The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

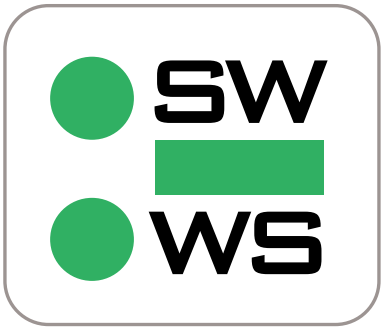
August 10 - 22, 2006

ICLP'06 Workshop

ALPSWS2006: Applications of Logic Programming in the Semantic Web and Semantic Web Services

August 16th, 2006

Proceedings



Editors:

A. Polleres, S. Decker, G. Gupta, and J. de Bruijn

Preface

The advent of the Semantic Web promises machine readable semantics and a machine-processable next Generation of the Web. The first step in this direction is the annotation of static data on the Web by machine processable information about knowledge and its structure by means of Ontologies. The next step in this direction is the annotation of dynamic applications and services invocable over the Web in order to facilitate automation of discovery, selection and composition of semantically described services and data sources on the Web by intelligent methods, which is called Semantic Web Services.

This volume contains the papers presented at the international workshop on *Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS2006)* held on August 16th, 2006 in Seattle, Washington as part of the 22$^{\text{nd}}$ International Conference on Logic Programming (ICLP06).

Many previous workshops and conferences were dedicated to these promising areas mostly with generic topics. With the ALPSWS2006 workshop we had a slighlty different goal. Rather that bringing together people from a widespread variety of research fields with different understandings of the topic we wanted to focus on the various applications areas and approaches in this area from declarative logic programming (LP).

The idea was to get a snapshot of the state of the work related to applications of LP to Semantic Web and Semantic Web Services with the following main objective major benefits:

- Bringing together people from different sub-disciplines of LP and focus on technological solutions and applications from LP to the problems of the Web.
- Promoting further research in this interesting application field.

Overall, there were 13 submissions. Each submission was reviewed by at least 3 programme committee members. The committee decided to accept 7 papers (6 full-length papers and one short paper) for presentations. 5 of the papers which could not be considered for full presentations, resubmitted extended abstracts which have been presented as posters during the workshop and are also included in this volume.

August 2006 Axel Polleres

Conference Organization

Organizing Committee

Axel Polleres
Stefan Decker
Gopal Gupta
Jos de Bruijn

Programme Committee

Juergen Angele
Chitta Baral
Robert Baumgartner
Leopoldo Bertossi
Franois Bry
Thomas Eiter
Pascal Hitzler
Giovambattista Ianni
Sheila McIlraith
David Pearce
Sebastian Schaffert
Andy Seaborne
Hans Tompits
Gerd Wagner
Adrian Walker
Guizhen Yang

Additional Reviewers

Ajay Bansal
Adrian Giurca
Andreas Harth
Srividya Kona
Markus Krtzsch
Sebastian Rudolph
Stefan Woltran

Table of Contents

LP and the Semantic Web.

LP and Semantic Web Services.

Extended Poster Abstracts.

Forgetting in Managing Rules and Ontologies*

Thomas Eiter[1], Giovambattista Ianni[1], Roman Schindlauer[1], Hans Tompits[1], and Kewen Wang[1,2]

[1] Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstrasse 9-11, A-1040 Vienna, Austria
{eiter, ianni, roman, tompits, kewen}@kr.tuwien.ac.at
[2] School of Information and Communication Technology, Griffith University,
Brisbane, QLD 4111, Australia

Abstract. The language of HEX-programs under the answer-set semantics is designed for interoperating with heterogeneous sources via external atoms and for meta-reasoning via higher-order literals in the context of the Semantic Web. As an important technique in managing knowledge bases, the notion of forgetting has received increasing interest in the knowledge-representation area. In this paper, we introduce a semantics-based theory of forgetting for HEX-programs and, in turn, for a class of OWL/RDF ontologies which allows to fully employ semantic information in managing ontologies like editing, merging, aligning, and redundancy removal.

1 Introduction

An ontology is a formal representation of concepts and relationships between them, making global interoperability possible. Managing ontologies is a central task for many Semantic-Web applications. However, it is often acknowledged that the Ontology Layer of the Semantic Web [1] is insufficient in its reasoning abilities. In particular, more and more ontologies are available on the Web and they are often very large in size and heterogeneous in location.

This phenomenon brings up a good deal of challenges to researchers in the Semantic Web. For example, when an ontology design is involved, we have to consider some issues like how to tailor an ontology or how to merge ontologies. Recently, these and related issues of managing ontologies have received considerable interests [12, 17, 18, 9, 10, 7]. Related issues include ontology editing, ontology segmentation, ontology merging, ontology aligning, ontology reusing, ontology update, and ontology redundancy removal. To some extent, all of these issues can be reduced to the problem of *extracting relevant segments out of large ontologies* for the purpose of effective management of ontologies so that the tractability for both humans and computers is enhanced. Such segments are not mere fragments of ontologies, but stand alone as ontologies in their own right. The intuition here is similar to views in databases: an existing ontology is

* This work was partially supported by the Austrian Science Fund (FWF) under grant P17212-N04, and by the European Commission through the IST Networks of Excellence REWERSE (IST-2003-506779).

tailored to a smaller ontology so that an optimal ontology is produced for specific applications. Although this problem has been identified and a number of approaches are proposed, like, e.g., [8, 20], a general framework for tailoring ontologies in a purely semantic way is still missing.

On the other hand, the notion of forgetting [4, 15, 14] is a promising technique for adequately handling a range of classical tasks such as query answering, planning, decision-making, reasoning about actions, or knowledge update and revision. The idea of forgetting consists, informally, in the intelligent and "painless" removal of information from a given knowledge base. In other words, one may select some literals, predicates, or concepts, for being discarded (or *forgotten*) in a given knowledge base. However, the information selected for elimination is usually logically connected with other portions of the same knowledge base. It is thus important to preserve, to the best extent, soundness and completeness of the information entailed after removal.

This is similar in nature to the aforementioned problem in the design and engineering of Web-based ontology languages. Consider a scenario from [8]: Suppose we start to design an ontology about various pets (like cats or dogs, but not lions or tigers). As currently there are numerous ontologies on the Web, suppose we searched the Web and found a large ontology on various animals including cats, dogs, tigers and lions. It may not be appropriate to adopt and use the whole ontology. For example, we may wish to discard (or "forget") tigers and lions from it.

While a literature on forgetting in logic programming exists (see, e.g., [22, 4]), and although forgetting takes relevance also in ontology-description formalism such as OWL, an explicit notion of forgetting has not been given yet for this class of languages. In this respect, the relationship between a notion of forgetting in ontologies and of forgetting in rule-based formalisms has not satisfactorily been investigated yet, and is thus matter of new research.

The problem of forgetting in ontologies can indeed be solved by exploiting the connection between ontology-description formalisms and logic programming. That is, given a sound notion of forgetting for logic programming, a knowledge base L, formulated under a generic semantics (e.g. RDFS, OWL, etc.) can be transposed to an equivalent logic program P_L, formulated under a different (and usually, nonmonotonic) semantics. Then, logic programming forgetting techniques are applied to P_L and a modified program, $\mathsf{forget}(P_L, l)$, is obtained and translated back to a knowledge base L', where l is the information to be discarded, which can be either a propositional atom, a concept, or a predicate.

Nonetheless, in order to fulfill the above approach, several issues, some of which already tackled in the literature, have to be solved and accommodated:

- A systematic way for translating L to P_L must be given. Attempts in this direction are several: for instance, Grosof *et al.* [11] translate a fragment of OWL-DL to Horn logic, whereas Swift [21] and Motik, Volz, and Maedche [16] port significant fragments of description logics to positive disjunctive logic programs.
- The pre-existing forgetting semantics [22, 4] mainly concentrates on discarding propositional information from ground programs. However, often P_L might be a non-ground program and l a non-propositional value (such as a predicate whose entire extension must be discarded). Also, many ontology description languages

(such as RDF and RDFS) include the possibility of exchanging the notion of class with the notion of individual, in order to enable meta-reasoning. In such a setting, P_L is better mapped to a higher-order logic program.
- Also it is unclear in which cases forget(P_L, l) can be mapped back to a valid knowledge base L'.

In the present paper, we aim at answering some of the questions above.

The logic programming language of choice is HEX, as defined in previous work [3]. This is a rule-based, fully declarative formalism which allows both for higher-order atoms and external atoms, under a well-defined generalization of the answer-set semantics [6].

Intuitively, a higher-order atom allows to quantify values over predicate names and to freely exchange predicate symbols with constant symbols, like in the rule

$$C(X) \leftarrow subClassOf(D, C), D(X).$$

An external atom facilitates the assignment of a truth value of an atom through an external source of computation. For instance, the rule

$$t(Sub, Pred, Obj) \leftarrow \&rdf[uri](Sub, Pred, Obj)$$

computes the predicate t taking values from the predicate $\&rdf$. The latter extracts RDF statements from the set of URIs specified by the extension of the predicate uri; this task is delegated to an external computational source (e.g., an external deduction system, an execution library, etc.). External atoms allow for a bidirectional flow of information to and from external sources of computation such as description-logic reasoners. By means of HEX-programs, powerful meta-reasoning becomes available in a decidable setting, e.g., not only for Semantic-Web applications, but also for meta-interpretation techniques in answer-set programming (ASP) itself, or for defining policy languages.

The contributions in this paper can be summarized as follows:

1. We introduce the notion of semantic forgetting for HEX-programs. Forgetting in logic programs has been previously considered by Eiter and Wang [4], who defined forgetting of a given literal l in the context of propositional disjunctive logic programs. This notion is extended in order to deal with external and higher-order atoms, as well as with positive non-ground programs.
2. We develop an algorithm for forgetting which is useful in the setting of ontology management. The basic idea of this algorithm is that certain rules that are locally redundant may become relevant afterwards and thus they are kept in the program.
3. We show how semantic forgetting of ontologies can be performed using an equivalent logic program, whose modified versions (after forgetting) are translated back to ontologies. In particular, that fragment of OWL-DL is taken into account which can be translated to description-logic programs [11]. The approach can be currently generalized to all those ontology languages for which a sound and complete mapping to positive logic programs is known.

Our approach is illustrated on some example application. For this, we use an ontology "Person-Relationship" in the paper, which can be scaled as large as one wishes.

The rest of the paper is organized as follows. Section 2 briefly recalls syntax and semantics of HEX-programs. Section 3 introduces the notion of semantic forgetting for HEX-programs and a novel algorithm for computing forgetting. As well, forgetting for non-ground positive programs is defined. Section 4, then, discusses a method for forgetting OWL/RDF-ontologies in terms of a transformation technique. Finally, Section 5 wraps up the paper with some concluding remarks.

2 HEX-Programs

2.1 Syntax

HEX programs are built on mutually disjoint sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ of *constant names*, *variable names*, and *external predicate names*, respectively. Unless stated otherwise, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are written with first letter in upper case (resp., lower case), and elements from $\mathcal{G}$ are prefixed with "&." Constant names serve both as individual and predicate names. Importantly, $\mathcal{C}$ may be infinite.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms and $n \geq 0$ is its *arity*. Intuitively, Y_0 is the predicate name; we thus also use the familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant. For example, $(x, rdf{:}type, c)$ and $node(X)$ are ordinary atoms, while $D(a, b)$ is a higher-order atom. An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m), \tag{1}$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input list* and *output list*, respectively), and $\&g$ is an *external predicate name*.

It is possible to specify *molecules* of atoms similar as in F-Logic [13]. For instance, $gi[father \rightarrow X, Z \rightarrow iu]$ is a shortcut for the conjunction $father(gi, X)$, $Z(gi, iu)$.

A HEX-*program*[1] is a set of rules of the form

$$\alpha \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m, \tag{2}$$

where $m \geq 0$, α is a higher-order atom, and $\beta_1, \ldots, \beta_m$ are either higher-order atoms or external atoms. The operator "not" is *negation as failure* (or *default negation*). For a rule r as in (2), we define $head(r) = \alpha$ and $body(r) = body^+(r) \cup body^-(r)$, where $body^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $body^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. If r contains only ordinary atoms, then r is *ordinary*. Furthermore, r is *quasi-negative* if $n = 0$. A HEX-program is *quasi-negative* if it contains only quasi-negative rules. An ordinary rule is *positive* iff $m = n$, i.e., if it contains no negation as failure. A program is positive iff all rules in it are positive.

We mention that higher-order features in logic programs have also been considered, e.g., by Chen, Kifer, and Warren [2] and Ross [19].

[1] In contrast to the original definition in [3], here we consider only HEX-programs without disjunctions in rule heads.

2.2 Semantics

The semantics of HEX-programs [3] is defined by generalizing the answer-set semantics [6]. The *Herbrand base* of a program P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, is defined accordingly, and the grounding of program P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ are implicitly given by P.

An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing only atoms. We say that an interpretation I is a *model* of an atom $a \in HB_P$ iff $a \in I$. Furthermore, I is a model of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$ iff $f_{\&g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1$, where $f_{\&g}$ is an $(n+m+1)$-ary Boolean function associated with $\&g$, called *oracle function*, assigning each element of $2^{HB_P} \times \mathcal{C}^{n+m}$ either 0 or 1. We write $I \models a$ to express that I is a model of a.

Let r be a ground rule. We define (i) $I \models body(r)$ iff $I \models a$ for all $a \in body^+(r)$ and $I \not\models a$ for all $a \in body^-(r)$, and (ii) $I \models r$ iff $I \models head(r)$ whenever $I \models body(r)$. We say that I is a *model* of a HEX-program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$.

The *Faber-Leone-Pfeifer reduct* [5] (or short *FLP-reduct*) of P with respect to $I \subseteq HB_P$, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models body(r)$. $I \subseteq HB_P$ is an *answer set of* P iff I is a minimal model of fP^I. By $\mathsf{AS}(P)$ we denote the set of all answer sets of P.

A HEX-program is *consistent* if it has at least one answer set. We call two HEX-programs, P and Q, *equivalent*, symbolically $P \equiv Q$, iff $\mathsf{AS}(P) = \mathsf{AS}(Q)$.

In practice, it is useful to differentiate between two kinds of input attributes for external atoms. For an external predicate $\&g$ (exploited, say, in an atom $\&g[p](X)$), a term appearing in an attribute position of type *predicate* (in this case, p) means that the outcomes of $f_{\&g}$ are dependent from the current interpretation I, for what the extension of the predicate named p in I is concerned. An input attribute of type *constant* does not imply a dependency of $f_{\&g}$ from some portion of I. An external predicate whose input attributes are all of type constant does not depend from the current interpretation.

Example 2.1. The external predicate $\&rdf$ introduced before is implemented with a single input argument of type predicate, because its associated function finds the RDF-URIs in the extension of the predicate uri:

$$tr(S, P, O) \leftarrow \&rdf[uri](S, P, O),$$
$$uri(\text{“}file\text{://}foaf.rdf\text{”}) \leftarrow .$$

Should the input argument be of type constant, an equivalent program would be:

$$tr(S, P, O) \leftarrow \&rdf[\text{“}file\text{://}foaf.rdf\text{”}](S, P, O)$$

or

$$tr(S, P, O) \leftarrow \&rdf[X](S, P, O), uri(X),$$
$$uri(\text{“}file\text{://}foaf.rdf\text{”}) \leftarrow .$$

□

3 Forgetting in HEX-Programs

As we have explained in Section 1, the technique of forgetting is useful in managing ontologies. So it is natural and interesting to generalize forgetting to HEX-programs. In fact, since HEX-programs have higher-order syntax but first-order semantics, it allows us to adapt the notion of forgetting to HEX-programs. In this section, we introduce the notion of forgetting for HEX-programs. The intuition behind the forgetting of an atom l in a HEX-program is to obtain a HEX-program which is equivalent to the original HEX-program if we ignore the existence of l.

In the next subsection, we assume that HEX-programs are ground and consistent. When a HEX-program with variables is given, it is a shorthand for its ground version. As we will see in Section 4, forgetting in an RDF ontology is defined in terms of forgetting in the corresponding logic program, which is a non-ground positive program. So, in Subsection 3.2, forgetting in non-grounded positive programs is considered.

3.1 Forgetting in Ground HEX-Programs

We call a set X' an *l-subset* of a set X, denoted $X' \subseteq_l X$, if $X' \setminus \{l\} \subseteq X \setminus \{l\}$. Similarly, a set X' is a *strict l-subset* of X, denoted $X' \subset_l X$, if $X' \setminus \{l\} \subset X \setminus \{l\}$. Two sets X and X' of literals are *l-equivalent*, denoted $X \sim_l X'$, if $(X \setminus X') \cup (X' \setminus X) \subseteq \{l\}$.

Definition 3.1. *Let P be a consistent* HEX*-program, let l be a (ground) atom in P, and let X be a set of atoms.*

1. *For a collection $\mathcal{S}$ of sets of atoms, $X \in \mathcal{S}$ is* l-minimal *in $\mathcal{S}$ if there is no $X' \in \mathcal{S}$ such that $X' \subset_l X$.*
2. *An answer set X of a* HEX*-program P is an* l-answer set *if X is l-minimal in $\mathsf{AS}(P)$.*

Example 3.1. Let $P = \{p \leftarrow not\ q;\ q \leftarrow not\ p;\ s \leftarrow p;\ s \leftarrow q\}$. It is easy to see that P has two answer sets, viz. $X = \{p, s\}$ and $X' = \{q, s\}$. Then, X is a p-answer set but X' is not. □

Having defined the notion of minimality about forgetting an atom, we are now in a position to define the result of forgetting about an atom in a HEX-program.

Definition 3.2. *Let P be a consistent* HEX*-program and let l be a (ground) atom. A* HEX*-program P' is a* result of forgetting about l in P*, if P' represents l-answer sets of P, i.e., such that the following conditions are satisfied:*

1. *$At(P') \subseteq At(P) - \{l\}$, where, for any program Q, $At(Q)$ denotes the set of atoms occurring in Q.*
2. *For any set X' of atoms with $l \notin X'$, X' is an answer set of P' iff there is an l-answer set X of P such that $X' \sim_l X$.*

Note that the first condition implies that l does not appear in P'. We use $\mathsf{forget}(P, l)$ to denote a possible result of forgetting about l in P.

Since an atom that does not appear in the head of a rule in a HEX-program is automatically assumed to be false in the process of forgetting for ordinary programs, all external atoms would be removed from the program. For this reason, the native algorithm for forgetting [4] is not helpful for HEX-programs. Thus, we introduce a new algorithm, which is inspired by Algorithm 4 in the system LPForget.[2]

Preparatory for describing the algorithm, below we introduce some program transformations for HEX-programs, which are generalizations of respective ones for ordinary programs [4].

In the following, let P and P' be HEX-programs.

Elimination of Tautologies: P' is obtained from P by *elimination of tautologies* iff there is a rule r in P such that $head(r) \in body^+(r)$ and $P' = P - \{r\}$.

Elimination of Head Redundancy: P' is obtained from P by *elimination of head redundancy* iff there is a rule r in P such that $head(r) \in body^-(r)$ and $P' = (P - \{r\}) \cup \{\leftarrow body(r)\}$.

Positive Reduction: P' is obtained from P by *positive reduction* iff there is a rule r in P such that $body^-(r)$ contains some c which does not occur in the head of any rule in P and P' is obtained from P by removing $not\, c$ from r.

Negative Reduction: P' is obtained from P by *negative reduction* iff there are two rules r and $r' : b' \leftarrow$ in P such that $b' \in body^-(r)$ and $P' = P - \{r\}$.

Elimination of Implications: Let r and r' be two distinct rules in a logic program. We say that r' is an *implication* of r if $head(r) = head(r')$ and $body(r) \subset body(r')$. Then, P' is obtained from P by *elimination of implications* iff there are two distinct rules r and r' of P such that r' is an implication of r and $P' = P - \{r'\}$.

Unfolding: For two rules r and r' with $head(r') \in body^+(r)$, the *unfolding* of r with r', denoted $unfold(r, r')$, is the rule $head(r) \leftarrow (body(r) - \{head(r')\}), body(r')$. Then, P' is obtained from P by *unfolding* if there is a rule r such that

$$P' = (P - \{r\}) \cup \{unfold(r, r') \mid r' \in P, head(r') \in body^+(r)\}.$$

A special case of unfolding is when there is no rule r' such that r' is resolved with r. In this case, $P' = P - \{r\}$.

We use $\mathcal{T}$ to denote the set of program transformations introduced above.

Lemma 3.1. *Using program transformations in $\mathcal{T}$, every* HEX*-program can be transformed into a quasi-negative program such that no atom appears in both head and body of a rule.*

The algorithm for computing the result of forgetting, referred to as *Algorithm 1*, is depicted in Figure 1. This algorithm can be easily implemented using the system LPForget. Note that the current form of Algorithm 1 is incomplete with respect to the semantic forgetting for some special cases while it is intuitive and can be seen an

[2] See `http://www.cit.gu.edu.au/~kewen/LPForget/`.

Algorithm 1 (Computing a result of forgetting)
Input: HEX-program P and an atom l in P.
Output: Program $\mathsf{forget}(P, l)$ as a result of forgetting l from P.
Method:
Step 1. Positive Splitting: Initially take Q as the set of all rules in which l appears. For every rule r in P such that either $head(r)$ or some literal of $body^-(r)$ appears in Q, add r to Q. Repeat this process until no new rule can be added. The resulting program is still denoted Q.
Step 2. Fully apply on Q the program transformations $\mathcal{T}$ and then obtain a quasi-negative program Q'. During this process, we keep record of the set $RU(Q, l)$ of all rules removed by unfolding but containing no appearance of l.
Step 3. Suppose that Q' has n rules with head l:

$$r_j : l \leftarrow not\, l_{j1}, ..., not\, l_{jm_j},$$

where $n \geq 0$, $j = 1, \ldots, n$ and $m_j \geq 0$ for all j.
If $n = 0$, then let Q'' denote the program obtained from Q' by removing all appearances of $not\, l$.
If $n = 1$ and $m_1 = 0$, then $l \leftarrow$ is the only rule in Q' having head l. In this case, remove every rule in Q' whose body contains $not\, l$. Let Q'' be the resulting program.
For $n \geq 1$ and $m_1 > 0$, let $D_1, \ldots, D_s$ be all possible conjunctions $(l_{1k_1}, \cdots, l_{nk_n})$, where $0 \leq k_1 \leq m_1$, ..., $0 \leq k_n \leq m_n$. Replace each occurrence of $not\, l$ in Q' by all possible D_i. Let Q'' be the result.
Step 4. Output $Q'' \cup RU(Q, l) \cup \bar{Q}$ as $\mathsf{forget}(P, l)$, where $\bar{Q} = P \setminus Q$.

