as-failure and strong negation collapse into classical negation. In the case of a partial represented relation or slot, negation-as-failure only reflects non-provability, but does not allow to infer the classical negation (for details see [11]).

In standard logics, there is a close relationship between a derivation rule and the corresponding implicational formula: they have the same models. For nonmonotonic rules (with negation-as-failure, [4]) this is no longer the case: the intended models of such a rule are in general not the same as the intended models of the corresponding implication (see [6] and [5]).

These are the main reasons for that we consider partial logic to be the most appropriate logic for interpreting RuleML rules.

3.2 Direct-Model Theoretic Semantics

Definition 9 (Vocabulary). *The* RuleML vocabulary *is defined by the following tuple*

$$\mathcal{V}oc = (\mathrm{Rel}, \mathrm{TRel}, \mathrm{Pr}, \mathrm{DLit}, \mathrm{Var}, \mathrm{Ind})$$

where Rel *is the set of relation names,* TRel *is the set of total relations names,* Pr *is the set of property names,* DLit *is the set of all data literals names,* Var *is the set of variables names and* Ind *is the set of individuals names.*

Note that Pr *is in fact the set of al URI references to binary relations, i.e.* $\mathrm{Pr} \subseteq \mathrm{Rel}$. *We suppose that all these sets are nonempty, pairwise disjoint and* $\mathrm{TRel} \subseteq \mathrm{Rel}$.

Let $\mathcal{O}$ be the set of all objects and $\mathcal{V}(DLit)$ the set of all data literal values. In this paper each relation or property is a pair $r = \langle r^t, r^f \rangle$ such that $r^t, r^f \subseteq \mathcal{O}^2$. We allow only coherent relations and properties (see [7] for details), i.e, if r is a relation, then $r^t \cap r^f = \emptyset$.

Definition 10 (Interpretation). *An abstract interpretation is a tuple of functions*

$$\mathcal{I} = (\mathcal{I}_{DLit}, \mathcal{I}_{Ind}, \mathcal{I}_{Pr}, \mathcal{I}_{Rel})$$

such that:

1. $\mathcal{I}_{DLit} : \mathrm{DLit} \longrightarrow \mathcal{V}(DLit)$, *maps each data literal into a value i.e.*

$$\mathcal{I}(Data(d)) = \mathcal{I}_{DLit}(d) \in \mathcal{V}(DLit)$$

   where $\mathcal{V}(DLit)$ *is the value space*[3].
2. $\mathcal{I}_{Ind} : \mathrm{Ind} \longrightarrow \mathcal{O}$, *maps individuals names to objects,*

$$\mathcal{I}(Ind(id)) = \mathcal{I}_{Ind}(id) \in \mathcal{O}$$

3. $\mathcal{I}_{Pr} : \mathrm{Pr} \longrightarrow \mathcal{O}^2 \times \mathcal{O}^2$, *maps each property name into a pair of binary relations,*

$$\mathcal{I}(Pr(p)) = \mathcal{I}_{Pr}(p) = \langle p^t, p^f \rangle, \text{ with } p^t, p^f \subseteq \mathcal{O}^2$$

   If p *is a total property name, then* $p^t \cup p^f = \mathcal{O}^2$. *Note that in this paper we allow only coherent properties (see [7] for details) i.e.* $p^t \cap p^f = \emptyset$.
4. $\mathcal{I}_{Rel} : \mathrm{Rel} \longrightarrow \mathcal{O}^n \times \mathcal{O}^n$, *maps each n-ary relation name into a pair of n-ary relations*

$$\mathcal{I}(Rel(r)) = \mathcal{I}_{Rel}(r) = \langle r^t, r^f \rangle, \text{ with } r^t, r^f \subseteq \mathcal{O}^n$$

   If r *is a total n-ary relation name, then* $r^t \cup r^f = \mathcal{O}^n$. *Again we allow only coherent relations ([7]) i.e.* $r^t \cap r^f = \emptyset$.

[3] As in RDF, a datatype is characterized by a lexical space, DLit which is a set of Unicode strings; a value space, $\mathcal{V}(Dlit)$; and a total mapping $\mathcal{I}_{DLit}$ from the lexical space to the value space.

Definition 11 (Valuation). *A valuation over an interpretation $\mathcal{I}$ is a function $\mathcal{V} : \mathrm{Var} \longrightarrow \mathcal{O}$, which associate each variable name with an object, i.e.*

$$\mathcal{I}(Var(v)) = \mathcal{V}(v) \in \mathcal{O}$$

Definition 12 (Satisfaction Relation). *Let $\mathcal{V}$ be a valuation. The* satisfaction relation $\models$ *is a pair $\langle \models^t, \models^f \rangle$ such that,*

1. *Let r be a relation name and $\mathcal{I}_{Rel}(r) = (r^t, r^f)$.*
   (a) If r is partial, i.e. $r \in \mathrm{Rel} - \mathrm{TRel}$, then
   $$\begin{aligned} \mathcal{I},_\mathcal{V} \models^t PosAtom(r, t_1, \dots, t_n) \; iff \; \langle \mathcal{I}(t_1), \dots, \mathcal{I}(t_n) \rangle \in r^t \\ \mathcal{I},_\mathcal{V} \models^f PosAtom(r, t_1, \dots, t_n) \; iff \; \langle \mathcal{I}(t_1), \dots, \mathcal{I}(t_n) \rangle \in r^f \end{aligned}$$
   (b) If r is total, i.e. $r \in \mathrm{TRel}$, then
   $$\begin{aligned} \mathcal{I},_\mathcal{V} \models^t PosAtom(r, t_1, \dots, t_n) \; iff \; \langle \mathcal{I}(t_1), \dots, \mathcal{I}(t_n) \rangle \in r^t \\ \mathcal{I},_\mathcal{V} \models^f PosAtom(r, t_1, \dots, t_n) \; iff \; \langle \mathcal{I}(t_1), \dots, \mathcal{I}(t_n) \rangle \notin r^t \end{aligned}$$
2. *Let $(p_i)_{i=1,n}$ be property names, i.e. $\mathcal{I}_{Pr}(p_i) = (p_i^t, p_i^f)_{i=1,n}$, t be a term and $S = \{Slot(Property(p_i), v_i)\}_{i=1,n}$ a slot. Then,*
   (a)
   $$\mathcal{I},_\mathcal{V} \models^t SlotAtom(t, S) \; iff \; \bigwedge_{i=1,n} \{\langle \mathcal{I}(t), \mathcal{I}(v_i) \rangle \in p_i^t\}$$
   (b)
   $$\mathcal{I},_\mathcal{V} \models^f SlotAtom(t, S)$$
   $$iff$$
   $$(\exists i, \; p_i \; total, \; \langle \mathcal{I}(t), \mathcal{I}(v_i) \rangle \notin p_i^t) \vee (\exists j, \; p_j \; partial, \; \langle \mathcal{I}(t), \mathcal{I}(v_j) \rangle \in p_j^f)$$
3. *If $R = Imp(Body(P_1, \dots, P_n)), Head(C))$ is a rule, then*
   $$\mathcal{I},_\mathcal{V} \models^t Head(C) \; if \; \mathcal{I} \models^t (P_1 \wedge \dots \wedge P_n)$$

Definition 13 (Model and Satisfaction Set). *Let R be a rule in RuleML. We say that an interpretation $\mathcal{I} = (\mathcal{I}_{DLit}, \mathcal{I}_{Ind}, \mathcal{I}_{Pr}, \mathcal{I}_{Rel})$ is a* model for *R and we denote $\mathcal{I} \models R$ if and only if*

$$\mathcal{I},_\mathcal{V} \models^t \; R$$

for all variable valuations $\mathcal{V}$.

Let $KB = \{R_1, \dots, R_n\}$ be a RuleML knowledge base. A satisfaction set *for KB is the set:*

$$Sat_\mathcal{I}(KB) = \{\mathcal{V} \mid \mathcal{I},_\mathcal{V} \models^t (R_1 \wedge \dots \wedge R_n)\}$$

Definition 14 (Rule Model). *Let $\mathcal{I}$ be an interpretation and $R = Imp(Body(P_1, \dots, P_n)), Head(C))$ a rule.*

$$\mathcal{I} \models R \; iff \; \bigcap_{i \le n} Sat_\mathcal{I}(P_i) \subseteq Sat_\mathcal{I}(C)$$

Example 2. Let $\mathcal{V}oc$ = (Rel, TRel, Pr, DLit, Var, Ind) be the following vocabulary:

$$\begin{aligned}
&\text{Rel} = \{"ex: RentalCar", "ex: requiresService",\\
&"ex: isAssignedToRentalContract", "ex: isAvailable"\}\\
&\text{TRel} = \{"ex: RentalCar", "ex: isAssignedToRentalContract"\}\\
&\text{Pr} = \emptyset\\
&\text{DLit} = \emptyset\\
&\text{Var} = \{Car\}\\
&\text{Ind} = \{"ex: CT2MDF", "ex: DJ02GCP"\}
\end{aligned}$$

Let $KB = \{R1, R2, R3, R4\}$ be a RuleML knowledge base which consist of the rule $R1$ as in Example 1 and the following facts (rules with empty body):

1. Rule $R2$ "ex:CT20MDF is a RentalCar".
2. Rule $R3$ "ex:DJ02GCP is a RentalCar".
3. Rule $R4$ "ex:DJ02GCP does not require service" ("requiredService is a partial relation").

Below is the abstract semantics of the new rules:

```
Imp("R2",
 Head(PosAtom(Rel("ex:RentalCar"),Ind("ex:CT20MDF")))
)
Imp("R3",
 Head(PosAtom(Rel("ex:RentalCar"),Ind("ex:DJ02GCP")))
)
Imp("R4",
 Head(
  Neg(
   PosAtom(Rel("ex:requireService"),Ind("ex:DJ02GCP"))
  )
 )
)
```

In the following we define an interpretation $\mathcal{I} = (\mathcal{I}_{DLit}, \mathcal{I}_{Ind}, \mathcal{I}_{Sl}, \mathcal{I}_{Rel})$ and the we illustrate some satisfaction results in KB.

1. $\mathcal{I}_{DLit}$ can be any arbitray function (we don't use this function because don't have data literals in our knowledge base).
2. $\mathcal{I}_{Ind}$ is defined below (for simplicity we keep the name of the individual removing the namespace):

   $$\begin{aligned}
   &\mathcal{I}_{Ind}("ex: CT20MDF") = "Opel - Corsa"\\
   &\mathcal{I}_{Ind}("ex: DJ02GCP") = "BMW6"
   \end{aligned}$$

3. $\mathcal{I}_{Sl}$ can be any arbitray function (again, we don't use this function because don't have slot relations in our knowledge base).

4. $\mathcal{I}_{Rel}$ is defined below:

$$\begin{aligned}
&\mathcal{I}_{Rel}(ex : RentalCar) = (rc^t, rc^f)\\
&\mathcal{I}_{Rel}(ex : requireService) = (rs^t, rs^f)\\
&\mathcal{I}_{Rel}(ex : isAssignedToRentalContract) = (irc^t, irc^f)\\
&\mathcal{I}_{Rel}(ex : isAvailable) = (ia^t, ia^f)
\end{aligned}$$

where

$$\begin{aligned}
&rc^t = \{Ind(\text{"}ex : CT20MDF\text{"}), Ind(\text{"}ex : DJ02GCP\text{"})\},\ rc^f = \emptyset,\\
&rs^t = \emptyset,\ rs^f = \{Ind(\text{"}ex : DJ02GCP\text{"})\},\\
&irc^t = \{Ind(\text{"}ex : CT20MDF\text{"})\},\ irc^f = \emptyset,\\
&ia^t = \emptyset,\ ia^f = \emptyset.
\end{aligned}$$

Now, let $\mathcal{V}_1 : \text{Var} \longrightarrow \mathcal{O}$, $\mathcal{V}_1(Car) = \text{"}DJ02GCP\text{"}$. Then,

$$\mathcal{I},_{\mathcal{V}_1} \models^t PosAtom(Rel(ex : isAvailable), Ind(\text{"}ex : DJ02GCP\text{"}))$$

that means "the rental car BMW6 is available for renting".

$$\mathcal{I},_{\mathcal{V}_1} \models^f PosAtom(Rel(ex : isAvailable), Ind(\text{"}ex : CT20MDF\text{"}))$$

that means "the rental car Opel Corsa is not available for renting".

4 Minimal Reasoning

Let $\mathcal{V}oc = (\text{Rel}, \text{TRel}, \text{Pr}, \text{DLit}, \text{Var}, \text{Ind})$ be a vocabulary and KB a RuleML knowledge base based on $\mathcal{V}oc$.

We denote by $Lit_{KB}(\text{Ind})$ the set of all atoms or strong negation of atoms without variables that are constructed using atoms from KB and indivivual constants from Ind i.e the Herbrand base of KB. Each $\mathcal{I}^H \subseteq Lit_{KB}(\text{Ind})$ is a Herbrand interpretation for KB.

Definition 15. *If $\mathcal{I} \in Lit_{KB}(\text{Ind})$ such that $\mathcal{I} \models KB$ we say $\mathcal{I}$ is a* Herbrand model of KB. *Usually we denote a Herbrand interpretation by $\mathcal{I}^H$, a Herbrand model by $\mathcal{M}^H$ and the set of all Herbrand models by $\mathcal{M}od^H(KB)$.*

Definition 16. *Let $\mathcal{I}_1^H, \mathcal{I}_2^H \subseteq Lit_{KB}(\text{Ind})$ be two Herbrand interpretations. We say that $\mathcal{I}_2^H$* extends *$\mathcal{I}_1^H$ and denote $\mathcal{I}_1^H \preceq \mathcal{I}_2^H$, if*

$$\{L \in Lit_{KB}(\text{Ind}) \mid \mathcal{I}_1^H \models L\} \subseteq \{L \in Lit_{KB}(\text{Ind}) \mid \mathcal{I}_2^H \models L\}$$

The informal meaning of this order is: $\mathcal{I}_1^H$ is "less informative than" $\mathcal{I}_2^H$.

Definition 17 (Interpretation Intervals). *The model* interval *is defined as:*

$$[\mathcal{I}_1^H, \mathcal{I}_2^H] = \{\mathcal{I}^H \in Lit_{KB}(\text{Ind}) \mid \mathcal{I}_1^H \preceq \mathcal{I}^H \preceq \mathcal{I}_2^H\}$$

Definition 18 (Stable Model ([7], [13])). *Let $\mathcal{M}^H \in \mathcal{M}od^H(KB)$ be a Herbrand model. $\mathcal{M}^H$ is called* minimally stable model of KB *if there is a chain of Herbrand interpretations $\mathcal{M}_0^H \preceq \ldots \preceq \mathcal{M}_k^H$ such that $\mathcal{M}^H = \mathcal{M}_k^H$ and:*

1. $\mathcal{M}_0^H = \emptyset$.
2. *For all $i \in \{1, \ldots, k\}$, $\mathcal{M}_i^H$ is a minimal extension of $\mathcal{M}_{i-1}^H$ satisfying the heads of all rules whose bodies hold in $\left[\mathcal{M}_{i-1}^H, \mathcal{M}^H\right]$, i.e.*

$$\mathcal{M}_i^H \in \min\left\{\mathcal{I}^H \mid \mathcal{M}_{i-1}^H \preceq \mathcal{I}^H \textit{ and } \mathcal{I}^H \models H(R), \textit{ for all } R \in \mathcal{R}_{\left[\mathcal{M}_{i-1}^H, \mathcal{M}^H\right]}\right\}$$

where R is a rule, $H(R)$ is the head of rule R, $B(R)$ is the body of R and

$$\mathcal{R}_{\left[\mathcal{M}_{i-1}^H, \mathcal{M}^H\right]} = \{R \in KB \mid \mathcal{M}^H \models B(R) \textit{ for all } \mathcal{M}^H \in \left[\mathcal{M}_{i-1}^H, \mathcal{M}^H\right]\}$$

Example 3. (continued) Let $KB = \{R1, R2, R3, R4\}$ be a RuleML knowledge base as in Example 2.

Let's test if the following model $\mathcal{M} = \{L_1, L_2, L_3, L_4\}$ where

$L_1 = PosAtom(Rel(ex : isAvailable), Ind(''ex : CT20MDF''))$
$L_2 = PositionalAtom(Rel(''ex : RentalCar''), Ind(''ex : CT20MDF''))$
$L_3 = PositionalAtom(Rel(''ex : RentalCar''), Ind(''ex : DJ02GCP''))$
$L_4 = Neg(PositionalAtom(Rel(''ex : requireService''), Ind(''ex : DJ02GCP'')))$

is a stable model of KB.
Let $\mathcal{M}_0^H = \emptyset$. Then $\mathcal{R}_{\left[\mathcal{M}_0^H\right]} = \{R_2, R_3, R_4\}$ and, consequently,

$$\mathcal{M}_1^H = \{L_2, L_3, L_4\}$$

Now, $\mathcal{R}_{\left[\mathcal{M}_0^H, \mathcal{M}_1^H\right]} = \{R_1, R_2, R_3, R_4\}$ and

$$\mathcal{M}_2^H = \{L_1, L_2, L_3, L_4\}$$

Finally, $\mathcal{R}_{\left[\mathcal{M}_1^H, \mathcal{M}_2^H\right]} = \{R_1, R_2, R_3, R_4\}$ and

$$\mathcal{M}_3^H = \mathcal{M}_2^H$$

5 Conclusions and Future Work

The paper provide an abstract syntax for the core RuleML sublanguage NafNeg-Datalog and a model theoretic semantic. The semantics is done using partial model theory, as a natural generalization of classical model theory. This semantics is able to capture many important distinctions arising in web rules reasoning, such as explicit falsity vs. non-truth, or total vs. partial predicates. At the object level, these distinctions can be expressed by means of the two negations of partial logic.

One possible quick extension of this work is to generalize the strong negation formulas that is to allow strong negations on and-or-naf-neg-formulas not only to atoms like below:

Example 4 (Allowing strong negation on and-or-naf-neg-formulas).

$\langle \mathit{StrongNegation} \rangle$ ::= **Neg(**$\langle \mathit{AndOrNafNegFormula} \rangle$**).**

All cases of a such formula composition are treated by the following DeMorgan-style rewrite rules expressing the falsification of compound formulas:

$$\begin{array}{l} Neg(F \wedge G) \rightarrow Neg(F) \vee Neg(G) \\ Neg(F \vee G) \rightarrow Neg(F) \wedge Neg(G) \\ Neg(Neg(F)) \rightarrow F \\ Neg(Naf(A)) \rightarrow A \end{array}$$

where F,G are and-or-naf-neg-formulas and A is an atom.

Another extension concern the improvement of the abstract syntax to allow the declaration of a total or partial relation/slot like:

Example 5 (Declare total or partial relations/slots).

$\langle \mathit{Rel} \rangle$ ::= **Rel(**$\langle \mathit{relID} \rangle$, $\langle \mathit{type} \rangle$**).**

$\langle \mathit{SlotAtom} \rangle$ ::= **SlotAtom(**$\langle \mathit{Term} \rangle$, $\langle \mathit{type} \rangle$, **{**$\langle \mathit{Slot} \rangle$**}).**

$\langle \mathit{type} \rangle$ ::= `total` | `partial`.

This improvement will be used by RuleML reasoning engines.

Using partial logic we solve the knowledge representation which need incomplete (partial) predicates. However, it is possible to extend RuleML with capabilities for representing uncertain information in the form of fuzzy sets, fuzzy numbers, possibility and necessity measures, discrete plausibility measures or other quantitative measures of uncertainty.

Also, we don't cover in this paper the important problems of non-coherent relations or non-coherent slots that must be also declared by the rule designer. Allowing non coherent relations and non-coherent slots means to accommodate an inconsistency tolerant reasoning in RuleML.

Finally, seems to be possible to extend RuleML to accommodate the distinction between OWL individuals, instead of using the traditional logic concept individual constants.

References

1. Berners Lee, T., Hendler J., Lassila, O., The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities, Scientific American May 2001.
2. Boley, H., Tabet, S., Wagner, G., Design Rationale of RuleML:A Markup Language for Semantic Web Rules, in Proc. of Int. Semantic Web Working Symposium (SWWS), July 30 - August 1, 2001, Stanford University, California, USA.
3. Boley, H., Mei, J., Sintek, M., Gerd Wagner, G., RDF/RuleML Interoperability, W3C Workshop on Rule Languages for Interoperability Position Paper: 27-28 April 2005, available at http://www.w3.org/2004/12/rules-ws/paper/93/

4. Clark, K.L., Negation as Failure, In H. Gallaire and J. Minker Editors, Logic and Databases, pp.293-322, Plenum Press, 1978.
5. van Gelder, A., Kenneth A. Ross A., K., Schlipf S., J., The Well-Founded Semantics for General Logic Programs, Journal of ACM, Vol.38, No. 3 July 1991, pp.620-650.
6. Gelfond, M., Lifschitz, V., The Stable Model Semantics For Logic Programming, In Proceedings of the 5th International Conference on Logic Programming, 1070–1080. Seattle, USA, August 1988. The MIT Press.
7. Herre, H., Jaspars, J., Wagner G., Partial Logics with Two Kinds of Negation as a Foundation for Knowledge-Based Reasoning, in D.M. Gabbay and H. Wansing (Eds.), What is Negation ?, Kluwer Academic Publishers, 1999.
8. Reiter, R., On Closed World Data Bases, In H. Gallaire and J. Minker Editors, Logic and Databases, pp. 55-76, Plenum Press, 1978.
9. Wagner, G., Antoniou, G., Tabet, S., and Boley, H.,The Abstract Syntax of RuleML Towards a General Web Rule Language Framework, Rule Markup Initiative (RuleML), http://www.RuleML.org
10. Wagner, G., Web Rules Need Two Kind of Negations, in Proc. of Principles and Practice of Semantic Web Reasoning, PPSWR 2003, pp.33-50.
11. Wagner, G., Foundations of Knowledge Systems with Applications to Databases and Agents, Kluwer Academic Publishers, 1998.
12. Wagner, G., Seven Golden Rules for a Web Rule Language, invited contribution to the Trends & Controversies section of IEEE Intelligent Systems 18:5, Sept/Oct 2003.
13. Wagner, G., Herre, H., Stable Models are Generated by a Stable Chain. Journal of Logic Programming 30 (1997) 2, 165 - 177.

A Semantic Web Framework for Interleaving Policy Reasoning and External Service Discovery

Jinghai Rao and Norman Sadeh

School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA, 15213, USA
{sadeh,jinghai}@cs.cmu.edu

Abstract. Enforcing rich policies in open environments will increasingly require the ability to dynamically identify external sources of information necessary to enforce different policies (e.g. finding an appropriate source of location information to enforce a location-sensitive access control policy). In this paper, we introduce a semantic web framework and a meta-control model for dynamically interleaving policy reasoning and external service discovery and access. Within this framework, external sources of information are wrapped as web services with rich semantic profiles allowing for the dynamic discovery and comparison of relevant sources of information. Each entity (e.g. user, sensor, application, or organization) relies on one or more *Policy Enforcing Agents* responsible for enforcing relevant privacy and security policies in response to incoming requests. These agents implement meta-control strategies to dynamically interleave semantic web reasoning and service discovery and access. The paper also presents preliminary empirical results. This research has been conducted in the context of *my*Campus, a pervasive computing environment aimed at enhancing everyday campus life at Carnegie Mellon University.

1 Introduction

The increasing reliance of individuals and organizations on the Web to help mediate a variety of activities is giving rise to a demand for richer security and privacy policies and more flexible mechanisms to enforce these policies. People may want to selectively expose sensitive information to others based on the evolving nature of their relationships, or share information about their activities under some conditions. This trend requires context-sensitive security and privacy policies, namely policies whose conditions are not tied to static considerations but rather conditions whose satisfaction, given the very same actors (or principals), will likely fluctuate over time. Enforcing such policies in open environments is particularly challenging for several reasons:

- Sources of information available to enforce these policies may vary from one principal to another (e.g. different users may have different sources of location tracking information made available through different cell phone operators);
- Available sources of information for the same principal may vary over time (e.g. when a user is on company premises her location may be obtained from the wireless LAN location tracking functionality operated by her company, but, when she is not, this information can possibly be obtained via her cell phone operator);

– Available sources of information may not be known ahead of time (e.g. new location tracking functionality may be installed or the user may roam into a new area).

Accordingly, enforcing context-sensitive policies in open domains requires the ability to opportunistically interleave policy reasoning with the dynamic identification, selection and access of relevant sources of contextual information. This requirement exceeds the capability of decentralized trust management infrastructures proposed so far and calls for privacy and security enforcing mechanisms capable of operating according to significantly less scripted scenarios than is the case today. It also calls for much richer service profiles than those found in early web service standards.

We introduce a semantic web framework and a meta-control model for dynamically interleaving policy reasoning and external service identification, selection and access. Within this framework, external sources of information are wrapped as web services with rich semantic profiles allowing for the dynamic discovery and comparison of relevant sources of information. While the framework is applicable to a number of domains where policy reasoning requires the automatic discovery and access of external sources of information (e.g. virtual/collaborative enterprise scenarios, coalition force scenarios, inter-agency homeland security collaboration scenarios), we look more particularly at the issue of enforcing privacy and security policies in pervasive computing environments. In this context, the owner of information sources (e.g. user, sensor, application, or organization) relies on one or more *Policy Enforcing Agents* (PEA) responsible for enforcing relevant privacy and security policies in response to incoming requests. These agents implement meta-control strategies to opportunistically interleave policy enforcement, semantic web reasoning and service discovery and access. The example used in this paper introduces one particular type of PEA we refer to as Information Disclosure Agents (IDA). These agents are responsible for enforcing two types of policies: access control policies and obfuscation policies. The latter are policies that manipulate the accuracy or inaccuracy with which information is released (e.g. disclosing whether someone is busy or not rather than disclosing what they are actually doing). The research reported herehas been conducted in the context of *MyCampus*, a pervasive computing environment aimed at enhancing everyday campus life at Carnegie Mellon University [7, 8, 19, 20].

The remainder of this paper is organized as follows. Section 2 provides a brief overview of relevant work in decentralized trust management and semantic web technologies. Section 3 introduces an *Information Disclosure Agent* architecture for enforcing privacy and security policies. It details its different modules and how their operations are opportunistically orchestrated by meta-control strategies in response to incoming requests. A motivating example is presented in Section 4. Section 5 details our meta-control model based on query status information. Operation of the architecture is illustrated in Section 6. Section 7 discusses our service discovery model. Section 8 presents our current implementation and discusses initial empirical results. Concluding remarks are provided in Section 9.

2 Related Work

The work presented in this paper builds on concepts of decentralized trust management developed over the past decade (see [3] as well as more recent research such as

[2,11,14]). Most recently, a number of researchers have started to explore opportunities for leveraging the openness and expressive power associated with semantic web frameworks in support of decentralized trust management (e.g. [1, 4, 9, 12, 13, 23, 24] to name just a few). Our own work in this area has involved the development of semantic web reasoning engines (or "Semantic e-Wallets") that enforce context-sensitive privacy and security policies in response to requests from context-aware applications implemented as intelligent agents [7, 8]. Semantic e-Wallets play a dual role of gatekeeper and clearinghouse for sources of information about a given entity (e.g. user, device, service or organization). In this paper, we introduce a more decentralized framework, where policies can be distributed among any number of agents and web services. The main contribution of the work discussed here is in the development and initial evaluation of a semantic web framework and a meta-control model for opportunistically interleaving policy reasoning and web service discovery in enforcing context-sensitive policies (e.g. privacy and security policies). This contrasts with the more scripted approaches to interleaving these two processes adopted in our earlier work on Semantic e-Wallets [7,8].

Our research builds on recent work on semantic web service languages, (e.g. OWL-S [26] and WSMO [27]) and semantic web service discovery functionality. Early work in this area by Paolucci et al. [28] focused on matching semantic descriptions of services being sought with semantic profiles of services being offered that include descriptions of input, output, preconditions and effects (see also our own work in this area [30]). More recently discovery functionality has also been proposed that takes into account security annotations [29].

Other relevant work includes languages for capturing user privacy preferences such as P3P's APPEL language [25], and for capturing access control privileges such as the Security Assertion Markup Language (SAML) [17], the XML Access Control Markup Language (XACML) [16] and the Enterprise Privacy Authorization Language (EPAL) [5]. These languages do not take advantage of semantic web concepts. On the other hand [12] describes a semantic web policy framework for distributed policy management. The framework allows policies to be described in terms of deontic concepts and speech acts. It has been used to encode security policies of web resources, agents and web services. Work by Uszok et al. has also resulted in the integration of KAoS policy services with semantic web services [24]. Our own work on Semantic e-Wallets as well as research described in this paper has relied on an extension of OWL Lite known as ROWL to represent security and privacy policies that refer to concepts defined with respect to OWL ontologies [7, 8]. While ROWL has been a convenient extension of OWL to represent and reason about rules, it is by no means the only available option. In fact, ROWL shares many traits with several other languages. One better known language in this area is RuleML [18], a proposed standard for a rule language, based on declarative logic programs. Another is SWRL [10], which uses OWL-DL to describe a subset of RuleML. The focus of the present paper is not on semantic web rule languages but rather on a semantic web framework and a meta-control model for enforcing context-sensitive policies. For the purpose of this paper, the reader can simply assume that the expressiveness of our own ROWL language is by and large similar to that of a language like SWRL with both languages supporting the combination of Horn-like rules with one or more OWL knowledge bases.

3 Overall Approach and Architecture

We consider an environment where sources of information are all modeled as services that can be automatically discovered based on rich ontology-based service profiles advertised in service directories. Each service has an owner, whether an individual or an organization, who is responsible for setting policies for it, with policies represented as rules. In this paper we focus on access control policies and obfuscation policies enforced by *Information Disclosure Agents*, though the framework we present could readily be used to enforce a variety of other policies.

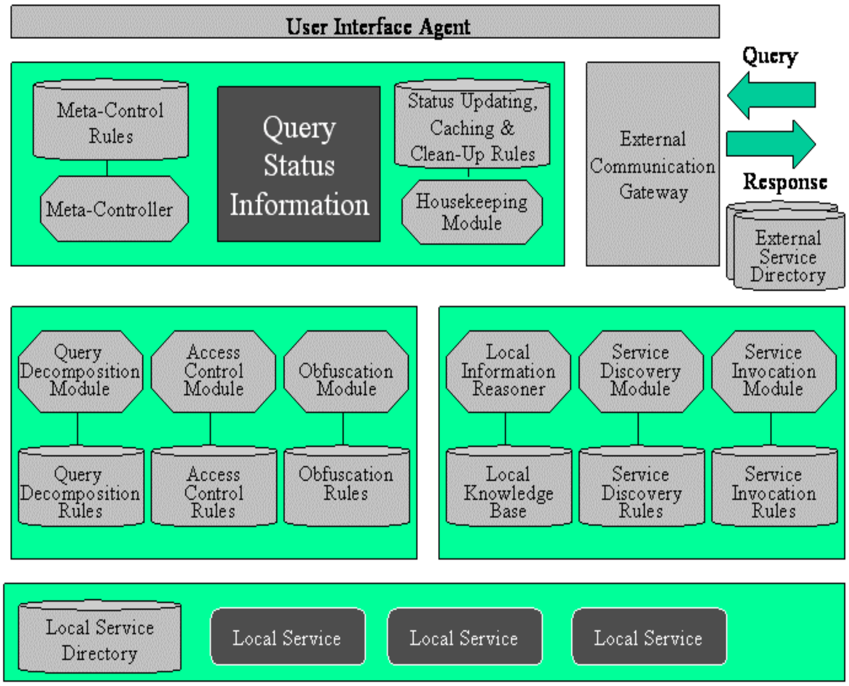


Fig. 1. Information Disclosure Agent: Overall Architecture

An Information Disclosure Agent (IDA) receives requests for information or service access. In processing these requests, it is responsible for enforcing access control and obfuscation polices specified by its owner and captured in the form of rules. As it processes incoming queries (or, more generally, requests), the agent records status information that helps it monitor its own progress in enforcing its policies and in obtaining the necessary information to satisfy the request. Based on this updated *query status information*, a meta-control module ("meta-controller") dynamically orchestrates the operations of modules it has at its disposal to process queries (Fig. 1). As these modules report on the status of activities they have been tasked to perform, this information is processed by a housekeeping module responsible for updating query status information (e.g. changing the status of a query from being processed to having been processed). Simply put, the agent continuously cycles through the following three basic steps:

1. The meta-controller analyzes its latest query status information and invokes one or more modules to perform particular tasks. As it invokes these modules the meta-controller also updates relevant query status information (e.g. updates the status of a query from “not yet processed” to “being processed”).
2. Modules complete their tasks (whether successfully or not) and report back to the housekeeping module – occasionally modules may also report on their ongoing progress in handling a task
3. The housekeeping module updates detailed status information based on information received from other modules and performs additional housekeeping activities (e.g. caching the results of recent requests to mitigate the effects of possible denial of service attacks, cleaning up status information that has become irrelevant, etc.)

For obvious efficiency reasons, while an IDA consists of a number of logical modules, each operating according to a particular set of rules, it is typically implemented as a single reasoning engine. In our current work we use JESS [6], a high-performance Java-based rule engine that supports both forward and backward chaining, the latter by reifying "needs for facts" as facts themselves, which in turn trigger forward-chaining rules. The following provides a brief description of each of the modules orchestrated by an IDA’s meta-controller:

– *Query Decomposition Module* takes as input a particular query and breaks it down into elementary needs for information, which can each be thought of as subgoals or sub-queries. We refer to these as *Query Elements*.
– *Access Control Module* is responsible for determining whether a particular query or sub-query is consistent with relevant access control policies – modeled as access control rules. While some policies can be checked just based on facts contained in the agent’s local knowledge base, many policies require obtaining information from a combination of both local and external sources. When this is the case, rather than immediately deciding whether or not to grant access to a query, the *Access Control Module* needs to request additional facts – also modeled as *Query Elements*.
– *Obfuscation Module* sanitizes information requested in a query according to relevant obfuscation policies – also modeled as rules. As it evaluates relevant obfuscation policies, this module too can post requests for additional *Query Elements.*
– *Local Information Reasoner* corresponds to domain knowledge (facts and rules) known locally to the IDA
– *Service Discovery Module* helps the IDA identify potential sources of information to complement its local knowledge. External services can be identified through external service directories (whether public or not), by communicating via the agent’s *External Communication Gateway*. Rather than relying solely on searching service directories, the service discovery module also allows for the specification of what we refer to as *service identification rules*. These rules directly map information needs on pre-specified services. An example of such rule might be: “when looking for my current activity, first try my calendar service”. When available, such rules can yield significant speedups, while allowing the module to revert to more general service directory searches when they fail. We currently assume that all service directories rely on OWL-S to advertise service profiles (see Section 7).

- *Service Invocation Module* allows the agent to invoke relevant services. It is important to note that, in our architecture, each service can have its own IDA. As requests are sent to services, their IDAs may in turn respond with requests for additional information to enforce their own policies.
- *User Interface Agent:* The meta-controller treats its user as just another module who is modeled both as a potential source of domain knowledge (e.g. to acquire relevant contextual information) as well as a potential source of meta-control knowledge (e.g. if a particular query element proves too difficult to locate, the user may be asked whether to stop looking - she could even be offered the option of making an assumption about the particular value of the query element).

Modules support one or more services that can each be invoked by the meta-controller along with relevant parameter values. For instance, the meta-controller may invoke the query decomposition module and request it to decompose a particular query; it may invoke the access control module and task it to proceed in evaluating access control policies relevant to a particular query; etc. In addition, meta-control strategies do not have to be sequential. For instance, it may be advantageous to implement strategies that enable the IDA to concurrently request the same or different facts from several services.

4 An Example

The following scenario will help illustrate how IDAs operate. Consider Mary and Bob, two colleagues who work for company X. They are both field technicians who constantly visit other companies. Mary's team changes from one day to the next depending on her assignment. Mary relies on an IDA to enforce her access control policies. In particular, she has specified that she is only willing to disclose the room that she is in to members of her team and only when they are in the same building.

Suppose that today Bob and Mary are on the same team. Bob is querying Mary's IDA to find out about her location. For the purpose of this scenario, we assume that Mary and Bob are visiting Company Y and are both in the same building at the time the query is issued. Both Bob and Mary have cell phone operators who can provide their locations at the level of the building they are in – but not at a finer level. Upon entering Company Y, Mary also registered with the company's location tracking service, which can track her at the room level. For the purpose of this scenario, we further assume that Mary's IDA needs to identify a service that can help it determine whether Bob is on her team. A discovery step helps identify a service operated by Company X (Bob and Mary's employer) that contains up-to-date information about teams of field technicians. This requires a directory with rich semantic service profiles, describing what each service does (e.g. type of information it can provide, level of accuracy or recency, etc.). To be interpretable by agents such as Mary's IDAs, these profiles also need to refer to concepts specified in shared ontologies (e.g. concepts such as projects, teams, days of the week, etc.). Once Mary's IDA has determined that Bob is on her team today, it proceeds to determine whether they are in the same building by asking Bob's IDA about the building he is in. Here Bob's IDA goes through a service discovery step of its own and determines that a location tracking service offered by his cell phone operator is adequate. Completion of the scenario involves a few additional steps of the same type. Note that in this scenario we have assumed that Mary's IDA trusts the location information returned by Bob's IDA. It is easy to imagine scenarios where her IDA would be better off looking for a completely independent source of information. It is also easy to see that these types of scenarios can lead to deadlocks. This is further discussed later in this paper.

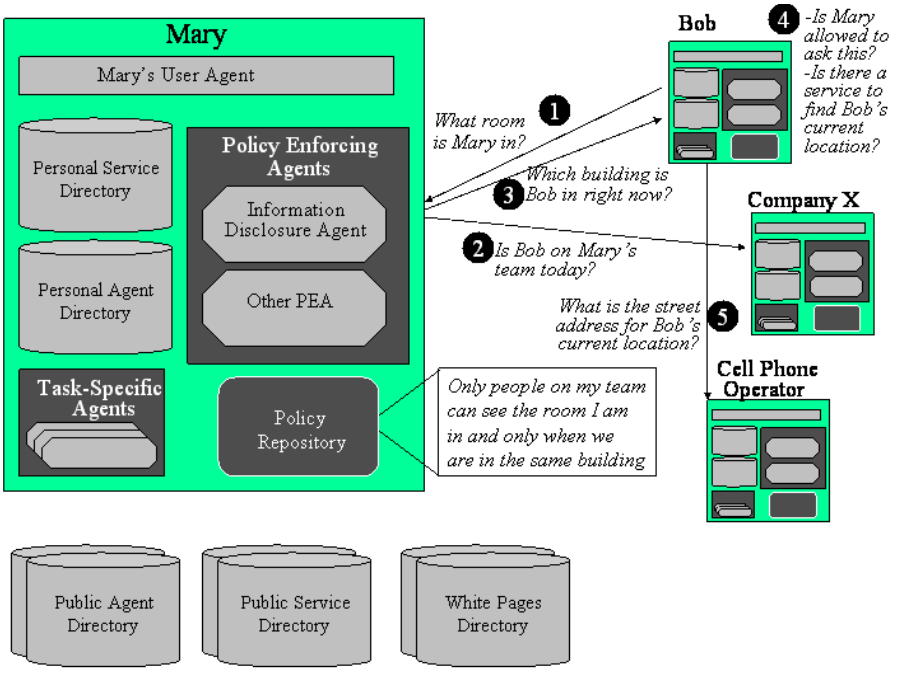

Mary
Mary's User Agent
Personal Service Directory
Personal Agent Directory
Policy Enforcing Agents
Information Disclosure Agent
Other PEA
Task-Specific Agents
Policy Repository
Only people on my team can see the room I am in and only when we are in the same building
1 What room is Mary in?
2 Is Bob on Mary's team today?
3 Which building is Bob in right now?
4 -Is Mary allowed to ask this? -Is there a service to find Bob's current location?
5 What is the street address for Bob's current location?
Bob
Company X
Cell Phone Operator
Public Agent Directory
Public Service Directory
White Pages Directory

Fig. 2. Illustration of first few steps involved in processing the example

5 Query Status Model

An IDA's *Meta Controller* relies on meta-control rules to analyze query status information and determine which module(s) to activate next. Meta-control rules are modeled as if-then clauses, with Left Hand Sides (LHSs) specifying their premises and Right Hand Sides (RHSs) their conclusions. LHS elements refer to query status information, while RHS elements contain facts that result in module activations. Query status information helps keep track of how far along the IDA is in obtaining the information required by each query and in enforcing relevant policies. Query status information in the LHS of meta-control rules is expressed according to a taxonomy of predicates that helps the agent keep track of queries and query elements - e.g., whether a query has been or is being processed, what individual query elements it has given rise to, whether these elements have been cleared by relevant access control policies and sanitized according to relevant obfuscation control policies, etc. All status information is annotated with time stamps. Specifically, query status information includes:

- **Status predicates** to describe the status of a query or query element
- **A query ID** or **query element ID** to which the predicate refers
- **A parent query ID** or **parent query element ID** to help keep track of dependencies (e.g. a query element may be needed to help check whether another query element is consistent with a context-sensitive access control policy). These dependencies, if passed between IDA agents, can also help detect deadlocks (e.g. two IDA agents each waiting for information from the other to enforce their policies)
- **A time stamp** that describes when the status information was generated or updated. This information is critical when it comes to determining how much time has elapsed since a particular module or external service was invoked. It can help the agent look for alternative external services or decide when to prompt the user (e.g. to decide whether to wait any longer).

A sample of query status predicates is provided in Table 1. Some of the predicates list in the Table will be used in Section 6, when we revisit the example introduced in Section 4. Clearly, different taxonomies of predicates can lead to more or less sophisticated meta-control strategies. For the sake of clarity, status predicates in Table 1 are organized in six categories: 1) communication; 2) query; 3) query elements; 4) access control; 5) obfuscation and 6) information collection.

Query status information is updated by asserting new facts (with old information being cleaned up by the IDA's housekeeping module). As query updates come in, they trigger one or more meta-control rules, which in turn result in additional query status information updates and the eventual activation of one or more of the IDA's modules. As already mentioned earlier, this meta-control architecture can also be used to model the user as a module that can be consulted by the meta-controller, e.g. to ask for a particular piece of domain knowledge or to decide whether or not to abandon a particular course of action such as looking for an external service capable of providing a particular query element.

6 Updating Query Status Information: Example Revisited

The following illustrates the processing of a query by an IDA, using the scenario introduced in Fig. 2. Specifically, Fig. 3 depicts some of the main steps involved in processing a request from Bob about the room Mary is in, highlighting some of the main query status information updates. Bob's query about the room Mary is in is first processed by the IDA's *Communication Gateway*, resulting in a query information status update indicating that a new query has been received. This information is expressed as a collection of *(predicate subject object)* triples of the form:

```
(triple "Status#predicate" "status1" "query-received")
(triple "Query#queryId" "status1" "query1")
(triple "Query#parentId" "status1" nil)
(triple "Query#timestamp" "querystatus1" "324455")
(triple "Query#sender" "query1" "bob")
(triple "Query#element" "query1" "element1")
(triple "Ontology#office" "mary" "element1")
```

Next, the meta-controller activates the *Query Decomposition Module,* resulting in the creation of two query elements – for the sake of simplicity we omit Mary's obfuscation policy: one query element to establish whether this request is compatible with Mary's access control policies and the other to obtain the room she is in:

Table 1. Sample list of status predicates

	Sample Status Predicates	**Description**
1)	Query-Received	A particular query has been received.
	Sending-Response	Response to a query is being sent
	Response-Sent	Response has been successfully sent
	Response-Failed	Response failed (e.g. message bounced back)
2)	Processing Query	Query is being processed
	Query Decomposed	Query has been decomposed (into primitive query elements)
	All-Elements-Available	All query elements associated with a given query are available (i.e. all the required information is available)
	All-Elements-Cleared	All query elements have been cleared by relevant access control policies
	Clearance-Failed	Failed to clear one or more access control policies
	All-Elements-Sanitized	All query elements have been sanitized according to relevant obfuscation policies
	Sanitization-Failed	Failed to pass one or more obfuscation policies
3)	Element-Needed	A query element is needed. Query elements may result from the decomposition of a query or may be needed to enforce policies. The query element's origin helps distinguish between these different cases
	Processing-Element	A need for a query element is being processed
	Element-Available	Query element is available
	Element-Cleared	Query element has been cleared by relevant access control policies
	Clearance-Failed	Failed to pass one or more access control policies
	Element-Sanitized	Query element has been sanitized using relevant obfuscation policies
	Sanitization-Failed	Failed to pass one or more obfuscation policies
4)	Clearance-Needed	A query or query element needs to be cleared by relevant access control rules
5)	Sanitization-Needed	Query or query element has to be sanitized subject to relevant obfuscation policies
6)	Check-Condition	Check whether a condition is satisfied. Special type of query element.
	Element-not-locally-available	The value of a query element can not be obtained from the local knowledge base
	Element-need-service	A query element requires the identification of a relevant service
	No-service-for-Element	No service could be identified to help answer a query element. This predicate can be refined to differentiate between different types of services (e.g. local versus external)
	Service-identified	One or more relevant services have been identified to help answer a query element
	Waiting-for-service-response	A query element is waiting for a response to a query sent to a service (e.g. query sent to a location tracking service to help answer a query element corresponding to a user's location)
	Failed-service-response	A service failed to provide a response. Again this predicate could be refined to distinguish between different types of failure (e.g. service down, access denied, etc.)
	service-response-available	A response has been returned by the service. This will typically result in the creation of an "Element-Available" status update.

```
(triple "Status#predicate" "status2" "clearance-needed")
(triple "Status#predicate" "status3" "element-needed")
```

Let us assume that the meta-controller decides to first focus on the "clearance-needed" query element and invokes the *Access Control Module.* This module determines that two conditions need to be checked and accordingly creates two new query elements ("check-conditions"). One condition requires checking whether Bob and Mary are on the same team:

```
(triple "Status#predicate" "status4" "element-needed")
(triple "Query#queryId" "status4" "element2")
(triple "Query#parentId" "status4" "query1")
(triple "Query#condition" "element2" "People#same-team")
(triple "People#same-team" "mary" "bob")
```

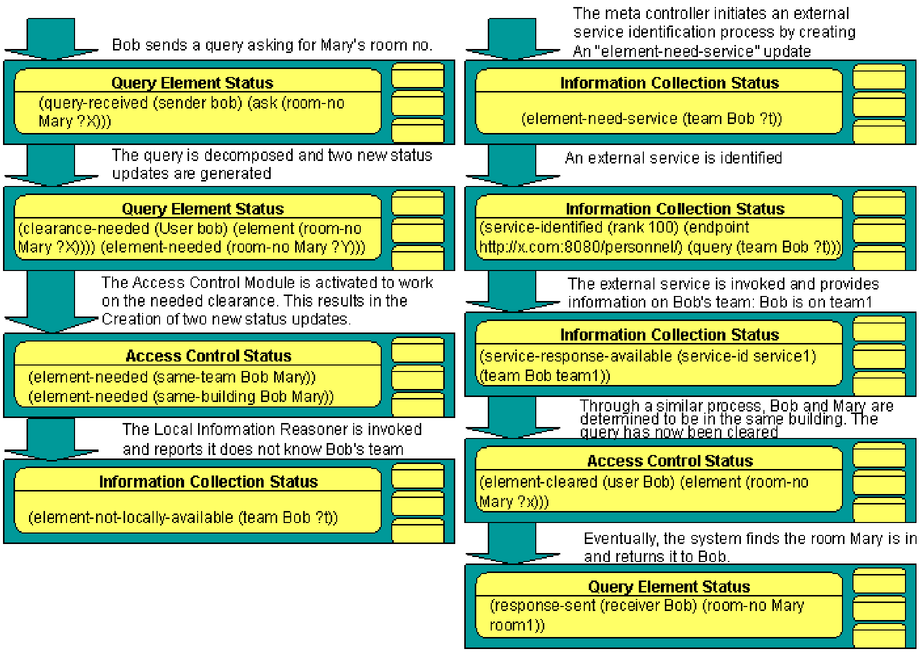

Bob sends a query asking for Mary's room no.
Query Element Status
(query-received (sender bob) (ask (room-no Mary ?X)))
The query is decomposed and two new status updates are generated
Query Element Status
(clearance-needed (User bob) (element (room-no Mary ?X)))) (element-needed (room-no Mary ?Y)))
The Access Control Module is activated to work on the needed clearance. This results in the Creation of two new status updates.
Access Control Status
(element-needed (same-team Bob Mary))
(element-needed (same-building Bob Mary))
The Local Information Reasoner is invoked and reports it does not know Bob's team
Information Collection Status
(element-not-locally-available (team Bob ?t))
The meta controller initiates an external service identification process by creating An "element-need-service" update
Information Collection Status
(element-need-service (team Bob ?t))
An external service is identified
Information Collection Status
(service-identified (rank 100) (endpoint http://x.com:8080/personnel/) (query (team Bob ?t)))
The external service is invoked and provides information on Bob's team: Bob is on team1
Information Collection Status
(service-response-available (service-id service1) (team Bob team1))
Through a similar process, Bob and Mary are determined to be in the same building. The query has now been cleared
Access Control Status
(element-cleared (user Bob) (element (room-no Mary ?x)))
Eventually, the system finds the room Mary is in and returns it to Bob.
Query Element Status
(response-sent (receiver Bob) (room-no Mary room1))

Fig. 3. Query status updates for a fragment of the scenario introduced in Fig 2

This condition in turn requires a series of information collection steps that are orchestrated by the meta-control rules in Mary's IDA. In this example, we assume that the IDA's local knowledge base knows which team Mary is on but not Bob. According the following query status information update is eventually generated:

```
(triple "Status#predicate" "status5" "element-not-locally-available")
(triple "Query#queryId" "status5" "element3")
(triple "Query#parentId" "status5" "element2")
(triple "People#team" "bob" "element3")
```

Mary's IDA has a meta-control rule to initiate service discovery when a query element can not be found locally. The rule, expressed in CLIPS [31], is of the form:

```
(triple "Status#predicate" ?s1 "element-not-locally-available")
(triple "Status#predicate" ?s2 "element-needed ")
(triple "Query#queryId" ?s1 ?e1)
(triple "Query#queryId" ?s2 ?e1)
=>
(assert (triple "predicate" ?newstatus "element-need-service"))
(assert (triple "Query#queryId" ?newstatus ?e1)
```

Using this rule, the meta-controller now activates the *Service Discovery Module.* A service to find Bob's team is identified (e.g. a service operated by company X). This results in a query status update of the type "service-identified".

```
(triple "Status#predicate" ?s1 "element-need-service")
(triple "Status#predicate" ?s2 "service-identified")
(triple "Query#queryId" ?s1 ?e1)
(triple "Query#queryId" ?s2 ?service)
(triple "Query#parentId" ?s2 ?e1)
=>
(assert (triple "Status#predicate" ?newstatus "waiting-for-service-
response"))
(assert (triple "Status#queryId" ?newstatus ?service))
```

Note that, if there are multiple matching services, the service discovery module needs rules to help select among them.

Let us assume that the service identified by the service discovery module is now invoked and that it returns the team that Bob is on. The Housekeeping module updates the necessary Query Status Information, indicating among other things that information about Bob's team has been found ("element-available") and cleaning old status information. This is done using a rule of the type:

```
?x <- (triple "Status#predicate" ?s1 "waiting-for-service-response")
?y <- (triple "Query#queryId" ?s1 ?service)
(triple "Status#predicate" ?s2 "service-response-available")
(triple "Query#queryId" ?s2 ?result)
=>
(retract ?x)
(retract ?y)
(assert (triple "Status#predicate" ?newstatus "element-available"))
(assert (triple "Query#queryId" ?newstatus ?result))
```

The scenario continues through several similar steps (see Fig. 3)

7 The Service Discovery Model

A central element of our architecture is the ability of IDA agents to dynamically identify sources of information needed by query elements. Sources of information are modeled as semantic web services and may operate subject to their own access control and obfuscation policies enforced by their own IDA agents. Accordingly service invocation is itself implemented in the form of queries sent to a service's IDA agent.

Each service (or source of information) is described by a *ServiceProfile* in OWL-S [26]. In general, a *ServiceProfile* consists of three parts: (1) information about the provider of the service, (2) information about the service's functionality and (3) information about non-functional attributes [21]. Functional attributes include the service's inputs, outputs, preconditions and effects. Non-functional attributes are other properties such as accuracy, quality of service, price, location, etc. An example of a location tracking service operated on the premises of Company Y can be described as follows:

```
<profileHierarchy:InformationService rdf:ID="PositioningServ">
  <!-- reference to the service specification -->
  <service:presentedBy rdf:resource="&Serv;#PositioningServ"/>
  <profile:has_process rdf:resource="&Process;#PositionProc"/>
  <profile:serviceName Positioning_Service_in_Y />

  <!-- specification of quality rating for profile -->
  <profile:qualityRating>
    <profile:QualityRating rdf:ID="SERVQUAL">
      <profile:ratingName SERVQUAL />
      <profile:rating rdf:resource="&servqual;#Good"/>
    </profile:QualityRating>
  </profile:qualityRating>
  <profile:hasPrecondition rdf:resource="&Process;#LocateInCompanyY"/>
  <profile:hasOutput rdf:resource="&Process;#RoomNoOutput"/>
</profileHierarchy:InformationService>
```

When invoking a service it has identified, an IDA may opt to provide upfront all the input parameters required by that service or it may withhold one or more of these parameters. The latter option forces the service to request the missing input parameters from the IDA, thereby enabling the IDA to more fully determine whether the invoked service meets its policies. This option is however more computation and communication intensive.

Service outputs are represented as OWL classes, which play the role of a typing mechanism for concepts and resources. Using OWL also allows for some measure of semantic inference as part of the service discovery process. If an agent requires a service that produces as output a contextual attribute of a specific type, then all services that output the value of that attribute as a subtype are potential matches.

Service preconditions and effects are also used for service matching. For instance., the positioning service above has a precondition specifying that it is only available on company Y's premises.

8 Current Implementation: Evaluation and Discussion

Our policy enforcing agents are currently implemented in JESS, a high-performance rule-based engine in Java [6]. Domain knowledge, including service profiles, queries, access control policies and obfuscation policies are expressed in OWL [8]. As already indicated earlier ROWL the language we currently use to define rules that relate to ontologies could easily be replaced with languages such as RuleML, SWRL or some similar language. XSLT transformations are used to translate OWL facts and extensions of OWL (e.g. to model rules and queries) into CLIPS. Agent modules are organized as JESS modules. Currently all information exchange between agents is done in the clear and without digital signatures. In the future, we plan to use SSL or some equivalent protocol for all information exchange. This will include signing all queries and responses.

We have evaluated our solution on an IBM laptop with a 1.80GHz Pentium M CPU and 1.50GB of RAM. The laptop was running Windows XP Professional OS, Java SDK 1.4.1 and Jess 6.1. As part of the evaluation, we implemented the example introduced in Section 4 and 6, using a light-weight rule/fact set. The set included 22 rules and 178 facts and features a single semantic service directory with 50 services, each represented by 5 to 10 Jess rules. A breakdown of the CPU times required to process Bob's query is provided in the table below. For each module the table provides a cumulative CPU time, namely the sum of the CPU times of all invocations of that module in processing the query.

Module	*CPU time in millisecond*
Meta-Controller	28
Access-Controller	32
Local-KB	49
Service discovery / invocation	72
Total	181

While these results provide just one data point, they seem to suggest that our solution can be viewed as practical in at least some simple settings. It should be noted that our solution is not JESS-specific. At the same time, a significant number of experiments still need to be conducted to gain a more comprehensive understanding of the scalability of our approach. Other complex issues such as dealing with deadlocks or reasoning about provenance (i.e. possible conflicts of interest of information sources used to build a proof) and inconsistent policies also require significant additional work. Differentiating between situations where a policy has been shown not to be satisfied and situations where the agent has not yet been able to determine whether a policy is satisfied will likely call for differentiating between classical negation and "negation as failure". One possible solution here would be to use a framework such as SweetRules as an add-on to our semantic web reasoner [22].

