

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

**Enrico Pontelli, Tran Cao Son, Omar Elkhatib**

*Department of Computer Science*

*New Mexico State University*

*epontell | tson | okhatib @cs.nmsu.edu*

**Chitta Baral**

*Department of Computer Science and Engineering*

*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 inter-operation through a well-defined module interface. In this paper we describe the basic syntax and semantics of the framework, and its preliminary implementation.

**Keywords:** Logic Programming, Rules, Inter-operation

### **1. Introduction**

One of the main goals of the Semantic Web initiative [6] is to extend the current Web technology to allow for the development of intelligent agents, which can *automatically* and *unambiguously* process the information available in millions of web pages. It has been recognized very early in the development of the Semantic Web [5] that *rules* are essential for the Web and for Semantic Web applications—e.g., in the description of semantic web services, business rules interchange for e-commerce applications.

The RuleML initiative is a response to the need of a shared rule markup language using XML markup, which has precisely defined semantics and enables 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 have been proposed. The initial design [7] 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 markup languages to encode ECA rules have appeared (e.g., [26, 29, 2, 9]), the work on ECA rules is still a “moving target”. The most recent modularized description of RuleML [22] reports this area (indicated as *PR RuleML*) as work in progress.

The derivation rules component of the RuleML initiative has originated a family of languages. Figure 1<sup>1</sup>, from [22], 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 [23]). Various sub-languages 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 [24], it is argued that any realistic architecture for the Semantic Web must be based on various independent but inter-operable languages, one of them being the logic programming language with negation-as-failure. The following example, a variation of the *reviewer selection* problem in [16], illustrates this point by proposing the encoding of two rule bases based on distinct flavors of RuleML.

### Example 1.1. (Reviewer Selection)

*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 sub-area 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`; these can be encoded by the following facts:*

*represents the fact that if  $X$  is a sub-area of  $Y$  and  $Y$  is a sub-area of  $Z$  then  $X$  is a sub-area 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. Observe 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 statements of the form*

<code>paper(#n)</code>	the paper is assigned the id number $n$
<code>kw(#n, kw)</code>	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 and paper #2 have been submitted, and they list `nmr` and `semantic_web` as respective keywords can be expressed by*

<sup>1</sup>[www.ruleml.org/modularization/ruleml\\_m12n\\_089\\_uml\\_05-06-01.png](http://www.ruleml.org/modularization/ruleml_m12n_089_uml_05-06-01.png).

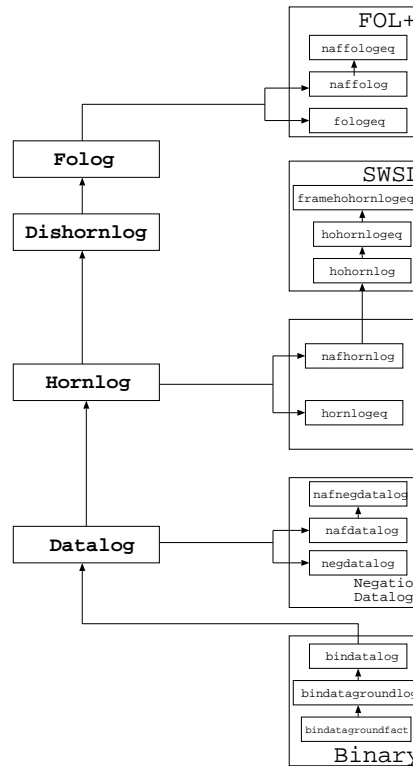


Figure 1. RuleML Derivation Rules

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

```
<fact> <head> <atom>
  <_opr> <rel> paper </rel> </_opr>
  <ind> 2 </ind>
</atom> </head> </fact>
```