Fig. 1. Algorithm 1 for computing a result of forgetting.

ideal approximation to the semantic forgetting. A complete algorithm is obtained by replacing Step 3 with Step 3 of Algorithm 2 given by Eiter and Wang [4].

For a consistent HEX-program P and an atom l, some program P' as in Definition 3.2 always exists. However, different such programs P' might exist. It follows from the above definition that they are all equivalent under the answer-set semantics.

Proposition 3.1. *Let P be a* HEX*-program and l an atom in P. If P' and P'' are two results of forgetting about l in P, then $P' \equiv P''$.*

Example 3.2. Suppose that L is a knowledge base on the Web consisting of various axioms about persons and their relationships. In particular, L contains assertions depicted in Figure 2.

Let P now be the following HEX-program, where $\&dlC$ and $\&dlR$ are external atoms that query the extensions of a specified concept resp. role from a single description logic ontology:[3]

$$\begin{aligned} sibling(X, Y) &\leftarrow \&dlR[siblingOf](X, Y); \\ sibling(X, Y) &\leftarrow \&dlR[childOf](X, Z), \&dlR[childOf](Y, Z); \\ inEurope(Y) &\leftarrow sibling(\text{“}Bob\text{”}, Y), not\ inAmerica(Y); \\ inAmerica(Y) &\leftarrow sibling(\text{“}Bob\text{”}, Y), not\ inEurope(Y). \end{aligned}$$

[3] For the sake of readability, we use a simplified version of the actual and implemented dl-atoms for HEX-programs here.

$Male \sqsubseteq Person$
$Female \sqsubseteq Person$
$\top \sqsubseteq \forall knows^{-}.Person$
$\top \sqsubseteq \forall knows.Person$
$friendOf \sqsubseteq knows$
$childOf \sqsubseteq knows$
$siblingOf \equiv siblingOf^{-}$
$siblingOf^{+} \sqsubseteq siblingOf$
$siblingOf \sqsubseteq knows$
$parentOf(Alice, Bob)$
$parentOf(Alice, Carl)$
$parentOf(Bob, Emma)$
$sameProject(Bob, Dennis)$
$spouseOf \sqsubseteq knows$
$spouseOf \equiv spouseOf^{-}$
$worksWith \equiv worksWith^{-}$
$worksWith^{+} \sqsubseteq worksWith$
$worksWith \sqsubseteq knows$
$parentOf \equiv childOf^{-}$
$parentOf \sqsubseteq ancestorOf$
$ancestorOf \sqsubseteq knows$
$ancestorOf^{+} \sqsubseteq ancestorOf$
$sameProject \sqsubseteq worksWith$

Fig. 2. Example ontology L.

To apply forgetting, we first have to obtain the ground program $grnd(P)$. In order to keep the example readable, we omit those ground rules whose bodies are not satisfied by L:

$$\begin{aligned}
sibling(\text{“}Bob\text{”}, \text{“}Carl\text{”}) \leftarrow\ & \&dlR[childOf](\text{“}Bob\text{”}, \text{“}Alice\text{”}),\\
& \&dlR[childOf](\text{“}Carl\text{”}, \text{“}Alice\text{”});\\
sibling(\text{“}Carl\text{”}, \text{“}Bob\text{”}) \leftarrow\ & \&dlR[childOf](\text{“}Carl\text{”}, \text{“}Alice\text{”}),\\
& \&dlR[childOf](\text{“}Bob\text{”}, \text{“}Alice\text{”});\\
inEurope(\text{“}Carl\text{”}) \leftarrow\ & sibling(\text{“}Bob\text{”}, \text{“}Carl\text{”}), not\ inAmerica(\text{“}Carl\text{”});\\
inAmerica(\text{“}Carl\text{”}) \leftarrow\ & sibling(\text{“}Bob\text{”}, \text{“}Carl\text{”}), not\ inEurope(\text{“}Carl\text{”}).
\end{aligned}$$

Thus, $grnd(P)$ has two answer sets, viz.

$X_1 = \{sibling(\text{“}Bob\text{”}, \text{“}Carl\text{”}), sibling(\text{“}Carl\text{”}, \text{“}Bob\text{”}), inEurope(\text{“}Carl\text{”})\}$ and
$X_2 = \{sibling(\text{“}Bob\text{”}, \text{“}Carl\text{”}), sibling(\text{“}Carl\text{”}, \text{“}Bob\text{”}), inAmerica(\text{“}Carl\text{”})\}$.

If we allow to forget about $sibling(\text{“}Carl\text{”}, \text{“}Bob\text{”})$ in $grnd(P)$, then the result of forgetting is obtained from $grnd(P)$ by removing the first rule. □

The above definitions of forgetting about an atom l can be extended to forgetting about a set F of atoms. Specifically, we can similarly define $X_1 \subseteq_F X_2$, $X_1 \sim_F X_2$, and F-answer sets of a HEX-program. In fact, the properties of forgetting about a single atom can be generalized to the case of forgetting about a set. Moreover, the result of forgetting about a set F can be obtained by forgetting each atom one by one in F.

Proposition 3.2. *Let P be a consistent* HEX*-program and $F = \{l_1, \ldots, l_m\}$ a set of atoms. Then,* $\mathsf{forget}(P, F) \equiv \mathsf{forget}(\ldots(\mathsf{forget}(\mathsf{forget}(P, l_1), l_2), \ldots), l_m)$.

Since higher-order atoms and external atoms can be treated as ordinary atoms in the process of forgetting, we can prove the above result similarly to the proof of Proposition 6 given by Eiter and Wang [4].

For HEX-programs, the notion of ordinary forgetting may not be sufficient for some applications in managing ontologies. In some cases, we need to forget a *predicate*. This can be easily accomplished by forgetting the set of all atoms with the same predicate.

Due to the presence of higher-order terms, we may need also to forget some other atoms when we want to forget a specific atom. This is illustrated in the following example.

Example 3.3. Suppose we want to forget the predicate "$brotherOf$" in the following program:

$$\begin{aligned} subRelation(brotherOf, siblingOf) &\leftarrow \\ brotherOf(john, al) &\leftarrow \\ siblingOf(john, joe) &\leftarrow \\ siblingOf(al, mick) &\leftarrow \\ R(X, Y) &\leftarrow subRelation(P, R), P(X, Y) \end{aligned}$$

Here, we should also forget $subRelation(brotherOf, siblingOf)$. □

For the above discussion, it is natural to define the following variant of forgetting, which is more intuitive for most applications.

Definition 3.3. *Let P be a* HEX*-program and l an atom in P. Denote by $sup(l)$ the set of all atoms in P that contain the predicate name of l. Then the result of* enforced forgetting *about l in P, written* $\mathsf{Forget}(P, l)$*, is defined as* $\mathsf{forget}(P, sup(l))$.

In Example 3.3, $\mathsf{Forget}(P, brotherOf)$, given by

$$\mathsf{forget}(P, \{brotherOf(john, al), subRelation(brotherOf, siblingOf)\}),$$

is the following program:

$$\begin{aligned} siblingOf(john, al) &\leftarrow ; \\ siblingOf(john, joe) &\leftarrow ; \\ siblingOf(al, mick) &\leftarrow . \end{aligned}$$

3.2 Forgetting in Non-Ground Positive Programs

As we will see in Section 4, the logic program P_L translated from an OWL/RDF ontology is non-ground in general and thus forgetting as defined by Eiter and Wang [4] cannot be directly applied here. However, since P_L has a special form and, in particular, has no negation as failure, we are able to lift the notion of forgetting for ground programs to this kind of non-ground programs.

To this end, we first need to define *weak unfolding* for logic programs.

Let $r : a \leftarrow b, B$ and $r' : b' \leftarrow B'$ be normal rules, where a, b, b' are atoms, and B, B' are conjunctions of literals. Note that no higher-order atoms occur here. When necessary, we can rename the variables of r' such that r and r' have no common variables. If the head b' of r' and b have a most general unifier (mgu) θ, then the rule $(a \leftarrow B, B')\theta$ is called a *resolvent* of r with r'.

Algorithm 2 (Computing forgetting for non-ground positive logic programs)
Input: Positive logic program P and a predicate R.
Output: Program $\mathsf{forget}(P, R)$ as the result of forgetting R from P.
Method:

1. Fully apply weak unfolding on P.
2. Remove all rules containing R.
3. Output the resulting program as $\mathsf{forget}(P, R)$.

Fig. 3. Computing forgetting for non-ground programs without negation as failure.

Weak Unfolding. A logic program P' is obtained from P by *weak unfolding* iff there are two rules r and r' in P such that r'' is a resolvent of r with r' and $P' = P \cup \{r''\}$.

For a positive logic program the result of forgetting can be easily obtained by Algorithm 2 depicted in Figure 3.

The following result shows that this lifting algorithm for forgetting is sound with respect to semantic forgetting for ground programs.

Theorem 3.1. *Let P be a non-ground positive program and R a predicate in P. For any extensional database E (i.e., a set of facts), we have*

$$\mathsf{forget}(P, R) \cup E \equiv \mathsf{forget}(grnd(P \cup E), const(R)),$$

where $const(R) = \{R(a) \mid a$ is a constant in $P \cup E\}$.

Proof. (Sketch) First, we observe the following two properties:

(α) If r'' is an instance of $unfold(r, r')$ in $P \cup E$, then $r'' = unfold(\bar{r}, \bar{r}')$, where $\bar{r}$ and $\bar{r}'$ are instances of r and r', respectively.
(β) If $r'' = unfold(\bar{r}, \bar{r}')$, for $\bar{r}$ and $\bar{r}'$ in $grnd(P \cup E)$, then r'' is an instance of $unfold(r, r')$ in $P \cup E$, where $\bar{r}$ and $\bar{r}'$ are instances of r and r', respectively.

Let $Q_1 = \mathsf{forget}(P, R) \cup E$ and $Q_2 = \mathsf{forget}(grnd(P \cup E), const(R))$. Since P is positive, $\mathsf{AS}(grnd(Q_1))$ and $\mathsf{AS}(Q_2)$ are singletons.

Let T_Q be the consequence operator of a positive program Q. Then, the unique answer set of Q is its least Herbrand model $\bigcup_{k \geq 0} T_Q \uparrow k$.

The unique element of $\mathsf{AS}(grnd(Q_1))$ is $\cup_{k \geq 0} T_{grnd(Q_1)} \uparrow k$, and the unique element of $\mathsf{AS}(Q_2)$ is $\bigcup_{k \geq 0} T_{Q_2} \uparrow k$. Using Properties ($\alpha$) and ($\beta$), we can easily show that $T_{grnd(Q_1)} \uparrow k = T_{Q_2} \uparrow k$, by induction on $k \geq 0$. So, $\mathsf{AS}(grnd(Q_1)) = \mathsf{AS}(Q_2)$. □

Algorithm 2 may be refined by applying Step 1 only to a subset of the rules and facts P which is relevant to R, while the rest of the program remains untouched. In this way, the cost of computing forgetting can be reduced radically.

For P and R in Algorithm 2, let Q initially be the set of all rules in which R appears. Then, add each rule r from P to Q such that $head(r)$ appears in Q, and repeat

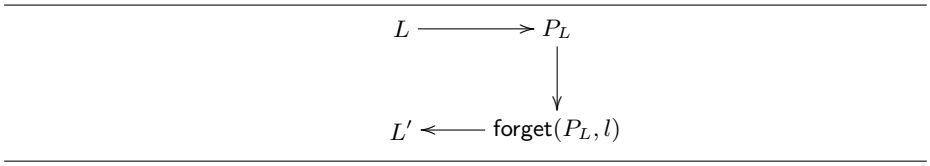


Fig. 4. Forgetting in ontologies via a logic program.

this process until no new rules can be added. Let the resulting program be denoted by $Q_{P,R}$. Intuitively, $\bar{Q}$ consists of rules that are irrelevant to R. For forgetting in a positive program, it is done by a series of unfolding and then removing some rules relevant to R. So, rules in $\bar{Q}$ are essentially unchanged during the process of forgetting.

Theorem 3.2. *Let P be a non-ground positive program P and let R be a predicate in P. For any extensional database E, it holds that*

$$\mathsf{forget}(P, R) \cup E \equiv \mathsf{forget}(Q_{P,R}, R) \cup \bar{Q} \cup E,$$

where $\bar{Q} = P \setminus Q_{P,R}$.

It should be noted that although the process of forgetting for non-ground programs is realized by the removal of certain rules, it has a semantic justification as Theorem 3.2 shows.

4 Forgetting in OWL/RDF-Ontologies

To apply forgetting purely to an ontology expressed in OWL or RDFS, we reuse the techniques defined for forgetting in logic programs. Figure 4 shows the general principle of this approach. First, an ontology L is translated into a rule representation P_L, taking the specific ontology semantics into account. Then, for any atom l in P_L, we can compute forget(P_L, l). Finally, we translate the result back into an ontology.

The translation of description-logic axioms into a logic program is shown in Tables 1 and 2. This translation covers most of the expressiveness of OWL Lite and corresponds to the translation given by Grosof *et al.* [11], mapping some subset of a description logic to positive equality-free datalog programs. Note that some description-logic constructs have no direct representation in logic-programming rules, such as cardinality constraints. Also, existential and universal quantification is restricted to the left-hand side resp. right-hand side of a subclass axiom. In general, a transformation from a set of rules back to ontology statements requires the rules in forget(P_L, l) to be in a form according to Tables 1 and 2.

Example 4.1. Consider again the ontology L in Figure 2. The translation of L into a logic program according to P_L is depicted in Figure 5. Suppose we do not want to keep the concepts $worksWith$, then we can use Theorem 3.1 to simplify the process of forgetting.

Take Q as a subprogram of P_L:

Table 1. Mapping of ontology statements to rules.

Statement	DL syntax	Rule representation
subClassOf	$D \sqsubseteq C$	$C(X) \leftarrow D(X).$
subPropertyOf	$P \sqsubseteq Q$	$Q(X,Y) \leftarrow P(X,Y).$
domain	$\top \sqsubseteq \forall P^{-}.C$	$C(X) \leftarrow P(X,Y).$
range	$\top \sqsubseteq \forall P.C$	$C(Y) \leftarrow P(X,Y).$
class-instance	$a : C$	$C(a) \leftarrow .$
property-instance	$\langle a,b\rangle : P$	$P(a,b) \leftarrow .$
class-equivalence	$D \equiv C$	$D(X) \leftarrow C(X);$
		$C(X) \leftarrow D(X).$
property-equivalence	$P \equiv Q$	$P(X,Y) \leftarrow Q(X,Y);$
		$Q(X,Y) \leftarrow P(X,Y).$
inverseOf	$P \equiv Q^{-}$	$P(X,Y) \leftarrow Q(Y,X);$
		$Q(X,Y) \leftarrow P(Y,X).$
transitiveProperty	$P^{+} \sqsubseteq P$	$P(X,Y) \leftarrow P(X,Z), P(Z,Y).$

Table 2. Mapping of ontology class constructors to rules.

Constructor	DL syntax	Rule representation
conjunction	$C_1 \sqcap C_2 \sqsubseteq D$	$D(X) \leftarrow C_1(X), C_2(X).$
	$C \sqsubseteq D_1 \sqcap D_2$	$D_1(X) \leftarrow C(X);$
		$D_2(X) \leftarrow C(X).$
disjunction	$C_1 \sqcup C_2 \sqsubseteq D$	$D(X) \leftarrow C_1(X);$
		$D(X) \leftarrow C_2(X).$
existential restriction	$\exists P.C \sqsubseteq D$	$D(X) \leftarrow P(X,Y), C(Y).$
universal restriction	$D \sqsubseteq \forall P.C$	$C(Y) \leftarrow P(X,Y), D(X).$

$$
\begin{aligned}
sameProject(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow ;\\
worksWith(X,Y) &\leftarrow worksWith(Y,X);\\
worksWith(X,Z) &\leftarrow worksWith(X,Y), worksWith(Y,Z);\\
knows(X,Y) &\leftarrow worksWith(X,Y);\\
worksWith(X,Y) &\leftarrow sameProject(X,Y).
\end{aligned}
$$

We can apply Algorithm 2 on the logic program Q by forgetting $worksWith$. First, fully apply weak unfolding on Q and obtain Q':

$$
\begin{aligned}
sameProject(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow ;\\
worksWith(X,Y) &\leftarrow worksWith(Y,X);\\
worksWith(X,Z) &\leftarrow worksWith(X,Y), worksWith(Y,Z);\\
knows(X,Y) &\leftarrow worksWith(X,Y);\\
worksWith(X,Y) &\leftarrow sameProject(X,Y);\\
knows(X,Y) &\leftarrow sameProject(X,Y);\\
worksWith(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow ;\\
knows(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow .
\end{aligned}
$$

$$
\begin{aligned}
parentOf(\text{“}Alice\text{”}, \text{“}Carl\text{”}) &\leftarrow .\\
female(\text{“}Alice\text{”}) &\leftarrow .\\
sameProject(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow .\\
male(\text{“}Bob\text{”}) &\leftarrow .\\
parentOf(\text{“}Carl\text{”}, \text{“}Emma\text{”}) &\leftarrow .\\
person(\text{“}Carl\text{”}) &\leftarrow .\\
person(\text{“}Dennis\text{”}) &\leftarrow .\\
knows(X,Y) &\leftarrow childOf(X,Y).\\
childOf(X,Y) &\leftarrow parentOf(Y,X).\\
male(X) &\leftarrow father(X).\\
person(X) &\leftarrow female(X).\\
friendOf(X,Y) &\leftarrow friendOf(Y,X).\\
knows(X,Y) &\leftarrow ancestorOf(X,Y).\\
ancestorOf(X,Z) &\leftarrow ancestorOf(X,Y),\\
&\qquad\quad ancestorOf(Y,Z).\\
knows(X,Y) &\leftarrow friendOf(X,Y).
\end{aligned}
\qquad
\begin{aligned}
person(X) &\leftarrow knows(X,Y).\\
person(Y) &\leftarrow knows(X,Y).\\
person(X) &\leftarrow male(X).\\
female(X) &\leftarrow mother(X).\\
ancestorOf(X,Y) &\leftarrow parentOf(X,Y).\\
parentOf(X,Y) &\leftarrow childOf(Y,X).\\
siblingOf(X,Y) &\leftarrow siblingOf(Y,X).\\
knows(X,Y) &\leftarrow siblingOf(X,Y).\\
spouseOf(X,Y) &\leftarrow spouseOf(Y,X).\\
knows(X,Y) &\leftarrow spouseOf(X,Y).\\
worksWith(X,Y) &\leftarrow worksWith(Y,X).\\
worksWith(X,Z) &\leftarrow worksWith(X,Y),\\
&\qquad\quad worksWith(Y,Z).\\
knows(X,Y) &\leftarrow worksWith(X,Y).\\
worksWith(X,Y) &\leftarrow sameProject(X,Y).
\end{aligned}
$$

Fig. 5. Translation of L into a logic program P_L.

Thus, the result of forgetting about $worksWith$ is the program

$$\mathsf{forget}(Q, worksWith) \cup \bar{Q},$$

where $\bar{Q} = P_L \setminus Q$ and $\mathsf{forget}(Q, worksWith)$ is as follows:

$$
\begin{aligned}
sameProject(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow ;\\
knows(X,Y) &\leftarrow sameProject(X,Y);\\
knows(\text{“}Bob\text{”}, \text{“}Dennis\text{”}) &\leftarrow .
\end{aligned}
$$

Translating this fragment back into the original description logic results in the following statements:

$sameProject(\text{“}Bob\text{”}, \text{“}Dennis\text{”});$
$sameProject \sqsubseteq knows;$
$knows(\text{“}Bob\text{”}, \text{“}Dennis\text{”}).$

The property $worksWith$ does not occur in the modified description-logic knowledge base any more, while the subproperty relation between $sameProject$ and $knows$ is preserved. □

Combining the approaches to forgetting of Sections 3 and 4, we are now able to forget any set of ordinary atoms, higher-order atoms, whole external atoms, and parts of external atoms in a HEX-program.

5 Related Work and Concluding Remarks

The notion of forgetting for HEX-programs introduced in this paper generalizes a respective notion for ordinary logic programs defined in previous work [4]. Forgetting for HEX-programs provides a means to handle forgetting at the user-view level, since HEX-programs are tailored to access sources like OWL/RDF ontologies at the extensional

level through external atoms, but does not go back to changes in these sources, as is done in the view-update problem of databases, for instance. However, such ontologies have been cast to a class of logic programs which constitute a small fragment of HEX-programs, and thus semantic forgetting for OWL/RDF may be facilitated through this mapping, as we have shown. Our work therefore provides a uniform basis for a framework for extracting ontology segments from a custom ontology, which is exploited at the user level. This approach is in an active area of *semantic integration* in ontologies (see [17] for a survey). However, the emphasis of our work is on conflict resolving in semantic integration of ontologies rather than on ontology mapping.

Forgetting for OWL/RDF ontologies can be used for various tasks in ontology management including the following:

- *Ontology segmentation:* This approach is to obtain segments from a custom ontology, thus having the same purpose as forgetting. Seidenberg and Rector [20] present a series of strategies for extracting ontology segments. However, it lacks a general semantic justification.
- *Ontology merging:* Given two ontologies O_1 and O_2, they could first be preprocessed by techniques in ontology mapping and then be merged into one ontology. In many cases, conflicts may be present in the process of merging. If the conflict is caused by some concept C, a natural approach is to forget C from one of these two ontologies or from both. Grau, Parsia, and Sirin [8] propose to use so-called "E-connections" for merging ontologies. In this approach, merging ontologies is defined in terms of link properties. However, it is difficult to find related link properties.

