A Framework for Composition and Inter-operation of Rules in the Semantic Web

Enrico Pontelli and Tran Cao Son
Dept. Computer Science
New Mexico State University
epontell | tson@cs.nmsu.edu

Chitta Baral
Dept. Computer Science
Arizona State University
chitta@asu.edu

Abstract

Recent developments in the RuleML initiative have led to the design of several languages for representing rules. In this paper we describe a framework, based on the integration of different flavors of logic programming, aimed at facilitate reasoning with multiple sources of knowledge expressed in an heterogeneity of RuleML languages.

The framework allows the derivation of logic programming modules from each rule base, and their interoperation through a well-defined module interface. In this paper we describe the basic syntax and semantics of the framework, and its preliminary implementation in the ASP-Prolog system.

1. Introduction

One of the main goals of the Semantic Web initiative [5] is to extend the current Web technology to allow for the development of intelligent agents, which can *automatically* and *unambiguously* process the information available on millions of web pages. It has been recognized very early in the development of the Semantic Web that rules are essential for the Web[1] and for Semantic Web applications—e.g., description of semantic web services, rules interchange for e-business applications.

The RuleML initiative is a response to the need of a shared rule markup language using XML markup, which has a precisely defined semantics and efficient implementations. In recent years, a significant amount of work has been devoted to develop knowledge representation languages suitable for the task and a variety of languages for rule markup has been proposed. The initial design [6] included a distinction (in terms of distinct DTDs) between *reaction rules* and *derivation rules*. The first type of rules is used for the encoding of event-condition-action (ECA) rules while the second is meant for the encoding of implicational/inference rules.

Despite the fact that many different proposals for ECA rules encoding have appeared (e.g., [21, 23, 2, 8]) the work on ECA rules is still very vague. The most recent modularized description of RuleML [17] reports this area (indicated as *PR RuleML* in that document) as work in progress.

The derivation rules component of the RuleML initiative has originated a family of languages. Figure 1[2], from [17], shows the most commonly referred languages; observe that Datalog plays the role of a core language, with simplified versions (unary and binary Datalog) developed for combining RuleML with OWL (as in SWRL [18]). Various sublanguages have been created to include features like explicit equality (e.g., `fologeq`), negation as failure (e.g., `naffolog`), and Hilog layers (e.g., `hohornlog`). In [19], it is argued that any realistic architecture for the Semantic Web must be based on various independent but interoperable languages, one of them is the logic programming language with negation-as-failure. The next example illustrates this point.

Example 1 (Reviewer Selection) *The following example is a variation of the* reviewer selection *problem in [13], and it illustrates the encoding of two rule bases based on two flavors of RuleML.*

Suppose that we need to assign reviewers to papers submitted to RuleML-2006. The goal is to assign papers to experts in the areas. Reviewers are registered in a knowledge base, called Reviewers, *with their expertise. The knowledge base also contains a description of the subareas of each research area. The knowledge base consists of concepts such as "area(A), "inArea(K,A)", "referee(P)", "keyword(K)", and "expertIn(P,A)". This information can be expressed in RuleML as a set of facts. For example,* semantic_web *is a keyword and a research area in* ai, *and* john *is a referee, who is an expert in* semantic_web *can be encoded by the following facts:*

[1] `http://www.w3.org/DesignIssues/Rules.html`

[2] `www.ruleml.org/modularization/ruleml_m12n_089_uml_05-06-01.png`.

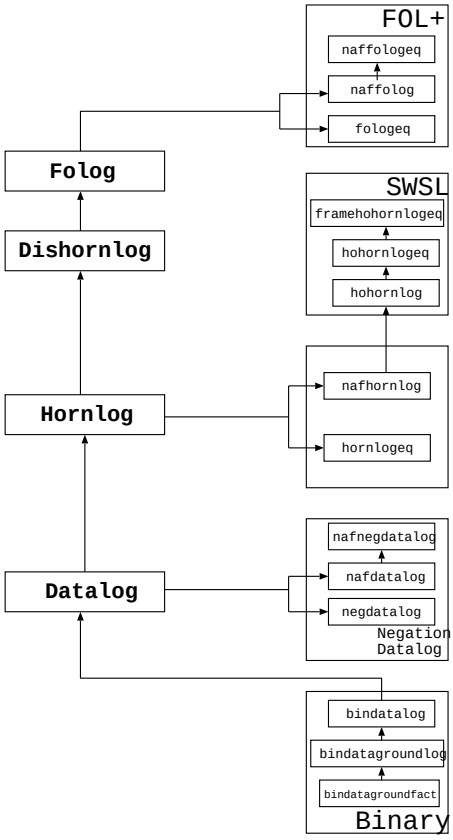

Figure 1. RuleML Derivation Rules

```
<fact> <head> <atom>
    <_opr> <rel> area </rel> </_opr>
       <ind>  ai </ind>
    </atom> </head> </fact>

<fact> <head> <atom>
    <_opr> <rel> keyword </rel> </_opr>
       <ind>  semantic_web </ind>
    </atom> </head> </fact>

<fact> <head> <atom>
    <_opr> <rel> inArea </rel> </_opr>
       <ind>  semantic_web </ind>
       <ind>  ai </ind>
    </atom> </head> </fact>

<fact> <head> <atom>
    <_opr> <rel> referee </rel> </_opr>
       <ind>  john </ind>
    </atom> </head> </fact>

<fact> <head> <atom>
    <_opr> <rel> expertIn </rel> </_opr> paper
       <ind>  john </ind>
       <ind>  semantic_web </ind>
    </atom> </head> </fact>
```