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

```
<fact> <head> <atom>
  <_opr> <rel> kw </rel> </_opr>
  <ind> 2 </ind>
  <ind> semantic_web </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., the rules used for this purpose are shown in Fig. 3.

Observe that in representing the rules for assigning reviewers, we make use of the RuleML `nafdatalog` sub-language. We assume that the knowledge base is available from the location

<http://www.ruleml2006.org/papers.ruleml>.

which we will simplify to papers hereafter.

□



Figure 2. Reviewers rule base

## 1.1. The Problem

Example 1.1 is a simple illustration of a common situation arising in the context of using rule bases and, more in general, in the context of the general Semantic Web: the co-existence of different languages with different semantics and reasoning mechanisms and the need to integrate reasoning across these languages. The need for these languages and their interaction have been extensively discussed (e.g., [26, 24]) and it is at the foundation of the most recent work of the Rule Interchange Format working group.<sup>2</sup> It is also important to note that many of the sub-languages of RuleML have been implemented either through translators, e.g.,

- GEDCOM [14], which translates to XSB and JESS
- or independent engines, e.g.,
- j-DREW [30], a top-down engine for RuleML,
  - DR-Device [4], an engine supporting defeasible logic and both strong and default negation, and

<sup>2</sup><http://www.w3.org/2005/rules/>

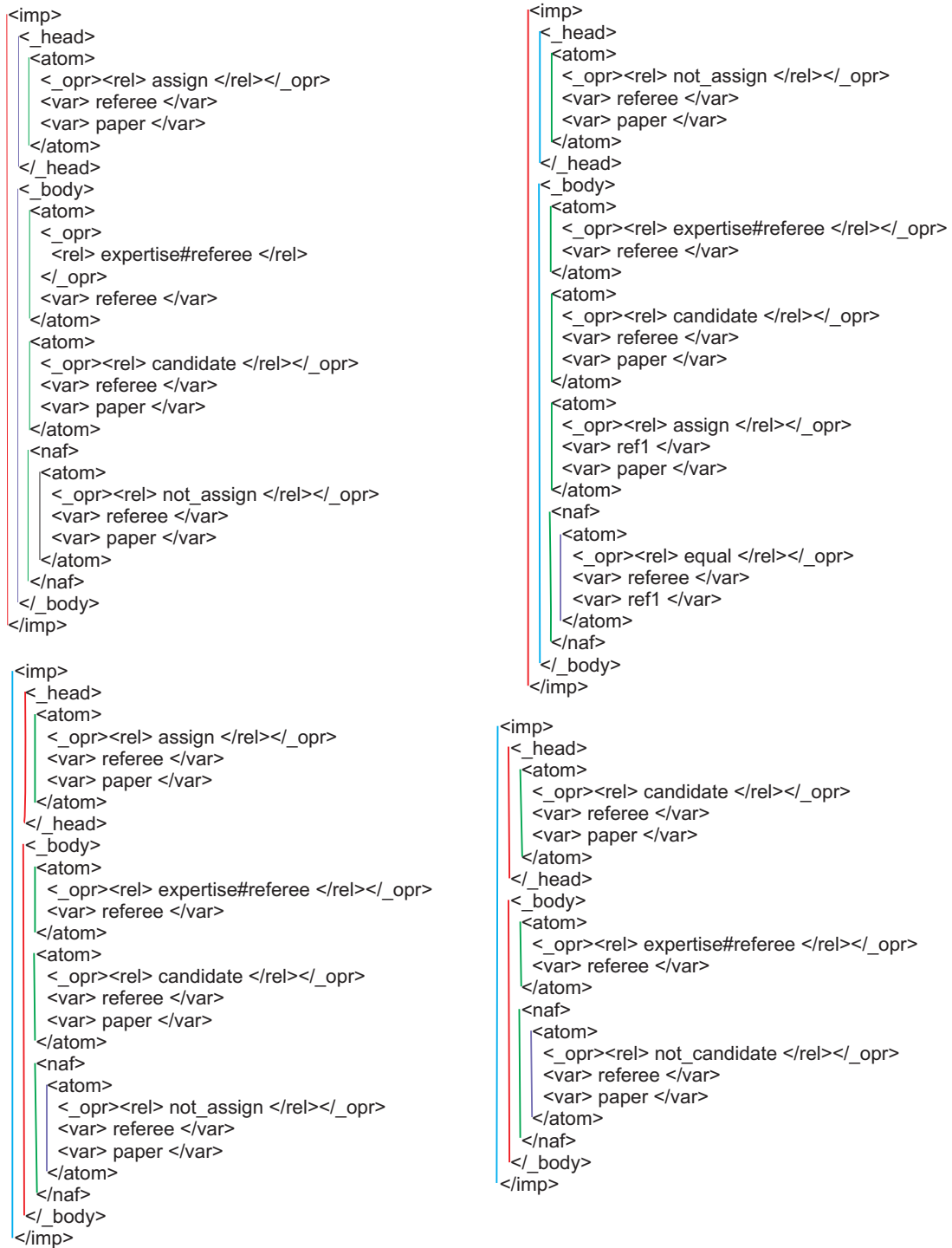


Figure 3. Rules to Assign Reviewers with Expertise

- CommonRules [13], a bottom-up engine for the Datalog sub-language.

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 inter-operate 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*: the capability of combining knowledge is essential for intelligent agents (e.g., this is necessary in the context of services composition). This requires
  - (a) The ability to exchange inference results between different knowledge bases (e.g., the interoperability problem between rules and OWL described in [24]);
  - (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 [24]).
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. Proposed Solution

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 the use of 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 provide foreign interfaces that allow declarative interfacing to other programming paradigms.

The idea is to combine the ASP-Prolog framework of [17]—which allows CIAO Prolog [10] programs to access and modify modules containing Answer Set Programming (ASP) code [27]—with the notation for modularization of Answer Set Programming of [3, 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 inter-operate.

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. 4. Most of the emphasis is on the use of Answer Set Programming to handle some of the sub-languages (e.g., datalog, ur-datalog, nafdatalog and negdatalog), even though the core framework will naturally support most of the languages (e.g., hornlog, hohornlog).

The problems mentioned in Sect. 1.1 are addressed by the proposed framework as follows:

- *Issue 1*: CIAO Prolog offers direct access to a collection of modules that support different forms of

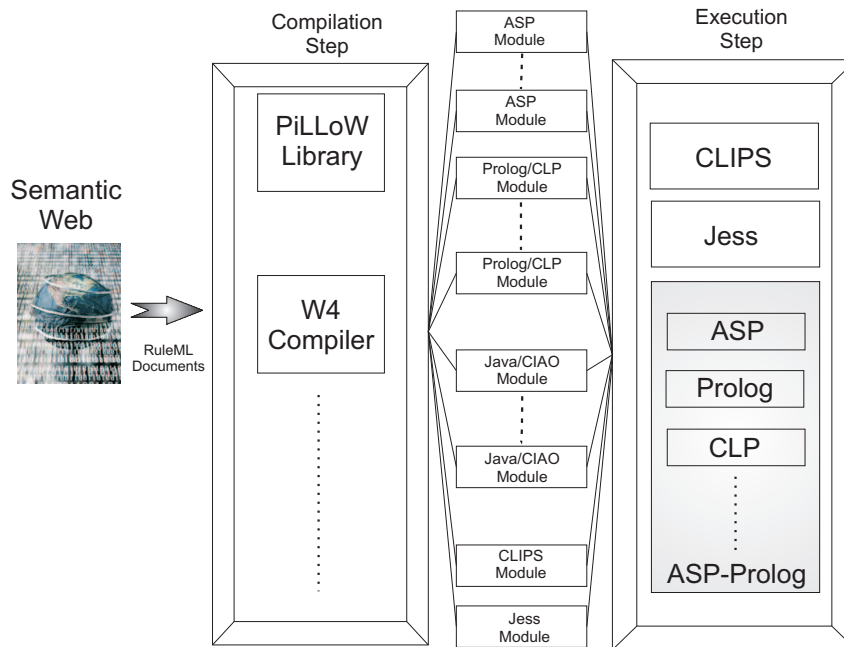


Figure 4. Overall Structure

logic programming reasoning, e.g., traditional Prolog, Constraint Logic Programming (over finite domains and reals), fuzzy Prolog, and a declarative ODBC interface. 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 packages. Furthermore, CIAO Prolog includes the *PiLLoW* library, a standardized Prolog library for Web Programming, which provides the framework with capabilities for Web access (e.g., management of URLs) and parsing of HTML and XML documents to Prolog terms.

Thus, we envision the core framework as the bridge between distinct execution models for different RuleML sub-languages.

- Issue 2:

- *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 [3] 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:* This aspect can be handled thanks to the combination, in ASP-Prolog and CIAO Prolog, of web access capabilities along with the full computational power of Prolog.

### 3. The Proposed Framework

#### 3.1. General Syntax

##### 3.1.1. 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.  $\Pi_u$  are called user-defined predicates while  $\Pi_d$  are called built-in predicates (and  $\Pi_u \cap \Pi_d = \emptyset$ ). We will assume that `assert`, `retract`, `model`, ... are elements of  $\Pi_d$ . 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  $f(t_1, \dots, t_n)$  where  $f \in \mathcal{F}$ ,  $ar(f) = n$ , and  $t_1, \dots, t_n$  are terms. We will say that a term  $t$  is ground if it contains no variables. We will denote with  $\mathcal{H}_P$  the Herbrand universe for this language—i.e., the set of all ground terms.

An atom is a formula of the form  $p(t_1, \dots, t_n)$ , where  $p \in \Pi$ ,  $ar(p) = n$ , and  $t_1, \dots, t_n$  are terms. The atom is ground if  $t_1, \dots, t_n$  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  $\Pi_d$ , 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/qualified 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)  $\ell$ , we denote with  $\pi(\ell)$  the predicate symbol used by the atom (qualified atom, negative literal)  $\ell$ .

A general rule is of the form

$$A :- B_1, \dots, B_k \quad (1)$$

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

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

- 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).
- ground datalog:** the  $B_i$  in rule (1) can only be atoms/qualified atoms, and the only terms allowed are constants.
- 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.
- datalog with negation:** the rules have the format as in case (a) but negative literals (*not*) are allowed in the body of the rule.
- 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  $\Xi$  rule (where  $\Xi$  is datalog, ground datalog, binary datalog, etc.) to denote a rule that meets the corresponding requirements. A  $\Xi$ -program is a collection of  $\Xi$  rules.

Given 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  $\Xi$ -program  $P$ , we denote with  $used(P) = \{t \mid \exists r \in P. t \in used(r)\}$ . We also introduce

$$def(P) = \{p \mid p \in \Pi_u, ar(p) = k, \exists r \in P. \exists t_1, \dots, t_k. head(r) = p(t_1, \dots, t_k)\}.$$

### 3.1.2. Module Structure

A module is composed of two parts: a module interface and a module body. A module interface has the form

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

where:

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

The body of a module is a  $\Xi$ -program for a given  $\Xi$ . In that case, we will say that the module is a  $\Xi$  module. Given a module named  $t$ , we identify with the export set of  $t$  (denoted by  $exp(t)$ ) the predicates  $q_1, \dots, q_k$  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  $P = \{M_{t_1}, \dots, M_{t_k}\}$  is a collection of modules named  $t_1, \dots, t_k$ . The *graph* of  $P$  ( $graph(P)$ ) is a graph  $(N, E)$  where the set of nodes  $N$  is  $\{t_1, \dots, t_k\}$  and  $(t_i, t_j) \in E$  iff  $t_i \in imp(t_j)$ . A program  $P = \{M_{t_1}, \dots, M_{t_k}\}$  is admissible if it satisfies the following properties:

- for each  $t_i$  we have that  $imp(t_i) \subseteq \{t_1, \dots, t_k\}$ ;
- 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 (e.g., as described in [3]). We omit this discussion due to lack of space.

**Example 3.1.** *The translation process mentioned in Section 3.3 and applied to the first knowledge base, will produce facts and rules of the form:*