Similar to other approaches to semantic integration, it is a hard issue to determine the set of concepts which should be forgotten if they are not explicitly specified by the user. This issue could be solved by employing heuristics and techniques from machine learning. Exploring this is left for future work.

References

1. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
2. W. Chen, M. Kifer, and D. Warren. HILOG: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, 15(3):187–230, 1993.
3. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence* (*IJCAI 2005*), pages 90–96. Morgan Kaufmann, 2005.
4. T. Eiter and K. Wang. Forgetting and Conflict Resolving in Disjunctive Logic Programming. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence* (*AAAI 2006*). AAAI Press, 2006.
5. W. Faber, N. Leone, and G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence* (*JELIA 2004*), pages 200–212, 2004.
6. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

7. S. Ghilardi, C. Lutz, and F. Wolter. Did I Damage My Ontology? A Case for Conservative Extensions in Description Logics. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 187–197. AAAI Press, 2006.
8. B. Grau, B. Parsia, and E. Sirin. Combining OWL Ontologies using E-Connections. *Journal of Web Semantics*, 4(1), 2005.
9. B. C. Grau, I. Horrocks, O. Kutz, and U. Sattler. Will my Ontologies Fit Together? In *Proceedings of the 2006 International Workshop on Description Logics* (*DL 2006*), 2006.
10. B. C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Modularity and Web Ontologies. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 198–208. AAAI Press, 2006.
11. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description Logic Programs: Combining Logic Programs with Description Logics. In *Proceedings of the Twelfth International World Wide Web Conference* (*WWW 2003*), pages 48–57, 2003.
12. Y. Kalfoglou and M. Schorlemmer. Ontology Mapping: the State of the Art. *The Knowledge Engineering Review*, 18:1–31, 2003.
13. M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.
14. J. Lang, P. Liberatore, and P. Marquis. Propositional Independence: Formula-Variable Independence and Forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.
15. F. Lin and R. Reiter. Forget it. In *Proceedings of the AAAI Fall Symposium on Relevance*, pages 154–159. New Orleans, 1994.
16. B. Motik, R. Volz, and A. Maedche. Optimizing Query Answering in Description Logics using Disjunctive Deductive Databases. In *Proceedings of the Tenth International Workshop on Knowledge Representation meets Databases* (*KRDB 2003*), 2003. `http://CEUR-WS.org/Vol79/`.
17. N. Noy. Semantic Integration: A Survey of Ontology-Based Approaches. *SIGMOD Record*, 33(4):65–70, 2004.
18. N. Noy and H. Stuckenschmidt. Ontology Alignment: An Annotated Bibliography. In *Semantic Interoperability and Integration*, 2005.
19. K. A. Ross. On Negation in HiLog. *Journal of Logic Programming*, 18(1):27–53, 1994.
20. J. Seidenberg and A. Rector. Web Ontology Segmentation: Analysis, Classification and Use. In *Proceedings of the Fifteenth International World Wide Web Conference* (*WWW 2006*), 2006.
21. T. Swift. Deduction in Ontologies via ASP. In *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR-7*), volume 2923 of *LNCS*, pages 275–288, 2004.
22. K. Wang, A. Sattar, and K. Su. A Theory of Forgetting in Logic Programming. In *Proceedings of the Twentieth National Conference on Artificial Intelligence* (*AAAI 2005*), pages 682–687. AAAI Press, 2005.

Query Evaluation and Optimization in the Semantic Web

Edna Ruckhaus, Eduardo Ruiz, and María-Esther Vidal

Dept. of Computer Science and Information Technology
Universidad Simón Bolívar, Venezuela
{ruckhaus,eruiz,mvidal}@ldc.usb.ve

Abstract. We address the problem of answering Web ontology queries efficiently. An ontology is formalized as a *Deductive Ontology Base* (DOB), a deductive database that comprises the ontology's inference axioms and facts, and we present a cost-based query optimization technique for DOB. A hybrid cost model is proposed to estimate the cost and cardinality of basic and inferred facts. Cardinality and cost of inferred facts are estimated using an adaptive sampling technique, while techniques of traditional relational cost models are used for estimating the cost of basic facts and conjunctive ontology queries. Finally, we implement a dynamic-programming optimization algorithm to identify query evaluation plans that minimize the number of intermediate inferred facts. We modeled a subset of the Web ontology language OWL Lite as a DOB, and performed an experimental study to analyze the predictive capacity of our cost model and the benefits of the query optimization technique. Our study has been conducted over synthetic and real-world OWL ontologies, and shows that the techniques are accurate and improve query performance.

1 Introduction

Ontology systems usually provide reasoning and retrieval services that identify the basic facts that satisfy a requirement, and derive implicit knowledge using the ontology's inference axioms. In the context of the Semantic Web, the number of inferred facts can be extremely large. On one hand, the amount of basic ontology facts (domain concepts and Web source annotations) can be considerable, and on the other hand, *Open World* reasoning in Web ontologies may yield a large space of choices. Therefore, efficient evaluation strategies are needed in Web ontology's inference engines.

In our approach, ontologies are formalized as a deductive database called a *Deductive Ontology Base* (DOB). The extensional database comprises all the ontology language's statements that represent the explicit ontology knowledge. The intensional database corresponds to the set of deductive rules which define the semantics of the ontology language. We provide a cost-based optimization technique for Web ontologies represented as a DOB.

Traditional query optimization techniques for deductive databases systems include join-ordering strategies, and techniques that combine a bottom-up evaluation with top-down propagation of query variable bindings in the spirit of the Magic-Sets algorithm [17]. Join-ordering strategies may be heuristic-based or cost-based; some cost-based approaches depend on the estimation of the *join selectivity*; others rely on the *fan-out* of a literal [22]. Cost-based query optimization has been successfully used by relational database management systems; however, these optimizers are not able to estimate the cost or cardinality of data that do not exist a priori, which is the case of intensional predicates in a DOB.

We propose a hybrid cost model that combines two techniques for cardinality and cost estimation: (1) the sampling technique proposed in [10, 11] is applied for the estimation of the evaluation cost and cardinality of intensional predicates, and (2) a cost model *à* la System R cost model is used for the estimation of the cost and cardinality of extensional predicates and the cost of conjunctive queries.

Three evaluation strategies are considered for "joining" predicates in conjunctive queries. They are based on the Nested-Loop, Block Nested-Loop, and Hash Join operators of relational databases [16]. To identify a good evaluation plan, we provide a dynamic-programming optimization algorithm that orders subgoals in a query, considering estimates of the subgoal's evaluation cost.

We modeled a subset of the Web ontology language OWL Lite [12] as a DOB, and performed experiments to study the predictive capacity of the cost model and the benefits of the ontology query optimization techniques. The study has been conducted over synthetic and real-world OWL ontologies. Preliminary results show that the cost-model estimates are pretty accurate and that optimized queries are significantly less expensive than non-optimized ones.

Our current formalism does not represent the OWL built-in constructor *ComplementOf*. We stress that in practice this is not a severe limitation. For example, this operator is not used in any of the three real-world ontologies that we have studied in our experiments; and in the survey reported in [23], only 21 ontologies out of 688 contain this constructor.

Our work differs from other systems in the Semantic Web that combine a Description Logics (DL) reasoner with a relational DBMS in order to solve the scalability problems for reasoning with individuals [3, 6, 7, 15]. Clearly, all of these systems use the query optimization component embedded in the relational DBMS; however, they do not develop cost-based optimization for the implicit knowledge, that is, there is no estimation of the cost of data not known a priori.

Other systems use Logic Programming (LP) to reason on large-scale ontologies. This is the case of the projects described in [5, 8, 13] . In Description Logic Programs (DLP) [5], the expressive intersection between DL and LP without function symbols is defined. DL queries are reduced to LP queries and efficient LP algorithms are explored. The project described in [8, 13] reduces a $\mathcal{SHIQ}$ knowledge base to a Disjunctive Datalog program. Both projects apply Magic-Sets rewriting techniques but to the best of our knowledge, no cost-based optimization techniques have been developed. The OWL Lite$^-$ species of the OWL language proposed in [2] is based in the DLP project; it corresponds to the por-

tion of the OWL Lite language that can be translated to Datalog. All of these systems develop LP reasoning with individuals, whereas in the DOB model we develop Datalog reasoning with both, domain concepts and individuals.

In [4], an efficient bottom-up evaluation strategy for HEX-programs based on the theory of *splitting sets* is described. In the context of the Semantic Web, these non-monotonic logic programs contain higher-order atoms and external atoms that may represent RDF and OWL knowledge. However, their approach does not include determining the best evaluation strategy according to a certain cost metric.

In the next section we describe our DOB formalism. Following this, we describe the DOB-S System architecture, Then, we model a subset of OWL Lite as a DOB and present a motivating example. Next, we develop our hybrid cost model and query optimization algorithm. We describe our experimental study and, finally, we point out our conclusions and future work.

2 The Deductive Ontology Base (DOB)

In general, an ontology knowledge base can be defined as:

Definition 1 (Ontology Knowledge Base) *An **ontology knowledge base** **O** is a pair $O = \langle \mathcal{F}, \mathcal{I} \rangle$, where $\mathcal{F}$ is the set of ontology facts that represent the explicit ontology structure (domain) and source annotations (individuals), and $\mathcal{I}$ is the set of axioms that allow the inference of new ontology facts regarding both domain and individuals.*

We will model O as a deductive database which we call a *Deductive Ontology Base* (DOB). A DOB is composed of an Extensional Ontology Base (EOB) and an Intensional Ontology Base (IOB). Formally, a DOB is defined as:

Definition 2 (DOB) *Given an ontology knowledge base $O = \langle \mathcal{F}, \mathcal{I} \rangle$, a **DOB** is a deductive database composed of a set of built-in EOB ground predicates representing $\mathcal{F}$ and a set of IOB built-in predicates representing $\mathcal{I}$, i.e. that define the semantics of the EOB built-in predicates.*

IOB predicates and DOB queries are defined as follows:

Definition 3 (Intensional Predicate) *Given a DOB composed of an EOB and an IOB, an **Intensional Predicate** is a rule $R{:}H(\overline{X}) \leftarrow \exists \overline{Y} B(\overline{X}, \overline{Y})$, where H is the **head**, B is the **body** that corresponds to a conjunction of predicates, and $\overline{X}$ and $\overline{Y}$ are called distinguished variables and non-distinguished variables, respectively. H belongs to the IOB. Predicates in B can belong to the EOB or to the IOB (no negations are allowed).*

Definition 4 (DOB Query) *A **DOB query** is defined as a rule $q : Q(\overline{X}) \leftarrow \exists \overline{Y} B(\overline{X}, \overline{Y})$, where B is the query's goal.*

Next, we provide the definitions related to query-answering for DOBs.

Definition 5 (Valuation) *Given a set of variables $\mathcal{V}$ and a set of constants $\mathcal{C}$, a mapping or valuation γ is a function $\gamma : \mathcal{V} \rightarrow \mathcal{C}$.*

Definition 6 (Valid Instantiation) *Given a Deductive Ontology Base $\mathcal{O}$, a set of constants $\mathcal{C}$ in $\mathcal{O}$, a set of variables $\mathcal{V}$, a rule R, and an interpretation $\mathbb{I}$ of $\mathcal{O}$ that corresponds to its Minimal Perfect Model [1], a valuation γ is a* ***valid instantiation*** *of R if and only if, $\gamma(R)$ evaluates to true in $\mathbb{I}$.*

Definition 7 (Intermediate Inferred Facts) *Given a Deductive Ontology Base $\mathcal{O}$, and a query $q : Q(\overline{X}) \leftarrow \exists \overline{Y} B(\overline{X}, \overline{Y}))$. A proof tree for q wrt $\mathcal{O}$ is defined as follows:*

- *Each node in the tree is labeled by a predicate in $\mathcal{O}$.*
- *Each leaf in the tree is labeled by a predicate in $\mathcal{O}$'s EOB.*
- *The root of the tree is labeled by Q*
- *For each internal node N including the root, if N is labeled by a predicate A defined by the rule R, $A(\overline{X}) \leftarrow \exists \overline{Y} C(\overline{X}, \overline{Y}))$, where $C(\overline{X}, \overline{Y}))$ is the conjunction of the predicates $C_1, ..., C_n$, then, for each valid instantiation of R, γ, the node N has a sub-tree whose root is $\gamma(A(\overline{X}))$ and its children are respectively labeled $\gamma(C1)$,..., $\gamma(C_n)$.*

The valuations needed to define all the valid instantiations in the proof tree correspond to the **Intermediate Inferred Facts** *of q.*

The number of intermediate inferred facts measures the evaluation *cost* of the query Q. Additionally, since the valid instantiations of Q in the proof tree correspond to the answers of the query, the *cardinality* of Q corresponds to the number of such instantiations.

Note that the sets of EOB and IOB built-in predicates of a DOB define an ontology framework, so our model is not tied to any particular ontology language. To illustrate the use of our approach we focus on OWL Lite ontologies.

3 The DOB-S System's Architecture

DOB-S is a system that allows an agent to pose efficient conjunctive queries to ontologies. The system's architecture can be seen in Figure 1.

A subset of a given OWL ontology is translated into a DOB using an OWL Lite to DOB **translator**. EOB and IOB predicates are stored as a deductive database. Next, an **analyzer** generates the ontology's statistics: for each EOB predicate, the analyzer computes the number of facts or valid instantiations in the DOB (cardinality), and the number of different values for each of its arguments (nKeys); for each IOB predicate, an adaptive sampling algorithm [10] is applied to compute cardinality and cost estimates.

When an agent formulates a conjunctive query, the DOB-S system's **optimizer** generates an efficient query evaluation plan. A dynamic-programming optimizer is based in a **hybrid cost model**: it uses the ontology's EOB and IOB statistics, and estimates the cost of a query according the different evaluation strategies implemented. Finally, an **execution engine** evaluates the query plan and produces a query answer.

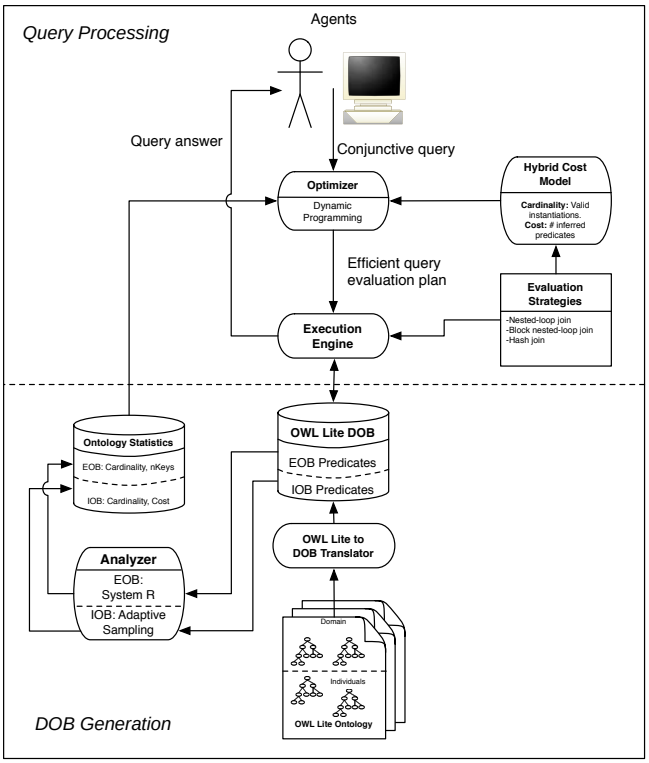


Fig. 1. DOB-S System Architecture

4 OWL Lite DOB

An OWL Lite ontology contains: (1) a set of axioms that provides information about classes and properties, and (2) a set of facts that represents individuals in the ontology, the classes they belong to, and the properties they participate in.

Restrictions allow the construction of class definitions by restricting the values of their properties and their cardinality. Classes may also be defined through the intersection of other classes. Object properties represent binary relationships between individuals; datatype properties correspond to relationships between individuals and data values belonging to primitive datatypes.

The subset of OWL Lite represented as a DOB does not include domain and range class intersection. The `someValuesFrom` restriction is not included as it involves an existential quantifier and cannot be translated to Datalog. Primitive datatypes are not handled; therefore, we do not represent ranges for Datatype properties[1].

[1] `EquivalentClasses`, `EquivalentProperties`, and `allDifferent` axioms, and the `cardinality` restriction are not represented as they are syntactic sugar for other language constructs.

4.1 OWL Lite DOB Syntax

Our formalism, DOB, provides a set of EOB built-in predicates that represents all the axioms and restrictions of an OWL Lite subset.

EOB predicates are *ground*, i.e., no variables are allowed as arguments. A set of IOB built-in predicates represents the semantics of the EOB predicates. We have followed the OWL Web Ontology Language Overview presented in [12].

Table 1 illustrates the EOB and IOB built-in predicates for an OWL Lite subset[2]. Note that some predicates refer to domain concepts (e.g. `isClass, areClasses`), and some to individuals (e.g. `is isIndividual, areIndividuals`).

EOB PREDICATE	**DESCRIPTION**
`isOntology(O)`	An ontology has an Uri `O`
`isImpOntology(O1,O2)`	Ontology `O1` imports ontology `O2`
`isClass(C,O)`	`C` is a class in ontology `O`
`isOProperty(P,D,R)`	`P` is an object property with domain `D` and range `R`
`isDProperty(P,D)`	`P` is a datatype property with domain `D`
`isTransitive(P)`	`P` is a transitive property
`subClassOf(C1,C2)`	`C1` is subclass of `C2`
`AllValuesFrom(C,P,D)`	`C` has property `P` with all values in `D`
`isIndividual(I,C)`	`I` is an individual belonging to class `C`
`isStatement(I,P,J)`	`I` is an individual that has property `P` with value `J`
IOB PREDICATE	**DESCRIPTION**
`areSubClasses(C1,C2)`	`C1` are the direct and indirect subclasses of C2
`areImpOntologies(O1,O2)`	`O1` import the ontologies `O2` directly and indirectly
`areClasses(C,O)`	`C` are all the classes of an ontology and its imported ontologies `O`
`areIndividuals(I,C)`	`I` are the individuals of a class and all of its direct and indirect superclasses `C`; or `I` are the individuals that participate in a property and belong to its domain or range `C`, or are values of a property with all values in `C`

Table 1. Some built-in EOB and IOB Predicates for a subset of OWL Lite

OWL ABSTRACT SYNTAX	EOB PREDICATES
$Ontology(O)$	`isOntology(O)`
$Individual(O1\ value(owl:imports\ O2))$	`impOntology(O1, O2)`
$Ontology(O)$, $Class(C\ partial$	`isClass(C,O)`
$Class(A\ partial\ C)$	`subClassOf(A,C)`
$Class(C1\ partial\ restriction(P\ allValuesFrom(C2)))$	`allValuesFrom(C1,P,C2)`
$Class(A\ partial\ C1 \ldots Cn)$	`subClassOf(A,C1),...,` `subClassOf(A,Cn)`
$ObjectProperty(P\ domain(D))$, $ObjectProperty(P\ range(R))$	`isOProperty(P,D,R)`
$DatatypeProperty(P\ domain(D))$	`isDProperty(P,D)`
$Property(P\ Transitive)$	`isTransitive(P)`
$Individual(I\ type(C))$	`isIndividual(I,C)`
$Individual(I\ value(P\ J))$	`isStatement(I,P,J)`

Table 2. Mapping OWL Lite subset to EOB Predicates

[2] We assume that the class *owl:Thing* is the default value for the domain and range of a property.