In addition, the knowledge base also contains rules for reasoning about the inclusion of a keyword in a research area, the expertise of a referee, etc. For example, the rule

```
<imp>
 <_head> <atom>
    <_opr> <rel> inArea </rel> </_opr>
    <var> keyword </var>
    <var> area </var>
    </atom>
 </_head>
 <_body>
    <atom>
    <_opr> <rel> inArea </rel> </_opr>
    <var> keyword </var>
    <var> area_one </var>
    </atom>
    <atom>
    <_opr> <rel> inArea </rel> </_opr>
    <var> area_one </var>
    <var> area </var>
    </atom>
 </_body>
</imp>
```

represents the fact that if X is a subarea of Y and Y is a subarea of Z then X is a subarea of Z. We assume that the knowledge base is available from the location `http://www.ruleml2006.org/expertise.ruleml` *which we will simplify to* `expertise` *hereafter. Observed that the representation of this information only employs the* `binary datalog` *language.*

The second knowledge base, called Submissions*, consists of information about the submissions to the conference and rules for assigning papers to reviewers. Each paper is assigned an identification number and is submitted with a set of keywords. This information is represented as a collection of atoms of the form*

$paper(\#n)$	the paper is assigned the id number n
$kw(\#n,\ kw)$	the paper $\#n$ is listed with keyword kw

This can also be expressed as a set of facts. For example, the facts that paper #1 has been submitted and it lists nmr *as its keyword can be expressed by*

```
<fact> <head> <atom>
    <_opr> <rel> paper </rel> </_opr>
       <ind>  1 </ind>
    </atom> </head> </fact>

<fact> <head> <atom>
    <_opr> <rel> kw </rel> </_opr>
       <ind>  1 </ind>
       <ind>  nmr </ind>
    </atom> </head> </fact>
```

The rule for assigning reviewers to papers states that we can only assign reviewer with expertise in the area. E.g., two of the rules used for this purpose are:

```
<imp>
 <_head><atom>
```

```
    <_opr><rel> assign </rel></_opr>
    <var> referee </var>
    <var> paper </var>
    </atom>
 </_head>
 <_body>
    <atom>
    <_opr><rel> expertise#referee </rel></_opr>
    <var> referee </var>
    </atom>
    <atom>
    <_opr><rel> candidate </rel></_opr>
    <var> referee </var>
    <var> paper </var>
    </atom>
    <naf>
      <atom>
       <_opr><rel> not_assign </rel></_opr>
       <var> referee </var>
       <var> paper </var>
      </atom>
    </naf>
  </_body>
</imp>

<imp>
 <_head><atom>
    <_opr><rel> not_assign </rel></_opr>
    <var> referee </var>
    <var> paper </var>
    </atom>
 </_head>
 <_body>
    <atom>
    <_opr><rel> expertise#referee </rel></_opr>
    <var> referee </var>
    </atom>
    <atom>
    <_opr><rel> candidate </rel></_opr>
    <var> referee </var>
    <var> paper </var>
    </atom>
    <atom>
    <_opr><rel> assign </rel></_opr>
    <var> ref1 </var>
    <var> paper </var>
    </atom>
    <naf>
      <atom>
       <_opr><rel> equal </rel></_opr>
       <var> referee </var>
       <var> ref1 </var>
      </atom>
    </naf>
  </_body>
</imp>
```

Observe that in representing the rules for assigning reviewers, we make use of the RuleML `nafdatalog` *sublanguage. We assume that the knowledge base is available from the location* `http://www.ruleml2006.org/papers.ruleml`.

1.1. The Problem

Example 1 highlights a trend in the development of the Semantic Web in general and in the RuleML initiative in particular: the co-existence of different languages with different semantics and reasoning mechanisms. The need for these languages and their interaction have been discussed (e.g., [21, 19]). It is also of note that many of the sublanguages of RuleML have been implemented either through translators (e.g., GEDCOM [12], which translates to XSB and JESS) or engines (e.g., j-DREW [24], a top-down engine for RuleML, DR-Device [3], an engine supporting defeasible logic and both strong and default negation, and CommonRules [11], a bottom-up engine for the Datalog sublanguage).

In the development of intelligent agents that interact with the Semantic Web, we can identify a number of issues that need to be addressed:

1. *Reasoning within one knowledge base*: Being able to reason within a knowledge base implies the ability to interoperate with a computational framework capable of handling the type of knowledge present in the knowledge base (e.g., a production system for ECA rules, a Datalog system for Datalog rules).
2. *Reasoning across different knowledge bases*: This requires
   a. the ability to exchange inference results between different knowledge bases (e.g., the interoperability problem between rules and OWL described in [19]);
   b. the ability to combine reasoning results produced by different reasoning engines;
   c. the ability to properly scope the reasoning w.r.t. a specific knowledge base (e.g., the *scoped inference* issue described in [19]).
3. *Utilizing available knowledge*: This requires the ability to use the results produced by different reasoning processes from different information sources in the construction/implementation of complex Semantic Web applications.

2. A Possible Approach

In this work, we propose a general framework to address the problem of *(i)* inter-operation between knowledge bases encoded using different RuleML languages, and *(ii)* development and integration of different components that reason about RuleML knowledge bases.

The approach adopted in this work relies on using a core logic programming framework to address the issues of integration and inter-operation. In particular, the spirit of our approach relies on the following beliefs:

- the natural semantics of various levels of the RuleML

deduction rules hierarchy can be captured by different flavors of logic programming;

- modern logic programming systems are provided with foreign interfaces that allow declarative interfacing to other paradigms.

The idea is to combine the ASP-Prolog framework of [14] with the notations for modularization of answer set programming of [4, 1]. The result is a logic programming framework, where modules responding to different logic programming semantics (e.g., Herbrand minimal model, well-founded semantics, answer set semantics) can co-exist and interoperate.

The framework provides a natural answer to the problems of use and inter-operation of RuleML knowledge bases described earlier. The overall structure is depicted in Fig. 2. Most of the emphasis will be on using answer set programming to handle some of the sublanguages (e.g., datalog, urdatalog, nafdatalog and negdatalog), though the core framework will naturally support most of the languages (e.g., hornlog, hohornlog).