```

:- module : expertise
:- export : referee/1, inArea/2, expert/2, ...

keyword(semantic_web).
...
inArea(semantic_web, ai).
...
expertIn(john, semantic_web).
...
referee(john).
...
inArea(K, K) :- keyword(K).
inArea(K, A) :- inArea(K, A1), inArea(A1, A).
...
expert(P, A) :- expertIn(P, A).
expert(P, A) :- inArea(A, A1), expert(P, A1).

```

The second knowledge base will be translated to a collection of facts and rules as shown in Figure 5. For later use, let us assume that we have the following:

- *REFS* is the set of referees consisting of *john*, *sue*, and *mike*;
- *KEYS* is the set of keywords consisting of the following keywords: *semantic\_web*, *mathematics*, *bioinformatics*, *hci*, *logic\_programming*, *nmr*, *ai*, *answering\_system*, and *planning*;
- *EXPS* is the set of pairs of the form  $(r, a)$  where  $r$  is a referee and  $a$  is an area encoding referee's expertise, which consists of the following information
  - *john* is an expert in *semantic\_web*, *logic\_programming*, and *hci*
  - *sue* is an expert in *hci*, *logic\_programming*, and *bioinformatics*
  - *mike* is an expert in *semantic\_web*, *answering\_system*, *mathematics*, and *planning*
 (e.g., we have  $(john, semantic\_web) \in EXPS$ .)

Let us assume that we have the following information about the papers:

- There are four papers 1, 2, 3, and 4.
- The keywords of the papers are given by
  - Paper 1: *semantic\_web* and *hci*
  - Paper 2: *bioinformatic* and *hci*
  - Paper 3: *logic\_programming*
  - Paper 4: *answering\_system*

```

:- module : papers
:- import : expertise
:- export : paper/1, ...