9 Concluding Remarks

In this paper, we presented a semantic web framework for dynamically interleaving policy reasoning and external service discovery and access. Within this framework, external sources of information are wrapped as web services with rich semantic profiles allowing for the dynamic discovery and comparison of relevant sources of information. Each entity (e.g. user, sensor, application, or organization) relies on one or more *Policy Enforcing Agents* responsible for enforcing relevant privacy and security policies in response to incoming requests. These agents implement meta-control strategies to dynamically interleave semantic web reasoning, service discovery and access. These meta-control strategies can also be extended to treat the user as another source of information, e.g. to confirm whether a given fact holds or to provide meta-control guidance such as deciding when to abandon trying to determine whether a policy is satisfied.

The Information Disclosure Agent presented in this paper is just one instantiation of our more general concept of Policy Enforcing Agents (PEAs). Other policies (e.g. information collection policies, notification preference policies) will typically rely on slightly different sets of modules and different meta-control strategies, yet they could all be implemented using the same meta-control architecture and many of the same principles presented in this paper. In general, PEAs rely on a taxonomy of query information status predicates to monitor their own progress in processing incoming queries and enforcing relevant security and privacy policies. They use meta-control rules to decide which action to take next (e.g. decomposing queries, seeking local or external information, etc.). Preliminary evaluation of an early implementation of our framework seems encouraging. At the same time, it is easy to see that the generality of the framework also gives rise to a number of challenging issues. Future work will focus on exploring scalability issues, evaluating tradeoffs between the expressiveness of privacy and security policies we allow and associated computational and communication requirements. Other issues of particular interest include studying opportunities for concurrency (e.g. simultaneously accessing multiple web services), dealing with real-time meta-control issues (e.g. deciding when to give up or when to look for additional sources of information/web services), breaking deadlocks [15], and integrating the user as a source of information.

Acknowledgements

The work reported herein has been supported in part under DARPA contract F30602-02-2-0035 ("DAML initiative") and in part under ARO research grant D20D19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab.

This research has also benefited from interactions with Lujo Bauer, Lorrie Cranor, Fabien Gandon, Jason Hong, Bruce McLaren, Mike Reiter and Peter Steenkiste.

References

1. R. Ashri, T. Payne, D. Marvin, M. Surridge and S. Taylor, Towards a Semantic Web Security Infrastructure. In Proceedings of Semantic Web Services Symposium, 2004.
2. L. Bauer, M.A. Schneider and E.W. Felten. "A General and Flexible Access Control System for the Web", In Proceedings of the 11th USENIX Security Symposium, August 2002.
3. M. Blaze, J. Feigenbaum, an J. Lacy. "Decentralized Trust Management". In Proceedings of IEEE Conference on Security and Privacy. Oakland, CA. May 1996.
4. L. Ding, P. Kolari, T. Finin, A. Joshi, Y. Peng and Y. Yesha. On Homeland Security and the Semantic Web: A Provenance and Trust Aware Inference Framework, In Proceedings of the AAAI Spring Symposium on AI Technologies for Homeland Security, 2005.
5. IBM, EPAL 1.1. http://www.zurich.ibm.com/security/enterprise-privacy/epal/.
6. E. Friedman-Hill. Jess in Action: Java Rule-based Systems, Manning Publications Company, June 2003, ISBN 1930110898, http://herzberg.ca.sandia.gov/jess/
7. F. Gandon, and N. Sadeh. A semantic e-wallet to reconcile privacy and context awareness. In Proceedings of the Second International Semantic Web Conference (ISWC03), 2003.
8. F. Gandon, and N. Sadeh. Semantic web technologies to reconcile privacy and context awareness. Web Semantics Journal, 1(3), 2004.
9. R. Hull, B. Kumar, D. Lieuwen, P. Patel-Schneider, A. Sahuguet, S. Varadarajan, and A. Vyas. Enabling context-aware and privacy-conscious user data sharing. In Proceedings of 2004 IEEE International Conference on Mobile Data Management, January 2004.
10. I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, SWRL: Semantic Web Rule Language Combining OWL and RuleML. Version 0.6.
11. T. van der Horst, T. Sundelin, K. E. Seamons, and C. D. Knutson. Mobile Trust Negotiation: Authentication and Authorization in Dynamic Mobile Networks. Eighth IFIP Conference on Communications and Multimedia Security, Lake Windermere, England, 2004
12. L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. IEEE 4th International Workshop on Policies for Distributed Systems and Networks, 2003
13. L. Kagal, M. Paolucci, N. Srinivasan, G. Denker, T. Finin and K. Sycara, Authorization and Privacy for Semantic Web Services, In Proceedings of Semantic Web Services Symposium, AAAI 2004 Spring Symposium Series, Stanford University, California, March 2004.
14. L.Bauer, S. Garriss, J. McCune, M.K. Reiter, J. Rouse, and P Rutenbar, "Device-Enabled Authorization in the Grey System", Submitted to USENIX Security 2005.
15. T. Leithead, W. Nejdl, D. Olmedilla, K. Seamons, M. Winslett, T. Yu, and C. Zhang, How to Exploit Ontologies in Trust Negotiation. Workshop on Trust, Security, and Reputation on the Semantic Web, part of ISWC04, Hiroshima, Japan, November 2004.
16. OASIS, eXtensible Access Control Markup Language (XACML)
17. OASIS, Security Assertion Markup Language (SAML)
18. The Rule Markup Initiative. (http://www.ruleml.org)
19. N. M. Sadeh, T.C. Chan, L. Van, O. Kwon, and K. Takizawa. Creating an open agent environment for context-aware m-commerce. In Agentcities: Challenges in Open Agent Environments, 2003.

20. N.M. Sadeh, F. Gandon, and Oh Byung Kwon. Ambient Intelligence: The MyCampus Experience. Carnegie Mellon University Technical Report. CMU-ISRI-05-123. June 2005.
21. J. O'Sullivan, D. Edmond, and A.T. Hofstede. What's in a service? Towards accurate description of non-functional service properties. Distributedand Parallel Databases, 12:117.133, 2002.
22. SweetRules. http://sweetrules.projects.semwebcentral.org/
23. J. Undercoffer, F. Perich, A .Cedilnik, L. Kagal, and A. Joshi. A secure infrastructure for service discovery and access in pervasive computing. ACM Monet: Special Issue on Security in Mobile Computing Environments, October 2003
24. A. Uszok, J. M. Bradshaw, R. Jeffers, M. Johnson, A. Tate, J. Dalton and S. Aitken, Policy and Contract Management for Semantic Web Services. In Proceedings of Semantic Web Services Symposium, AAAI 2004 Spring Symposium Series, Stanford California.
25. A P3P Preference Exchange Language(APPEL1.0) http://www.w3.org/TR/P3P-preferences/
26. OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S
27. Web Service Modeling Ontology, WSMO. http://www.wsmo.org/
28. M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara, Semantic Matching of Web Services Capabilities, In Proceedings of the First Intl Semantic Web Conference, 2002.
29. G. Denker, L. Kagal, T. Finin, M. Paolucci and K. Sycara, Security For DAML Web Services: Annotation and Matchmaking, In Proceedings of the Second Intl Semantic Web Conference, 2003.
30. J. Rao. Semantic Web Service Composition via Logic-based Program Synthesis. PhD Thesis. Norwegian University of Science and Technology. December 10, 2004.
31. CLIPS. http://www.ghg.net/clips/CLIPS.html.

Reactive Rules-Based Dependency Resolution for Monitoring Dynamic Environments

Dagan Gilat, Royi Ronen, Ron Rothblum, Guy Sharon, and Inna Skarbovsky

IBM Haifa Research Laboratory
{dagang,royi,ron,guysh,inna}@il.ibm.com

Abstract. Monitoring systems commonly use data dependencies and are very often required to have real-time, or near real-time, capabilities. Resolution of dependencies using a reactive rule engine is an evident choice, since it provides inherent real-time characteristics.
We introduce the novel approach taken by Active Dependency Integration (ADI) technology in using reactive rules for dependency resolution, i.e., for the purpose of calculating an updated value using the value elements on which it depends. The salient property of this approach is that it demonstrates autonomic behavior. The set of reactive rules used for dependency resolution does not depend on the model for which it provides dependency resolution. The same rules handle every dependency model and support dynamic models, where elements may be added or deleted, without having to change any code or rule definitions, or stop the monitoring for manual system reconfiguration and redeployment. The rules are implemented in AMIT, an event-driven rule engine.

1 Introduction

Dependencies of various types are common in monitoring systems. A dependency expresses the exact way in which the value of a data element (i.e., the dependency target) is affected by other elements (i.e., the dependency sources). For example, a metric representing the availability of a website is a value that depends on other values, such as the availability of the site's server and communication lines. For users who define an ontology that describes the impact of elements on other elements, the use of dependencies is more natural and intuitive than the use of reactive rules, since it provides a higher level of abstraction [2].

Dependencies have to be resolved in run-time. A possible mechanism for this resolution is a set of reactive rules. Unlike traditional queries that return an answer only when (and if) activated, a reactive rule functions as a watchdog that sends an alert whenever the conditions associated with it are satisfied. Reactive rules have inherent real-time characteristics [1], which make them a good choice for the task, since many monitoring systems have real-time requirements [6].

Dilman and Raz [7] distinguish between two types of monitoring: statistical and reactive. Statistical monitoring provides statistical properties of the system's behavior, while reactive monitoring immediately reacts to conditions that develop in the system. Reactive monitoring is therefore suitable for applications in which a timely alert is important. Domains with a need for reactive monitoring applications include security applications, safety applications, sensor monitoring applications with real-time (or near real-time) requirements, and financial real-time applications [4] [6] [1]. These domains all share the need for an immediate alert in response to developments in the monitored object and thus motivate the use of reactive rules for dependency resolution. Carney et al. [6] proposed the DBMS-Active, Human-Passive (DAHP) model for such systems. DAHP systems are database systems that do not get their data from humans issuing transactions, but from external sources. Users do not query data, but alerts are emitted when needed. Monitoring systems can be looked at as DAHP systems that evaluate continuous queries. The monitored value is the query output, actively brought to the passive user.

1.1 ADI

ADI is a language developed by IBM Research to model data dependencies between entities. Entities can also be affected by events, which are the input to ADI models. ADI uses AMIT—an event driven reactive rule engine—for the task of dependency resolution. In this paper, we introduce the novel approach taken by ADI in using AMIT for dependency resolution. The authors cover those AMIT and ADI issues that are important for the understanding of this paper. Further details for both ADI and AMIT are available in [2] and [1], respectively.

An ADI Model Example. This section informally describes an ADI model example. Section 1.2 formally discusses dependency resolution. Consider a website that provides online stock trading services. The trading process is monitored. It is composed of: Transaction tasks (buy, sell), View Portfolio, Login and Logout. The first two are operated by a trading server and the latter two are operated by an authentication server. These dependencies are modeled in ADI using a *mandatory dependency* between the services and the related servers, as illustrated in Figure 1. Login and Transaction tasks are crucial for the proper functioning of the site. Therefore, in the ADI model, the site depends on them in a *mandatory dependency*, which means that the site fails to function if at least one of them fails. Logout and View Portfolio are also important, but the site can function even if at most one of them fails. Therefore, the modeling requires *one-out-of dependency*, whose positive result is *mandatory* for the site. The trading and authentication servers depend, in a *mandatory* manner, on a DB server and on *two-out-of* three WASes.

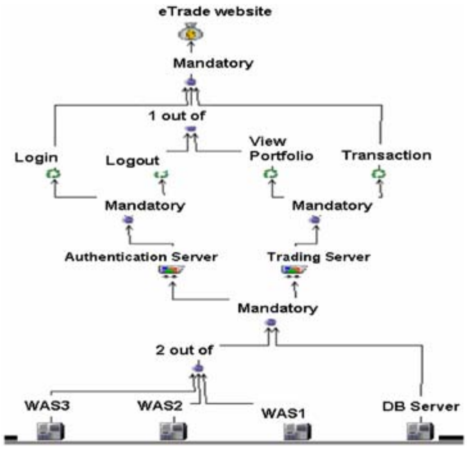


Fig. 1.

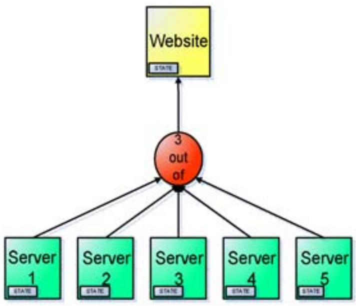


Fig. 2.

Self-management in ADI. Intuitively, one expects that different sets of reactive rules will be arranged and deployed for different dependency models. In the case of dynamic models where the topology changes, rules will have to be added or deleted accordingly in order to comply with the new topology. For such a solution to work, it is imperative that a mechanism for adding and deleting rules exists and that loss of context is not implied by such changes. Change in rules may also be an expensive, time consuming, action due to synchronization needs in the rule engine; this drawback can gravely harm the engine's adequacy for real-time purposes. Monitoring systems usually deal with massive amounts of input that can also provide information on changes in the model topology. This input, in turn, may necessitate frequent and expensive changes in rules. To enable the use of reactive rules for real-time or semi real-time monitoring, we need a set of rules that will not have to be changed during execution of the monitoring, even if the monitored model does change. Such a set of rules will enable the autonomic management of monitoring the changing model, since it will never have to undergo external modifications, despite changes in topology. Moreover, this approach does not require the existence of rule modification capabilities in the reactive rules engine.

In ADI, the same set of reactive rules is used for dependency resolution in any dependency model, including dynamic models whose topology is not fixed. The rules have a mechanism that imprints the initial model and runtime changes in them, in addition to performing the dependency resolution itself. The ability to imprint every ADI model, including runtime changes, is the key to the system's property of self management.

1.2 Dependency Resolution

For the sake of simplicity, our discussion uses the example depicted in Figure 2, which is simpler than the example presented in Section 1.1. The simpler model in Figure 2 monitors the availability of a website. The target entity, named "Website" represents the website and its state attribute expresses the state of the site's availability. The site has five servers, where two of them are for backup purposes. Each of the servers is represented as a source entity with a state attribute. The possible values for state attributes are: OK, WARNING, PROBLEM, and FAIL (This restriction and another one are discussed at the end of this section). The goal of this simple model is to monitor the state of the "Website" entity, which depends on other values in the model. The attribute state of "Website" must be updated according to the values in the state attributes of its five source entities. We use the semantics of an n-out-of dependency, where n=3. Using n-out-of, the state of the target entity is the worst state of the best n source values. The n-out-of semantics are appropriate for this example since the website is assumed to choose the three servers that have the best state out of the available five. The website's state of availability will be the worst state of the three chosen servers, namely the worst out of the best three (recall that n=3). Alternatively, n-out-of chooses the state of the nth entity under the total order {FAIL < PROBLEM < WARNING < OK}, where n elements are counted from OK "downwards". For example, if the states of the servers are {OK, WARNING, OK, PROBLEM, FAIL}, the website will be in the WARNING state. The resolution of the dependency is the process of deriving WARNING from the multiset of states, according to the dependency semantics.

Requiring that the set of possible values be a finite known set is crucial to the proposed resolution algorithm. A similar restriction was proposed in the context of memory limitations for continuous queries [3]. This restriction enables a synopsis of data from an unbounded number of dependency sources, as explained in Section 2.3. This restriction renders our solution suitable either for dependencies with a potentially unbounded, unknown number of sources reporting values from a finite set, or for dependencies with a finite, known number of sources reporting any value. More details related to this issue appear in Section 4.

1.3 Structure of this Paper

In Section 2, we show how AMIT reactive rules are used to resolve dependencies in a dynamic model, with the input in the form of events. The monitoring system receives events that may update the attributes of each of the sources and may influence the topology of the model, making the model dynamic. Our main focus is on state dependencies that derive the target's state from the state of its sources. Modifications needed for the resolution of other dependencies are also discussed, with the example of arithmetic dependencies. In Section 3 we discuss related work and Section 4 concludes the paper.

2 Reactive Rules for Dependency Resolution

ADI uses AMIT as an internal component in charge of dependency resolution. ADI informs AMIT of changes in the model topology and changes in the values of existing entities. AMIT then sends ADI events with resolved values of dependant entities. Figure 3 depicts the relationship between AMIT and ADI. Section 2.1 gives a concise overview of AMIT. Sections 2.2 through 2.4 delve into its use by ADI.

2.1 AMIT

The AMIT situation manager is the engine's component in charge of detecting situations (i.e., complex events), which are essentially correlations between events. The AMIT user, in our case ADI, defines patterns of complex events that are of interest, and the situation manager emits an event when the pattern is detected. An operator is defined as the type of pattern of events and the operands are the events on which the pattern is defined. Each situation has a single operator and one or more operands.

The following are some usage examples of operators:

- Operator "All": The situation using this operator is detected when all its operands occurred.
- Operator "Sequence": The situation using this operator is detected when all its operands occurred in the defined order.
- Operator "At Least n": The situation using this operator is detected when at least n instances of the operands (not necessarily different operands) occurred.

The situation manger uses *keys*. A key defines how operand instances are partitioned for the purpose of situation detection, by specifying the context of detection. Partitioning is done on the basis of equality between attributes that participate in the key. In the presence of a key, patterns will only be detected over instances with the same value for attributes that participate in the key. For example, consider the definition *All(e1,e2).* This situation's semantic is as follows: the situation is detected when both e1 and e2 are detected, in any order. Now, assume that e1 and e2 have both an attribute, named a1. In the case where no key is defined for the situation, all the detected instances of e1 and e2 will be considered, regardless of the value of a1 in any of the events. However, in the presence of a key in which e1.a1 (the attribute a1 of event e1) and e2.a1 participate, only those instances with the same value for the two a1 attributes will be considered. An instance of e1 with a1=4 and an instance of e2 with a1=6 do not result in a situation detection in the presence of the above defined key.

The presented set of AMIT rules uses event references. An event reference is a mechanism that produces an event as an action triggered by the detection of another event, with a possible condition. It resembles the ECA (event – condition – action) model [14], where the action is an event emission. For example, if e3 references e4 under the condition e3.a1>9, then the detection of e3 with a1>9 will result in the emission of e4, and both instances will be considered for the detection of situations.

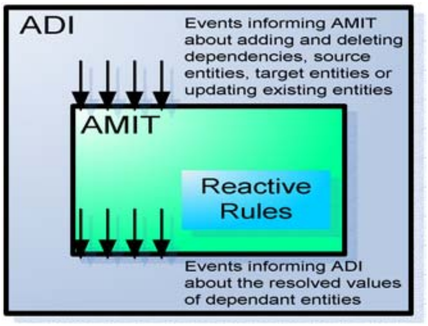

ADI
Events informing AMIT about adding and deleting dependencies, source entities, target entities or updating existing entities
AMIT
Reactive Rules
Events informing ADI about the resolved values of dependant entities

Fig. 3.

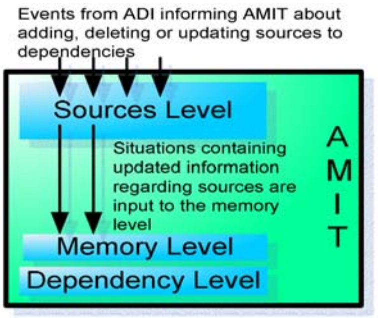

Events from ADI informing AMIT about adding, deleting or updating sources to dependencies
Sources Level
Situations containing updated information regarding sources are input to the memory level
Memory Level
Dependency Level
AMIT

Fig. 4.

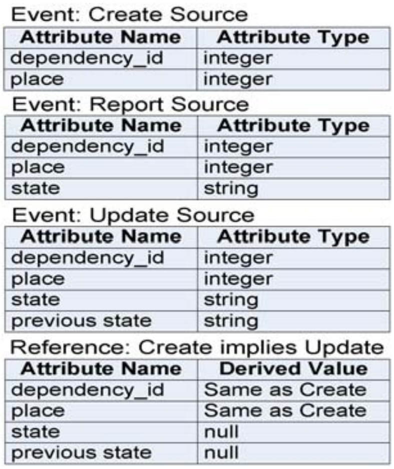
In AMIT, every situation is also an input event. Therefore, a situation can play the role of an event in every part of the language. A comprehensive description of the situation manager language can be found in [1]. Our proposed paradigm of dependency resolution using reactive rules implemented in AMIT can be seen as composed of three levels of AMIT rules: the sources level, the memory level, and the dependency level. The following sections review these three levels, using the example depicted in Figure 2.

2.2 Sources Level

The sources level is responsible for reporting the creation, deletion, and change in status of sources to the memory level. Figure 4 shows the input and output of this level. The sources level events, i.e., the events that it uses as input, are: ***createSource,***

reportSource, updateSource, and ***removeSource.*** The three uppermost tables in Figure 5 show the structure of the first three events. The event ***removeSource*** has the same structure as ***createSource.***

ADI sends these events to AMIT, which uses them to track the changes in both the model's topology and in values of existing entities. Whenever a new source to a dependency is created, ADI sends AMIT a ***createSource*** event. Its attribute *dependency_id* expresses the dependency to which the new source belongs and the *place* attribute expresses the location of the source in the dependency relative to other sources. ***removeSource*** informs AMIT about a deletion of a source from a dependency. ***reportSource*** is sent by ADI to report the latest value of an attribute that belongs to an already existing entity. A change in value always results in an event informing AMIT of the change. ***createSource*** references (i.e., causes the triggering of) an ***updateSource*** event (see Section 2.1), whose *dependency_id* and *place* attributes have the same values as ***createSource***.


Event: Create Source

Attribute Name	Attribute Type
dependency_id	integer
place	integer

Event: Report Source

Attribute Name	Attribute Type
dependency_id	integer
place	integer
state	string

Event: Update Source

Attribute Name	Attribute Type
dependency_id	integer
place	integer
state	string
previous state	string

Reference: Create implies Update

Attribute Name	Derived Value
dependency_id	Same as Create
place	Same as Create
state	null
previous state	null

Fig. 5.

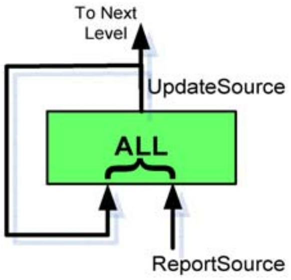


Fig. 6.

The lowermost table in Figure 5 shows the structure of the referenced ***updateSource*** event. As a result of the reference, every ***createSource*** event is immediately followed by an ***updateSource*** event with *null* values in *state* and *previous_state* attributes. After ***createSource***, ADI always sends a ***reportSource*** event with the state in which the entity was created.

The heart of the sources level is an "All" situation over ***reportSource*** and ***updateSource***. This situation is immediately detected and reported when both these events occur. The situation is keyed by two attributes, *place* and *dependency_id*. This means that only pairs of ***updateSource*** and ***reportSource*** events that agree on the values in the two attributes *dependency_id* and *place* will be considered for the detection of the "All" situation. Since ***createSource*** references ***updateSource***, the first occurrence of ***reportSource*** that refers a certain source results in the detection of the "All" situation over ***updateSource*** (referenced) and ***reportSource*** (occurred).

The sources level situation (i.e., the "All" situation just discussed), in what may seem surprising, is the ***updateSource*** event itself. This is perhaps the most non-intuitive idea in the design of the rules. Up to this point in the discussion, ***updateSource*** was an event and not a situation. As mentioned above, a situation in AMIT can play the role of an event in any context. We take advantage of this by defining ***updateSource*** to be a situation operand as illustrated in Figure 6. Initially, it looks as though such a situation can never be detected, since it must first occur in order to be detected. By referencing ***updateSource*** at the creation of the source, we enable the first detection of the situation. When the first ***reportSource*** event occurs, it results in a situation because an ***updateSource*** was referenced upon creation of the source. When a ***reportSource*** event other than the first occurs, it results in a situation as well, since the previous ***reportSource*** resulted in an ***updateSource*** situation. Section 2.3 discusses how the memory level makes use of this situation as input. The following is the XML definition of the situation, using the situation manager language. The elements are discussed in the order (depth-first) in which they appear.

```
<situation name="updateSource">
 <all>
       <operandAll eventType="reportSource"/>
       <operandAll eventType="updateSource"/>
       <keyBy name="place"/>
       <keyBy name="dependency_id"/>
 </all>
 <situationAttribute attributeName="dependency_id" expression="key(dependency_id)"/>
 <situationAttribute attributeName="place" expression="reportSource.place"/>
 <situationAttribute attributeName="state" expression="reportSource.state"/>
 <situationAttribute attributeName="previous state" expression="updateSource.state"/>
</situation>
```

The main element, *situation*, has an attribute with the situation's name. Its first child, *all*, means the situation operator is an "All" type. The *operandAll* elements define the events that participate in the situation. *Keyby* elements define the keys according to which partition of the event will take place. Only pairs that agree on the values in both *place* and *dependency_id* attributes can result in a situation. *Situation-Attribute* elements define the attributes of the situation; the expressions beside them determine their values. The attributes must match the ***updateSource*** event structure (Figure 5). We can see that the detected situation has a *dependency_id* attribute taken from the key. The *place* attribute is taken from the ***reportSource*** operand, but could have been taken from the other key or from the other operand since they all have the exact same value. The *state* attribute is taken from the ***reportSource*** operand and the *previous_state* attribute from the ***updateSource*** operand. Note that *previous_state* stores the value that was received with the previous ***reportSource*** event. The memory level uses ***updateSource*** as input.

Before moving on to the memory level, we briefly summarize this section. The sources level is in charge of informing the memory level about changes in the states of the dependency's sources.

- ***reportSource*** events are correlated with ***updateSource*** events (or situations). Each such correlation is an ***updateSource*** situation that refers to a specific source in one dependency.
- Only operands that agree on *dependency_id* and *place* values are correlated.
- ***updateSource*** situation is a candidate for another correlation with the next ***reportSource*** event. The correlation purpose is another ***updateSource*** situation. Because the ***updateSource*** situation is needed for its own detection, it is referenced

by ***createSource*** (rather than detected) when the source is created. This enables the first ***reportSource*** event to be correlated with the referenced ***updateSource*** event, resulting in a detected (rather than referenced) ***updateSource*** situation that will be ready to be correlated with the next ***reportSource*** event and so on and so forth.

- The ***updateSource*** situation includes information on the current and the previous state of the source whose change in state it represents, as well as *place* and *dependency- id* information.
- In addition to being a situation, ***updateSource*** plays the role of an event when used as an operand. The rules define it both as an event and as a situation.

The notion of an "All" situation that is its own operand appears in the memory level as well. In both levels, it functions as a means to capture the previous state of some part of the system. The *updateSource* event/situation, which 'waits' for a *reportSource* event to be correlated with it, contains information about the source's current and previous states. Together with the *reportSource* event, the two events allow the creation of a new situation with the updated information.

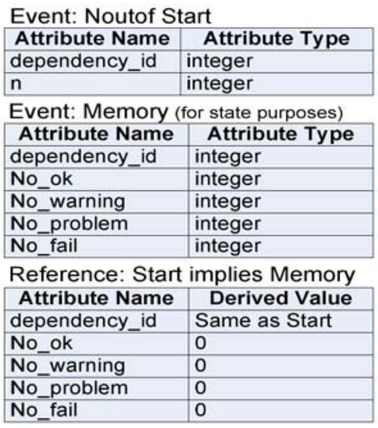
2.3 Memory Level

The memory level is in charge of the bookkeeping task for the sources of a single dependency. The memory level is, in fact, a situation. It keeps track of the sources' values as they get updated. Section 2.2 discusses the sources level by describing the events and situations associated with the changes in the state of a single source, since the key attributes *dependency_id* and *place* uniquely identify a source. We say that the memory level is one level above the sources level, because it receives its input events primarily from the sources level and because the situation is keyed only by the *dependency_id*. As a result, when the incoming events are partitioned for correlation, they no longer need to have the same value in the *place* attribute; *place* is ignored. Here, the partition is coarser. Events having the same *dependency_id* attribute are suitable for correlation in this level, regardless of the *place* attribute value.

We call this level's situation ***memory***. Dependency types that deal with states, such as *n-out-of* from the example in Figure 2, all have the same format. This section discusses the format extensively. Slight modifications for other types of dependencies are discussed in Section 2.5. The ***memory*** situation attributes also reflect the fact that they are one level above the sources level. ***memory*** has attributes comprising information about all the sources of a dependency. They are therefore affected by all sources, unlike the attributes of ***updateSource***, which refer to a single source and are not affected by other sources. When a new dependency is created, ADI sends AMIT a ***start*** event. Its structure is different for every dependency type. In the case of dependencies that need parameters, ***start*** is responsible for providing the initial parameters' values. For example, in the model in Figure 2, the ***start*** event for the *n-out-of* dependency has an attribute called *n* with the value of 3 to express that the specific instance of this dependency type is *3-out-of*, while the ***start*** event for most arithmetic dependencies does not have parameters. Figure 7 shows the ***memory*** event structure.

The idea of a situation that is its own operand is also used here. The ***memory*** event will function as its own operand, bringing the previous state of part of the system as input. At the sources level, we saw that an initial reference of such a situation is required for its first detection. From the first detection onwards, the reference is no longer needed since the first detection enables further detections. The same happens at the memory level, where the cycle begins with a reference of ***memory*** by ***start***. We see that ***memory*** has an integer attribute for each of the possible states; these attributes function as counters for the number of sources in each state.

The ***start*** event triggers a ***memory*** event and sets its attribute values as defined in Figure 7.



Event: Noutof Start

Attribute Name	Attribute Type
dependency_id	integer
n	integer

Event: Memory (for state purposes)

Attribute Name	Attribute Type
dependency_id	integer
No_ok	integer
No_warning	integer
No_problem	integer
No_fail	integer

Reference: Start implies Memory

Attribute Name	Derived Value
dependency_id	Same as Start
No_ok	0
No_warning	0
No_problem	0
No_fail	0

Fig. 7.

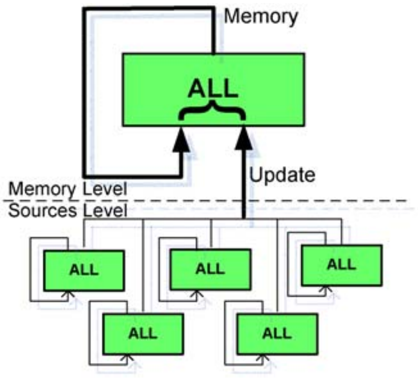


Fig. 8.

Up to this point, there was no connection between the two levels. Now, we see that ***memory*** is an "All" situation over ***updateSource*** from the sources level, as well as over itself. Figure 8 illustrates the skeleton of the memory situation and its relation to the sources level. The ***updateSource*** situations are the product of the sources level and the input to the memory level. The following is the detailed XML definition of the ***memory*** situation. The ***memory*** situation definition uses the "All" operator over the discussed events, namely ***update*** and ***memory***. For convenience, in the remainder of the situation definition, operands are referenced as **u** and **m** respectively, which are the values of the "as" attribute in the *operandAll* element.

```
<situation name="memory">
 <all>
     <operandAll eventType="updateSource" as="u"/>
     <operandAll eventType="memory" as="m"/>
     <keyBy name="dependency_id"/>
 </all>
 <situationAttribute attributeName="dependency_id"  expression="key(dependency_id)"/>
 <situationAttribute attributeName="No_ok" expression="if (u.STATE='state_ok') then (m.No_ok+1)
    elseif (u.PREV_STATE='state_ok') then (m.No_ok-1) else m.No_ok endif"/>
 <situationAttribute attributeName="No_warning" expression="if (u.STATE='state_warning')
    then (m.No_warning+1) elseif (u.PREV_STATE='state_warning') then (m.No_warning-1)
    else m.No_warning endif"/>
 <situationAttribute attributeName="No_problem" expression="if (u.STATE='state_problem')
    then (m.No_problem+1) elseif (u.PREV_STATE='state_problem') then (m.No_problem-1)
    else m.No_problem endif"/>
 <situationAttribute attributeName="No_fail" expression="if (u.STATE='state_fail') then (m.No_fail+1)
    elseif (u.PREV_STATE='state_fail') then (m.No_fail-1) else m.No_fail endif"/>
</situation>
```

Each of the four attributes of the situation: No_ok, No_warning, No_problem and No_fail is a counter for the number of sources for each of the four states. The following explanation describes the consequences, in view of the four counters, of adding a new source to a dependency. We assume that the dependency was already created and an initial ***memory*** situation was referenced setting the four counters to zero. When a source is added, an ***updateSource*** situation is referenced with null values for its *state* and *previous_state* attributes. This ***updateSource*** situation is correlated with the referenced ***memory*** situation to produce a new ***memory*** situation. The counters remain the same since we still don't know the state of the new source. Then, the first ***reportSource*** event related to this source is detected, producing an ***updateSource*** situation with new values. If the ***reportSource*** event changes the source's state from null to OK, the ***updateSource*** situation will have OK in the *state* attribute and null in the *previous_state* attribute. The ***memory*** situation uses this event as an operand, but this time, the No_ok attribute will be affected and will be incremented by one, to reflect that a new source was added to the dependency in state OK. Suppose another ***reportSource*** event occurs, changing the state of our single source from OK to PROBLEM. The sources level produces an ***updateSource*** situation with the values PROBLEM and OK in the *state* and *previous_state* attributes, respectively. This ***updateSource*** situation is correlated with the latest ***memory*** situation and a new ***memory*** situation is detected. Here, two attributes are affected: No_ok and No_problem. Because the *previous_state* attribute in the ***updateSource*** situation is OK, the second 'if' condition in the No_ok expression evaluates to true, and the value is decremented by one. The decrement reflects that there is now one source less in state OK. A similar expression associated with the No_problem attribute increments its value in the same way No_ok was incremented. The increment reflects that there is now one more source in state PROBLEM. We now return to Figure 2, where the dependency has five sources. Whenever one of them changes, two situations are detected. At the sources level, an ***updateSource*** situation is detected with the current and previous states. As a consequence, at the memory level, a ***memory*** situation is detected with attributes that represent the updated number of sources in each of the four states.

A source is removed by a ***removeSource*** event. ***removeSource*** references a ***reportSource*** event with null value for its *state* attribute. Should another source be created in the freed place, it will replace the removed source. In other words, the sources level associated with a specific place in the dependency is not terminated when the source is removed, but reused by the next source in the same place. Until now, no reference to the dependency semantics was made, except for noting that it is state related. The implementation of the semantics is done at the dependency level.

The following briefly summarizes the main points of the memory level.

- ***updateSource*** situations, whether referenced or detected. are correlated with ***memory*** situations. Each such correlation is a ***memory*** situation that refers to a single dependency.
- Operands have to have the same *dependency_id* in order to be correlated.
- Every ***memory*** situation is a candidate for another correlation with the next ***updateSource*** situation.
- A ***memory*** situation includes attributes for the number of sources in each of the four states. In dependencies that are not aimed at resolving a state, these attributes will be slightly different.

2.4 Dependency Level

The dependency level is in charge of deriving the final result, according to the semantics of the dependency. The dependency level is one level above the memory level and uses the ***memory*** situation, which is the memory level product, as its input. The dependency level consists of a simple "All" situation, whose operands are the ***memory*** situation and the ***start*** event discussed in Section 2.3.

Consider the 3-out-of dependency from Figure 2. According to what we have seen in Section 2.3, if the sources' states are: {OK, WARNING, OK, PROBLEM, FAIL}, then the attributes No_ok, No_warning, No_problem and No_fail will have the values 2,1,1,1, respectively. We resolve the dependency by looking for the state of the third best source. No_ok = 2, so the third best source is worse than OK. No_ok + No_warning =3, and since $3 \geq n$, the resolved state is warning. The following is the XML definition of the ***Noutof*** situation associated with the dependency level in the case of *n-out-of.*

```
<situation name="Noutof">
        <all detectionMode="immediate">
                    <operandAll eventType="Noutof Start" as="s" retain="true"/>
                    <operandAll eventType="memory" as="m">
                    <keyBy name="dependency_id"/>
        </all>
        <situationAttribute attributeName="dependency_id" expression="m.dependency_id)"/>
        <situationAttribute attributeName="STATE" expression="if ((m.No_ok-s.n)>=0) then 'state_ok'
          elseif (((m.No_ok+m.No_warning)-s.n)>=0) then 'state_warning'
          elseif (((m.No_ok+m.No_warning+m.No_problem)-s.n)>=0) then 'state_problem' else 'state_fail'
          endif"/>
</situation>
```

Note that the expression calculating the state attribute is the first point where the *n-out-of* semantics is of importance. The situation attribute expression performs the check to find the state of the nth best source, with respect to the n provided by the ***start*** event. With the creation of the dependency, ***start*** is sent once to AMIT. This is why the operand definition has a true value in *retain,* meaning the event is not consumed and thus available for further detections of the situation. The key is *dependency_id*, and is used to associate only corresponding operands, namely ***memory*** situations and ***start*** events that refer to the same dependency. The dependency resolution outcome is in the *state* attribute of the dependency level situation. In the example from Figure 2, this value is also the value monitored by our simple system and is available for use. Another situation will be detected if this value is changed.

2.5 Modifications for Other Types of Dependencies

This section deals with rule modifications that are required to implement dependencies other than *n-out-of.* Consider the ADI *mandatory* dependency type, where the resolved state value is the worst state value of all the sources. Bookkeeping of the number of sources in each state is done exactly as in n-out-of (sources and memory levels), but the dependency level is slightly different. Its *state* attribute expression checks for the first non-zero value in the *memory* situation state related attributes, in the order of FAIL, PROBLEM, WARNING, and OK. The first non-zero value implies the state. There are also dependency types that are not state related. One example is *sum*. In analogy to the *state* attribute that existed in each of the sources and in the target of any *n-out-of* dependency, the sources and target of a *sum* dependency all have an attribute named "value".

The semantics simply sum up the constantly changing values of the sources. In this case, the sources level remains almost the same, and still has to report the current and the previous values to the memory level, as it did with states. The memory level deducts the previous value and adds the current one. The dependency level is empty. New sources are created with a zero value and then updated with their initial values. To remove sources, their current value is set to zero. The *product* dependency type, whose semantics multiply the sources' values, functions almost the same. The difference is that in *product*, the memory level divides the total by the previous value and multiplies it by the current value. Sources are created with the multiplicity-neutral value, namely one. To remove sources, their current value is set to one.

3 Related Work

3.1 Dependency Related Work

Dependency discovery, studied in [9], involves the deduction of how elements affect each other. In our work, we assume that dependencies are already known and the focus is on how to use reactive rules to resolve the dependencies.

The work in [2] studies the ADI model language, how it expresses a model, its capabilities and expandability. This work provides a paradigm for the implementation of a dependency resolution mechanism.

The same difference exists between our work and the descriptions of the Mercury Dynamic Application Relationship Mapping [11] (which also discovers dependencies), the Micromuse NetCool [12], and Managed Objects' Formula [10].

In [1], a detailed description of the AMIT event algebra and context notation is provided, whereas this work uses that language as a tool.

3.2 Work Related to Monitoring

Communication effective monitoring is proposed in [7]. While in [7] dependency resolution is embodied in iterative procedures, this work studies the resolution itself and how reactive rules are used to implement it.

An ADI model can be looked upon as one or more continuous queries, of a procedural nature, over an event stream. The model topology dictates the processing procedure. Continuous queries are addressed in [13], in the context of append-only databases as an analogy to conventional queries on a regular database. Semantic issues and efficiency considerations regarding continuous queries over streams are addressed in [5]. The difference between [5] and [13] on the one hand, and ADI on the other, is that ADI-like data (i.e., dependencies) can be embodied in declarative SQL continuous queries. Our work addresses the issue of dependency models in view of resolution execution.

4 Conclusion

We propose a general paradigm for the implementation of dependency resolution using reactive rules. The rules are designed so there is no need to change them even if the dependency model changes, since the model is imprinted in the rules themselves. Three levels of data processing are used: information related to a single source in the first level, information related to all the sources in the second level, and processing related to the dependency semantics in the third level. AMIT rules using this approach are used by ADI for dependency resolution.

We reduced the dependency resolution problem of any model to the same fixed set of reactive rules. This requires that appropriate rules for each dependency *type* exist (i.e., one for *n-out-of*, one for *product* etc.), using the rule engine in a compact elegant manner.

The proposed approach offers a solution that does not limit the number of sources allowed for a dependency. This is made possible at the cost of restricting the possible values of dependency sources to a finite known set of values. Allowing any value as a source would require the number of sources to be bounded. Common monitoring system requirements are compliant with this restriction.

References

1. A.Adi and O. Etzion, "The situation manger rule language", *RuleML*, Sardinia, Italy. 2002 pp.36-57.
2. A. Adi, O. Etzion, D. Gilat, and G. Sharon, “Inference of Reactive Rules from Dependency Models”, *LNCS*, Springer-Verlag, Heidelberg, November 2003, Vol. 2876, pp. 49-64.
3. A. Arasu, B. Babcock, S. Babu, J. McAlister and J. Widom, "Characterizing Memory Requirements for Queries over Continuous Data Streams", *TODS*, 2004, vol. 29, pp. 162-194.
4. B. Babcock, S. Babu, M. Datar, R. Motwani and J. Widom, "Models and Issues in Data Stream Systems", *PODS*, Madison, Wisconsin, 2002, pp. 221-232.
5. S. Babu and J. Widom, "Continuous Queries over Data Streams, *SIGMOD Record*, 2001, vol. 30 (3), pp. 109-120.
6. D. Carney, U. Cetinternel, M. Cheriack, C Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul and S. Zodnik. "Monitoring streams – a new class of data management applications", *Brown Computer Science Technical Report*, TR-CS-02-04.
7. M. Dilman and D. Raz, "Efficient Reactive Monitoring", *INFOCOM*, Anchorage, Alaska, 2001, pp. 1012-1019.
8. A. Kochut and G. Kar, "Managing Virtual Storage Systems: An Approach Using Dependency Analysis", *Integrated Network Management* Colorado Springs, Colorado, 2003, pp. 593-604.
9. Managed Objetcs: http://www.managedobjects.com
10. Mercury: http://www.mercury.com
11. Micromuse: http://www.micromuse.com
12. D. Terry, D. Goldberg, D. Nichols and Brian Oki, "Continuous Queries over Append Only Databases", ***SIGMOD Int'l Conference on Management of Data,*** San Diego, California,1992, pp. 321-330.
13. J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California,1996.

Towards Discovery of Frequent Patterns in Description Logics with Rules

Joanna Józefowska, Agnieszka Ławrynowicz, and Tomasz Łukaszewski

Institute of Computing Science, Poznań University of Technology
ul. Piotrowo 3a, 60-965 Poznań, Poland
{joanna.jozefowska,agnieszka.lawrynowicz,tomasz.lukaszewski}
@cs.put.poznan.pl
http://www.cs.put.poznan.pl

Abstract. This paper follows the research direction that has received a growing interest recently, namely application of knowledge discovery methods to complex data representations. Among others, there have been methods proposed for learning in expressive, hybrid languages, combining relational component with terminological (description logics) component. In this paper we present a novel approach to frequent pattern discovery over the knowledge base represented in such a language, the combination of the basic subset of description logics with $\mathcal{DL}$-safe rules, that can be seen as a subset of Semantic Web Rule Language. Frequent patterns in our approach are represented as conjunctive $\mathcal{DL}$-safe queries over the hybrid knowledge base. We present also an illustrative example of our method based on the financial dataset.

1 Introduction

Discovery of frequent patterns has been investigated in many data mining settings and is now a well-researched area. An input to the existing data mining algorithms is usually a single table (so-called attribute-value representation). Recently there has been growing interest in applying knowledge discovery methods to more complex data representations. A good example of such approach is *relational data mining* (RDM) (Džeroski & Lavrač, 2001) from the field of *inductive logic programming* – ILP (Nienhuys-Cheng & de Wolf, 1997). RDM methods are designed to operate on multiple, linked tables to discover patterns involving multiple relations from a relational database. These methods have two significant advantages over the classical, propositional ones. Firstly, they operate on the original, not preprocessed relational datasets decreasing the risk of information loss. Secondly, as they use the formalisms of *logic programming*, they posses the ability to include *background knowledge* specific to the given domain into the knowledge discovery process. Background knowledge may be expressed for example in the form of rules. Because of the present efforts being made to move to meaningful systems, thus from databases to knowledge bases it may seem quite straightforward to transform the existing algorithms so that they were able to deal with data represented by ontologies. Exploiting ontologies as a kind of background knowledge can potentially improve the process and the results of knowledge discovery. It can be useful in driving the search process in the space of patterns and in the in-depth interpretation of discovered patterns. We refer particularly to the emerging Semantic Web (Berners-Lee *et al*, 2001), and the languages developed to represent the knowledge in Semantic Web (e.g. Web Ontology Language - *OWL*, McGuinness & van Harmelen, 2004) that are based on the well-researched knowledge representation formalisms of Description Logic (DL). Although the expressive power of DL in certain aspects goes far beyond the expressive power of logic programs, there exist relationships that can be easily expressed in a logic program (e.g. in the form of rules) but cannot be in turn expressed in DL. Thus the natural step was to extend the expressive abilities of ontologies by adding rules on top of ontologies that materialized in Semantic Web Rule Language *SWRL* (Horrocks *et al*, 2004). Whereas itemsets are the language of patterns in traditional frequent pattern mining setting, queries over DATALOG bases are the language of patterns in RDM setting, in our setting we propose the language of *conjunctive queries* (see 3.2) over the ontology expressed in the subset of *SWRL*. Moreover we use the intensional part of the ontology as background knowledge for the knowledge discovery task. In the scope of this paper and as a starting point of our investigations we consider the basic subset of *OWL*, *OWL*-DLP, known also as Description Logic Programs or *OWL*-Light, described in (Grosof *et al*, 2003) as an expressive intersection of the Semantic Web approaches to rules (RuleML Logic Programs) and ontologies (*OWL*-DL). As we are going to extent our work to more expressive *OWL* subsets, we describe our frequent pattern discovery task in the wider framework of the presented very recently (Motik *et al*, 2004) query answering mechanism for *OWL*-DL with function-free Horn rules, where rules are restricted to so-called $\mathcal{DL}$*-safe* ones (see 3.1) to preserve the *decidability* of such a combination.

The rest of this paper is organized as follows. In section 2 we introduce our learning task. In section 3 we introduce description logic that is the formalism of *OWL*-DLP language, $\mathcal{DL}$-safe rules and query answering for *OWL* with rules and we point out the representation formalisms of our patterns, background knowledge and instances. In section 4 we present our approach to discovery of frequent patterns. In section 5 we present the case study of our method based on experimental dataset. In Section 6 we compare our approach to related work. Section 7 concludes the paper.

2 Frequent Pattern Discovery Task

With respect to the general formulation of the frequent pattern discovery problem by Mannila and Toivonen (Mannila & Toivonen, 1997) and its further version for RDM (Dehaspe & Toivonen, 1999), we define our task of frequent pattern discovery over the knowledge base in *OWL*-DLP as:

Definition 1. Given

- a knowledge base in *OWL*-DLP with **DL**-safe rules **KB**,
- a set of patterns in the language $\mathcal{L}$ of queries Q that all contain a reference concept C_{ref},
- a minimum support treshold *minsup* specified by the user

and assuming that queries with support s are frequent in $\mathcal{KB}$ given C_{ref}, denoted as $support(C_{ref}, Q, \mathcal{KB})$, if $s \geq minsup$, the task of *frequent pattern discovery* is to find the set $\mathcal{F}$ of frequent queries.

In this definition, similarly to the RDM formulation, there is the C_{ref} parameter added to the frequent pattern discovery task which determines what is counted. A support of the query in our setting is defined as follows.

Definition 2. A *support* of the query Q with respect to the knowledge base KB is defined as the ratio between the number of instances of the C_{ref} concept that satisfy the query Q and the total number of instances of the C_{ref} concept (obtained as a result of submitting a trivial query denoted Q_{ref}):

$$support(C_{ref}, Q, KB) = \frac{|\,answerset(C_{ref}, Q, KB)\,|}{|\,answerset(C_{ref}, Q_{ref}, KB)\,|}$$

3 The Data Mining Setting

In our approach we assume that the knowledge base KB contains the terminological (*TBox*) and the assertional (*ABox*) part where the latter one is consistent with the former one (quite probably the terminological part was learnt from the assertional part or it was created manually). Thus we do not assume terminology learning from interpretations. Moreover we assume the presence of rules in our KB that generally can describe the relations impossible to be described by DL alone. In the case of our current setting, restricted to *OWL*-DLP, it should be noted however that *OWL*-DLP rules are expressible in DL. Our goal is to find frequent patterns in the form of conjunctive queries over KB where the search for patterns is ontology-guided.

In the following subsections we would like to describe in more detail the representation formalisms of patterns, knowledge base and instances. First, we are going to introduce the basic notions: the subset of DL of *OWL*-DLP, $\mathcal{DL}$-safe rules and query answering for conjunctive queries.

3.1 Preliminaries

OWL-DLP is the Horn fragment of *OWL*-DL i.e. we can say that *OWL*-DL statement is in DLP if it can be written, semantically equivalently, as a set of Horn clauses in first-order logic. We direct the reader to (Grosof *et al*, 2003) for more details about the bidirectional translation of premises and inferences from/to the *OWL*-DLP to/from Logic Programs. *OWL*-DLP has the desired property of polynomial data complexity and exponential time combined complexity. Referring to the practical definition from (Hitzler *et al*, 2005) that *an OWL-DL statement is in OWL-DLP if and only if some given transformation algorithm can rewrite it as a semantically equivalent Horn clause in first-order logic* and following their reference implementation KAON2[1] we define the *OWL*-DLP syntactic fragment used in our current work as follows (where a, b, a_i, o_i stand for individuals, C stands for a concept name, and R, Q, R_i, $Q_{i,j}$ stand for role names).

- *ABox*:

$C\,(a)$	(indiv. assertion)
$R\,(a, b)$	(property assertion)
$a = b$	(indiv. equivalence)

[1] http://kaon2.semanticweb.org

- Property Characteristics:

$R \equiv Q$	(equivalence)
$R \sqsubseteq Q$	(subproperty)
$\top \sqsubseteq \forall R.C \quad (C \not\equiv \bot)$	(domain)
$\top \sqsubseteq \forall R^{-}.C \, (C \not\equiv \bot)$	(range)
$R \equiv Q^{-}$	(inverse)
$R \equiv R^{-}$	(symmetry)
$\top \sqsubseteq \leq 1R$	(functionality)
$\top \sqsubseteq \leq 1R^{-}$	(inverseFunctionality)

- *TBox*: expressions of the form:
  $$\exists Q_{1,1}^{(-)} \ldots \exists Q_{1,m_1}^{(-)}.\mathtt{Left}_1 \sqcap \ldots \sqcap \exists Q_{k,1}^{(-)} \ldots \exists Q_{k,m_k}^{(-)}.\mathtt{Left}_k \sqsubseteq \forall R_1^{(-)} \ldots \forall R_n^{(-)}.\mathtt{Right}$$
  where $\mathtt{Left}_j$ can be of the forms C, $\{o_1, \ldots, o_n\}$, $\bot$ or $\top$, and $\mathtt{Right}$ can be of the forms C, $\top$, $\bot$ or $\{o\}$. The superscript $^{(-)}$ means that an inverse symbol may occur where indicated.

For rules, we use the following definitions. Given

- *NP*: a set of predicate symbols where each symbol is either a concept name or a role name,
- T : a set of *terms* where a term is either a constant (denoted by *a, b, c*) or a variable (denoted by *x, y, z*)
- *A*: a set of *atoms* having the form $P(s_1, \ldots, s_n)$, where P is a predicate symbol and s_i are terms

a *rule* has the form

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

where H and B_i are atoms and H is called the *rule head*, and the set of all B_i is called the *rule body*. The rule $H \leftarrow B_1, \ldots, B_n$ is equivalent to the clause $H \vee \neg B_1 \vee \ldots \vee \neg B_n$. A *program P* is a finite set of rules. Moreover for the scope of our work we restrict the atoms of rules to the concepts and roles existing in the terminology (that is to so-called *DL-atoms*) and equality and inequality statements, except those, implicitly added to rules to preserve $\mathcal{DL}$*-safety* as introduced next. Following the work of (Motik, Sattler and Studer, 2004) where the notion of $\mathcal{DL}$-safety and the reasoning algorithms have been introduced, we define $\mathcal{DL}$*-safe rules* as follows.

Definition 2. ($\mathcal{DL}$-safe rules) Let $\mathcal{KB}$-DL be a description logics knowledge base, and let *NP* be a set of predicate symbols where each symbol is either a concept name or a role name. A rule r is called $\mathcal{DL}$*-safe* if each variable in r occurs in a non-DL-atom in the rule body. A program P is $\mathcal{DL}$-safe if all its rules are $\mathcal{DL}$-safe.

The semantics of the combined knowledge base ($\mathcal{KB}$-DL, P) is given by translation into first-order logic as $\pi(\mathcal{KB}\text{-DL}) \cup P$. The main inference in ($\mathcal{KB}$-DL, P) is query answering, i.e. deciding whether $\pi(\mathcal{KB}\text{-DL}) \cup P \models \alpha$ for a ground atom α.

For the details of the transformation π we refer the reader to the (Motik, Sattler and Studer, 2004). We are going instead to explain some issues. $\mathcal{DL}$-safety is a notion similar to the safety in DATALOG where the rule is called safe when each variable occurs in a positive atom in the body which causes it to be bound to constants present in the database. $\mathcal{DL}$-safety works similarly □ makes sure that each variable is bound only to individuals present in the *ABox*. The rule can be made $\mathcal{DL}$-safe by adding special non-DL-literals in the form $\mathcal{O}(x)$ where x is a variable and by adding a fact $\mathcal{O}(a)$ for each individual a, which can be read as adding the phrase □where the identity of all objects is known□ to the meaning of the rule. For example the rule:

ForLoanPermanentOrderPayment $(x) \leftarrow$ Payment(x), paidFor(x, z), Loan(z), orderedFor(y, z), PermanentOrder(y), $\mathcal{O}(x)$, $\mathcal{O}(y)$, $\mathcal{O}(z)$

is $\mathcal{DL}$-safe.

3.2 Representation Formalisms for Frequent Pattern Discovery Task

In this subsection we define the representation formalism for our task of frequent pattern discovery. Background knowledge in our approach is represented as an *OWL*-DLP $\mathcal{KB}$ *TBox* with $\mathcal{DL}$-safe rules. Instances in our approach are assertions in *ABox*. Frequent patterns that we look for have the form of the conjunctive $\mathcal{DL}$-safe queries whose answer set contains individuals of the C_{ref} concept. In our work we adapt the definition of conjunctive query from (Hustadt, Motik and Sattler, 2004) to our restricted subsets of the languages of the $\mathcal{KB}$.

Definition 3. Let $\mathcal{KB}$ be an *OWL*-DLP with $\mathcal{DL}$-safe rules knowledge base, and let $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ be sets of distinguished and non-distinguished variables, denoted as **x** and **y**, respectively. A *conjunctive query* over $\mathcal{KB}$, written as $Q(\mathbf{x}, \mathbf{y})$, is a conjunction of DL-atoms of the form $A(s)$ or $R(s, t)$ for R an atomic role, and s and t individuals from $\mathcal{KB}$, distinguished or non-distinguished variables. The basic inferences are:

– *Query answering*. An answer of a query $Q(\mathbf{x}, \mathbf{y})$ w.r.t. $\mathcal{KB}$ is an assignment θ of individuals to distinguished variables, such that $\pi(\mathcal{KB}) \models \exists y : Q(\mathbf{x}\theta, \mathbf{y})$,
– *Query containment*. A query $Q_2(\mathbf{x}, \mathbf{y_1})$ is contained in a query $Q_1(\mathbf{x}, \mathbf{y_2})$ w.r.t. $\mathcal{KB}$ if $\pi(\mathcal{KB}) \models \forall \mathbf{x} : [\exists \mathbf{y_2} : Q_2(\mathbf{x}, \mathbf{y_2}) \rightarrow \exists \mathbf{y_1} : Q_1(\mathbf{x}, \mathbf{y_1})]$.

For the sake of clarity we will further use the following notation for queries:

$$q(key)\text{:-}C_{ref}(key), \alpha_1, \ldots, \alpha_n$$

where $q(key)$ denotes that *key* is the only distinguished one of query variables and $\alpha_1, \ldots, \alpha_n$ represent DL-atoms of the query.