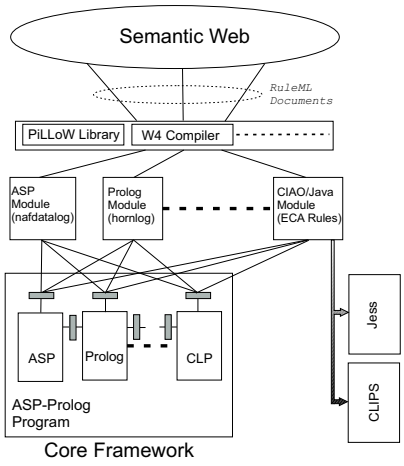

Figure 2. Overall Structure

The problems mentioned in Sect. 1.1 are tackled as follows:

- Issue 1.: Ciao Prolog offers direct access to a collection of modules that support different forms of logic programming reasoning, e.g., traditional Prolog, constraint logic programming, fuzzy Prolog, SQL. In addition, Ciao Prolog provides a mechanism that allows Prolog programs to invoke Java methods, offering a bi-directional communication and a reflection of Java objects into Prolog. This provides, for example, a natural way to execute Java-based engines (e.g., Jess) and communicate between the core framework and external Java modules.
  Furthermore, Ciao Prolog provides PiLLoW, a standardized Prolog library for Web Programming, which provides the core framework with capabilities for Web access (e.g., management of URIs and URLs).
  Thus, we envision the core framework as the bridge between distinct execution models for different RuleML sublanguages.
- Issue 2.a: this issue will be addressed through the introduction of a module system, where different knowledge bases can be encoded (directly or indirectly) as distinct modules. The original import/export of Ciao Prolog can be combined with the languages for answer set modules of [4] to allow forms of bi-directional communication between the core framework and the modules representing the knowledge bases.
- Issue 2.b: The core framework will provide the full computational power of Prolog, Constraint Logic Programming, and Answer Set Programming, combined through a sophisticated module and class system. Module interfaces will allow extraction of semantic information from the various knowledge bases (e.g., result of queries, models of knowledge bases) and reason with them.
- Issue 2.c: the scoped inference is naturally supported by the module system of ASP-Prolog (e.g., skeptical and credulous reasoning w.r.t. answer set modules).
- Issue 3.: can be handled thanks to the combination, in ASP-Prolog and Ciao Prolog, of web access capabilities along with full Prolog computation power.

3. The Proposed Framework

3.1. General Syntax

Languages and Rules: We will consider a logic language $\langle \mathcal{F}, \Pi, \mathcal{V} \rangle$ where $\mathcal{F}$ is a denumerable collection of function symbols, $\Pi = \Pi_u \cup \Pi_d$ is a denumerable collection of predicate symbols and $\mathcal{V}$ is a collection of variables. Π_u are called user-defined predicates while Π_d are called built-in predicates (and $\Pi_u \cap \Pi_d = \emptyset$). We will assume that `assert`, `retract`, `model`, ... are elements of Π_d. We will denote with $ar(\alpha)$ the arity of the symbol $\alpha \in \Pi \cup \mathcal{F}$.

A term is either a variable, an element of $\mathcal{F}$ of arity 0, or a formula of the form $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$, $ar(f) = n$, and $t_1, \ldots, t_n$ are terms. We will say that a term t is ground if it contains no variables. We will denote with $\mathcal{H}_P$ the Herbrad universe for this language.

An atom is a formula of the form $p(t_1, \ldots, t_n)$, where $p \in \Pi$, $ar(p) = n$, and $t_1, \ldots, t_n$ are terms. The atom is ground if $t_1, \ldots, t_n$ are ground. A qualified atom is a formula of the form $t : A$ where t is a ground term (called the *label* of the qualified atom) and A is an atom. In particular, if the predicate p of an atom belongs to Π_d, then the atom can only appear qualified in a rule. A literal is either an atom, a qualified atom, or a formula $not\ A$, where A is an atom. We will denote with $\mathcal{B}_P$ the Herbrand base for this

language (i.e., the set of all ground atoms). For an atom (qualified atom, negative literal) ℓ, we denote with $\pi(\ell)$ the predicate symbol used by the atom (qualified atom, negative literal) ℓ.

A general rule is of the form

$$A \ :- \ B_1, \ldots, B_k \tag{1}$$

where A is an atom and $B_1, \ldots, B_k$ are literals.

Depending on the type of programs we wish to represent, different restrictions can be imposed on the rules:

- **a.** datalog: the B_i in rule 1 can be only atoms/qualified atoms and the terms used in the literals can be only variables or constants (i.e., of arity 0).
- **b.** ground datalog: the B_i in rule 1 can only be atoms/qualified atoms, and the only terms allowed are constants.
- **c.** ground binary datalog: the rules satisfy the conditions of case **b.**, and in addition we require all predicates used to construct atoms to have arity at most 2.
- **d.** datalog with negation: the rules have the format as in case **a.** but negative literals (not) are allowed in the body of the rule.
- **e.** pure prolog: rules of the format 1 are allowed, where B_i are atoms/qualified atoms, and arbitrary terms can be employed, but no `assert` and `retract` are allowed in the rules.
- **f.** impure prolog: rules of the format as in case **e.**, with the additional ability to use the predicates `assert/2` and `retract/2` in the body of the rules.

We will refer to a rule as a Ξ rule (where Ξ is datalog, ground datalog, binary datalog, etc.) to denote a rule that meets the corresponding requirements. An Ξ-program is a collection of Ξ rules.

Give a rule r, we denote with $used(r)$ the set of ground terms t such that t is a label of a qualified atom in r. Given a Ξ program P, we denote with $used(P) = \{t \mid \exists r \in P.\, t \in used(r)\}$. We also introduce $def(P) = \{p \mid p \in \Pi_u, ar(p) = k, \exists r \in P \exists t_1, \ldots, t_k.\, head(r) = p(t_1, \ldots, t_k)\}$.

Module Structure: A module is composed of two parts: a module interface and a module body.