paper(1).
kw(1, nmr).
...
not_candidate(R, P) :- expertise:referee(R), kw(P, K),
                       expertise:inArea(K, A),
                       not expertise:expert(P, A).

candidate(R, P) :- expertise:referee(R),
                  not not_candidate(R, P).

assign(R, P) :- candidate(R, P),
                not not_assigned(R, P).

not_assigned(R, P) :- candidate(R1, P),
                      assign(R1, P), R1 ≠ R.

done(P) :- paper(P),
           expertise:referee(R), assign(R, P).

false :- assign(R, P), assign(R, P1), P ≠ P1.
false :- paper(P), not done(P).

```

Figure 5. Translation of Second Knowledge Base

The program for assigning referees can be given by

$$P_{ref} = \{expertise, papers\}$$

It is easy to see that `expertise` is a datalog module while `papers` is a nafdatalog module:  $\square$

## 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  $\tau$  is an 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 function  $\tau$  is fixed.

Given a program  $P = \{M_{t_1}, \dots, M_{t_k}\}$ , the acyclic nature of  $graph(P)$  guarantees the ability to construct a topological sort of  $\{t_1, \dots, t_k\}$ , say  $\eta_1, \dots, \eta_k$  such that if  $(\eta_i, \eta_j)$  is an edge of the graph, then  $i < j$ . For the program  $P_{ref}$ , the topological order is given by `expertise` followed by `papers`.

Given the program  $P$  and a topological sorting of the modules  $\eta_1, \dots, \eta_k$ , 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  $\Xi$ -program  $T$ , not containing any qualified atoms and not containing any occurrence of predicates from  $\Pi_d$ , 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 (i.e., definite logic program without extra-logical predicates such as `assert` and `retract`), then  $\mathcal{NAT}(T)$  contains the least Herbrand model of  $T$ .

This suggests a natural way to handle the semantics  $\mathcal{M}^\tau$  of a program  $P$ .  $\mathcal{M}^\tau$  is a mapping of the form  $\mathcal{M}^\tau : P \mapsto 2^{\mathcal{B}_P}$ . Once the topological sort  $\eta_1, \dots, \eta_k$  of the modules is given, we can construct  $\mathcal{M}^\tau$  as follows:

- the semantics of  $M_{\eta_1}$  is given, since it does not import any other modules, and

$$\mathcal{M}^\tau(M_{\eta_1}) = \mathcal{NAT}(M_{\eta_1})$$

- the semantics of  $M_{\eta_i}$  can be constructed by computing the natural semantics of a “reduct” of the module,

$$\mathcal{M}^\tau(M_{\eta_i}) = \mathcal{NAT}(MR(M_{\eta_i}, \mathcal{M}^\tau))$$

as defined next.

Let us consider a module  $M_{\eta_i}$  of  $P$ . Then:

- If  $t : A$  is a ground qualified atom and  $t \in \text{imp}(M_{\eta_i})$ , then

$$M_{\eta_i} \models_{\mathcal{M}^\tau} t : A \quad \text{iff for each model } M \in \mathcal{M}^\tau(M_t) \text{ we have that } M \models A$$

- if  $t : A$  is a ground qualified atom and  $t \notin \text{imp}(M_{\eta_i})$ , then  $M_{\eta_i} \models_{\mathcal{M}^\tau} t : A$  iff there exist  $t_i \in \text{imp}(M_{\eta_i})$ ,  $M \in \mathcal{M}^\tau(M_{t_i})$  such that  $\tau(t) = M$  and  $M \models A$ .
- $M_{\eta_i} \models_{\mathcal{M}^\tau} t : \text{model}(t')$  iff  $t \in \text{imp}(M_{\eta_i})$  and  $\tau(t') \in \mathcal{M}^\tau(M_t)$ .
- If *not*  $t : A$  is a ground qualified literal and  $t \in \text{imp}(M_{\eta_i})$ , then

$$M_{\eta_i} \models_{\mathcal{M}^\tau} \text{not } t : A \quad \text{iff } M_{\eta_i} \not\models_{\mathcal{M}^\tau} t : A$$

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  $x$  such that  $M_i \not\models_{\mathcal{M}^\tau} x$ ;
- remove from the remaining rules all occurrences of qualified elements.

One can easily see that  $MR(M_i, \mathcal{M}^\tau)$  is a program without qualified atoms whose semantics is defined by  $\mathcal{NAT}$  (as assumed). This allows us to set

$$\mathcal{M}^\tau((M_{\eta_i})) = \mathcal{NAT}(MR(M_{\eta_i}, \mathcal{M}^\tau))$$

From now on, we will denote with  $\mathcal{M}_P^\tau$  the semantics of a program  $P$ .

**Example 3.2.** Let consider the program  $P_{ref}$  from Example 3.1. We have the following:

- `expertise` does not contain any qualified atoms and is a datalog module. Thus, its semantics, is given by its least Herbrand model,  $M_0$ , i.e.,  $\mathcal{NAT}(\text{expertise}) = \{M_0\}$ .  $M_0$  contains the following atoms:

- $\text{keyword}(K)$  for  $K \in KEYS$ ,
- $\text{inArea}(K, K)$  for  $K \in KEYS$  and  $\text{inArea}(\text{semantic\_web}, ai)$ ,
- $\text{referee}(R)$  for  $R \in REFS$ ,
- $\text{expertIn}(R, K)$  for  $(R, K) \in EXPS$ , and
- $\text{expert}(R, K)$  for  $(R, K) \in EXPS$ ,  $\text{expert}(\text{john}, ai)$ , and  $\text{expert}(\text{mike}, ai)$ .
- $\text{papers}$  is a nafdatalog module. It imports predicates from the module  $\text{expertise}$ . The model reduct of this module is the program containing the following rules:
  - The set of facts of the form  $\text{paper}(i)$  and  $\text{kw}(i, k)$  (e.g.,  $\text{paper}(1)$  and  $\text{kw}(1, \text{semantic\_web})$ ) which encode the keywords associated to each paper, as described in Example 3.1.
  - The rules defining  $\text{not\_candidate}$  will be simplified to ground rules of the following form

$$\text{not\_candidate}(r, p) :- \text{kw}(p, k)$$