With regard to our definition of frequent pattern discovery we look for the patterns containing the C_{ref} concept. We call them $\mathcal{K}$*-queries*.

Definition 4. Given the C_{ref} concept A, the $\mathcal{K}$*-query* is the conjunctive query that contains, among other atoms, the atom of the form either $A(key)$ or $C(key)$ in the body (in the latter case we assume having in the terminological part of the $\mathcal{KB}$ explicitly stated that $C \sqsubseteq A$) and where variable *key* is the distinguished variable.

A trivial pattern is the query of the form: $q(key)$:-C_{ref} (key). Moreover, we assume our queries to have linked-ness property (thus having all of its variables linked), similarly as it is defined in (Helft, 1987), and we assume all variables to be linked to the variable appearing in the literal with the C_{ref} concept. For example, for the *Client* being the C_{ref} concept the following $\mathcal{K}$-query can be imagined:

$q(key)$:- Client(key), isOwnerOf(key, x), Account(x), hasLoan(x, y), Loan(y), $\mathcal{O}(key)$, $\mathcal{O}(x)$, $\mathcal{O}(y)$

We assume all the queries to be $\mathcal{DL}$-safe and will not write in further examples the additional atoms of the form $\mathcal{O}(x)$.

4 The Levelwise Algorithm

The majority of existing approaches for frequent pattern mining adopts the levelwise method (Mannila&Toivonen, 1997) known from the APRIORI algorithm (Agrawal et al, 1996), where the space of patterns is searched one level at a time starting from the most general patterns. The pattern space is a lattice spanned by a specialization relation $\preceq$ between patterns, where $p1 \preceq p2$ denotes that pattern $p1$ is more general than pattern $p2$. The method iterates between the phase of *candidate generation*, where the lattice structure is used for pruning non-frequent patterns $\mathcal{I}$ from the given level, and the phase of *candidate evaluation* where support values of candidates are computed with respect to the database and the frequent patterns $\mathcal{F}$ are found. The lattice structure based on the specialization relation permits the algorithm to run intelligently across the space of patterns which in the other case would be very huge. The main algorithm of frequent pattern discovery is shown in Table 1.

Table 1. Main algorithm

```
mineFrequentPatterns($\mathcal{KB}$, $C_{ref}$, $minsup$)
 1. $l \leftarrow 1$;
 2. $\mathcal{F} \leftarrow \emptyset$;
 3. $Q_l \leftarrow Q(C_{ref})$;
 4. while $Q_l$ not empty do
 5.     $\mathcal{F}_l \leftarrow$ evaluateCandidates($\mathcal{KB}$, $C_{ref}$, $minsup$, $Q_l$)
 6.     $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}_l$;
 7.     $Q_{l+1} \leftarrow$ generateCandidates($\mathcal{KB}$, $\mathcal{F}_l$)
 8.     $l \leftarrow l + 1$;
 9. endwhile
10. return $\mathcal{F}$
```

In our current approach the specialization relation between patterns is based on the query support. The query support forms the quasi-order for $\mathcal{K}$*-queries*. It follows from the definition of an *answer* (see: *Definition 3*) that more general query contains more answers in the *answer set* than less general query, where the *answer set* is defined as follows.

Definition 5. The *answer set* of $\mathcal{K}$*-query Q over the* $\mathcal{KB}$, denoted as *answerset* (C_{ref}, Q, $\mathcal{KB}$), contains all the answers to Q w.r.t. $\mathcal{KB}$.

The approach discussed here can be extended to checking the *query containment* (see: *Definition 3*) during the pattern generation phase by using the generality notion presented below.
Definition 6. Given two $\mathcal{K}$-*queries* Q_1 and Q_2 to the knowledge base $\mathcal{KB}$ we say that Q_1 is *at least as general as* Q_2 under query containment, $Q_2 \preceq Q_1$, iff Q_2 is contained in a query Q_1.

According to the definition of the query *support* we can say that the query containment is monotonic w.r.t. *support* in the case of queries with the same sets of distinguished variables.

As the evaluation of a candidate pattern Q is based on the computation of the pattern *support* w.r.t. knowledge base $\mathcal{KB}$, it in turn boils down in our approach to query answering where the queries have the form of $\mathcal{K}$-*queries*. The quasi-ordered set $(\mathcal{L}, \succeq)$, where $\mathcal{L}$ is the language of $\mathcal{K}$-*queries*, can be searched by refinement operators where a refinement operator is defined by (Nienhuys-Cheng & deWolf, 1997) as follows.

Definition 7. In a quasi-ordered set$(\mathcal{L}, \succeq)$, a *downward* (resp. *upward*) *refinement operator* is a mapping ρ (resp. δ) from $\mathcal{L}$ to $2^{\mathcal{L}}$ such that $\forall P \in \mathcal{L}\ \rho(P) \subseteq \{Q \in \mathcal{L} \mid P \succeq Q\}$ (resp. $\delta(P) \subseteq (Q \in \mathcal{L} \mid Q \succeq P\}$).

For the task of frequent pattern discovery, downward refinement operators are more interesting than upward refinement operators. It is because the main algorithm assumes searching the pattern space levelwise from the patterns more general to the less general ones where the less general patterns are generated only from the patterns found frequent in the previous level. This approach helps to prune infrequent patterns found on each level and drive the search into more interesting directions. Thus we concentrate ourselves on downward refinement operators. Additionally to the generalization relation, several kinds of biases are also usually used to limit the search space where one of the most natural ones is the pattern language bias which defines syntactic constraints on the patterns to be found. Such restrictions on the search space not only limit the patterns to be investigated but also ensure that only well-formed ones are investigated. In our work we benefit from the background knowledge, in the form of an ontology, to limit our patterns. Well-formed patterns in our approach follow the restrictions imposed by our $\mathcal{KB}$ e.g. we can only add a literal to our query if its relationships with the literals already added to this query are logically consistent with the relationships existing in our $\mathcal{KB}$.

In (Lisi & Malerba, 2004), Object Identity bias is imposed on the $\mathcal{KB}$ that states that in a formula, terms denoted with different symbols must be distinct, i.e. represent different entities of the domain. As shown in (Lisi, Ferilli, and Fanizzi 2002) for the case of DATALOG queries, this bias can be the starting point for the definition of the quasi-ordering for constrained DATALOG clauses and as such can be more suitable for the purposes of frequent pattern discovery. Following their investigations we explicitly impose on our $\mathcal{KB}$ the corresponding bias of Unique Names Assumption (UNA) which is not assumed as default in the KAON2 framework and we assume that two differently named variables are distinct.

The outline of candidate generation algorithm is presented in Table 2.

Table 2. Candidate generation algorithm

```
generateCandidates(KB, $\mathcal{F}_l$)
1. $Q_{l+1} \leftarrow \bar{\ }$;
2. foreach pattern $P \in \mathcal{F}_l$ do
3.     $P' \leftarrow$ refine($P$)
4.     $Q_{l+1} \leftarrow Q_{l+1} \cup P'$
5. endforeach
6. return $Q_{l+1}$
```

Intuitively, with regard to the language of patterns, we can expect that the query Q_1 can be more specific than query Q_2 if Q_1 contains in its body literals not contained in Q_2 or Q_1 contains in its body literals more specific (w.r.t. to the $\mathcal{KB}$) on the same variables than Q_2. Thus we define the refinement operator for our task as follows.

Definition 8. Let Q be the conjunctive $\mathcal{DL}$-safe query in the form of $\mathcal{K}$-*query*. A downward refinement operator ρ for these conjunctive queries over a knowledge base $\mathcal{KB}$ is defined by the following rules:

[*Is-a-class*] Replace literal's class by its explicitly asserted in $\mathcal{KB}$ subclass.

[*Is-a-property*] Replace literal's property by its explicitly asserted in $\mathcal{KB}$ subproperty.

[*Lit-property*] Add a literal representing one of the properties of literals' classes unless such property already exists as a literal in the query and is functional and add a class literal of the variable already introduced in property literal for variables in property's domain or range.

The refinement rules are applied with checking if a new refinement doesn't cause redundancies or inconsistencies in a variable description in a query w.r.t. restrictions such as property functionality or inverse. Below the example application of refinement rules are presented for the ontology described in section 5.1.

Query:
q(*key*):-isCreditCardOf(*key*, *x*), Client(*x*), Gold(*key*)

Refinements:
q(*key*):-isCreditCardOf(*key*, *x*), Woman(*x*), Gold(*key*)
q(*key*):-isCreditCardOf(*key*, *x*), Man(*x*), Gold(*key*)
q(*key*):-isCreditCardOf(*key*, *x*), livesIn(*x*, *y*), Region(*y*), Client(*x*), Gold(*key*)
q(*key*):-isCreditCardOf(*key*, *x*), hasSexValue(*x*, *y*), SexValue(*y*), Client(*x*), Gold(*key*)
q(*key*):-isCreditCardOf(*key*, *x*), isUserOf(*x, y*), Account(*y*), Client(*x*), Gold(*key*)
q(*key*):-isCreditCardOf(*key*, *x*), hasAgeValue(*x*, *y*), AgeValue(*y*), Client(*x*), Gold(*key*)
q(*key*):-isCreditCardOf(*key*, *x*), isOwnerOf(*x*, *y*), Account(*y*), Client(*x*), Gold(*key*)

From patterns to association rules. From the discovered frequent patterns the association rules can be generated. Talking about the association rules in our setting we have in mind the semantics from RDM of a *query extension* (Dehaspe & Toivonen, 1999).

Definition 9. Let $P, Q \in \mathcal{L}$ be such patterns that $P \subseteq Q$. An *association rule* is an implication of the form $Q \rightarrow P$ (s, c), where s is rule *support* and c is rule *confidence*. The support s of the given association rule is the support of P and the confidence c is the probability that the consequent P occurs in the dataset when the antecedent Q occurs in the dataset.

Notice that the confidence of the association rule is also computed by means of the support.

5 Case Study

While selecting the dataset for illustrating our approach we faced the problem of a quite few ontologies with assertional component available whereas it is not difficult to find ontologies with only the terminological component available. As we address the problem of finding regularities in the dataset with help of a background knowledge in the form of an ontology we need a complete knowledge base. Thus we decided to use the existing, known from the PKDD'99 Discovery Challenge, financial dataset and on the basis of the relational schema and problem description, we created a simple ontology. Then we imported the text files into the relational database and we preprocessed the interesting for our current experiment part of the database into the instances of the ontology.

5.1 Financial Dataset

The financial dataset domain describes a bank that offers services (like managing of accounts, offering loans) to individual clients. The data describes the accounts of bank clients, the loans granted, the credit cards issued, etc. One client can have more accounts and more clients can manipulate with a single account. To an account more credit cards can be issued, but at most one loan can be granted. Also some additional demographic data about clients is publicly available like the age, sex or address.

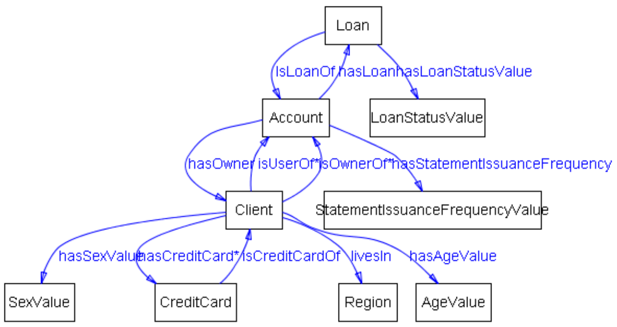


Fig. 1. A part of the financial ontology

In Figure 1, basic part of our ontology is presented that was created on the basis of the PKDD'99 financial dataset and that we used for the experiment described in this section (visualized by Protégé Ontoviz plugin). The concepts presented in Figure 1 have the subconcepts that are presented for the sake of clarity in Table 3. The values of the client�s age attribute were divided into the six ranges given in the table.

Table 3. Ontology concepts details

Concepts	Subconcepts
AgeValue	Below18, From18To25, From25To35, From35To50, From50To65, Above65
Client	Man, Woman
CreditCard	Classic, Gold, Junior
Loan	Finished, Running
Finished	NoProblemsFinishedLoan, NotPaidFinishedLoan
Running	DebtRunningLoan, OKRunningLoan
LoanStatusValue	OKStatus, ProblemStatus
Region	CentralBohemia, EastBohemia, NorthBohemia, SouthBohemia, WestBohemia, Prague, NorthMoravia, SouthMoravia
SexValue	FemaleSex, MaleSex
StatementIssuanceFrequencyValue	AfterTransaction, Monthly, Weekly

Our ontology is in the *OWL*-DLP fragment. The part of the ontology used for the experiment contains 41 classes, 12 properties and 11 464 instances in total.

5.2 Experimental Results

We performed a preliminary descriptive analysis of the gold credit card holders. For this preliminary experiment we haven�t used any pruning strategy that could benefit for example from the query containment notion. Thus we generated every possible pattern. We searched for the frequent patterns containing the reference concept in the form of Gold(*key*).

As an underlying query answering engine for our experiments we used KAON2 library. Our application is written in Java. We decided to use the support threshold of 20% and to perform computation until 8th level of depth, because it was the greatest level where the computation took reasonable amount of time. Below the quantitative results of the experiment are presented.

Table 4. Quantitative results of the experiment on gold credit card owners

Level No.	No. of candidates	No. of patterns
1	1	1
2	1	1
3	7	6
4	43	27
5	257	124
6	1338	480
7	5898	1522
8	20262	3849

The study revealed some interesting characteristics, for example the overrepresentation of the region North Moravia within gold credit card holders and overrepresentation of men. Next, what could be expected, it reported the overrepresentation of the clients at age 35-50 and 50-65 years in this group and monthly statement issuance frequency. Example pattern found at the 8th level of depth is presented below:

q(Key):-
  hasCreditCard(1_X1,Key), hasSexValue(1_X1,2_X2), MaleSex(2_X2)
  hasAgeValue(1_X1,4_X3), From50To65(4_X3), isOwnerOf(1_X1,5_X3)
  hasOwner(5_X3,1_X1), Account(5_X3), livesIn(1_X1,6_X3),
  Region(6_X3), Client(1_X1), Gold(Key)

where the variables have the prefix of the level on which they were added. For example □4_X3□ denotes that that the given variable was added to the pattern at the fourth level during the pattern discovery process. The pattern describes □the client with the male sex and the age between 50-65 years who has an account and the gold credit card□. The pattern was found with the support 21,59%.

The example association rules that can be generated from the pattern presented above are presented below using the semantics of the query extension (see: *Definition 9*):

hasCreditCard(1_X1,Key), hasAgeValue(1_X1,4_X3), From50To65(4_X3), isOwnerOf(1_X1,5_X3), hasOwner(5_X3,1_X1), Account(5_X3), livesIn(1_X1, 6_X3), Region(6_X3), Client(1_X1), Gold(Key)

$\rightsquigarrow$ hasSexValue(1_X1,2_X2), MaleSex(2_X2) (21,59%, 65,51%)

hasCreditCard(1_X1,Key), hasSexValue(1_X1,2_X2), MaleSex(2_X2), isOwnerOf(1_X1,5_X3), hasOwner(5_X3,1_X1), Account(5_X3), livesIn(1_X1, 6_X3), Region(6_X3), Client(1_X1), Gold(Key)

$\rightsquigarrow$ hasAgeValue(1_X1,4_X3), From50To65(4_X3) (21,59%, 35,84%)

Discussion. As can be seen in Table 4, applying a straightforward approach for the candidate generation leads to large increase of the number of candidates from level to level. Applying such an approach can cause that we result with the set of semantically equivalent and redundant queries. Thus a mechanism during candidate generation is desirable that performs a check for generated specializations to detect and prune such redundancies. Although there exist approaches that use pruning strategy not based on the background knowledge in the form of the intensional part of $\mathcal{KB}$, we do not want to apply such a pruning strategy, because it can mislead the search process.

We did also some experimental work with candidate pruning strategy based on query containment. Our idea was similar to the technique described in (Dehaspe & Toivonen, 1999). Each generated candidate, before being added to the candidates list, was examined whether it is not contained in any infrequent query or whether queries generated so far and given candidate are mutually inequivalent. This technique can potentially eliminate huge number of redundant patterns. For example, at the 4th level of the experiment we obtained 36 instead of 43 candidates and at the 5th level □ 98 instead of 257 candidates. However, our experiment confirmed, what was also raised in (Dehaspe & Toivonen, 1999), that such a method is very time consuming.

We are currently working on further qualitative and quantitative results on the financial ontology. The next step of our research that we are currently working on, investigates more efficient pattern generation and pattern pruning strategies. Our preprocessed knowledge base is published online[2] and can serve as a benchmark for further research.

6 Related Work

It should be noted first that to the best of our knowledge our approach is the first attempt so far to define a task of mining frequent patterns represented as conjunctive queries over description logic knowledge base and where the terminological part of the knowledge base is taken into account as a background knowledge (despite of SPADA described later in this section). The closest work to ours are the methods from RDM, and we are going to mention WARMR (Dehaspe & Toivonen, 1999) as the best known example and the system SPADA described in (Lisi & Malerba, 2004). As WARMR during the candidate generation phase uses the syntactic notion of θ-subsumption as a generality relation and furthermore it doesn't take the knowledge base into account it couldn't detect that having given two following patterns (DATALOG queries) Q_1 and Q_2

```
?- account(A), hasLoan(A,B), loan(B)
?- account(A), hasLoan(A,B), runningLoan(B)
```

Q_1 is more general than Q_2. WARMR generates patterns, by adding atoms from the set of atoms specified as valid and thus discovers the patterns like the following:

```
?- account(A), hasLoan(A,B), loan(B)
?- account(A), hasLoan(A,B), loan(B), runningLoan(B)
```

To discover semantic redundancies semantic generality relation should be adopted instead of a syntactic one. Very recently (De Raedt & Ramon, 2004) have introduced various types of condensed representations for avoiding redundancies in patterns in frequent DATALOG queries mining task. Still mining frequent DATALOG queries is a different class of problems that those that we would like to aim at. In our case we assume mining frequent patterns in the description logics knowledge bases and although we start the investigation from very restricted fragment of *OWL*, *OWL*-DLP, we are going to extend it to more expressive languages.

The system SPADA is the only system to our knowledge that explicitly assumes frequent pattern discovery in combined description logics and relational knowledge base. It aims at association rule discovery in multiple levels of description granularity and relies on the hybrid language $\mathcal{AL}$-log (Donini *et al*, 1998) which allows a treatment of both the relational and structural features of data. However the current version of SPADA admits only $\mathcal{ALC}$ primitive concepts in the structural knowledge (i.e. taxonomies) whereas roles and complex concepts have been disregarded. And most importantly the task of frequent pattern discovery is formulated in such a way that the patterns that can be found contain concepts only from the same level of a taxonomy. For example assuming that {`Client`, `Loan`, `Account`} are the concepts at the same level of taxonomy and {`FemaleClient`, `MaleClient`, `RunningLoan`} at the next level of a taxonomy. In SPADA the following patterns (*constrained* DATALOG *clauses*) can be obtained:

[2] http://www.cs.put.poznan.pl/alawrynowicz/semintec.htm

Q_1 =`q(X)` ← `hasLoan(X,Y), hasOwner (X,Z)`
  & `X:Account, Y:Loan, Z:Client`
Q_2 =`q(X)` ← `hasLoan(X,Y), hasOwner (X,Z)`
  & `X:Account, Y:RunningLoan, Z:FemaleClient`

but not the query

Q_3 =`q(X)` ← `hasLoan(X,Y), hasOwner (X,Z)`
  & `X:Account, Y:RunningLoan, Z:Client`

unless some concepts are replicated in some, lower levels of a taxonomy which in turn causes redundancies.

7 Conclusion and Future Work

In this work we present a new setting of frequent pattern discovery that we believe is very promising. In particular it concerns frequent pattern discovery in description logic knowledge bases assertional parts, taking as a background knowledge description logics terminological part, where the discovered patterns have the form of conjunctive queries formed by the conjunction of DL-atoms. It has several advantages. First of all, it is according to the best of our knowledge the first attempt so far to define a task of frequent pattern discovery in such a knowledge base, combining DL with rules. The terminological part of an ontology can serve as a natural bias for structuring the search space of patterns. Moreover, the terminology can be used to explain in more depth the results of knowledge discovery. Furthermore the discovered patterns can be processed to become association rules in the knowledge base, where the relationships discovered hadn□t been already captured by existing ontology. However, although we present and discuss the subsequent steps of such an approach, we treat it rather as a proof-of-concept method and a starting point of our investigations. In particular we do not investigate deeply the properties of our refinement operator and we treat the very basic subset of DL. In the next step we are going to carry out an extensive research concerning the properties of refinement operators for conjunctive queries over a knowledge base represented in different DLs with rules, especially in the light of properties of ideal refinement operators. We are currently investigating different optimization techniques for candidate generation and candidate pruning, especially those introduced for RDM methods. In the near future we are also going to extend our system to association rules generation from the discovered patterns.

Acknowledgments

Work partially supported by Polish Ministry of Scientific Research and Information Technology (under grant number KBN 3T11F 025 28).

References

1. Agrawal, R. Mannila, H., Srikant, R., Toivonen, H. and Verkamo, A. I., Fast discovery of association rules. Advances in Knowledge Discovery and Data Mining. AAAI Press, Menlo Park, CA, pp. 307 □ 328, (1996)

2. Berners-Lee T., Hendler J., and Lassila O., The Semantic Web. Scientific American, 284(5):34-43, (2001)
3. Dehaspe, L., Toivonen, H.: Discovery of frequent Datalog patterns. Data Mining and Knowledge Discovery, 3(1): 7 - 36, (1999)
4. De Raedt L., and J. Ramon, Condensed representations for Inductive Logic Programming, Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004) (Dubois, D. and Welty C., eds.), pp. 438-446, (2004)
5. Donini, F., Lenzerini, M., Nardi, D., & Schaerf, A., AL-log: Integrating datalog and description logics, Journal of Intelligent Information Systems, 10:3, 227□252, (1998)
6. D□eroski S., Lavrač N., (Eds.), Relational data mining. Springer, (2001)
7. Grosof B. N., Horrocks I., Volz R., and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In Proc. of the Twelfth Int□l World Wide Web Conf. (WWW 2003), pages 48□57. ACM, (2003)
8. Helft, N. Inductive generalization: A logical framework. In I. Bratko, & N. Lavrač (Eds.), Progress in Machine Learning-Proceedings of EWSL87: 2nd European Working Session on Learning (pp. 149□157), Wilmslow, U.K.: Sigma Press, (1987)
9. Hitzler P., Studer R., and Y. Sure, Description Logic Programs: A Practical Choice For the Modelling of Ontologies. In: Proceedings of the 1st Workshop on Formal Ontologies meet Meet Industry, FOMI'05, Verona, Italy, (2005)
10. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M., SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission 21 May 2004, http://www.w3.org/Submission/SWRL/
11. Hustadt U., Motik B. and U. Sattler. Reasoning for Description Logics around SHIQ in a Resolution Framework. FZI Technical Report 3-8-04/04, (2004)
12. Lisi, F.A., Ferilli, S., & Fanizzi, N., Object identity as search bias for pattern spaces. In F. van Harmelen (Ed.), ECAI 2002. Proceedings of the 15th European Conference on Artificial Intelligence (pp. 375□379). Amsterdam: IOS Press, (2002)
13. Lisi F.A., Malerba D., Inducing Multi-Level Association Rules from Multiple Relation, Machine Learning Journal, 55, 175-210, (2004)
14. McGuinness D.L., van Harmelen F., (eds), Overview of OWL Web Ontology Language, W3C Recommendation 10 February 2004, http://www.w3.org/TR/owl-features/
15. Mannila, H., Toivonen, H.: Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery 1(3): 241 - 258, (1997)
16. Motik B., Sattler U., Studer R.. Query Answering for OWL-DL with Rules. Proc. of the 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan, November, 2004, pp. 549-563, (2004)
17. Nienhuys-Cheng, S., de Wolf, R. *Foundations of inductive logic programming*, vol. 1228 of Lecture Notes in Artificial Intelligence. Springer, (1997)

Design and Implementation of an ECA Rule Markup Language

Marco Seiriö[1] and Mikael Berndtsson[2]

[1] Analog Software, Sweden
marco@analog.se
http://www.rulecore.com
[2] University of Skövde, Sweden
mikael.berndtsson@his.se
http://www.his.se/berk

Abstract. This paper presents the design and implementation of the rule engine ruleCore and the ECA rule markup language rCML. In particular, an extensive set of event operators are shown in the rCML rule markup language.

1 Introduction

Event Condition Action (ECA) rules were first proposed in the late 1980s and extensively explored during the 1990s within the active database community for monitoring state changes in database systems [11, 12, 14]. Briefly, ECA rules have the following semantics: when an event occurs, evaluate a condition, and if the condition is satisfied then execute an action. Recently, the concept of ECA rules have been transferred to the Web community for supporting ECA rules for XML data, see [3] for an overview.

We see a great need for an ECA rule markup language, in terms of having a suitable format for exchanging ECA rules between different applications and platforms. The need for an ECA rule markup language has also been identified elsewhere, e.g. [2]:

> " ... ECA rules themselves must be represented as data in the (Semantic) Web. This need calls for a (XML) Markup Language of ECA Rules."

Existing work on ECA rule markup languages is still very much in the initial phase, for example, the RuleML [5] standardization initiative has no markup for events, only for conditions and actions. In addition related work, e.g., [4, 6], on ECA rules for XML usually has an XML programming style for specifying ECA rules, rather than specifying ECA rules in a markup language.

We approach the above need for an ECA rule markup language and the lack of markup for events in existing literature by presenting the design and implementation of the ECA rule engine ruleCore[1] [13] and the ruleCore Markup

[1] ruleCore is a registered trademark of MS Analog Software kb

Language (rCML)[2]. The rCML Language has a clear separation between specification of the three different parts (event, condition, action). In addition, it also supports specification of an extensive set of composite events.

The work reported is joint work between industry (Analog Software) and academia (University of Skövde) that has taken place since 2002. The overall purpose of the joint project has been to transfer research results from the active database community into a commercial product. Although ruleCore is a commercial product it is free for academic research.

The reminder of the paper is structured as follows. Section 2 presents a brief overview of ruleCore. Section 3 presents details about the rCML markup language for ECA rules. Finally, Section 4 presents our conclusions.

2 A Brief Overview of ruleCore

In this Section we present an overview of the rule engine ruleCore [13]. RuleCore is implemented in Python, Qt, XML, and it supports ECA rules and event monitoring in heterogeneous environments. For example, a broker system can be used to integrate heterogeneous systems, and ruleCore can be attached to such a broker system and react to events that are sent through the broker.

2.1 Architecture

The ruleCore engine is built around a concept of loosely coupled components and is internally event driven. Components communicate indirectly using events and the publish/subscribe event passing model. The functionality of the ECA rules are provided by a number of components working in concert, where each component provides the functionality in a well defined small area. As the components are not aware of the recipient of the event they publish, it is easy to reconfigure the engine to experiment with other models besides the more well known ECA model. For example, one could insert an additional processing step between any of the event, condition or action steps. All internal and external events are stored in a relational database (PostgreSQL). Storing the event occurrences in a database implies that traditional database tools can be used for off-line analysis, visualization, simulation and reporting.

Handling of time is performed with timer events. If a component wishes to provide some kind of temporal feature, it can subscribe to timer events that are guaranteed to be generated at a specific time. A special timer component is responsible for publishing timer events according to wall clock time. The engine does not provide any hard real-time capabilities.

At the core of ruleCore lies a component framework, see Figure 1. The framework provides services for loading, initializing, starting and stopping components. It also handles persistence for the components and manages automatic crash recovery for the engine. The crash recovery mechanism is fully automatic

[2] rCML is a trademark of MS Analog Software kb

and restores the last known state of the engine at startup in case the engine was not shut down properly. The recovery mechanism uses the transaction management features of the PostgreSQL database to roll forward transactions to keep the internal state of the engine consistent at all times. The temporal features of the engine are fully integrated into the recovery process and thus all time dependencies are managed in a best effort manner even in case of engine failure or down time.

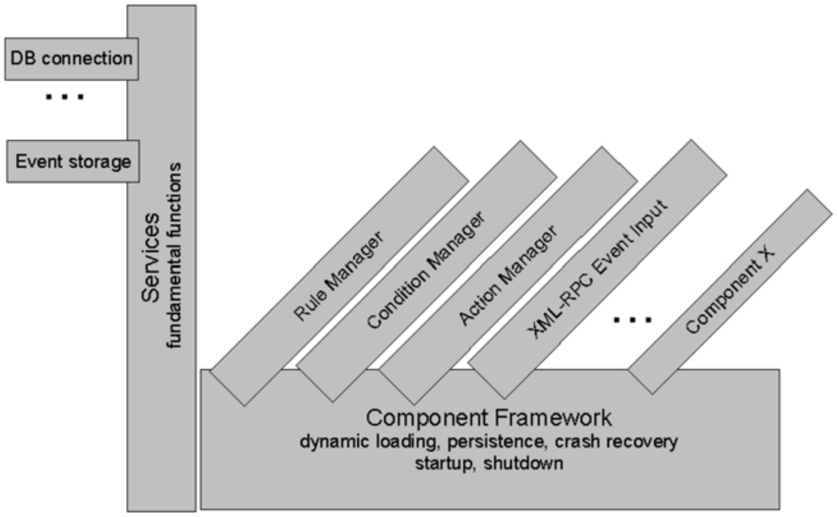


Fig. 1. ruleCore architecture

Components that receive events contain a local worker thread that processes incoming events. Processing of events are done asynchronously and in parallel in the different components. The framework controls the starting and stopping of threads in order to provide for an ordered and consistent startup and shutdown procedure. A service layer provides simple services that are available for all components and the component framework itself. The services of the service layer are always accessible as opposed to the services provided by the components that must be loaded and initialized before usage. The service layer also provides for encapsulation of external components such as databases.

The components that provide the functionality can broadly be divided into three groups.

- *Input components* are responsible for implementing support for a specific transport protocol and accepting events through it. Currently support exists for receiving events with XML-RPC, TCP/IP Sockets, SOAP, IBM WebSphere MQ, and TIBCO Rendezvous.
- *Rule components* provide the functionality of the rules. The rule manager, condition manager and action manager components work together using event passing to implement the ECA rule execution model.

– *Support components* provide functionality that is directly or indirectly used by other components. In this group we find components for event routing, event flow management, persistent state management and management of the configuration of the engine.

2.2 Situation Detector

In ruleCore terminology a composite event is called a *situation*. As the main focus of ruleCore is situation detection, we describe the functionality of the situation detection component in more detail.

The situation detector is implemented by using a number of detector nodes connected in a tree structure called a detector tree. Each ECA rule instance contains its own private instance of a detector tree, where each node in the detector tree implements some type of event detection.

Each node in the detector tree decides locally what event(s) to subscribe to, for example, a node might need to know when a specific point in time occurs and can then subscribe to a timer event in order to be informed when this particular point in time occurs. When a node detects a change it considers to be of interest it sends an event to its parent node.

When the root node in the detector tree receives an event signal, the situation is considered detected and the rule instance is triggered for condition evaluation. The situation detector can also detect if there is no possibility for detecting the situation in the future and will inform its enclosing rule instance about this fact which will then delete itself.

3 The ruleCore Markup Language (rCML)

In this Section we describe the ruleCore Markup Language (rCML) that is used for specification of events and rules in ruleCore. All described features of rCML have been implemented and are supported by ruleCore. Encodings in rCML are in UTF-8. In order to ease comparisons with previous work in active databases we use the term composite event in this section rather than situation.

3.1 Specification of Event Types

Events are specified inside the <event-defs> element and each individual event type is described with an <event-def> element. Two different event types are supported in rCML: basic events and composite events.

Basic Events. A basic event in rCML is defined with an <event-def> element that has two attributes:

– **type**. The attribute *type* is set to *basic* for basic events.
– **name**. A unique value for the *name* attribute that identifies the event.

Event parameters are specified inside the sub element <parameters> and each individual parameter is specified with a <parameter> element. Each <parameter> element has two attributes:

- **name**. A unique value for the *name* attribute that identifies the parameter.
- **type**. Specification of data type for the parameter. Three data types are supported:
  1. *string*. A string in valid Unicode.
  2. *number*. A decimal number according to ANSI standard X3.274-1996.
  3. *date*. A date in ISO format, e.g., YYYY-MM-DD HH:MM:SS.

Below is an example of a basic event E1 with three parameters:

```
<event-defs>
    <event-def type='basic' name='E1'>
        <parameters>
            <parameter type='string' name='parameter1'/>
            <parameter type='number' name='parameter2'/>
            <parameter type='date' name='parameter3'/>
        </parameters>
    </event-def>
</event-defs>
```

Composite Events. A simple composite event can be defined by using two basic events. However, more complex composite events are defined by building composite events out of other composite events. A composite event that contributes to the detection of another composite event is called a sub-composite event.

A composite event in rCML is defined with an <event-def> element that has two attributes:

- **type**. The attribute *type* is set to *composite* for composite events.
- **name**. A unique value for the *name* attribute that identifies the event.

Each composite event defined in rCML has four sub elements:

- <detect-event>. The <detect-event> element specifies the event that is generated when the composite event is detected.
- <no-detect-event>. The <no-detect-event> element specifies the event that is generated when the composite event can never be detected. When a composite event cannot be detected, e.g., a composite event using a time point that has already passed, the rule instance is automatically deleted. The <no-detect-event> event is generated just prior to deletion of the rule instance containing the composite event that never be detected.
- <event-selector> The <event-selector> element specifies a logical condition (or filter) for the composite event. See section on specification of conditions for further details.

– <detector>. The composite event detector itself is defined under the <detector> element. The detector consists of a number of sub elements (event operators) that describe the composite event.

Each sub element to the <detector> element is an event operator. The following event operators are supported:

– The conjunction operator is supported by the <and> element. Similar event operators are supported in Snoop [8], Ode [10], and SAMOS [9]. The following example specifies that events E1, E2, and E3 must have occurred before the composite event is detected:

```
<detector>
    <and>
        <event-ref type='event'>E1</event-ref>
        <event-ref type='event'>E2</event-ref>
        <event-ref type='event'>E3</event-ref>
    </and>
</detector>
```

– The disjunction operator is supported by the <or> element. Similar event operators are supported in Snoop [8], Ode [10], and SAMOS [9]. The following example specifies that event E1 or event E2 or event E3 must have occurred before the composite event is detected:

```
<detector>
    <or>
        <event-ref type='event'>E1</event-ref>
        <event-ref type='event'>E2</event-ref>
        <event-ref type='event'>E2</event-ref>
    </or>
</detector>
```

– The sequence operator is supported by the <sequence> element. The sequence operator is perhaps most useful when the sub-events are basic events. Similar event operators are supported in Snoop [8], Ode [10], and SAMOS [9]. The following example specifies that event E1 is followed by event E2, and that event E3 occurs after E2:

```
<detector>
    <sequence>
        <event-ref type='event'>E1</event-ref>
        <event-ref type='event'>E2</event-ref>
        <event-ref type='event'>E3</event-ref>
    </sequence>
</detector>
```

– The prior sequence operator is supported by the <prior> element. The <prior> element behaves like the <sequence> element when all of its sub events are basic events. However, when the sub events are composite events the semantics of the composite event detection are as follow. The terminating event in a sub-composite event must occur before the terminating event in the following sub-composite event occurs. The semantics of the prior sequence operator in rCML is similar to the semantics of the prior event operator as defined in Ode [10]. The following example specifies that the detection of the sub-composite event CE1 must be completed before the terminating event in the following sub-composite event CE2 occurs:

```
<detector>
    <prior>
        <event-ref type='event'>CE1</event-ref>
        <event-ref type='event'>CE2</event-ref>
    </prior>
</detector>
```

– The relative sequence operator is supported by the <relative> element. The <relative> element behaves like the <prior> and <sequence> element when all of its sub events are basic events. The <relative> element requires that the terminating event in a sub-composite event is detected before the detection of the initiating event in the following sub-composite event. The semantics of the relative sequence operator in rCML is similar to the semantics of the relative event operator as defined in Ode [10]. The following example specifies that the terminating event of the sub-composite event CE1 must be detected before the detection of the initiating event in the following sub-composite event CE2.

```
<detector>
    <relative>
        <event-ref type='event'>CE1</event-ref>
        <event-ref type='event'>CE2</event-ref>
    </relative>
</detector>
```

– The any operator is supported by the <any> element. A similar event operator is found in Snoop [8]. The any event operator will detect when any m events out of the n specified sub events have occurred, where $m \leq n$. The order of detection of the sub events is not important. Thus, the semantics of the <any> element is similar to the <and> element, but with the difference that the user can choose that only a limited number of the sub events need to be detected. The following example specifies that the composite event is detected when two out of the three specified sub events E1, E2, or E3 have occurred.

```
<detector>
    <any number='2'>
        <event-ref type='event'>E1</event-ref>
        <event-ref type='event'>E2</event-ref>
        <event-ref type='event'>E3</event-ref>
    </any>
</detector>
```

- The between operator is supported by the <between> element. The between event operator uses one initiating event and one terminating event to detect the composite event. Any number of events can occur between the initiating event and the terminating event. All the events that occur between the initiating event and the terminating event can be stored for condition evaluation. The between event operator is usable when the initiating and terminating events of a composite event are known but not how many events that will occur in between them. The following example specifies that the composite event is detected when E1 is followed by zero or more events before event E2 occurrs.

```
<detector>
    <between>
        <event-ref type='event'>E1</event-ref>
        <event-ref type='event'>E2</event-ref>
    </between>
</detector>
```

- The not operator is supported by the <not> element. The classical semantics of the NOT operator when specifying composite events for ECA rules are that an event E is not detected during an interval specified by two events. For example, a composite event NOT E3 (E1,E2) is detected if event E3 is not detected between the detection of E1 and E2. Previous systems [7, 9] have restricted the use of the NOT operator to: (i) a conjunction [7], i.e., event E3 should not occur between (E1 and E2), or ii) a time interval [9], i.e., event E3 should not occur between 18:00 and 20:00.
  The approach taken in rCML generalize the usage of the NOT operator to any type of event interval. Thus, the NOT operator extends previous usage of the NOT operator for specifying composite events for ECA rules. The following example specifies that the composite event is detected when event E5 is not detected during the detection of the sequence (E1, E2,E3).

```
<detector>
    <sequence>
        <event-ref type='event'>E1</event-ref>
        <event-ref type='event'>E2</event-ref>
        <event-ref type='event'>E3</event-ref>
        <not>
            <event-ref type='event'>E5</event-ref>
        </not>
    </sequence>
</detector>
```

– The count operator is supported by the <count> element. The count event operator is used to count how many times its only sub event is detected within an interval. The interval is configured in such a way that the count operator knows when it should start and stop counting event occurrences. Thus, a <count> element is either in an open state (counting) or in a closed state (not counting). The <countcfg> element has six sub elements that must be in the cocfg namespace:
  1. <open-output> - This element specifies the value a <count> element has when it is in the open state. Possible values are:
     • true - Output true to the <count> element.
     • false - Output false to the <count> element.
     • input - Output the same value as the <count> element receives from the its sub event.
  2. <closed-output> - This element specifies the value a <count> element has when it is in the closed state. The possible values are the same as for <open-output>.
  3. <open-count> - This element describes when to activate (or open) the count operator. It contains an integer that specifies the number of event occurrences that must have occurred before the counter starts.
  4. <close-count> - This element describes when to close the count operator. It contains an integer that specifies the number of event occurrences that must have occurred before the counter is closed. The <close-count> integer should be greater than the <open-count> integer.
  5. <periodic> - This element is used to specify a periodic behaviour of the <count> element. It means that the <count> element is in the open state as many times as specified by <open-count> and then in the closed state as many times as specified by <closed-count>. Valid values are “True” or “False”.
  6. <initial-state> - This element specifies the initial state of the <count> element, which can be *Open* or *Closed*.

  In its simplest form, the count operator counts the number of event occurrences:

```
<count>
    <event-ref>E1</event-ref>
</count>
```

  In more advanced forms, the count operator counts the number of event occurrences when specific conditions are met. For example, the following composite event ignores the first two occurrences of E1, and counts the following three occurrences of E1:

```
<detector>
   <count>
      <cocfg:countcfg xmlns:cocfg='http://www.rulecore.com/cocfg'>
      <cocfg:open-output>true</cocfg:open-output>
      <cocfg:closed-output>false</cocfg:closed-output>
      <cocfg:open-count>2</cocfg:open-count>
      <cocfg:close-count>5</cocfg:close-count>
      <cocfg:periodic>False</cocfg:periodic>
      <cocfg:initial-state>Open</cocfg:initial-state>
      </cocfg:countcfg>
      <event-ref type='event'>E1</event-ref>
   </count>
</detector>
```

– The timeport operator is supported by the <timeport> element, and it supports specification of absolute, relative, and periodic time events. Similar events have been proposed by the active database community.
  The following example in rCML specifies an absolute time event Event2, i.e., the timeport is opened 15th of June 2004 at 12:30:30, and that the timeport closes 15th of June 2004 at 12:30:30.

```
<detector>
  <timeport xmlns:xmlns='http://www.rulecore.com/tpcfg'
    xmlns:tpcfg='http://www.rulecore.com/tpcfg'>
    <tpcfg:timers xmlns:tpcfg='http://www.rulecore.com/tpcfg'>
      <tpcfg:timer tpcfg:name='Timer'>
        <tpcfg:start-date>
          <tpcfg:year tpcfg:mode='constant'>2004</tpcfg:year>
          <tpcfg:month tpcfg:mode='constant'>6</tpcfg:month>
          <tpcfg:day tpcfg:mode='constant'>15</tpcfg:day>
          <tpcfg:weekday tpcfg:mode="> </tpcfg:weekday>
          <tpcfg:hour tpcfg:mode='constant'>12</tpcfg:hour>
          <tpcfg:minute tpcfg:mode='constant'>30</tpcfg:minute>
          <tpcfg:second tpcfg:mode='constant'>30</tpcfg:second>
        </tpcfg:start-date>
        <tpcfg:stop-date>
          <tpcfg:year tpcfg:mode='constant'>2004</tpcfg:year>
          <tpcfg:month tpcfg:mode='constant'>6</tpcfg:month>
          <tpcfg:day tpcfg:mode='constant'>15</tpcfg:day>
          <tpcfg:weekday tpcfg:mode="> </tpcfg:weekday>
          <tpcfg:hour tpcfg:mode='constant'>12</tpcfg:hour>
          <tpcfg:minute tpcfg:mode='constant'>30</tpcfg:minute>
          <tpcfg:second tpcfg:mode='constant'>30</tpcfg:second>
        </tpcfg:stop-date>
        <tpcfg:open-output>true</tpcfg:open-output>
        <tpcfg:closed-output>false</tpcfg:closed-output>
```

```
            </tpcfg:timer>
        </tpcfg:timers>
        <event-ref alias='e2,alias2' type='alias'>Event2</event-ref>
    </timeport>
</detector>
```

The output of a <timeport> element is controlled by a timer. The timer is configured to open and close the <timeport> element at specific dates and times. Each date specification consists of seven fields: year, month, day, weekday, hour, minute, and second. Each of the seven fields can be individually set to different modes that decides how the actual value of the time field will be calculated:

- The *constant* mode is the simplest. It requires a constant to be entered for the date part, such as "2004" for the year. The constant mode requires you to know in advance the exact date or time for the field.
- In order to base dates on the occurrence of events, the *offset* mode can be used. The offset mode lets you specify an offset in relation to an event occurrence. This is useful if a date or time should occur some time after a certain event. To create a time that occurs ten minutes after a certain event you would set all the fields modes to offset and set the value to zero for all fields except the minutes that would be set to +10.
- Dates and times that occur at a regular interval can be specified with by using the *each* mode of a field. For example, if the hour field is set to mode each the time will occur each hour. By setting other fields to constant mode the time will occur only on those times and dates. This could for example be used to have a time to occur each day but only during a certain month.

Each time field does not have to have the same mode. By using different combinations of the modes for the different part it is possible to specify dates in an advanced way, for example, specification of a time point that occurs five hour after an certain event but on the last Sunday of the next month.

– The state gate operator is supported by the <state-gate> element. A <state-gate> element can be used to detect whether an object is in a particular state, for example, between 12:00 and 13:00 the object is in the "LUNCH" state. A <state-gate> element can be opened or closed depending upon whether a state exists and what parameters a state instance has. States are specified by the aid of conditions, this means that the actual specification of when the state LUNCH begins and ends is done in the <condition-def> element, i.e.. the reference <sgcfg:condition>condition12 </sgcfg:condition>.

```
<detector>
    <state-gate>
        <sgcfg:state-gate-cfg xmlns:sgcfg='http://www.rulecore.com/sgcfg'>
```

```
            <sgcfg:open-output¿true< /sgcfg:open-output>
            <sgcfg:closed-output¿true< /sgcfg:closed-output>
            <sgcfg:state-exists¿open</sgcfg:state-exists>
            <sgcfg:state-selector>
                <sgcfg:condition>condition12</sgcfg:condition>
            </sgcfg:state-selector>
        </sgcfg:state-gate-cfg>
    </state-gate>
</detector>
```

All state instances are created and deleted by a rule actions.

3.2 Specification of Conditions

Conditions are used in a number of places, e.g., as logical conditions for events (event selector), or as rule conditions. All types of conditions are defined under the <condition-defs> element and each individual condition is specified with a <condition-def> element.

The <condition-def> element has the following attributes:

- name - The name of the condition. All condition names must be unique.
- composite - Specifies whether the condition is a basic condition or a composite condition (contains logical operators). Allowed values are “yes” and “no”
- always-true - Allowed values are “yes” and “no”

Due to space limitations of this paper we only show an example of a condition and do not go further into the details of how conditions and expressions are specified in rCML. The interested reader is referred to [1] for additional details.

```
<condition-defs>
    <condition-def always-true=”no” composite=”yes” name=”Condition 1”>
        <parameters>
            <parameter name=”selection 1”>
                <event-ref>Event1</event-ref>
                <param-ref>parameter 1</param-ref>
                <instance>first</instance>
                <function> mul(, 1.3)</function>
            </parameter>
        </parameters>
    <condition-def>
</condition-defs>
```

3.3 Specification of Actions

Actions are defined under the <action-defs> element and each individual rule action is specified using the <action-def> element. An <action-def> element can in turn be composed of several <action-item> elements that are executed in the order they are defined. Thus, <action-def> element can launch the execution of two applications, where each application call is defined by a separate <action-item> element.

Four types of rule actions are supported in rCML:

– script - Script actions execute external scripts or applications
– event - Event actions send event occurrences to the ruleCore rule engine
– create_state - Rule actions that create new state items
– delete_state - Rule actions that delete state items

Below is an example of a script action that executes an external application.

```
<action-defs>
    <action-def name='Action1'>
        <action-item type="script" name="exec app">c:\actions\action1.exe
        </action-item>
    </action-def>
<action-defs>
```

3.4 Specification of ECA Rules

ECA rules are specified inside the <rules> element and each individual ECA rule is described with a <rule> element. The <rule> element has the following attributes:

– **name**. A unique value for the name attribute that identifies the rule.
– **create**. The create attribute controls rule instance creation. Possible values are:
  1. *single* - only a single rule instance is created.
  2. *single_replace* - only a single rule instance exists at any point in time. This implies that when a new initiator event occurs, the old rule instance is deleted and replaced with a new rule instance.
  3. *init* - create a rule instance each time an initiator event occurs.
  4. *reject* - create a new rule instance if the event is rejected by all rule instances. Rejection can be done by the event selector of the rule.
– **parameter**. Controls how the event parameters of each event are stored during the event detection process. Allowed values are:
  1. *append* - store all parameters of each event that are involved in the detection of the current event,
  2. *first* - store only the parameters of the first event in each situation,
  3. *last* - store only the last parameter of the event detection of each event type, or
  4. *never* - do not store parameters at all.

The <rule> element has the following subelements:

- The <description> element contains a description of the rule. This subelement is only for user convenience and it is not used by the engine.
- The <event-ref> element contains a reference to the composite event that trigger the rule. The main target of applications for rCML are applications that react on composite events. However, a simple basic event can also act as a triggering event for a rule by constructing a composite event with only one sub event.
- The <condition-ref> contains a reference to a condition definition element <condition-def> that specifies a condition that is evaluated when the rule is triggered by its event.
- The <action-ref> contains a reference to a <action-def> element that is executed if the rule condition is evaluated to true.
- The <minus-action-ref> contains a reference to a <action-def> element that is executed if the triggering event can never be detected.
- The <instance-limit> element is used to limit the number of the rule instance for each type of rule. Possible values are:
  1. None
  2. An integer specifying the maximum number of rule instances.

The <event-ref>, <condition-ref> and <action-ref> and <minus-action-ref> all contain an attribute called *enabled* with the possible values of *yes* or *no*. Thus a rule whose rule condition should always be evaluates to true is specified as <condition-ref enabled='no'></condition-ref>.

Below is an example of an ECA rule in rCML:

```
<rules>
    <rule parameter='append' create='single' name='Rule1'>
        <description>A description of this rule</description>
        <event-ref enabled='yes'>E1</event-ref>
        <condition-ref enabled='yes'>Condition12</condition-ref>
        <action-ref enabled='yes'>Action1</action-ref>
        <minus-action-ref enabled='yes'>Action2</minus-action-ref>
        <instance-limit>None</instance-limit>
    </rule>
</rules>
```

4 Conclusions

This paper has presented an overview of the ruleCore rule engine that is build on a component framework. We have also presented a first proposal for an ECA rule markup language. In particular, we have showed how to specify and implement an extensive set of event operators. Finally, we have also presented some novel event operators, e.g., state gate, that have previously not been suggested in the literature.

Although rCML is ruleCore specific, we believe that the rCML design and implementation can contribute to standardization efforts on developing a more general ECA rule markup language.

Acknowledgement

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

1. J. J. Alferes, R. Amador, E. Behrends, M. Berndtsson, F. Bry, G. Dawelbait, A. Doms, M. Eckert, O. Fritzen, W. May, P. L. Patranjan, L. Royer, F. Schenk, and M. Schroeder. Specification of a Model, Language and Architecture for Reactivity and Evolution. Technical Report REWERSE deliverable I5-D4, 2005.
2. J. J. Alferes, R. Amador, and W. May. A general language for Evolution and Reactivity in the Semantic Web. In *Proceedings of the 3rd Workshop on Principles and Practice of Semantic Web Reasoning*, 2005.
3. J. J. Alferes, J. Bailey, M. Berndtsson, F. Bry, J. Dietrich, A. Kozlenkov, W. May, P. L. Patranjan, A. Pinto, M. Schroeder, and G. Wagner. State-of-the-art on Evolution and Reactivity. Technical Report REWERSE deliverable I5-D1, 2004.
4. J. Bailey, A. Poulovassilis, and P. T. Wood. An Event Condition Action Language for XML. In *Proceedings of WWW'2002*, pages 486–495, 2002.
5. H. Boley, B. Grosof, M. Sintek, S. Tabet, and G. Wagner. RuleML Design. RuleML Initiative, http://www.ruleml.org/, 2002.
6. F. Bry and P.-L. Patranjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing SAC'2005*, 2005.
7. S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra. Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules. *Information and Software Technology*, 36(9):559–568, 1994.
8. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite Events for Active Databases: Semantics Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, September 1994.
9. K. R. Dittrich, H. Fritschi, S. Gatziu, A. Geppert, and A. Vaduva. SAMOS in hindsight: experiences in building an active object-oriented DBMS. *Information Systems*, 28(5):369–392, July 2003.
10. N. Gehani, H. V. Jagadish, and O. Smueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.
11. N. W. Paton, editor. *Active Rules in Database Systems*. Monographs in Computer Science. Springer, 1999. ISBN 0-387-98529-8.
12. N. W. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
13. ruleCore. The ruleCore home page: http://www.rulecore.com/.
14. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996. ISBN 1-55860-304-2.

Extending the SweetDeal Approach for e-Procurement Using SweetRules and RuleML

Sumit Bhansali and Benjamin N. Grosof

Massachusetts Institute of Technology
Sloan School of Management, Cambridge, MA 02139, USA
{bhansali,bgrosof}@mit.edu
http://ebusiness.mit.edu/bgrosof

Abstract. We show the first detailed realistic e-business application scenario that uses and exploits capabilities of the SweetRules V2.1 toolset for e-contracting using the SweetDeal approach. SweetRules is a uniquely powerful integrated set of tools for semantic web rules and ontologies. SweetDeal is a rule-based approach to representation of business contracts that enables software agents to create, evaluate, negotiate and execute contacts with substantial automation and modularity. The scenario that we implement is of electronic procurement of computers, with request-response iterated B2B supply-chain management communications using RuleML as content of the contracting discovery/negotiation messages. In particular, the capabilities newly exploited include: SweetJess or SweetXSB to do inferencing in addition to the option of SweetCR inferencing, SweetOnto to incorporate/merge-in OWL-DLP ontologies, and effectors to launch real-world actions. We identify desirable additional aspects of query and message management to incorporate into RuleML and give the design of experimental extensions to the RuleML schema/model, motivated by those, that include specifically: fact queries and answers to them. We present first scenario of using SCLP RuleML for rebates and financing options, in particular exploiting the courteous prioritized conflict handling feature. We give a new SweetDeal architecture for the business messaging aspect of contracting, in particular exploiting the situated feature to exchange rulesets, that obviates the need to write new (non-rule-based) agents as in the previous SweetDeal V1 prototype. We finally analyze how the above techniques, and SweetDeal, RuleML and SweetRules overall, can combine powerfully with other e-business technologies such as RosettaNet and ebXML.

1 Introduction

In this paper, we describe in detail a practical electronic contracting scenario that uses RuleML[1], the Situated Courteous Logic Programs (SCLP) knowledge representation [6], and the SweetRules V2.1 semantic web rules toolset [2] together to show how a real-world business application such as electronic procurement can be supported with semantic web technologies including also OWL [3]. The electronic procurement application was chosen not only because of its wide applicability in e-business but also because it allows us to showcase different features of the new SweetRules V2 implementation. Specifically, we show how powerful features of the new implementation such as importing OWL-DLP ontologies into a rule-based knowledge base, executing real-world business processes such as sending e-mail from rules, and inferencing on RuleML rules obtained from ontologies as well as rulebases possibly expressed in different types of KR. The procurement example allows us to also see how different business functions/features such as rebates, financing scenarios, payment options, which might be applicable in a wide variety of business applications, can be expressed using the RuleML KR language.

From our investigation of the electronic procurement scenario, we suggest inclusion of specific features in future versions of the RuleML KR to support query and message management that would be useful especially in business applications involving iterated request-response communication, such as e-contracting applications. Finally, we also explain how our electronic contracting approach based on RuleML and SweetRules can relate to other e-business technologies such as RosettaNet [4] and ebXML [5].

The paper is organized as follows. In section 2, we provide a brief overview of the technologies – RuleML, SweetRules, and SweetDeal – that we use in this research. Section 3 provides an overview of our approach and scenario. Section 4 illustrates the expressive power of RuleML in representing key contract provisions, specifically those of financial incentives. Section 5 describes the iterated contract construction process in great detail. Section 6 concludes the paper.

2 Overview of Technologies

We provide below a short description of the different technologies used in this research.

2.1 RuleML

RuleML [1] is the emerging standard for representing semantic web rules. The fundamental KR used in RuleML is situated courteous logic program or SCLP, which has been demonstrated to be expressively powerful [6]. The courteous part of SCLP enables prioritized conflict handling, which in turn enables modularity in specification, modification, merging and updating. The situated part of SCLP enables attached procedures for “sensing” (i.e. testing rule antecedents) and “effecting” (i.e. performing actions when certain conclusions are reached).

2.2 SweetRules

SweetRules [2], a uniquely powerful integrated set of tools for semantic web rules and ontologies, is newly enhanced in V2.1. The new version of SweetRules include capabilities such as first-of-a-kind semantics-preserving translation and interoperability between a variety of rule and ontology languages (including XSB Prolog [7], Jess [8] production rules, HP Jena-2 [9], IBM CommonRules [10], and the SWRL [11] subset of RuleML), highly scaleable backward and forward inferencing, and easy merging of heterogeneous distributed rulebases/ontologies.