A module interface has the form

$$\begin{array}{lll} :- & \textit{module} & : \ t \\ :- & \textit{import} & : \ t_1, \ldots, t_k \\ :- & \textit{export} & : \ q_1/k_1, \ldots, q_m/k_m \end{array}$$

where t is a ground term, called the *name* of the module, $t_1, \ldots, t_k$ are ground terms (representing names of modules), $q_1, \ldots, q_m$ are predicates, and $k_1, \ldots, k_m$ are non-negative integers, such that $ar(q_i) = k_i$.

The body of a module is a Ξ program for a given Ξ. In that case, we will say that the module is a Ξ module. Given a module named t, we identify with the export set of t ($exp(t)$) the predicates $q_1, \ldots, q_k$ exported by t. We also identify with $imp(t)$ the import set of t, i.e., the names of the modules imported by t.

A program $P = \{M_{t_1}, \ldots, M_{t_k}\}$ is a collection of modules named $t_1, \ldots, t_k$. The *graph* of P ($graph(P)$) is a graph (N, E) where the set of nodes N is $\{t_1, \ldots, t_k\}$ and $(t_i, t_j) \in E$ iff $t_i \in imp(t_j)$. A program P is admissible if it satisfies the following properties:

- for each t_i we have that $imp(t_i) \subseteq \{t_1, \ldots, t_k\}$;
- the graph $graph(P)$ is acyclic.

The module structure can be expanded by allowing cyclic dependencies (i.e., two-way communications between modules) as well as OO-style organization of modules, as described in [4]. We omit this discussion due to lack of space.

Example 2 *The translation process mentioned in Section 3.3 and applied to the first knowledge base, will produce facts and rules of the form:*

$:- module : \texttt{expertise}$
$:- export : \texttt{keyword/1}, \texttt{inArea/2}, \texttt{expertIn/2}, \ldots$

$\texttt{keyword}$(semantic_web).
...
$\texttt{inArea}$(semantic_web, ai).
...
$\texttt{expertIn}$(john, semantic_web).
...
$\texttt{referee}$(john).
...
$\texttt{inArea}(K, A) :- \texttt{inArea}(K, A1), \texttt{inArea}(A1, A).$
...
$\texttt{expert}(P, A) :- \texttt{expertIn}(P, A).$
$\texttt{expert}(P, A) :- \texttt{inArea}(A, A1), \texttt{expert}(P, A1).$

The second knowledge base will be translated in a collection of facts and rules such as:

$:- module : \texttt{papers}$
$:- import : \texttt{expertise}$
$:- export : \texttt{paper/1}, \ldots \texttt{paper}(1).$
...
$\texttt{kw}(1, nmr).$
...
$\texttt{not\_candidate}(R, P) :- \texttt{expertise:referee}(R),$
$\quad \texttt{kw}(P, K),$
$\quad \texttt{expertise:inArea}(K, A),$
$\quad not\ \texttt{expertise:expert}(P, A).$
...
$\texttt{candidate}(R, P) :- \texttt{expertise:referee}(R),$
$\quad not\ \texttt{not\_candidate}(R, P).$
...
$\texttt{assign}(R, P) :- \texttt{candidate}(R, P),$
$\quad not\ \texttt{not\_assigned}(R, P).$
$\texttt{not\_assigned}(R, P) :- \texttt{candidate}(R1, P),$
$\quad \texttt{assign}(R1, P), R1 \neq R.$
$\texttt{done}(P) :- paper(P), expertise : referee(R), assign(R, P).$
$false :- \texttt{assign}(R, P), \texttt{assign}(R, P1), P \neq P1.$
$false :- paper(P), not\ done(P).$

3.2. General Semantics

3.2.1 Pure Programs

In this section we propose a model-theoretic semantics for programs that do not contain any impure prolog module.

Given a program P, a *model naming* function τ is a one-to-one and onto function $\tau : \mathcal{H}_P \mapsto 2^{\mathcal{B}_P}$. We will use this function to assign distinct names to the models of the different modules. In the rest of this work, we will assume that the τ function is fixed.

Given a program $P = \{M_{t_1}, \ldots, M_{t_k}\}$, the acyclic nature of $graph(P)$ guarantees the ability to construct a topological sort of $\{t_1, \ldots, t_k\}$, say $\eta_1, \ldots, \eta_k$ such that if (η_i, η_j) is an edge of the graph, then $i < j$.

Given the program P and a topological sorting of the modules $\eta_1, \ldots, \eta_k$, we construct the semantics module by module, following the topological sort order. The semantics of each module M_i will be given by a collection of models $\mathcal{M}^\tau(M_i)$, where $\mathcal{M}^\tau(M_i) \subseteq 2^{\mathcal{B}_P}$. Given a Ξ program T not containing any qualified atoms and not containing any occurrence of predicates from Π_d, then we assume that its semantics $\mathcal{NAT}(T)$ is given. E.g., if T is a datalog with negation program meeting these conditions, then $\mathcal{NAT}(T)$ will be the set of answer sets of T, while if T is a pure Prolog program, then $\mathcal{NAT}(T)$ contains the least Herbrand model of T. An interpretation of a program P is thus a mapping $\mathcal{M}^\tau : P \mapsto 2^{\mathcal{B}_P}$.

Let us consider a module M_i of P, an interpretation $\mathcal{M}^\tau$, and let τ be a model naming function. Then:

- if A is a ground atom, then $\mathcal{M}^\tau, M_i \models_\tau A$ iff $A \in \mathcal{M}^\tau(M_i)$;
- if $not\ A$ is a ground literal and A is a ground atom, then $\mathcal{M}^\tau, M_i \models_\tau not\ A$ iff $A \notin \mathcal{M}^\tau(M_i)$;
- if $t : A$ is a ground qualified atom and $t \in imp(M_i)$, then $\mathcal{M}^\tau, M_i \models_\tau t : A$ iff for each model $M \in \mathcal{M}^\tau(M_t)$ we have that $M \models_\tau A$.
- if $t : A$ is a ground qualified atom and $t \notin imp(M_i)$, then $\mathcal{M}^\tau, M_i \models_\tau t : A$ iff there exist $t_i \in imp(M_i)$, $M \in \mathcal{M}^\tau(M_{t_i})$ such that $\tau(t) = M$ and $M \models_\tau A$.
- $\mathcal{M}^\tau, M_i \models_\tau t : model(t')$ iff $t \in imp(M_i)$ and $\tau(t') \in \mathcal{M}^\tau(M_t)$.