where  $p, k, r$  are the paper number, the keyword, and the referee, such that  $r$  is not an expert in any areas relevant to  $k$ . (Here, we require that a candidate must be an expert in areas that cover all the keywords of the paper.) For example, both  $\text{sue}$  and  $\text{mike}$  are not candidates for the paper 1:

$$\text{not\_candidate}(\text{sue}, 1) :- \text{kw}(1, \text{semantic\_web})$$

$$\text{not\_candidate}(\text{mike}, 1) :- \text{kw}(1, \text{hci})$$

- The rules defining candidates are reduced to the following set of rules of the form

$$\text{candidate}(r, p) :- \text{not not\_candidate}(r, p)$$

where  $r$  is a referee and  $p$  is a paper number. For instance, we have

$$\text{candidate}(\text{mike}, 1) :- \text{not not\_candidate}(\text{mike}, 1)$$

- The rules defining  $\text{done}$  are obtained by replacing the variable  $R$  in  $\text{assign}(R, P)$  with the name of one of the referees.
- Other rules do not change.

We have that  $MR(\text{papers}, \mathcal{M}^\tau)$  is a nafdatalog program which has two answer sets. In both answer sets, paper 1 is assigned to  $\text{john}$ , paper 2 is assigned to  $\text{sue}$ , and paper 4 is assigned to  $\text{mike}$ . One of the answer sets assigns paper 3 to  $\text{john}$  while the other assigns it to  $\text{sue}$ .  $\square$

### 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  $P = \{M_{t_1}, \dots, M_{t_k}\}$  be a program; we assume that  $t_1, \dots, t_k$  is already a topological sort of  $\text{graph}(P)$ . 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,  $\theta$  is a substitution, and  $P$  is a program. The operational semantics is defined through a state transition system  $\langle G, \theta, P \rangle \mapsto \langle G', \theta', P' \rangle$  where<sup>3</sup>

- 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  $\sigma$  be a ground substitution for  $A\theta$  such that  $A\theta\sigma$  is true in each model in  $\mathcal{M}_P^\tau(M_t)$  then  $G' = Rest$ ,  $\theta' = \theta \circ \sigma$ , and  $P' = P$ .
- If  $G = t : model(t') \wedge Rest$ ,  $t \in imp(M_{t_k})$ , and  $\sigma$  is a substitution such that  $t'\sigma$  is ground and  $\tau(t'\sigma) \in \mathcal{M}_P^\tau(M_t)$  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  $\sigma$  such that  $(t : A)\sigma$  is ground,  $\tau(t\sigma) \in \mathcal{M}_P^\tau(M_{t'})$  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  $G = t : assert(Head, Body) \wedge Rest$ ,  $\sigma$  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  $G = t : retract(Head, Body) \wedge Rest$ ,  $\sigma$  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  $\theta$  is a solution of  $G$  if  $\langle G, \epsilon, P \rangle \mapsto^* \langle true, \theta, P' \rangle$ .

**Example 3.3.** *Let us continue with Example 3.2. Let us assume that a referee john 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  $\mathcal{M}_P^\tau(papers) \neq \emptyset$  after the change). This can be done by introducing a Prolog module consisting of the following rules:*

$$\begin{aligned}
 & :- \text{ module : } \text{remove\_refs} \\
 & :- \text{ import : } \text{expertise, papers} \\
 \text{can\_remove}(R) & :- \text{ expertise : } \text{retract(referee}(R)), \text{ papers : } \text{model}(T). \quad (*)
 \end{aligned}$$

Let  $P = P_{ref} \cup \{\text{remove\_refs}\}$ . We can solve the question posed above—i.e., the effect of removing the fact `referee(john)`—by issuing the Prolog query

$$? - \text{can\_remove}(\text{john}).$$

The execution of this goal  $G = \text{can\_remove}(\text{john})$  will be considered with respect to the program  $P$  above, and will go through the following steps:

<sup>3</sup>Recall that, in the current proposal, we do not allow *not* to appear in Prolog modules.

- The initial state of the computation is  $\langle G, \theta, P \rangle$ , where
  - $G$  is the goal `can_remove(john)`
  - $\theta$  is the empty substitution  $\epsilon$
  - $P$  is the original program  $P_{ref} \cup \{remove\_refs\}$
- $G$  can be unified with the head of the rule (\*) in `remove_refs` and the step produces a new state  $\langle G', \theta', P' \rangle$  where
  - $G'$  is the goal `expertise : retract(referee(john)), papers : model(T)`
  - $\theta'$  is the substitution  $\{R/john\}$
  - $P'$  is equal to the original program  $P$ .
- The subgoal `expertise : retract(referee(john))` is executed, leading to a state  $\langle G'', \theta'', P'' \rangle$ , where
  - $G''$  is the goal `papers : model(T)`
  - $\theta''$  is equal to  $\theta'$
  - The program  $P''$  consists of the following modules:
    - `remove_refs` and `papers` are unchanged; and
    - A new module  $Q = expertise \setminus \{referee(john)\}$  which is obtained from `expertise` by removing the fact `referee(john)`.
- The goal `papers : model(T)` is evaluated. This requires the computation of  $\mathcal{M}_{P''}^T(\text{papers})$ . It is easy to see that without `referee(john)` as a fact in `expertise`,  $\mathcal{M}_{P''}^T(\text{papers}) = \emptyset$ . This means that the goal fails.

Backtracking does not help since the first two evaluations in this computation are deterministic. This means that the goal `? – can_remove(john)` fails, i.e., without the service of `john` there are some papers that cannot be assigned.  $\square$

### 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 [29]). For the sake of discussion, we assume that the ECA rules drawn from the RuleML file are converted to CLIPS format [18] 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. 6.

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  $W$ ,