OWL LITE INFERENCE RULES	IOB RULE DEFINITIONS

If `subClassOf(C1,C2)` and `subClassOf(C2,C3)` then `subClassOf(C1,C3)`	`areSubClasses(C1,C2):-subClassOf(C1,C2).`
	`areSubClasses(C1,C2):-subClassOf(C1,C3),`
	`areSubClasses(C3,C2).`

If `impOntology(O1,O2)` and `impOntology(O2,O3)` then `impOntology(O1,O3)`	`areImpOntologies(O1,O2):-impOntology(O1,O2).`
	`areImpOntologies(O1,O2):-impOntology(O1,O3),`
	`areImpOntologies(O3,O2).`

If `isClass(C1,O2)` and `impOntology(O1,O2)` then `isClass(C1,O1)`	`areClasses(C,O):-isClass(C,O).`
	`areClasses(C,O1):-isClass(C,O2),`
	`areImpOntologies(O1,O2).`

If `isSubClassOf(C1,C2)` and `isIndividual(I,C1)` then `isIndividual(I,C2)`	`areIndividuals(I,C):-isIndividual(I,C).`
	`areIndividuals(I,C2):-isIndividual(I,C1),`
	`areSubClasses(C1,C2).`

If `isStatement(I,P,J)` and `isOProperty(P,C,R)` then `isIndividual(I,C)`	`areIndividuals(I,C):-isOProperty(P,C,R),` `areStatements(I,P,J).`
If `isStatement(I,P,J)` and `isOProperty(P,D,C)` then `isIndividual(J,C)`	`areIndividuals(J,C): isOProperty(P,D,C),` `areStatements(I,P,J).`
If `isStatement(I,P,J)` and `isDProperty(P,C)` then `isIndividual(I,C)`	`areIndividuals(I,C):-isDProperty(P,C),` `areStatements(I,P,J).`

If `AllValues(C1,P,C)` and `isStatement(I,P,J)` and `isIndividual(I,C1)` then `isIndividual(J,C)`	`areIndividuals(J,C):-isIndividual(I,C1),`
	`allValuesFrom(C1,P,C),`
	`areStatements(I,P,J).`

Table 3. Mapping OWL Lite subset Inference Rules to IOB Predicates

4.2 OWL Lite DOB Semantics

A model-theoretic semantics for an OWL Lite (subset) DOB is as follows:

Definition 8 (Interpretation) *An **Interpretation** $I = (\Delta^I, \mathcal{P}^I, .^I)$ consists of:*

- *A non-empty interpretation domain Δ^I corresponding to the union of the sets of valid URIs of ontologies, classes, object and datatype properties, and individuals. These sets are pairwise disjoint.*
- *A set of interpretations $\mathcal{P}^I$, of the EOB and IOB built-in predicates in Table 1.*
- *An interpretation function $.^I$ which maps each n-ary built-in predicate $p^I \in \mathcal{P}^I$ to an n-ary relation $\prod_{i=1}^{n} \Delta^I$.*

Definition 9 (Satisfiability) *Given an OWL Lite DOB $\mathcal{D}$, an interpretation I, and a predicate $p \in \mathcal{D}$, $I \models p$ iff:*

- *p is an EOB predicate $p(t_1, ..., t_n)$ and $(t_1, ..., t_n) \in p^I$.*
- *p is an IOB predicate $R{:}H(\overline{X}) \leftarrow \exists \overline{Y} B(\overline{X}, \overline{Y})$, and whenever I satisfies each predicate in the body B, I also satisfies the predicate in the head H.*

Definition 10 (Model) *Given an OWL Lite DOB $\mathcal{D}$ and an interpretation I, I is a **model** of $\mathcal{D}$ iff for every predicate $p \in \mathcal{D}$, $I \models p$.*

4.3 Translation of OWL Lite to OWL Lite DOB

A definition of a translation map from OWL Lite to OWL Lite DOB is the following:

Definition 11 (Translation) *Given an OWL Lite theory $\mathcal{O}$ and an OWL Lite DOB theory $\mathcal{D}$, an* ***OWL Lite to DOB Translation*** *$\mathcal{T}$ is a function $\mathcal{T} : \mathcal{O} \rightarrow \mathcal{D}$.*

Given an OWL Lite ontology $\mathcal{O}$, an OWL Lite DOB ontology $\mathcal{D}$ is defined as follows:

- (Base Case) If o is an axiom or fact belonging to the sets of axioms or facts of $\mathcal{O}$, then an EOB predicate $\mathcal{T}(o)$ is defined according to the EOB mappings in Table 2.
- If o is an OWL Lite inference rule, then an IOB predicate $\mathcal{T}(o)$ is defined according to the IOB mappings in Table 3.

The translation ensures that the following theorem holds:

Theorem 1 *Let $\mathcal{O}$ and $\mathcal{D}$ be OWL Lite and OWL Lite DOB theories respectively, and $\mathcal{T}$ be an OWL Lite to DOB Translation such that, $\mathcal{T}(\mathcal{O}) = \mathcal{D}$, then $\mathcal{D} \models \mathcal{O}$.*

5 A Motivating Example

Consider a 'cars and dealers' domain ontology `carsOnt` and Web source ontologies `source1` and `source2`. Source `source1` publishes information about all types of vehicles and dealers, whereas `source2` is specialized in SUVs.

The OWL Lite ontologies can be seen in Table 4.

Ontology carsOnt	**Ontology source1**	**Ontology source2**
`Class vehicle partial Thing)`	`imports carsOnt`	`imports carsOnt`
`SubClassOf(suv,vehicle)`		`individual(s123 type(suv))`
`SubClassOf(car,vehicle)`		
`Property(price domain(vehicle))`		
`Class dealer partial Thing)`		
`Property(sells domain(dealer))`		
`Property(sells range(vehicle))`		
`Property(traction domain(suv))`		
`Property(model domain(vahicle))`		

Table 4. Example OWL Lite ontology

A portion of the example's EOB can be seen in Table 5.

EOB PREDICATES		
`isOntology(carsOnt)`	`isOntology(source1)`	`isOntology(source2)`
`impOntology(source1,carsOnt)`	`impOntology(source2,carsOnt)`	`isClass(vehicle,carsOnt)`
`isClass(vehicle,carsOnt)`	`isClass(dealer,carsOnt)`	`subClassOf(car,vehicle)`
`subClassOf(suv,vehicle)`	`isOProperty(sells,dealer,vehicle)`	`isDProperty(model,vehicle)`
`isDProperty(price,vehicle)`	`isDProperty(traction,suv)`	`isIndividual(s123,suv)`

Table 5. Example DOB ontology

To illustrate a rule evaluation, we will take a query `q` that asks for *the Web sources that publish information about 'traction'*:

```
q(O):-areClasses(C,O),isDProperty(traction,C).
```

The answer to this query corresponds to all the ontologies with classes characterized by the property `traction`, i.e., ontologies `source1`, `source2` and `carsOnt`.

If we invert the ordering of the first two predicates in `q`, we will have an equivalent query `q'`:

```
q'(O):-isDProperty(traction,C),areClasses(C,O).
```

The cost or total number of inferred facts for `q` is larger than the cost for `q'`. In `q`, the number of instantiations or cardinality for the first intensional predicate `areClasses(C,O)` is twelve, four for each ontology, as `source1` and `source2` inherit the classes in `carsOnt`. The cost of inferring these facts is dependent on the cost of evaluating the `areClasses` rule. In `q'`, for the first subgoal `isDProperty(traction,C)`, we have one instantiation: `isDProperty(traction,suv)`. Again, the cost of inferring this fact depends on the cost of the `isDProperty` predicate.

Note that statistics on the size and argument values of the EOB `isDProperty` predicate can be computed, whereas statistics for the IOB `areClasses` predicate will have to be estimated as data is not known a priori. Once the cost of each query predicate is determined, we may apply a cost-based join-ordering optimization strategy.

6 DOB Hybrid Cost Model

The process of answering a query relies on inferring facts from the predicates in the DOB. Our cost metric is focused on the number of intermediate facts that need to be inferred in order to answer the query. The objective is to find an order of the predicates in the body of the query, such that the number of intermediate inferred facts is reduced. We will apply a join-ordering optimization strategy *à* la System R using Datalog-relational equivalences [1]. To estimate the cardinality and evaluation cost of the intensional predicates, we have applied an adaptive sampling technique. Thus, we propose a hybrid cost model which combines adaptive sampling and traditional relational cost models.

6.1 Adaptive Sampling Technique

We have developed a sampling technique that is based on the *adaptive sampling method* proposed by Lipton, Naughton, and Schneider [10, 11]. This technique assumes that there is a population $\mathbb{P}$ of all the different valid instantiations of a predicate P, and that $\mathbb{P}$ is divided into n partitions according to the n possible instantiations of one or more arguments of P. Each element in $\mathbb{P}$ is related to its evaluation cost and cardinality, and the population $\mathbb{P}$ is characterized by the statistics mean and variance.

The objective of the sampling is to identify a sample of the population $\mathbb{P}$, called $\mathbb{EP}$, such that the mean and variance of the cardinality (resp. evaluation cost) of $\mathbb{EP}$ are valid to within a predetermined accuracy and confidence level.

To estimate the mean of the cardinality (resp. cost) of $\mathbb{EP}$, say $\overline{Y}$, within $\frac{\overline{Y}}{d}$ with probability p, where $0 \leq p < 1$ and $d > 0$, the sampling method assumes an *urn* model.

The urn has n balls from which m samplings are repeatedly taken, until the sum z of the cardinalities (resp. costs) of the samples is greater than $\alpha \times (\frac{S}{Y})$, where $\alpha = \frac{d \times (d+1)}{(1-\sqrt{p})}$. The estimated mean of the cardinality (resp. cost) is: $\overline{Y} = \frac{z}{m}$.

The values d and $\frac{1}{(1-\sqrt{p})}$ are associated with the relative error and the confidence level, and S and Y represent the cardinality (resp. cost) variance and mean of $\mathbb{P}$. Since statistics of $\mathbb{P}$ are unknown, the upper bound $\alpha \times \frac{S}{Y}$ is replaced by $\alpha \times b(n)$.

To approximate $b(n)$ for cost and cardinality estimates, we apply Double Sampling [9]. In the first stage we randomly evaluate k samples and take the maximum value among them:

$b(n) = max_{i=1}^{k}(card(P_i))$ (resp. $b(n) = max_{i=1}^{k}(cost(P_i))$), where $1 \leq k \leq n$

It has been shown that a few samples are necessary in order for the distribution of the sum to begin to look normal. Thus, the factor $1/(1-\sqrt{p})$ may be improved by central limit theorem [11]. This improvement allows us to achieve accurate estimations and lower bounds.

Estimating cardinality. Given an intensional predicate P, the *cardinality* of P corresponds to the number of the valid instantiations of P (Definition 6). In our previous example, the number of ontology values obtained in the answer of the query is estimated using this metric.

To estimate the cardinality of P, we execute the adaptive sampling algorithm explained before, by selecting any argument of P, and partitioning $\mathbb{P}$ according to the chosen argument. The cardinality estimation will be $card(P) = \overline{Y} \times n$, where n is the number of partitions, i.e. the number of different instantiations for the chosen argument.

Note that once the cardinality of the non-instantiated P is estimated, we can estimate the cardinality of the instantiated predicate by using the selectivity value(s) of the instantiated argument(s).

Estimating cost. The *cost* of P measures the number of intermediate inferred facts (Definition 7). For instance, to estimate the cost of a predicate $P(X, Y)$, we consider the different instantiation patterns that the predicate can have, i.e., we independently estimate the cost for $P(X^b, Y^b)$, $P(X^b, Y^f)$, $P(X^f, Y^b)$ and $P(X^f, Y^f)$, where b and f indicate that the argument is bound and free, respectively.

The computation of several cost estimates is necessary because in Datalog top-down evaluation [1], the cost of an instantiated intensional predicate cannot

be accurately estimated from the cost of a non-instantiated predicate (using selectivity values). Instantiated arguments will propagate in the IOB rule's body through sideways-passing, and cost varies according to the binding patterns. For example, the cost of `areClasses(C1`^{b}`,C2`^{f}`)` may be smaller than the cost of `areClasses(C1`^{f}`,C2`^{b}`)`, i.e., the bound argument `C1` "pushes" instantiations in the definition of the rule:

```
areSubClasses(C1,C2):-isSubClass(C1,C3),areSubClasses(C3,C2).
```

making its body predicates more selective.

For $P(X^b, Y^b)$, $P(X^b, Y^f)$ and $P(X^f, Y^b)$, we partition $\mathbb{P}$ according to the bound arguments. In these cases we are estimating the cost of one partition. Therefore, $cost(P) = \frac{\overline{Y} \times n}{n} = \overline{Y}$.

Finally, to estimate the cost of $P(X^f, Y^f)$, we choose an argument of P and partition P according to the chosen argument. To reduce the cost of computing the estimate, we choose the most selective argument. The cost estimate is $cost(P) = \overline{Y} \times n$.

Determining the number of partitions n. For both, cost and cardinality estimates, we need to determine the number of possible instantiations, n, of the chosen argument. This value depends on the semantics of the particular predicate. For instance, for an interpretation I, $areClasses(Class, Ont)^I \subseteq \mathcal{C} \times \mathcal{O}$, where $\mathcal{C}$ is the set of valid class URIs and $\mathcal{O}$ is the set of valid ontology URIs. $|\mathcal{C}|$ corresponds to the number of EOB predicates $isClass(Class, Ont)$, i.e. $|\mathcal{C}|$=$Card(isClass(Class, Ont))$. Similarly, $|\mathcal{O}|$=$Card(isOntology(Ont))$; these cardinalities have been computed previously. We assume that the values are uniformly distributed.

6.2 System R Technique

To estimate the cardinality and cost of two or more predicates, we use the cost model proposed in System R. The cardinality of the conjunction of predicates P_1,P_2 is described by the following expression:

$$card(P_1, P_2) = card(P_1) \times card(P_2) \times reductionFactor(P_1, P_2)$$

$reductionFactor(P_1, P_2)$ reflects the impact of the sideways passing variables in reducing the cardinality of the result. This value is computed assuming that sideways passing variables are independent and each is uniformly distributed [18]. For cost estimation, we consider three evaluation strategies:

1. Nested-Loop Join
   Following a Nested-Loop Join evaluation strategy, for each valid instantiation in P_1, we retrieve a valid instantiation in P_2 with a matching "join" argument value:

   $$cost(P_1, P_2) = cost(P_1) + card(P_1) \times cost^{inst}(P_2)$$

   $cost^{inst}(P_2)$ corresponds to the estimate of the cost of the predicate P_2 where the "join" arguments are instantiated in P_2, i.e., all the sideways

passing variables from P_1 to P_2 are bound in P_2. These binding patterns were considered during the sampling-based estimation of the cost of P_2.

2. Block Nested-Loop Join
   Predicate P_1 is evaluated into blocks of fixed size, and then each block is "joined" with P_2.

$$cost(P_1, P_2) = cost(P_1) + \lceil \frac{card(P_1)}{BlockSize} \rceil \times cost(P_2)$$

3. Hash Join
   A hash table is built for each predicate according to their join argument. The valid instantiations of predicates P_1 and P_2 with the same hash key will be joined together:

$$cost(P_1, P_2) = cost(P_1) + cost(P_2)$$

Although the sampling technique is appropiate for estimating a single predicate, it may be inefficient for estimating the size of a conjunction of more than two predicates.

The sampling algorithm in [10] suggests that for a conjunction of 2 predicates, P, Q, if the size of P is n, the query is n-partitionable, that is, for each valid instantiation p in P, the corresponding partition of Q is all the valid instantiations q in Q such that q "joins" p. Therefore, when the size of the first predicate in a query is small, our sample size may be larger. This problem can be extended to conjunctive queries with several subgoals, so when the number of intermediate results is small, sampling time may be as large as evaluation time.

6.3 Query Optimization

In Figure 2 we present the algorithm used to optimize the body of a query. The proposed optimization algorithm extends the System R dynamic-programming algorithm by identifying orderings of the n EOB and IOB predicates in a query. During each iteration of the algorithm, the best intermediate sub-plans are chosen based on cost and cardinality. In the last iteration, final plans are constructed and the best plan is selected in terms of the cost metric.

During each iteration i between 2 and *n-1*, different orderings of the predicates are analyzed. Two subplans are considered equivalents if and only if, they are composed by the same predicates. A subplan SP_i is better than a subplan SP_j if and only if, the cost and cardinality of SP_j are greater than cost and cardinality of SP_i. If SP_i cost is greater than SP_j cost, but SP_j cardinality is greater than SP_i cardinality, i.e. they are un-comparable, then the equivalence class is annotated with the two subplans.

7 Experimental Results

An experimental study was conducted for synthetic and real-world ontologies. Experiments on synthetic ontologies were executed on a SunBlade 150 (650MHz)

Algorithm Dynamic Programming
INPUT: `Predicate`: a set of predicates, P_1,...,P_n.
OUTPUT: `OrderedPredicate`: an ordering of `Predicate`

1. `SubPaths`=`Predicate`;
2. For i=1 to n
   (a) For each solution Sub_j in `SubPaths`
      i. For each predicate P_z in `Predicate`
         – If there are sideways passing variables from Sub_j to P_z, then add Sub= Sub_j,P_z to `NewSubPaths`
   (b) Remove from `NewSubPaths` any subpath Sub_k iff there is another subpath Sub_l in `NewSubPaths`, such that, Sub_l and Sub_k are **equivalent**, and Sub_l is **better** than Sub_k.
   (c) `SubPaths`=`NewSubPaths`
   (d) Reset `NewSubPaths`
3. Return the path in `SubPaths` with lowest cost.

Fig. 2. Query Optimization Algorithm

with 1GB RAM; experiments on real-world ontologies were executed on a Sun-Fire V440 (1281MHz) with 4GB RAM. Our system was implemented in SWI-Prolog 5.6.1.

We have studied three real-world ontologies: Travel [19], EHR_RM [21], and Galen [14].

Our cost metrics are the number of intermediate facts for synthetic and real-world ontologies, and the evaluation time for real-world ontologies. In our experiments, the sampling parameters d (the error), p (the confidence level), and k (the size of the sample for the first stage) were set to 0.2, 0.7 and 7, respectively. Also, these experiments only considered the Nested-Loop Join evaluation strategy.

Our study consisted of the following:

– *Cost Model Predictive Capability*: In Figure 3a, we report the correlation among the estimated values and the actual cost for synthetic ontologies. Synthetic ontologies were randomly generated following a uniform distribution. We generated ten ontology documents and three chain and star queries with three subgoals for each ontology; the cost of each ordering was estimated with our cost model, and each ordering was then evaluated against the ontology; this gives us a total of six hundred queries. The correlation is 0.92.
  In Figure 3b, we report the same correlation for the real-world ontology Galen. Correlation values are 0.86 for Travel, 0.54 for EHR_RM, and 0.62 for Galen.
– *Cost improvements*: We also conducted experiments to study cost improvement using the optimizer. For each query, we evaluated all its orderings, then we ran the optimizer and evaluated the optimized query. Figure 3c reports the ratio of the cost of the optimal ordering to the cost of the worst ordering, $\frac{costOptimalOrdering}{costWorstOrdering}$, for queries against synthetic ontologies. In Figure 3d, we report this metric for Galen. Both in synthetic and real-world ontologies, this ratio is less than 10% for most of the queries. We also computed the proportion of the optimal ordering cost with respect to the median ordering

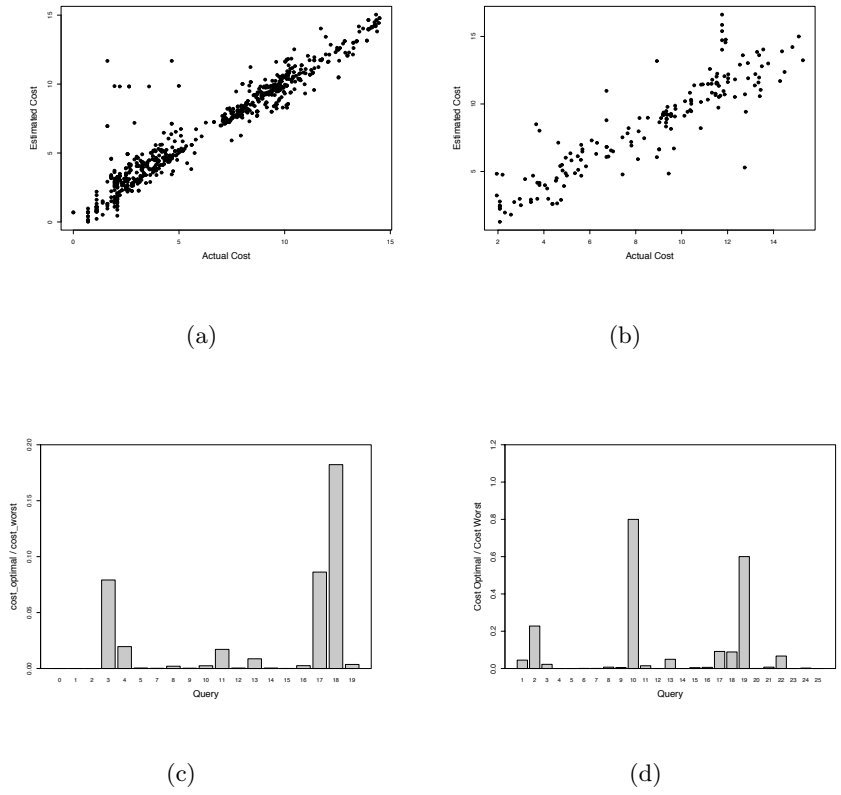


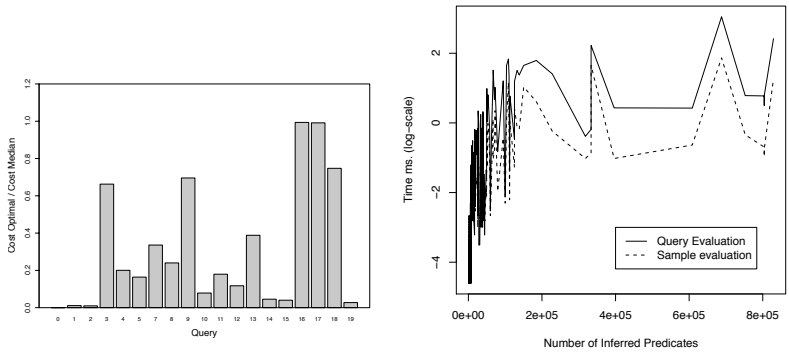
Fig. 3. (a) Correlation of estimated cost to actual cost (log. scale) - Synt. ontologies; (b) Correlation of estimated cost to actual cost (log. scale) - GALEN; (c) #Pred. optimal ordering vs. #Pred. worst ordering - Synt. Ontologies; (d) #Pred. optimal ordering vs. #Pred. worst ordering - GALEN

cost. The results for synthetic ontologies show that the optimal ordering cost is less than 40% of the median for fifteen of twenty queries; this result can be observed in Figure 4a.

Correlation results show that estimates produced by our cost model are quite accurate. The lower correlation results for the EHR_RM and Galen ontologies are related to the uniform distribution assumption of our cost model.

Additionally, the results show a significant improvement in the evaluation cost for the optimized queries with respect to the worst-case and median-case query orderings. This property holds for synthetic and real-world ontologies. However, for synthetic ontologies we notice that for star-shaped queries, the difference between the median cost and the optimal cost is very small; this

indicates that the form of the query may influence the cost improvement achieved by the optimizer.



(a) (b)

Fig. 4. (a) #Pred. optimal ordering vs. #Pred. median ordering - Synt. Ontologies; (b) Sampling Conjunctions - Query Eval. time and Sample Eval. time vs. # Inf. Pred.

Finally, we would like to point out that we also studied the use of an adaptive sampling technique for the cost estimation of the conjunction of two or more predicates (instead of System R cost model). Although, the sampling technique gives a better correlation result than the combination of sampling and System R cost model, the time required to compute the cost estimation may be as large as the time needed to evaluate the query. In 4b we can observe that the time difference is marginal.

8 Conclusions and Future Work

We have developed a cost model that combines System R and adaptive sampling techniques. Adaptive sampling is used to estimate data that do not exist a priori, data related to the cardinality and cost of intensional rules in the DOB. The experimental results show that our proposed techniques produce in general a significant improvement in the evaluation cost for the optimized query.