We can extend the notion of satisfaction to the case of rules and modules. A model of a module is an interpretation that satisfies all rules in the module. A model of a program is an interpretation that satisfies all the modules in the program.

The *model reduct* of M_i w.r.t. $\mathcal{M}^\tau$ (denoted $MR(M_i, \mathcal{M}^\tau)$) is defined as follows:

- remove from M_i all rules that contain in the body a qualified element $t : A$ such that $\mathcal{M}^\tau, M_i \not\models t : A$;
- remove from the remaining rules all occurrences of qualified atoms.

The *model reduct* of the program $P = \{M_{t_1}, \ldots, M_{t_k}\}$ is defined as $\{MR(M_{t_1}, \mathcal{M}^\tau), \ldots, MR(M_{t_k}, \mathcal{M}^\tau)\}$. We define $\mathcal{M}^\tau$ to be an answer set of P if, for each $M_i \in P$ we have that $\mathcal{NAT}(MR(M_i, \mathcal{M}^\tau)) = \mathcal{M}^\tau(M_i)$.

These definitions suggest a natural way to handle the semantics of a program P. Once the topological sort $\eta_1, \ldots, \eta_k$ of the modules is given, we can proceed as follows:

- the semantics of M_{η_0} is given, since it does not import any other module, and thus it can be set equal to $\mathcal{NAT}(M_{\eta_0})$;
- the semantics of M_{η_i} can be constructed by computing $\mathcal{NAT}(MR(M_{\eta_i}, \mathcal{M}^\tau))$.

3.2.2 Impure Programs

We say that a program is impure if it contains impure Prolog modules and/or it contains modules that are not based on logic programming. Let $P = \{M_{t_1}, \ldots, M_{t_k}\}$ be a program; we assume that $t_1, \ldots, t_k$ is already a topological sort of $graph(P)$. For the sake of simplicity, we consider impure programs under the following restrictions:

- the user executes the program and interacts with it through the module M_{t_k}, which is a Prolog (pure or impure) module; the interaction is driven by a Prolog goal.
- the impure predicates `assert` and `retract` are allowed to appear only in Prolog modules, and in particular, we will consider them only in the M_{t_k} module (though it is easy to relax this restriction).

Because of the non-logical nature of the impure predicates, we rely on an operational semantics to characterize the meaning of programs.

The *state* of a computation is given by a tuple $\langle G, \theta, P\rangle$ where G is a Prolog goal, θ is a substitution, and P is a program. The operational semantics is defined through a transition system between states $\langle G, \theta, P\rangle \mapsto \langle G', \theta', P'\rangle$ where[3]

- if $G = A \wedge Rest$, $\pi(A) \in def(M_{t_k})$, and $h :\!- body$ is a variant of a rule in M_{t_k} such that $A\theta\sigma = h\sigma$, then we have that $G' = body \wedge Rest$, $\theta' = \theta \circ \sigma$, and $P' = P$.
- if $G = t : A \wedge Rest$, $t \in imp(M_{t_k})$, and let σ be a ground substitution for $A\theta$ such that $A\theta\sigma$ is true in each model in $\mathcal{M}^\tau(M_t)$ (where $\mathcal{M}^\tau$ in the answer set of P then $G' = Rest$, $\theta' = \theta \circ \sigma$, and $P' = P$.
- if $G = t : model(t') \wedge Rest$, $t \in imp(M_{t_k})$, and σ is a substitution such that $t'\sigma$ is ground and $\tau(t'\sigma) \in \mathcal{M}^\tau(M_t)$ in answer set of P, then $G' = Rest$, $\theta' = \theta \circ \sigma$, and $P' = P$.
- if $G = t : A \wedge Rest$, there is $t' \in imp(M_{t_k})$ and a substitution σ such that $(t : A)\sigma$ is ground, $\tau(t\sigma) \in$

[3] Recall that, in the current proposal, we do not allow not to appear in Prolog modules.

$\mathcal{M}^\tau(M_{t'})$ in the answer set of P, and $A\sigma$ is true in $\tau(t\sigma)$, then $G' = Rest$, $\theta' = \theta \circ \sigma$, and $P' = P$.

- if $G = t : assert(Head, Body) \wedge Rest$, σ is such that $t\sigma \in imp(M_{t_k})$, then $G' = Rest$, $\theta' = \theta$, and $P' = (P \setminus \{M_{t\sigma}\}) \cup M'_{t\sigma}$ where $M'_{t\sigma} = M_{t\sigma} \cup \{Head :\!- Body\}$
- if $G = t : retract(Head, Body) \wedge Rest$, σ is a ground substitution such that $t\sigma \in imp(M_{t_k})$ and $(Head :\!- Body)\sigma \in M_{t\sigma}$, then $G' = Rest$, $\theta' = \theta$, and $P' = (P \setminus \{M_{t\sigma}\}) \cup M'_{t\sigma}$ where $M'_{t\sigma} = M_{t\sigma} \setminus \{(Head :\!- Body)\sigma\}$

Given a goal G and a program P with main module M_{t_k}, we say that θ is a solution of G if $\langle G, \epsilon, P\rangle \mapsto^* \langle true, \theta, P'\rangle$.

Example 3 *Let us continue with Example 1. Let us assume that a referee joe has suddenly requested to be excused from reviewing papers, and we wish to check whether without his/her presence we can still referee the papers (i.e., whether $\mathcal{M}($`papers`$) \neq \emptyset$). We can check this by trying to see the effect of removing the* `referee`*(joe):*

$$can\_remove(R) :\!- \texttt{expertise} : retract(referee(R)), \texttt{papers:model}(T).$$

3.2.3 Non-Logical Programming Modules

The ability to interact with non-logical programming modules is also considered vital to the integration of different rule-bases. In particular, we consider here the possibility of accessing ECA rule bases (e.g., the PR RuleML layer or some of the existing formalizations of ECA rules [23]). For the sake of discussion, we assume that the ECA rules drawn from the RuleML file are converted to CLIPS format [15] and handled by CLIPS. CLIPS has an elegant foreign interface that makes it easy to embed CLIPS into Prolog.