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

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  $E(\mathcal{E}, W) = \text{undef}$ .

Let  $M_{t_1}, \dots, M_{t_k}$  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  $\exists M_1 \in \mathcal{M}^\tau(M_{t_1}) \dots M_k \in \mathcal{M}^\tau(M_{t_k})$  s.t.  $S = E(\mathcal{E}, \bigcup_{i=1}^k \varphi(M_i))$ .

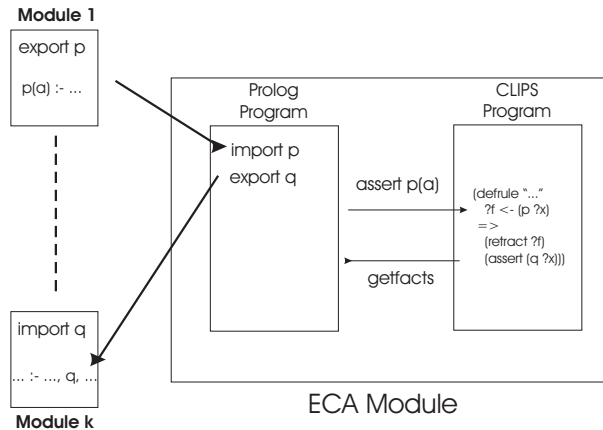


Figure 6. Embedding of Non-logical Modules

### 3.3. Framework Implementation

Most of the proposed framework has already been implemented as part of the ASP-Prolog system [17].



### 3.3.1. Logical Core Implementation

The implementation of the logical core is based on the combination of two logic programming systems: CIAO Prolog<sup>4</sup> and Smodels.<sup>5</sup>

CIAO Prolog is a 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.

The overall structure of the implementation is depicted in Figure 7. The system is composed of two parts, a *preprocessor* and the actual CIAO Prolog system.

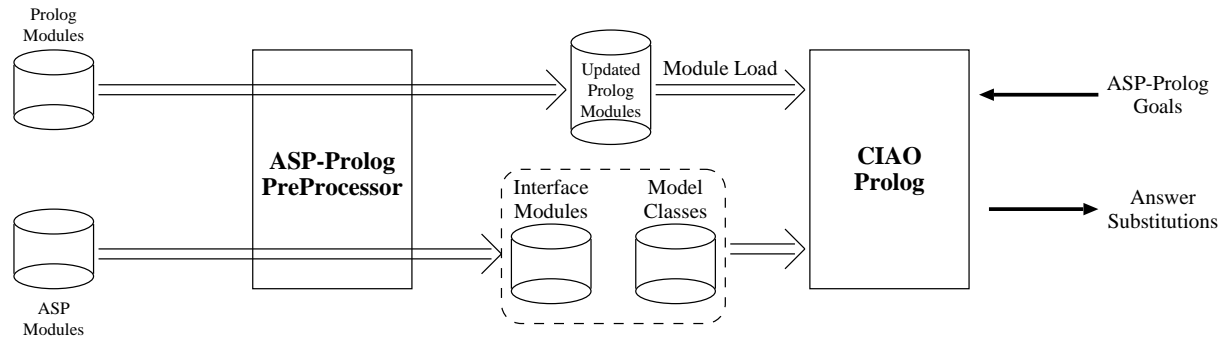


Figure 7. Overall Structure of ASP-Prolog Implementation

### 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 ( $m_1, m_2, \dots, m_n$ );
- (iii) a collection of ASP modules ( $e_1, e_2, \dots, e_m$ ).

The output of the preprocessor is: a modified version of the main Prolog module ( $NP$ ), a modified version of the other Prolog modules ( $nm_1, nm_2, \dots, nm_n$ ), and for each ASP module  $e_i$  the preprocessor creates a CIAO module ( $im_i$ ) and a class definition ( $c_i$ ).<sup>6</sup>

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

<sup>4</sup>[clip.dia.fi.upm.es/Software/Ciao](http://clip.dia.fi.upm.es/Software/Ciao)

<sup>5</sup>[www.tcs.hut.fi/Software/smodels](http://www.tcs.hut.fi/Software/smodels)

<sup>6</sup>CIAO provides the ability to define classes and create class instances.

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.

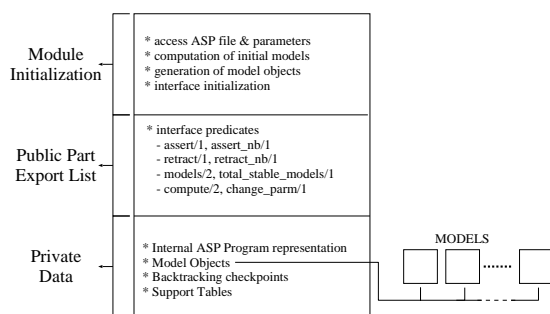


Figure 8. Structure of the Interface Module

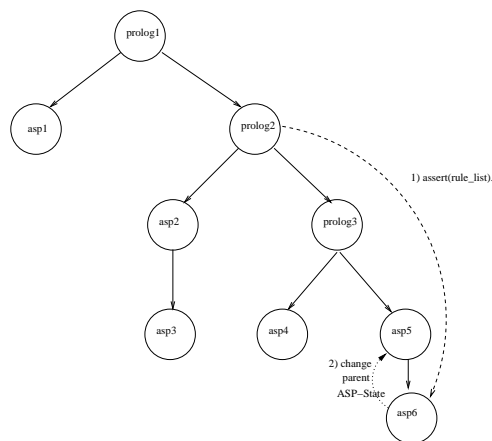


Figure 9. Parents change their ASP-state when  $asp_i$  is changed

### 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 8. 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.

The preprocessor creates a graph-module structure which represents the call hierarchy of modules. If a Prolog/ASP module  $u_p$  is calling another Prolog/ASP module  $u_c$ , then  $u_p$  is called the parent module and  $u_c$  is one of its children. The tree-module structure is stored in each interface module.

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.

Finally, the pre-processing will add some code to handle the three types of module qualification. E.g., assume the qualification  $(\alpha:t)$  appears in a rule ( $r$ ), it will be changed as follows: (i) if  $\alpha$  is an ASP module child, and  $t \neq model(Q)$  (where  $Q$  is a variable), then  $(\alpha:t)$  will be changed into  $(\alpha:getSkepticalValue(t))$ . (ii) if  $\alpha$  is a model name, then it will be changed into  $(\alpha:atom\_value(t, 1))$ ; this returns the values for  $t$  that are true in the model  $\alpha$ . (iii) otherwise  $(\alpha:t)$  will not be changed.

### 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  $s(\text{fact})$ . 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.

To facilitate computation, the preprocessor generates an additional module for each ASP module, called `skeptical module`. The module is instantiated to a collection of facts representing the elements that have the same truth value in all answer sets of the corresponding ASP module.

### 3.3.5. Further Implementation Details

**ASP-State:** Every ASP-module has a set of models, atoms, and skeptical model called *ASP-states*. A stack is used to store the ASP-states which will be used for backtracking. Any change in the ASP-state of an ASP module  $m$ , requires that all the ASP ancestor modules of  $m$  change their own ASP-states. This can be done with the help of the tree-module structure. Figure 9 illustrates how a tree-module structure is used in backtracking. Assume that the module hierarchy is represented as the tree. First `prolog2` asserts a rule in `asp6`, which causes `asp6` to change its ASP-state; in turn, this will cause a change of the ASP-state of `asp5`, which is a parent of `asp6`.

**Partial Grounder:** In order for the Smodels system to ground the ASP-modules, all qualification of the form  $(\alpha:t)$  have to be computed and stored as new facts with a newly created ASP-variable. Therefore, if we have a positive qualification literal  $(\alpha:t)$ , then the partial grounder will evaluate the predicate  $t$  in module  $\alpha$ . If  $\alpha$  is a Prolog module, then the builtin predicate `findall` is used to compute all possible answers to the goal  $?- t$ . If  $\alpha$  is an ASP-module, then the predicate  $t$  is evaluated against the skeptical

answers of the ASP-module  $\alpha$ . If  $\alpha$  is an ASP-model name, then predicate  $t$  is computed against the corresponding answer set named  $\alpha$ . After that, all ground values of  $t$  are added as facts into the ASP-module.

**Example 3.4.** Assume a rule in an ASP-module (in our system):