2.3 SweetDeal

SweetDeal [12] is an electronic contracting approach that uses SCLP RuleML to support creation, evaluation, negotiation, execution and monitoring of formal electronic contracts between agents such as buyers and sellers. The approach builds on top of the SweetRules toolest to showcase the power of SCLP, RuleML, and SweetRules, as a design – and implemented prototype software – in the specific business application of electronic contracting.

3 Overview of Approach and Scenario

The extended SweetDeal approach described in this paper consists of three primary pieces: communication protocol between the contracting agents, contract knowledge bases and agent communication knowledge bases. We briefly describe these below in the context of our specific scenario of electronic procurement.

3.1 Communication Protocol

In our scenario, the buyer, Acme Corp, is interested in purchasing computers of a particular configuration. The buyer attempts to establish a procurement contract with the seller, Dell Computers. We assume that Dell Computers is a preferred vendor of computers for Acme Corp. To establish the terms of the contract, the buyer and seller agents exchange messages in an iterated fashion.

The protocol of message exchanges is as follows: the buyer first sends an RFP (request for proposal) to the seller. The seller responds to the RFP with the proposal. Based on specific business criteria, the buyer chooses to accept or reject the proposal. The buyer may also suggest modifications to the proposal before accepting or rejecting it. The RFP message from the buyer contains specific details about the desired computer configuration. It also contains any queries to which the seller must provide answers in its proposal. The proposal message from the seller contains several formal contract fragments which describe useful business provisions such as rebates, financing options, as well as payment options for the buyer. In addition to specifying the contractual provisions, the seller also provides answers to the queries posed by the buyer. Finally, it may pose additional queries for the buyer that the buyer in turn must provide answers to in the next negotiation message. After the buyer is satisfied with the final contract proposal from the seller, it generates a purchase order that is sent to the seller. To complete the transaction, the seller delivers the order and the buyer makes arrangements to pay the seller via the chosen payment option. Any contingencies in the execution of the order/transaction are handled according to the terms of the contract.

3.2 Contract Knowledge Bases

Contract negotiation messages exchanged between the agents are RuleML knowledge bases that are executable within SweetRules V2.1 software. Contract knowledge bases contain the following six main technical components: rules, facts, ontologies including OWL-based ontologies as well as object-oriented default inheritance ontologies, effectors, f-queries and their answers, and conditional queries. We briefly describe each of these components below. Since RuleML as an XML-based markup language is fairly verbose and since the presentation syntax of RuleML has not yet been implemented completely in SweetRules, we use the IBM CommonRules (CR) V3.3 syntax in all our examples to allow for concise presentation and easier comprehension. In future, it would be more desirable instead to use the RuleML presentation syntax. See [16], especially the Rules language description, for the initial version of that presentation syntax, and see [2], especially its documentation, for its experimental extension to include the Situated feature and for its (currently, still partial) support in SweetRules.

3.2.1 Rules

RuleML rules express the if-then implications of the contractual fragments and form the bulk of the contract knowledge base. Each rule has a head and a body. The "head" is the part of the rule after the "then", whereas the "body" is the part of the rule that follows "if" and precedes "then". The example below shows a simple <rebate> rule: the seller might wish to provide a rebate offer to the buyer in the proposal. Specifically, the seller might wish to offer a rebate in the amount of $1000 to the buyer if the number of computers ordered by the buyer is more than 75. Due to current tool limitations of numeric types in translating CommonRules to RuleML, all numeric constants in the rule examples below are represented using strings, e.g., "75" is represented as "seventyfive".

```
<rebate>
if
      quoteID(?QuoteID) AND quantityOfItemOrdered(?Q) AND
            isGreaterThan(?Q, seventyfive)
then
      rebateAmount(?QuoteID, thousand);
```

3.2.2 Facts

RuleML facts or assertions are rules that have no bodies. The simple examples below show facts that are specified in the RFP from the buyer to the seller. The quantity of item ordered by the buyer is 80 (computers) and the buyer is located in the state of Florida. (We assume that both buyer and seller are located in USA).

```
      quantityOfItemOrdered(eighty);
      buyerLocationState(florida);
```

3.2.3 Ontologies

Ontologies are vocabularies that express the background knowledge used by the contract rules. They can be either OWL [15] ontologies or rule-based object-oriented default inheritance ontologies. OWL ontologies used must be in the Description Logic Programs (DLP) [13] subset of OWL, i.e. in the subset of OWL that is translatable into LP rules. SweetRules V2.1 software allows for translation from OWL-DLP to RuleML rules. We show below a simple example of an OWL ontology that is used by the buyer. The ontology (procurement.owl) has three classes: *buyer, seller,* and *product*, and three object properties: *preferredVendorIs, buysProduct,* and *sellsProduct.* The ontology fragment also has some instance data: *computers is a product, Dell sells computers, Acme buys computers, Acme has Dell as a preferred vendor.* Since the ontology is in the DLP subset of OWL, a translation from OWL to RuleML exists and SweetRules V2.1 software can be used (see command C1 below) to convert the ontology to a rule-based knowledge base in RuleML.

```
translate owl clp c:\procurement.owl c:\procurement.clp    (C1)
```

The ontology (procurement.owl) is shown below:

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://www.procurement.org/procurement.owl#"
    xml:base="http://www.procurement.org/procurement.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="buyer"/>
  <owl:Class rdf:ID="seller"/>
  <owl:Class rdf:ID="product"/>

  <owl:ObjectProperty rdf:ID="preferredVendorIs">
        <rdfs:domain rdf:resource="#buyer"/>
        <rdfs:range rdf:resource="#seller"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="buysProduct">
        <rdfs:domain rdf:resource="#buyer"/>
        <rdfs:range rdf:resource="#product"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="sellsProduct">
        <rdfs:domain rdf:resource="#seller"/>
        <rdfs:range rdf:resource="#product"/>
  </owl:ObjectProperty>

  <seller rdf:ID="dell">
       <sellsProduct rdf:resource="#computers"/>
  </seller>
  <buyer rdf:ID="acme">
        <preferredVendorIs rdf:resource="#dell"/>
         <buysProduct rdf:resource="#computers"/>
  </buyer>

  <product rdf:ID="computers"/>
</rdf:RDF>
```

The translation of the ontology to rules is shown below. The translation has been slightly modified for ease of readability. Each of the predicates below would be prefixed in the original translation with a long namespace URI indicated in the OWL document above. The namespace URI has been removed from all predicates below.

```
<emptyLabel> if buysProduct(?X, ?Y) then buyer(?X);
<emptyLabel> if buysProduct(?X, ?Y) then product(?Y);
<emptyLabel> if sellsProduct(?X, ?Y) then seller(?X);
<emptyLabel> if sellsProduct(?X, ?Y) then product(?Y);
<emptyLabel> if preferredVendorIs(?X,?Y) then buyer(?X);
<emptyLabel> if preferredVendorIs(?X, ?Y) then seller(?Y);
<emptyLabel> sellsProduct(dell, computers);
<emptyLabel> preferredVendorIs(acme, dell);
<emptyLabel> buyer(acme);
```

```
<emptyLabel> product(computers);
<emptyLabel> Class(product);
<emptyLabel> Class(buyer);
<emptyLabel> Class(seller);
<emptyLabel> seller(dell);
<emptyLabel> buysProduct(acme, computers);
```

Next we show a simple example of expressing an object-oriented default inheritance ontology using rules. In the example, *BuyWithCredit* is a subclass of *Buy*. *Buy* assigns the value "invoice" to the *paymentMode* property, but *BuyWithCredit* assigns the value "credit" to the *paymentMode* property, i.e., *BuyWithCredit* overrides the *paymentMode* property inherited by default from *Buy*. The courteous feature of SCLP RuleML is a powerful way to express default inheritance using rules. If only Buy(quoteID) is asserted (i.e. the buyer asserts that it wants to buy), then the payment mode is assumed to be invoice (by default). If the buyer specifically asserts BuyWithCredit(quoteID), then the default payment mode is overridden to be credit instead.

```
<buyRegular>if Buy(?quoteID) then paymentMode(?quoteID,invoice);
/* BuyWithCredit is a subclass of Buy */
if BuyWithCredit(?quoteID) then Buy(?quoteID);
<buyCredit> if BuyWithCredit(?quoteID)then paymentMode(?quoteID,credit);
overrides(buyCredit, buyRegular);
```

3.2.4 Effectors

Effectors are a feature of the Situated extension of logic programs. An effector procedure is an attached procedure that is associated with a particular predicate. This association is specified via an effector statement that is part of the rulebase. When a conclusion is drawn about the predicate, an action is triggered; this action is the invocation of the effector procedure, and is side-effect-ful. In general, there may be multiple such effector statements and procedures in a given rulebase, e.g.., in a given SweetDeal contract/proposal. Effectors can execute real-world business processes associated with the execution of the contract. For example, an effector can be used by the buyer to send the purchase order (PO) to the seller (see <sendPO> rule below). If the vendor proposal has been approved, then the buyer sends the PO to the sales e-mail address of the vendor. The effector *sendPOtoVendor* is associated with the Java procedure *emailMessage* in the *Effector_EmailPO* class, whose path is indicated as *com.ibm.commonrules.examples.situated_programming_examples.familymsg.aprocs*.

The Java procedure not shown here for brevity handles the e-mail messaging aspect of sending the PO to the vendor. The arguments to the effector predicate – seller e-mail address, location of the purchase order, approved proposal identifier – are passed as arguments to the Java procedure.

```
<sendPO>
if
  approvedVendorProposal(?Vendor, ?ProposalID) AND
  emailSalesAddress(?Vendor, ?SellerAddress) AND locationOfPO(?Location)
then
  sendPOtoVendor(?SellerAddress, ?Location, ?ProposalID);

<emptylabel>
   Effector: sendPOtoVendor
   Class: Effector_EmailPO
   Method: emailMessage
   path:
"com.ibm.commonrules.examples.situated_programming_examples.familymsg.aprocs";
```

3.2.5 Fact-Queries or F-Queries

The traditional notion of the answer to a query in logic programs (and databases) is: a set of variable-binding lists. In modeling the exchange of contract proposals and associated dialogue between contracting parties, however, it is often convenient to model the answer to an inquiry as a set of facts instead. Accordingly, we have developed the design of f-queries (short for "fact queries") as a (fairly simple) experimental extension to RuleML. Note that, unlike the rest of what we describe of the SweetDeal approach in this paper, this f-queries feature is *not yet implemented in SweetRules*. RuleML f-queries are queries which have facts as their answers. They facilitate the iterated development of procurement contracts. The example below shows a sample f-query. It is an f-query from buyer to seller in which the buyer requests the seller for the unitPriceOfItem. The answer to the f-query is provided by the seller as a RuleML fact.

```
Query Example
<query>
  <_body>
    <fclit cneg="no" fneg="no">
      <_opr>
        <rel>unitPriceOfItem</rel>
      </_opr>
      <var>QuoteID</var>
      <var>Price</var>
    </fclit>
  </_body>
</query>
```

3.3 Agent Communication Knowledge Bases

In addition to the contract knowledge bases that are shared/exchanged, the agents also have internal RuleML knowledge bases that contain rules to facilitate agent communication. The effectors feature of SCLP RuleML allows the agents to execute real-world business processes such as e-mail messaging. This feature is used by the agents to send the contract rulesets to each other. The actual e-mail messaging effector procedure is implemented as a Java method that employs the JavaMail API [14]. The communication process is triggered using the internal agent communication KB and the SweetRules V2.1 software that supports execution of Java methods attached as effectors to specified predicates in the KB. A simple example follows: the situated rule <sendRFP> allows the buyer to send the RFP ruleset to the sales e-mail address of the seller. The name of the effector in the situated rule is sendRFPtoComputerSeller. The effector specification consists of the name of the Java procedure (emailMessage), the Java implementation class that contains the method (Effector_EmailRFP), and the path to the class (com.ibm.commonrules.examples.situated_programming_examples.familymsg.aprocs).

The effector is executed when the buyer wants to buy computers and the seller sells computers and is in the preferred vendor list of the buyer. When the sendRFPtoComputerSeller predicate is concluded, the attached procedure "emailMessage" is called to execute the required action. The action consists of reading the RFP from the local file system and sending it via e-mail to the specified e-mail address of the sales department of the seller. For brevity, the Java code to implement the e-mail messaging is not shown here.

```
<sendRFP>
if
   wantToBuy(?Buyer, computers) AND seller(?Vendor) AND
     sell(?Vendor, computers) AND inPreferredVendorList(?Buyer, ?Vendor) AND
     emailSalesAddress(?Vendor, ?Address) AND
     locationofRFP(?Buyer, computers, ?Location)
then
   sendRFPtoComputerSeller(?Address, ?Location);

<emptylabel>
   Effector: sendRFPtoComputerSeller
   Class: Effector_EmailRFP
   Method: emailMessage
   path:
"com.ibm.commonrules.examples.situated_programming_examples.familymsg.aprocs";
```

4 Contract Business Provisions Using RuleML

In this section, we present a few key contract fragments in the procurement contracting scenario and how SCLP RuleML can be used to express them. We intend to show how the expressive/declarative power of RuleML allows for easy addition and modification of key B2B contracting provisions. Specifically, we focus on expressing commonly used financial incentives such as rebates, discount pricing, and financing options. These incentives could be specified by the seller in its proposal. For the sake of simplicity and brevity, in this paper version some of the rules (e.g., about monthly payments in financing options) are highly specific to the particular scenario, rather than specified in more realistically general form.

4.1 Rebate

For example: the seller wishes to offer a rebate in the amount of $1000 to the buyer if the quantity of item ordered is greater than 75. This is represented as the <rebate> rule below.

```
<rebate>
if
      quoteID(?QuoteID) AND quantityOfItemOrdered(?Q) AND
      isGreaterThan(?Q, seventyfive)
then
      rebateAmount(?QuoteID, thousand);
```

4.2 Pricing Options

For example: If the buyer makes the purchase before April 1 then the unit price offered by the seller is $600; if the purchase is made before April 15, then the unit price offered is $650. This is specified as the <earlyPurchase> and <latePurchase> rules below. If both these rules apply, i.e., if the purchase was made before April 1, then precedence is given to the earlyPurchase rule. This precedence is specified using the courteous prioritization feature of SCLP (and of RuleML): see the overrides fact rule below.

```
<earlyPurchase>
if
      quoteID(?QuoteID) AND purchaseDate(?QuoteID, ?Date) AND
      isLessThan(?Date, oneApr05)
then
      unitPriceOfItem(?QuoteID, sixhundred);

<latePurchase>
if
      quoteID(?QuoteID) AND purchaseDate(?QuoteID, ?Date) AND
      isLessThan(?Date, fifteenApr05)
then
      unitPriceOfItem(?QuoteID, sixhundredfifty);

overrides(earlyPurchase, latePurchase);

MUTEX
      unitPriceOfItem(?QuoteID, sixhundred) AND
      unitPriceOfItem(?QuoteID, sixhundredfifty);
```

4.3 Financing Option

For example: If the financing is requested for 36 months by the buyer, the unit price of the item is determined to be $600, and the quantity ordered is 50, then the financing option offered by the seller is such that the monthly payment is $958 and the total interest paid is $4500 (see the <financing> rule below).

```
<financing>
if
     quoteID(?QuoteID) AND financeForMonths(?QuoteID, thirtysixMonths) AND
     unitPriceOfItem(?QuoteID, sixhundred) AND
     quantityOfItemOrdered(?QuoteID, fifty)
then
     monthlyPayment(?QuoteID, ninehundredfiftyeight) AND
     totalInterest(?QuoteID, fourthousandfivehundred);
```

5 Details of Procurement Contract Construction Using RuleML and SweetRules V2.1

In this section, we describe in detail the specific steps taken in constructing an e-contract between the buyer and seller using SCLP RuleML and SweetRules V2.1 in our electronic procurement scenario.

As described earlier, the buyer has a solo (or unshared) agent communication knowledge base that can be used to initiate the action of sending an RFP to a specific seller (in our example – Dell). We call this solo knowledge base – BSO1. BSO1 has the names of the different sellers, types of products offered by them, their respective sales e-mail addresses, and whether the sellers are in the preferred vendor list maintained by the buyer. The location of the RFP (which itself is a rule-based knowledge base) is indicated using the *locationofRFP* predicate. The rule that triggers sending the RFP to the seller is indicated by <sendRFP>: if the buyer wants to buy computers and the seller sells computers and is in the preferred vendor list of the buyer, send the RFP from the indicated local filesystem location to the seller's sales e-mail address. The predicate *sendRFPtoComputerSeller* is associated with the situated effector procedure *emailMessage*, which uses the JavaMail API to send the RFP ruleset to the seller via e-mail.

Buyer Solo KB – BSO1

```
wantToBuy(acme, computers);
seller(dell);
seller(staples);
sell(dell, computers);
sell(staples, officesupplies);
inPreferredVendorList(acme, dell);
inPreferredVendorList(acme, staples);
emailSalesAddress(dell, "sales@dell.com");
emailSalesAddress(staples, "sales@staples.com");
locationofRFP(acme, computers, "c:\\buyertosellerRFP.clp");

<sendRFP>
if
   wantToBuy(?Buyer, computers) AND seller(?Vendor) AND
  sell(?Vendor, computers) AND inPreferredVendorList(?Buyer, ?Vendor) AND
  emailSalesAddress(?Vendor, ?Address) AND
  locationofRFP(?Buyer, computers, ?Location)
then
  sendRFPtoComputerSeller(?Address, ?Location);

<emptylabel>
   Effector: sendRFPtoComputerSeller
   Class: Effector_EmailRFP
   Method: emailMessage
   path:
"com.ibm.commonrules.examples.situated_programming_examples.familymsg.aprocs";
```

In SweetRules V2.1, the "exhaustForwardInfer" command is given to derive all the conclusions from a given rulebase, and along with those conclusions to perform all the associated effecting actions that those conclusions trigger (i.e., sanction). For example, the command C2 below generates all the conclusions of BSO1 and (as an effecting action) sends the RFP to the seller. The "clp" in the first two arguments of the command indicates that CommonRules V3.3. format is the input and output knowledge base format, the third argument gives the location of BSO1, and the fourth argument specifies that IBM CommonRules should be used indirectly as an underlying inference engine when performing inferencing. SweetRules V2.1 software allows for a choice of such underlying engines. In our example, SweetRules enables Jess or XSB, as well as CommonRules, to be used as indirect underlying engine; for each choice of underlying engine, it would generate semantically equivalent conclusions and perform the same set of triggered effecting actions

```
exhaustForwardInfer clp clp c:\buyertosellerSendRFP.clp CommonRules (C2)
```

The RFP sent by the buyer to the seller is a collection of rules. The RFP consists of two parts – a shared knowledge base that contains most importantly the required computer configuration details (we call this knowledge base BSH1) and a set of f-queries that request specific answers from the seller (we call this set of queries BFQ1).

BSH1 indicates the buyer name, quantity of item ordered, buyer state, and the required computer configuration details. The rule <checkOfferedConfiguration> is used by the buyer to check whether the vendor offered configuration satisfies the minimum requirements. Since RuleML built-ins are not currently directly and smoothly supported in SweetRules V2.1 beyond the SWRL subset of RuleML, we also provide several facts to support arithmetic comparison.

Buyer to Seller RFP (BSH1)

```
buyerName(acme);  /* buyer name is acme */
quantityOfItemOrdered(fifty);  /* quantity of item ordered is fifty */

/* buyer is located in the state of Florida */
buyerLocationState(florida);

/* speed of processor should be at least 2GHz */
requiredMinProcessorSpeedInGHZ(twogigahertz);
if
 requiredMinProcessorSpeedInGHZ(?Speed) and
 offeredProcessorSpeedInGHZ(?OfferSpeed) and isGreaterThan(?OfferSpeed, ?Speed)
then
 isSpeedAcceptable(true);

/* not shown here for brevity:  there are also additional computer system configu-
ration details (memory size, hard disk storage capacity, monitor size, monitor
type (flat?), monitor resolution) */ ...
...

/* check if the configuration is acceptable */
<checkOfferedConfiguration>
if
  isSpeedAcceptable(true) and isMemorySizeAcceptable(true) and
  isHardDiskSizeAcceptable(true) and isMonitorSizeAcceptable(true) and
  offeredMonitorType(flat) and
  offeredMonitorResolution(tenTwentyFourBySevenSixtyEight)
then
  isOfferedConfigurationAcceptable(true);

/* The following are some facts in lieu of arithmetic built-ins. */
isGreaterThan(fourgigahertz, twogigahertz);
isGreaterThan(onezerotwofourmb, fivetwelvemb);
isGreaterThan(sixtyGB, fortyGB);
isGreaterThan(seventeen, fifteen);
```

BFQ1 is the collection of f-queries that ask the seller to specify the vendor quote identifier, the offered computer configuration details, the unit price of item, taxes as percent of price, service charge as percent of price, delivery charges for shipment, and the delivery time in days. For brevity, only a few of the f-queries are shown below.

Buyer to Seller f-Queries (BFQ1)

```
<rulebase>
  <_rbaselab>
    <ind>FQueries</ind>
  </_rbaselab>
  <query>
    <_body>
      <fclit cneg="no" fneg="no">
        <_opr>
          <rel>quoteID</rel>
        </_opr>
        <var>QuoteID</var>
      </fclit>
    </_body>
  </query>
  <query>
    <_body>
```

```
      <fclit cneg="no" fneg="no">
        <_opr>
          <rel>offeredProcessorSpeedInGHZ</rel>
        </_opr>
        <var>Speed</var>
      </fclit>
    </_body>
  </query>
...
```

After the seller receives the RFP, the seller sends its rule-based contract proposal to the buyer. The proposal contains three parts – BSH1 (i.e. shared knowledge base transmitted from buyer to seller – see above), answers to f-queries posed by the buyer plus the shared knowledge base that contains rules about pricing, rebates, financing options and other business provisions (we call this SSH1), and lastly f-queries for the buyer (SFQ1).

Seller to Buyer (SSH1)

```
/* quote ID is 1 */
quoteID(one);
/* computer configuration details */
offeredProcessorSpeedInGHZ(fourgigahertz);
offeredSizeofmemoryInMB(onezerotwofourmb);
offeredSizeofharddiskInGB(sixtyGB);
offeredMonitorSizeInInches(seventeen);
offeredMonitorType(flat);
offeredMonitorResolution(tenTwentyFourBySevenSixtyEight);

/* Pricing Rules */
/* if purchase date is before April 1 2005, then unit Price is $600;
   if purchase date is before April 15 2005, then unit Price is $650*/
<earlyPurchase>
if
        quoteID(?QuoteID) and purchaseDate(?QuoteID, ?Date) and
        isLessThan(?Date, oneApr05)
then
        unitPriceOfItem(?QuoteID, sixhundred);
<latePurchase>
if
        quoteID(?QuoteID) and purchaseDate(?QuoteID, ?Date) and
        isLessThan(?Date, fifteenApr05)
then
        unitPriceOfItem(?QuoteID, sixhundredfifty);

overrides(earlyPurchase, latePurchase);

MUTEX
        unitPriceOfItem(?QuoteID, sixhundred) and
        unitPriceOfItem(?QuoteID, sixhundredfifty);

/* there is no service charge */
if
        quoteID(?QuoteID)
then
        serviceChargeAsPercentOfPrice(?QuoteID, zeroPercent);

/* Delivery Options */

/* if delivery type is standard then delivery charge is $2500 for the order
   if delivery type is express then delivery charge is $5000 for the order
*/
<standard>
if
        quoteID(?QuoteID) and deliveryType(?QuoteID, standard)
then
        deliveryChargesForShipment(?QuoteID, twentyfivehundred);
```

```
<express>
if
        quoteID(?QuoteID) and deliveryType(?QuoteID, express)
then
        deliveryChargesForShipment(?QuoteID, fivethousand);
MUTEX
        deliveryType(?QuoteID, standard) and deliveryType(?QuoteID, express);

/* if delivery type is standard then delivery time in days is 14 days
   if delivery type is express then delivery time in days is 7 days
*/
<standardDeliveryTime>
if
        quoteID(?QuoteID) and deliveryType(?QuoteID, standard)
then
        deliveryTimeInDays(?QuoteID, fourteendays);

<expressDeliveryTime>
if
        quoteID(?QuoteID) and deliveryType(?QuoteID, express)
then
        deliveryTimeInDays(?QuoteID, sevendays);

MUTEX
        deliveryTimeInDays(?QuoteID, fourteendays) and
        deliveryTimeInDays(?QuoteID, sevendays);

/* Additional assertions from Seller */
/* Financial Incentives section */
/* not shown here for brevity: the financial incentives of discount pricing,
rebate, financing option already shown above in section 4 */
...
/* Sales Tax */
/* no sales tax in Florida */
<tax0>
if
        quoteID(?QuoteID) and buyerLocationState(florida)
then
        taxesAsPercent(?QuoteID, zeroPercent);

/* 5% sales tax in states other than Florida */
<tax5>
if
        quoteID(?QuoteID) and buyerLocationState(?X) and NotEquals(?X, florida)
then
        taxesAsPercent(?QuoteID, fivePercent);

MUTEX
        taxesAsPercent(?QuoteID, zeroPercent) and
        taxesAsPercent(?QuoteID, fivePercent);

/* Object-oriented default inheritance using rules */
/* If you buy, then default payment mode is invoice */
<buyRegular> if Buy(?QuoteID) then
paymentMode(?QuoteID, invoice);

/* BuyWithCredit is a subclass of Buy */
if BuyWithCredit(?QuoteID) then Buy(?QuoteID);

<buyCredit>
if BuyWithCredit(?QuoteID) then paymentMode(?QuoteID, credit);

overrides(buyCredit, buyRegular);

MUTEX
        paymentMode(?QuoteID, credit) and paymentMode(?QuoteID, invoice);

isLessThan(twentyfiveMarch05, oneApr05);
isLessThan(twentyfiveMarch05, fifteenApr05);
isLessThan(fiveApr05, fifteenApr05);
isGreaterThan(eighty, seventyfive);
NotEquals(massachusetts, florida);
```

SFQ1 is a collection of f-queries posed by the seller for the buyer. The seller asks whether the buyer would like to buy and whether the buyer would like to buy with a credit card. The seller also queries for the purchase date, delivery type and number of months of financing requested. For brevity, only a few of the f-queries are shown below.

Seller to Buyer F-Queries (SFQ1)

```
<rulebase>
  <_rbaselab>
  <ind>FQueries</ind>
  </_rbaselab>
  <query>
    <_body>
       <fclit cneg="no" fneg="no">
         <_opr>
              <rel>purchaseDate</rel>
          </_opr>
         <var>QuoteID</var>
         <var>Date</var>
        </fclit>
    </_body>
  </query>
...
```

When the buyer receives the proposal ruleset from the seller, it answers the queries posed by the seller (see BA1 below) and then performs exhaustive inferencing on the resulting ruleset (BSH1 + SSH1 + BA1) to obtain the derived conclusion set (CS1). Logical inferencing allows the buyer to determine the key parameters (such as unit price, delivery charges, taxes, etc.) of the proposal and also whether the proposal meets minimum specified criteria in the RFP.

Answers to F-Queries posed by seller (BA1)

```
Buy(one);
BuyWithCredit(one);
purchaseDate(one, fiveApr05);
deliveryType(one, express);
financeForMonths(one, thirtysixMonths);
```

The conclusion set (CS1) tells the buyer that the offered configuration is acceptable, unit price of item will be $650, delivery time will be 7 days, % discount already included in the price is 13%, taxes are 5%, rebate amount is $1000, and payment mode is credit.

Conclusion Set (CS1) obtained from BSH1 + SSH1 + BA1

```
isLessThan(twentyfiveMarch05, oneApr05);
isLessThan(twentyfiveMarch05, fifteenApr05);
isLessThan(fiveApr05, fifteenApr05);
requiredMinProcessorSpeedInGHZ(twogigahertz);
quoteID(one);
requiredMinSizeofmemoryInMB(fivetwelvemb);
offeredSizeofmemoryInMB(onezerotwofourmb);
requiredMonitorResoluton(tenTwentyFourBySevenSixtyEight);
purchaseDate(one, fiveApr05);
quantityOfItemOrdered(eighty);
BuyWithCredit(one);
deliveryType(one, express);
```

```
NotEquals(massachusetts, florida);
isGreaterThan(fourgigahertz, twogigahertz);
isGreaterThan(onezerotwofourmb, fivetwelvemb);
isGreaterThan(sixtyGB, fortyGB);
isGreaterThan(seventeen, fifteen);
isGreaterThan(eighty, seventyfive);
creditCardNumber(one, ccNumber9876543298765432);
offeredSizeofharddiskInGB(sixtyGB);
overrides(earlyPurchase, latePurchase);
overrides(earlyPurchaseDiscount, latePurchaseDiscount);
overrides(buyCredit, buyRegular);
offeredMonitorSizeInInches(seventeen);
requiredMinSizeofharddiskInGB(fortyGB);
offeredProcessorSpeedInGHZ(fourgigahertz);
financeForMonths(one, thirtysixMonths);
requiredMonitorType(flat);
offeredMonitorType(flat);
buyerName(acme);
buyerLocationState(massachusetts);
requiredMinMonitorSizeInInches(fifteen);
offeredMonitorResolution(tenTwentyFourBySevenSixtyEight);
vendorName(dell);
serviceChargeAsPercentOfPrice(one, zeroPercent);
deliveryChargesForShipment(one, fivethousand);
isSpeedAcceptable(true);
Buy(one);
rebateAmount(one, thousand);
isMonitorSizeAcceptable(true);
isMemorySizeAcceptable(true);
isHardDiskSizeAcceptable(true);
isOfferedConfigurationAcceptable(true);
deliveryTimeInDays(one, sevendays);
discountPercentAlreadyIncluded(one, thirteen);
unitPriceOfItem(one, sixhundredfifty);
taxesAsPercent(one, fivePercent);
paymentMode(one, credit);
```

6 Relationship of Other B2B Technologies to Our Approach

RosettaNet and ebXML are two very important and influential approaches to XML-based e-business messaging including about contracting and e-commerce. It is desirable to be able to use our SweetDeal approach together with such XML-based e-business messaging infrastructure. In this section, we discuss how SweetDeal and (SCLP) RuleML can be used with RosettaNet and with ebXML. The punchline is that they play well together; the SweetDeal contract rulesets can be carried as the "letters" content within the "envelopes" of RosettaNet or ebXML messages, i.e., within their messaging interfaces and protocols. In doing so, it is both possible and useful to utilize the (non-OWL) ontologies provided by RosettaNet and ebXML, and to perform sending of messages as actions.

6.1 RosettaNet

Next, we begin with RosettaNet, and discuss specifically how RosettaNet Partner Interface Processes (PIPs) can be used with RuleML in the context of our electronic procurement scenario. RosettaNet is a consortium of information technology, electronic components, semiconductor manufacturing and solutions providers, which seeks to establish a common language and standard processes for business-to-business

(B2B) transactions. RosettaNet PIPs define business processes between trading partners. The PIP specifies the roles of the trading partners that participate in the business process as well as the business activities that compose the process. The PIP also specifies XML-based action messages or business documents that are exchanged between the roles during business activities. The specification of a standard structure for the business documents is a major part of the PIP specification. An example of a RosettaNet PIP is PIP3A1 which provides a detailed XML message guideline for implementing the Request Quote business process. A message fragment from PIP3A1 is shown below –

```
<ContactInformation>
    <contactName>
          <FreeFormText>A</FreeFormText>
          <EmailAddress>abc@xyz.com</EmailAddress>
          …..
    </contactName>
</ContactInformation>
```

The message fragment above specifies the structure for contact information for the buyer who sends the request for quote to the seller. Our SweetDeal approach can be used straightforwardly in combination with the exchange of RosettaNet PIP messages between the two parties. We can also directly use the standardized (non-OWL) ontological terms from the PIP messages in our rulebases. For example, the request for proposal (RFP) sent by the buyer to the seller in our scenario allows for use of the ontological terms in the RosettaNet PIP3A1 XML message guidelines. A SweetDeal quote (contract proposal) rulebase cf. our earlier scenario can then employ as predicates (i.e., as ontological terms) various properties drawn from the PIP specification, e.g., the unit price of the product, which is specified in RosettaNet using the following DTD segment –

```
<!ELEMENT unitPrice ( ProductPricing ) >
<!ELEMENT ProductPricing ( FinancialAmount , GlobalPriceTypeCode ) >
<!ELEMENT FinancialAmount ( GlobalCurrencyCode , MonetaryAmount ) >
<!ELEMENT GlobalCurrencyCode ( #PCDATA ) >
<!ELEMENT MonetaryAmount ( #PCDATA ) >
```

For example, the seller would specify the following fact rule in the proposal to the buyer:

```
unitPrice(?GlobalCurrencyCode, ?MonetaryAmount).
```

6.2 ebXML

Likewise, ebXML can be used in our scenario along with RuleML and the SweetDeal approach to support electronic contracting between two parties. Both the buyer and the seller in our scenario would maintain ebXML collaboration protocol profiles (CPPs) that would describe the specific business collaborations supported by each of the parties using the ebXML business process specification schema (BPSS). For example, the buyer CPP would show that the "request for proposal" is a business process that is supported by it. The details of the "request for proposal" business process would be specified using the ebXML BPSS. The parties that will engage in the interaction protocol will reach agreement on how to collaborate by exchanging the CPPs to construct a collaboration protocol agreement (CPA), which fixes the protocol for interaction between the parties. Once agreement has been reached, ebXML messages in accordance with the collaboration agreement can be exchanged using ebMS (or ebXML Message Service). The payload of these messages can contain the RuleML rulebases to establish the electronic procurement contract.

7 Conclusions

In this paper, we have extended the SweetDeal approach and applied the extended approach using the new SweetRules V2.1 semantic web rules prototype software to a practical, real-world B2B application in the domain of electronic contracting. The electronic procurement contracting scenario that we have described in detail shows how semantic web rules technology, specifically RuleML and SweetRules, can be powerfully used in e-contracting.

Acknowledgements

Thanks to Shashidhara Ganjugunte, Chitravanu Neogy, Said Tabet, and the rest of the SweetRules development team, for helping to realize the implementation, and for useful discussions.

References

1. Rule Markup Language Initiative, http://www.ruleml.org and http://www.mit.edu/~bgrosof/#RuleML
2. SweetRules, http://sweetrules.projects.semwebcentral.org/
3. OWL and the Semantic Web Activity of the World Wide Web Consortium. http://www.w3.org/2001/sw
4. RosettaNet, http://www.rosettanet.org
5. ebXML (ebusiness XML) standards effort, http://www.ebxml.org
6. Grosof, B.N., "Representing E-Business Rules for Rules for the Semantic Web: Situated Courteous Logic Programs in RuleML". Proc. Wksh. on Information Technology and Systems (WITS '01), 2001.
7. XSB logic programming system. http://xsb.sourceforge.net/
8. Jess (Java Expert System Shell). http://herzberg.ca.sandia.gov/jess/
9. Jena, http://jena.sourceforge.net/
10. IBM CommonRules. http://www.alphaworks.ibm.com and http://www.research.ibm.com/rules/
11. SWRL, A Semantic Web Rule Language Combining OWL and RuleML, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/
12. Grosof, B.N., C.T. Poon, "SweetDeal: Representing Agent Contracts With Exceptions using Semantic Web Rules, Ontologies, and Process Descriptions". International Journal of Electronic Commerce (IJEC), 8(4):61-98, Summer 2004, Special Issue on Web E-Commerce.
13. Grosof, B.N., Horrocks, I., Volz, R., and Decker, S., "Description Logic Programs: Combining Logic Programs with Description Logic". Proc. 12th Intl. Conf. on the World Wide Web (WWW-2003).
14. JavaMail, http://java.sun.com/products/javamail/
15. OWL Web Ontology Language, http://www.w3.org/TR/owl-features/
16. Semantic Web Services Framework Version 1.0, http://www.daml.org/services/swsf/1.0

Using SWRL and OWL to Capture Domain Knowledge for a Situation Awareness Application Applied to a Supply Logistics Scenario

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

[1] Versatile Information Systems, Inc., Framingham, Massachusetts USA
cmatheus@vistology.com
http://www.vistology.com
[2] Northeastern University, Boston, Massachusetts USA
ken@baclawski.com, mkokar@ece.neu.edu
[3] Western New England College, Springfield, MA, USA
jletkows@wnec.edu

Abstract. When developing situation awareness applications we begin by constructing an OWL ontology to capture a language of discourse for the domain of interest. Such an ontology, however, is never sufficient for fully representing the complex knowledge needed to identify what is happening in an evolving situation – this usually requires general implication afforded by a rule language such as SWRL. This paper describes the application of SWRL/OWL to the representation of knowledge intended for a supply logistics scenario. The rules are first presented in an abstract syntax based on n-ary predicates. We then describe a process to convert them into a representation that complies with the binary-only properties of SWRL. The application of the SWRL rules is demonstrated using our situation awareness application, SAWA, which can employ either Jess or BaseVISor as its inference engine. We conclude with a summary of the issues encountered in using SWRL along with the steps taken in resolving them.

1 Introduction

The problem of Situation Awareness involves the context-dependent analysis of the characterization of objects as they change over time in an evolving situation with the intent of establishing an understanding of "what is going on". Classic examples of tasks where situation awareness is of great importance include air traffic control, crisis management, financial market analysis and military battlespaces. For domains such as these, significant effort has gone into both understanding the problem and developing automated techniques and applications for establishing situation awareness [1]. A key part of this problem is identifying relations among the objects that are relevant to the situation and to the goals of the situation analyst. For any non-trivial situation the number of possible relations that might be considered is so vast that it is necessary to reduce the space of candidate relations by using additional knowledge about the situation's domain and about the specific objectives of the current situation. In an automated system designed to assist in establishing situation awareness this additional knowledge falls under the broad heading of "domain knowledge" and its use requires some form of knowledge representation (KR).

In our work on developing situation awareness systems we have been exploring the use of Semantic Web technologies for domain knowledge representation. We have found the OWL Web Ontology Language to be a very useful means for capturing the basic classes and properties relevant to a domain. These domain ontologies establish a language of discourse for eliciting more complex domain knowledge from subject matter experts (SME). Due to the nature of OWL, these more complex knowledge structures are either not easily represented in OWL or, in many cases, are not representable in OWL at all. The classic example of such a case is the relationship uncleOf(X,Y). This relation, and many others like it, requires the ability to constrain the value of a property (brotherOf) of one term (X) to be the value of a property (childOf) of the other term (Y); in other words, the siblingOf property applied to X (i.e., brotherOf(X,Z)) must produce a result Z that is also a value of the childOf property when applied to Y (i.e., childOf(Y,Z). This "joining" of relations is outside of the representation power of OWL. One way to represent knowledge requiring joins of this sort is through the use of the implication (→) and conjunction (AND) operators found in rule-based languages. The rule for the uncleOf relationship appears as follows:

```
brotherOf(X,Z) AND childOf(Y,Z) → uncleOf(X,Y)
```

We initially started developing the complex knowledge structures needed for our situation awareness applications using RuleML [2] as described in [4]. With the introduction of the Semantic Web Rule Language [3] (SWRL) we decided to investigate the potential for its use in our applications. SWRL was attractive because of its close connection with OWL DL and the fact that it, like OWL, has well-defined semantics. The two biggest drawbacks we saw at the time were its restriction to binary predicates (a characteristic inherited from OWL) and the lack of tools, in particular editors and consistency checkers. We confronted the lack of tools in part by developing our own graphical editor for SWRL called RuleVISor, but there is still an outstanding need for tools to check for consistency 1) within SWRL rules, 2) across SWRL rules and 3) between SWRL rules and the OWL ontologies upon which they are built. As for the issue of binary predicates we employ an approach by which n-ary predicates, such as the unconstrained predicates permitted by RuleML, can be systematically converted into binary predicates represented in SWRL; we describe this approach in Section 5 of this paper.

As we worked further on developing and using SWRL rules we encountered a number of additional issues that needed to be addressed before a practical implementation of our application could be realized. These issues include 1) the lack of negation as failure, 2) the need for procedural attachments and 3) the implementation of SWRL built-ins. Other concerns of particular importance to situation awareness – such as the representation of time, data pedigree and uncertainty – are not explicitly addressed in either SWRL or OWL; in fact, for the case of time (more specifically the changes of property values over time) the languages' monotonicity assumption technically precludes them or at least requires significant extra effort to circumvent the imposed constraints. We reported on some of these issues in our position paper and presentation [4] at the W3C Workshop on Rule Languages for Interoperability [5] and further elaborate on them in this paper.

The primary intent of this paper is to describe our experience of using SWRL and OWL to represent the domain knowledge for a supply logistics scenario and show

how this knowledge was employed in our situation awareness application, SAWA.[6, 7] We begin the paper by introducing the supply logistics "repairable assets" scenario and then describe the OWL ontology we developed to capture the scenario's key classes and properties. We then describe the domain knowledge rules for the scenario, starting with an abstract set of higher-order rules (i.e., rules that permit n-ary predicates) that are relatively easy to understand. These rules are then converted into an abstract representation in which the n-ary predicates have been converted to instances of classes representing the predicates and properties corresponding to the n-ary terms. These abstract rules are then converted into the less easy to read SWRL syntax. The SWRL rules are made operational by translating them into rules appropriate for interpretation by a forward-chaining inference engine – a process requiring additional operators such as `gensym`, `assert` and procedures to implement SWRL built-ins. The processing of these rules by SAWA is briefly summarized and a performance comparison is made between the use of two inference engines, BaseVISor (a Rete-based inference engine optimized for triples) and Jess (a Java implementation of the Rete-based CLIPS inference engine), either of which can be plugged into SAWA.

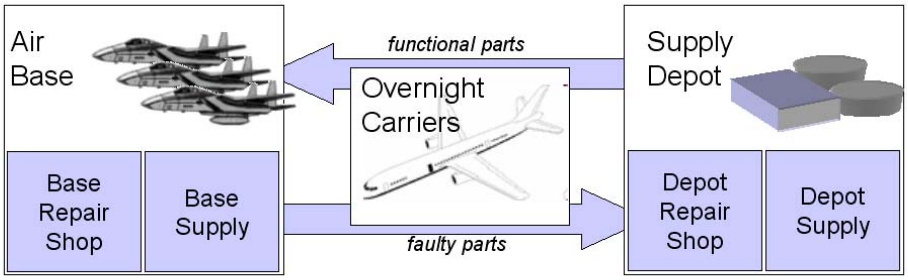
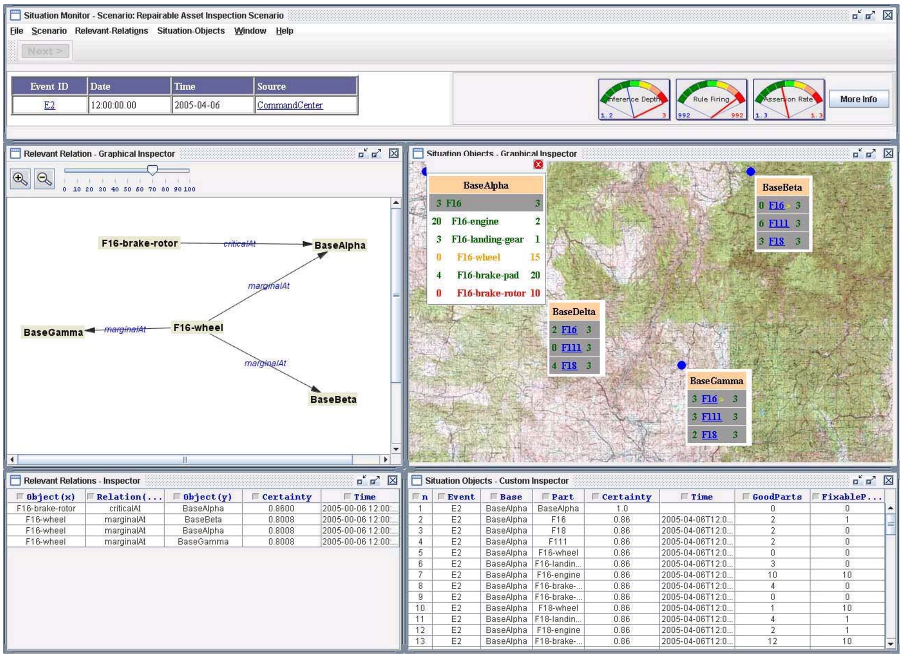


Fig. 1. Repairable Assets Pipeline

2 Repairable Assets Domain

With assistance from SMEs at AFRL Wright Research Site we analyzed a supply logistics scenario involving the monitoring of "repairable assets". Repairable assets for the USAF represent aircraft parts that when found to be malfunctioning on an aircraft can be repaired for reuse either locally at the airbase's repair shop or at a remote repair depot. The diagram in Figure 1 shows a simplified version of the repairable assets pipeline used by the USAF. Each airbase has a supply of aircraft parts maintained at its local base supply along with the capability of repairing certain types of parts at its local repair shop. Some parts cannot be repaired locally and so they are shipped (usually by commercial overnight carriers) to remote supply depots that have more extensive repair capabilities. The supply depots repair whatever parts they can and place them into their depot supplies from which they are shipped out to airbases as needed. Keeping repairable parts at a sufficient level across a collection of airbases is a complicated process as it involves an understanding of the aircraft based at each facility, the repairable parts needed by each aircraft type, the repair capabilities of each base and the supply levels and repair capacity of all remote supply facilities.

The specific objective of our application was to demonstrate the ability to effectively monitor the supply levels across a handful of airbases kept in supply by a set of



Event ID	Date	Time	Source
E2	12:00:00.00	2005-04-06	CommandCenter

Relevant Relations - Inspector

Object(x)	Relation(...	Object(y)	Certainty	Time
F16-brake-rotor	criticalAt	BaseAlpha	0.8600	2005-00-06 12:00:...
F16-wheel	marginalAt	BaseBeta	0.8008	2005-00-06 12:00:...
F16-wheel	marginalAt	BaseAlpha	0.8008	2005-00-06 12:00:...
F16-wheel	marginalAt	BaseGamma	0.8008	2005-00-06 12:00:...

Situation Objects - Custom Inspector

n	Event	Base	Part	Certainty	Time	GoodParts	FixableP...
1	E2	BaseAlpha	BaseAlpha	1.0		0	0
2	E2	BaseAlpha	F16	0.86	2005-04-06T12:0...	2	1
3	E2	BaseAlpha	F18	0.86	2005-04-06T12:0...	2	0
4	E2	BaseAlpha	F111	0.86	2005-04-06T12:0...	2	0
5	E2	BaseAlpha	F16-wheel	0.86	2005-04-06T12:0...	0	0
6	E2	BaseAlpha	F16-landin...	0.86	2005-04-06T12:0...	3	0
7	E2	BaseAlpha	F16-engine	0.86	2005-04-06T12:0...	10	10
8	E2	BaseAlpha	F16-brake-...	0.86	2005-04-06T12:0...	4	0
9	E2	BaseAlpha	F16-brake-...	0.86	2005-04-06T12:0...	0	0
10	E2	BaseAlpha	F18-wheel	0.86	2005-04-06T12:0...	1	10
11	E2	BaseAlpha	F18-landin...	0.86	2005-04-06T12:0...	4	1
12	E2	BaseAlpha	F18-engine	0.86	2005-04-06T12:0...	2	1
13	E2	BaseAlpha	F18-brake-...	0.86	2005-04-06T12:0...	12	10

Fig. 2. SAWA Interface for the Repairable Assets Domain

remote supply depots. While it would certainly be possible to develop a one-time, stand-alone application to achieve this task we were interested in demonstrating the applicability of our general-purpose situation awareness application, SAWA, to this problem. SAWA uses OWL and SWRL to represent knowledge relevant to a domain of situation awareness problems and then employs a generic inference engine, BaseVISor or Jess, to reason about a specific evolving situation. SAWA has been described elsewhere [6, 7] and so we will not go into the details of its workings in this paper. To provide a glimpse into what SAWA does, a screenshot of its interface adapted for the Repairable Assets domain is shown in Figure 2. The interface is comprised of five windows. The top-most window is the control pane in which resides the control menus, current event info and performance meters. In the middle right-hand window, a map depicts the physical locations of the airbases along with drillable summary sub-windows showing the status of the planes and parts present at each base. The middle left-hand window provides an interactive, graphical representation of the relations detected in the situation; these relations are also described in detail in the relations table that appears in the lower left-hand window. Detailed information about all of the objects and object attributes appear in the object table in the lower right-hand window.

3 Repairable Assets Ontology

As is our practice [4], we began the process of capturing the domain knowledge pertinent to the repairable assets problem by first developing an ontology in OWL. The primary objects in this ontology (shown in Figure 3) include airbases, airplanes, parts, facilities and remote supply depots. Because we are concerned with quantities of items such as parts and aircraft it was also necessary to create a QuantityOf class that permits the association of a numeric count with a specific plane or part type. In this way we can say that a particular facility has a QuantityOf instance relating a particular item with a specific number. It was also necessary to be able to associate each airbase with an ordered list of remote supply facilities available to provide additional parts; as shown, this was achieved using an rdf:List structure.

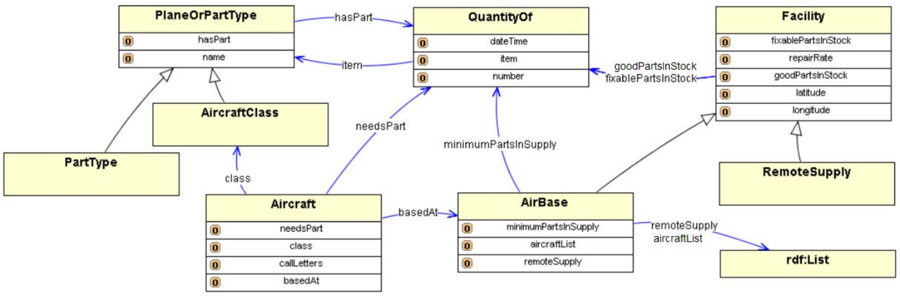


Fig. 3. Repairable Assets Ontology

Simulated data was constructed for this scenario consisting of the inventory of aircraft and parts at four airbases and three remote supply bases taken at various times. This data was annotated using both the Repairable Assets ontology and the Event ontology shown in Figure 5. Each event contained facility-specific information such as the quantity of good aircraft of each type, the quantity of aircraft parts in stock, and the quantity of fixable parts in stock along with the current need for parts that needed to be replaced on aircraft undergoing repair. In addition to this "event" data, a file of annotations was created containing descriptions of the various aircraft types and the parts that make them up, while another annotation file was constructed to provide descriptions of the specific airbases, their aircraft and their remote supply facilities.

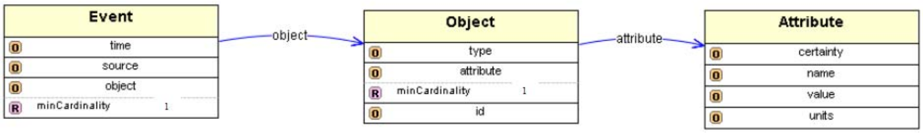


Fig. 4. Event Ontology

4 Repairable Assets Domain Rules

The objective in our repairable assets scenario is to monitor the supply levels of various parts at a number of airbases and compare them to the current needs for those parts by specific aircraft. We developed an initial set of SWRL rules to achieve this using RuleVISor, a graphical rule editor we developed at Versatile Information Systems. These rules deal with local and remote supply levels, local demand levels and repair rates on a per-part and per-facility basis, identifying when a specific part-type at a specific airbase is "critical", "marginal" or "nominal". In all there were nine rules, some of them recursive, that were developed for this task. The logic captured by these rules is shown here using an abstract Prolog-like Horn-clause representation, in which variables are prefaced with question marks:

```
criticalPartAtFacility(?Part, ?Facility, ?Time) :-
  localNeed(?Part, ?Facility, ?Time, ?Need)
  localSupply(?Part, ?Facility, ?Time, ?Supply)
  ?Need <= ?Supply.
 marginalPartAtFacility (?Part,?Facility,?Time) :-
  localSupply(?Part, ?Facility, ?Time, ?Supply)
  localRepairable(?Part, ?Facility, ?Time, ?Repairable)
  remoteAvailable(?Part, ?Facility, ?Time, ?RemoteSupply)
  surplusRequired(?Part, ?Facility, ?SurplusRequired)
?Supply + ?Repairable + ?RemoteSupply < ?SurplusRequired).
```

In plain English these rules state that a part at a facility is determined to be “critical” if the current demand at the facility exceeds the current local supply; it is classified as “marginal” if the total resuppliable rate for the part at the facility is below a required-surplus threshold; and it is deemed “nominal” otherwise (note, there are no rules for this state as it is the normal state of all parts that are neither marginal nor critical). The notion of “resuppliable rate” used in the marginalPartAtFacility rule represents the total number of parts of a specific type that a facility could have on hand by the next day if its current local supply level of that part, its current local “repair capacity” for that part and the current supply levels of that part at remote depots are all added together. If this total falls below the required-surplus level (a static number established by an SME) the part at that facility is given a status of “marginal’. It is common practice in the USAF to ship parts as needed between facilities by overnight delivery, which leads to the natural choice of a day as the basis for determining part resuppliability. The “repair capacity” at an airbase is a function of the number of parts waiting to be repaired locally, the repair capacity of the local repair shop and the status of any sub parts required for the repairs (this value is calculated by additional supporting rules not shown above).

5 Converting from N-Ary to Binary Predicates

As is evident in the abstract rules for criticalPartAtFacility and marginalPartAtFacility, we used predicates with more than two terms. We did this because it was the natural way to represent the critical concepts, all of which simultaneously involve a Part, a Facility and a Time; unfortunately such n-ary predicates are not permitted in SWRL. As a result, we needed to convert these rules into ones that contained only binary and unary predicates. The presentation of two design patterns usable for this purpose have been described by Noy and Rector [8]; our approach is in line with the second of these patterns. This conversion was done manually for this small set of rules but the process we employed is systematic enough to automate; for a set of rules defined in RuleML a single XSLT script would suffice. The approach involves converting the n-ary predicates into instances of unique classes (one for each predicate) that are then given properties corresponding to the each of their respective terms. The results of this process can be seen in the following rule corresponding to the marginalPartAtFacility rule described above.