The notion of program is extended to allow some of the modules M_{t_i} to represent the CLIPS encoding of a RuleML module. Reasoning within an ECA module requires the module to be provided with the events necessary for the match-select-execute cycle of CLIPS; we envision these events to be provided by the other modules. Similarly, the content of the working memory at the end of the complete execution should be made available to other modules. We take the simple approach of embedding the CLIPS execution in a Prolog module (to be automatically generated during the process of compiling RuleML into CLIPS). The intuitive structure of such module is depicted in Fig. 3.

CLIPS facts are associated to Prolog facts; outside events are imported and converted into CLIPS facts (and added to the working memory of the CLIPS module). The result of the computation (i.e., the final content of the working memory) is retrieved by the embedding module, represented as Prolog facts, and publicized through the export declaration.

The semantics of an ECA module is represented by the content of the working memory when the match-select-execute cycle has reached a fixpoint. In particular, given an ECA module $\mathcal{E}$, let us introduce the following notation:

- given a ground Prolog fact p, we will denote with $\varphi(p)$ its representation in the ECA language (e.g., CLIPS fact); given an ECA fact q, we will denote with $\varphi^{-1}(q)$ its representation as a Prolog fact. These functions can be naturally extended to sets of atoms/facts.
- $r(\mathcal{E}, W)$ is the result of the match-selection process, i.e., the rule of $\mathcal{E}$ to selected during the match-selection process w.r.t. the working memory content W;
- $\delta(r, W)$ is the working memory resulting from executing rule r in the working memory W;
- The execution is described as follows: given the working memory W,

$$\begin{array}{lcl} E^0(\mathcal{E}, W) & = & W \\ E^{i+1}(\mathcal{E}, W) & = & \delta(r(\mathcal{E}, E^i(\mathcal{E}, W)), E^i(\mathcal{E}, W)) \end{array}$$

Let $S = \{i \mid E^i(\mathcal{E}, W) = E^{i+1}(\mathcal{E}, W)\}$. If $S \neq \emptyset$, then $E(\mathcal{E}, W) = E^{k+1}(\mathcal{E}, W)$, where $k = min(S)$; otherwise $E(\mathcal{E}, W) =$ *undef*.

Let $M_{t_1}, \ldots, M_{t_k}$ the modules imported by $\mathcal{E}$. Then we define $\mathcal{M}^\tau(\mathcal{E})$ as follows: $\varphi^{-1}(S) \in \mathcal{M}^\tau(\mathcal{E})$ iff $\exists M_1 \in \mathcal{M}^\tau(M_{t_1}) \ldots M_k \in \mathcal{M}^\tau(M_{t_k})$ s.t. $S = E(\mathcal{E}, \bigcup_{i=1}^k \varphi(M_i))$.

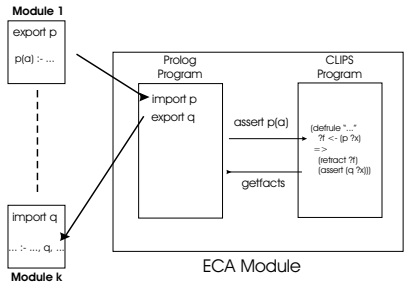

Figure 3. Embedding of Non-logical Modules

3.3. Framework Implementation

Part of the proposed framework has already been implemented as part of the ASP-Prolog system [14].

3.3.1 Logical Core Implementation

The implementation of the logical core is based on the combination of two logic programming systems: Ciao Prolog[4] and Smodels.[5]

[4] `clip.dia.fi.upm.es/Software/Ciao`
[5] `www.tcs.hut.fi/Software/smodels`

Ciao Prolog is full-fledged Prolog system, with a sophisticated module system, and designed to handle a variety of flavors of logic programming, including constraint logic programming (over reals and finite domains), fuzzy logic programming, and concurrent logic programming.

Smodels is a logic programming engine which supports computation of the well-founded and answer set semantics for NAF-datalog programs.

3.3.2 Preprocessing

The input to the preprocessor is composed of *(i)* the main Prolog module (Pr); *(ii)* a collection of Ciao Prolog modules ($m_1, m_2, \ldots, m_n$); *(iii)* a collection of ASP modules ($e_1, e_2, \ldots, e_m$). The output of the preprocessor is: a modified version of the main Prolog module (NP), a modified version of the other Prolog modules ($nm_1, nm_2, \ldots, nm_n$), and for each ASP module e_i the preprocessor creates a Ciao module (im_i) and a class definition (c_i).[6]

The transformation of the Prolog modules consists of a simple rewriting process, used to adapt the syntax of the interface constraints and make it compatible with Ciao Prolog's syntax. For example, the rules passed as arguments to `assert` and `retracts` have to be quoted to allow the peculiarities of ASP syntax to be accepted. The transformation of each ASP module leads to the creation of two entities that will be employed during the actual program execution: an *interface module* and a *model class*. These are described in the following subsections. The preprocessor will also automatically invoke the Ciao Prolog top-level and load all the appropriate modules for execution. The interaction with the user is the same as the standard Ciao Prolog top-level.

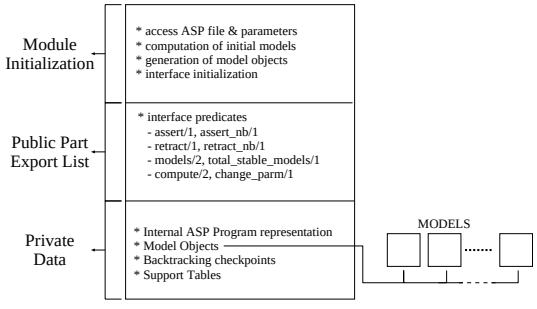

Figure 4. Structure of the Interface Module

3.3.3 Interface Modules

The preprocessor generates one interface module for each ASP module present in the original input program. The interface module is implemented as a standard Ciao Prolog module and it provides the client Prolog modules with the predicates used to access and manage the ASP module. The interface module is created for each ASP module by instantiating a generic module skeleton.