```
1 #import(pr, 'prolog1.pl').
2 p :- ..., pr:p(X,Y), ...
```

The partial grounder will find all ground values of  $p(X,Y)$  by executing:

```
findall((X,Y),p(X,Y),L).
```

Assume  $L = [(1,1), (2,2)]$ , then the second rule is changed to:

```
p :- ..., asprolog0(X,Y), ...
asprolog0(1,1).
asprolog0(2,2).
```

where `asprolog0` is a new predicate. A symbol table entry is needed to record that `asprolog0/2` is associated with `pr:p(X,Y)`.

The same rewriting rules are needed for negative qualification literals (*not*  $\alpha:t$ ), domain qualification literals, cardinality qualification literals, and weight qualification literals.

### 3.3.6. Non-Logical Components

The non-logical components can be implemented through the high-level foreign interface of CIAO Prolog. As illustrated in Fig. 6, 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).

### 3.3.7. RuleML Specific Issues

As illustrated in Fig. 4, 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 3.5.** Let us continue with the rule bases described in Example 1.1. 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 : papers, expertise.

print_assignment :- papers:model(T), findall([P, R], T:assign(R, P), List),
                  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) :- expertise:referee(R),
                 findall(R, (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:

```
print_assignment :- papers:model(T),
                   T:assign(joe, 6),!,
                   findall([P, R], T:assign(R, P), List),
                   print_list(List).

print_assignment :- papers:model(T),
                   findall([P, R], T:assign(R, P), List),
                   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. Two relevant contributions in this field are represented by the work of Goldbreich [20] and Laera et al. [25]. In [20], the author 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). Differently from our framework, the system is not based on logic programming (relying on Java's Jess platform) and limited to SWRL's rule format.

A wider-breadth approach, and closer to the spirit of ASP-Prolog, is SweetProlog [25]. SweetProlog offers a platform for integrating rules and OWL ontologies build using logic programming. Rules are encoded using OWLRuleML, which represents an OWL ontology for the RuleML dialect captured by the Courteous Logic Programming scheme [21]. Both rules and OWL ontologies are mapped to Prolog (specifically, the Java-Internet Prolog engine) for interrogation and reasoning. This project has a greater emphasis on integration between a fixed rule language (the fragment of RuleML captured by OWLRuleML) and OWL ontologies, providing an orthogonal contribution to the integration problem.

A remarkable direction has been explored in the context of the DLV project; *d1vhex* [15] is an extension of DLV which

- allows answer set programs to invoke external source of computation—i.e., truth value of selected predicates can be achieved through external deduction systems, e.g., accessing RDF statements;
- allows answer set programs to contain higher-order atoms.

This language has been motivated by similar reasons as those in this paper, to support semantic web reasoning [16, 15], with particular emphasis on integration of ontologies.

Relatively few proposals have appeared in the literature dealing with 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 [17], 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 [12].

Lower level interfaces between answer set systems (*dlv* and Smodels) and traditional imperative languages have been developed [28, 31, 11].

## 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., [8]) 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. The proposed framework is also expected to have a significant role in facilitate the deployment of and reasoning about rule-bases constructed according to the guidelines of the Rule Interchange Format (RIF) initiative [19]. Recent developments in RIF have highlighted the importance of being able to merge rule sets, and the requirement of supporting standard ways to characterize rule sets dialects. The proposed framework could easily avail of such standardized identification of dialects to guide the automated translation of RIF rule sets to ASP-Prolog modules.