Currently we are concluding an experimental study that considers the three evaluation strategies: Nested-Loop, Block Nested-Loop, and Hash Join; query plans now include orderings with different combinations of these evaluation operators. Initial results show correlation values among estimated and actual cost of approximately 0.8 for real-world ontologies. We also plan to apply similar optimization techniques for conjunctive queries to DL ontologies. Initially, we will

work on ABox queries extending the the techniques proposed in [20]. In a next stage, we will consider mixed TBox and ABox conjunctive queries.

References

1. S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases. *Addison-Wesly Publishing Co*, 1995.
2. J. Bruijn, A. Polleres, and D.Fensel. OWL Lite- WSML working draft. DERI institute, 2004.
3. D. Calvanese, G. Di Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tailoring OWL for data intensive ontologies. In *Proc. of the Workshop on OWL: Experiences and Directions*, 2005.
4. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Towards efficient evaluation of hex-programs. In *11TH Nonmonotonic Reasoning Workshop*, 2006.
5. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. of WWW*, 2003.
6. V. Haarslev and R. Moller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *ICAI: Proceedings of the sixth AAAI*. AAAI, 2004.
7. I. Horrocks and D. Turi. The OWL instance store: System description. In *CADE-20: 20th International Conference on Automated Deduction*, 2005.
8. Ul. Hustadt and B. Motik. Description logics and disjunctive datalog the story so far. In *DL 2005 - Description Logics 2005*, 2005.
9. Y. Ling and W. Sun. A supplement to sampling-based methods for query size estimation in a database system. *SIGMOD RECORD*, 1992.
10. R. Lipton and J. Naughton. Query size estimation by adaptive sampling (extended abstract). *Proceedings of ACM Sigmod*, pages 40–46, 1990.
11. R. Lipton, J. Naughton, and D. Schneider. Practical selectivity estimation through adaptive sampling. *Proceedings of ACM Sigmod*, pages 1–10, 1990.
12. D. McGuinness and F. van Harmelen. OWL web ontology language overview. *W3C Recommendation*, 2004.
13. B. Motik, R. Volz, and A. Maedche. Optimizing query answering in description logics using disjunctive deductive databases. In *KRDB*, 2003.
14. Open Clinical Organization. GALEN common reference model.
15. Z. Pan and J. Hefflin. DLDB: Extending relational databases to support semantic web queries. In *PSSS-03: Practical and Scalable Semantic Systems*, 2003.
16. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Mc Graw Hill, 2003.
17. R. Ramakrishnan and J. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 1993.
18. P. Selingerl, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. *Proceedings of ACM Sigmod*, 1979.
19. M. Shell. SchemaWeb website, 2002.
20. E. Sirin and B. Parsia. Optimizations for answering conjunctive abox queries. In *DL-06: International Workshop on Description Logics*, 2006.
21. Protege staff. Protege OWL: Ontology editor for the semantic web.
22. M. Staudt, R. Soiron, C. Quix, and M. Jarke. Query optimization for repository-based applications. In *Selected Areas in Cryptography*, 1999.
23. T. Wang. Gauging ontologies and schemas by numbers. *In WWW06 Workshop Proceedings EON2006*, 2006.

dlvhex: A Tool for Semantic-Web Reasoning under the Answer-Set Semantics*

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits

Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9–11, A-1040 Vienna, Austria
{eiter,ianni,roman,tompits}@kr.tuwien.ac.at

Abstract. We briefly report about the development status of dlvhex, a reasoning engine for HEX-programs, which are nonmonotonic logic programs featuring both higher-order atoms as well as external ones. Higher-order features are widely acknowledged as useful for various tasks and are essential in the context of meta-reasoning. Furthermore, the possibility to exchange knowledge with external sources in a fully declarative framework such as answer-set programming (ASP) is particularly important in view of applications in the Semantic-Web area. Through external atoms, HEX-programs can deal with external knowledge and reasoners of various nature, such as RDF datasets or description-logic bases.

1 Introduction

Nonmonotonic semantics is often requested by Semantic-Web designers in cases where the reasoning capabilities of the *Ontology Layer* of the Semantic Web turn out to be too limiting, since they are based on monotonic logics. The widely acknowledged answer-set semantics of nonmonotonic logic programs [5], which is arguably the most important instance of the *answer-set programming* (ASP) paradigm, is a natural host for giving nonmonotonic semantics to the *Rules*, *Logic*, and *Proof Layers* of the Semantic Web.

However, for important issues such as *meta-reasoning* in the context of the Semantic Web, no adequate answer-set engines have been made available so far. Motivated by this fact and the observation that, furthermore, interoperability with other software is an important issue (not only in this context), in previous work [3], the answer-set semantics has been extended to HEX *programs*, which are *higher-order logic programs* (which accommodate meta-reasoning through *higher-order atoms*) with *external atoms* for software interoperability. Intuitively, a higher-order atom allows to quantify values over predicate names, and to freely exchange predicate symbols with constant symbols, like in the rule $C(X) \leftarrow subClassOf(D, C), D(X)$. An external atom facilitates the assignment of a truth value of an atom through an external source of computation. For instance, the rule $t(Sub, Pred, Obj) \leftarrow \&RDF[uri](Sub, Pred, Obj)$ computes the predicate t taking values from the predicate $\&RDF$. The latter predicate extracts RDF

* This work was partially supported by the Austrian Science Fund (FWF) under grant P17212-N04, and by the European Commission through the REWERSE IST Network of Excellence (IST-2003-506779).

statements from the set of URIs specified by the extension of the predicate uri; this task is delegated to an external computational source (e.g., an external deduction system, an execution library, etc.). External atoms allow for a bidirectional flow of information to and from external sources of computation such as description-logic reasoners. By means of HEX-programs, powerful meta-reasoning becomes available in a decidable setting, e.g., not only for Semantic-Web applications, but also for meta-interpretation techniques in ASP itself, or for defining policy languages.

Other logic-based formalisms, like TRIPLE [10] or F-Logic [8], feature also higher-order predicates for meta-reasoning in Semantic-Web applications. Our formalism is fully declarative and offers the possibility of nondeterministic predicate definitions with higher complexity in a decidable setting. This proved already useful for a range of applications with inherent nondeterminism, such as ontology merging (cf. [11]) or matchmaking, and thus provides a rich basis for integrating these areas with meta-reasoning.

2 HEX-Programs

2.1 Syntax

HEX-programs are built on mutually disjoint sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ of *constant names*, *variable names*, and *external predicate names*, respectively. Unless stated otherwise, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are written with first letter in upper case (resp., lower case), and elements from $\mathcal{G}$ are prefixed with "&". Constant names serve both as individual and predicate names. Importantly, $\mathcal{C}$ may be infinite.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms and $n \geq 0$ is its *arity*. Intuitively, Y_0 is the predicate name; we thus also use the familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant. For example, $(x, rdf{:}type, c)$ and $node(X)$ are ordinary atoms, while $D(a, b)$ is a higher-order atom. An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m), \tag{1}$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input list* and *output list*, respectively), and $\&g$ is an *external predicate name*.

It is possible to specify *molecules* of atoms in F-Logic-like syntax. For instance, $gi[father \rightarrow X, Z \rightarrow iu]$ is a shortcut for the conjunction $father(gi, X)$, $Z(gi, iu)$.

HEX-programs are sets of rules of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m, \tag{2}$$

where $m, k \geq 0$, $\alpha_1, \ldots, \alpha_k$ are higher-order atoms, and $\beta_1, \ldots, \beta_m$ are either higher-order atoms or external atoms. The operator "not" is *negation as failure* (or *default negation*).

2.2 Semantics

The semantics of HEX-programs is given by generalizing the answer-set semantics [3]. The *Herbrand base* of a program P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with

constants from $\mathcal{C}$. An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing only atoms.

We say that an interpretation $I \subseteq HB_P$ is a *model* of an atom $a \in HB_P$ iff $a \in I$. Furthermore, I is a model of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$ iff $f_{\&g}(I, y_1 \ldots, y_n, x_1, \ldots, x_m) = 1$, where $f_{\&g}$ is an $(n+m+1)$-ary Boolean function associated with $\&g$, called *oracle function*, assigning each element of $HB_P \times \mathcal{C}^{n+m}$ either 0 or 1 (i.e., *false* or *true*, respectively).

This definition of satisfaction, together with a modified notion of a *reduct* as defined by Faber *et al.* [4], enables us to define a conservative extension of the answer-set semantics for HEX-programs. For more details, cf. [3].

Note that the answer-set semantics may yield multiple models (i.e., answer sets) in general. Therefore, for query answering, *brave* and *cautious reasoning* (truth in some resp. all models) is considered in practice, depending on the application.

2.3 Usability of HEX-Programs

An interesting application scenario, where several features of HEX-programs come into play, is *ontology alignment*. Merging knowledge from different sources in the context of the Semantic Web is a crucial task [2] that can be supported by HEX-programs in various ways:

Importing external theories. This can be achieved as in the following manner:

$$\begin{aligned} triple(X, Y, Z) &\leftarrow \&RDF[uri](X, Y, Z), \\ triple(X, Y, Z) &\leftarrow \&RDF[uri2](X, Y, Z), \\ proposition(P) &\leftarrow triple(P, rdf{:}type, rdf{:}Statement). \end{aligned}$$

Searching in the space of assertions. In order to choose nondeterministically which propositions have to be included in the merged theory and which not, statements like the following can be used:

$$pick(P) \vee drop(P) \leftarrow proposition(P).$$

Translating and manipulating reified assertions. E.g., it is possible to choose how to put RDF triples (possibly including OWL assertions) in an easier manipulable and readable format, and to make selected propositions true such as in the following way:

$$\begin{aligned} (X, Y, Z) &\leftarrow pick(P), triple(P, rdf{:}subject, X), triple(P, rdf{:}predicate, Y), \\ &\qquad triple(P, rdf{:}object, Z), \\ C(X) &\leftarrow (X, rdf{:}type, C). \end{aligned}$$

Defining ontology semantics. The semantics of the ontology language at hand can be defined in terms of entailment rules and constraints expressed in the language itself or in terms of external knowledge, like in

$$D(X) \leftarrow subClassof(D, C), C(X) \quad \text{and} \quad \leftarrow \&inconsistent[pick],$$

where the external predicate $\&inconsistent$ takes a set of assertions as input and establishes through an external reasoner whether the underlying theory is inconsistent.

Performing default and closed-world reasoning. Assuming that a generic external atom $\&DL[C](X)$ is available for querying the concept C in a given description logics base, the *closed-world assumption* (CWA) can be stated as

$$C'(X) \leftarrow not\ \&DL[C](X), concept(C), cwa(C, C'),$$

where $concept(C)$ is a predicate which holds for all concepts and $cwa(C, C')$ states that C' is the CWA of C, i.e., each individual not explicitly found in C should be in C'.

Inconsistency of the CWA can be checked by pushing back inferred values to the external knowledge base:

$$\begin{aligned} set\_false(C, X) &\leftarrow cwa(C, C'), C'(X), \\ inconsistent &\leftarrow \&DL1[set\_false](b), \end{aligned}$$

where $\&DL1[N](X)$ effects a check whether a knowledge base, augmented with all negated facts $\neg c(a)$ such $N(c, a)$ holds, entails the empty concept $\bot$ (entailment of $\bot(b)$, for any constant b, is tantamount to inconsistency).

3 Implementation

The evaluation principle of dlvhex is to split the program according to its dependency graph into components and alternately call an answer-set solver (DLV [9]) and the external atom functions for the respective subprograms. The framework takes care of traversing the tree of components in the right order and combining their resulting models. Composing the initial dependency graph from a nonground program is not a trivial task, since higher-order atoms as well as the input list of an external atom have to be considered. To this end, we defined a novel notion of *atom dependency*, which extends the traditional understanding of dependencies within a logic program. This leads to novel types of *stratification* which help splitting a HEX-program and choosing the suitable model generation strategies.

Further methods of increasing the efficiency of computation include a general classification of external atoms regarding their functional properties. For instance, their evaluation functions may be *monotonic* or *linear* (in the sense of a linear function) with respect to a given input. Formalizing such knowledge allows for an intelligent caching algorithm and thus for a reduction of interactions with the external computation source. Latest developments also include a directive to syntactically handle namespaces and an algorithm for traversing the component graph for disjunctive programs, eventually implementing the full HEX-program semantics.

To keep the development and usage of external atoms as flexible as possible, we decided to embed them into *plug-ins*, i.e., libraries that define and provide one or more external atoms. Such plug-ins are implemented as shared libraries, which link dynamically to the main application at runtime. A lean, object-oriented interface reduces the effort of developing custom plug-ins to a minimum.

Currently, dlvhex provides the following extension to pure HEX-reasoning: (i) parsing both templates as well as frame syntax by using DLT [7] as a preparser; (ii) in addition to strict constraints, accepting *weak constraints* for optimization problems; and (iii) returning the result in XML syntax according to the RuleML specification [1].

The following external atoms are available in dlvhex:

The RDF plug-in. The RDF plug-in provides a single external atom, the $\&rdf$-atom, which enables the user to import RDF-triples from any RDF knowledge base. It takes a single constant as input, which denotes the RDF-source (a file path or Web address). The $\&rdf$-atom interfaces the Raptor RDF library.

The description-logic plug-in. To query description-logic knowledge bases, we developed the description-logic plug-in, which includes four external atoms, allowing for extending a description-logic knowledge base before submitting a query, by means of the atoms' input parameters:

- the $\&dlC$ atom, which queries a concept (specified by an input parameter of the atom) and retrieves its individuals,
- the $\&dlR$ atom, which queries an object property and retrieves its individual pairs,
- the $\&dlDR$ atom, which queries a datatype property and retrieves its pairs, and
- the $\&dlConsistent$ atom, which tests the (possibly extended) description-logic knowledge base for consistency.

The description-logic plug-in can access OWL ontologies, i.e., description-logic knowledge bases in the language $\mathcal{SHOIN}(\mathbf{D})$, utilizing the RACER reasoning engine [6].

The string plug-in. For simple string manipulation routines, we provide the string plug-in. It currently consists of five atoms:

- the $\&concat$ atom, which lets the user specify two constant strings in the input list and returns their concatenation as a single output value,
- the $\&strstr$ atom, which tests two strings for substring inclusion,
- the $\&split$ atom, which splits a string along a given delimiter and retrieves a specific part,
- the $\&cmp$ atom, which lexicographically compares two strings, and
- the $\&sha1sum$ atom, which calculates a SHA1 160-bit checksum for a given string.

The policy plug-in. The policy plug-in was created to satisfy the needs of optimization problems that cannot be tackled using conventional methods such as weak or weight constraints in an intuitive way. In the area of policy specification, answer-set programming is used to generate a search space of valid combinations of credentials, which then need to be ranked based on the specific selection of credentials in each solution. For instance, credentials might have various levels of sensitivity regarding their publication in a business transaction, and the overall goal is to find a set of credentials with minimum overall sensitivity. As soon as this overall value is composed in a more complicated way than just the sum of all single sensitivity values, the $\&policy$-atom provided by the policy plug-in offers a natural solution. It takes a single predicate as input and returns a numerical value, which is computed according to the predicate's extension and a custom function, implemented by the program designer. Using this value in a single weight constraint facilitates the compact formulation of such an optimization task.

The following code fragment illustrates this technique. We assume that the guessing part of the program creates various combinations of ground facts for $credential$.

Each credential has a sensitivity value, all of which are fed into the external atom by the extension of $selected$ for each guessed model. The weak constraint (3) causes the optimization strategy of dlvhex to single out the model that has the least numerical value for $modelWeight$:

$$\begin{aligned} selected(C,V) &\leftarrow credential(C), hasSens(C,V), \\ modelWeight(X) &\leftarrow \&policy[selected](X), \\ &\Leftarrow modelWeight(X)\ [X:1]. \end{aligned} \tag{3}$$

Eventually, it is up to the author of the external atom how to compute this value within the evaluation function of the $\&policy$-atom. We offer a template for this plug-in, where only the actual function for the computation of the cost value needs to be inserted.

On `http://www.kr.tuwien.ac.at/research/dlvhex/`, we provide a Web-interface to evaluate HEX-programs online, along with a more detailed documentation of all available external atoms. Currently, dlvhex and the presented plug-ins are publicly available as source packages. Moreover, we also supply a tool kit for developing custom plug-ins, embedded in the GNU autotools environment, which takes care for the low-level, system-specific build process and lets the plug-in author concentrate his or her efforts on the implementation of the plug-in's actual core functionality.

References

1. H. Boley, S. Tabet, and G. Wagner. Design Rationale for RuleML: A Markup Language for Semantic Web Rules. In *Proc. SWWS 2001*, pages 381–401, 2001.
2. D. Calvanese, G. D. Giacomo, and M. Lenzerini. A Framework for Ontology Integration. In *Proc. SWWS 2001*, pages 303–316, 2001.
3. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *Proc. IJCAI 2005*. Morgan Kaufmann, 2005.
4. W. Faber, N. Leone, and G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In *Proc. JELIA 2004*, pages 200–212, 2004.
5. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
6. V. Haarslev and R. Möller. RACER System Description. In *Proc. IJCAR 2001*, pages 701–705, 2001.
7. G. Ianni, G. Ielpa, A. Pietramala, M. C. Santoro, and F. Calimeri. Enhancing Answer Set Programming with Templates. In *Proc. NMR 2004*, pages 233–239, 2004.
8. M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.
9. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*. To appear.
10. M. Sintek and S. Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *Proc. ISWC 2002*, pages 364–378, 2002.
11. K. Wang, G. Antoniou, R. W. Topor, and A. Sattar. Merging and Aligning Ontologies in dl-Programs. In *Proc. RuleML 2005* pages 160–171, 2005.

G-Hybrid Knowledge Bases

Stijn Heymans[1], Livia Predoiu[1], Cristina Feier[1], Jos de Bruijn[1], and Davy Van Nieuwenborgh[2⋆]

[1] Digital Enterprise Research Institute (DERI)
University of Innsbruck, Austria
{stijn.heymans,livia.predoiu,cristina.feier,jos.debruijn}@deri.org
[2] Dept. of Computer Science
Vrije Universiteit Brussel, VUB
Pleinlaan 2, B1050 Brussels, Belgium
dvnieuwe@vub.ac.be

Abstract. Recently, there has been a lot of interest in the integration of Description Logics and rules on the Semantic Web. We define *g-hybrid knowledge bases* as knowledge bases that consist of a Description Logic knowledge base and a *guarded* logic program, similarly to the $\mathcal{DL}$+*log* knowledge bases from [25]. G-hybrid knowledge bases enable an integration of Description Logics and Logic Programming where, unlike in other approaches, variables in the rules of a guarded program do not need to appear in positive non-DL atoms of the body: DL atoms can act as *guards* as well. Decidability of satisfiability checking of g-hybrid knowledge bases is shown for the particular DL $\mathcal{DLRO}^{-\{\leq\}}$, which is close to, and in some respects more expressive than, OWL DL, by a reduction to guarded programs under an open answer set semantics. Moreover, we show 2-EXPTIME-completeness for satisfiability checking of those $\mathcal{DLRO}^{-\{\leq\}}$ g-hybrid knowledge bases. Finally, we discuss advantages and disadvantages of our approach compared with $\mathcal{DL}$+*log* knowledge bases.

1 Introduction

There has been a lot of attention recently in the integration of Description Logics with rules for the Semantic Web [23, 25, 6, 22, 16]. R-hybrid knowledge bases [23], and the extension $\mathcal{DL}$+*log* [25], is an elegant formalism based on combined models for Description Logic knowledge bases and nonmonotonic logic programs. We propose a variant of r-hybrid knowledge bases, called *g-hybrid knowledge bases*, which do not require standard names or a safeness restriction on rules. We show several computational properties by a reduction to guarded open answer set programming [13].

Open answer set programming (OASP) [14, 13] combines the logic programming and first-order logic paradigms. From the logic programming paradigm it inherits a rule-based presentation and a nonmonotonic semantics by means of negation as failure.

⋆ The first four authors were partially supported by the European Commission under projects Knowledge Web (IST-2004-507482) and DIP (FP6-507483) and by the FIT-IT under the project RW^2 (FIT-IT 809250). The last author was supported by the Flemish Fund for Scientific Research (FWO-Vlaanderen).

In contrast with usual logic programming semantics, see, e.g., the answer set semantics [8], OASP allows for domains consisting of other objects than those present in the logic program at hand. Such open domains are inspired by first-order logic based languages such as Description Logics (DLs) [2] and make OASP a viable candidate for conceptual reasoning. Due to its rule-based presentation and its support for nonmonotonic reasoning and open domains, OASP can be used to reason with both rule-based and conceptual knowledge on the Semantic Web, as illustrated in [14].

The main challenge for OASP is to control undecidability of satisfiability checking, a challenge it shares with DL-based languages. In [13], a decidable class of programs is identified, so-called *guarded programs*, for which decidability of satisfiability checking is obtained by a translation to guarded fixed point logic [10]. In [12], we show the expressiveness of such guarded programs by simulating a DL with n-ary roles and nominals. In particular, we extend the DL $\mathcal{DLR}$ [4] with both *concept nominals* $\{o\}$ and *role nominals* $\{(o_1, \ldots, o_n)\}$, resulting in $\mathcal{DLRO}$. The DL $\mathcal{DLRO}$ with the number restrictions left out yields $\mathcal{DLRO}^{-\{\leq\}}$ and we show in [13] a reduction of satisfiability of concept expressions w.r.t. $\mathcal{DLRO}^{-\{\leq\}}$ knowledge bases to guarded programs.