The overall structure of the interface module is illustrated in Figure 4. The module has an export list which includes all the predicates used to manipulate ASP modules (e.g., `assert`, `retract`, `model`) as well as all the predicates that are defined within the ASP module.

The definition of the various exported predicates (except for the predicates defined in the ASP module) is derived by instantiating a generic definition of each predicate. Each module has an initialization part, which is in charge of setting up the internal data structures and invoke the ASP solver (*Smodels*) for the first time on the ASP module. The result of the computation of the models will be encoded as a collection of *Model Objects* (see the description of the Model Classes in the next subsection). The module will maintain a number of internal data structures, including a representation of the ASP code, a representation of the parameters to be used for the computation of the answer sets (e.g., values of constants), a list containing the objects representing the models of the ASP module, a counter of the number of answer sets currently present, etc.

Each interface provides also a `timestamp` predicate, which is used to inform of the time at which the module's semantics have been last computed (recorded as a discrete system time); each interface module will recompute the local semantics whenever the timestamp of one of the imported modules changes. This allows the system to propagate the effect of changes (e.g., assert/retract) to all modules that depend on the modified one.

3.3.4 Model Classes

The preprocessor generates a Ciao class definition for each module. The objects obtained from the instantiation of such class will be used to represent the individual models of the module. In particular, for an ASP module we will have one instance for each answer set, while for a Prolog module we will have a single model.

Prolog and ASP modules can obtain reference to these objects (e.g., using the `model` predicate supplied by the interface module) and use them to directly query the content of one model. The definition of the class is obtained through a straightforward parsing of the export declaration of each module, to collect the names of the predicates defined in it; the class will provide a public method for each of the predicates present in the module's export list. The class defines also a public method `add/1` which is used by the interface module to initialize the content of the model.

Each model of an ASP module is stored in one instance of the class; the actual atoms representing the model are stored internally in the objects as facts of the form

[6] Ciao provides the ability to define classes and create class instances.

$s(\langle$`fact`$\rangle)$. In the case of Prolog modules, the class module will directly link to the corresponding predicate in the Prolog module—thus, allowing computation of the model on-the-fly.

3.3.5 Non-Logical Components

The non-logical components can be implemented through the high-level foreign interface of Ciao Prolog. As illustrated in Fig. 3, each non-logical component is implemented as a Prolog module, which embeds a CLIPS program. The Prolog module is automatically generated, during pre-processing, to collect from the CLIPS program the events appearing in the left-hand side of the rule and that are exogenous to the ECA rule base. URIs of events are employed to recognize what modules these events will come from and to generate the appropriate import declarations. Events asserted by ECA rules are automatically exported, and the corresponding export declarations generated.

Communication between CLIPS and Prolog is realized using

- the Ciao Prolog C interface, which allows a rich set of functionalities to convert between Prolog terms and C structures
- the CLIPS foreign interface, which provides C functions to access all the interface commands of CLIPS (e.g., `Assert` facts, `Load` rule bases, `Run` an execution, retrieve the content of the working memory `GetFactList`)

3.3.6 RuleML Specific Issues

As illustrated in Fig. 2, RuleML knowledge bases are retrieved and encoded as modules to support the reasoning activities. The translation process relies on the PiLLoW library (which supports HTTP protocol and basic XML manipulation) and the sophisticated XML library provided by Ciao Prolog (which allows XML parsing, URIs management, and even XPath queries).

The translation process is performed in two steps. During the first step, the RuleML document is parsed and converted into a Prolog XML representation (as a compound Prolog term). In the second phase, the Prolog XML representation is parsed and translated into logical rules and collected into a module.

The import component of the module is automatically derived by retrieving those atoms used in the program and linking (through URIs) to external components (e.g., used in the `rel` elements). By default, the export list will contain all the `rel` that appear as heads of rules/facts in the knowledge base.

Example 4 *Let us continue with the rule bases described in Example 1. Once the two modules have been derived, it is possible to write Ciao Prolog programs and queries to access the semantics of the two modules and reason about them. For example, if we wish to retrieve one possible assignment of referees and print it:*

$$:\!- import\ :\ \texttt{papers}, \texttt{expertise}.$$

$$\texttt{print\_assignment} :\!- \texttt{papers:model}(T),\ \texttt{findall}([P,R], \texttt{T:assign}(R,P), List),\ \texttt{print\_list}(List).$$

We can also discover whether there are referees that have not been assigned any paper in any possible assignment with the following rules (thus, the referee could be excused):

$$unassigned(R) :\!- \texttt{expertise:referee}(R),\ findall(R, (\texttt{papers:model}(T), T : assign(R,P)), []).$$

It is easy to implement soft constraints and/or preferences; for example, if `joe` *has a preference for reviewing paper* 6*, we can modify the* `print_assignment` *predicate as:*

$$\texttt{print\_assignment} :\!- \texttt{papers:model}(T),\ T : \texttt{assign}(joe, 6), !,\ \texttt{findall}([P,R], \texttt{T:assign}(R,P), List),\ \texttt{print\_list}(List).$$
$$\texttt{print\_assignment} :\!- \texttt{papers:model}(T),\ \texttt{findall}([P,R], \texttt{T:assign}(R,P), List),\ \texttt{print\_list}(List).$$

Similar approaches can be taken to handle other forms of preferences or conflicts of interest.

4. Related Work

The importance of developing languages and frameworks to integrate different aspects of semantic web reasoning has been highlighted in the literature. Most of the existing focus has been on integrating rule-based reasoning with ontology reasoning; e.g., the work in [16] describes a combination of reasoning about ontologies (encoded in OWL) with rule-based reasoning (with rules encoded in SWRL and processed by the Jess system). SweetProlog [20] relies on converting OWL ontologies to description logics, RuleML rules into Prolog, and using a Prolog system to integrate the two components.