```
if
    ;; find the Local Surplus/Deficit (from another rule)
    rdf:type(?SMNStatement, #SupplyMinusNeed)
    #smnPart(?SMNStatement, ?Part)
    #smnFacility(?SMNStatement, ?Facility)
    #smnTime(?SMNStatement, ?Time)
    #smnNumber(?SMNStatement, ?LocalSurplusOrDeficit)

    ;; find the number Locally Repairable (from another rule)
    rdf:type(?PLRStatement, #PartsLocallyRepairable)
    #localPart(?PLRStatement, ?Part)
    #localFacility(?PLRStatement, ?Facility)
    #localNumber(?PLRStatement, ?NumberRepairable)

    ;; find number Available Remotely (from another rule)
    rpa:remoteSupply(?Facility, ?RemoteSupplyList)
    rdf:type(?PRAStatement, #PartsAvailableAtRemoteFacility)
    #remotePart(?PRAStatement, ?Part)
    #facilityList(?PRAStatement, ?RemoteSupplyList)
    #remoteNumber(?PRAStatement,?NumberAvailableRemotely)
    #remoteTime(?PRAStatement, ?Time)

    ;; look up Minimum Threshold
    rpa:minimumPartsInSupply(?Facility,
                             #MinimumThresholdStatement)
    rpa:item(?MinimumThresholdStatement, ?Part)
    rpa:number(?MinimumThresholdStatement, ?MinimumThreshold)

    ;; add SurplusOrDeficit, Repairable, & RemotelyAvailable
    swrlb:add(?TotalAvailable, ?LocalSurplusOrDeficit,
        ?NumberRepairable, ?NumberAvailableRemotely)

    ;; test if the Threshold is greater than the Total
    swrlb:greaterThan(?MinimumThreshold, ?TotalAvailable)

then
    rdf:type(?MPFStatement, #MarginalPartAtFacility)
    #marginalPart(?MPFStatement, ?Part)
    #marginalFacility(?MPFStatement, ?Facility)
    #marginalTime(?MPFStatement, ?Time)
    #marginalNumber(?MPFStatement, ?TotalAvailable)
```

In the conclusion of the rule (i.e., the five lines of code after the "then") it can be seen that the marginalPartAtFacility(?Part,?Facility,?Time) predicate has been converted into an instance of a locally defined class MarginalPartAtFacility represented by the variable ?MPFStatement. To this instance three properties (marginalPart, marginalFacility and marginalTime) have been attributed with values corresponding to the variables ?Part, ?Facility and ?Time. (A fourth property "marginalNumber" is used to encode the degree to which the required surplus level has been unmet.) In the body of the rule there are three places where the predicates localSupply, localRepairable and remoteSupply from the abstract rule have been converted into statements referring to instances of local classes with properties corresponding to each of their original four terms. For example, the localSupply(?Part,?Facility,?Time,?Supply) predicate from the abstract rule is converted into a reference to an instance of the class SMNStatement (SMN stands for Supply Minus Need) which has the four properties - localPart, localFacility, localTime and localNumber - associated with the four terms corresponding to Part, Facility, Time and Supply, respectively. When this rule is processed, the inference engine will look for the occurrence of these class instances and their corresponding properties in working memory; it will only find them if other rules (not shown) have previously fired and as a result asserted these instances and properties.

Note that the classes used to stand in place of the n-ary predicates are all defined local to the rule set (as indicated by the preceding hash mark #) and are not a part of the main ontology described in Section 3. These classes do not in fact have to be explicitly defined in the rule set because the mere use of one as the object of the rdf:type property requires (by the axioms of RDF/OWL) that there be a local class identified by that reference; any OWL-compliant reasoner will infer its existence. The same holds true for the properties that stand in place of the predicate terms. The only requirements of these local classes are that they be uniquely named and that whenever one is used in the body of a rule that there be at least one rule that asserts an instance of the same class in its head, otherwise the first rule will never fire.

6 SWRL Rules and Their Execution

The SWRL code for the marginalPartAtFacility rule, developed with the help of RuleVISor, appears in Appendix A. The primary reason for including the SWRL code in this paper is to demonstrate how a relatively simple rule expressible in just six lines of high-level code mushrooms into a much more complex listing of over one hundred lines of code that is extremely difficult to interpret and even more difficult to debug. The nine rules making up the SWRL rule set for the Repairable Assets scenario amounted to nearly 1200 lines of code and demanded countless hours of debugging. One can argue that SWRL was never intended to be a language for the manual development of rules and that what is needed are more powerful editors that permit rules to be represented more abstractly and then compiled into SWRL for execution. Neither RuleVISor nor the SWRL editor provided with the Protégé OWL plug-in [9] provide this level of functionality – both simply permit the direct editing of SWRL code with its inherent constraint to binary predicates. The requirement to work at such a painfully low level of representation is a major hurdle for anyone wishing to use SWRL for even moderately complex tasks.

Assuming one has a set of SWRL rules, such as the repairable assets rules described above, there remains the question of how to execute them. There are no inference engines known to the authors that have full native support for SWRL rules. The Institut für Informatik, Freie Universität Berlin [10] has developed a prototype engine for SWRL but it does not (as of this writing) permit full OWL reasoning (only inheritance reasoning is supported) nor does it implement the SWRL built-ins. We have implemented a Jess-based reasoner that includes a reasonably complete set of axioms for RDF/OWL and supports a large subset of SWRL built-ins; this reasoner is at the heart of our consistency checking service ConsVISor [11]. We have also recently implemented a high-performance Java-based inference engine that incorporates a subset of the axioms of RDF/OWL sufficient to support the reasoning required for our repairable assets problem domain as well as support for the SWRL built-ins used by these rules. To execute SWRL rules in either the Jess or BaseVISor engines it is first necessary to translate them into the corresponding rule languages. In doing so, engine specific characteristics need to be accounted for that lie outside the scope of SWRL. Both Jess and BaseVISor are Rete-based, forward chaining inference engines that work by continuously evaluating the contents of working memory and firing rules when their antecedents are satisfied. The firing of a rule can result in the "assertion" of new facts into working memory but this requires an explicit call to the "assert" operator. There is nothing in SWRL that corresponds to the assert operator – the atoms in the head of a SWRL rule are simply inferred to be true whenever all of the atoms in the body are true but there is no notion of "working memory" into which facts must be asserted. In the translation from SWRL to Jess or BaseVISor it is necessary to surround all atoms in the heads of rules with the assert() operator; this is done automatically by XSLT translation scripts.

Because of the use of instances of classes to represent n-ary predicates it is necessary to be able to "generate" these instances as needed. These instances in need of generation exist as variables in the head of rules that have no corresponding occurrence in the body of the rule. When such a variable is detected the translation scripts add a gensym() operator to the head to generate a unique symbol to represent the instance. At first this technique seems to be at odds with the "safety" condition in which all variables in the head of a SWRL rule must be present in the body, but as suggested in the SWRL specification, it would be possible to add a someValuesFrom restriction on this variable in the body; we don't actually do this because it would have no bearing at all on the firing of the rule.

There is also an issue with implementing the SWRL built-ins. The definitions of the built-ins in the SWRL specification are given as relations with no explicit input/output designations assigned to their arguments. In our rules we have always found that we need to use the built-in capabilities as if they were functions in which you specify the values for all but one of the arguments, which becomes the output variable. What this lack of input/output designation in SWRL built-ins means from an implementation perspective is that the code for the built-ins must determine which of the arguments is supposed to be the output variable and then select the appropriate functional method to apply to the other arguments. For example, consider swrlb:add which can be applied to an arbitrary number of two or more arguments with the first argument being the sum of the remaining arguments. If you use the atom (swrlb:add ?X 1 2) the processor of this statement must detect that the first argument is unbound (i.e. a variable) and thus it becomes the output argument. Knowing that the value of the first argument is to be calculated the processor must apply its *summation* method to the remaining arguments. If, on the other hand, you pass (swrlb:add 10 ?X 5) to the processor, it needs to figure out that it should *subtract* 5 from 10 to calculate the value of the variable ?X. In this case we have used swrlb:add to do *subtraction* even though there is also a swrlb:subtract built-in. Now consider the case where more than one argument is unbound: (swrlb:add ?X 100 ?Y). What should the processor do in this case? According to the semantics of SWRL this is perfectly legal even though it would result in an infinite set. In our system we treat this case as an error, which it would be for the kinds of practical applications we are interested in. Alternatively, if one really needed the set of all relations consistent with the bound terms, procedures could be implemented that return some notational form from which the members of the set could be derived. In our cases, however, this approach would needlessly complicate the rules and make it cumbersome to deal with the most common case where what is really desired is a function.

A final issue encountered with the practical application of SWRL was the need for some form of negation as failure. Since there are so many parts on an aircraft it was highly desirable to require airbases to only report when there was a need for the repair of a specific part rather than report the status of all parts on the aircraft. The quite natural assumption in this case is that a part is working properly unless informed otherwise. This becomes an issue because we need to be able to count the number of pairs needing repair at an airbase which requires looking up the number of parts needed for each aircraft in the airbase's rdf:List of stationed aircraft. As the rules walk through this aircraft list they can only fire if there is something in working memory that triggers them. That is unless the not() operator is used in which case the absence of a particular fact can lead to the firing of a rule, which is exactly what we want. In our rules that count the number of a specific part needing repair at an airbase there is one rule that checks for the occurrence of a fact stating that an aircraft at that base needs that part and another rule looking for the absence of such a fact. In our scenario this is a perfectly reasonable thing to do since airbases are required to report on all parts needing repair. We can thus safely assume that our world is closed within the scope of the status reports sent out by the airbases. Both BaseVISor and Jess provide a not() operator that implements negation as failure (NAF). Although neither of them provide a scoped NAF operator (SNAF) it would be easy to define the specific scope in this case using the URI specifying the Event that transmitted a base's status report data for a specific Time. Our rules actually implement a notion of Event scoping as most of them include an explicit reference to an Event for a specific Time (see the beginning of the rule listed in Appendix A).

7 BaseVISor Versus Jess

We have used Jess as an inference engine for a number of RDF/OWL applications over the last couple years. On several occasions we developed code to extend its capabilities, most markedly in the area of SWRL built-ins and performance monitoring. When working with the internals of the Jess source code it becomes apparent that a lot of legacy code exists that is there to support the LISP-like list structures that CLIPS and OPS-5 supported. Since our RDF/OWL applications deal only with triples there is no need for all of the complexity enabled by Jess' data structures, which carry with them significant overhead particularly when doing lookups within the Rete network. When it came time to augment the Rete network with uncertainty processing capabilities (an important requirement for many situation awareness tasks) we took the opportunity to develop a triples-based Rete network from scratch. The result is BaseVISor, which has now replaced Jess as the core of SAWA. Space limitations prevent us from going into more details of the internals of BaseVISor or its support for SWRL; however, we do want to highlight the performance improvement afforded by BaseVISor over Jess as depicted in Figure 7. This graph compares the performance of BaseVISor versus Jess as the number of ground facts increases. Except for a very small portion at the lower left of the graph where BaseVISor is slightly slower (due to some index optimization that does not payoff on small data sets of less than 500 facts), BaseVISor outperforms Jess and shows a near linear rate of change compared to Jess' polynomial increase.

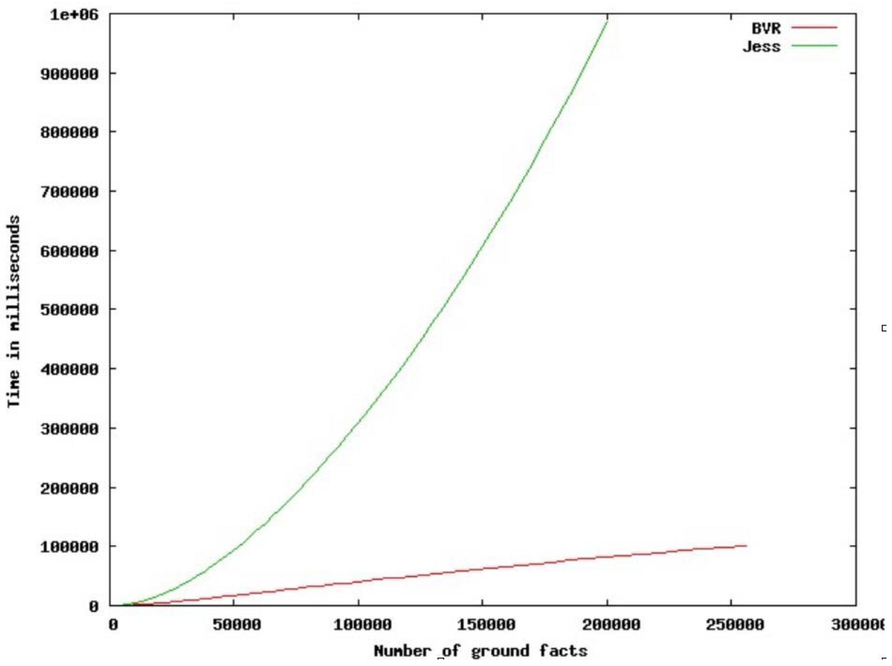


Fig. 5. BaseVISor vs. Jess performance

8 Conclusion

In our recent efforts to develop a situation awareness application for a supply logistics scenario we explored the utility of using SWRL and OWL to represent pertinent domain knowledge and apply it to simulated data using forward chaining inference engines. We encountered several challenges in applying SWRL to this problem including foremost the language's limitation to binary predicates and the lack of tools for editing and checking SWRL rules. We partially resolved the latter problem by developing RuleVISor, a graphical SWRL editor that permits the construction of rules using elements from OWL ontologies. Even with RuleVISor the restriction to binary predicates forced us to operate at a very low implementation level compared with the n-ary predicates that were more natural for representing the important domain concepts. Our approach to this problem was to develop abstract higher-arity rules by hand and then systematically convert the n-ary predicates into classes representing the predicates and collections of properties to associate values of the higher-arity terms with the predicate's class. While this was a manual process it is such that a simple translation script could perform the process automatically. The final steps of our rule development effort involved converting the abstract rules represented with binary-only predicates into SWRL syntax followed by the application of an XSLT script to translate the SWRL into either Jess or BaseVISor rules for their execution within the appropriate inference engine. We used Jess as our engine up until the completion of our own Rete-based inference engine, BaseVISor, which we have optimized for the processing of triples and incorporated support for uncertainty reasoning. Initial comparisons between the two engines demonstrated BaseVISor's near linear performance in the number of ground facts compared with Jess' polynomial performance.

References

1. M. Endsley and D. Garland, *Situation Awareness, Analysis and Measurement,* Lawrence Erlbaum Associates, Publishers, Mahway, New Jersey, 2000.
2. Rule Markup Language Initiative, http://www.ruleml.org/
3. Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. http://www.daml.org/rules/proposal/
4. C. Matheus, Using Ontology-based Rules for Situation Awareness and Information Fusion. Position Paper presented at the W3C Workshop on Rule Languages for Interoperability, April 2005. http://www.w3.org/2004/12/rules-ws/program2
5. W3C Workshop on Rule Languages for Interoperability. http://www.w3.org/2004/12/rules-ws/
6. C. Matheus, M. Kokar, K. Baclawski, J. Letkowski, C. Call, M. Hinman, J. Salerno and D. Boulware, SAWA: An Assistant for Higher-Level Fusion and Situation Awareness. In Proceedings of SPIE Conference on Multisensor, Multisource Information Fusion, Orlando, FL., March 2005.
7. C. Matheus, M. Kokar, K. Baclawski, J. Letkowski, C. Call, M. Hinman, J. Salerno and D. Boulware, Lessons Learned From Developing SAWA: A Situation Awareness Assistant, FUSION'05, Philadelphia, PA, July, 2005.
8. N. Noy and A. Rector, Defining N-ary Relations on the Semantic Web: Use With Individuals, W3C Working Draft 21, July 2004.
9. SWRL Editor for Protégé with the OWL plugin. http://protege.stanford.edu/plugins/owl/swrl/
10. ConsVISor Consistency Checking Service, http://www.vistology.com/consvisor/
11. Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, An Engine for SWRL rules in RDF graphs. http://www.inf.fu-berlin.de/inst/ag-nbi/research/swrlengine/

Appendix A: SWRL Code for "Marginal Part at Facility" Rule

```
<ruleml:imp>
    <ruleml:_rlab
        ruleml:href="#Marginal part at facility"/>
    <ruleml:_body>
      <swrlx:classAtom>
        <owlx:Class owlx:name="&evt;Event"/>
        <ruleml:var>?Event</ruleml:var>
      </swrlx:classAtom>
      <swrlx:datavaluedPropertyAtom
          swrlx:property="&evt;time">
        <ruleml:var>?Event</ruleml:var>
        <ruleml:var>?Time</ruleml:var>
      </swrlx:datavaluedPropertyAtom>
```

```
<swrlx:classAtom>
  <owlx:Class owlx:name="#SupplyMinusNeed" />
  <ruleml:var>?SMNStatement</ruleml:var>
</swrlx:classAtom>
<swrlx:individualPropertyAtom
    swrlx:property="#smnPart">
    <ruleml:var>?SMNStatement</ruleml:var>
    <ruleml:var>?Part</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="#smnFacility">
    <ruleml:var>?SMNStatement</ruleml:var>
    <ruleml:var>?Facility</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datavaluedPropertyAtom
      swrlx:property="#smnTime">
    <ruleml:var>?SMNStatement</ruleml:var>
    <ruleml:var>?Time</ruleml:var>
</swrlx:datavaluedPropertyAtom>
<swrlx:datavaluedPropertyAtom
     swrlx:property="#smnNumber">
    <ruleml:var>?SMNStatement</ruleml:var>
    <ruleml:var>?LocalSurplusOrDeficit</ruleml:var>
</swrlx:datavaluedPropertyAtom>
<swrlx:classAtom>
  <owlx:Class owlx:name="#PartsLocallyRepairable"/>
  <ruleml:var>?PLRStatement</ruleml:var>
</swrlx:classAtom>
<swrlx:individualPropertyAtom
      swrlx:property="#localPart">
    <ruleml:var>?PLRStatement</ruleml:var>
    <ruleml:var>?Part</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datavaluedPropertyAtom
      swrlx:property="#localNumber">
    <ruleml:var>?PLRStatement</ruleml:var>
    <ruleml:var>?NumberRepairable</ruleml:var>
</swrlx:datavaluedPropertyAtom>
    <ruleml:var>?PLRStatement</ruleml:var>
    <ruleml:var>?Facility</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
     swrlx:property="&rpa;remoteSupply">
    <ruleml:var>?Facility</ruleml:var>
    <ruleml:var>?RemoteSupplyList</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:classAtom>
   <owlx:Class
     owlx:name="#PartsAvailableAtRemoteFacility" />
  <ruleml:var>?PRAStatement</ruleml:var>
</swrlx:classAtom>
```

```
<swrlx:individualPropertyAtom
      swrlx:property="#remotePart">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?Part</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="#facilityList">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?RemoteSupplyList</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datavaluedPropertyAtom
      swrlx:property="#remoteNumber">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?NumberAvailableRemotely
    </ruleml:var>
</swrlx:datavaluedPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="#remoteTime">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?Time</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="&rpa;minimumPartsInSupply">
    <ruleml:var>?Facility</ruleml:var>
    <ruleml:var>?MinimumThresholdStatement
    </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="&rpa;item">
    <ruleml:var>?MinimumThresholdStatement
    </ruleml:var>
    <ruleml:var>?Part</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datavaluedPropertyAtom
      swrlx:property="&rpa;number">
    <ruleml:var>?MinimumThresholdStatement
    </ruleml:var>
    <ruleml:var>?MinimumThreshold</ruleml:var>
</swrlx:datavaluedPropertyAtom>
<swrlx:builtinAtom swrlx:builtin="&swrlb;add">
       <ruleml:var>?TotalAvailable</ruleml:var>
       <ruleml:var>?LocalSurplusOrDeficit
       </ruleml:var>
       <ruleml:var>?NumberRepairable</ruleml:var>
       <ruleml:var>?NumberAvailableRemotely
       </ruleml:var>
</swrlx:builtinAtom>
<swrlx:builtinAtom
          swrlx:builtin="&swrlb;greaterThan">
        <ruleml:var>?MinimumThreshold</ruleml:var>
        <ruleml:var>?TotalAvailable</ruleml:var>
      </swrlx:builtinAtom>
    </ruleml:_body>
    <ruleml:_head>
      <swrlx:classAtom>
      <owlx:Class owlx:name="#MarginalPartAtFacility"/>
          <ruleml:var>?MPFStatement</ruleml:var>
      </swrlx:classAtom>
      <swrlx:individualPropertyAtom
            swrlx:property="#marginalPart">
          <ruleml:var>?MPFStatement</ruleml:var>
          <ruleml:var>?Part</ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom
            swrlx:property="#marginalFacility">
          <ruleml:var>?MPFStatement</ruleml:var>
          <ruleml:var>?Facility</ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:datavaluedPropertyAtom
            swrlx:property="#marginalTime">
          <ruleml:var>?MPFStatement</ruleml:var>
          <ruleml:var>?Time</ruleml:var>
      </swrlx:datavaluedPropertyAtom>
      <swrlx:datavaluedPropertyAtom
            swrlx:property="#marginalNumber">
          <ruleml:var>?MPFStatement</ruleml:var>
          <ruleml:var>?TotalAvailable</ruleml:var>
      </swrlx:datavaluedPropertyAtom>
    </ruleml:_head>
  </ruleml:imp>
```

A Semantic Web Based Architecture for e-Contracts in Defeasible Logic

Guido Governatori and Duy Pham Hoang

School of Information Technology and Electrical Engineering
The University of Queensland, Brisbane, Queensland, Australia, QLD 4072
`{guido,pham}itee.uq.edu.au`

Abstract. We introduce the DR-CONTRACT architecture to represent and reason on e-Contracts. The architecture extends the DR-device architecture by a deontic defeasible logic of violation. We motivate the choice for the logic and we show how to extend *RuleML* to capture the notions relevant to describe e-contracts for a monitoring perspective in Defeasible Logic.

1 Introduction

Business contracts are mutual agreements between two or more parties engaging in various types of economic exchanges and transactions. They are used to specify the obligations, permissions and prohibitions that the signatories should be hold responsible to and to state the actions or penalties that may be taken in the case when any of the stated agreements are not being met.

We will focus on the monitoring of contract execution and performance: contract monitoring is a process whereby activities of the parties listed in the contract are governed by the clauses of the contract, so that the correspondence of the activities listed in the contract can be monitored and violations acted upon. In order to monitor the execution and performance of a contract we need a precise representation of the 'content' of the contract to perform the required actions at the required time.

The clauses of a contract are usually expressed in a codified or specialised natural language, e.g., legal English. At times this natural language is, by its own nature, imprecise and ambiguous. However, if we want to monitor the execution and performance of a contract, ambiguities must be avoided or at least the conflicts arising from them resolved. A further issue is that often the clauses in a contract show some mutual interdependencies and it might not be evident how to disentangle such relationships. To implement an automated monitoring system all the above issues must be addressed.

To deal with some of these issues we propose a formal representation of contracts. A language for specifying contracts needs to be formal, in the sense that its syntax and its semantics should be precisely defined. This ensures that the protocols and strategies can be interpreted unambiguously (both by machines and human beings) and that they are both predictable and explainable. In addition, a formal foundation is a prerequisite for verification or validation purposes.

One of the main benefits of this approach is that we can use formal methods to reason with and about the clauses of a contract. In particular we can

– analyse the expected behaviour of the signatories in a precise way, and
– identify and make evident the mutual relationships among various clauses in a contract.

Secondly, a language for contracts should be conceptual. This, following the well-known *Conceptualization Principle* of [13], effectively means that the language should allow their users to focus only and exclusively on aspects related to the content of a contract, without having to deal with any aspects related to their implementation.

Every contract contains provisions about the obligations, permissions, entitlements and others mutual normative positions the signatories of the contract subscribe to. Therefore a formal language intended to represent contracts should provide notions closely related to the above concepts.

A contract can be viewed as a legal document consisting of a finite set of articles, where each article consists of finite set of clauses. In general it is possible to distinguish two types of clauses:

1. definitional clauses, which define relevant concepts occurring in the contract;
2. normative clauses, which regulate the actions of the parties for contract performance, and include deontic modalities such as obligations, permissions and prohibitions.

For example the following fragment of a contract of service taken from [8] are definitional clauses

> 3.1 A "Premium Customer" is a customer who has spent more that $10000 in goods.
> 3.2 Service marked as "special order" are subject to a 5% surcharge. Premium customers are exempt from special order surcharge.

while

> 5.2 The (Supplier) shall on receipt of a purchase order for (Services) make them available within one day.

and

> 5.3 If for any reason the conditions stated in 4.1 or 4.2 are not met the (Purchaser) is entitled to charge the (Supplier) the rate of $100 for each hour the (Service) is not delivered.

are normative clause. The above contract clauses make it clear that there is the need to differentiate over Clauses 3.1 and 3.2 on one side, and Clauses 5.2 and 5.3 on the other.The first two clauses are factual/definitional clauses describing states of affairs, defining notions in the conceptual space of the contract. For example clause 3.1 defines the meaning of "Premium Customer" for the contract, and Clause 3.2 gives a recipe to compute the price of services. On the other hand Clauses 5.2 and 5.3 state the (expected) legal behaviour of the parties involved in the transaction.In addition there is a difference between Clause 5.2 and Clause 5.3. Clause 5.2 determines an obligation for one of the party; on the other hand Clause 5.3 establishes a permission. Hence, according to our previous discussion about the functionalities of the representation formalism, a logic meant to capture the semantics of contracts has to account for such issues. Since the seminal work by Lee [16] Deontic Logic has been regarded as one on the most prominent paradigms to formalise contracts. In [8] we further motivate on the need of deontic logic to capture the semantics of contracts and the reasons to choose it over other formalisms.

Clause 3.2 points out another feature. Contract languages should account for exceptions. In addition, given the normative nature of contracts, exceptions can be open ended, that is, it is not possible to give a complete list of all possible exception to a condition. This means that we have to work in an environment where conclusions are defeasible, i.e., it is possible to retract conclusions when new pieces of information become available.

From a logical perspective every clause of a contract can be understood as a rule where we have the conditions of applicability of the clause and the expected behaviour. Thus we have that we can represent a contract by a set of rules, and, as we have already argued, these rules are non-monotonic. Thus we need a formalism that is able to reason within this kind of scenario. Our choice here is Defeasible Logic (we will motivate this choice in section 2).

Finally Clause 5.3 highlights an important aspect of contracts: contracts often contain provisions about obligations/permissions arising in response to violations. Standard Deontic Logic is not very well suited to deal with violations. Many formalisms have devised to obviate some problems of violations in deontic logic. In this paper we will take a particular approach to deal with violation that can be easily combined with the other component we have outlined here.

The paper is organised as follows: in Section 2 we present the logic on which the DR-CONTRACT architecture is based. Then in Section 3 we explain the extension of *RuleML* corresponding to the logic of the previous section, and we establish a mapping between the two languages. Then, in Section 4 we discuss the system architecture of the DR-CONTRACT framework. Finally we relate our work to similar approaches and we give some insights about future developments in Section 5.

2 Defeasible Deontic Logic of Violation

For a proper representation of contracts and to be able to reason with and about them we have to combine and integrate logics for various essential component of contracts. In particular we will use the Defeasible Deontic Logic of Violation (DDLV) proposed in [8]. This logic combines deontic notions with defeasibility and violations. More precisely DDLV is obtained from the combination of three logical components: Defeasible Logic, deontic concepts, and a fragment of a logic to deal with normative violations. Before presenting the logic we will discuss the reasons why such notions are necessary for the representation of contracts.

In [14] Courteous Logic Programming (CLP) has been advanced as the inferential engine for business contracts represented in *RuleML*. Here, instead, we propose Defeasible Logic (DL) as the inferential mechanism for *RuleML*. In fact, CLP is just a notational variant of one of the many logics in the family proposed by [1, 3] (see [4] for the relationships between DL and CLP). Accordingly, it may be possible to integrate the extensions we develop in the rest of the paper within a CLP framework. Over the years DL proved to be a flexible non-monotonic formalism able to capture different and sometimes incompatible facets of non-monotonic reasoning [1], and efficient and powerful implementations have been proposed [3, 5, 6, 18]. The primary use of DL in the present context is aimed at the resolution of conflicts that might arise from the clauses of a contract; in addition DL encompasses other existing formalisms proposed in the AI & Law field (see, [9]), and recent work shows that DL is suitable for extensions with modal and deontic operators [10, 12].

DL analyses the conditions laid down by each rule in the contract, identifies the possible conflicts that may be triggered and uses priorities, defined over the rules, to eventually solve a conflict. A defeasible theory contains here four different kinds of knowledge: facts, strict rules, defeasible rules, and a superiority relation.

Facts are indisputable statements, for example, "the price of the spam filter is $50". Facts are represented by predicates

$$\mathit{Price}(\mathit{SpamFilter}, 50).$$

Strict rules are rules in the classical sense: whenever the premises are indisputable then so is the conclusion. An example of a strict rule is "A 'Premium Customer' is a customer who has spent $10000 on goods", formally:

$$\mathit{TotalExpense}(X, 10000) \rightarrow \mathit{PremiumCustomer}(X).$$

Defeasible rules are rules that can be defeated by contrary evidence. An example of such a rule is "Premium Customer are entitled to a 5% discount":

$$\mathit{PremiumCustomer}(X) \Rightarrow \mathit{Discount}(X).$$

The idea is that if we know that someone is a Premium Customer, then we may conclude that she is entitled to a discount *unless there is other evidence suggesting that she may not be* (for example if she buys a good in promotion).

The *superiority relation* among rules is used to define priorities among them, that is, where one rule may override the conclusion of another rule. For example, given the defeasible rules

$$r : \mathit{PremiumCustomer}(X) \Rightarrow \mathit{Discount}(X)$$
$$r' : \mathit{SpecialOrder}(X) \Rightarrow \neg\mathit{Discount}(X)$$

which contradict one another, no conclusive decision can be made about whether a Premium Customer who has placed a special order is entitled to the 5% discount. But if we introduce a superiority relation $>$ with $r' > r$, then we can indeed conclude that special orders are not subject to discount.

We now give a short informal presentation of how conclusions are drawn in Defeasible Logic. A conclusion p can be derived if there is a rule whose conclusion is p, whose prerequisites (antecedent) have either already been proved or given in the case at hand (i.e. facts), and any stronger rule whose conclusion is $\neg p$ has prerequisites that fail to be derived. In other words, a conclusion p is derivable when:

- p is a fact; or
- there is an applicable strict or defeasible rule for p, and either
  - all the rules for $\neg p$ are discarded (i.e., are proved to be not applicable) or
  - every applicable rule for $\neg p$ is weaker than an applicable strict[1] or defeasible rule for p.

For a full presentation of Defeasible Logic see [2, 8],

The next step is to integrate deontic logic in defeasible logic. To this end we follow the idea presented in [10]. In the context of contract we introduced the directed deontic operators $O_{s,b}$ and $P_{s,b}$. Thus, for example the expression $O_{s,b}A$ means that A is obligatory such that s is the subject of such an obligation and b is its beneficiary; similarly for $P_{s,b}$, where $P_{s,b}A$ means that s is permitted to do A in the interest of b. In this way it is possible to express rules like the following

$$PurchaseOrder \Rightarrow O_{Supplier,Purchaser}Deliver\_Within1Day$$

that encodes Clause 5.2 of the contract presented above.

Finally, let us sketch how to incorporate a logic for dealing with normative violations within the framework we have described so far. A violation occurs when an obligation is disattended, thus $\neg A$ is a violation of the obligation OA. However, a violation of an obligation does not imply the cancellation of such an obligation. This makes often difficult to characterise the idea of violation in many formalisms for defeasible reasoning [22]. We will take and adapt some intuitions we developed fully in [11]. To reason on violations we have to represent contrary-to-duties (CTDs) or reparational obligations. As is well-known, these last are in force only when normative violations occur and are meant to "repair" violations of primary obligations. In the spirit of [11] we introduce the non-classical connective $\otimes$, whose interpretation is such that $OA \otimes OB$ is read as "OB is the reparation of the violation of OA". The connective $\otimes$ permits to combine primary and CTD obligations into unique regulations. The operator $\otimes$ is such that $\neg\neg A \equiv A$ for any formula A and enjoys the properties of associativity, duplication and contraction. For the purposes of this paper, it is sufficient to

[1] Notice that a strict rule can be defeated only when its antecedent is defeasibly provable

define the following rule for introducing $\otimes$[2]:

$$\frac{\Gamma \Rightarrow O_{s,b}A \otimes (\bigotimes_{i=1}^{n} O_{s,b}B_i) \otimes O_{s,b}C \qquad \Delta, \neg B_1, \ldots, \neg B_n \Rightarrow \mathbf{X}_{s,b}D}{\Gamma, \Delta \Rightarrow O_{s,b}A \otimes (\bigotimes_{i=1}^{n} O_{s,b}B_i) \otimes \mathbf{X}_{s,b}D} \tag{1}$$

where $\mathbf{X}$ denotes an obligation or a permission. In this last case, we will impose that D is an atom. Since the minor premise states that $\mathbf{X}_{s,b}D$ is a reparation for $O_{s,b}B_n$, i.e., the last literal in the sequence $\bigotimes_{i=1}^{n} O_{s,b}B_i$, we can attach $\mathbf{X}_{s,b}D$ to such sequence. In other words, this rule permits to combine into a unique regulation the two premises.

Suppose the theory includes

$$r : \mathit{Invoice} \Rightarrow O_{s,b}\mathit{Pay\_Within7Days}$$
$$r' : \neg \mathit{Pay\_Within7Days} \Rightarrow O_{s,b}\mathit{Pay\_WithInterest}.$$

From these rules we obtain

$$r'' : \mathit{Invoice} \Rightarrow O_{s,b}\mathit{Pay\_Within7Days} \otimes O_{s,b}\mathit{Pay\_WithInterest}.$$

As soon as we applied ($\otimes$I) as much as possible, we have to drop all redundant rules. This can be done by means of the notion of subsumption:

Definition 1. *Let* $r_1 = \Gamma \Rightarrow A \otimes B \otimes C$ *and* $r_2 = \Delta \Rightarrow D$ *be two rules, where* $A = \bigotimes_{i=1}^{m} A_i$*,* $B = \bigotimes_{i=1}^{n} B_i$ *and* $C = \bigotimes_{i=1}^{p} C_i$*. Then* r_1 subsumes r_2 *iff*

1. $\Gamma = \Delta$ *and* $D = A$*; or*
2. $\Gamma \cup \{\neg A_1, \ldots, \neg A_m\} = \Delta$ *and* $D = B$*; or*
3. $\Gamma \cup \{\neg B_1, \ldots, \neg B_n\} = \Delta$ *and* $D = A \otimes \bigotimes_{i=0}^{k \leq p} C_i$*.*

The idea behind this definition is that the normative content of r_2 is fully included in r_1. Thus r_2 does not add anything new to the system and it can be safely discarded. In the example above, we can drop rule r, whose normative content is included in r''.

Formally a *conclusion* in DDLV is a tagged literal and can have one of the following forms:

- $+\Delta q$ to mean that the literal q is definitely provable (i.e., using only facts and strict rules),
- $-\Delta q$ when q is not definitely provable,
- $+\partial q$, whenever q is defeasibly provable, and
- $-\partial q$ to mean that we have proved that q is not defeasibly provable.

Provability is based on the concept of a *derivation*. A derivation is a finite sequence $P = (P(1), \ldots, P(n))$ of tagged literals satisfying four conditions (which correspond to inference rules for each of the four kinds of conclusion). Here we will give only the conditions for $+\Delta$ and $+\partial q$. $P(1..i)$ denotes the initial part of the sequence P of length i:

[2] The $\otimes$ is allowed only in the head of defeasible rules. See [8] for a full motivation of this design choice

The inference rule for $\pm\Delta$ are just those for forward chaining of strict rules, thus they corresponds to detachment or Modus Ponens for $+\Delta$ and a full search that modus ponens cannot be applied for $-\Delta$.

To accommodate the new connective ($\otimes$) in DDLV we have to revise the inference mechanism of Defeasible Logic. The first thing we have to note is that now a defeasible rule can be used to derive different conclusions. For example given the rule

$$r : A \Rightarrow O_{s,b}B \otimes O_{s,b}C \tag{2}$$

we can use it to derive $O_{s,b}B$ if we have A, but if we know A and $\neg B$ then the same rule supports the conclusion $O_{s,b}C$.

With $R[c_i = q]$ we denote the set of rules where the head of the rule is $\otimes_{j=1}^{n} c_j$ where for some i, $1 \leq i \leq n$, $c_i = q$. For example, given the rule r in (2), $r \in R[c_1 = O_{s,b}B]$ and $r \in R[c_2 = O_{s,b}C]$. Given an obligation $O_{s,b}A$, we use $\overline{O_{s,b}A}$ to denote the complement of A, i.e., $\sim A$.

We are now ready to give the proof condition for $+\partial$.

```
+∂: If $P(i+1) = +\partial q$ then either
    (1) $+\Delta q \in P(1..i)$ or
    (2) (2.1) $\exists r \in R[c_i = q]$
              (2.1.1) $\forall a \in A(r) : +\partial a \in P(1..i)$ and
              (2.1.2) $\forall i' < i, \exists a = \overline{c_{i'}} : +\partial a \in P(1..i)$
        (2.2) $-\Delta{\sim}q \in P(1..i)$ and
        (2.3) $\forall s \in R[c_j = {\sim}q]$ either
              (2.3.1) $\exists a \in A(s) : -\partial a \in P(1..i)$ or
              (2.3.2) $\exists j' < j, \forall c_{j'} - \partial\overline{c_{j'}} \in P(1..i)$ or
              (2.3.3) $\exists t \in R_{sd}[q]$ such that
                      $\forall a \in A(t) : +\partial a \in P(1..i)$
                      $\forall k' < k, +\partial\overline{c_{k'}} \in P(1..i)$ and $t > s$.
```

The above condition is very similar to the same condition for basic defeasible logic [2]. The main differences account for the $\otimes$ connective. What we have to ensure is that reparations of violations are in force when we try to prove them. For example if we want to prove $O_{s,b}C$ given the rule $r : A \Rightarrow O_{s,b}B \otimes O_{s,b}C$, we must show that we are able to prove A, and that the primary obligation B has been violated. In other words we have already proved $\neg B$ or any other formula incompatible with B (Clause 2.1.2). A similar explanation holds true for Clause 2.3.2 where we want to show that a rule does not support an attack on the intended conclusion.

3 Contracts in RuleML

In order to integrate the DR-CONTRACT engine with Semantic Web technology we decided to use RuleML [20] as an open and vendor neutral XML/RDF syntax for contracts. We tried to re-use as many features of standard RuleML syntax as possible. However, since some notions essential for the representation of contracts are not present in standard RuleML we have created our DR-CONTRACT DTD (Figure 1)[3].

```
<!ELEMENT Atom       (Not?,Rel,(Ind|Var)*)>
<!ELEMENT Not        (Rel,(Ind|Var)*)>
<!ELEMENT Rel        (#PCDATA)>
<!ELEMENT Var        (#PCDATA)>
<!ELEMENT Ind        (#PCDATA)>
<!ELEMENT Fact       (Atom)>
<!ELEMENT Imp        ((Head,Body)|(Body|Head))>
<!ATTLIST Imp        label ID #REQUIRED
                     strength (strict|defeasible) #REQUIRED>
<!ELEMENT Body       (And)>
<!ELEMENT And        (Atom|Obligation|Permission)*>
<!ELEMENT Head       (Atom|Obligation|Permission|Behaviour)>
<!ELEMENT Behaviour  ((Obligation)+,Permission?)>
<!ELEMENT Obligation (Not?,Rel,(Ind|Var)*)>
<!ATTLIST Obligation subject IDREF #REQUIRED beneficiary IDREF #REQUIRED>
<!ELEMENT Permission (Not?,Rel,(Ind|Var)*)>
<!ATTLIST Permission subject IDREF #REQUIRED beneficiary IDREF #REQUIRED>
```

Fig. 1. DR-CONTRACT Basic DTD

The main limitations of RuleML is that it does not support modalities and it is unable to deal with violations. The DR-CONTRACT RuleML DTD takes two different types of literals: unmodalised predicates and modalised literals. Thus to appropriately represent the deontic notions of obligation and permission we introduce two new elements `<Obligation>` and `<Permission>`, which are intended to replace `<Atom>` in the conclusion of normative rules. In addition deontic elements can be used in the body of derivation rules. Hence we have to extend the definition of `<And>` and `<Head>`. In this way it is possible to distinguish from brute fact and normative facts. As we have already argued this is essential if one wants to use RuleML to represent business contracts.

The elements `<Var>` and `<Ind>` are, respectively, placeholders for individual variables to be instantiated by ground values when the rules are applied and individual constants. Individual constants can be just simple names or URIs referring to the appropriate individuals. `<Rel>` is the element that contains the name of the predicate. `<Not>` is intended to represent classical negation. Thus its meaning is that the atom it negates is not the case (or the proposition represented by the atom is false and consequently the proposition the element represents is true). RuleML contains two types of negation, classical negation and negation

[3] Although the current version of RuleML (Version 0.89) is based on XML Schema, here due to space limitation and for ease of presentation, we will give the XML grammar using simplified DTD definitions

as failure [7, 23]. However, negation as failure can be simulated by other means in Defeasible Logic [4], so we do not include it in our syntax.

RuleML provides facilities for many types of rule. However, we believe that the distinction has a pragmatic flavour more than a conceptual one. In this paper we are interested in the logical and computational aspects of the rules, thus we decided to focus only on derivation rules `<Imp>`.

Derivation rules allow the derivation of information from existing rules [23]. They are able to capture concepts not stored explicitly in the existing information. For example, a customer is labelled as a "Premium" customer when he buys $10000 worth of goods. As such, the rule here states that the customer must have spent $10000 on goods, thus deriving the information here that the customer is a "Premium" customer. A derivation rule has an attribute `strength` whose value ranges over `strict` and `defeasible` and it denotes the type of rule to be associated to it when computed in defeasible logic.

A derivation rule has two immediate sub-elements, *Condition* (`<Body>`) and *Conclusion* (`<Head>`); the latter being either an atomic predicate formula or a sequence of obligations, and the former a conjunction of formulas [24], meaning that derivation rules consist of one more conditions and a conclusion.

The ability to deal with violations and the obligations arising in response to them is one of the key features in the representation of business contracts. To this end the conclusion of a derivation rule corresponding to a normative rule is a `<Behaviour>` element, defined as a sequence of `<Obligation>` and `<Permission>` elements with the constraints that the sequence contains at most one `<Permission>` element, and this element is the last of the sequence. This construction is meant to simulate the behaviour of $\otimes$.

As we have alluded to in the previous section RuleML provides a semantically neutral syntax for rules and different types of rules can be reduced to other types and rules in RuleML can be mapped to native rules in other formalism. For the relationships between RuleML and Defeasible Logic we will translate derivation rules (`<Imp>`s) into rules in Defeasible Logic specifications. In this perspective a derivation rule

```
<Imp label="r" strength="defeasible">
  <Body>...</Body>
  <Head>
    <Behaviour>
      <Obligation>A1</Obligation>
      ...
      <Deontic>An</Deontic>
    </Behaviour>
  </Head>
</Imp>
```

is transformed into a defeasible rule

$$r : \texttt{body} \Rightarrow OA_1 \otimes \cdots \otimes \mathbf{X}A_n$$

where $\mathbf{X}$ is the translation of the `<Deontic>` (meta) element.

We give now an example of a rule based on the following contract clause

> 6.1 The payment terms shall be in full upon receipt of invoice. Interest shall be charged at 5 % on accounts not paid within 7 days of the invoice date.

```
<Imp label="6.1" strength="defeasible">
   <Body><And>
           <Atom><Rel>Invoice</Rel>
                 <Var>InvoiceDate</Var>
                 <Var>Amount</Var></Atom>
         </And>
   </Body>
   <Head>
      <Behaviour>
         <Obligation subject="Purchaser"
                     beneficiary="Supplier">
               <Rel>PayInFullWithin7Days</Rel>
               <Var>InvoiceDate</Var>
               <Var>Amount</Var>
         </Obligation>
         <Obligation subject="Purchaser"
                     beneficiary="Supplier">
               <Rel>PayWithInterest</Rel>
               <Var>Amount * 1.05</Var>
         </Obligation>
      </Behaviour>
   </Head>
</Imp>
```

The new deontic tags in the DR-CONTRACT extended DTD in Figure 2 –`<Reparation>`, `<Penalty>` and `<Violation>`– do not increase the expressive

```
<!ELEMENT And       (Atom|Obligation|Permission|Violation)*>
<!ELEMENT Violation EMPTY>
<!ATTLIST Violation rule IDREF #REQUIRED>
<!ELEMENT Behaviour ((Obligation+,Reparation)|(Obligation*,Permission?))>
<!ELEMENT Reparation EMPTY>
<!ATTLIST Reparation penalty IDREF #REQUIRED>
<!ELEMENT Penalty   ((Obligation+,Reparation)|(Obligation*,Permission?))>
<!ATTLIST Penalty   label ID #REQUIRED>
```

Fig. 2. DR-CONTRACT Extended DTD

power of the language but are included as convenient shortcuts. It is possible to express a violation explicitly by saying that a particular rule is triggered in response to a violation (i.e., when an obligation is not fulfilled) . However, it can be convenient to have facilities to represent violations directly –just look at the formulation of Clause 5.3. In general a violation can be one of the conditions that trigger the application of a rule. Accordingly a `<Violation>` element can be included in the body of a rule. A violation cannot subsist without a rule that is violated by it. Hence the attribute `rule` is a reference to the rule that has been violated. Many contract languages [15, 19] contain similar constructions. The activation of such constructions/processes requires the generation of a violation event/literal. On the contrary our approach does not require it. All we have to do is to check for a sequence of literals joined with the $\otimes$ operator where the initial part of the sequence is not satisfied.

A `<Violation>` occurs in the body of rule and the `rule` attribute refers to the violated rule. Every `<Violation>` element can be replaced by the conjunction of the elements in the `<Body>` of the violated rule, i.e., the rule the `rule` attribute refers to, plus the negation of the un-modalised elements of the elements in the `<Head>` of the violated rule.

```
<Imp label="v">
  <body>B1</body>
  <head>
    <Behaviour>
      <Obligation>A1</Obligation>
      ...
      <Obligation>An</Obligation>
    </Behaviour>
  </head>
</Imp>
```

```
<Imp label="r">
  <body>
    <And>
      B2
      <Violation rule="v"/>
    </And>
  </body>
  <head>
    <Behaviour>
      <Obligation>C1</Obligation>
      ...
      <Deontic>Cm</Deontic>
    </Behaviour>
  </head>
</Imp>
```

From the above *RuleML* code we generate two rules in DDLV, namely

$$v : B_1 \Rightarrow OA_1 \otimes \cdots \otimes OA_n,$$
$$r : B_1, B_2, \neg A_1, \ldots, \neg A_n \Rightarrow OC_1 \otimes \cdots \otimes \mathbf{X}C_m.$$

Eventually the two rules can be combined via the schema (1) in

$$vr : B_1, B_2 \Rightarrow OA_1 \otimes \cdots \otimes OA_n \otimes OC_1 \otimes \cdots \otimes \mathbf{X}C_m.$$

In some cases one might have recurrent general penalties and it may be convenient to state them once and refer back to them when they are called. To deal with this case we introduce two additional elements `<Reparation>` and `<Penalty>`. A `<Reparation>` element is just an empty element with a reference to a `<Penalty>` element that can occur only after an obligation in a `<Behaviour>` element, where a `<Penalty>` element is a premiseless rule with a normative head that is triggered only when its corresponding violations are raised.

For example given the following fragment of a contract