G-hybrid knowledge bases consist of Description Logic knowledge base and a guarded program. The $\mathcal{DL}$+*log* knowledge bases from [25] are *weakly safe*, i.e., the interaction between the program and the DL knowledge base is limited by imposing that head variables should appear in atoms that cannot be DL atoms (i.e., concepts or roles in the knowledge base). However, in g-hybrid knowledge bases such a restriction does not hold; variables should appear in a *guard* of the rule but this guard can be a DL atom as well. We show decidability of g-hybrid knowledge bases for $\mathcal{DLRO}^{-\{\leq\}}$ DL knowledge bases by a reduction to guarded programs only, as well as provide a 2-EXPTIME complexity characterization of such g-hybrid knowledge bases. $\mathcal{DLRO}^{-\{\leq\}}$ includes a large fragment of $\mathcal{SHOIN}$, the Description Logic underlying OWL DL [15]. Compared with $\mathcal{SHOIN}$, $\mathcal{DLRO}^{-\{\leq\}}$ does not include transitive roles and number restrictions, but does include n-ary roles and complex role expressions.

The remainder of the paper starts with an introduction to open answer set programming and Description Logics in Subsections 2.1 and 2.2. Section 3 defines g-hybrid knowledge bases, translates them to guarded programs when the $\mathcal{DLRO}^{-\{\leq\}}$ DL is considered, and provides a complexity characterization for satisfiability checking of these particular g-hybrid knowledge bases. In Section 4, we discuss the relation of g-hybrid knowledge bases with $\mathcal{DL}$+*log* and point to other related work. We conclude and give directions for further research in Section 5.

2 Preliminary Definitions: Open Answer Set Programming and Description Logics

In this section, we introduce Open Answer Set Programming and the Description Logic $\mathcal{DLRO}^{-\{\leq\}}$.

2.1 Decidable Open Answer Set Programming

We introduce the open answer set semantics from [13], modified as in [12] such that it does not take on a unique name assumption for constants by default. *Constants*, *vari-*

ables, *terms*, and *atoms* are defined as usual. A *literal* is an atom $p(\boldsymbol{t})$ or a *naf-atom* $not\ p(\boldsymbol{t})$.[3] The *positive part* of a set of literals α is $\alpha^+ = \{p(\boldsymbol{t}) \mid p(\boldsymbol{t}) \in \alpha\}$ and the *negative part* of α is $\alpha^- = \{p(\boldsymbol{t}) \mid not\ p(\boldsymbol{t}) \in \alpha\}$. We assume the existence of binary predicates $=$ and $\neq$, where $t = s$ is considered as an atom and $t \neq s$ as $not\ t = s$. E.g., for $\alpha = \{X \neq Y, Y = Z\}$, we have $\alpha^+ = \{Y = Z\}$ and $\alpha^- = \{X = Y\}$. A *regular* atom is an atom that is not an equality atom. For a set A of atoms, $not\ A = \{not\ l \mid l \in A\}$.

A *program* is a countable set of rules $\alpha \leftarrow \beta$, where α and β are finite sets of literals, $|\alpha^+| \leq 1$ (but α^- may be of arbitrary size) , and $\forall t, s \cdot t = s \notin \alpha^+$, i.e., α contains at most one positive atom and this atom cannot be an equality atom.[4] The set α is the *head* of the rule and represents a disjunction of literals, while β is called the *body* and represents a conjunction of literals. If $\alpha = \emptyset$, the rule is called a *constraint*. *Free rules* are rules of the form $q(\boldsymbol{t}) \vee not\ q(\boldsymbol{t}) \leftarrow$ for a tuple $\boldsymbol{t}$ of terms; they enable a choice for the inclusion of atoms. We call a predicate p free if there is a free rule $p(\boldsymbol{t}) \vee not\ p(\boldsymbol{t}) \leftarrow$. Atoms, literals, rules, and programs that do not contain variables are *ground*.

For a literal, rule, or program o, let $cts(o)$ be the constants in o, $vars(o)$ its variables, and $preds(o)$ its predicates. A *pre-interpretation* U for a program P is a pair (D, σ) where D is a non-empty *domain* and $\sigma : cts(P) \rightarrow D$ is a function that maps all constants to elements from D.[5] We call P_U the ground program obtained from P by substituting every variable in P by every possible element from D and every constant c by $\sigma(c)$. E.g., for a rule $r : p(X) \leftarrow f(X, c)$ and $U = (\{x, y\}, \sigma)$ where $\sigma(c) = x$, we have that the grounding w.r.t. U is

$$p(x) \leftarrow f(x, x)$$
$$p(y) \leftarrow f(y, x)$$

Let $\mathcal{B}_P$ be the set of regular atoms that can be defined using the language of the ground program P.

An *interpretation* I of a ground P is any subset of $\mathcal{B}_P$. For a ground regular atom $p(\boldsymbol{t})$, we write $I \models p(\boldsymbol{t})$ if $p(\boldsymbol{t}) \in I$; for an equality atom $p(\boldsymbol{t}) \equiv t = s$, we have $I \models p(\boldsymbol{t})$ if s and t are equal terms. We have $I \models not\ p(\boldsymbol{t})$ if $I \not\models p(\boldsymbol{t})$. For a set of ground literals A, $I \models A$ if $I \models l$ for every $l \in A$. A ground rule $r : \alpha \leftarrow \beta$ is *satisfied* w.r.t. I, denoted $I \models r$, if $I \models l$ for some $l \in \alpha$ whenever $I \models \beta$. A ground constraint $\leftarrow \beta$ is satisfied w.r.t. I if $I \not\models \beta$.

For a ground program P without *not*, an interpretation I of P is a *model* of P if I satisfies every rule in P; it is an *answer set* of P if it is a subset minimal model of P. For ground programs P containing *not*, the *GL-reduct* [8, 20] w.r.t. I is defined as P^I, where P^I contains $\alpha^+ \leftarrow \beta^+$ for $\alpha \leftarrow \beta$ in P, $I \models not\ \beta^-$ and $I \models \alpha^-$. I is an *answer set* of a ground P if I is an answer set of P^I. Note that allowing for negation in the head of rules leads to the loss of the *anti-chain property* which says that no answer set can

[3] We do not allow "classical" negation $\neg$, however, programs with $\neg$ can be reduced to programs without it, see e.g. [21].

[4] The condition $|\alpha^+| \leq 1$ makes the GL-reduct non-disjunctive, ensuring that the *immediate consequence operator* is well-defined, see [13].

[5] In [13], we only use the domain D which is there a non-empty superset of the constants in P. This corresponds to a pre-interpretation (D, σ) where σ is the identity function on D.

be a strict subset of another answer set. In the presence of negation in the head answer sets can be subsets of other answer sets. E.g, a rule $a \vee not\ a \leftarrow$ has the answer sets $\emptyset$ and $\{a\}$. However, we need negation in the head to be able to simulate a first-order behavior for certain predicates, e.g., when simulating Description Logic reasoning.

In the following, a program is assumed to be a finite set of rules; infinite programs only appear as byproducts of grounding a finite program with an infinite pre-interpretation. An *open interpretation* of a program P is a pair (U, M) where U is a pre-interpretation for P and M is an interpretation of P_U. An *open answer set* of P is an open interpretation (U, M) of P with M an answer set of P_U. An n-ary predicate p in P is *satisfiable* if there is an open answer set $((D, \sigma), M)$ of P and a $\boldsymbol{x} \in D^n$ such that $p(\boldsymbol{x}) \in M$. A program P is satisfiable iff it has an open answer set. Note that satisfiability checking of programs can be easily reduced to satisfiability checking of predicates: P is satisfiable iff p is satisfiable w.r.t. $P \cup \{p(\boldsymbol{X}) \vee not\ p(\boldsymbol{X}) \leftarrow\}$, where p is a new predicate not in P and $\boldsymbol{X}$ is a tuple of variables. In the following, when we speak of satisfiability checking we are referring to satisfiability checking of predicates, unless specified otherwise.

Satisfiability checking w.r.t. the open answer set semantics is undecidable in general. In [13], we identify a syntactically restricted fragment of programs, so-called *guarded programs*, for which satisfiability checking is decidable and obtained by a reduction to guarded fixed point logic [10]. The decidability of guarded programs relies on the presence of an atom in each rule that contains all variables of the rule, the *guard* of the rule. Formally, a rule $r : \alpha \leftarrow \beta$ is *guarded* if there is an atom $\gamma_b \in \beta^+$ such that $vars(r) \subseteq vars(\gamma_b)$; we call γ_b a *guard* of r. A program P is a *guarded program (GP)* if every non-free rule in P is guarded. E.g., a rule $a(X, Y) \leftarrow not\ f(X, Y)$ is not guarded, but $a(X, Y) \leftarrow g(X, Y), not\ f(X, Y)$ is guarded with guard $g(X, Y)$. Satisfiability checking of predicates w.r.t. guarded programs under the open answer set semantics is 2-EXPTIME-complete [13] – a result that stems from the corresponding complexity in guarded fixed point logic.

We do not have a unique name assumption, i.e., it might be the case that for two distinct c_1 and c_2, $\sigma(c_1) = \sigma(c_2)$ for a pre-interpretation (D, σ).

2.2 The Description Logic $\mathcal{DLRO}^{-\{\leq\}}$

The DL $\mathcal{DLR}$ [4, 2] is a DL that supports n-ary roles, instead of the usual binary ones. We introduce the extension of $\mathcal{DLR}$ with nominals, called $\mathcal{DLRO}$, as in [12]. The basic building blocks in $\mathcal{DLR}$ are *concept names* A and *relation names* $\mathbf{P}$ where $\mathbf{P}$ denotes arbitrary n-ary relations for $2 \leq n \leq n_{max}$ and n_{max} is a given finite non-negative integer. Role expressions $\mathbf{R}$ and concept expressions C can be formed according to the following syntax rules:

$$\mathbf{R} \rightarrow \top_n \mid \mathbf{P} \mid (\$i/n : C) \mid \neg\mathbf{R} \mid \mathbf{R_1} \sqcap \mathbf{R_2} \mid \{(o_1, \ldots, o_n)\}$$
$$C \rightarrow \top_1 \mid A \mid \neg C \mid C_1 \sqcap C_2 \mid \exists[\$i]\mathbf{R} \mid \leq k[\$i]\mathbf{R} \mid \{o\}$$

where we assume i is between 1 and n in $(\$i/n : C)$, and similarly in $\exists[\$i]\mathbf{R}$ and $\leq k[\$i]\mathbf{R}$ if $\mathbf{R}$ is an n-ary relation. Moreover, we assume that the above constructs are well-typed, e.g., $\mathbf{R}_1 \sqcap \mathbf{R}_2$ is defined only for relations of the same arity. The semantics

of $\mathcal{DLRO}$ is given by interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ is a non-empty set, the *domain*, and $\cdot^{\mathcal{I}}$ is an interpretation function such that $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, $\mathbf{R}^{\mathcal{I}} \subseteq (\Delta^{\mathcal{I}})^n$ for an n-ary relation $\mathbf{R}$, and the following conditions are satisfied ($\mathbf{P}, \mathbf{R}, \mathbf{R}_1$, and $\mathbf{R}_2$ have arity n):

$$\begin{aligned}
\top_n^{\mathcal{I}} &\subseteq (\Delta^{\mathcal{I}})^n \\
\mathbf{P}^{\mathcal{I}} &\subseteq \top_n^{\mathcal{I}} \\
(\neg \mathbf{R})^{\mathcal{I}} &= \top_n^{\mathcal{I}} \backslash \mathbf{R}^{\mathcal{I}} \\
(\mathbf{R}_1 \sqcap \mathbf{R}_2)^{\mathcal{I}} &= \mathbf{R}_1^{\mathcal{I}} \cap \mathbf{R}_2^{\mathcal{I}} \\
(\$i/n : C)^{\mathcal{I}} &= \{(d_1, \ldots, d_n) \in \top_n^{\mathcal{I}} \mid d_i \in C^{\mathcal{I}}\} \\
\\
\top_1^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
A^{\mathcal{I}} &\subseteq \Delta^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}} \\
(C_1 \sqcap C_2)^{\mathcal{I}} &= C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(\exists [\$i] \mathbf{R})^{\mathcal{I}} &= \{d \in \Delta^{\mathcal{I}} \mid \exists (d_1, \ldots, d_n) \in \mathbf{R}^{\mathcal{I}} \cdot d_i = d\} \\
(\leq k [\$i] \mathbf{R})^{\mathcal{I}} &= \{d \in \Delta^{\mathcal{I}} \mid |\{(d_1, \ldots, d_n) \in \mathbf{R}^{\mathcal{I}} \mid d_i = d\}| \leq k\} \\
\{o\}^{\mathcal{I}} &= \{o^{\mathcal{I}}\} \subseteq \Delta^{\mathcal{I}} \\
\{(o_1, \ldots, o_n)\}^{\mathcal{I}} &= \{(o_1^{\mathcal{I}}, \ldots, o_n^{\mathcal{I}})\}
\end{aligned}$$

Note that in $\mathcal{DLRO}$ the negation of role expressions is defined w.r.t. $\top_n^{\mathcal{I}}$ instead of $(\Delta^{\mathcal{I}})^n$. A $\mathcal{DLRO}$ knowledge base consists of terminological axioms and role axioms defining subset relations between concept expressions and role expressions (of the same arity) respectively. A terminological axiom $C_1 \sqsubseteq C_2$ is *satisfied* by $\mathcal{I}$ iff $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$. A role axiom $\mathbf{R}_1 \sqsubseteq \mathbf{R}_2$ is *satisfied* by $\mathcal{I}$ iff $\mathbf{R}_1^{\mathcal{I}} \subseteq \mathbf{R}_2^{\mathcal{I}}$. An interpretation is a *model* of a knowledge base if all axioms are satisfied by the interpretation, in which case we call the knowledge base *satisfiable*. A concept expression C is satisfiable w.r.t. a knowledge base Σ if there is a model $\mathcal{I}$ of Σ such that $C^{\mathcal{I}} \neq \emptyset$.

Note that for every interpretation $\mathcal{I}$, one has

$$(\{(o_1, \ldots, o_n)\})^{\mathcal{I}} = ((\$1/n : \{o_1\}) \sqcap \ldots \sqcap (\$n/n : \{o_n\}))^{\mathcal{I}} ,$$

such that we will restrict ourselves in the remainder of the paper to nominals of the form $\{o\}$ only.

We denote the fragment of $\mathcal{DLRO}$ without the number restriction $\leq k[\$i]\mathbf{R}$ as $\mathcal{DLRO}^{-\{\leq\}}$.

3 G-hybrid Knowledge Bases

G-hybrid knowledge bases are a variant of r-hybrid knowledge bases [23], based on guarded programs, without the standard names assumption. Given a particular Description Logic $\mathcal{DL}$, we define *g-hybrid knowledge bases* as pairs consisting of on the one hand a $\mathcal{DL}$ knowledge base and on the other hand a guarded program (GP).

Definition 1. *Given Description Logic $\mathcal{DL}$, a* g-hybrid knowledge base *is a pair (Σ, P) where Σ is a $\mathcal{DL}$ knowledge base and P is a guarded program.*

Note that in the above definition there are no conditions on the occurrence of predicates, however, by definition, we call the atoms and literals in P that have underlying predicates that correspond to concept names or role names in the DL knowledge base, *DL atoms* and *DL literals* respectively. Variables in rules are not required to appear in positive non-DL atoms as is the case in, e.g., the $\mathcal{DL}{+}log$ knowledge bases in [25], the r-hybrid knowledge bases in [23], or the DL-safe rules in [22]. DL-atoms can appear in the head of rules, thus allowing for a bi-directional flow of information between the DL knowledge base and the program.

Example 1. Consider the $\mathcal{DLRO}^{-\{\leq\}}$ knowledge base Σ where $socialDrinker$ is a concept, $drinks$ is a ternary role such that, intuitively, (x, y, z) is in the interpretation of $drinks$ if a person x drinks with person y some z:

$$socialDrinker \sqsubseteq \exists[\$1](drinks \sqcap (\$3/3 : \{wine\})) \ .$$

The knowledge base indicates that social drinkers drink wine with someone. Consider a GP P that indicates that someone has an increased risk of alcoholism if the person is a social drinker and knows someone from the association of Alcoholics Anonymous (AA). Furthermore, we state that *john* is a social drinker and knows *michael* from AA.

$$\begin{aligned} problematic(X) &\leftarrow socialDrinker(X), knowsFromAA(X, Y) \\ knowsFromAA(john, michael) &\leftarrow \\ socialDrinker(john) &\leftarrow \end{aligned}$$

Together Σ and P form a g-hybrid knowledge base. The literals $socialDrinker(X)$ and $socialDrinker(john)$ are DL atoms where the latter appears in the head of a rule in P. The literal *knowsFromAA(X,Y)* appears only in the program P (and is thus not a DL atom).

We define the semantics of g-hybrid knowledge bases (Σ, P) using interpretations $(U, \mathcal{I}, M)$. Given a DL interpretation $(D, \mathcal{I})$ and a ground program P, define $\Pi(P, \mathcal{I})$ as the *projection* of P with respect to $\mathcal{I}$ obtained as follows: for every rule r in P,

- if there exists a DL literal in the head of the form
  - $A(\boldsymbol{t})$ with $\boldsymbol{t} \in A^{\mathcal{I}}$, or
  - $not\ A(\boldsymbol{t})$ with $\boldsymbol{t} \notin A^{\mathcal{I}}$,

  then delete r,
- if there exists a DL literal in the body of the form
  - $A(\boldsymbol{t})$ with $\boldsymbol{t} \notin A^{\mathcal{I}}$, or
  - $not\ A(\boldsymbol{t})$ with $\boldsymbol{t} \in A^{\mathcal{I}}$, or

  then delete r,
- otherwise, delete all DL literals from r.

Intuitively, the projection of a ground program transforms this grounded program by removing rules and DL literals consistently with $\mathcal{I}$; conceptually this is similar to the GL-reduct which removes the rules and negative literals consistently with an interpretation.

Definition 2. *Let (Σ, P) be a g-hybrid knowledge base. Then $(U, \mathcal{I}, M)$ is an interpretation of (Σ, P) iff*

- *$U = (D, \sigma)$ is a pre-interpretation for P,*
- *$(D, \mathcal{I})$ is an interpretation of Σ,*
- *M is an interpretation of $\Pi(P_U, \mathcal{I})$, and*
- *$b^{\mathcal{I}} = \sigma(b)$ for every constant symbol b appearing both in Σ and in P,*

For $(U = (D, \sigma), \mathcal{I}, M)$ to be a model of a g-hybrid knowledge base, we require that $(D, \mathcal{I})$ should be a model of the Description Logic knowledge base and that M should be an answer set of the projection of the grounding of the program with U.

Definition 3. *An interpretation $(U, \mathcal{I}, M)$ with $U = (D, \sigma)$ is then a* model *of (Σ, P) iff*

1. *$(D, \mathcal{I})$ is a model of Σ, and*
2. *M is an answer set of $\Pi(P_U, \mathcal{I})$.*

For p a concept expression from Σ or a predicate from P, we have that p is satisfiable *w.r.t (Σ, P) if there is a model $(U, \mathcal{I}, M)$ such that $p^{\mathcal{I}} \neq \emptyset$ or $p(\boldsymbol{x}) \in M$ for some $\boldsymbol{x}$ from D, respectively.*

Example 2. Consider the g-hybrid knowledge base in Example 1. Take $U = (D, \sigma)$ with $D = \{john, michael, wine, x\}$ and σ the identity function on the constant symbols in (Σ, P). Furthermore, define $\cdot^{\mathcal{I}}$ as follows:

- $socialDrinker^{\mathcal{I}} = \{john\}$,
- $drinks^{\mathcal{I}} = \{(john, x, wine)\}$,
- $wine^{\mathcal{I}} = wine$.

and $M \equiv \{knowsfromAA(john, michael), problematic(john)\}$. Then $(U, \mathcal{I}, M)$ is a model of this g-hybrid knowledge base. Note that the projection of the program will no longer contain the rule $socialDrinker(john) \leftarrow$.

We can translate the g-hybrid knowledge base from Example 1 to a GP such that the knowledge base is satisfiable iff the logic program is satisfiable. The axiom

$$socialDrinker \sqsubseteq \exists[\$1](drinks \sqcap (\$3/3 : \{wine\})) \ .$$

is translated to a constraint:

$$\leftarrow socialDrinker(X), not\ (\exists[\$1](drinks \sqcap (\$3/3 : \{wine\})))(X)$$

Thus, the concept expressions on either side of the $\sqsubseteq$ symbol in an axiom are associated with a new unary predicate name. For convenience, we denote the predicates like the corresponding concept expressions. The constraint simulates the behavior of the $\mathcal{DLR}\mathcal{O}^{-\{\leq\}}$ axiom. If the left-hand side of the axiom holds and the right-hand side does not hold, we have a contradiction.

It remains to define those newly introduced predicates according to the DL semantics. First, all the concept and role names occurring in the axiom above need to be

defined as free predicates, in order to simulate the first-order semantics of concept and role names in DLs. Intuitively, in DLs a tuple is in the extension of a concept or role or not; this behavior can be captured exactly by free predicates:

$$socialDrinker(X) \vee not\ socialDrinker(X) \leftarrow$$
$$drinks(X, Y, Z) \vee not\ drinks(X, Y, Z) \leftarrow$$

Note that concept names are translated to unary free predicates while n-ary role names are translated to n-ary free predicates.

The definition of the truth symbols $\top_1$ and $\top_3$ that are implicitly present in our $\mathcal{DLRO}^{-\{\leq\}}$ axiom (since the axiom contains a concept name and a ternary role) are defined as free predicates as well. Note that we do not need a predicate for $\top_2$ since the axiom does not contain binary predicates.

$$\top_1(X) \vee not\ \top_1(X) \leftarrow$$
$$\top_3(X, Y, Z) \vee not\ \top_3(X, Y, Z) \leftarrow$$

We ensure that for the ternary $\mathcal{DLRO}^{-\{\leq\}}$ role $drinks$, $drinks^{\mathcal{I}} \subseteq \top_3^{\mathcal{I}}$ holds by adding the constraint:

$$\leftarrow drinks(X, Y, Z), not\ \top_3(X, Y, Z)$$

To ensure that $\top_1^{\mathcal{I}} = \Delta^{\mathcal{I}}$, we add the constraint:

$$\leftarrow not\ \top_1(X)$$

For rules containing only one variable, we can always assume that $X = X$ is in the body and acts as the guard of the rule such that the latter rule is a guarded rule when regarded as the equivalent rule $\leftarrow not\ \top_1(X), X = X$. Note that we can allow for such an equality guard without affecting decidability as decidability for guarded programs was shown in [13] by a translation to guarded fixed point logic where one allows for guards $X = X$ as well [9].