Relatively few proposals have appeared in the literature dealing with the the high-level integration of different forms of logic programming reasoning (specifically, top-down goal-oriented Prolog and bottom-up answer set semantics). ASP-Prolog [14], on which the work described in this paper builds, is a system that provides Prolog programs with the ability to seamlessly access modules processed under answer set semantics. A simplified interface (between the Smodels system and XSB Prolog) has been described in [10]. Lower level interfaces between answer set systems (dlv and Smodels) and traditional imperative languages have been developed [22, 25, 9].

5. Conclusions

In this paper, we presented a framework aimed at supporting inter-operation between logic programming modules operating under different semantics (e.g., pure Prolog, answer set semantics). The objective is to reflect in this framework the content of RuleML documents and allow cooperative reasoning. RuleML documents are converted into logic programming modules and manipulated according to their natural semantics. A clearly defined module interface allows modules to exchange information, in the form of content of their model-theoretical semantics. We presented the syntax and semantics of the framework, as well as discussed a preliminary implementation of parts of this framework within the ASP-Prolog system.

The framework allows sophisticated forms of reasoning, including scoped inference, and the ability to use logic programming (either in the form of Prolog or in the form of Answer Set Programming) to reason about the semantics implied by distinct RuleML documents. For example, some of the initial RuleML specifications (e.g., [7]) provide the ability to associate labels and salience/priority to rules; this opens the doors to the possibility of encoding qualitative (e.g., overriding of rules) as well as quantitative (e.g., comparison of salience factors) preferences. This is very important, for example, in the context of ECA rules—where salience is a commonly used criteria to control order of execution. Our framework provides a natural way to handle qualitative and quantitative preferences, by allowing the development of modules that reason on the semantics of other modules—thus, allowing a module to apply preferences and filter out only the acceptable solutions (in a traditional generate&test approach).

As future work, we propose to complete the implementation of the framework and demonstrate it on real-world applications, with particular focus on applications on description and manipulation of bioinformatics web services.

References

[1] S. Anwar, C. Baral, L. Tari. A language for modular answer set programming: application to ACC tournament scheduling. *ASP*, 2005.

[2] J. Bailey, A. Poulovassilis, P.T. Wood. An Event Condition Action Language for XML. *WWW*, pp. 486–495, 2002.

[3] N. Bassiliades, G. Antoniou, I. Vlahavas. DR-DEVICE, a Defeasible Logic Reasoner for the Semantic Web. *Int. Journal on Semantic Web and Information Systems*, 2(1):1–41, 2006.

[4] C. Baral, J. Dzifcak, and H. Takahasi. Macros, macro calls, and use of ensembles in modular answer set programming. *ICLP*, Springer Verlag, 2006.

[5] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. In Scientific American, May 2001.

[6] H. Boley, B. Grosof, M. Sintek, S. Tabe, G. Wagner. RuleML Design, Version 0.8. 2002-09-03, 2002.

[7] H. Boley, S. Tabet, G. Wagner. Design Rationale of RuleML: a Markup Language for Semantic Web Rules. In *International Semantic Web Working Symposium (SWWS)*, 2001.

[8] F. Bry and P-L. Patranjan. Reacitivity on the Web: Paradigms and Applications of the Language XChange. *ACM SAC*, ACM Press, 2005.

[9] F. Calimeri and G. Ianni. External Sources of Computation for Answer Set Solvers. *LPNMR*, 2005.

[10] L. Castro et al. XASP: Answer Set Programming with XSB and Smodels. SUNY Stony Brook, 2002.

[11] H. Chan and B. Grosof. CommonRules. `http://www.alphaworks.ibm.com/tech/commonrules`, 1999.

[12] M. Dean. RuleML Experiments with GEDCOM. `http://www.daml.org/2001/02/gedcom-ruleml`, 2001.

[13] T. Eiter et al. Combining Answer Set Programming with Description Logics for the Semantic Web. *KR*, pp. 141–151, 2004.

[14] O. Elkhatib, E. Pontelli, T.C. Son. A Tool for Knowledge Base Integration and Querying. *AAAI Spring Symposium*, AAAI Press, 2006.

[15] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. Course Technology, 2004.

[16] C. Golbreich. Combining Rule and Ontology Reasoners for the Semantic Web. *RuleML*, Springer Verlag, pp. 6–22, 2004.

[17] D. Hirtle, H. Boley. The Modularization of RuleML. 2005-12-15, 2005. `http://www.ruleml.org/modularization`.

[18] I. Horrocks et al. SWRL: a Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission SUBM-SWRL-20040521, 2004. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521`.

[19] M. Kifer, J. de Bruijn, H. Boley, D. Fensel. A Realistic Architecture for the Semantic Web. *RuleML*, Springer Verlag, pp. 17–29, 2005.

[20] L. Laera, V. Tamma, T. Bench-Capon, G. Semeraro. SweetProlog: a System to Integrate Ontologies and Rules. *RuleML*, Springer Verlag, pp. 188–193, 2004.

[21] W. May, J.J. Alferes, R. Amador. Active Rules in the Semantic Web: Dealing with Language Heterogeneity. *RuleML*, Springer Verlag, pp. 30–44, 2005.

[22] F. Ricca. The DLV Java Wrapper. *APPIA-GULP-PRODE*, pp. 263–274, 2003.

[23] M. Seiriö, M. Berndtsson. Design and Implementation of an ECA Rule Markup Language. *RuleML*, Springer Verlag, pp. 98–112, 2005.

[24] B. Spencer. The Design of j-DREW, a Deductive Reasoning Engine for the Semantic Web. `http://www.cs.unb.ca/~bspencer/dagstuhl-jdrew.pdf`, 2002.

[25] T. Syrjanen. Lparse 1.0: User's Manual. Helsinki University of Technology, 1998.