## Acknowledgments

The authors wish to thank the anonymous reviewers of a previous version of this paper for their comments and suggestions. The research has been partially supported by NSF grants 0544373, 0454066, 0420407, 0220590, 0412000, and grants N61339-06-C-0143 and N61339-06-C-0123.

## References

- [1] Anwar, S., Baral, C., Tari, L.: A language for modular ASP: application to ACC tournament scheduling, *ASP Workshop*, 2005.
- [2] Bailey, J., Poulouvassilis, A., Wood., P.: An Event Condition Action Language for XML, *World Wide Web Conference*, 2002.
- [3] Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls, and use of ensembles in modular answer set programming, *International Conference on Logic Programming*, Springer Verlag, 2006.
- [4] Bassiliades, N., Antoniou, G., Vlahavas, I.: DR-DEVICE, a Defeasible Logic Reasoner for the Semantic Web, *Int. Journal on Semantic Web and Information Systems*, 2(1), 2006, 1–41.
- [5] Berners-Lee, T.: Design Issues – Architectural and Philosophical Points, Personal Notes, [www.w3.org/DesignIssues](http://www.w3.org/DesignIssues).
- [6] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web, *Scientific American*, 284(5), 2001, 34–43.
- [7] Boley, H., Grosz, B., Sintek, M., Tabe, S., Wagner, G.: RuleML Design, Version 0.8, [www.ruleml.org/indesign.html](http://www.ruleml.org/indesign.html), September 2002.
- [8] Boley, H., Tabet, S., Wagner, G.: Design Rationale of RuleML: a Markup Language for Semantic Web Rules, *International Semantic Web Working Symposium (SWWS)*, Stanford University, 2001.
- [9] Bry, F., Patranjan, P.-L.: Reactivity on the Web: Paradigms and Applications of the Language XChange, *Symposium on Applied Computing*, ACM Press, 2005.
- [10] Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: *The Ciao Prolog System. Reference Manual*, Technical Report CLIP3/97.1, Technical University of Madrid (UPM), 1997.
- [11] Calimeri, F., Ianni, G.: External Sources of Computation for Answer Set Solvers, *Logic Programming and Non-Monotonic Reasoning*, Springer Verlag, 2005.
- [12] Castro, L., Swift, T., Warren, D.: *XASP: Answer Set Programming with XSB and Smodels*, Technical report, SUNY Stony Brook, [xsb.sourceforge.net/manual2/node149.html](http://xsb.sourceforge.net/manual2/node149.html), 2002.

- [13] Chan, H., Grosf, B.: CommonRules, [www.alphaworks.ibm.com/tech/commonrules](http://www.alphaworks.ibm.com/tech/commonrules), 1999.
- [14] Dean, M.: RuleML Experiments with GEDCOM, [www.daml.org/2001/02/gedcom-ruleml](http://www.daml.org/2001/02/gedcom-ruleml), 2001.
- [15] Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning, *European Semantic Web Conference*, Springer Verlag, 2006.
- [16] Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining ASP with Description Logics for the Semantic Web, *Principles of Knowledge Representation and Reasoning*, AAAI Press, 2004.
- [17] Elkhatab, O., Pontelli, E., Son, T.: A Tool for Knowledge Base Integration and Querying, *AAAI Spring Symposium*, number SS-06-05, AAAI Press, 2006.
- [18] Giarratano, J., Riley, G.: *Expert Systems: Principles and Programming*, Course Technology, 2004.
- [19] Ginsberg, A., Hirtle, D., McCabe, F., Patranjan, P.-L.: *RIF Use Cases and Requirements*, Technical Report WD-rif-ucr-20060710, W3C RIF Working Group, 2006.
- [20] Golbreich, C.: Combining Rule and Ontology Reasoners for the Semantic Web, *RuleML*, Springer Verlag, 2004.
- [21] Grosf, B.: *A Courteous Compiler from Generalized Courteous Logic Programs to Ordinary Logic Programs*, Technical report, IBM T.J. Watson Research Center, 1999.
- [22] Hirtle, D., Boley, H.: The Modularization of RuleML, [www.ruleml.org/modularization](http://www.ruleml.org/modularization), 2005.
- [23] Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosf, B., Dean, M.: *SWRL: a Semantic Web Rule Language Combining OWL and RuleML*, Technical Report SUBM-SWRL-20040521, W3C Member Submission, 2004.
- [24] Kifer, M., de Bruijn, J., Boley, H., Fensel, D.: A Realistic Architecture for the Semantic Web, *RuleML*, Springer Verlag, 2005.
- [25] Laera, L., Tamma, V., Bench-Capon, T., Semeraro, G.: SweetProlog: a System to Integrate Ontologies and Rules, *RuleML*, Springer Verlag, 2004.
- [26] May, W., Alferes, J., Amador, R.: Active Rules in the Semantic Web: Dealing with Language Heterogeneity, *RuleML*, Springer Verlag, 2005.
- [27] Niemela, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm, *Annals of Mathematics and Artificial Intelligence*, **25**(3–4), 1999, 241–273.
- [28] Ricca, F.: The DLV Java Wrapper, *Italian Congress on Computational Logic (CILC)*, 2003.
- [29] Seiriö, M., Berndtsson, M.: Design and Implementation of an ECA Rule Markup Language, *RuleML*, Springer Verlag, 2005.
- [30] Spencer, B.: The Design of j-DREW, a Deductive Reasoning Engine for the Semantic Web, *Workshop on Logic-based Program Synthesis and Transformation*, number TR CLIP 4/02.0, Univ. Pol. Madrid, 2002.
- [31] Syrjanen, T.: *Lparse 1.0: User's Manual*, Helsinki University of Technology, 1998.