We define the nominal $\{wine\}$ by the rule

$$\{wine\}(wine) \leftarrow$$

Intuitively, since this rule will be the only rule with the predicate $\{wine\}$ in the head, every open answer set of the translated program will only contain $\{wine\}(x)$ with $\sigma(wine) = x$ if and only if the corresponding interpretation $\{wine\}^{\mathcal{I}} = \{x\}$ for $wine^{\mathcal{I}} = x$.

The $\mathcal{DLRO}^{-\{\leq\}}$ role expression $(\$3/3 : \{wine\})$ indicates the ternary tuples for which the third argument belongs to the extension of $wine$, which translates to the following rule:

$$(\$3/3 : \{wine\})(X, Y, Z) \leftarrow \top_3(X, Y, Z), \{wine\}(Z)$$

Note that the above rule is guarded by the $\top_3$ literal.

Finally, the concept expression $(drinks \sqcap (\$3/3 : \{wine\}))$ can be represented by the following rule:

$$(drinks \sqcap (\$3/3 : \{wine\}))(X, Y, Z) \leftarrow drinks(X, Y, Z), \\ (\$3/3 : \{wine\})(X, Y, Z)$$

The translation thus translates the DL constructor $\sqcap$ as conjunction in the body of a rule.

The $\mathcal{DLR}\mathcal{O}^{-\{\leq\}}$ role $\exists[\$1](drinks \sqcap (\$3/3 : \{wine\}))$ can be represented by the following rule:

$$(\exists[\$1](drinks \sqcap (\$3/3 : \{wine\})))(X) \leftarrow (drinks \sqcap (\$3/3 : \{wine\}))(X, Y, Z)$$

Indeed, the elements that belong to the extension of $\exists[\$1](drinks \sqcap (\$3/3 : \{wine\}))$ are exactly those that are connected with the role $(\$3/3 : \{wine\})$ as specified in the rule.

This concludes the translation of the DL knowledge base part of the g-hybrid knowledge base in Example 1. The program part can be considered as is, since, by definition of g-hybrid knowledge bases, this is already a GP.

We define the formal translation from g-hybrid satisfiability checking to satisfiability checking w.r.t. programs using the notion of *closure*. Define the *closure* $clos(\Sigma)$ of a $\mathcal{DLR}\mathcal{O}^{-\{\leq\}}$ knowledge base Σ as the smallest set satisfying the following conditions:

- $\top_1 \in clos(\Sigma)$,
- for each $C \sqsubseteq D$ an axiom in Σ (role or terminological), $\{C, D\} \subseteq clos(\Sigma)$,
- for every D in $clos(\Sigma)$, $clos(\Sigma)$ should contain every subformula that is a concept expression or a role expression,
- if $clos(\Sigma)$ contains n-ary relation names, it must contain $\top_n$.

Formally, we define $\Phi(\Sigma)$ for a $\mathcal{DLR}\mathcal{O}^{-\{\leq\}}$ knowledge base Σ to be the following program:

- For each terminological axiom $C \sqsubseteq D \in \Sigma$, add the constraint

$$\leftarrow C(X), not\ D(X) \tag{1}$$

- For each role axiom $\mathbf{R} \sqsubseteq \mathbf{S} \in \Sigma$ where $\mathbf{R}$ and $\mathbf{S}$ are n-ary, add the constraint

$$\leftarrow \mathbf{R}(X_1, \ldots, X_n), not\ \mathbf{S}(X_1, \ldots, X_n) \tag{2}$$

- For each $\top_n \in clos(\Sigma)$, add the free rule

$$\top_n(X_1, \ldots, X_n) \vee not\ \top_n(X_1, \ldots, X_n) \leftarrow \tag{3}$$

  Furthermore, for each n-ary relation name $\mathbf{P} \in clos(\Sigma)$, we add the constraint

$$\leftarrow \mathbf{P}(X_1, \ldots, X_n), not\ \top_n(X_1, \ldots, X_n) \tag{4}$$

  Intuitively, the latter rule ensures that $\mathbf{P}^{\mathcal{I}} \subseteq \top_n^{\mathcal{I}}$. We add a constraint

$$\leftarrow not\ \top_1(X) \tag{5}$$

  which enforces that for every element x in the pre-interpretation, $\top_1(x)$ is true in the open answer set. The latter rule ensures that $\top_1^{\mathcal{I}} = D$ for the corresponding interpretation. The rule is implicitly guarded with $X = X$.

– Next, we distinguish between the types of concept and role expressions that appear in $clos(\Sigma)$. For $D \in clos(\Sigma)$:
  - if D is a concept nominal $\{o\}$, add

$$D(o) \leftarrow \tag{6}$$

    This will ensure that $\{o\}(x)$ holds in an open answer set iff $x = \sigma(o) = o^{\mathcal{I}}$ for an interpretation of (Σ, P).
  - if D is a concept name, add

$$D(X) \vee not\ D(X) \leftarrow \tag{7}$$

  - if $\mathbf{D}$ is an n-ary relation name, add

$$\mathbf{D}(X_1, \ldots, X_n) \vee not\ \mathbf{D}(X_1, \ldots, X_n) \leftarrow \tag{8}$$

  - if $D = \neg E$ for a concept expression E, add

$$D(X) \leftarrow not\ E(X) \tag{9}$$

    Note that we can again assume that such a rule is guarded by $X = X$.
  - if $D = \neg\mathbf{R}$ for an n-ary role expression $\mathbf{R}$, add

$$D(X_1, \ldots, X_n) \leftarrow \top_n(X_1, \ldots, X_n), not\ \mathbf{R}(X_1, \ldots, X_n) \tag{10}$$

    Note that if negation was defined w.r.t. to D^n instead of $\top_n^{\mathcal{I}}$, we would not be able to write the above as a guarded rule.
  - if $D = E \sqcap F$ for concept expressions E and F, add

$$D(X) \leftarrow E(X), F(X) \tag{11}$$

  - if $D = \mathbf{E} \sqcap \mathbf{F}$ for n-ary role expressions $\mathbf{E}$ and $\mathbf{F}$, add

$$D(X_1, \ldots, X_n) \leftarrow \mathbf{E}(X_1, \ldots, X_n), \mathbf{F}(X_1, \ldots, X_n) \tag{12}$$

  - if $D = (\$i/n : C)$, add

$$D(X_1, \ldots, X_i, \ldots, X_n) \leftarrow \top_n(X_1, \ldots, X_i, \ldots, X_n), C(X_i) \tag{13}$$

  - if $D = \exists[\$i]\mathbf{R}$, add

$$D(X) \leftarrow \mathbf{R}(X_1, \ldots, X_{i-1}, X, X_{i+1}, \ldots, X_n) \tag{14}$$

We now show that this translation preserves satisfiability.

Theorem 1. *Let (Σ, P) be a g-hybrid knowledge base where Σ is a $\mathcal{DLRO}^{-\{\leq\}}$ knowledge base. Then, a predicate or concept expression p is satisfiable w.r.t. (Σ, P) iff p is satisfiable w.r.t. $\Phi(\Sigma) \cup P$.*

Proof. ($\Rightarrow$) Assume p is satisfiable w.r.t. (Σ, P), i.e., there exists a model $(U, \mathcal{I}, M)$ of (Σ, P) where U is the pre-interpretation (D, σ) that gives p a non-empty extension. Construct then the open interpretation (V, N) of (Σ, P) such that $V = (D, \sigma')$ with $\sigma' : cts(\Phi(\Sigma) \cup P) \rightarrow D$ defined such that $\sigma'(x) = \sigma(x)$ for a constant symbol x from P and $\sigma'(x) = x^{\mathcal{I}}$ for a constant symbol from Σ. Note that σ' is well-defined since for constant symbols x that are in both Σ and P, we have that $\sigma(x) = x^{\mathcal{I}}$. The set N is defined as follows:

$$N \equiv M \cup \{C(x) \mid x \in C^{\mathcal{I}}, C \in clos(\Sigma)\} \\ \cup \{\mathbf{R}(x_1, \ldots, x_n) \mid (x_1, \ldots, x_n) \in \mathbf{R}^{\mathcal{I}}, R \in clos(\Sigma)\}$$

with C and $\mathbf{R}$ concept expressions and role expressions respectively.

It is easy to verify that (V, N) is an open answer set of $\Phi(\Sigma) \cup P$ that satisfies p.

($\Leftarrow$) Assume (V, N) is an open answer set of $\Phi(\Sigma) \cup P$ with $V = (D, \sigma')$ such that p is satisfied. Define a tuple $(U, \mathcal{I}, N)$, with

- $U \equiv (D, \sigma)$ where $\sigma : cts(P) \rightarrow D$ with $\sigma(x) \equiv \sigma'(x)$ (note that this is possible since $cts(P) \subseteq cts(\Phi(\Sigma) \cup P)$). U is then a pre-interpretation for P.
- An interpretation function $\mathcal{I}$ defined such that $A^{\mathcal{I}} \equiv \{x \mid A(x) \in N\}$ for concept names A, $\mathbf{R}^{\mathcal{I}} \equiv \{(x_1, \ldots, x_n) \mid \mathbf{R}(x_1, \ldots, x_n) \in N\}$ for n-ary role names $\mathbf{R}$ and $o^{\mathcal{I}} = \sigma'(o)$, for o a constant symbol in Σ (note that σ' is indeed defined on o). $(D, \mathcal{I})$ is then an interpretation of Σ.
- $M \equiv N \setminus \{p(\boldsymbol{x}) \mid p \in clos(\Sigma)\}$, such that M is an interpretation of $\Pi(P_U, \mathcal{I})$.

Moreover, for every constant symbol b appearing both in Σ and in P, $b^{\mathcal{I}} = \sigma(b)$, making $(U, \mathcal{I}, M)$ an interpretation of (Σ, P).

It is easy to verify that $(U, \mathcal{I}, M)$ is a model of (Σ, P) that satisfies p. □

Theorem 2. *Let (Σ, P) be a g-hybrid knowledge base where Σ is a $\mathcal{DLR}\mathcal{O}^{-\{\leq\}}$ knowledge base. Then, $\Phi(\Sigma) \cup P$ is a GP, with a size that is polynomial in the size of (Σ, P).*

Proof. Observing the rules that originate from Σ, it is clear that they are guarded. Furthermore, the program P itself is a GP such that $\Phi(\Sigma) \cup P$ is as well.

The size of $clos(\Sigma)$ is of the order $n \log n$ where n is the size of Σ. Indeed, intuitively, given that the size of an expression is n, we have that the size of the set of its subexpressions is at most the size of a tree with depth $\log n$ where the size of the subexpressions at a certain level of the tree is at most n. The size of the GP $\Phi(\Sigma)$ is polynomial in the size of $clos(\Sigma)$. However, note that we assume here that the size of Σ increases such that the n in an added n-ary role expression is polynomial in the size of the maximal arity of role expressions in Σ. If we were to add a relation name $\mathbf{R}$ with arity 2^n, where n is the maximal arity of relation names in C and Σ, the size of Σ would increase linearly, but the size of $\Phi(\Sigma) \cup P$ would increase exponentially: one needs to add, e.g., rules

$$\top_{2^n}(X_1, \ldots, X_{2^n}) \vee not\ \top_{2^n}(X_1, \ldots, X_{2^n}) \leftarrow \ ,$$

which introduce an exponential number of arguments while the size of the role $\mathbf{R}$ does not depend on its arity. □

Note that in g-hybrid knowledge bases, we consider the fragment $\mathcal{DLRO}^{-\{\leq\}}$ of $\mathcal{DLRO}$ without the expressions $\leq k[\$i]\mathbf{R}$ since such expressions cannot be simulated with guarded programs. E.g., consider the concept expression $\leq 1[\$1]R$ where R is a binary role. One can simulate the $\leq$ by negation as failure:

$$\leq 1[\$1]R(X) \leftarrow not\ q(X)$$

for some new q with q defined such that there are at least 2 different R-successors:

$$q(X) \leftarrow R(X, Y_1), R(X, Y_2), Y_1 \neq Y_2$$

However, the latter rule is not a guarded rule – there is no atom that contains X, Y_1, and Y_2. So, in general, expressing number restrictions such as $\leq k[\$i]\mathbf{R}$ is out of reach for GPs. From Theorems 1 and 2 follows:

Corollary 1. *Satisfiability checking w.r.t. g-hybrid knowledge bases where the DL part is a $\mathcal{DLRO}^{-\{\leq\}}$ knowledge base can be polynomially reduced to satisfiability checking w.r.t. GPs.*

Since satisfiability checking w.r.t. GPs is 2-EXPTIME-complete [13], we have the same 2-EXPTIME characterization for g-hybrid knowledge bases. We first make explicit a corollary of Theorem 1.

Corollary 2. *Let P be a GP. Then, p is satisfiable w.r.t. P iff p is satisfiable w.r.t. $(\emptyset, P)$.*

Theorem 3. *Satisfiability checking w.r.t. g-hybrid knowledge bases where the DL part is a $\mathcal{DLRO}^{-\{\leq\}}$ knowledge base is 2-EXPTIME-complete.*

Proof. Membership in 2-EXPTIME follows from Corollary 1. Hardness follows from 2-EXPTIME-hardness of satisfiability checking w.r.t. GPs and the reduction to satisfiability checking in Corollary 2. □

4 Relation with $\mathcal{DL}$+*log* and other Related Work

In [25], so-called $\mathcal{DL}$+*log* knowledge bases combine a Description Logic knowledge base with a *weakly-safe* disjunctive logic program. Formally, for a particular Description Logic $\mathcal{DL}$, a $\mathcal{DL}$+*log knowledge base* is a pair (Σ, P) where Σ is a $\mathcal{DL}$ knowledge base consisting of a *TBox* (a set of terminological axioms) and an *ABox* (a set of *assertional axioms*), and P contains rules $\alpha \leftarrow \beta$ such that for every rule $r : \alpha \leftarrow \beta \in P$:

- $\alpha^- = \emptyset$,
- β^- does not contain DL atoms (call this *DL-positiveness*),
- each variable in r occurs in an atom from β^+ (*Datalog safeness*), and
- each head variable in r occurs in a non-DL atom from β^+ (*weak safeness*).

The semantics for $\mathcal{DL}$+*log* is the same as it is for g-hybrid knowledge bases[6], with some exceptions:

[6] Strictly speaking, we did not define answer sets of disjunctive programs, however, the definitions of Subsection 2.1 can serve for disjunctive programs without modification. Also, we did not consider ABoxes in our definition of DLs in Subsection 2.2. However, the extension of the semantics of DL knowledge bases with ABoxes is straightforward.

– We do not have a *standard name assumption* such as [25] has, which basically assumes every interpretation is over the same infinitely countable number of constants. Neither do we have the implied *unique name assumption*, making the semantics for g-hybrid knowledge bases more in line with current Semantic Web standards such as OWL [3] where neither the standard names assumption nor the unique names assumption holds. Note that Rosati also presented a version of hybrid knowledge bases which does not adhere to the unique name assumption in an earlier work [24]. However, the grounding of the program part is with the constant symbols explictly appearing in the program or DL part only, which yields a less tight integration of the program and the DL part than in [25] or in g-hybrid knowledge bases.
– Furthermore, we defined an interpretation as a triple $(U, \mathcal{I}, M)$ instead of a pair $(U, \mathcal{I}')$ where $\mathcal{I}' = \mathcal{I} \cup M$; this is, however, equivalent to [25].

We balance the key differences of the two approaches:

– In [25] the head of a rule is of the form $p_1(\boldsymbol{X}_1) \vee \ldots \vee p_n(\boldsymbol{X}_n)$ with n possibly 0, i.e., the requirement $|\alpha^+| \leq 1$ does not hold as it does for our programs. On the other hand, this implies that $|\alpha^-| = 0$ in [25], while there is no such restriction in our case.
– Instead of Datalog safeness we have *guardedness*, i.e., while with Datalog safeness every variable in the rule should appear in some positive atom of the body of the rule, guardedness requires that there is a positive atom that contains every variable in the rule. E.g., $a(X) \leftarrow b(X), c(Y)$ is Datalog safe since X appears in $b(X)$ and Y appears in $c(Y)$ but it is not guarded since there is no atom that contains both X and Y in its arguments. Note that we could easily extend the approach taken in this paper to *loosely guarded programs* which require that every two variables in the rule should appear together in a positive atom, however, this would still be less expressive than Datalog safeness.
– We do not have the requirement for weak safeness, i.e., head variables do not need to appear positively in a non-DL atom. The guardedness may be provided by a DL atom.

  Example 3. Example 1 contains the rule

  $$problematic(X) \leftarrow socialDrinker(X), knowsFromAA(X, Y)$$

  This allows to deduce that X might be a problem case even if X knows someone from the AA but is not drinking with that person, indeed, as illustrated by the example model in Example 1, $john$ is drinking wine with some anonymous x and knows $michael$ from the AA. More correct would be the rule

  $$problematic(X, Z) \leftarrow drinks(X, Y, Z), knowsFromAA(X, Y)$$

  where we explicitly say that X and Y in the $drink$ and $knowsFromAA$ relation should be the same and we extend the $problematic$ predicate with the kind of drink that X has a problem with. Then, the head variable Z is guarded by the DL atom $drinks$ and the rule is thus not weakly-safe but it is guarded nonetheless. Thus, the resulting knowledge base is not a $\mathcal{DL}$+*log* knowledge base but is a g-hybrid knowledge base.

– We do not have the requirement for DL-positiveness, i.e., DL atoms may appear negated in the body of rules (and also in the heads of rules). However, one could allow this in $\mathcal{DL}{+}log$ knowledge bases as well, since $not\ A(\boldsymbol{X})$ in the body of the rule has the same effect as $A(\boldsymbol{X})$ in the head, where the latter is allowed in [25]. Vice versa, we can also loosen our restriction on the occurrence of positive atoms in the head (which allows at most one positive atom in the head), to allow for an arbitrary number of positive DL atoms in the head (but still keep the number of positive non-DL atoms limited to one). E.g., a rule $p(X) \vee A(X) \leftarrow \beta$, where $A(X)$ is a DL atom, is not a valid rule in the programs we considered since the head contains more than one positive atom. However, we can always rewrite such a rule as the rule $p(X) \leftarrow \beta, not\ A(X)$, which contains at most one positive atom in the head.
  Arguably, DL atoms should not be allowed to occur negatively, because DL predicates are interpreted classically and thus the negation in front of the DL atom is not nonmonotonic. However, Datalog predicates which depend on DL predicates are also (partially) interpreted classically.
– We do not take into account ABoxes in the DL knowledge base like [25] does. However, the DL we consider includes nominals such that one can simulate the ABox using terminological axioms. Moreover, even if the DL does not include nominals the ABox can be written as ground facts in a program and ground facts are always guarded.
– Decidability for satisfiability checking[7] of $\mathcal{DL}{+}log$ knowledge bases in [25] is guaranteed if decidability of the conjunctive query containment problem is guaranteed for the DL at hand. However, we relied for showing decidability on a translation of DLs to guarded programs, and, as explained in the previous section, e.g., DLs with number restrictions cannot be translated to them.

[18] and [26] simulate reasoning in DLs with a LP formalism by using an intermediate translation to first-order clauses. In [18], $\mathcal{SHIQ}$ knowledge bases are reduced to first-order formulas, on which basic superposition calculus is then applied.

[26] translates $\mathcal{ALCQI}$ concept expressions to first-order formulas, grounds them with a finite number of constants, and transforms the result to a logic program. One can use a finite number of constants by the finite model property of $\mathcal{ALCQI}$; in the presence of terminological axioms this is no longer possible since the finite model property is lost.

In [19], the DL $\mathcal{ALCNR}$ ($\mathcal{R}$ stands for role intersection) is extended with Horn clauses $q(\boldsymbol{Y}) \leftarrow p_1(\boldsymbol{X}_1), \ldots, p_n(\boldsymbol{X}_n)$ where the variables in $\boldsymbol{Y}$ must appear in $\boldsymbol{X}_1 \cup \ldots \cup \boldsymbol{X}_n$; $p_1, \ldots, p_n$ are either concept names, role names, or ordinary predicates not appearing in the DL part, and q is an ordinary predicate. There is no safeness in the sense that every variable must appear in a non-DL atom (i.e., with an ordinary predicate), as it is in, e.g., [22]. The semantics is as in [22]: extended interpretations that satisfy both the DL and clauses part (as FOL formulas). Query answering is undecidable if recursive Horn clauses are allowed, but decidability can be regained by restricting the DL part or

[7] [25] considers satisfiability checking of knowledge bases instead of predicate satisfiability checking as we do, however, the former can easily be reduced to the latter.

by enforcing that the clauses are role safe (each variable in a role atom $R(X, Y)$ for a role R must appear in a non-DL atom). Note that the latter restriction is less strict than the DL-safeness of [22], where also variables in concept atoms $A(X)$ need to appear in non-DL atoms. On the other hand, [22] allows for the more expressive DL $\mathcal{SHOIN}(\mathbf{D})$, and the head predicates may be DL atoms as well. In relation with our work: we needed the guardedness and not just role safeness as in [19].

An $\mathcal{AL}$-log [5] system consists of two subsystems: an $\mathcal{ALC}$ knowledge base and a set of Horn clauses of the above form, where variables in the head must appear in the body, only concept names besides ordinary predicates are allowed in the body (thus no role names), and there is a safeness condition as in [22] saying that every variable appears in a non-DL atom.

In [6, 7] *Description Logic programs* are introduced; atoms in the program component may be *dl-atoms* such that one can query the knowledge in the DL part and each query can also provide the DL with information that the rule part deduced, yielding a bi-directional flow of information.