```
<Imp label='r'>                         <Penalty label="p">
  <body>...</body>                         <Obligation>B1</Obligation>
  <head>                                   ...
    <Behaviour>                            <Deontic>Bm</Deontic>
      <Obligation>A1</Obligation>       </Penalty>
       ...
      <Obligation>An</obligation>
      <Reparation penalty="p"/>
    </Behaviour>
  </head>
</Imp>
```

the rule corresponding to it is

$$r : \texttt{body} \Rightarrow OA_1 \otimes \cdots \otimes OA_n \otimes OB_1 \otimes \cdots \otimes \mathbf{X}B_m.$$

4 DR-CONTRACT System Architecture

The system architecture of DR-CONTRACT is inspired by the system architecture of the family of DR-DEVICE applications [5, 6, 21] and consists of four main modules (see Figure 3):

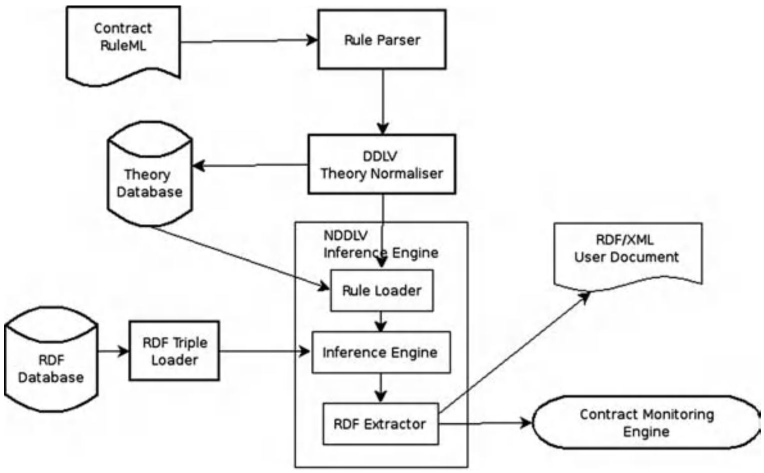


Fig. 3. DR-CONTRACT System Architecture

1. A Rule Parser to transform a DR-CONTRACT compliant document (a contract) into a theory to be passed to the next module. The parser is based on the XML processor and it is rather similar in nature to the Logic Loader module of the DR-Device family applications [5, 6, 21].

2. A DDLV normaliser. The normaliser takes as input a DDLV theory (obtained from the previous step) and iteratively merges rules in the theory according to the inference rule 1 and then removes rules subsumed by a more general rule according to Definition 1. It repeats the cycle till it reaches the fixed-point of such a construction (which is guaranteed to exist and to be unique [11]). Once a theory has been normalised the normal form is saved to a repository (for faster loading in successive calls), and the normalised theory NDDLV is passed to the DDLV engine. In addition the normaliser applies a transformation that removes superiority relation by compiling it into new rules (the technique used here is similar to that of [2]).
3. The RDF loader downloads/queries the input documents, including their schemata, and it translates the RDF descriptions into fact objects according to the RDF-NDDLV translation schema based on the DR-CONTRACT DTD.
4. The NDDLV inference engine consists of two components:
   – The *Rule Loader* compiles the rules in a NDDLV theory in objects. We distinguish two types of objects: Rules and Literals or atoms. Each rule object has associated to it a list of (pointers to) modal literals (corresponding to head of the rule) and a set of (pointers to) modal literals implemented as an hash table. Each atom object has associated to it four hash tables: the first with pointers to the rules where the atom occurs positively in the head, the second with pointers to the rules where the atom occurs negatively in the head, the third with pointers to the rules where the atom occurs positively in the body and the last with pointers where the atom occurs negatively in the body.
   – The *Inference Engine* is based on an extension of the *Delores* algorithm/implementation proposed in [18] as a computational model of Basic Defeasible Logic. In turn:
     • It asserts each fact (as an atom) as a conclusion and removes the atom from the rules where the atom occurs positively in the body, and it "deactivates" the rules where the atom occurs negatively in the body. The complement of the literal is removed from the head of rules where it does not occur as first element. The atom is then removed from the list of atoms.
     • It scans the list of rules for rules where the body is empty. It takes the first element of the head and searches for rule where the negation of the atom is the first element. If there are no such rules then, the atom is appended to the list of facts, and removed from the rules
     • It repeats the first step.
     • The algorithm terminates when one of the two steps fails. On termination the algorithm outputs the set of conclusions[4].

[4] Notice that the algorithm runs in linear time. Each atom/literal in a theory is processed exactly once and every time we have to scan the set of rules, thus the complexity of the above algorithm is $O(|L| * |R|)$, where L is the set of distinct modal literals and R is the set of rules

5. Finally the conclusions are exported either to the user or to a monitoring contract facility such as BCL [17, 19] as an RDF/XML document through an RDF extractor.

5 Conclusion and Related Works

In this paper we have presented a system architecture for a Semantic Web based system for reasoning about contracts. The architecture is inspired by the system architecture of the DR-DEVICE family of applications. The main differences between our approach and the DR-DEVICE is in the use of an extended variant of Defeasible Logic. The extensions are in the use of modal operator and a non classical operator for violations. The same difference applies for the SweetDeal approach by Grosof [14, 15]. We have also argued that the extension with modal (deontic) operators is not only conceptually sound but also necessary to capture the semantics of contracts. In the same way the implementation of the inference engine is an extension of the algorithm used by Delores [18] to cope with deontic operators and the $\otimes$ operator.

The handling of temporal aspects is a very delicate matter in contract monitoring. The current architecture does not cover temporal reasoning. However, [12] proposes an extension of Defeasible Logic that can represent and reason with temporalised normative positions. In particular the framework offers facilities to initiate and terminate obligations, permissions, prohibitions and other complex normative positions. We have planned to study how to efficiently incorporate such features in our Deontic Defeasible Logic of Violations.

Currently we have implemented prototypes of the inference engine in Python and Java, and experimental results show that the Python implementation is able to deal with some of the benchmark theories of [18] with theories in some case with over 50000 rules.

Acknowledgements

This work was partially supported by the UQ Early Career Researcher Grant no. 2004001458 on "A System for Automated Agent Negotiation with Defeasible Logic-Based Strategies".

References

1. G. Antoniou, D. Billington, G. Governatori, and M. Maher. A flexible framework for defeasible logics. In *(AAAI-2000)*, 401–405. AAAI/MIT Press, 2000.
2. G. Antoniou, D. Billington, G. Governatori, and M. Maher. Representation results for defeasible logic. *ACM Trans. on Computational Logic*, 2(2):255–287, 2001.
3. G. Antoniou, D. Billington, G. Governatori, M. Maher, and A. Rock. A family of defeasible reasoning logics and its implementation. In W. Horn, editor, *ECAI 2000*, 459–463. IOS Press. 2000.

4. G. Antoniou, M. Maher, and D. Billington. Defeasible logic versus logic programming without negation as failure. *J. of Logic Programming*, 41(1):45–57, 2000.
5. N. Bassiliades, G. Antoniou, and I. Vlahavas. A defeasible logic reasoner for the semantic web. In G. Antoniou and H. Boley, editors, *RuleML 2004*, LNCS 3323, 49–64. Springer-Verlag, 2004.
6. N. Bassiliades, G. Antoniou, and I. Vlahavas. DR-DEVICE: A defeasible logic system for the Semantic Web. In H.J. Ohlbach and S. Schaffert, editors, *2nd PPSWR*, LNCS 3208, 134–148. Springer-Verlag, 2004.
7. H. Boley, S. Tabet, and G. Wagner. Design rationale for ruleml: A markup language for semantic web rules. In I.F. Cruz, S. Decker, J. Euzenat, and D.L. McGuinness, editors, *SWWS'01*, 381–401, 2001.
8. G. Governatori. Representing business contracts in RuleML. *Int. J. of Cooperative Information Systems*, 14(2-3):181–216, 2005.
9. G. Governatori, M. Maher, D. Billington, and G. Antoniou. Argumentation semantics for defeasible logics. *J. of Logic and Computation*, 14(5):675–702, 2004.
10. G. Governatori and A. Rotolo. Defeasible logic: Agency, intention and obligation. In A. Lomuscio and D. Nute, editors, *Deontic Logic in Computer Science*, LNAI 3065, 114–128. Springer-Verlag, 2004.
11. G. Governatori and A. Rotolo. Logic of Violations: A Gentzen system for reasoning with contrary-to-duty obligations. *Australasian Journal of Logic*, 2005.
12. G. Governatori, A. Rotolo, and G. Sartor. Temporalised normative positions in defeasible logic. In A. Gardner, editor, *10th ICAIL*, 25–34. ACM Press, 2005.
13. J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publ. nr. ISO/TC97/SC5/WG3-N695, ANSI, 11 West 42nd Street, New York, NY 10036, 1982.
14. B.N. Grosof. Representing e-commerce rules via situated courteous logic programs in RuleML. *Electronic Commerce Research and Applications*, 3(1):2–20, 2004.
15. B.N. Grosof and T. C. Poon. SweetDeal: representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *12th WWW*, 340–349. ACM Press, 2003.
16. R.M. Lee. A logic model for electronic contracting. *Decision Support Systems*, 4:27–44, 1988.
17. P. Linington, Z. Milosevic, J. Cole, S. Gibson, S. Kulkarni, and S. Neal. A unified behavioural model and a contract for extended enterprise. *Data & Knowledge Engineering*, 51:5–29, 2004.
18. M. Maher, A. Rock, G. Antoniou, D. Billington, and T. Miller. Efficient defeasible reasoning systems. *Int. J. of AI Tools*, 10(4):483–501, 2001.
19. Z. Milosevic, S. Gibson, P.F. Linington, J. Cole, and S. Kulkarni. On design and implementation of a contract monitoring facility. In *1st IWEC*, 62–70. IEEE Press, 2004.
20. RuleML. The Rule Markup Initiative, 1 September 2005.
21. T. Skylogiannis, G. Antoniou, N. Bassiliades, and G. Governatori. DR-NEGOTIATE – a system for automated agent negotiation with defeasible logic-based strategies. In *EEE'05*, 44–49. IEEE Press, 2005.
22. L. van der Torre and Y.-H. Tan. The many faces of defeasibility. In D. Nute, editor, *Defeasible Deontic Logic*, 79–121. Kluwer, 1997.
23. G. Wagner. How to design a general rule markup language. In *Proceedings of XML Technology for the Semantic Web (XSW 2002)*, *LNI* 14, 19–37. GI, 2002.
24. G. Wagner, S. Tabet, and H. Boley. MOF-RuleML: The abstract syntax of RuleML as a MOF model. In *OMG Meeting*, Boston, 2003.

Merging and Aligning Ontologies in dl-Programs

Kewen Wang[1], Grigoris Antoniou[2], Rodney Topor[1], and Abdul Sattar[1]

[1] Griffith University, Australia
{k.wang,r.topor,a.sattar}@griffith.edu.au
[2] University of Crete, Greece
ga@csd.uoc.gr

Abstract. The language of dl-programs is a latest effort in developing an expressive representation for Web-based ontologies. It allows to build answer set programming (ASP) on top of description logic and thus some attractive features of ASP can be employed in the design of the Semantic Web architecture. In this paper we first generalize dl-programs by allowing multiple knowledge bases and then accordingly, define the answer set semantics for the dl-programs. A novel technique called forgetting is developed in the setting of dl-programs and applied to ontology merging and aligning.

1 Introduction

A key part of the Semantic Web architecture is designing a set of languages so that web-based ontologies can be represented and reasoned easily and correctly. The Web Ontology Language (OWL) is the latest standard recommended by the World Wide Web Consortium (W3C). The design and standardization of OWL is largely influenced by description logic [2]. As observed by many researchers, for example, [1, 7, 10, 11, 16], OWL is still too limited in representing, reasoning about and merging ontologies on the Web. In particular, the following three issues are still far from solved:

- *How to represent commonsense knowledge in ontologies.*
- *How to represent and reason with multiple ontologies.*
- *How to effectively reuse and share ontologies in the Semantic Web.*

Many researchers believe that the next step in the development of the Semantic Web is to realize the logic layer. This layer will be built on top of the ontology layer and provide sophisticated representation and reasoning abilities. Given that most current reasoning systems are based on rules, it is a key task to combine rules with ontologies. The RuleML initiative (http://www.ruleml.org) is considered to be a first attempt in this direction. Theoretically, the problem of integrating the ontology layer with the logic layer is reduced to combine rule-based systems with description logics. Recently, a number of attempts at combining description logic with logic programs have been made, for example, [5, 11]. A more recent work in [7] aimed to build nonmonotonic logic programs on the top of description logic (or OWL) by combining answer set programming (ASP) and description logic. In particular, the notion of dl-atoms allows to query and virtually align the dl-knowledge base.

ASP is a paradigm of logic programming under the answer sets [9] and it is becoming one of the major tools for knowledge representation due to its simplicity, expressive power, connection to major nonmonotonic logics and efficient implementations, such as DLV [6] and Smodels [14] However, some aspects of dl-programs introduced in [7] should be extended and improved:

– dl-programs can only query or align a fixed dl-knowledge base.
– there is no construct provided in dl-programs so that concepts from different ontologies can be used. This issue can be reduced to the problem of how to merge or align ontologies.
– The operator $\ominus$ provides a means to constrain some objects from a concept but there is no construct for discarding a concept. For example, suppose we want to define a concept of $Top100Singers$. We may wish to use an ontology "MUSICIAN" on the Web and thus have to import the ontology. However, we do not want to import some irrelevant concepts like "Violinist".

In this paper we first generalize the language of dl-programs and its semantics. Then the notion of forgetting [17] is introduced into dl-programs and thus show how to use this technique to merge and align different ontologis in dl-programs.

2 Preliminaries

The language for representing ontologies in this paper is a combination of a simple description logic and extended logic programs. In this section we briefly recall some background knowledge of logic programs, description logic, and their relation to Web markup languages.

2.1 Description Logic and Web Markup Languages

Although the Web is a great success, it is basically a collection of human-readable pages that cannot be automatically processed by computer programs.The Semantic Web is to provide tools for explicit markup of Web content and to help create a repository of computer-readable information. RDF is a language that can represent explicit metadata and separate content of Web pages from their structure. However, as noted by the W3C Web Ontology Working Group (http://www.w3.org/2001/sw/WebOnt/) , RDF/RDFS is too limited to describe some application domains which require the representation of ontologies on the Web and thus, a more expressive ontology modeling language was needed. This led to a number of ontology languages for the Web including the well-known DAML+OIL [3] and OWL [4]. In general, if a language is more expressive, then it is less efficient. To suit different applications, the OWL language provides three species for users to get a better balance between expressive power and reasoning efficiency: OWL Full, OWL DL and OWL Lite.

The cores of these Semantic Web languages are description logics, and in fact, the designs of OWL and its predecessor DAML+OIL were strongly influenced by description logics, including their formal semantics and language constructors. In these Semantic Web languages, an ontology is represented as a knowledge base in a description logic.

Description logics are a family of concept-based knowledge representation languages [2]. They are fragments of first order logic and are designed to be expressively powerful and have an efficient reasoning mechanism.

A dl-knowledge base L has two components: a TBox and an ABox.

The TBox specifies the vocabulary of an application domain, which is actually a collection of concepts (sets of individuals) and roles (binary relations between individuals). So the TBox can be used to assign names to complex descriptions. For example, we may have a concept named $area$ which specifies a set of areas in computer science. Suppose we have another concept $expert$ which is a set of names of experts in computer science. We can have a role $expertIn$ which relates $expert$ to $area$. For instance, $expertIn(John,$ *"Semantic Web"*$)$ means "John is an expert in the Semantic Web".

The ABox contains assertions about named individuals.

A dl-knowledge base can also reason about the knowledge stored in the TBox and ABox, although its reasoning ability is a bit too limited for some practical applications. For example, the system can determine whether a description is consistent or whether one description subsumes another description.

The knowledge in both the TBox and ABox are represented as formulas of the first order language but they are restricted to special forms so that efficient reasoning is guaranteed. For our purpose, we deal with a simple description logic called $\mathcal{S}AL$. The formulas in $\mathcal{S}AL$ are called *concept descriptions*. Elementary descriptions consists of both *atomic concepts* and *atomic roles*. Complex concepts are built inductively as follows (in the rest of this subsection, A is an atomic concept, C and D are concept descriptions, R is a role): A (atomic concept); $\top$ (universal concept); $\bot$ (bottom concept); $\neg A$ (atomic negation); $C \sqcap D$ (intersection); $C \sqcup D$ (union). Note that we can use $\sqcup$ and $\sqcap$ to represent $\forall R.C$ (value restriction) and $\exists R.C$ (existential quantification).

To define a formal semantics of concept descriptions, we need the notion of *interpretation*. An interpretation $\mathcal{I}$ of $\mathcal{S}AL$ is a pair $(\Delta, \cdot^{\mathcal{I}})$ where Δ is a non-empty set called the *domain* and $\cdot^{\mathcal{I}}$ is an interpretation function which associates each atomic concept A with a subset $A^{\mathcal{I}}$ of Δ and each atomic role R with a binary relation $R^{\mathcal{I}} \subseteq \Delta \times \Delta$. The function $\cdot^{\mathcal{I}}$ can be naturally extended to complex descriptions:

- $\top^{\mathcal{I}} = \Delta$
- $\bot^{\mathcal{I}} = \emptyset$
- $(\neg A)^{\mathcal{I}} = \Delta - A^{\mathcal{I}}$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$

A *terminology axiom* is of the form $C \sqsubseteq D$ or $C \equiv D$ where C and D are concepts (roles). An interpretation $\mathcal{I}$ satisfies $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; it satisfies $C \equiv D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$.

2.2 Extended Logic Programs

We deal with extended logic programs [9] whose rules are built from some atoms where default negation not and strong negation $\neg$ are allowed. A literal is either an atom a or its strong negation $\neg a$[1]. For any atom a, we say a and $\neg a$ are complementary literals.

If l is a literal, then $not\ l$ is called a *negative literal*. For any set S of literals, $not\ S = \{not\ l \mid l \in S\}$.

[1] We use the same sign "$\neg$" to represent the negation in description logic and the strong negation in extended logic programs

An *extended logic program* is a finite set of rules of the following form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{1}$$

where l_0 is either a literal or empty, each l_i is a literal for $i = 1, \ldots, n$, and $0 \leq m \leq n$. If l_0 is empty, then the rule is a *constraint*.

If a rule of form (1) contains no default negation, it is called *positive*; P is a *positive program* if every rule of P is positive.

If a rule of form (1) contains only negative literals, it is called *negative*; P is a *negative program* if every rule of P is negative.

Given a rule r of form (1), $head(r) = l_0$ and $body(r) = body^+(r) \cup not\ body^-(r)$ where $body^+(r) = \{l_1, \ldots, l_m\}$, $body^-(r) = \{l_{m+1}, \ldots, l_n\}$. The set $head(P)$ consists of all literals appearing in rule heads of P.

In the rest of this section we assume that P is an extended logic program and S is a set of ground literals. A rule r in P is satisfied by S, denoted $S \models r$, iff "if $body^+(r) \subseteq S$ and $body^-(r) \cap S = \emptyset$, then $head(r) \in S$". S is a model of P, denoted $S \models P$ if every rule of P is satisfied by S.

The Answer Set Semantics. The *reduct* of logic program P on a set S of literals, written P^S, is obtained as follows:

- Delete every r from P such that there is a $not\ q \in body^-(r)$ with $q \in S$.
- Delete all negative literals from the remaining rules.

Notice that P^S is a set of rules without any negative literals. Thus P^S may have no model or have a unique minimal model, which coincides with the set of literals that can be derived by resolution.

S is an *answer set* of P if S is the minimal model of P^S.

A logic program may have zero, one or more answer sets. We use $\parallel P \parallel$ to denote the collection of answer sets of P.

A program is *consistent* if it has at least one answer set.

Two logic programs P and P' are *equivalent*, denoted $P \equiv P'$, if they have the same answer sets.

As usual, B_P is the *Herbrand base* of logic program P, that is, the set of all (ground) literals in P.

3 Answer Sets for dl-Programs

The dl-program introduced in this section is a generalization of the description logic program introduced in [7]. This language is a combination of the description logic $\mathcal{SAL}$ and extended logic programs, which allows to build logic programs on top of description logic (and thus some description logic-based web ontology languages like OWL). We will also define the answer set semantics for dl-programs.

3.1 Syntax

Informally, a dl-program consists of some dl-knowledge bases $\mathsf{L} = \{L_1, \ldots, L_s\}$ and a logic program P whose rule bodies may contain queries to some knowledge base (or its update) in L.

A *dl-query* $Q[t]$ is either a concept, or its negation, or a role, or its negation where t is a term. A *dl-atom* has the form $DL[L, S_1 \circ P_1, \ldots, S_m \circ P_m, Q](t)$ where L is a dl-knowledge base, each S_i is either a concept or a role, each P_i is a predicate; each $\circ$ is either $\oplus$ or $\ominus$, and $Q[t]$ is a dl-query. Intuitively, $S_i \oplus P_i$ and $S_i \ominus P_i$ implement views of inserting and deleting the objects satisfying the property P_i (see Section 3.2 for formal definition).

Note that we do not include the third operator in [7] because it can be represented by $\ominus$. Thus we do not need the notion of monotonicity of dl-atoms.

In the ontology $MUSICIAN$, if we are not interested in Jazz music, dl-atom $DL[Singer \ominus jazz(x), Cui]$ can be used. When there is no confusion, the L in the dl-atom can be omitted.

Definition 1. *A* dl-rule *is of the form*

$$a \leftarrow b_1, \ldots, b_r, not\ b_{r+1}, \ldots, not\ b_n$$

where $a, b_{r+1}, \ldots, b_n$ are ordinary atoms; each of $b_1, \ldots, b_r$ can be either an ordinary atom or a dl-atom.

A dl-program *is a pair (L, P) where L is a set of knowledge bases in description logic and P is a finite set of dl-rules. Sometimes, we just say P is a dl-program if there is no confusion caused.*

The dl-programs here are a bit different from the programs introduced in [7] in that a multitude of dl-knowledge bases can be queried in the same program. This is more useful for Web-based ontology representation.

A dl-program (L, P) is *positive* if it does not contain negation as failure "not ".

The *dl-base* D_P of a dl-program P is defined as the set of all ground ordinary literals in P (including the literals appearing in dl-atoms). A ground instance of a rule $r \in P$ is a rule obtained by replacing every variable in r by a constant. $ground(P)$ denotes the set of all ground instances of P. An *interpretation* I of a dl-program P is a consistent set of literals in D_P.

Let us consider the following example, which extends a scenario from http://www.kr.tuwien.ac.at/staff/roman/asp_sw/.

Example 1. Suppose that we have an ontology called ARTISTS. According to the ontology, Artists are either Singers or Painters. Some artists in the ontology are "Jodie Nash", "Vincent Van Gogh", "Luciano Pavarotti".

We also have some rules for formalizing commonsense knowledge.

If someone is an artist and there is no evidence to show that he is a painter, then he is not a painter:

$$r_1 : \neg painter(A) \leftarrow DL[L, artist](A), not\ painter(A)$$

Similarly, if someone is an artist and there is no evidence to show that he is a singer, then he is not a singer:

$$r_2 : \neg singer(A) \leftarrow DL[L, artist](A), not\ singer(A).$$

Single out cases when an artist ought to be a painter or singer (but not necessarily both)

$$r_3 : painter(A) \leftarrow DL[L \oplus Wynne(X), artist](A), DL[L, painter](A)$$

Here $L \oplus Wynne(X)$ guarantees that a Wynne Prize Winner is treated as an artist in case he is not included the ontology.

$$r_4 : singer(A) \leftarrow DL[L \ominus Jazz(X), artist](A), DL[L, singer](A).$$

Here $DL[L \ominus Jazz(X)$ shows that we are not interested in Jazz music.

Suppose we have another dl-knowledge base L_1 to check if A is a singer:

$$r_5 : singer(A) \leftarrow DL[L_1, singer](A).$$

Let L be the ontology ARTIST and $P = \{r_1, r_2, r_3, r_4, r_5\}$. Then $(\{L\}, P)$ is a dl-program.

The above dl-program intends to provide a compact representation for an ontology which extends the following owl-ontology (we omit an ontology ”Awards” containing ”Wynne” and the class ”Jazz” of ”Artist”).

```
<!DOCTYPE rdf:RDF [] > <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:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns="file://artist#"
  xmlns:base="file://artist">

  <owl:Ontology rdf:ID="artist"/>

  <owl:Class rdf:ID="Painter" />
  <owl:Class rdf:ID="Singer" />

  <owl:Class rdf:ID="Artist">
    <owl:unionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#Painter" />
      <owl:Class rdf:about="#Singer" />
    </owl:unionOf>
  </owl:Class>

  <Artist rdf:ID="Jodie_Nash"/>
  <Painter rdf:ID="Vincent_Van_Gogh"/>
  <Singer rdf:ID="Luciano_Pavarotti"/>

    </rdf:RDF>
```

3.2 Semantics

The semantics for a dl-program (L, P) is defined by the *dl-answer sets*, which modify and generalize the corresponding notion in [7].

An interpretation I of a dl-program P is a consistent set of ground literals. So we also allow dl-atoms in an interpretation.

The following interpretations for operators $\oplus$ and $\ominus$ is expected to remedy some problems in the original definition.

Definition 2. *Let I be an interpretation of the dl-program P and $DL[L, S_1 \circ P_1, \ldots, S_m \circ P_m, Q](t)$ is a ground dl-atom in P.*

Denote $\odot^I$ the interpretation obtained from $\cdot^I$ by replacing $(S_i)^I$

– *with $(S_i)^I \cup \{t \mid P_i(t)\}$ if S_i is a concept and $\circ = \oplus$;*
– *with $(S_i)^I - \{t \mid P_i(t)\}$ if S_i is a concept and $\circ = \ominus$;*
– *with $(S_i)^I \cup \{(t_1, t_2) \mid P_i(t_1, t_2)\}$ if S_i is a role and $\circ = \oplus$;*
– *with $(S_i)^I - \{(t_1, t_2) \mid P_i(t_1, t_2)\}$ if S_i is a role and $\circ = \ominus$;*

The resulting semantic relation is denoted $\models_{dl}$.

$L \models DL[L, S_1 \circ P_1, \ldots, S_m \circ P_m, Q](t)$ if and only if $L \models_{dl} Q(t)$.

Note that every positive dl-program P has a least model, denoted $M(P)$. A general dl-program can be reduced into a positive dl-program with respect to an interpretation.

Let P be a dl-program and I a set of atoms in P. The reduct P^I of P on I is the positive dl-program obtained from $ground(P)$ by the following three ordered steps :

1. adding a rule $Q(t) \leftarrow$ for each dl-atom d in the body of a rule in $ground(P)$ such that $L \models Q(t)$, where $Q(t)$ is the query for d;
2. deleting every rule r in $ground(P)$ such that $b \in I$ for some $not\ b$ in the body of r and
3. deleting every $not\ b$ in the remaining rules.

The first condition says a dl-atom must be true in the model if it occurs in the program and can be derived from the corresponding dl-knowledge base; the second and third conditions are inherited from the definition of standard answer sets.

Definition 3. *Let P be a dl-program and S a set of literals in P. S is a dl-answer set if $M(P^S) = S$.*

A dl-answer set S is *consistent* if (1) there is no atom a such that both a and $\neg a$ in S and (2) if d is a dl-atom and $d \in S$, then $L \not\models_{dl} \neg Q(t)$ where $Q(t)$ is the query of d.

A dl-program may have zero, one or more dl-answer sets. We use $\parallel P \parallel$ to denote the collection of answer sets of P.

Consider the dl-program $(\{L, L_1\}, P)$ again where L_1 is a dl-knowledge base which includes “Tweety” as a singer. Then this dl-program has the unique dl-answer set which contains some information like $painter(VincentVanGogh)$ and $singerTweety$.

A dl-program is *consistent* if it has at least one consistent dl-answer set.

Two dl-programs P and P' are *equivalent*, denoted $P \equiv P'$, if they have the same dl-answer sets.

4 Forgetting in dl-Programs

In this section we introduce the notion of forgetting for dl-programs. That is, we want to define what it means to forget about (or discard) a literal l in a dl-program P. The intuition behind the forgetting theory is to obtain a dl-program which is equivalent to the original dl-program if we ignore the existence of the literal l.

4.1 Forgetting Ordinary Atoms

It is easy to forget a literal l in a set X of literals, that is, just remove l from X if $l \in X$. This notion of forgetting can be easily extended to subsets. A set X' is an l-subset of X if $X' - \{l\} \subseteq X - \{l\}$. Similarly, a set X' is a true l-subset of X if $X' - \{l\} \subset X - \{l\}$.

Two sets X and X' of literals are l-equivalent, denoted $X \sim_l X'$, iff $(X - X') \cup (X' - X) \subseteq \{l\}$.

Given a consistent dl-program P and an ordinary ground literal l, we could define a result of forgetting about l in P as a dl-program P' whose dl-answer sets are exactly $\parallel P \parallel -l = \{X - \{l\} \mid X \in \parallel P \parallel\}$. However, such a notion of forgetting cannot even guarantee the existence for some simple programs as illustrated in [17]. So *we need a notion of minimality of dl-answer sets which can naturally combine the definition of dl-answer sets, minimality and forgetting together.*

Definition 4. *Let P be a consistent dl-program, l an ordinary ground literal in P and X a set of ground literals.*

1. *We say X is* l-minimal *in a collection $\mathcal{S}$ of sets of ground literals if $X \in \mathcal{S}$ and there is no $X' \in \mathcal{S}$ such that X' is a true l-subset of X . In particular, if $\mathcal{S}_P$ is the set of models of P, then we say X is an* l-minimal model *of dl-program P if X is a model of P and it is l-minimal in $\mathcal{S}_P$.*
2. *X is a* dl-answer set of P by forgetting l *(briefly,* l-answer set*) if X is the l-minimal model of the reduct P^X.*

The above definition is a filter for dl-answer sets rather than a new semantics.

Having the notion of minimality about forgetting an ordinary ground literal, we are now in a position to define the result of forgetting about a literal in a dl-program.

Definition 5. *Let P be a consistent dl-program and l be an ordinary ground literal. A dl-program P' is a result of* forgetting about l in P *if the following conditions are satisfied:*

1. *$D_{P'} \subseteq D_P - \{l\}$.*
2. *For any set X' of ground literals, X' is a dl-answer set of P' iff there is an l-answer set X of P such that $X' \sim_l X$.*

Notice that the first condition implies that l does not appear in P'. In particular, no new symbol is introduced in P'.

Suppose we have a dl-knowledge base L which contains some concepts "bird", "parrot" and "penguin". An ontology "BIRD" is specified as a dl-program (L, P) where

$P = P_1 \cup P_2$, P_2 contains no information about "penguin" and P_1 consists of the following rules:

$$\begin{aligned} bird(A) &\leftarrow penguin(A) \\ flies(A) &\leftarrow bird(A), not\ penguin(A) \\ \neg flies(A) &\leftarrow penguin(A) \\ penguin(Tweety) &\leftarrow \end{aligned}$$

If we do not want to import the concept "penguin", we can discard the information on "penguin" by forgetting and get $\mathsf{forget}((L,P), penguin) = \{flies(A) \leftarrow bird(A)\} \cup P_2$.

A dl-program P may have different dl-programs as results of forgetting about the same ordinary ground literal l. However, it follows from the above definition that any two results of forgetting about the same literal in P are equivalent under dl-answer sets.

Proposition 1. *Let P be a dl-program and l an ordinary ground literal in P. If P' and P'' are two results of forgetting about l in P, then P' and P'' are equivalent (i.e. they have the same dl-answer sets).*

We use $\mathsf{forget}(P, l)$ to denote the result of forgetting about l in P.

To compute $\mathsf{forget}(P, l)$, we can easily adapt the corresponding algorithms in [17] to dl-programs.

Similarly, we can forget a set of ordinary literals F in a dl-program P and thus define $\mathsf{forget}(P, F)$.

4.2 Forgetting dl-Atoms

To discard or forget an unwanted ground dl-atom d in a dl-program P, we need to remove all the effects caused by d in both P and L. This can be accomplished by the following steps:

Step 1. Forget d in dl-knowledge base L by removing all those concepts, roles and terminology axioms of L in which $Q(t)$ occurs. Denote the resulting dl-knowledge base $L - d$.
Step 2. Replace each occurrence of the dl-atom d by d' in P where d' is obtained from d by replacing L with $L - d$. The resulting dl-program is denoted P'.
Step 3. Forget the dl-atom d' in P' by treating d' as an ordinary atom.

5 Merging and Aligning Ontologies

In recent years, researchers have developed many ontologies. These different groups of researchers are now beginning to work with each other, so they must bring together ontologies from different sources. Approaches to this problem usually fall into one of the two categories:

– merging the ontologies to create a single coherent ontology.
– aligning the ontologies by establishing links between them to reuse information from one another.

In this section we show how to merge and align ontologies in dl-programs by forgetting.

For simplicity, throughout the discussion, we assume that only two ontologies are being merged or aligned.

5.1 Merging Ontologies by Forgetting

When two ontologies are merged, a new ontology is created, which is a merged version of the original ontologies. Usually, overlapping domains are kept in the merged ontology. Some algorithms like SMART [15] tried to automate parts of the merging process of ontologies. However, their languages are relatively simple and thus reasoning is almost not involved.

As shown in the algorithm of SMART, those concepts and roles to be merged can be specified by users and/or automatic processes. For instance, Linguistically similar names can be found automatically. Linguistic similarity can be determined in a couple of different ways including by synonymy or shared substrings.

Let O_1 and O_2 be two ontologies expressed as dl-programs. Suppose we have determined two sets of literals F_1 and F_2 for O_1 and O_2, respectively. F_1 and F_2 correspond to certain concepts that need to be handled separately in the merging. A literal is put into F_1 or/and F_2 due to a number of reasons. However, in an expressive language like dl-programs, these two concepts may be related to some other concepts by terminology axioms. Thus we may have to deal with some issues related to reasoning like conflict resolving and consistency maintaining. Another possibility is that some concepts are useless and we want to discard them as mentioned in the introduction. The second scenario can be satisfactorily handled by the following algorithm.

Algorithm 1 Input*: Two ontologies O_1 and O_2 (in dl-programs).*
Output*: A merged ontology O.*
Process*:*

```
Step 1. Determine the sets $F_1$ and $F_2$ of literals that need to be handled separately.
Step 2. Compute $\mathsf{forget}(O_i, F_i)$ for $i = 1, 2$.
Step 3. $F_1$ and $F_2$ are handled by user and thus $O_0$ is obtained.
Step 4. Merged ontology: $O = O_0 \cup \mathsf{forget}(O_1, F_1) \cup \mathsf{forget}(O_2, F_2)$.
```

As for Step 3, it depends on the application. There may be a couple of possible different approaches. For example, we may remove some literals from F_1 and/or F_2; we may replace some literal of one F_i with a literal in the other; or we may even replace two literals $l_i \in F_i$, $i = 1, 2$ with a new literal.

5.2 Aligning Ontologies by Forgetting

In alignment, the two original ontologies persist but one of the aligned ontologies (say O_1, more general) is preferred over another (say O_2, more specific), with links established between their concepts and roles. These links can be represented as a mapping or a view (virtual ontology). However, the domain-specific ontology O_2 does not become part of the more general ontology O_1; rather O_2 is a separate ontology that includes O_1 and uses O_1's top-level distinctions. For example, many ontologies in the domain of military are structured around a central ontology, the CYC knowledge base [13]. The developers of these domain-specific ontologies then align their ontologies to CYC by establishing links into CYC's upper- and middle-level ontologies [8]. However, in many cases the alignment cannot be done by only some simple links. As in the case of merging, conflicts and/or inconsistencies may arise when aligning two ontologies. In particular, we have to deal with their subsumption relations. For example, if we have two ontologies ARTIST, where a concept "Singer" is included, and MUSICIAN, where a concept "singer" is included, we may wish to merge "singer" and "Singer". Moreover, ARTIST is established by the Australian Association of Arts and MUSICIAN by the College of Music at Griffith University. Since both "Singer" and "singer" may be related to some other concepts in their ontologies by roles and axioms, we cannot simply replace "singer" by "Singer".

By employing the notion of forgetting, the tasks of conflict resolution and consistency maintenance during aligning can be done automatically.

Algorithm 2 Input*: Two ontologies O_1 and O_2 (in dl-programs) where O_1 is preferred to O_2.*
Output*: Aligned ontology O.*
Process*:*

Step 1. *Determine the set F_2 of atoms that will be aligned in O_2.*
Step 2. *Compute* $\mathsf{forget}(O_2, F_2)$.
Step 3. *Aligned ontology:* $O = O_1 \cup \mathsf{forget}(O_2, F_2)$.

6 Conclusions

The language of dl-programs is a latest effort in developing an expressive representation for Web-based ontologies. It allows to build answer set programming (ASP) on top of description logic and thus some attractive features of ASP can be employed in the design of the Semantic Web architecture. Often, an ontology on the Web is based on more than one knowledge base. In this paper we have generalized dl-programs by allowing multiple knowledge bases and then accordingly, defined the answer set semantics for the dl-programs. The notion of forgetting has been proved an extremely useful technique for updating knowledge bases, constraint problem solving and query answering [12, 17, 18]. In this paper we have imported the notion of forgetting into dl-programs. We have also applied the technique of forgetting to two important tasks of representing ontologies, that is, merging and aligning ontologies. In particular, we have introduced two algorithms for these two tasks. This is only preliminary report of our work. There are a couple of issues to be pursued in the future:

– More constructs in ASP can be introduced into dl-programs, like disjunction and preference. The major difficulty in allowing these constructs is how to design corresponding algorithms for forgetting.
– It is also important to see if more expressive description logic can be allowed in the forgetting of dl-programs.
– The two algorithms for merging and aligning ontologies need further improvement. We are also planning to implement them and apply to some practical application domains.

References

1. G. Antoniou and G. Wagner. A rule-based approach to the semantic web (preliminary report). In *Proceedings of the 2nd Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML2003)*, pages 111–120, 2003.
2. F. Baader, D. Calvanese, D.McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2002.
3. D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Daml+oil reference description. http://www.w3.org/tr/2001/note-daml+oil-reference-20011218.html, W3C Note, 18 December 2001.
4. M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. Owl web ontology language reference. http://www.w3.org/tr/2004/rec-owl-ref-20040210/, 3C Recommendation, 10 February 2004.
5. F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. AL-log: Integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.
6. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A kr system dlv: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann Publishers, 1998.
7. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning*, pages 141–151, 2004.
8. R. Fikes and A. Farquhar. Large-scale repositories of highly expressive reusable knowledge. *IEEE Intelligent Systems*, 14(2):73–79, 1999.
9. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the International Conference on Logic Programming*, pages 579–597, 1990.
10. B. Grau, B. Parsia, and E. Sirin. Combining owl ontologies using e-connections. Technical Report TR-2005-01, University of Maryland Institute for Advanced Computer Studies (UMIACS), January 2005.
11. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logics. In *Proceedings of the 12th International World Wide Web Conference*, pages 48–57, 2003.
12. J. Lang, P. Liberatore, and P. Marquis. Propositional independence: Formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.
13. D. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of ACM*, 38(11):33–38, 1995.
14. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 420–429. Springer-Verlag, 1997.
15. N. Noy and M. Musen. An algorithm for merging and aligning ontologies: Automation and tool support. In *Proceedings of the Workshop on Ontology Management at AAAI-99*, 1999.
16. T. Swift. Deduction in ontologies via asp. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, 2004.
17. K. Wang, A. Sattar, and K. Su. A theory of forgetting in logic programming. In *Proceedings of the AAAI National Conference on Artificial Intelligence*. AAAI Press, 2005.
18. Y. Zhang, N. Foo, and K. Wang. Solving logic program conflicts through strong and weak forgettings. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 627–632. the Professional Book Centre, USA, 2005.

A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web

Nick Bassiliades[1], Efstratios Kontopoulos[1], and Grigoris Antoniou[2]

[1] Department of Informatics, Aristotle University of Thessaloniki
GR-54124 Thessaloniki, Greece
{nbassili,skontopo}@csd.auth.gr
[2] Institute of Computer Science, FO.R.T.H., P.O. Box 1385, GR-71110, Heraklion, Greece
antoniou@ics.forth.gr

Abstract. Defeasible reasoning is a rule-based approach for efficient reasoning with incomplete and inconsistent information. Such reasoning is useful for many applications in the Semantic Web. However, the RuleML syntax of defeasible logic may appear too complex for many users. Furthermore, the interplay between various technologies and languages, such as defeasible reasoning, RuleML, and RDF impose a demand for using multiple, diverse tools for building rule-based applications for the Semantic Web. In this paper we present VDR-Device, a visual integrated development environment for developing and using defeasible logic rule bases on top of RDF ontologies. VDR-Device integrates in a user-friendly graphical shell, a visual RuleML-compliant rule editor that constrains the allowed vocabulary through analysis of the input RDF ontologies and a defeasible reasoning system that processes RDF data and RDF Schema ontologies.

1 Introduction

Although the Semantic Web represents a recent initiative to improve the potential of the existing Web, it undoubtedly constitutes the inspiration for a vast number of applications. However, only the basic layers of the Semantic Web [9] have achieved a certain level of maturity, with the highest one of them being the ontology layer, where OWL has become the dominant standard. The next layers that have to become more "concrete" are the logic and proof layers. Rule-based systems seem to possess a key role in this affair, since (a) they can serve as extensions of, or alternatives to, description logic based ontology languages; and (b) they can be used to develop declarative systems on top of (using) ontologies.

Defeasible reasoning [20], a member of the non-monotonic reasoning family, constitutes a simple rule-based approach to reasoning with incomplete and conflicting information. This approach offers two main advantages: (a) enhanced representational capabilities, allowing one to reason with incomplete and contradictory information, coupled with (b) low computational complexity compared to mainstream non-monotonic reasoning. Defeasible reasoning can represent facts, rules as well as priorities and conflicts among rules. Such conflicts arise, among others, from rules with exceptions, which are a natural representation for policies and business rules [2]. And priority information is often available to resolve conflicts among rules. Potential applications include security policies ([6], [17]), business rules [1], e-contracting [14], personalization, brokering [5], bargaining and agent negotiations ([13], [21]).

Although defeasible logic is certainly a very promising reasoning technology for the Semantic Web, the development of rule-based applications for the Semantic Web can be greatly compromised by two factors. First, the RuleML syntax of defeasible logic is certainly too complicated for an end-user language. Furthermore, the interplay between various technologies and languages involved in such applications, namely defeasible reasoning, RuleML and RDF, impose a demand for using multiple, diverse tools, which is a high burden even for the developer.

In this paper we present VDR-Device, a visual integrated development environment for developing and using defeasible logic rule bases on top of RDF ontologies. VDR-Device integrates in a user-friendly graphical shell, a visual RuleML-compliant rule editor and a defeasible reasoning system that processes RDF data and RDF Schema ontologies [7]. The rule editor helps users to develop a defeasible logic rule base by constraining the allowed vocabulary after analyzing the input RDF ontologies. Therefore, it removes from the user the burden of typing-in class and property names and prevents potential semantical and syntactical errors. The visualization of rules follows the tree model of RuleML.

VDR-DEVICE supports multiple rule types of defeasible logic, as well as priorities among rules. Furthermore, it supports two types of negation (strong, negation-as-failure) and conflicting (mutually exclusive) literals. DR-DEVICE has a RuleML-compatible [10] syntax, which is the main standardization effort for rules on the Semantic Web. Input and output of data and conclusions is performed through processing of RDF data and RDF Schema ontologies. The system is built on-top of a CLIPS-based implementation of deductive rules, namely the R-DEVICE system [8]. The core of the system consists of a translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes.

The rest of the paper is organized as follows: Section 2 introduces a brokering trade example that is used throughout the paper. Section 3 briefly introduces the semantics of defeasible logics. Section 4 presents the architecture and functionality of the VDR-Device system, including the visual rule editor and the core reasoning system. Finally, section 5 discusses related work and section 6 concludes the paper and discusses future work.

2 A Defeasible Logic Example

This section briefly presents an example of a defeasible logic program, adopted from [4], that is used throughout this paper to explicate the workings of defeasible logic and VDR-Device. The example deals with a brokered trade application that takes place via an independent third party, the broker, and more specifically with apartment renting. A number of available apartments reside in an RDF document along with the properties of each apartment (Fig. 1). The potential user expresses his/her requirements in defeasible logic (as explained in the following section), regarding the apartment he/she wishes to rent. The broker then tries to match the customer's requirements and the apartment specifications and proposes a deal when both parties can be satisfied by the trade.

The potential renter is looking for an apartment of at least 45m^2 with at least 2 bedrooms. If it is on the 3rd floor or higher, the house must have an elevator. Also, pet animals must be allowed. He is willing to pay \$300 for a centrally located 45m^2

apartment, and \$250 for a similar flat in the suburbs. In addition, he is willing to pay an extra \$5 per m^2 for a larger apartment, and \$2 per m^2 for a garden. He is unable to pay more than \$400 in total. If given the choice, he would go for the cheapest option. His 2nd priority is the presence of a garden; lowest priority is additional space.

```
<rdf:RDF ... xmlns:carlo="&carlo;" xmlns:carlo_ex="&carlo_ex;">
   <carlo:apartment rdf:about="&carlo_ex;a1">
   <carlo:bedrooms rdf:datatype="&xsd;integer">1</carlo:bedrooms>
   <carlo:central>yes</carlo:central>
   <carlo:floor rdf:datatype="&xsd;integer">1</carlo:floor>
   <carlo:gardenSize rdf:datatype="&xsd;integer">0</carlo:gardenSize>
   <carlo:lift>no</carlo:lift>
   <carlo:name>a1</carlo:name>
   <carlo:pets>yes</carlo:pets>
   <carlo:price rdf:datatype="&xsd;integer">300</carlo:price>
   <carlo:size rdf:datatype="&xsd;integer">50</carlo:size>
   </carlo:apartment>
   ...
</rdf:RDF>
```

Fig. 1. RDF document excerpt for available apartments

3 Defeasible Logics – An Introduction

A *defeasible theory D* (i.e. a knowledge base or a program in defeasible logic) consists of three basic ingredients: a set of facts (F), a set of rules (R) and a superiority relationship (>). Therefore, D can be represented by the triple (F, R, >).

In defeasible logic, there are three distinct types of rules: strict rules, defeasible rules and defeaters. *Strict rules* are denoted by $A \rightarrow p$ and are interpreted in the typical sense: whenever the premises are indisputable, so is the conclusion. An example of a strict rule is: "*Apartments are houses*", which, written formally, would become:

r_1: apartment(X) $\longrightarrow$ house(X)

Defeasible rules are rules that can be defeated by contrary evidence and are denoted by $A \Rightarrow p$. An example of such a rule is "*Any apartment is considered to be acceptable*", which becomes: r_2: apartment(X) $\Rightarrow$ acceptable(X).

Defeaters, denoted by *A ~> p*, are rules that do not actively support conclusions, but can only prevent some of them. In other words, they are used to defeat some defeasible rules by producing evidence to the contrary. An example of a defeater is:

r_3: ¬pets(X),gardenSize(X,Y),Y>0 ~> acceptable(X)

which reads as: *"If pets are not allowed in the apartment, but the apartment has a garden, then it might be acceptable"*. This defeater can defeat, for example, rule

r_4: ¬pets(X) $\Rightarrow$ ¬acceptable(X).

Finally, the *superiority relationship* among the rule set R is an acyclic relation > on R. For example, given the defeasible rules r_2 and r_4, no conclusive decision can be made about whether the apartment is acceptable or not, because rules r_2 and r_4 contradict each other. But if a superiority relation > with $r_4 > r_2$ is introduced, then r_4 overrides r_2 and we can indeed conclude that the apartment is considered unacceptable. In this case rule r_4 is called *superior* to r_2 and r_2 *inferior* to r_4.

Another important element of defeasible reasoning is the notion of *conflicting literals*. In applications, literals are often considered to be conflicting and at most one of a certain set should be derived. An example of such an application is price negotiation, where an offer should be made by the potential buyer. The offer can be determined by several rules, whose conditions may or may not be mutually exclusive. All rules have `offer(X)` in their head, since an offer is usually a positive literal. However, only one offer should be made; therefore, only one of the rules should prevail, based on superiority relations among them. In this case, the conflict set is:

```
C(offer(x,y)) = { ¬offer(x,y) } ∪ { offer(x,z) | z ≠ y }
```

For example, the following two rules make an offer for a given apartment, based on the buyer's requirements. However, the second one is more specific and its conclusion overrides the conclusion of the first one.

r_5: `size(X,Y),Y≥45,garden(X,Z) ⇒ offer(X,250+2Z+5(Y-45))`
r_6: `size(X,Y),Y≥45,garden(X,Z),central(X) ⇒ offer(X,300+2Z+5(Y-45))`
$r_6 > r_5$

4 VDR-Device System Architecture

The VDR-Device system consists of two primary components:

1. DR-Device, the reasoning system that performs the RDF processing and inference and produces the results, and
2. DRREd (Defeasible Reasoning Rule Editor), the rule editor, which serves both as a rule authoring tool and as a graphical shell for the core reasoning system.

Although these two subsystems utilize different technologies and were developed independently, they intercommunicate efficiently, forming a flexible and powerful integrated environment.

4.1 Architecture and Functionality of the Reasoning System

The core reasoning system of VDR-Device is DR-Device [6] and consists of two primary components (Fig. 2): The *RDF loader/translator* and the *rule loader/translator*. The user can either develop a rule base (program, written in the RuleML-like syntax of VDR-Device) with the help of the rule editor described in the following sections, or he/she can load an already existing one, probably developed manually. The rule base contains: (a) a set of rules, (b) the URL(s) of the RDF input document(s), which is forwarded to the RDF loader, (c) the names of the derived classes to be exported as results and (d) the name of the RDF output document.

The rule base is then submitted to the *rule loader* which transforms it into the native CLIPS-like syntax through an XSLT stylesheet and the resulting program is then forwarded to the *rule translator*, where the defeasible logic rules are compiled into a set of CLIPS production rules [11]. This is a two-step process: First, the defeasible logic rules are translated into sets of deductive, derived attribute and aggregate attribute rules of the basic deductive rule language, using the translation scheme described in [7]. Then, all these deductive rules are translated into CLIPS production rules according to the rule translation scheme in [8]. All compiled rule formats are also kept in local files (structured in project workspaces), so that the next time they are needed they can be directly loaded, improving speed considerably (running a compiled project is up to 10 times faster).

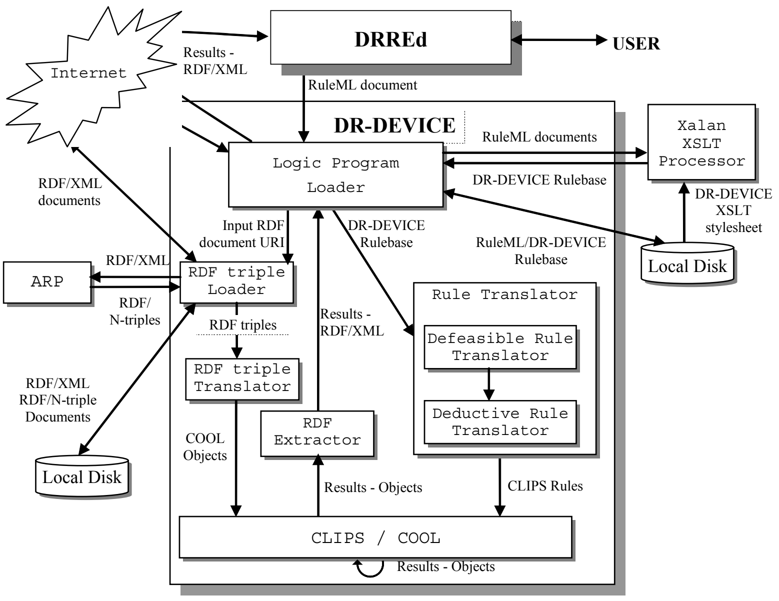


Fig. 2. The architecture of the core reasoning system

Meanwhile, the *RDF loader* downloads the input RDF documents, including their schemas, and translates RDF descriptions into CLIPS objects [11], according to the RDF-to-object translation scheme in [8], which is briefly described below.

The inference engine of CLIPS performs the reasoning by running the production rules and generates the objects that constitute the result of the initial rule program. The compilation phase guarantees correctness of the reasoning process according to the operational semantics of defeasible logic. Finally, the result-objects are exported to the user as an RDF/XML document through the RDF extractor. The RDF document includes the instances of the exported derived classes, which have been proved.

The Object-Oriented RDF Data Model

The DR-Device system employs an OO RDF data model, which is different from the established triple-based data model for RDF. The main difference is that DR-Device treats properties both as first-class objects and as normal encapsulated attributes of resource objects. In this way properties of resources are not scattered across several triples as in most other RDF inferencing systems, resulting in increased query performance due to less joins. For example, the apartment in Fig. 1 is transformed into the COOL object displayed in Fig. 3.

The Defeasible Logic Language

DR-Device supports two syntaxes for defeasible logic rules: a native CLIPS-like one and a RuleML-compatible one. Here we focus solely on the latter, since the rule editor of the system allows the expression of rules only in this syntax. While the RuleML syntax utilizes as many features of the official RuleML as possible, several of the features of the rule language cannot be expressed by the existing RuleML DTDs and/or XML Schema documents. A new DTD (v. 0.85 compatible) and new XML Schema documents (0.86, 0.89 compatible) were, therefore, developed using the modularization scheme of RuleML, extending the OO-Datalog with strong negation and negation-as-failure version of RuleML. Fig. 4 shows a self-contained simplified version of the DTD, while the original DTD and the XML Schema documents can be found at `http://lpis.csd.auth.gr/systems/dr-device.html`, along with the system itself. Notice, that the system currently uses the v. 0.85 compatible DTD.

```
[carlo_ex:a1] of carlo:apartment
(carlo:size 50)                  (carlo:gardenSize 0)
(carlo:price 300)                (carlo:floor 1)
(carlo:pets "yes")               (carlo:central "yes")
(carlo:name "a1")                (carlo:bedrooms 1)
(carlo:lift "no")
```

Fig. 3. COOL object for the apartment of Fig. 1

A defeasible logic rule is represented by an `imp` element and consists of three sub-elements: the head and body of the rule (`_head` and `_body` elements respectively) as well as a label, encoded in a `_rlab` element, which includes the rule's unique ID (`ruleID` attribute) and its type (`ruletype` attribute). The latter can only take three distinct values (`strictrule`, `defeasiblerule`, `defeater`).

For example, the defeasible rule r_2 of the previous section is represented as:

```
<imp>
  <_rlab ruleID="r2" ruletype="defeasiblerule"><ind>r2</ind></_rlab>
  <_head>   <atom> <_opr><rel>acceptable</rel></_opr>
                   <_slot name="name"><var>X</var></_slot> </atom>
  </_head>
  <_body>   <atom> <_opr><rel href="carlo:apartment"/></_opr>
                   <_slot name="name"><var>X</var></_slot> </atom>
  </_body>
</imp>
```

The names (`rel` elements) of the operator (`_opr`) elements of atoms are class names, since atoms actually represent CLIPS objects. RDF class names used as base classes in the rule condition are referred to through the `href` attribute of the `rel` element, while derived class names (e.g. `acceptable`) are text values of the `rel` element. Atoms have named arguments, called slots, which correspond to object properties. Since RDF resources are represented as CLIPS objects, atoms correspond to queries over RDF resources of a certain class with certain property values.

Superiority relations are represented as attributes of the superior rule. For example, rule r_4, which is superior to r_2, is represented as follows:

```
<imp>
  <_rlab ruleID="r4" ruletype="defeasiblerule" superior="r2">
        <ind>r4</ind> </_rlab>
  <_head><neg> <atom> <_opr><rel>acceptable</rel></_opr>
                      <_slot name="name"><var>X</var></_slot> </atom>
        </neg> </_head>
  <_body>   <atom> <_opr><rel href="carlo:apartment"/></_opr>
```

```
<!ENTITY % URI "CDATA">
<!ELEMENT rulebase (_rbaselab, (imp | comp_rules)*)>
<!ATTLIST rulebase      rdf_import CDATA #IMPLIED
                        rdf_export_classes NMTOKENS #IMPLIED
                        rdf_export CDATA #IMPLIED>
<!ELEMENT _rbaselab (ind)>
<!ELEMENT imp (_rlab, _head, _body)>
<!ELEMENT comp_rules (_crlab)>
<!ATTLIST comp_rules    c_rules IDREFS #REQUIRED
                        slotnames NMTOKENS #IMPLIED>
<!ELEMENT _rlab (ind)>          <!ELEMENT _crlab (ind)>
<!ATTLIST _rlab ruleID ID #REQUIRED
        ruletype (strictrule | defeasiblerule | defeater) #REQUIRED
        superior IDREFS #IMPLIED>
<!ELEMENT _head (calc?, (atom | neg))>
<!ELEMENT _body (atom | neg | and)>
<!ELEMENT calc (fun_call+)>
<!ELEMENT fun_call (ind|var|fun_call)*>
<!ATTLIST fun_call              name CDATA #REQUIRED>
<!ELEMENT naf (atom | and)>             <!ELEMENT neg (atom)>
<!ELEMENT and ((atom | naf)*)>  <!ELEMENT atom (_opr, _slot*)>
<!ELEMENT _opr (rel)>                   <!ELEMENT rel (#PCDATA)>
<!ATTLIST rel           href %URI; #IMPLIED>
<!ELEMENT _slot (ind | var | _not | _or | _and)>
<!ATTLIST _slot name CDATA #REQUIRED>
<!ELEMENT _not (ind | var)>
<!ELEMENT _or ((_not|ind|var|fun_call),(_not|ind|var|fun_call)+)>
<!ELEMENT _and ((_not|ind|var|fun_call),(_not|ind|var|fun_call)+)>
<!ELEMENT ind (#PCDATA)>                <!ELEMENT var (#PCDATA)>
<!ATTLIST ind           type CDATA #IMPLIED     href %URI; #IMPLIED>
```

Fig. 4. DTD for the RuleML syntax of the defeasible logic rule language

```
                <_slot name="carlo:name"><var>X</var></_slot>
                <_slot name="carlo:pets"><ind>no</ind></_slot>
        </atom>   </_body>
</imp>
```

Negation is represented via a `neg` element that encloses an `atom` element. Apart from rule declarations, there are `comp_rules` elements that declare groups of competing rules which derive competing positive conclusions (*conflicting literals*). For example, in the apartment rent example, rules r_5 and r_6 are competing over the conclusion `offer(X,Y)`, since at most one offer can be made:

```
<comp_rules c_rules="r5 r6">
  <_crlab> <ind>cr1</ind> </_crlab>
</comp_rules>
```

Further extensions to the RuleML syntax, include function calls that are used either as constraints in the rule body or as new value calculators in the rule head. Additionally, multiple constraints in the rule body can be expressed through the logical operators: `_not`, `_and`, `_or`, whose semantics are similar to the CLIPS *connective constraints* [11]. Finally, the header of the rule base, namely the `rulebase` root element of the RuleML document, includes a number of important parameters, which are implemented as attributes: `rdf_import` declares the input RDF file(s), `rdf_export` represents the RDF file that contains the exported results and `rdf_export_classes` represents the derived classes, whose instances will be exported in RDF/XML format. An example of all of the above is shown below:

```
<rulebase rdf_import="http://lpis.csd.auth.gr/.../carlo.rdf#"
          rdf_export="http://lpis.csd.auth.gr/.../export-carlo.rdf"
          rdf_export_classes="acceptable rent">
```

4.2 Rule Editor – Design and Functionality

Writing rules in RuleML can often be a highly cumbersome task. Thus, the need for authoring tools that assist end-users in writing and expressing rules is apparently imperative. VDR-Device is equipped with DRREd, a Java-built visual rule editor that aims at enhancing user-friendliness and efficiency during the development of VDR-Device RuleML documents. Its implementation is oriented towards simplicity of use and familiarity of interface. Other key features of the software include: (a) functional flexibility - program utilities can be triggered via a variety of overhead menu actions, keyboard shortcuts or popup menus, (b) improved development speed - rule bases can be developed in just a few steps and (c) powerful safety mechanisms – the correct syntax is ensured and the user is protected from syntactic or RDF Schema related semantic errors.

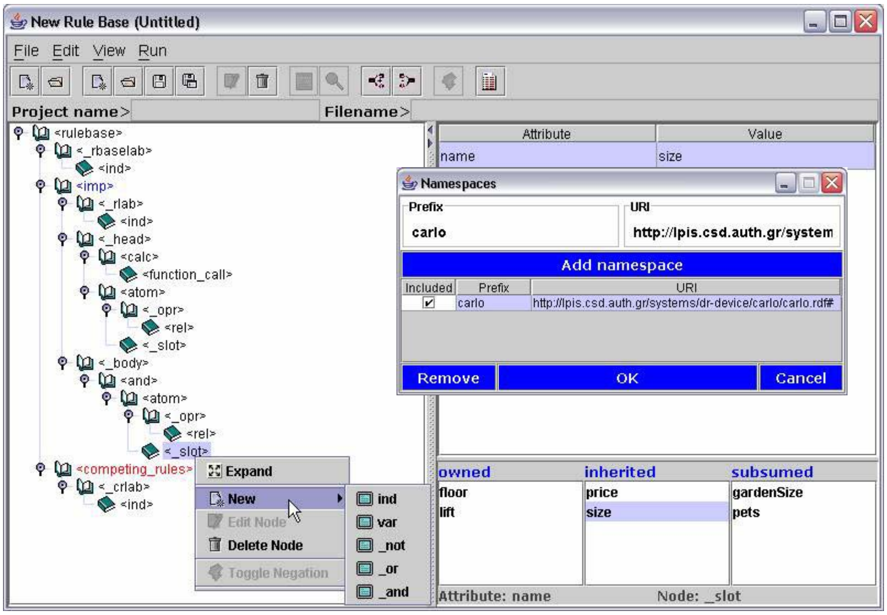
More specifically, and as can be observed in Fig. 5, the main window of the program is composed of two major parts: a) the upper part includes the menu bar, which contains the program menus, and the toolbar that includes icons, representing the most common utilities of the rule editor, and b) the central and more "bulky" part is the primary frame of the main window and is in turn divided in two panels:

The left panel displays the rule base in XML-tree format, which is the most intuitive means of displaying RuleML-like syntax, because of its hierarchical nature. The user has the option of navigating through the entire tree and can add to or remove elements from the tree. However, since each rule base is backed by a DTD document, potential addition or removal of tree elements has to obey to the DTD limitations. Therefore, the rule editor allows a limited number of operations performed on each element, according to the element's meaning within the rule tree.

The right panel shows a table, which contains the attributes that correspond to the selected tree node in the left-hand area. The user can also perform editing functions on the attributes, by altering the value for each attribute in the panel that appears below the attributes table on the right-hand side. The values that the user can insert are obviously limited by the chosen attribute each time.

The development of a rule base using VDR-Device is a delicate process that depends heavily on the parameters around the node that is being edited each time. First of all, there is an underlying procedure behind tree expansion, which is "launched" each time the user is trying to add a new element to the rule base. Namely, when a new element is added to the tree, all the mandatory sub-elements that accompany it are also added. In the cases where there are multiple alternative sub-elements, none of them is added to the rule base and the final choice is left to the user to determine which one of them has to be added. The user has to right-click on the parent element and choose the desired sub-element from the pop-up menu that appears (Fig. 5).

Another important component is the namespace dialog window (Fig. 5), where the user can determine which RDF/XML namespaces will be used by the rule base. Actually, we treat namespaces as addresses of input RDF Schema ontologies that contain the vocabulary for the input RDF documents, over which the rules will be run. The namespaces entered by the user, as well as those contained in the input RDF documents (indicated by the `rdf_import` attribute of the `rulebase` root element), are analyzed in order to extract all the allowed class and property names for the rule base being developed (see next section). These names are then used throughout the authoring phase of the RuleML rule base, constraining the corresponding allowed names that can be applied and narrowing the possibility for errors on behalf of the user.



Attribute	Value
name	size

Prefix	URI
carlo	http://lpis.csd.auth.gr/system

Included	Prefix	URI
✓	carlo	http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf#

owned	inherited	subsumed
floor	price	gardenSize
lift	size	pets

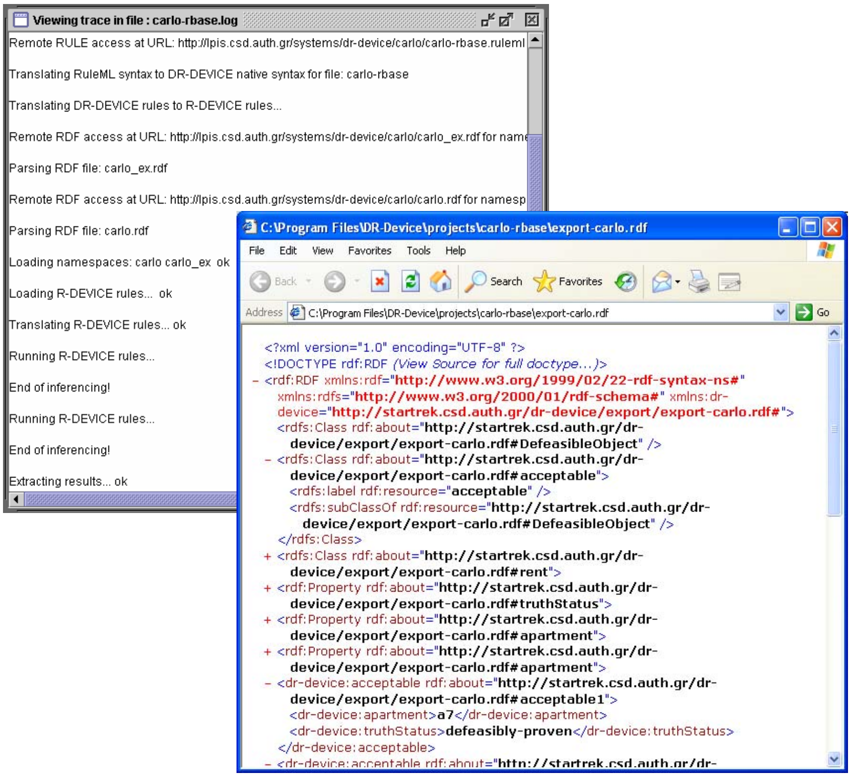
Fig. 5. The graphical rule editor and the namespace dialog window

Moving on to more node-specific features of the rule editor, one of the rule base elements that are treated in a specific manner is the `atom` element, which can be either negated or not. The response of the editor to an atom negation is performed through the wrapping/unwrapping of the `atom` element within a `neg` element and it is performed via a toggle button, located on the overhead toolbar.

Some components that also need "special treatment" are the rule IDs, each of which uniquely represents a rule within the rule base. Thus, the rule editor has to collect all of the RuleIDs inserted, in order to prohibit the user from entering the same RuleID twice and also equipping other IDREF attributes (e.g. `superior` attribute) with the list of RuleIDs, constraining the variety of possible values.

The names of the functions that appear inside a `fun_call` element are also partially constrained by the rule editor, since the user can either insert a custom-named function or a CLIPS built-in function. Through radio-buttons the user determines whether he/she is using a custom or a CLIPS function. In the latter case, a list of all built-in functions is displayed, once again constraining possible entries.

Finally, users can examine all the exported results via an Internet Explorer window, launched by VDR-Device. Also, to improve reliability, the user can also observe the execution trace of compilation and running, both during run-time and also after the whole process has been terminated (Fig. 6).


Viewing trace in file : carlo-rbase.log
C:\Program Files\DR-Device\projects\carlo-rbase\export-carlo.rdf
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Address
C:\Program Files\DR-Device\projects\carlo-rbase\export-carlo.rdf
Go

```
Remote RULE access at URL: http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-rbase.ruleml
Translating RuleML syntax to DR-DEVICE native syntax for file: carlo-rbase
Translating DR-DEVICE rules to R-DEVICE rules...
Remote RDF access at URL: http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo_ex.rdf for name
Parsing RDF file: carlo_ex.rdf
Remote RDF access at URL: http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf for namesp
Parsing RDF file: carlo.rdf
Loading namespaces: carlo carlo_ex ok
Loading R-DEVICE rules... ok
Translating R-DEVICE rules... ok
Running R-DEVICE rules...
End of inferencing!
Running R-DEVICE rules...
End of inferencing!
Extracting results... ok
```

```
  <?xml version="1.0" encoding="UTF-8" ?>
  <!DOCTYPE rdf:RDF (View Source for full doctype...)>
- <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" xmlns:dr-
    device="http://startrek.csd.auth.gr/dr-device/export/export-carlo.rdf#">
    <rdfs:Class rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#DefeasibleObject" />
  - <rdfs:Class rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#acceptable">
      <rdfs:label rdf:resource="acceptable" />
      <rdfs:subClassOf rdf:resource="http://startrek.csd.auth.gr/dr-
        device/export/export-carlo.rdf#DefeasibleObject" />
    </rdfs:Class>
  + <rdfs:Class rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#rent">
  + <rdf:Property rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#truthStatus">
  + <rdf:Property rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#apartment">
  + <rdf:Property rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#apartment">
  - <dr-device:acceptable rdf:about="http://startrek.csd.auth.gr/dr-
      device/export/export-carlo.rdf#acceptable1">
      <dr-device:apartment>a7</dr-device:apartment>
      <dr-device:truthStatus>defeasibly-proven</dr-device:truthStatus>
    </dr-device:acceptable>
  - <dr-device:acceptable rdf:about="http://startrek.csd.auth.gr/dr-
```

Fig. 6. The Trace and Results windows

Parsing RDF Schema Ontologies

As mentioned above, the RDF Schema documents contained in the namespace dialog window undergo certain processing and, more specifically, they are being parsed, using the ARP parser of Jena [19], a flexible Java API for processing RDF documents. The names of the classes found are collected in the *base class vector* (CV_b), which already contains `rdfs:Resource`, the superclass of all RDF user classes. Therefore, the CV_b vector is constructed as follows:

$$\texttt{rdfs:Resource} \in CV_b \wedge (\forall C\,(C\ \texttt{rdf:type rdfs:Class}) \rightarrow C \in CV_b)$$

where ($X\ Y\ Z$) represents an RDF triple found in the RDF Schema documents.

Except from the base class vector, there also exists the *derived class vector* (CV_d), which contains the names of the derived classes, i.e. the classes which lie at rule heads (*conclusions*). CV_d is initially empty and is dynamically extended every time a new class name appears inside the `rel` element of the `atom` in a rule head. This vector is mainly used for loosely suggesting possible values for the `rel` elements in the rule head, but not constraining them, since rule heads can either introduce new derived classes or refer to already existing ones.

The union of the above two vectors results in CV_f, which is the *full class vector* ($CV_f = CV_b \cup CV_d$) and it is used for constraining the allowed class names, when editing the contents of the `rel` element inside `atom` elements of the rule body.

Furthermore, the RDF Schema documents are also being parsed for property names and their domains. Similarly to the procedure described above, the properties detected are placed in a *base property vector* (PV_b), which already contains some built-in RDF properties (*BIP*) whose domain is `rdfs:Resource`:

BIP = {`rdf:type, rdfs:label, rdfs:comment, rdfs:seeAlso, rdfs:isDefinedBy, rdf:value`} $\subseteq PV_b$

$\forall P$, (P `rdf:type rdf:Property`) $\rightarrow P \in PV_b$

Apparently, there also exists the *derived property vector* (PV_d), which contains the names of the properties of the derived classes. This vector is initially empty and is extended each time a new property name appears inside the `_slot` element of the `atom` in a rule head. Therefore, the *full property vector* (PV_f) is a union of the above two vectors: $PV_f = PV_b \cup PV_d$.

Each of the properties in the PV_f vector has to be equipped with the corresponding superproperties and domains. Through the detected superproperties, the system can retrieve the indirect domains for each property and, thus, enrich its set of domains. The domain set of each property is needed, so that, for each `atom` element appearing inside the rule body, when a specific class C is selected, the names of the properties that can appear inside the `_slot` subelements are constrained only to those that have C as their domain, either directly or inherited.

So, the *superproperty set SUPP*(P) of each property P initially contains only the direct superproperties of P. The rest of the properties (including the derived class properties) have an empty $SUPP(P)$:

$\forall P \in PV_b\ \forall SP \in PV_b$, ($P$ `rdfs:subPropertyOf` SP) $\rightarrow SP \in SUPP(P)$

In the next step, the $SUPP(P)$ set is further populated with the indirect superproperties of each property, by recursively traversing upwards the property hierarchy:

$\forall P \in PV_b\ \forall SP \in SUPP(P)\ \forall SP' \in SUPP(SP) \rightarrow SP' \in SUPP(P)$

On the other hand, the $DOM(P)$ set of domains for each property P initially contains only the direct domain of P: $\forall P \in PV_b\ \forall C$, ($P$ `rdfs:domain` C) $\rightarrow C \in DOM(P)$

The RDF built-in properties (*BIP*) have `rdfs:Resource` as their domain:

$\forall P \in BIP$, `rdfs:Resource` $\in DOM(P)$

If a property does not have a domain, then `rdfs:Resource` is assumed:

$\forall\ P \in (PV_b\text{-}BIP)$, ($\neg\exists C\ P$ `rdfs:domain` C) $\rightarrow$ `rdfs:Resource` $\in DOM(P)$

In the next step, the $DOM(P)$ set is further populated, by inheriting the domains of all the superproperties (both direct and indirect), according to the RDFS semantics:

$\forall P \in PV_b\ \forall SP \in SUPP(P)\ \forall C \in DOM(SP) \rightarrow C \in DOM(P)$

Since the properties are now fully described (each one of them containing the corresponding superproperty and domain sets), each class C in the CV_f vector has to be linked with the allowed properties. More specifically, for each class C, five distinct sets have to be defined: *superclass set SUPC*(C), *subclass set SUBC*(C), *owned property set OWNP*(C), *inherited property set INHP*(C), and *subsumed property set SUBP*(C).

The *SUPC*(*C*) set initially contains all the direct superclasses of *C*:

$\forall C\in CV_f\, \forall SC\in CV_f,\ (C\ \texttt{rdfs:subClassOf}\ SC) \rightarrow SC \in SUPC(C)$

If a class does not have a superclass, then it is considered to be a subclass of `rdfs:Resource`. This also applies for the derived classes:

$$\forall C\in CV_f,\ C\neq\texttt{rdfs:Resource} \wedge (\neg\exists SC\ SC\in CV_f \rightarrow (C\ \texttt{rdfs:subClassOf}\ SC)) \\ \rightarrow \texttt{rdfs:Resource} \in SUPC(C)$$

In the next phase, the *SUPC*(*C*) set is further populated with the indirect superclasses of each class, by recursively traversing upwards the class hierarchy:

$\forall C\in CV_f\, \forall SC\in SUPC(C)\ \forall SC'\in SUPC(SC) \rightarrow SC' \in SUPC(C)$

The *SUBC*(*C*) set can now be easily constructed, by inversing all the subclass relationships (both direct and indirect): $\forall C\in CV_f\, \forall SC\in SUPC(C) \rightarrow C \in SUBC(SC)$

The *OWNP*(*C*) set of owned properties is constructed, by examining the domain set of each property object in the full property vector:

$\forall P\in PV_f\, \forall C\in DOM(P) \rightarrow P \in OWNP(C)$

The inherited property set *INHP*(*C*) is constructed, by inheriting the owned properties from all the superclasses (both direct and indirect), according again to the RDFS semantics: $\forall C\in CV_b\, \forall SC\in SUPC(C)\ \forall\ P\in OWNP(SC) \rightarrow P \in INHP(C)$

Finally, the subsumed property set *SUBP*(*C*) is constructed, by copying the owned properties from all the subclasses (both direct and indirect):

$\forall C\in CV_b\, \forall SC\in SUBC(C)\ \forall P\in OWNP(SC) \rightarrow P \in SUBP(C)$

Although the domain of a subsumed property of a class *C* is not compatible with class *C*, it can still be used in the rule condition for querying objects of class *C*, implying that the matched objects will belong to some subclass *C'* of class *C*, which is compatible with the domain of the subsumed property. For example, consider two classes *A* and *B*, the latter being a subclass of the former, and a property *P*, whose domain is *B*. It is allowed to query class *A*, demanding that property *P* satisfies a certain condition; however, only objects of class *B* can possibly satisfy the condition, since direct instances of class *A* do not even have property *P*.

The above mentioned three property sets comprise the *full property set FPS*(*C*):

$FPS(C) = OWNP(C) \cup INHP(C) \cup SUBP(C)$

which is used to restrict the names of properties that can appear inside a `_slot` element (see Fig. 5), when the class of the atom element is *C*.

An example of all of the above is shown in Table 1. Assume an RDF Schema ontology with three classes connected through a hierarchy: the class `apartment` is a subclass of the `house` class and a superclass of the `suburban-apartment` class. Some typical properties of these classes are displayed in the "owned properties" row.

After the RDF Schema document is parsed, these classes are detected and included in the *base class vector* (CV_b). Furthermore, the corresponding properties are determined and added to the *base property vector* (PV_b). Eventually, every available class will be linked to the respective properties, but also to the properties of its super- and subclasses, following the rationale developed before in this section. The final status of the class properties is displayed in Table 1.

This logic is reflected in the rule editor, as Fig. 5 shows. If, for example, the user wishes to formulate the rule r_4 (section 4.1), then he/she selects the `carlo:apartment` class as the value of the `href` attribute of the `_opr` element of an atom in the rule body and the allowed properties to be entered at the `_slot` element are all the properties included in Table 1. This facilitates the user, since he/she does not have to worry about which properties can be applied to apartment instances.

Table 1. Example of inherited, owned and subsumed properties

Classes	**house**	**apartment**	**suburban-apartment**
Owned Properties	size price	floor lift	gardenSize pets
Inherited Properties		size price	size, price floor, lift
Subsumed Properties	floor, lift gardenSize, pets	gardenSize pets	

5 Related Work

There exist several previous implementations of defeasible logics, although to the best of our knowledge none of them is supported by a user-friendly integrated development environment or a visual rule editor. *Deimos* [18] is a flexible, query processing system based on Haskell. It implements several variants, but neither conflicting literals nor negation as failure in the object language. Also, the current implementation does not integrate with Semantic Web, since it is solely a defeasible logic engine (for example, there is no way to treat RDF data and RDFS/OWL ontologies; nor does it use an XML-based or RDF-based syntax for syntactic interoperability). Therefore, it is only an isolated solution, although external translation modules could provide such interoperability. Finally, it is propositional and does not support variables.

Delores [18] is another implementation, which computes all conclusions from a defeasible theory. It is very efficient, exhibiting linear computational complexity. Delores only supports ambiguity blocking propositional defeasible logic; so, it does not support ambiguity propagation, nor conflicting literals, variables and negation as failure in the object language. Also, it does not integrate with other Semantic Web languages and systems, and is thus an isolated solution.

SweetJess [16] is another implementation of a defeasible reasoning system (situated courteous logic programs) based on Jess. It integrates well with RuleML. However, SweetJess rules can only express reasoning over ontologies expressed in DAMLRuleML (a DAML-OIL like syntax of RuleML) and not on arbitrary RDF data, like DR-DEVICE. Furthermore, SweetJess is restricted to simple terms (variables and atoms). This applies to DR-DEVICE to a large extent. However, the basic R-DEVICE language [8] can support a limited form of functions in the following sense: (a) path expressions are allowed in the rule condition, which can be seen as complex functions, where allowed function names are object referencing slots; (b) aggregate and sorting functions are allowed in the conclusion of aggregate rules. Finally, DR-DEVICE can also support conclusions in non-stratified rule programs due to the presence of truth-maintenance rules [7].

Mandarax [12] is a Java rule platform, which provides a rule mark-up language (compatible with RuleML) for expressing rules and facts that may refer to Java objects. It is based on derivation rules with negation-as-failure, top-down rule evaluation, and generating answers by logical term unification. RDF documents can be loaded into Mandarax as triplets. Furthermore, Mandarax is supported by the Oryx graphical rule management tool. Oryx includes a repository for managing the vocabulary, a formal-natural-language-based rule editor and a graphical user interface library. Contrasted, the rule authoring tool of DR-DEVICE lies closer to the XML nature of its rule syntax and follows a more traditional object-oriented view of the RDF data model [8]. Furthermore, DR-DEVICE supports both negation-as-failure and strong negation, and supports both deductive and defeasible logic rules.

6 Conclusions and Future Work

In this paper we argued that defeasible reasoning is useful for many applications in the Semantic Web, mainly due to conflicting rules and rule priorities. However, the development of defeasible rule bases on top of Semantic Web ontologies may appear too complex for many users. To this end, we have implemented VDR-Device, a visual environment for developing defeasible logic rule base by constraining the allowed vocabulary after analyzing the input RDF ontologies. Furthermore, the system employs a user-friendly graphical shell and a defeasible reasoning system that supports direct import from the Web and processing of RDF data and RDF Schema ontologies.

In the future, we plan to delve into the proof layer of the Semantic Web architecture by enhancing further the graphical environment with rule execution tracing, explanation, proof exchange in an XML or RDF format, proof visualization and validation, etc. We will try to visualize the semantics of defeasible logic in an intuitive manner, by providing graphical representations of rule attacks, superiorities, conflicting literals, etc. These facilities would be useful for increasing the trust of users for the Semantic Web agents and for automating proof exchange and trust among agents in the Semantic Web. Furthermore, we will include a graphical RDF ontology and data editor that will comply with the user-interface of the RuleML editor. Finally, concerning the implementation of the graphical editor we will adhere to newer XML Schema-based versions of RuleML.

References

1. Antoniou G. and Arief M., "Executable Declarative Business rules and their use in Electronic Commerce", *Proc. ACM Symposium on Applied Computing*, 2002.
2. Antoniou G., Billington D. and Maher M.J., "On the analysis of regulations using defeasible rules", *Proc. 32nd Hawaii International Conference on Systems Science*, 1999.
3. Antoniou G., Billington D., Governatori G. and Maher M.J., "Representation results for defeasible logic", *ACM Trans. on Computational Logic*, 2(2), 2001, pp. 255-287.
4. Antoniou G., Harmelen F. van, *A Semantic Web Primer*, MIT Press, 2004.
5. Antoniou G., Skylogiannis T., Bikakis A., Bassiliades N., "DR-BROKERING – A Defeasible Logic-Based System for Semantic Brokering", *IEEE Int. Conf. on E-Technology, E-Commerce and E-Service*, pp. 414-417, Hong Kong, 2005.

6. Ashri R., Payne T., Marvin D., Surridge M. and Taylor S., “Towards a Semantic Web Security Infrastructure”, *Proc. of Semantic Web Services*, 2004 Spring Symposium Series, Stanford University, California, 2004.
7. Bassiliades N., Antoniou, G., Vlahavas I., “A Defeasible Logic Reasoner for the Semantic Web”, *RuleML 2004*, Springer-Verlag, LNCS 3323, pp. 49-64, Hiroshima, Japan, 2004.
8. Bassiliades N., Vlahavas I., “R-DEVICE: A Deductive RDF Rule Language”, *RuleML 2004*, Springer-Verlag, LNCS 3323, pp. 65-80, Hiroshima, Japan, 2004.
9. Berners-Lee T., Hendler J., and Lassila O., “The Semantic Web”, *Scientific American*, 284(5), 2001, pp. 34-43.
10. Boley H., Tabet S., *The Rule Markup Initiative*, www.ruleml.org/
11. *CLIPS Basic Programming Guide* (v. 6.21), www.ghg.net/clips/CLIPS.html.
12. Dietrich J., Kozlenkov A., Schroeder M., Wagner G., "Rule-based agents for the semantic web", *Electronic Commerce Research and Applications*, 2(4), pp. 323–338, 2003.
13. Governatori G., Dumas M., Hofstede A. ter and Oaks P., “A formal approach to protocols and strategies for (legal) negotiation”, *Proc. ICAIL 2001*, pp. 168-177, 2001.
14. Governatori, G., "Representing business contracts in RuleML", *International Journal of Cooperative Information Systems*, 14 (2-3), pp. 181-216, 2005.
15. Grosof B. N., “Prioritized conflict handing for logic programs”, *Proc. of the 1997 Int. Symposium on Logic Programming*, pp. 197-211, 1997.
16. Grosof B.N., Gandhe M.D., Finin T.W., “SweetJess: Translating DAMLRuleML to JESS”, *Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML 2002).*
17. Li N., Grosof B. N. and Feigenbaum J., “Delegation Logic: A Logic-based Approach to Distributed Authorization”, *ACM Trans. on Information Systems Security*, 6(1), 2003.
18. Maher M.J., Rock A., Antoniou G., Billington D., Miller T., “Efficient Defeasible Reasoning Systems”, *Int. Journal of Tools with Artificial Intelligence*, 10(4), 2001, pp. 483-501.
19. McBride B., “Jena: Implementing the RDF Model and Syntax Specification”, *Proc. 2nd Int. Workshop on the Semantic Web*, 2001.
20. Nute D., “Defeasible Reasoning”, *Proc. 20th Int. Conference on Systems Science*, IEEE Press, 1987, pp. 470-477.
21. Skylogiannis T., Antoniou G., Bassiliades N., Governatori G., “DR-NEGOTIATE – A System for Automated Agent Negotiation with Defeasible Logic-Based Strategies”, *IEEE Int. Conf. on E-Technology, E-Commerce and E-Service*, pp. 44-49, Hong Kong, 2005.

Flavours of XChange, a Rule-Based Reactive Language for the (Semantic) Web

James Bailey[1], François Bry[2], Michael Eckert[2], and Paula-Lavinia Pătrânjan[2]

[1] NICTA Victoria Laboratory, Dept. of Computer Science and Software Engineering, University of Melbourne, 3010, Australia
http://www.cs.mu.oz.au/~jbailey/
[2] University of Munich, Germany
http://pms.ifi.lmu.de/

Abstract. This article introduces XChange, a rule-based reactive language for the (Semantic) Web. Stressing application scenarios, it first argues that high-level reactive languages are needed for both Web and Semantic Web applications. Then, it discusses technologies and paradigms relevant to high-level reactive languages for the (Semantic) Web.

1 Introduction

A common perception of the Web is that of a distributed repository of hypermedia documents, with clients (in general browsers) that download documents, and servers that store and update documents. This perception is not fully accurate: Many Web applications rely on the updating of server data in response to requests of clients, or client data in response to requests of servers. This article first argues that complementing HTTP, the Web's communication infrastructure, with high-level languages for updates and reactivity is needed for both standard Web and Semantic Web applications. It then introduces XChange, a novel high-level language for updates and reactivity on the (Semantic) Web based on Event-Condition-Action rules.

Many Web applications build upon servers that update data according to client requests or actions. This is the case of e-Commerce systems that receive, process and buy orders, of e-Learning systems that select and deliver teaching materials depending on students' test performances, and of communication platforms such as wikis, where several users modify the same documents. Conversely, some Web applications also build upon clients that update data according to server requests. This is the case with so-called *cookies,* i.e. descriptions on a client of the states of a connection to a server, or when a client keeps, after a connection to a server, data collected during the connection, e.g. a railway or airline electronic ticket.

Many Web applications also build upon complex reactions to messages or events, exchanged not only between clients and servers but also (via servers) between clients. This is the case of Web-based communication platforms (contributors are informed of other contributors joining in or leaving a session), of

Web-based business management systems (business travel applications, planning and reimbursement in large companies rely upon complex work-flows of actions and messages), of Web-based systems offering context-dependent services (e.g. a time and location dependent car park directory adapting the information it delivers and reacting to changes such as clients changing places and car parks announcing their free parking capacities), etc.

Updates and reactivity are as much "Semantic Web issues", as they are "standard Web issues". The applications mentioned above might involve both standard Web and Semantic Web data such as HTML, XML, RDF, Topic Maps, and OWL data, as well as inference from RDF triples.

Updates and reactivity on the Web are realised using HTTP/1.1; its communication paradigm is a *client-server model* of *request-response interactions.* Message *headers* give rise to the specification of network communication and system parameters. Although HTTP provides a communication infrastructure to implement updates and reactivity on the Web, more abstract and higher-level languages are needed that i) abstract away network communication and system issues, ii) ease the specification of complex updates of Web resources (e.g. XML, RDF, and OWL data), iii) are convenient for specifying complex flows of actions and reactions on the Web.

2 Technologies and Paradigms

Atomic Events, Event Messages, and Composite Events. Web applications require a number of different kinds of *atomic events:* i) events exchanged between Web nodes for a node to trigger reactions at remote nodes, ii) events local to a node, to help express local reactivity, e.g. local updates, and iii) system events for reacting to the functioning, or non-functioning, of the encompassing "system(s)". A natural assumption is that events exchanged between nodes are expressed in XML as *event messages.* Reacting to *complex events* is essential. Complex events have received considerable attention in active databases [11, 12]. However, differences between (generally centralised) active databases and the Web, where central clock and management are missing, message deliveries between Web nodes can be delayed, and user-centered (instead of system-centered) paradigms are expected, necessitate new approaches.

Temporal Dependencies often have to be expressed when composite events are specified, e.g. "depend on an event E_1 occurring *before* an event E_2", or "depend on an event E occurring within a time interval I". Sophisticated temporal notions and temporal event composition constructs are needed.

Event Messages vs. Web Resources. Event messages (volatile data)and standard Web resources (persistent data) should be kept in two separate data kinds, since otherwise the development of the "reactive" Web may be insufficiently distinguishable from the (Semantic) Web.

Event-Condition-Action Rules (*ECA rules*) fit well with the widespread and intuitive view of the Web as a distributed repository of documents. Indeed, ECA rules build on queries by the use of "conditions". Thus, ECA rules building on a

Web or Semantic Web query language are a natural paradigm for reactivity on the Web.

Distributed Processing and Communication. On the Web, reactive programs call for distributed processing. Reactive languages making each node capable of controlling its own reactive behaviour fit the decentralised management of the (Semantic) Web.

3 XChange in a Nutshell

XChange is a language of *ECA rules.* Each rule consists of three parts: i) an *"event"*-part, more precisely an event query, accessing event messages and (local) system events, ii) a Web query referred to as the *"condition"* accessing standard Web data, and iii) an *"action"* expressing iii.a) single updates, iii.b) messages to be sent to Web nodes, or iii.c) transactions i.e. a group of actions to be realised in an all-or-nothing manner. Figure 1 presents an XChange rule sending SMS notifications of delayed flights.

```
RAISE xchange:event [
        xchange:recipient [ "http://sms-gateway.org/us/206-240-1087/" ],
        text-message [ "Your flight", var N, "has been cancelled." ]
      ]
ON    xchange:event {{
        flight-cancellation {{
          flight-number{var N}, passenger{{ name {"John Public"} }}
        }}
     }}
FROM in { resource { "http://www.example.com/lufthansa.xml", "xml" },
         flights {{ flight {{ number { var N } }} }}
     }
END
```

Fig. 1. An XChange ECA rule

The *atomic events* of XChange are happenings (e.g. an update of a possibly remote Web resource) to which each Web node (through a reactive program) may or may not react. XChange has *explicit events* and *implicit events. Explicit events* are raised by a user or by an XChange program at a Web node and sent to this and/or other Web nodes as *event messages.* XChange's event messages are (arbitrary) XML documents within an *event message envelope* expressed itself as a (specific) XML document.

Figure 2 presents an XChange event message in the *term syntax* of Xcerpt [2, 4] and XChange; the (arbitrary) content is surrounded by a fixed envelope. Nesting messages with their former envelopes makes it possible to track the origin of messages, removing envelopes before forwarding messages hides their origin.

```
xchange:event [
   xchange:sender { "http://www.pms.lmu.de/" },
   xchange:recipient { "http://ruleml.org" },
   xchange:recipient { "http://www.cs.mu.oz.au/~jbailey/" },
   xchange:raising-time { "2005-06-29T18:15:00" },
   info { "Here is an article for RuleML'06!" },
   article [ title {"Flavours of XChange"}, authors [...], body [...] ]
]
```

Fig. 2. An XChange Event Message

Implicit events are local events such as updates of Web resources and system events. Events are transmitted from one Web node to another via event messages. Thus, an event sent from one Web node to another is necessarily explicit. *Composite events* are defined in XChange as *answers* to composite *event queries*, cf. [3, 7]

XChange makes a strict distinction between *persistent data,* i.e. Web resources, and *volatile data,* i.e. *by definition* events. XChange relies on the query language Xcerpt [2, 13] for accessing persistent data i.e. Web resources. XChange uses a novel query language especially tuned to events for accessing volatile data, i.e. events. This *event query language* [7] builds upon Xcerpt and extends it with constructs for *temporal event composition.* Event messages can be turned into Web resources, and Web resources might be included in event messages.

XChange's *communication model* is *peer-to-peer,* i.e. all Web nodes have the same communication capabilities and every party can initiate a communication with every other Web node. Two basic *communication strategies* are possible on a network: a *push strategy* where senders inform recipients of messages they want to send to them, and a *pull strategy* where (potential) recipients keep querying all (potential) senders for messages. Arguably, the pull strategy is convenient for querying (persistent) Web resources, while the push strategy is convenient for querying (volatile) events. XChange relies on the push strategy for event queries and on the pull strategy for Web resource queries. XChange's message communication is *asynchronous,* i.e. XChange's 'send operation' is *non-blocking:* the execution of an XChange program immediately continues after a 'send operation' without waiting for the message transmission, an acknowledgment of receipt, or a reply. Note that blocking sending can easily be implemented using XChange rules.

XChange programs are processed in a *distributed* manner. Each (XChange-aware) Web node processes, possibly by delegation to another Web node, the XChange programs locally specified. XChange relies neither on "super-peers", nor on central services, such as a central synchronisation point.

XChange ensures a *local control of events*, as well as of *event memorisation.* A Web node might reject an update request from a remote Web node (sent in an event message), e.g. because of a lack of credentials. Furthermore, the events memorised at a Web node only depend on the XChange event queries posed *at that node.* The time during which an atomic event, e.g. an event message or a local implicit event, is kept in memory at a Web node only depends on the event queries posed *at that node.* By design, XChange composite event queries can be evaluated without keeping any event forever in memory.

XChange event queries have a *declarative semantics.* XChange event query evaluation is *data-driven,* incoming events are used for *incrementally evaluating* queries. In contrast, the evaluation of queries against Web resources, e.g. Xcerpt queries and XChange conditions, is in general query-driven.

4 Related Work and Conclusion

Allen's Temporal Relations [1] and the composite events of *Active Databases* [5, 6, 8] have been important inspirations in defining XChange's temporal event composition operators. As opposed to XChange, *high-level reactive languages for the Web* formerly developed, e.g. [10], support only *simple* update operations on XML (and RDF) documents. They offer no means to specify several updates to be executed in a given order or in an *all-or-nothing* manner. Another related system is Xyleme [9], a system for monitoring and subscription on the Web with 'alerters' monitoring simple updates of Web resources and a 'monitoring query processor' for complex event detection. Its reactive functionality is highly tuned to its specific application field.

XChange significantly differs from and/or extends over the above-mentioned approaches with i) its structured event messages, ii) its distinction between Web resources and event messages, iii) its logical variables possibly shared by its event queries, conditions, and actions, iv) its declarative semantics, and v) its communication and distributed processing models.

Acknowledgments

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (`http://rewerse.net`)

References

1. J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Comm. ACM*, 26:832–843, 1983.
2. J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In *Reasoning Web*, LNCS 3564. Springer-Verlag, 2005.
3. F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proc. 20th ACM Symp. Applied Computing*, 2005.
4. F. Bry and S. Schaffert. 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.
5. A. Buchmann, A. Deutsch, and J. Zimmermann. The REACH Active OODBMS. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1995.

6. S. Chakravarthy and D. Mishra. SNOOP: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(1), 1994.
7. M. Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis, Inst. for Informatics, Univ. Munich, Germany, 2005.
8. R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In *Proc. 5th Int. Conf. on Extending Database Technology*, 1996.
9. B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML Data on the Web. In *Proc. ACM SIGMOD Conf. on the Management of Data*, 2001.
10. G. Papamarkos, A. Poulovassilis, and P. Wood. Event-Condition-Action Rules Languages for the Semantic Web. In *Proc. Workshop on Semantic Web and Databases*, 2003.
11. N. W. Paton. *Active Rules in Database Systems.* Springer-Verlag, 1999.
12. J. Widom and S. Ceri. *Active Database Systems.* Morgan Kaufmann, 1996.
13. Xcerpt. `http://xcerpt.org`.

Rule-Based Framework for Automated Negotiation: Initial Implementation

Costin Bădică[1], Adriana Bădiţă[1], Maria Ganzha[2], Alin Iordache[1], and Marcin Paprzycki[3]

[1] University of Craiova, Software Eng. Dept. Bvd. Decebal 107, Craiova, 200440, Romania
badica_costin@software.ucv.ro

[2] Elblag University of Humanities and Economy, ul Lotnicza 2, 82-300 Elblag, Poland
ganzha@op.pl

[3] Computer Science Institute, Warsaw School of Social Psychology, 03-815 Warsaw, Poland
Marcin.Paprzycki@swps.edu.pl

Abstract. The note reports on the current status of an implementation of a rule-based negotiation mechanism in a model e-commerce multi-agent system. Here, we briefly describe the conceptual architecture of the system and its initial implementation utilizing JADE and JESS. A particular negotiation scenario involving English auctions performed in parallel is also discussed.

1 Introduction

Recently, we have started developing, implementing and experimenting with a multi-agent e-commerce system (see [5] and work referenced there). One of the directions of our work is to provide agents with flexibility required for price negotiations [10] governed by mechanisms unknown in advance. In this context, rule-based approaches have been indicated as a very promising technique for parameterizing the negotiation design space ([1, 2, 4, 9, 11, 12]). Proposals have been put forward to use rules for describing either negotiation strategies ([4, 11]), mechanisms ([1]) or both ([6]), while special attention has been paid to auctions, as one of the best understood forms of negotiations ([12]). Note that when designing systems for automated negotiations one should distinguish between *negotiation protocols* (or *mechanisms*) that define ”rules of encounter” between participants and *negotiation strategies* that define behaviors aiming at achieving a desired outcome.

In this paper we discuss design and implementation of a rule-based framework for enforcing specific negotiation mechanisms inspired by work presented in [1]. We proceed as follows. In the next section we describe briefly the negotiation framework introduced in [1] and show how it fits into our e-commerce model. In section 3 we outline our design and give some details of the sample implementation using JADE ([7]) and JESS ([8]). In particular we highlight how rules are activated by the negotiation host in response to messages received from the negotiation participants. Furthermore we show how, in our implementation, the rule-based sub-agents of the negotiation host (as described in [1]) share a single JESS rule engine, rather than having separate rule engines within each sub-agent. We follow with description of two experiments: a simple experiment to highlight agent interactions and a more complex experiment with many agents and many parallel negotiations performed to asses the scalability of the implementation.

2 Conceptual Architecture

Authors of [1] sketched a complete framework for implementing portable agent negotiations. Their framework comprises: (1) negotiation infrastructure, (2) generic negotiation protocol and (3) taxonomy of declarative rules. The *negotiation infrastructure* defines roles of negotiation participants and of a host. Participants negotiate by exchanging proposals within a negotiation locale managed by the host. Depending on the negotiations type, the host can also play the participant role participant. The *generic negotiation protocol* defines the three phases of a negotiation: admission, exchange of proposals and formation of an agreement, in terms of how and when messages should be exchanged between the host and participants. *Negotiation rules* are used for enforcing the negotiation mechanism. Rules are organized into a taxonomy: rules for participants admission to negotiations, rules for checking the validity of negotiation proposals, rules for protocol enforcement, rules for updating the negotiation status and informing participants, rules for agreement formation and rules for controlling the negotiation termination.

Our goal is to create a model system in which agents perform functions typically observed in e-commerce. In this environment, e-shops and e-buyers are represented by shop and seller, and respectively client and buyer agents. Let us consider a simplified version of this scenario that involves a single shop agent S and n client agents C_i, $1 \leq i \leq n$. The shop agent is selling m products $\mathcal{P} = \{1, 2, \ldots, m\}$. We assume that each client agent C_i, $1 \leq i \leq n$, is seeking a set $\mathcal{P}_i \subseteq \mathcal{P}$ of products (we therefore restrict our attention to the case where all sought products are available through shop agent S). Shop agent S is using m seller agents S_j, $1 \leq j \leq m$ and each seller agent S_j is responsible for selling single product j. Each client agent C_i is using buyer agents B_{ik} to purchase products in set $\mathcal{P}_i$. Each buyer agent B_{ik} is responsible with negotiating and buying exactly one product $k \in \mathcal{P}_i$, $1 \leq i \leq n$. To attempt purchase buyer agents B_{ik} migrate to the shop agent S and engage in negotiations; a buyer agent B_{ik}, that was spawned by client agent C_i, will engage in negotiation with seller S_k, to purchase product k. This simple scenario is sufficient for the purpose of our paper, i.e. to illustrate how a number of rule-based automated negotiations can be performed concurrently. In this setting, each seller agent S_j plays the role of a negotiation host defined in [1]. Therefore, in our system, we have exactly m instances of the framework described in [1]. Each instance is managing a separate negotiation "locale", while all instances are linked to the shop agent S. For each instance we shall have one separate set of rules that describes the negotiation mechanism implemented by that host (seller agent). See figure 1a for an example.

3 Design and Implementation

Let us now discuss: (i) how the negotiation host is structured into sub-agents; (ii) how rules are executed by the host in response to messages received from participants and how rule firing control is switched between sub-agents; (iii) how the generic negotiation protocol was implemented using JADE agent behaviors and ACL message exchanges.

The Negotiation Host. Host and negotiation participant agents are ordinary JADE agents. The host agent encapsulates a number of sub-agents that are implemented as ordinary Java classes: *Gatekeeper*, *Proposal Validator*, *Protocol Enforcer*, *Information*

Updater, *Negotiation Terminator* and *Agreement Maker*. Each sub-agent has a *handle()* method that is activated whenever the sub-agent must react to check the category of rules it is responsible for. In addition to sub-agents, the host encapsulates two objects: the *Negotiation Locale* stores the *negotiation template* (a structure that defines negotiation parameters; see [1]) and the list of negotiation participants; the *Blackboard* object is a JESS rule engine (class *jess.Rete*) that is initialized with negotiation rules. Whenever category of negotiation rules is checked, the rule engine is activated. The host contains handler methods that are activated by *action()* methods of the agent behaviors. Each handler method delegates the call to the responsible sub-agent. Finally, the sub-agent activates the rule engine via a member object that points to the parent host agent.

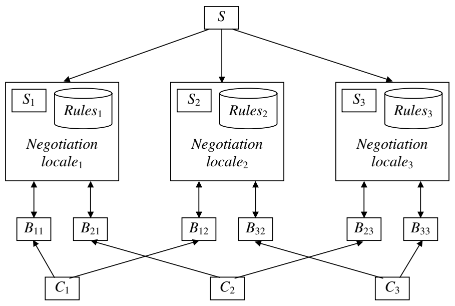
Controlling Rule Execution. Rather then implementing each sub-agent of the negotiation host as a separate rule engine [1], we use a single JESS engine shared by all sub-agents. Rules and facts managed by the rule engine are partitioned into JESS modules. *Blackboard facts* are instances of JESS *deftemplate* statements and they represent: the negotiation template; the active proposal that was validated by the *Proposal Validator* and the *Proposal Enforcer* sub-agents; seller reservation price (not visible to participants); negotiation participants; the negotiation agreement that is eventually generated at the end of a negotiation; the information digest that is visible to the negotiation participants; the maximum time interval for submitting a new bid before the negotiation is declared complete; the value of the current highest bid. Each category of rules for mechanism enforcement is stored in a separate JESS module. This module is controlled by the corresponding sub-agent of the negotiation host. Whenever the sub-agent handles a message it activates the rules for enforcing the negotiation mechanism. Taking into account that all rules are stored internally in a single JESS rule-base (attached to a single JESS rule engine), the JESS *focus* statement is used to control the firing of rules located only in the focused module. This way, the JESS facility for partitioning the rule-base into distinct JESS modules proves very useful for controlling separate activation of each category of rules. Note that JADE agent behaviors are scheduled for execution in a non-preemptive way and this implies that firings of rule categories are correctly serialized and thus they do not cause any synchronization problems.

Generic Negotiation Protocol and Agent Behaviors. The negotiation process has three phases: (1) admission, (2) proposal submission and (3) agreement formation. Tasks of sending and receiving messages according to the constraints stated by the negotiation protocol are implemented using JADE agent behaviors.

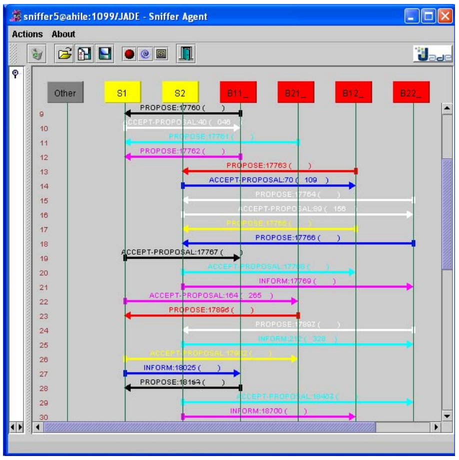
The *admission* phase starts when a new participant requests admission by sending a PROPOSE message to the host. The host grants (or not) the admission of the participant to the negotiation and responds with either an ACCEPT-PROPSAL or a REJECT-PROPOSAL message. Currently, the PROPOSE message is sent by the participant immediately after its initialization stage, just before its *setup()* method returns. The task of receiving the admission proposal and issuing an appropriate response is implemented as a separate host behavior. When a participant is admitted, it receives from the host a template representing auctions parameters: auction type, auctioned product, minimum bid increment termination time window, currently highest bid.

A participant enters the phase of *submitting proposals* immediately after it was admitted (participants join negotiation dynamically). This event is signaled by the reception of an ACCEPT-PROPOSAL message together with the negotiation template containing currently highest bid. As soon as they obtain the negotiation template and currently highest bid agent sends its first bid. The negotiation protocol states also that a participant will be notified by the host if its proposal was accepted (ACCEPT-PROPOSAL) or rejected (REJECT-PROPOSAL). When a proposal is accepted, the protocol requires that all other participants are notified accordingly with INFORM messages. Strategies of participant agents must be defined in accordance with the generic negotiation protocol. The strategy defines when a negotiation participant submits a proposal and what are proposal parameters. For the time being we utilize a simplistic solution: participant submits first bid immediately after it was admitted and subsequently, whenever it receives notification that another proposal was accepted by the host. Each time the value of the bid is equal to that of the currently highest bid plus an increment (that is private to the participant). Additionally, each participant has its own reservation price and if the value of the new bid exceeds it then the proposal submission is canceled.

Finally, the *agreement formation* phase can be triggered at any time. When the agreement formation rules signal that an agreement was reached, the protocol states that all the participants involved will be notified by the host with INFORM messages. The agreement formation check is implemented as a timer task (class *java.util.TimerTask*) that is executed in the background thread of a *java.util.Timer* object.



a.Conceptual architecture of the system



```
9   PROPOSE:17760 (     )
10  ACCEPT-PROPOSAL:40 (  046  )
11  PROPOSE:17761 (     )
12  PROPOSE:17762 (     )
13  PROPOSE:17763 (     )
14  ACCEPT-PROPOSAL:70 (  109  )
15  PROPOSE:17764 (     )
16  ACCEPT-PROPOSAL:89 (  156  )
17  PROPOSE:17765 (     )
18  PROPOSE:17766 (     )
19  ACCEPT-PROPOSAL:17767 (     )
20  ACCEPT-PROPOSAL:17768 (     )
21  INFORM:17769 (     )
22  ACCEPT-PROPOSAL:164 (  265  )
23  PROPOSE:17896 (     )
24  PROPOSE:17897 (     )
25  INFORM:212 (  328  )
26  ACCEPT-PROPOSAL:17902 (     )
27  INFORM:18025 (     )
28  PROPOSE:18163 (     )
29  ACCEPT-PROPOSAL:18402 (     )
30  INFORM:18700 (     )
```

b.First part of the negotiation

Fig. 1.

4 Experiments

In the first experiment we consider that shop is selling 2 products, both products have a reservation price of 50 and require a minimum bid increment of 5. There are 2 clients C_1 and C_2, each seeking both products. Client C_1 has a reservation price of 52 for product 1, a reservation price of 61 for product 2 and a bid increment of 9. Client C_2 has a reservation price of 54 for product 1, a reservation price of 63 for product 2 and a bid increment of 11. Client C_1 is using buyers B_{11} and B_{12}, and similarly client C_2 is

Table 1. Explanation of message exchanged during negotiation in experiment 1

B_{11} 52 9	B_{21} 54 11	B_{12} 61 9	B_{22} 63 11
request admission	request admission	request admission	request admission
admission granted 0	admission granted 9	admission granted 0	admission granted 0
bid 9	bid 20	bid 9	bid 11
accept bid 9	accept bid 20	accept bid 9	inform 9
inform 20	inform 29	inform 20	bid 20
bid 29	bid 40	bid 29	reject bid 11
accept bid 29	accept bid 40	accept bid 29	accept bid 20
inform 40	inform 49	inform 40	inform 29
bid 49		bid 49	accept bid 40
accept bid 49		inform 60	inform 49
			bid 60
			accept bid 60
			win 60

using buyers B_{21} and B_{22}. Some of the messages exchanged between agents in this experiment are shown in figure 1b (note that only sellers and buyers are shown on that figure (clients are not shown, as they only play the role of creating buyers and sending them to negotiations). While Figure 1b show messages exchanged between agents during negotiation, their content is not visible. Therefore we provide an explanation of message exchanges in Table 1. The table header contains buyer names together with their reservation prices and bid increments.

There are some interesting facts to note in table 1. First, when buyer B_{21} is granted admission to the negotiation, buyer B_{11} had already submitted a bid and that bid was accepted. Therefore B_{21} will get a value of 9 in the negotiation template for the currently highest bid; note that this is an example of a participant that dynamically joins negotiation in progress. Second, the negotiation between S_1 and agents B_{11} and B_{21} ended without a winner. The highest accepted bid was 49 from B_{11} but this value is lower than the reservation price 50 of S_1. According to their strategies, none of the participants B_{11} and B_{21} is able to issue a higher bid that is still lower than their own reservation prices. Third, negotiation between S_2 and agents B_{21} and B_{22} ended with agent B_{22} becoming a winner and the highest bid 60. Finally, note that bid 11 of buyer B_{22} was rejected because at the time this bid was submitted there was already a highest bid of 9 accepted, and thus, the rule saying that the minimum value of the bid increment is 5 was violated. However, by the time B_{22} submitted its bid, it wasn't aware that the other participant B_{12} also posted a bid and got it accepted.

In the second experiment we considered 10 products and 12 clients seeking all of them. The auction parameters were the same for all auctions: reservation price 50 and minimum bid increment 5. Clients reservation prices were randomly selected from the interval [50,72] and their bid increments were randomly selected from the interval [7,17]. In the experiment 143 agents were created: 1 shop, 10 sellers, 12 clients and 120 buyers and 10 English auctions were run in parallel. The average number of messages exchanged per negotiation was about 100 and all the auctions finished successfully. While the total number of agents is still small (as compared to [3]) this experiment indicates that the proposed approach has good potential for scalability.

5 Concluding Remarks

In this note we have discussed a multi-agent system that utilizes a rule-based approach to implement flexible automated negotiations. This system is being implemented using JADE and JESS and its simplified version works for the case of English auctions. As future work we plan to: i) complete the integration of the rule-based framework into our e-commerce model; ii) asses the generality of our implementation by extending it to include other price negotiations; iii) allow the logical specification of the rules in order to asses their correctness; iv) investigate the effectiveness of describing and/or publishing negotiation rules using rule markup languages. We will report on our progress in subsequent papers.

Acknowledgement

We would like to thank authors of [1] for providing us with their sample implementation. This was a valuable input for producing the implementation and experimental results described here.

References

1. Bartolini, C., Preist, C., Jennings, N.R.: A Software Framework for Automated Negotiation. In: *Proc. of SELMAS'2004*, LNCS 3390, Springer Verlag (2005) 213–235.
2. Benyoucef, M., Alj, H., Levy, K., Keller, R.K.: A Rule-Driven Approach for Defining the Behaviour of Negotiating Software Agents. In: J.Plaice et al. (eds.): *Proc. of DCW'2002*, LNCS 2468, Springer verlag (2002) 165–181.
3. Chmiel, K., Tomiak, D., Gawinecki, M., Karczmarek, P., Szymczak, Paprzycki, M.: Testing the Efficiency of JADE Agent Platform. In: *Proc. of the* 3^{rd} *International Symposium on Parallel and Distributed Computing*, Cork, Ireland. IEEE Computer Society Press, Los Alamitos, CA, USA, (2004), 49–57.
4. Dumas, M., Governatori, G., ter Hofstede, A.H.M., Oaks, P. (2002): A Formal Approach to Negotiating Agents Development. In: *Electronic Commerce Research and Applications*, Vol.1, Issue 2 Summer, Elsevier Science, (2002) 193–207.
5. Ganzha, M., Paprzycki, M., Pîrvănescu, A., Bădică, C, Abraham, A.: JADE-based Multi-Agent E-commerce Environment: Initial Implementation, In: *Analele Universităţii din Timişoara, Seria Matematică-Informatică* (2005) (to appear)
6. Governatori, G., Dumas, M., ter Hofstede, A.H.M., and Oaks, P.: A formal approach to protocols and strategies for (legal) negotiation. In: Henry Prakken, editor, *Proc. of the 8th International Conference on Artificial Intelligence and Law*, IAAIL, ACM Press, (2001) 168–177.
7. JADE: Java Agent Development Framework. See `http://jade.cselt.it`.
8. JESS: Java Expert System Shell. See `http://herzberg.ca.sandia.gov/jess/`.
9. Lochner, K.M., Wellman, M.P.: Rule-Based Specification of Auction Mechanisms. In: *Proc. AAMAS'04*, ACM Press, New York, USA, (2004).
10. Lomuscio, A.R., Wooldridge, M., Jennings, N.R.: A classification scheme for negotiation in electronic commerce. In: F. Dignum, C. Sierra (Eds.): *Agent Mediated Electronic Commerce: The European AgentLink Perspective*, LNCS 1991, Springer Verlag (2002) 19–33.
11. Skylogiannis, T., Antoniou, G., Bassiliades, N.: A System for Automated Agent Negotiation with Defeasible Logic-Based Strategies - Preliminary Report. In: Boley, H., Antoniou, G. (eds): *Proc. of RuleML'04*, Hiroshima, Japan. LNCS 3323 Springer-Verlag (2004) 205–213.
12. Wurman, P.R., Wellman, M.P., Walsh, W.E.: A Parameterization of the Auction Design Space. In: *Games and Economic Behavior*, 35, Vol. 1/2 (2001), 271–303.

Uncertainty and RuleML Rulebases: A Preliminary Report

Giorgos Stoilos[1], Giorgos Stamou[1], Vassilis Tzouvaras[1], and Jeff Z. Pan[2]

[1] Department of Electrical and Computer Engineering, National Technical University of Athens, Zographou 15780, Greece
[2] School of Computer Science, The University of Manchester Manchester, M13 9PL, UK

Abstract. Uncertainty, like imprecision and vagueness, has gained considerable attention the last decade. To this extend we present a preliminary report on extending the Rule Markup Language (RuleML) with fuzzy set theory, in order to be able to represent and handle vague knowledge. We also provide semantics for the case of fuzzy FOL RuleML.

1 Introduction

According to widely known proposals for a Semantic Web architecture, ontologies will play a key role in the Semantic Web This has led to considerable efforts to developing a suitable ontology language, culminating in the design of the OWL Web Ontology Language [1], which is now a W3C recommendation. Although OWL adds considerable expressive power with respect to languages such as RDF, it does have expressive limitations, particularly with respect to what can be said about properties, as well as the total absence of rules, which are valuable in many real life applications. To this end the extension of the current semantic web architecture with some form of rules language has lead to several proposals, like SWRL [2], FOL RuleML [3], and many more.

Even though the combination of OWL and rules results in the creation of a highly expressive language, there are still many occasions where this language fails to accurately represent knowledge of our world. In particular these languages fail at representing vague and imprecise knowledge and information. Uncertainty, is both a characteristic of information itself, like the concepts of a "tall" preson, a "happy" person, a "fast" car, a "hot" place, a "faulty" part, and many more. Experience with these domains has shown that in many cases dealing with such type of information yields more realistic applications, which derive better results. The need for covering uncertainty in the Semantic Web context has been stressed out in literature many times the last years; some examples are [4–6].

In order to capture imprecision in rules, we propose a fuzzy extension of the RuleML framework, called f-RuleML. In f-RuleML, facts about the world can include a specification of the "degree" (a truth value between 0 and 1) of confidence with which one can assert that a tuple of individuals is an instance of a given relation.

2 Preliminaries

Fuzzy set theory and fuzzy logic constitute a widely used framework for the representation and management of various forms of uncertainty, like vagueness and imprecision, introduced in real-life applications. They are based on the notion of fuzzy sets, introduced in [7]. While in terms of classical set theory any element belongs or not to a set, in fuzzy set theory this is a matter of degree. More formally, let X be a collection of elements with cardinality m, i.e $X = \{x_1, x_2, \ldots, x_m\}$. A crisp subset S of X is any collection of elements of X that can be defined with the aid of its *characteristic function* $\chi_A(x)$ that assigns any $x \in X$ to a value 1 or 0 if this element belongs to X or not, respectively. On the other hand, a fuzzy subset A of X, is defined by a membership function $\mu_A(x)$, or simply $A(x)$, $x \in X$. This membership function assigns any $x \in X$ to a value between 0 and 1 that represents the degree in which this element belongs to X.

The classical set theoretic operation of complement, union, intersection and the logical operation of implication are also extended to this new framework and are performed by special mathematical functions over the unit interval called fuzzy complement (c), fuzzy intersection or t-norm (t or $*$), fuzzy union or t-conorm (u) and fuzzy implication ($\Rightarrow$), respectively [8]. Among these operations fuzzy implications play an important role as they determine the properties of the resulting fuzzy logic [9]. In our context we will consider only the class of R-implications [8], cause of their interesting properties. These implications are given by the equation: $x \Rightarrow y = \sup\{z \in [0,1] \mid t(x,z) \leq y\}$, and they have the nice property that $x \Rightarrow y = 1$ if and only if $x \leq y$.

3 Dealing with Uncertainty in RuleML

In the current section we will use a motivating use case to present the syntactic changes that need to be applied to the RuleML framework.

Consider a casting company, which has a knowledge base that consists of models. Advertisement companies are using this knowledge base to look for models to be used in tv commercials. Each entry in the knowledge base contains a photo of the model, personal information and some body and face characteristics. The casting company has created a user interface for inserting the information of the models as instances of a predefined ontology. It also provides a query engine to search for models with specific characteristics, which in the case of advertisement companies usually are complex characteristics, like the hair quality, color, body fitness, skin quality, etc, in order to determine if they qualify for a certain commercial. Obviously, such a knowledge base contains a wealth of vague concepts, like long hair, brown eyes, athletic body, and many more. In such a case one might want to specify the membership degree of an individual, say "SUSAN" to a fuzzy concept like *brown_eyes*, as $brown\_eyes(SUSAN) \geq 0.7$, to indicate the least degree that "SUSAN" participates to the fuzzy concept *brown_eyes*. We can make these assertions explicit to the system by encoding them as RuleML *fuzzy facts*. In that case we can write,

```
<Atom>
  <degree><Data>0.8</Data></degree>
  <_opr><Rel>brown_eyes</Rel></_opr>
  <Ind>SUSAN</Ind>
</Atom>
```

From the above examples we can see that the syntactic changes that need to take place are minimal and only involve the syntax of fuzzy facts. So the additional change that needs to take place in the XML Schema definition of the element `Atom` [10] is the following:

```
<xs:group name="Atom.content">
...................
  <xs:choice>
    <xs:sequence>
      <xs:element name="degree" type="degree.type" minOccurs="0"
maxOccurs="1"/>
    <xs:choice>
      <xs:element ref="opr"/>
....................
</xs:group>
```

The XML Schema definition of the new tag `degree` could be given by the following schema definition:

```
<xs:attributeGroup name="degree.attlist"/>
  <xs:group name="degree.content">
    <xs:sequence>
      <xs:element ref="Data"/>
    </xs:sequence>
  </xs:group>
  <xs:complexType name="degree.type">
    <xs:group ref="degree.content"/>
    <xs:attributeGroup ref="degree.attlist"/>
  </xs:complexType>
<xs:element name="degree" type="degree.type"/>
```

Subsequently, one can use such fuzzy facts to encode knowledge in the form of fuzzy rules without any further syntax modification.

4 Fuzzy FOL RuleML

In the current section we will present the syntax and semantics of Fuzzy FOL RuleML (f-FOL RuleML). Our presentation follows the one in [11].

Definition 1. *[11] A* predicate language *consists of a non-empty set of* predicates*, each together witha positive natural number (the arity), and a (possibly empty) set of* object constants*. Predicates are mostly denoted by $P, Q, R, ...$, constants by $c, d, ...$.* Logical Symbols *are* object variables $x, y, ...$, connectives $\&, \rightarrow$, truth constants $\bar{r}$ *for each rational* $r \in [0, 1]$ *and* quantifiers $\exists, \forall$. *Other connectives are defined as follows:*

$$\phi \wedge \psi = \psi \& (\phi \to \psi), \quad \phi \equiv \psi = (\phi \to \psi)\&(\psi \to \phi)$$
$$\neg\phi = \phi \to \bar{0}, \qquad \phi \vee \psi = ((\phi \to \psi) \to \psi) \wedge ((\psi \to \phi) \to \phi)$$

Atomic formulas *have the form $P(t_1, ..., t_n)$, where P is a predicate of arity n and $t_1, ..., t_n$ are terms. If ϕ, ψ are formulas and x is an object variable then $\phi \to \psi, \phi\&\psi, (\exists x)\phi, (\forall x)\phi, \bar{r}$ are formulas.*

Definition 2. *[11] Let $\mathcal{J}$ be a predicate language and let $\boldsymbol{L}$ be a linearly ordered BL-algebra. An $\boldsymbol{L}$*-structure *$\boldsymbol{M} = \langle M, (r_P)_P, (m_c)_c \rangle$ for $\mathcal{J}$ has a non-empty* domain *M, for each n-ary predicate P a $\boldsymbol{L}$-fuzzy n-ary relation $r_P : M^n \to \boldsymbol{L}$ on M, associating to each n-tuple of elements of M the degree $r_P(m_1, ..., m_n) \in \boldsymbol{L}$ of the membership of $(m_1, ..., m_n)$ to the fuzzy relation, and for each object constant c, m_c is an element of M.*

Definition 3. *[11] Let $\mathcal{J}$ be a predicate language and $\boldsymbol{M}$ an $\boldsymbol{L}$-structure for $\mathcal{J}$. An $\boldsymbol{M}$*-evaluation *of object variables is a mapping u assigning to each object variable x an elements $u(x) \in M$. Let u, u' be two evaluations. $u \equiv_x u'$ means that $u(y) = u'(y)$ for each variable y distinct from x. The value of a term given by $\boldsymbol{M}$, u is defined as follows: $\|x\|_{\boldsymbol{M},u} = u(x)$, $\|c\|_{\boldsymbol{M},u} = m_c$. We define the* truth value *$\|\phi\|^{\boldsymbol{L}}_{\boldsymbol{M},u}$ of a formula. as follows:*

$$\begin{aligned} \|P(t_1, ..., t_n)\|^{\boldsymbol{L}}_{\boldsymbol{M},u} &= r_P(\|t_1\|_{\boldsymbol{M},u}, ..., \|t_n\|_{\boldsymbol{M},u}), & \|\bar{r}\|^{\boldsymbol{L}}_{\boldsymbol{M},u} &= r \\ \|\phi\&\psi\|^{\boldsymbol{L}}_{\boldsymbol{M},u} &= \|\phi\|^{\boldsymbol{L}}_{\boldsymbol{M},u} * \|\psi\|^{\boldsymbol{L}}_{\boldsymbol{M},u}, & \|\phi \to \psi\|^{\boldsymbol{L}}_{\boldsymbol{M},u} &= \|\phi\|^{\boldsymbol{L}}_{\boldsymbol{M},u} \Rightarrow \|\psi\|^{\boldsymbol{L}}_{\boldsymbol{M},u} \\ \|(\exists x)\phi\|^{\boldsymbol{L}}_{\boldsymbol{M},u} &= \sup\{\|\phi\|^{\boldsymbol{L}}_{\boldsymbol{M},u'} | u \equiv_x u'\} & \|(\forall x)\phi\|^{\boldsymbol{L}}_{\boldsymbol{M},u} &= \inf\{\|\phi\|^{\boldsymbol{L}}_{\boldsymbol{M},u'} | u \equiv_x u'\} \end{aligned}$$

At last a rule is interpreted as: $\|\phi \to \psi\|_{\boldsymbol{M},u} = 1$

Observe that since fuzzy rules are in general not equivalent fuzzy implications, as it is also the case in classical rules, we have interpreted rules as fuzzy implications which are 1-tautologies, i.e. their truth value is 1, in each safe **L**-structure **M** and each **M**-valuation of object variables. Since in our case we use R-implications the above equation further means that $\|\phi\|^{\mathbf{L}}_{\mathbf{M},u} \leq \|\psi\|^{\mathbf{L}}_{\mathbf{M},u}$.

Logics like the one presented above is often referred to as *Rational Pavelka predicate logic* ($RPL\forall$). The reader is referred to [11] for more information on these logics.

5 Discussion

Recently, a lot of interest towards dealing with uncertainty, like imprecision and vagueness (fuzzyness), in the semantic web context has been demonstrated. It is well established in the AI community that dealing with such type of information yields more intelligent and realistic applications. Since rules will also play an important role in the realization of the semantic web and its wider acceptance by the industry community, the need for dealing with uncertainty in rule systems is evident. The idea of adding fuzzyness in logic programs is not new. Several approaches to fuzzy logic programming have been presented [12–15]. Recently the interest has been extended to the semantic web where both fuzzy description logics [16] and fuzzy rules, are integrated providing a fuzzy extension to the SWRL language [17]. In the current paper we have extended the RuleML framework in order to make it capable to represent and handle imprecise and vague information. We showed that the syntactic changes that need to take place are minimal and only regard facts.

References

1. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., eds., L.A.S.: OWL Web Ontology Language Reference (2004)
2. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language — Combining OWL and RuleML. W3C Member Submission, `http://www.w3.org/Submission/SWRL/` (2004)
3. Boley, H., Dean, M., Grosof, B., Sintek, M., Spencer, B., Tabet, S., Wagner, G.: FOL RuleML: The First-Order Logic Web Language. W3C Member Submission, `http://www.w3.org/Submission/FOL-RuleML/` (2005)
4. Bechhofer, S., Goble, C.: Description Logics and Multimedia - Applying Lessons Learnt from the GALEN Project. In: KRIMS 96 Workshop on Knowledge Representation for Interactive Multimedia Systems, ECAI 96, Budapest (1996)
5. Stoutenburg, S., Obrst, L., Nichols, D., Peterson, J., Johnson, A.: Toward a standard rule language for semantic integration in the dod enterprise, W3C Workshop on Rule Languages for Interoperability (2005)
6. Chen, H., Fellah, S., Bishr, Y.: Rules for geospatial semantic web applications, W3C Workshop on Rule Languages for Interoperability (2005)
7. Zadeh, L.A.: Fuzzy sets. Information and Control **8** (1965) 338–353
8. Klir, G.J., Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice-Hall (1995)
9. Klement, E., Navara, M.: A survey of different triangular norm-based fuzzy logics. Fuzzy Sets and Systems **101** (1999) 241–251
10. Hirtle, D., Boley, H., Grosof, B., Kifer, M., Sintek, M., Tabet, S., Wagner, G.: Schema Specification of RuleML 0.89. `http://www.ruleml.org/0.89/` (2005)
11. Hajek, P.: Metamathematics of fuzzy logic. Kluwer (1998)
12. Vojtás, P.: Fuzzy logic programming. Fuzzy Sets and Systems **124** (2001) 361–370
13. Ebrahim, R.: Fuzzy logic programming. Fuzzy Sets and Systems **117** (2001) 215–230
14. Cao, T.: Annotated fuzzy logic programs. Fuzzy Sets and Systems **113** (2000) 277–298
15. Damásio, C.V., Pereira, L.M.: Monotonic and residuated logic programs. In: Proceedings of the 6th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, Springer-Verlag (2001) 748–759
16. Stoilos, G., Stamou, G., Tzouvaras, V., Pan, J., Horrocks, I.: A fuzzy description logic for multimedia knowledge representation, Proc. of the International Workshop on Multimedia and the Semantic Web (2005)
17. Pan, J.Z., Stamou, G., Tzouvaras, V., Horrocks, I.: f-SWRL: A Fuzzy Extension of SWRL. In: Proc. of the International Conference on Artificial Neural Networks, Special section on "Intelligent multimedia and semantics". (2005) To appear.

Nested Rules in Defeasible Logic

Insu Song and Guido Governatori

School of Information Technology & Electrical Engineering
The University of Queensland, Brisbane, QLD, 4072, Australia
{insu,guido}@itee.uq.edu.au

Abstract. Defeasible Logic is a rule-based non-monotonic logic with tractable reasoning services. In this paper we extend Defeasible Logic with nested rules. We consider a new Defeasible Logic, called DL^{ns}, where we allow one level of nested rules. A nested rule is a rule where the antecedent or the consequent of the rule are rules themselves. The inference conditions for DL^{ns} are based on reflection on the inference structures (rules) of the particular theory at hand. Accordingly DL^{ns} can be considered an amalgamated reflective system with implicit reflection mechanism. Finally we outline some possible applications of the logic.

1 Introduction

Nested rules arise naturally in our daily reasoning activities and in many applications: from artificial societies and normative reasoning, to configuration systems to security. Every time we have policies that are represented as sets of rules we have to consider the possibility that a policy contains rules about itself.

For instance, we often make decisions or classify objects based on some consequence relations. For example, in security, the usual definition of confidentiality is that a piece of information is regarded as confidential for an organization when the release of it would harm the interest of the organization. This can be formally written as:

$$(\mathit{Disclosed}(x) \Rightarrow \mathit{HarmInterest}) \Rightarrow \mathit{Confidential}(x).$$

In addition the security policy can give conditions (sometimes explicitly, sometime implicitly) about when the disclosure of a piece of information will harm the interest of the organization. In similar way many normative concepts frequently used in contracts, such as for example, delegation, empowerment, require definitions based on nested rules (see, for example, [1]).

In other cases, we often have rule dependencies that rule $r2$ is placed in a system only when rule $r1$ holds in the system:

$$r1 \rightarrow r2.$$

These dependencies are usually stored outside of the system. If rule dependencies can be expressed directly in the system, then $r2$ can be removed automatically whenever $r1$ does not hold in the system providing automatic system maintenance functionality. This feature is also useful in system integration because it allows context dependant rules.

One major problem for adding nested rule expressions to any knowledge representation system is defining a proper evaluation of the nested rules. Evaluating the nested conditionals based on the material conditional fails miserably. The paradoxes associated with the material conditional are well known. For example, $(Disclosed(x) \supset HarmInterest)$ is logically equivalent to $(\neg Disclosed(x) \vee HarmInterest)$ so that the following statements are logically true:

1. If x is not disclosed x is confidential.
2. If interest is harmed by any reason, x is confidential.

In this paper we present DL^{ns}, which is a defeasible logic (DL) with nested rules and rule provability (see [2] for an introduction to defeasible logic). DL^{ns} allows one level of nesting of rules both in the antecedent and the consequent of non-monotonic statements.

The next section presents the proof theory of DL^{ns}. Then we show an example illustrating the use of nested rules. We conclude the paper with some remarks.

2 DL^{ns}: DL with Singly Nested Rules

In this section we outline a defeasible logic with singly nested rules (DL^{ns}) which admits one level of nesting of rules. A *nested rule* is a rule in the antecedent or the consequent of another rule. For example, $(a \rightarrow b) \rightarrow (c \rightarrow d)$ has two nested rules: $(a \rightarrow b)$ in the antecedent and $(c \rightarrow d)$ in the consequent. Rules in a DL^{ns} theory can contain nested rules, but nested rules cannot contain nested rules.

As in a standard defeasible logic (DL), a DL^{ns} theory is a triple $(F, R, \succ)$ where F is a set of literals (called facts), R is a finite set of rules, $\succ$ is a superiority relation on R. For the definitions of *literal* and *superiority relation* $\succ$, refer to [2]. A *relation* $r : (A(r), C(r))$ consists of its unique label r, its antecedent $A(r)$ which is a finite set of literals and nested rules, and its consequent $C(r)$ which is either a literal or a nested rule. A relation just says that $C(r)$ depends on $A(r)$. A *rule* $r_{\hookrightarrow}$ (i.e., $A(r) \hookrightarrow C(r)$) is a relation r with a rule type $\hookrightarrow$ specified. Replacing the placeholder $\hookrightarrow$ with the three rule types defined in DL yields three kinds of rules: $r_{\rightarrow}$ is a strict rule in the form of $A(r) \rightarrow C(r)$; $r_{\Rightarrow}$ is a defeasible rule in the form of $A(r) \Rightarrow C(r)$; $r_{\leadsto}$ is a defeater rule in the form of $A(r) \leadsto C(r)$. For example, a rule $(p \Rightarrow q)$ consists of its antecedent $A(r) = \{p\}$, its consequent $C(r) = q$, and its rule type $\Rightarrow$. A literal l is a strict rule with the antecedent the empty set and the consequent the literal itself: $\{\} \rightarrow l$. Given a rule $r_{\hookrightarrow}$, the *negation* of $r_{\hookrightarrow}$, $(N(r_{\hookrightarrow}))$ is the rule $(A(r) \Rightarrow {\sim}C(r))$. In Section 2.1 we will provide an intuition for this definition.

In DL, each type of rules represents a different strength of dependency between antecedents and consequents. We define *rule strength order* by which rules with the same relation can be ordered as follows:

$$r_{\rightarrow} > r_{\Rightarrow} > r_{\leadsto}$$

where $r_{\rightarrow}$ is a strict rule, $r_{\Rightarrow}$ is a defeasible rule, $r_{\Rightarrow}$ is a defeater, and r is a relation. The rules with the same rule types and relations have the same rule strength. Thus, rules with the same relation can be compared for their strength. For example, the following statements are true: $r_{\rightarrow} \geq r_{\Rightarrow}$, $r_{\Rightarrow} \geq r_{\Rightarrow}$, $r_{\rightarrow} \geq r_{\leadsto}$. Since a literal l is a strict rule $\{\} \rightarrow l$, the following statements are true as well: $l = (\{\} \rightarrow l)$, $l > (\{\} \Rightarrow l)$, $l > (\{\} \leadsto l)$.

The *final consequent* of a relation r is the right most consequent. For example, the final consequent of $a \rightarrow (b \rightarrow c)$ is c. If r is a literal, the final consequent of r is r itself. $A(r)_q$ is the union of antecedents in r for the consequent q. That is, $A(r)_q$ is the set of premises that need to be satisfied to conclude q. $A(r)_q$ is formally defined as follows:

$$A(r)_q = \begin{cases} A(r) & \text{If } C(r) = q \\ A(r) \cup A(C(r)) & \text{If } C(C(r)) = q \end{cases}$$

For example, given $r = a \rightarrow (d \rightarrow e)$, we have $A(r)_{(d \rightarrow e)} = \{a\}$ and $A(r)_e = \{a, d\}$.

Given a set R of rules, we denote the set of all strict rules in R by R_s and the set of strict rules and defeasible rules in R by R_{sd}. $R[q]$ denotes the set of rules in R of which q is either the consequent or the final consequent. For example, given R=$\{a \rightarrow (b \rightarrow c), d \rightarrow c\}$, we have $R[c]$=R and $R[b \rightarrow c]$=$\{a \rightarrow (b \rightarrow c)\}$.

$R[q]_{\hookrightarrow}$ is the set of rules satisfying the conditions of $R[q]$ and that all the rule strengths toward q are stronger than or equal to $\hookrightarrow$. For example: given R=$\{a \rightarrow (b \rightarrow c), a \rightarrow (b \Rightarrow c), a \rightarrow (b \leadsto c)\}$, we have $R[c]_{\rightarrow}$=$\{a \rightarrow (b \rightarrow c)\}$, $R[c]_{\Rightarrow}$=$\{a \rightarrow (b \rightarrow c), a \rightarrow (b \Rightarrow c)\}$, and $R[(b \leadsto c)]_{\rightarrow}$=$\{a \rightarrow (b \leadsto c)\}$.

2.1 Proof Theory

In order to make the presentation concise, in this paper we only consider DL^{ns} theories that R does not contain defeaters and rules with the empty set as their antecedent, such as $\{\} \rightarrow p$.

Unlike DL, a conclusion of a DL^{ns} theory is a tagged rule instead of just a tagged literal. Since a literal is considered a strict rule in DL^{ns}, this representation of conclusions includes tagged literals as well. The same set, $\{+\Delta, -\Delta, +\partial, -\partial\}$, of tags defined in DL is used in DL^{ns} with the exact same meaning.

In the course of derivations we will make use of auxiliary (sub) theories of a basic theory, and the elements of a derivation refer to these auxiliary (sub) theories. Thus given a theory D and a tagged literal $\pm\#q$, we use the notation $D(\pm\#q)$ to indicate that the tagged literal $\pm\#q$ has been derived/refers to the theory D.

Provability is defined below. It is based on the concept of a derivation (or proof) in $D = (F, R, \succ)$ as in DL. A derivation is a finite sequence $P = (P(1), \ldots, P(n))$ of tagged rules (or literals) satisfying the following conditions ($P(1..i)$ denotes the initial part of the sequence P of length i):

$+\Delta$: $P(i+1) = D(+\Delta q)$ if
(1) $\exists s \in R \cup F$ such that $s \geq q$ or
(2) $\exists t \geq q\ \exists s \in R_s[t]_{\rightarrow}\ \forall a \in A(s)_t$: $D(+\Delta a) \in P(1..i)$ or
(3) For $D' = (A(q), R_s, \emptyset)$, $D'(+\Delta C(q)) \in P(1..i)$.

To show that a rule (or a literal) q is definitely provable in D, i.e., $D(+\Delta q)$, we have three choices: (1) we show that a rule at least as strong as q is a rule of D; or (2) we show that a rule at least as strong as q can be deduced only from strict rules; or (3) we show that the consequent of q is provable definitely in the new theory D' consisting of the supposition (the antecedent of q) and the strict rules of D.

$+\partial$: $P(i+1) = D(+\partial q)$ if
(1) $D(+\Delta q) \in P(1..i)$ or
(2)(2.1) $\exists u \geq q\ \exists r \in R_{sd}[u]\ \forall a \in A(r)_u$:$D(+\partial a) \in P(1..i)$ and
  (2.2) $D(-\Delta N(q)) \in P(1..i)$ and
  (2.3) $\forall v \geq N(q)\ \forall s \in R[v]$ either
    (2.3.1) $\exists a \in A(s)_v$: $D(-\partial a) \in P(1..i)$ or
    (2.3.2) $\exists u \geq q\ \exists t \in R_{sd}[u]\ \forall a \in A(t)_u$: $D(+\partial a) \in P(1..i)$ and $t \succ s$ or
(3) For $D' = (A(q), R, \succ)$, $D'(+\partial C(q)) \in P(1..i)$.

To show that a rule (or a literal) q is defeasibly provable in D, i.e., $D(+\partial q)$, we have three choices: (1) we show that q is already definitely provable; or (2) we show that a rule at least as strong as q is defeasibly deduced from the defeasible part of D and that "attacks", which are reasoning chains in support of $N(q)$, are either not provable or defeated (i.e., they are weaker than appliccable rules for the conclusion we want to prove); or (3) we show that the consequent of q is defeasibly provable from an auxiliary theory D' consisting of the supposition (the antecedent of q) and all the rules of D. In (2.2) and (2.3), unlike DL, we consider reasoning chains in support of $(A(q) \Rightarrow {\sim}C(q))$ as "attacks" instead of ${\sim}q$. This is just one of the possible interpretations of a negation of a rule that has been considered in this paper. An argument for this is that one would be reluctant to accept $+\partial(a \rightarrow b)$ (and/or $+\partial(a \Rightarrow b)$) if any of the followings are supported: $+\Delta(a \rightarrow {\sim}b)$, $+\Delta(a \Rightarrow {\sim}b)$, $+\partial(a \rightarrow {\sim}b)$, or $+\partial(a \Rightarrow {\sim}b)$. Another possible interpretation of $N(r)$ is that the rule is not present in the theory. However, we do not pursue this interpretation here since it treats negation of literals and negation of rules differently.

The conditions for negative provability ($-\Delta$ and $-\partial$) can be constructed similarly following the principle of strong negation described in [2]. Thus, given the limited space, they are not presented here.

2.2 An Example

Let us consider the following scenario. A company has the policy that all confidential documents must be encrypted when they are sent by email, and no confidential document can be sent to people outside the company. A document is classified as confidential when its disclosure would harm the interests of the company. Let us suppose we have a document d describing the details of an application for a patent. Here we have that if the document is disclosed before the grant of the patent then the knowledge in it will be classified as public domain, and if something is public domain, other concurrent companies can use the technology described in the document. But if other companies use the technology, then its usage will generate less revenue than if it were secret and this will harm the interest of the company. This scenario can be described in a very natural fashion by the following DL^{ns} theory (in this example we use rule schemas, where each rule must be understood as the set of its ground instances):

r1: $(\mathit{Disclose}(x) \Rightarrow \mathit{HarmInterests}) \Rightarrow \mathit{Confidential}(x)$
r2: $\mathit{Confidential}(x) \Rightarrow \mathit{Encrypt}(x)$
r3: $\mathit{Disclose}(x) \Rightarrow \mathit{PublicDomain}(x)$
r4: $\mathit{PublicDomain}(x) \Rightarrow \mathit{FreeUseOf}(x)$
r5: $\mathit{FreeUseOf}(x) \Rightarrow \mathit{LessRevenue}(x)$
r6: $\mathit{LessRevenue}(x) \Rightarrow \mathit{HarmInterests}$

Now the question is whether a document describing a pending patent must be encrypted. To prove $Encrypt(d)$ we have to determine whether the document is classified as confidential. In this case we have to see whether we can prove the antecedent of the rule giving the condition to determine whether a document is confidential or not. Thus we have to use the rules in the theory to verify whether there is a relationship between the disclosure of the document and the potential harm caused to the interests of the company. In this case we assume hypothetically the $Disclose(x)$ holds and we try to derive $HarmIntersts$. This derivation succeeds and thus we can prove that the document must be encrypted.

3 Conclusion

We presented an extension of Defeasible Logic called DL^{ns}, which admits one level of nesting of rules both in the antecedent and the consequent of non-monotonic rules. It is constructed to demonstrate the general idea of our approach in developing DL^{n}, which accepts arbitrary nesting of rules such as $(a \rightarrow (b \Rightarrow c)) \rightarrow d$.

In the derivation of rules, our approach ensures appropriate connections between the antecedent and consequent of the rule as in relevant logic (see [3]) by insisting on relevance between antecedents and consequents by explicit rules being present in the theory for evaluating the rules.

We have introduced the concept that rules with the same relations can be ordered by their rule strengths (rule types). Using this concept and the requirement for an appropriate consequence connection between the antecedent and consequent of a rule, we have defined provability for both rules and literals. The provability condition is simple and it allows nested rule expressions and provides additional forms of conclusions such as $+\Delta r_{\rightarrow}$ and $+\partial r_{\Rightarrow}$.

References

1. Gelati, J., Governatori, G., Rotolo, A., Sartor, G.: Normative autonomy and normative co-ordination: Declarative power, representation, and mandate. Artificial Intelligence and Law **12(1-2)** (2004) 53–81
2. Antoniou, G., Billington, D., Governatori, G., Maher, M.J.: A flexible framework for defeasible logics. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, AAAI Press / The MIT Press (2000) 405–410
3. Anderson, A.E., Belnap, N.D.: Entailment: the Logic of Relevance and Necessity Vol 1. Princeton University Press (1975)

ContractLog: An Approach to Rule Based Monitoring and Execution of Service Level Agreements

Adrian Paschke[1], Martin Bichler[1], and Jens Dietrich[2]

[1] Internet-based Information Systems, Technische Universität München
{Paschke,Bichler}@in.tum.de
[2] Information Sciences & Technology, Massey University
J.B.Dietrich@massey.ac.nz

Abstract. In this paper we evolve a rule based approach to SLA representation and management which allows separating the contractual business logic from the application logic and enables automated execution and monitoring of SLA specifications. We make use of a set of knowledge representation (KR) concepts and combine adequate logical formalisms in one expressive formal framework called **ContractLog.**

1 Introduction

Service Level Management (SLM) and Service Level Agreements (SLAs) are of growing commercial interest with a deep impact on the strategic and organisational processes as intensified interest in accepted management standards like ITIL[1] or the new BS15000[2] shows. Additionally, IT virtualisation and upcoming flexible IT infrastructures like e.g. new middleware products, storage area networks or grids services pave the way for new service oriented business models (e.g. “on-demand”, “pay-per-use”, “utility computing”) with flexible and more individual contracts. [1] This needs new levels of flexibility and automation in SLA management not available with the current technology and tools [2, 3]. This paper proposes a rule based representation of SLAs using sophisticated, logic-based knowledge representation (KR) concepts as an alternative to natural language defined contracts or pure procedural implementations in programming languages such as Java or C++. We combine selected adequate logical formalisms in one expressive framework called **ContractLog** with which to describe formal rule based contract specifications which can be automatically monitored and executed**.** The essential advantages of ContractLog are:

1. Contract rules are separated from the service management application. This allows easier maintenance and management and facilitates contract arrangements which are adaptable to meet changes to service requirements dynamically with the indispensable minimum of service execution disruption at runtime, even possibly permitting coexistence of differentiated contract variants.
2. Rules can be automatically linked (rule chaining) and executed by rule engines in order to enforce complex business policies and individual and graduated contractual agreements.

[1] IT Infrastructure Library (ITIL): www.itil.co.uk
[2] BS15000 IT Service Management Standard: www.bs15000.org.uk

3. Test-driven validation and verification methods can be applied to determine the correctness and completeness of contract specifications against user requirements [4] and large rule sets can be automatically checked for consistency. Additionally, explanatory reasoning chains provide means for debugging and explanation. [5]
4. Contract norms like rights and obligations can be enforced and contract violations can be (proactively) detected and treated via automated monitoring processes and hypothetical reasoning. [5]

The rest of the paper is structured as follows. In section 2 we define the requirements for a logic based rule language capable of representing, monitoring and enforcing service contracts. In section 3 we present an overview of our solution to meet these requirements – the ContractLog framework. In section 4 we describe our implementation effort based on the open source rule engine Mandarax and compare our rule based approach to common procedural implementations. Further information on our implementation and more details on the applied logical formalisms can be found in [3, 5, 6] and on our project web site [7]. Finally, in section 6 we conclude this paper with a short summary and an outlook on the higher level **R**ule **B**ased **SLA** language: **RBSLA** [7].

2 Requirements

We analyzed several real world SLAs from different service providers in different branches and several commercial tools like e.g. Tivoli SLA, in order to identify the problems and to derive the requirements on an adequate, automated SLA representation language. The two main problems which we found are:

1. In many companies SLAs are informally described in natural language. This leads to simplified SLA rules and many manual processes in management and monitoring of SLAs.
2. Existing SLM tools with their hard coded application logic and common reference models like ITIL are too little automated, flexible and adaptable.

As a consequence SLA management needs new ways of knowledge representation for contractual agreements and new technical solutions for contract monitoring and enforcement. Beside basic information about the roles of the parties, the contract life time, the agreed services, etc. SLAs contain (business) rules on rights and obligations, prices and costs, quality of service (QoS) and service levels, penalties for contract violations, termination conditions etc. Automated monitoring and execution of such rules requires formalization of the rule syntax. Logic-based rule languages and dedicated rule engines can be used to solve this task [2, 3]. However, SLAs have a number of requirements regarding an adequate knowledge representation. Figure 1 shows the main requirements. For a detailed description of these and further requirements not listed here see [6].

3 The ContractLog Framework

Table 1 summarizes the main concepts used in ContractLog, our solution to the requirements identified in section 2 (see fig. 1).

1. Rule chaining
2. Default rules
3. Rule prioritization
4. Contract modularization
5. External data integration
6. Procedural attachments
7. OO type system and integration of business objects
8. Semantic (business) vocabularies and domain descriptions
9. Situated processing of events and actions
10. Temporal reasoning on events and their effects
11. Contract norms on an individual and group level
12. Verification, validation and conflict resolution
13. Declarative rule syntax and rule serialization

Fig. 1. Main requirements on a declarative rule language

Table 1. Main logic concepts of ContractLog

Logic	Usage
Derivation rules (horn logic with NaF)	Enables deductive reasoning on business rules.
Event-Condition-Action rules (ECA)	Active event detection and situative behaviour by event-triggered executable actions.
Event Calculus (temporal reasoning)	Temporal reasoning about dynamic systems, e.g. effects of events on the contract state.
Defeasible logic / GCLP (priorities)	Default rules and priority relations of rules. Facilitates conflict detection and resolution as well as revision/ updating and modularity of rules.
Deontic logic	Enables representing rights and obligations as deontic contract norms „permission, prohibition, obligation“.
Description logic	Enables semantic domain descriptions (e.g. contract ontologies) to hold rules domain independent. Facilitate exchangeability and interpretation.
Procedural object-oriented logic / procedural attachments	Procedural attachments integrate object oriented programming into declarative rules. Merits the benefits of procedural logic (e.g. Java EJBs) and declarative logic programming (representing business logic).

In the following we want to describe the main formalism used in ContractLog. More information can be found in [1-3, 5, 6] and on our project site [1].

Derivation Rules with Procedural Attachments and External Data Integration
Derivation rules based on horn logic supplemented with negation as failure (NaF) and rule chaining enable a compact representation and a high level of flexibility in automatically combining rules to form complex business policies and graduated contract rules [3, 8]. On the other hand procedural logic as used in programming languages is highly optimized in solving computational problems - however, with the disadvantage that the complete control flow must be implemented. Procedural attachments and the use of a typed logic[3] offer a clean way of integrating programming languages into logic based rule execution paving the way for intelligently accessing or generating data for which the highest level of performance is needed and the logical component

[3] ContractLog supports typed variables and constants based on the object-oriented type system of Java

is minimal. This includes accessing external databases using optimized query languages like SQL to temporarily populate the knowledge base with the needed facts for the inference processes at query time. After the query has been answered (using backward reasoning) these facts can be discarded from memory and therefore replication of data is not necessary any more, which is crucial, as in SLA management we are facing a knowledge intensive domain which needs flexible data integration from multiple rapidly changing data sources, e.g. business data from data warehouses, system data from system management tools, process data from work flows, domain data from ontologies etc. Additionally, the tight integration with Java enables (re-)using existing business objects implementations such as EJBs and system management tools.

ECA Rules

A key feature of a SLA monitoring system is its ability to actively detect and react to events and many rules in SLAs are basically Event Condition Action (ECA) rules, e.g.: "*If the service becomes unavailable (Event) then send a notification message to the service administrator (Action)*". We implemented support for active ECA rules in our backward reasoning system based on an independent daemon process, which monitors all ECA rules by periodically querying the rule base using a thread pool for parallel execution of rules. We represent an ECA rule as a derivation rule: *eca(T,E,C,A)*. Each term *T* (time), *E* (event), *C* (condition) and *A* (action) references to a further derivation rule which implements the respective functionality of the term. The additional term *T* (time) is introduced to define the monitoring intervals/schedules in order to control monitoring costs for each rule. Example:

eca(everyMinute, serviceUnavailable, notScheduledMaintanance, sendNotification)			
everyMinute(DT) ← ...	*serviceUnavailable(DT) ← ...*	*notScheduledMaintanance(DT) ← ...*	*sendNotification(DT) ← ...*

Rule chaining combining derivation rules can be used to build complex functionalities, which can be referenced from several ECA rules. More details on the ECA implementation can be found in [6].

Event Calculus

The Event Calculus (EC) [9] is a formalism for temporal reasoning about events and their effects on a knowledge system. It defines a model of change in which *events* happen at *time-points* and *initiate* and/or *terminate time-intervals* over which some *properties* (time-varying **fluents**) of the world hold. We implemented the classical logic formulations using horn clauses and made some extensions to the core set of axioms to represent derived fluents, delayed effects (e.g. countdowns, validity time of norms), continuous changes (e.g. time-based counter) and epistemic knowledge (planned events e.g. for hypothetical reasoning) [5, 6]:

Classical Domain independent predicates/axioms		ContractLog Extensions	
happens(E,T)	*event E happens at time point T*	*valueAt(P,T,X)*	*parameter P has value X at time point T*
initiates/terminates(E,F,T)	*event E initiates/terminates fluent F*	*planned(E,T)*	*event E is believed to happen at time point T*
holdsAt(F,T)	*fluent F holds at time point T*	*occurred(E,T)*	*event E actually happened*
		derivedFluent(F)	*derived fluent F*

The EC and ECA can be combined and used vice versa, for example fluents (holdsAt) can be used in the condition parts of ECA rules or ECA rules can be used to persistently assert detected events to the EC knowledgebase and define e.g. ECA rules with post conditions (a.k.a. ECAP rules). The EC enables us to model the effects of events on changeable SLA properties (e.g. deontic contract norms describing rights and obligations) and to reason about the contract state at certain time points. In addition we can define complex state transition rules similar to workflows. This is very useful for deontic contract norms, e.g., for representing violations of norms (e.g. violation of fulfilling an obligation in a defined period).

Deontic Logic
Deontic Logic (DL) studies the logic of normative concepts such as obligation (O), permission (P) and prohibition (F). However, classical standard deontic logic (SDL) offers only a static picture of the relationships between co-existing norms and does not take into account the *effects of events on the given norms* and *coherences between norms*, e.g. violations of norms. Another limitation is the inability to express *personalized statements*. In real world applications deontic norms refer to an explicit concept of an agent. These limitations make it difficult to satisfy the needs of practical contract management. Therefore, we extended the concepts of DL with a role-based model and integrated it in our Event Calculus implementation in order to model the effects of events on deontic norms and to represent coherences between deontic norms. [5] A deontic norm consists of the normative concept (*norm*), the subject (*S*) to which the norm pertains, the object (*O*) on which the action is performed and the action (*A*) itself. We represent a role based deontic norm ($N_{s,o}A$) as an EC fluent: *norm(S, O, A)*, e.g. *inititates(e1,* ***permit(s,o,a)****,t1)*. Additionally, we implemented typical DL inference axioms in ContractLog, e.g.: $O_{s,o}A \rightarrow P_{s,o}A$: *holdsAt(permit(S,O,A),T) ← holdsAt(oblige(S,O,A),T)* or $F_{s,o}A \rightarrow W_{s,o}A$: *holdsAt(waived(S,O,A),T) ← holdsAt(forbid(S,O,A),T)* etc. and additional rules to deal with deontic conflicts, violations of deontic norms and their contrary-to-duty paradoxes, e.g. Authorization Conflict: *holdsAt(authConflict(S,O,A),T) ←holdsAt(permit(S,O,A),T). holdsAt(forbid(S,O,A),T)*. The tight combination of the time based EC with role based deontic norms enables the definition of institutional power assignment rules (e.g. empowerment rules) for creating institutional facts which are initiated by a certain event and hold until another event terminates them. Further we can define complex coherences between norms in workflow like settings which exactly define the actual contract state and all possible state transitions. In particular *derived fluents* and *delayed effects* (with *trajectories and parameters* [5]) offer the possibility to define violations of contract norms and their consequential secondary norms e.g., conditional contrary-to-duty (CTD) obligations a.k.a. exceptions. A typical example which can be found in many SLAs is a primary obligation which must be fulfilled in a certain period, but if it is not fulfilled in time, then the norm is violated and a certain "reparational" norm is in force, e.g., a secondary obligation to pay a penalty or a permission to cancel the contract etc. [5, 6] Example:

"If the service is unavailable, the SP is ***obliged*** *to restore it within $t_{deadline}$. If the SP fails to restore the service in $t_{deadline}$, (violation) the SC is* ***permitted*** *to cancel the contract (consequence)."*
<u>Representation in ContractLog</u>
initiates(unavailable, oblige(SP, Service, start()),T) // *defines the primary obligation initiated by an certain event* *terminates(available, oblige(SP, Service, start()),T)* // *defines the event which normally terminates the obligation* *trajectory(oblige(SP,Service,start()),T1,deadline,T2,(T2 - T1))* // *defines the period in which the obligation must be fulfilled* *happens(elapsed,T) ← valueAt(deadline,T, $t_{deadline}$)* // *defines the violation event which happens when the deadline is reached* *initiates(elapsed, permit(SC, Contract, cancel()),T)* // *initiates the derived permission to cancel the contract*

Defeasible Logic

We adapt two basic concepts in ContractLog to solve conflicting rules (e.g. conflicting positive and negative information) and to represent rule precedences: Nute's defeasible logic (DfL) [10] and Grosof´s Generalized Courteous Logic Programs (GCLP) [11]. There are four kinds of knowledge in DfL: *strict rules, defeasible rules, defeaters and priority relations*. We base our implementation on the meta-program found in [12] to translate defeasible theories into logic programs and extended it to support priority relations *r1>r2: overrides(r1,r2)* and conflict relations in order to define conflicting rules not just between positive and negative literals, but also between arbitrary conflicting literals. Example:

> *Rule1 "discount": All gold customers get 10 percent discount."*
> *Rule2 "nodiscount": Customers who have not paid get no discount."*
> *ContractLog DfL: … overrides(discount, nodiscount) … // rule 1 overrides rule 2*

GCLP is based on concepts from DfL. It additionally implements a so called *Mutex* to handle arbitrary conflicting literals. We use DfL to handle conflicting and incomplete knowledge and GLCP for prioritisation of rules. A detailed formulation of our implementation can be found in [6].

Description Logics

Inspired by recent approaches to combine description logics and logic programming [13, 14] we have implemented support for RDF/RDFS/OWL descriptions to be used in ContractLog rules. At the core of our approach is a mapping from RDF triples (constructed from RDF/XML files via a parser) to logical facts: RDF triple:*subject predicate object* → LP Fact: *predicate(subject, object)*, e.g.:

$a : C$, i.e., the individual a is an instance of the class C: *type(a,C)*
$< a,b >: P$, i.e., the individual a is related to the individual b via the property P: *property(P,a,b)*

On top of these facts we have implemented a rule-based inference layer and a class and instance mapping[4] [7] to answer typical DL queries (RDFS and OWL Lite/DL inference) such as class-instance membership queries, class subsumption queries, class hierarchy queries etc. For example:

<u>RDFS inference examples:</u>

$C \sqsubseteq D$, i.e., class C is subclass of class D: *type(a, D)* ← *subClassOf(C,D), type(a,C)*
$Q \sqsubseteq P$, i.e., Q is a subproperty of P: *property(Q,a,b)* ← *subPropertyOf(Q,P), property(P,a,b)*
$T \sqsubseteq \forall P.C$,i.e., the range of property P is class C: *type(b,C) <-- range(P, C), property(P, a, b)*
$T \sqsubseteq \forall P`.C$,i.e., the domain of property P is class C: *type(b,C) <-- range(P, C), property(P, a, b)* *etc.*

<u>OWL inference examples:</u>

$C \equiv D$, i.e., class C is equivalent to class D: *type(a,C) <-- equivalentClass(C, D), type(a,D)*
type(a,D) <-- equivalentClass(C, D), type(a,C)
$P^{+} \sqsubseteq P$ i.e., property P is transitive: *property(P,a,c)* ← *type(P,"owl:TransitiveProperty"),property(P, a,b),property(P,b,c)* *etc.*

This enables reasoning over large scale DL ontologies and it provides access to ontological definitions for vocabulary primitives (e.g. properties, class variables and indi-

[4] To avoid backward-reasoning loops in the inference algorithms

vidual constants) to be used in LP rules. In addition to the existing Java type system, we allow domain independent logical objects in rules to be typed with external ontologies (taxonomical class hierarchies) represented in RDF, RDFS or OWL.

4 Implementation and Discussion

We implemented the ContractLog framework based on the backward-reasoning rule engine Mandarax [8] and the Prova language extension [15], which provides a Prolog related syntax. Mandarax is an open source java-based rule engine for backward reasoning derivation rules. It provides a typed logic (typed rule terms) and procedural attachments which wrap java methods. This allows combining the benefits of LP and object-oriented programming and provides a high level of flexibility and automation. It offers the option to restrict the applicability of rules and to control the level of generality in queries and most importantly it makes possible the desired tight integration of Java code into logical rules, e.g. using monitoring functions from existing system management tools and delegating computation intensive tasks to Java (e.g. for computing average performance values and service levels), or triggering action functionalities from existing business object implementations like EJBs in ECA rules. Additionally, it supports clause sets to ground rules on data stored in external databases. This enables integrating facts from external databases (e.g. a data warehouse) via highly optimized query languages such as SQL into rule executions. Because we are using well understood and sound logic formalism and implement them on the basis of horn logic our logic framework stays computational tractable and efficient although it provides rich expressiveness.

In contrast to procedural programming approaches where the control flow must be completely implemented, the logic based rule approach allows a more compact representation of SLAs. Additionally, dynamic reaction on external events with ECA rules and temporal conclusions about their effects on the contract state, e.g. on rights, obligations or violations are enabled by computational models like the Event Calculus. A static procedural code representing this type of temporal event based logic would have been much more cumbersome to implement and maintain. Other examples are *graduated rules* e.g. graduated penalty rules for missing certain availability levels, *dynamic rules*, e.g. to adapt to special situations or complex *coherences between rules*. From these examples it is easy to see why the declarative style of logic programming can be superior to pure procedural languages in situations when flexibility and code economy are required to represent business logic which is likely to change over time.

5 Conclusion and Outlook

In this paper we have describe a rule based approach to SLA representation and management. We have summarized the requirements on an adequate representation language and evolved our ContractLog framework on the basis of horn rules and meta programming techniques to solve this needs. In contrast to conventional pure procedural programming approaches our logic based approach simplifies maintenance, management and execution of SLA rules and allows easy combination and revision of rule sets to build sophisticated and graduated contract agreements, which are more suitable in a dynamic service oriented environment than the actually applied, simplified rules and the less adaptable procedural management tools. However, real usage of a representation language which is usable by others than its inventors immediately makes rigorous demands on the syntax: e.g. comprehension, readability and usability of the language by users, compact representation, exchangeability with other formats, means for serialization, tool support in writing and parsing rules etc. and in particular a declarative syntax. We try to address these requirements with our superimposed declarative **R**ule **B**ased **SLA** language **RBSLA** [7], which is implemented on top of ContractLog. It adapts and extends RuleML [16] to the needs of the SLA domain. The main additional features we introduce are: (1) *definitions and terms* defining the meaning of the concepts used in SLAs by referencing on external contract vocabularies and semantic ontologies (RDFS/OWL); (2) *ECA rules* including monitoring schedules/intervals, active event monitoring/measurement functions and actively triggered, executable actions; (3) *deontic personalized contract norms with consequential violations and penalties* triggered by time based events; (4) *integration of external data and system/object functionalities* via procedural attachments, clause sets and typed constants and variables (Java or RDFS/OWL); (5) *modularization of contract structures and rule sets* including defeasible rules and priority relations; This includes a transformation implementation which maps the declarative RBSLA into executable ContractLog rules (Prova/Prolog syntax) and additionally performs validation, optimization and refactoring of the declarative rule sets during this process.

References

1. Bichler, M., Diernhofer, N., Fay, F., König, C., MacWilliams, A., Paschke, A., Setzer, T., Völk, G., *Dynamic Value Webs for IT-Services: IT-Service Technologies and Management*. 2004, Siemens SBS / TUM, research study, 10/2004: Munich.
2. Paschke, A. and M. Bichler, *Rule-based Languages for the Representation of Electronic Contracts - A concept for using Knowledge-based Systems in the Development of flexible Internet-based Information Systems. (in german language)*. 04/2003, IBIS, TUM, Working Paper.
3. Paschke, A., *Rule Based SLA Management - A rule based approach on automated IT service management (in german language)*. 6/2004, IBIS, TUM, Working Paper.
4. Dietrich, J. and A. Paschke. *On the Test-Driven Development and Validation of Business Rules*. in *ISTA05*. 2005.
5. Paschke, A. and M. Bichler. *SLA Representation, Management and Enforcement - Combining Event Calculus, Deontic Logic, Horn Logic and Event Condition Action Rules*. in *EEE05*. 2005. Hong Kong, China.
6. Paschke, A., *ContractLog - A Logic Framework for SLA Representation, Management and Enforcement*. 7/2004, IBIS, TUM.
7. Paschke, A., *RBSLA: Rule-based SLA, http://ibis.in.tum.de/staff/paschke/rbsla/index.htm*. 2005.
8. Dietrich, J. *A Rule-Based System for eCommerce Applications*. in *KES 2004*. 2004.
9. Kowalski, R.A. and M.J. Sergot, *A logic-based calculus of events*. New Generation Computing, 1986. 4: p. 67-95.
10. Nute, D., *Defeasible Logic*, in *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 3*, D.M. Gabbay, C.J. Hogger, and J.A. Robinson, Editors. 1994, Oxford University Press.

11. Grosof, B.N., *A Courteous Compiler From Generalized Courteous Logic Programs To Ordinary Logic Progams.* IBM, 1999.
12. Antoniou, G., et al. *A flexible framework for defeasible logics*. in *AAAI-2000*. 2000.
13. Levy, A. and M.-C. Rousset. *A Representation Language Combining Horn Rules and Description Logics*. in *ECAI96*. 1996.
14. Grosof, B.N., et al. *Description Logic Programs: Combining Logic Programs with Description Logic*. in *WWW03*. 2003: ACM.
15. Kozlenkov, A. and M. Schroeder, *Prova*. 2004, http://comas.soi.city.ac.uk/prova/.
16. Wagner, G., S. Tabet, and H. Boley. *MOF-RuleML: The abstract syntax of RuleML as a MOF model. OMG Meeting*. 2003.

The OO jDREW Reference Implementation of RuleML

Marcel Ball[1], Harold Boley[2], David Hirtle[1,2], Jing Mei[1,2], and Bruce Spencer[2]

[1] Faculty of Computer Science, University of New Brunswick
Fredericton, NB, E3B 5A3, Canada
{maball,david.hirtle,jingmei.may}@gmail.com
[2] Institute for Information Technology – e-Business
National Research Council of Canada
Fredericton, NB, E3B 9W4, Canada
{Harold.Boley,David.Hirtle,Jing.Mei,Brunce.Spencer}@nrc.gc.ca

Abstract. This paper presents the open source reference implementation of RuleML based on modular XML Schema definitions and bidirectional OO jDREW interpreters written in Java. For the family of RuleML sublanguages, schema modularization and RDF rules are discussed. The central bidirectional interpreters are introduced via jDREW principles, and explained w.r.t. OO jDREW slots, types, OIDs, and extensions.

1 Introduction

The syntax of RuleML derivation rules has been defined by XML Schema definitions. The model-theoretic semantics of several RuleML sublanguages (including Datalog, Hornlog, and Folog) is defined in the classical way; for sublanguages with negation-as-failure, well-founded models have been proposed. We have implemented the operational semantics of Derivation RuleML using XSLT translators and the bidirectional interpreter (the OO jDREW rule engine) described in this paper. This reference implementation is available open source via the RuleML and jDREW websites.

2 Modular Schemas for a Family of RuleML Sublanguages

The top-level of the current family of RuleML sublanguages shows the major distinction between Derivation Rules, including Hornlog above Datalog, and Action Rules, including Production Rules. We focus here on various expressive classes of Derivation Rules and their XML Schema Definitions (XSDs) as described in the Modularization document. The most recent (public) schema specification of RuleML is always available at http://www.ruleml.org/spec.

2.1 Schema Modularization

We employ modular XSDs, using a content model-based approach to take advantage of inheritance between schemas. Each expressive class syntactically distinguishable via an XSD (such as Datalog vs. Hornlog) can thus be addressed by the URI of its XSD. This permits receivers of a rulebase to validate if it conforms to the specified expressive class, before applying any class-specific tools (such as a Datalog vs. Hornlog interpreter). Moreover, a syntactic class is associated with a semantic class (such as Datalog vs. Hornlog with a function-free vs. function-containing Herbrand model). The relationships between these elements of the model are either aggregation, e.g. "Datalog is part of Hornlog", or generalization, e.g. "Bindatalog is a Datalog".

From an implementation perspective, elementary non-standalone modules contain only element and/or attribute definitions and are not intended to be used directly for validation. They may, however, be used to create new document types by others wishing to "borrow" certain elements of RuleML. The actual sublanguages, on the other hand, are schema drivers composed in whole or in part of these modules or derived entirely from other schema drivers.

2.2 RDF Rules as Anchored, Slotted Datalog with Blank Nodes

As an important sublanguage example, the definition of RDF Rules can be introduced in the following steps:

– Datalog is a language corresponding to relational databases (ground facts without complex domains or "constructors") augmented by views (possibly recursive rules).
– Slots permit non-positional arguments. RuleML's user-level metarole 'slot' takes a name-filler pair, accommodating RDF properties.
– Anchors provide object identity via webizing through URIs. Such "URI grounding" is available in RuleML via dual attributes 'wlab' and 'wref', corresponding to RDF's 'about' and 'resource'.
– Blank Nodes are local aliases for existing individuals without need for global names. In RuleML, the F-logic/Flora-2 Skolem-constant approach [1] is used to notate, generate, and refer to Blank Nodes.

Illustrating an RDF-like Business Rule 1:

```
<Implies>
  <body>
    <And>
      <Atom>
       <oid><Var>x</Var></oid>
       <Rel>product</Rel>
       <slot><Ind wref=":price"/><Var>y</Var></slot>
       <slot><Ind wref=":weight"/><Var>z</Var></slot>
      </Atom>
      <Atom>
       <Rel wref="swrlb:greaterThan"/><Var>y</Var><Data>200</Data>
      </Atom>
      <Atom>
      <Rel wref="swrlb:lessThan"/><Var>z</Var><Data>50</Data>
      </Atom>
    </And>
  </body>
  <head>
    <Atom>
      <oid><Var>x</Var></oid>
      <Rel>product</Rel>
      <slot><Ind wref=":shipping"/><Data>0</Data></slot>
    </Atom>
  </head>
</Implies>
```

3 Bidirectional Interpreters in Java

As part of the implementation of RuleML, a systen of bidirectional interpreters, was created in Java. In particular, the OO jDREW reasoning engine [3] contains two modes: a Bottom-Up (forward chaining) version, and a goal driven top-down (backward chaining) version that works in a fashion similar to most Prolog systems. Demo applications (interfaced through Java Web Start) are available at http://www.jdrew.org/oojdrew/demo.html, and the source has been made available for download. A Roadmap for Open Source OO jDREW Development has recently been outlined (http://mail.jdrew.org/pipermail/jdrew-all/2005-June/000001.html). Principles, specifics, and extensions of the features available in OO jDREW are detailed below.

3.1 jDREW Principles

The jDREW toolbox approach [2] provides the flexibility to quickly cope with changes to the implementation of the evolving RuleML standard. There are utilities in jDREW for various tasks: reading files of RuleML statements into the internal clause data structure, storing and manipulating clauses, unification of clauses according to the positions of the selected literals, a basic resolution engine, clause to clause subsumption and clause to clause-list subsumption, choice point managers, priority queues for various reasoning tasks, and readable top-level procedures.

Much of the flow of control is oriented around iterators, objects that maintain the state of a partially completed computation. Thus you pay as you go when you want the engine to perform the next step. The advantages of this architecture are its consistency and efficiency. There is a common interface for many different reasoning tasks, and there are few additional data structures introduced for storing intermediate results, other than those required by the abstract reasoning procedure. For instance, in the bottom-up system, solutions are generated one-at-a-time, so asking for the next solution may cause the following steps: An iterator will be asked to select the next clause in the so-called 'new results' list that matches eligibility requirements (like not being already subsumed).

3.2 OO jDREW Slots

During the creation of the internal structures, the OO jDREW terms representing atoms and complex terms are normalized, producing the following order for the parameters: oid (object identifier), positional parameters (in their original order), the positional rest term, slotted parameters (in the encounter order), and finally the slotted rest term. Since the ordering of slots within RuleML atoms and complex terms does not carry information, any order can be imposed. In OO jDREW, the slots are ordered based upon the sequence in which they are initially encountered to permit the incremental addition of slots without any reordering.

By using such a normalized form we are able to implement an efficient unification algorithm that has time complexity O(m + n) (where m and n are the numbers of parameters), instead of O(m * n). In our algorithm we scan the two lists of parameters – matching up roles (and positions in the case of positional parameters) – and unify those parameters. If a role is present in one term but not in the other then the unmatched role is added to a list of rest terms in case the other has an appropriate rest term (otherwise unification fails). Such a collection of rest terms is used to dynamically generate a Plex (RuleML's generalization of a list) to be assigned to the corresponding rest parameter.

3.3 OO jDREW Types

OO jDREW includes an order-sorted type system as a core component. This type system allows the user to specify a partial order for a set of classes in RDFS via their (multiple) superclasses, allowing for the reuse of lightweight taxonomies of the Semantic Web. Currently, the system only models the taxonomic relationships between the classes, and cannot model properties with their domain and range restrictions. For example, the current system can model that 'Car' is a 'Motor Vehicle', but cannot model that a car must have a make, model, year, etc.

By building an order-sorted type system into OO jDREW we are able to restrict the search space to only those clauses that have the appropriate types specified for their parameters, leading to a faster and more robust system than one where types are reduced to unary predicate calls in the body.

Extensions to the type system are being considered that would expand its modeling ability. In particular, the user could define a signature using RDFS properties to specify that certain slots must be present for a typing to be valid. We would then be able to prescribe that 'Car' has slots for at least make, model, year, which is not possible in the current system.

3.4 OO jDREW OIDs

The current implementation of OO jDREW, version 0.88, has a preliminary implementation of object identifiers (OIDs). Currently, OIDs are only supported in an atomic formula (<Atom> in RuleML), either as a fact or as part of a rule (<Implies> in RuleML). In this version only symbolic names are allowed as OIDs. The URI-valued wref and wlab attributes, which are part of the RuleML specification, are currently not supported; this is primarily due to W3C issues with the normaliztion of URIs, creating difficulties in determining what URIs should be considered to be equivalent.

The open source roadmap for OO jDREW includes plans to extend support for OIDs beyond their current level. It is envisioned that by the release of version 0.89, OIDs will be supported on levels other than atoms, such as for connectives and performatives. Additionally, wlab and wref should be supported with a preliminary URI normalization system, possibly implementated in OO RuleML [4] itself.

3.5 OO jDREW Extensions

Negation-as-failure (Naf) has first been implemented in OO jDREW TD, and recently introduced into OO jDREW BU for stratified programs. In bottom-up mode, Naf attempts to look up its argument atom via a unifying fact (when no other rule is applicable). If this look-up succeeds, hence the Naf fails, then this rule will be deleted from the given list, else the rule will be partially evaluated into one without Naf.

Equivalence classes have been implemented in OO jDREW BU for the sublanguage datalogeq (Datalog with Equality). For equality ground facts, a newly-built data structure called EqualTable is used to map all equal individuals to one equivalence class. For each equivalence class, we append a fresh symbol to the original OO jDREW SymbolTable, and all equal individuals are redirected to this new symbol. That is, the process of unification and resolution will deal with this new symbol, representing the equivalence class as a whole.

Illustrating Naf and Equal with a Datalog-like Business Rule 2:

```
<Implies>
  <head>
    <Atom>
      <Rel>discount</Rel>
      <Var>customer</Var><Var>product</Var><Ind>5.0 percent</Ind>
    </Atom>
  </head>
  <body>
    <And>
      <Atom><Rel>premium</Rel><Var>customer</Var></Atom>
      <Atom><Rel>onsale</Rel><Var>product</Var></Atom>
      <Naf>
        <Atom><Rel>special</Rel><Var>product</Var></Atom>
      </Naf>
    </And>
  </body>
</Implies>
```

```
<Equal><Ind>fatherOFtom</Ind><Ind>bob</Ind></Equal>
<Equal><Ind>fatherOFtom</Ind><Ind>uncleOFmary</Ind></Equal>
<Atom><Rel>premium</Rel><Ind>bob</Ind></Atom>
<Atom><Rel>onsale</Rel><Ind>clothes</Ind></Atom>

Results: discount("bob", clothes, "5.0 percent").
         discount("uncleOFmary", clothes, "5.0 percent").
         discount("fatherOFtom", clothes, "5.0 percent").
```

A detailed design of an indexing system has been completed for OO jDREW (http://www.jdrew.org/oojdrew/docs/OOjDREWIndexDesign.pdf) that will index the combined positional and slotted parameters on the top-level of RuleML atoms, along with their associated rest parameters. Once implemented, it should provide a significant increase in efficiency for the common cases, without creating too much overhead for the more unusal boundary cases.

4 Conclusions

RuleML has an open source implementation that is freely available and maintained as the standard evolves. The syntax of the family of sublanguages is specified by modular XML Schema definitions. The operational semantics of RuleML is implemented by a set of bidirectional interpreters (OO jDREW) written in Java for cross-platform compatibility. For interoperability with other standards, translators have also been realized, primarily via XSLT.

References

1. Reasoning about Anonymous Resources and Meta Statements on the Semantic Web, G. Yang and M. Kifer, In Journal on Data Semantics, Volume 1, Pages 69-97, 2003.
2. The Design of j-DREW: A Deductive Reasoning Engine for the Web, B. Spencer, In Proceedings of the First CologNET Workshop on Component-Based Software Development and Implementation Technology for Computational Logic Systems. CBD ITCLS 2002, Madrid, Spain. September 20, 2002. pp. 155-166.
3. OO jDREW: Design and Implementation of a Reasoning Engine for the Semantic Web, Marcel Ball, CS 4997, Faculty of Computer Science, University of New Brunswick, Fredericton, Canada, April 2005.
4. Object-Oriented RuleML: User-Level Roles, URI Grounded Clauses, and Order-Sorted Terms, H. Boley, In Proc. Rules and Rule Markup Languages for the Semantic Web (RuleML-2003). Sanibel Island, Florida, LNCS 2876, Springer-Verlag, October 2003.

Author Index