Finally, SWRL [17] is a *Semantic Web Rule Language* and extends the syntax and semantics of OWL DL (i.e., $\mathcal{SHOIN}(\mathbf{D})$) with unary/binary Datalog RuleML [1], i.e., Horn-like rules. This extension is undecidable [16].

5 Conclusions and Directions for Further Research

We defined g-hybrid knowledge bases which combine Description Logic (DL) knowledge bases with guarded programs. In particular, we combined knowledge bases of the DL $\mathcal{DLRO}^{-\{\leq\}}$, which is close to OWL DL, with guarded programs and showed decidability of this framework by a reduction to guarded programs under the open answer set semantics [13]. We discussed the relation with $\mathcal{DL}$+*log* knowledge bases: g-hybrid knowledge bases overcome some of the limitations of $\mathcal{DL}$+*log*, such as the unique name assumption, the requirement for DL-positiveness, Datalog safeness, and weak DL-safeness, but introduces the requirement of guardedness. At present, a significant disadvantage of our approach is the lack of support for DLs with number restrictions which is inherent to the use of guarded programs as our decidability vehicle. A solution for this would be to consider other types of programs, such as *conceptual logic programs* [11]. This would allow for the definition of an hybrid knowledge base (Σ, P) where Σ is a $\mathcal{SHIQ}$ knowledge base and P is a conceptual logic program since $\mathcal{SHIQ}$ knowledge bases can be translated to conceptual logic programs.

At present, there is no implemented system for open answer set programming available; this is part of future research.

References

1. The Rule Markup Initiative. http://www.ruleml.org.
2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
3. S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference. `http://www.w3.org/TR/owl-ref/`, 2004.

4. D. Calvanese, G. De Giacomo, and M. Lenzerini. Conjunctive Query Containment in Description Logics with n-ary Relations. In *Proc. of the 1997 Description Logic Workshop (DL'97)*, pages 5–9, 1997.
5. F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. AL-log: Integrating Datalog and Description Logics. *J. of Intell. and Cooperative Information Systems*, 10:227–252, 1998.
6. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with DLs for the Semantic Web. In *Proc. of KR 2004*, pages 141–151, 2004.
7. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Well-Founded Semantics for Description Logic Programs in the Semantic Web. In *Proc. of RuleML 2004*, number 3323 in LNCS, pages 81–97. Springer, 2004.
8. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080, Cambridge, Massachusetts, 1988. MIT Press.
9. E. Grädel, C. Hirsch, and M. Otto. Back and Forth Between Guarded and Modal Logics. *ACM Transactions on Computational Logic*, 3:418–463, 2002.
10. E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *Proc. of LICS '99*, pages 45–54. IEEE Computer Society, 1999.
11. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Conceptual logic programs. *Annals of Mathematics and Artificial Intelligence (Special Issue on Answer Set Programming)*, 2006.
12. S. Heymans, D. Van Nieuwenborgh, D. Fensel, and D. Vermeir. Reasoning with the Description Logic $\mathcal{DLRO}^{-\{\leq\}}$ using Bound Guarded Programs. In *Proc. of Reasoning on the Web workshop (RoW 2006)*, Edinburgh, UK, 2006.
13. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Guarded Open Answer Set Programming. In *LPNMR 2005*, number 3662 in LNAI, pages 92–104. Springer, 2005.
14. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Nonmonotonic Ontological and Rule-Based Reasoning with Extended Conceptual Logic Programs. In *Proc. of ESWC 2005*, number 3532 in LNCS, pages 392–407. Springer, 2005.
15. I. Horrocks and P. Patel-Schneider. Reducing OWL Entailment to Description Logic Satisfiability. *J. of Web Semantics*, 2004. To Appear.
16. I. Horrocks and P. F. Patel-Schneider. A Proposal for an OWL Rules Language. In *Proc. of WWW 2004*. ACM, 2004.
17. I. Horrocks, P. F. Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule language Combining OWL and RuleML, May 2004.
18. U. Hustadt, B. Motik, and U. Sattler. Reducing $\mathcal{SHIQ}^{-}$ Description Logic to Disjunctive Datalog Programs. FZI-Report 1-8-11/03, Forschungszentrum Informatik (FZI), 2003.
19. A. Y. Levy and M. Rousset. CARIN: A Representation Language Combining Horn Rules and Description Logics. In *Proc. of ECAI'96*, pages 323–327, 1996.
20. V. Lifschitz. Answer Set Programming and Plan Generation. *AI*, 138(1-2):39–54, 2002.
21. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
22. B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. In *Proc. of ISWC 2004*, number 3298 in LNCS, pages 549–563. Springer, 2004.
23. R. Rosati. On the decidability and complexity of integrating ontologies and rules. *Web Semantics*, 3(1):41–60, 2005.
24. R. Rosati. Semantic and computational advantages of the safe integration of ontologies and rules. In *Proc. of the Third International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR 2005)*, volume 3703 of *LNCS*, pages 50–64. Springer, 2005.
25. R. Rosati. DL+log: Tight integration of description logics and disjunctive datalog. In *Proc. of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 68–78. AAAI Press, 2006.
26. T. Swift. Deduction in Ontologies via Answer Set Programming. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *LNCS*, pages 275–288. Springer, 2004.

Intelligent Agents that Reason about Web Services: a Logic Programming Approach*

Viviana Mascardi[1] and Giovanni Casella[1,2]

[1]DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy
mascardi@disi.unige.it
[2]DMI, Università di Salerno, Via Ponte don Melillo, 84084, Fisciano (SA), Italy
casella@disi.unige.it

Abstract. The paper proposes to factor three leading edge technologies, namely Web Services, Intelligent Agents, and Computational Logic, for implementing logic-based agents that reason about interaction protocols specified using standard languages for Web Services.
A working multiagent system prototype, where agents implemented in Prolog reason about protocols expressed in WS-BPEL, has been developed.
Keywords. Intelligent Agent, Web Service, Agent-Interaction Protocol, Prolog

1 Introduction

"*Agent-based systems are one of the most vibrant and important areas of research and development to have emerged in information technology in the 1990s. Put at its simplest, an agent is a computer system that is capable of flexible autonomous action in dynamic, unpredictable, typically multiagent domains. [...] Agents provide software designers and developers with a way of structuring an application around autonomous, communicative components, and lead to the construction of software tools and infrastructure to support the design metaphor. In this sense, they offer a new and often more appropriate route to the development of complex computational systems, especially in open and dynamic environments.* "

The above quotation, taken from [23], provides the basic definition of an agent and suggests the potential impact of adopting the agent technology for correctly engineering distributed applications working in heterogeneous, dynamic, unpredictable and open environments. A well-suited example of such an environment is the World Wide Web.

The agent technology shares many common features with another technology, that of Web Services (WSs, [21]), strongly related to the Web. WSs are software applications written in various programming languages and running on various platforms, that can both expose themselves as WSs, and use other WSs. Thus, WSs are heterogeneous, distributed, and operate in an open and dynamic environment as the Web is. Actually,

* Partially supported by the Italian project MIUR PRIN 2005 "Specification and verification of agent interaction protocols".

the recent literature in the agents' field devotes much space to exploring the relationships between agents and WSs. The most well settled opinions are that either WSs provide the infrastructure, and agents provide the coordination framework [29,10,9], or that WS and agent technologies are related by the common goal of providing tools, languages, and methods necessary for engineering systems that behave in a correct way, for example w.r.t. a given interaction protocol [6].

When we consider the second part of the quotation that opens our paper - viewing agents as a design metaphor, the relationship between agents and another well-established technology, Computational Logic (CL), suggests itself. As pointed out in [12], in the Agent-Oriented Software Engineering (AOSE) field, formal methods are used in the specification of systems, for directly programming systems, and in the verification of systems. CL can be very effective for fitting all three roles above. In fact, if an agent is specified by means of a logic-based program, a working prototype of the given specification is immediately available and can be used for early testing and debugging the specification. The distinction between specifying and directly programming an agent is thus blurred. Moreover, the model checking approach to verification can be adopted to show that the agent implementation is correct with respect to its original specification. The fervid activity in this area is demonstrated by the success of many workshops, such as CLIMA[1] and DALT[2]. Various surveys and monographic collections on this topic are also available, such as [28,15,24].

Finally, CL has been used for reasoning about interaction protocols for a long time [7,8,4,2]. Since WSs define interaction protocols, it is a very natural step to adopt CL to represent [26], compose [27,25], and select [5] them.

In this paper, we factor the three technologies of WSs, agents, and CL, for implementing logic-based intelligent agents that reason about interaction protocols specified using standard languages for WSs. In particular, our agents are implemented in Prolog, are executed within the JADE agent platform, and reason about protocols expressed in WS-BPEL. The paper is organised as follows: Section 2 describes how agent interaction protocols can be expressed in WS-BPEL. Section 3 overviews our WS-aware agents, whereas Sections 4, 5, 6, and 7 provide details for the four phases of the WS-aware agent's life, namely the translation from WS-BPEL to a Prolog representation, the generation of a Prolog program that complies with the WS-BPEL interaction protocol, the activity of reasoning about the protocol, and the actual participation to the protocol, respectively. Section 8 concludes.

2 Representing Agent Interaction Protocols in WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL, [1]) is a notation, layered on top of WSDL [17], for specifying business process behaviours (business protocols) based on WSs. Using WS-BPEL it is possible to specify both executable processes, that describe the actual behaviour of a participant in a business interaction and can be executed by an engine, and business protocols, that describe the mutually visi-

[1] http://centria.di.fct.unl.pt/~clima/

[2] http://staff.science.uva.nl/~ulle/DALT-2006/home.html

ble message exchange of each of the parties involved in the protocol, without revealing their internal behaviour. For the purpose of this work, only business protocols are used.

Besides providing language types for defining the conversational relationship between two services (`partnerLinkType`), and activities for implementing message exchange (`invoke` and `receive`), WS-BPEL also offers activities for structuring the protocol execution flow, such as `sequence`, `switch`, `if`, and `while`.

The suitability of WS-BPEL for representing agent interaction protocols (AIPs) is demonstrated by the close relationships between the constructs offered by WS-BPEL, and those offered by one of the most popular notations for AIPs, AUML [18], summarised in Table 1. To make an example, the AUML protocol depicted in Figure 1

	AUML	**WS-BPEL**
Roles	`ag-name/ ag-role: ag-class` box on lifelines	`myRole` and `partnerRole` tags in Partner Links
Message	Labelled arrows between lifelines	`invoke` and `receive`
Content	Speech-act based	Unspecified
Sequence	`Weak Sequencing`	`Sequence`
Condition	`Alternative`	`Switch`
Option	`Option`	`If`
Cycle	`Loop`	`While`

Table 1. Correspondence between AUML and WS-BPEL concepts

corresponds to the WSDL and WS-BPEL documents partly shown below.

WS-BPEL specification

```
1:   <process xmlns="http://schemas.xmlsoap.org/..." ....>
2:    <partnerLinks>
3:     <partnerLink name="publisherPL" partnerLinkType="lns:SellerBuyer"
                    myRole="seller" partnerRole="buyer"/>
4:     <partnerLink name="readerPL" partnerLinkType="lns:BuyerSeller"
                    myRole="buyer" partnerRole="seller"/>
5:    </partnerLinks>
6:    <variables>
7:     <variable name="continue_1" element="lns:continue_1_type"/>
8:     <variable name="choose_2" element="lns:choose_2_type"/>
....
13:   </variables>
14:   <copy><from opaque="yes"/><to>$continue_1.value</to></copy>
15:   <while condition="$continue_1.value=true">
16:    <sequence>
17:     <receive partnerLink="publisherPL" portType="lns:publisherPT"
                    operation="RCV_Mess_1" createIstance="yes"/>
18:     <copy><from opaque="yes"/><to>$choose2.value</to></copy>
19:     <switch>
20:      <case condition="$choose2.value=1">
....
n-10:       <if condition="$condition_6.value=true"/><then>
n-9:        <invoke partnerLink="publisherPL" portType="lns:publisherPT"
                    operation="SND_Mess_15"/>
n-8:        <receive partnerLink="readerPL" portType="lns:readerPT"
                    operation="RCV_Mess_15"/>
n-7:       </then> </if>
n-6:      </sequence>
```

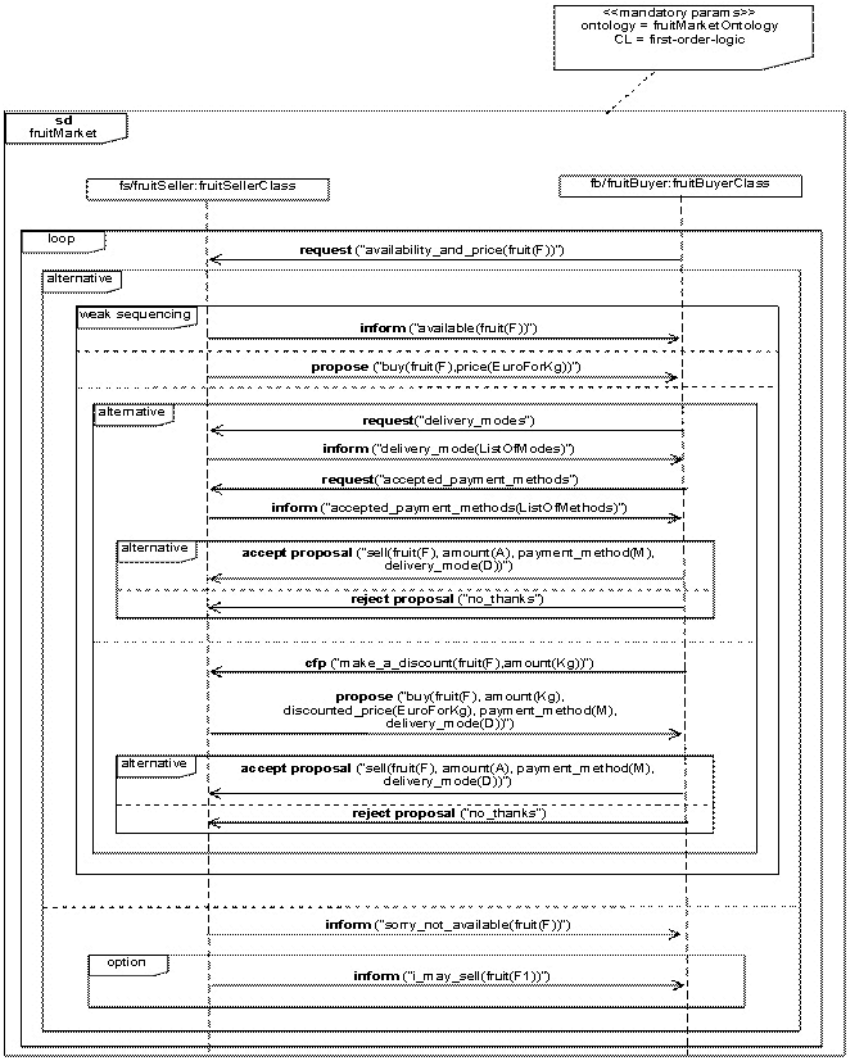


Fig. 1. Fruit marketplace protocol in AUML

```
n-5:      </case>
n-4:     </switch>
n-3:    </sequence>
n-2:   <copy><from opaque="yes"/><to>$continue_1.value</to></copy>
n-1:   </while>
n:   </process>
```

WSDL specification

```
j:   <xs:element name="content_of_Mess_15" type="xs:string"
                      fixed="i_may_sell(fruit(F1))"/>
  ....
k:   <message name="Mess_15">
k+1:   <part name="performative" element="inform_ca"/>
k+2:   <part name="Participant" type="tns:Participant_MsgFromPublisher"/>
k+3:   <part name="Content" element="tns:content_of_Mess_15"/>
k+4:   <part name="Content_Language" element="tns:content_language_name"/>
k+5:   <part name="Ontology" element="tns:ontology_name"/>
k+6:   <part name="Protocol" element="tns:protocol_name"/>
k+7:  </message>
```

For each condition to check in the AIP, a variable is defined in the WS-BPEL document (lines *6-13*), to which opaque values are associated (line *15*, where the condition of the `while` activity, corresponding to the AUML `Loop`, is given a value; line *20*, condition of the `switch` activity corresponding to the AUML `Alternative`; line *n-10*, condition of the `if` activity corresponding to the AUML `Option`). Since WS-BPEL (as AUML) does not allow to express which partner in a communication is responsible for making a condition true or false (namely, for assigning a value to a opaque variable), and since, in a very heterogeneous environment as a multiagent system is, it is not usually possible to know in advance which kind of conditions can be expressed and understood by the participants in a communication, the WS-BPEL document provides no details about conditions. The document just declares that at some point, someone will need to check a condition, and that this condition will need to be satisfied in order to allow the execution to proceed on that protocol branch (`condition = "$continue_1.value = true`, for example).

Apart from the first message of the protocol, that, according to the WS-BPEL specification [1], must be received by the service provider (the agent that publishes the document), both the point of view of the publisher and of the reader are taken into account when describing communicative actions. For example, lines *n-9* and *n-8* describe the delivery of the message identified by the number 15 from the publisher to the reader, both from the publisher's viewpoint, and from the reader's one. The WSDL document describes the details of each exchanged message: message 15 has an `inform` performative (line *k+1*) and `i_may_sell(fruit(F1))` content (line *j*).

A software tool for performing the automatic translation from AUML AIPs to WS-BPEL has been described in [11], and can be downloaded from `http://www.disi.unige.it/person/MascardiV/Software/AUML2WS-BPEL.html`.

3 Web Service-Aware Agents: a Gentle Introduction

A "WS-aware agent", whose main activities are depicted in Figure 2, is an agent able to find services, retrieve WS-BPEL documents specifying AIPs for accessing services, reason about them, and start an interaction with the document's publisher, provided that

the protocol satisfies the agent's desiderata. The core activities of the WS-aware agent

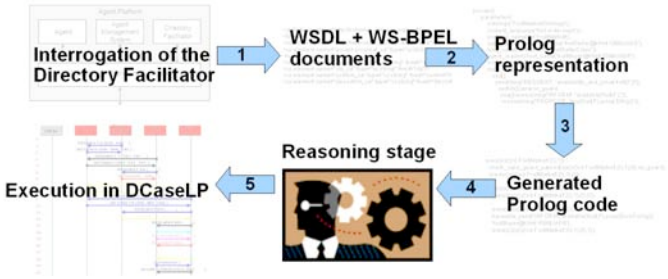


Fig. 2. The Web Service-Aware Agent

are implemented in tuProlog[3] [13] integrated into the JADE agent platform[4] [22] by means of the libraries offered by DCaseLP[5] [20]. The behaviour of the WS-aware agent (that sometimes we also name "reader agent", since it reads the document written and published by the service provider) is defined by the following activities:

1 The WS-aware agent exploits the facilities offered by JADE's "Directory Facilitator" in order to find the service it is interested in, and retrieves the WS-BPEL document specifying the protocol that must be followed in order to obtain the service.
2 By exploiting the JDOM technology[6], the agent translates the WS-BPEL specification of the protocol into an internal format, corresponding to a Prolog term.
3 Starting from the Prolog term, a Prolog program corresponding to a finite state machine where states are placeholders, and transitions are either communicative acts (send and receive) or conditions to check, is generated. This program generation stage is implemented in Prolog.
4 In order to verify if its desiderata (either existential or universal properties such as "There is one path where I will receive $message_1$", or "Whatever the path, I will send $message_2$") are met by the protocol, the agent exploits meta-programming facilities offered by Prolog. It makes a depth-first exploration of the SLD-tree for $P \cup \{G\}$ via R, where P is the program generated in the previous step, G is the goal that starts the execution of the protocol, and R is the leftmost selection rule. This exploration is aimed at either finding one path where the desired message is received (for demonstrating that an existential property holds), or finding a path where the final state is reached, and the expected message is not received (for demonstrating that a universal property does not hold).
5 If the desiderata of the WS-aware agent are met, the agent engages in a dialog with the publisher of the WS-BPEL document. In case the condition to satisfy was a

[3] `http://tuprolog.alice.unibo.it/`
[4] `http://jade.tilab.com/`
[5] `http://www.disi.unige.it/person/MascardiV/Software/DCaseLP.html`
[6] `http://www.jdom.org/`

universal property, the protocol may evolve in whatever way: the property will be satisfied by any path. Otherwise, in case of an existential property, the WS-aware agent may only try to force (as long as the decision is up to it) the execution of the path that satisfies the property, but it must take into account that at any time, the publisher might make a decision that causes another protocol branch to be followed. In this case, the WS-aware agent accepts to follow the protocol although it will not lead to the reception of the expected message. The actual execution of the protocol takes place within the JADE platform extended with the DCaseLP libraries for integrating the tuProlog interpreter.

4 From WS-BPEL to a Prolog Representation

As already anticipated, a WS-BPEL document is composed by a WSDL file and a WS-BPEL file. The WSDL file contains all the data type used to define the business process in the WS-BPEL file.

The first activity that we perform to obtain a Prolog representation of the WS-BPEL process, is to build the JDOM tree of the WSDL file. JDOM allows the programmer to represent an XML document as a tree, and to explore and modify it. In this phase, all the information about the agents and about the messages exchanged in the protocol (message sender, receiver, content, performative act) are extracted from the WSDL file. Then, a JDOM tree of the WS-BPEL file is built, and is visited following a pre-order strategy. When a WS-BPEL activity (`invoke`, `receive`, `seq`, `switch`, `loop`, etc.) is found, an appropriate Prolog representation of this activity is created. The final output is the `main_fragment(WS-BPELStructuredActivity)` term representing the protocol activities. By integrating this representation of the AIP activities with the general information about the agents and the protocol extracted from the WSDL file (protocol name, protocol parameters, etc.), the `process(Parameters, ProtocolName, PublisherData, ReaderData, MainFragment)` term is obtained. The Prolog term representing the fruit market AIP that we use as running example, is shown below.

```
process(
  %%% Ontology and message content language %%%
  parameters(ontology('FruitMarketOntology'),content_language('first order logic')),
  %%% Definition of the protocol name %%%
  protocol_name('sd FruitMarket'),
  %%% Definition of the agent publisher %%%
  agent_publisher(short_name('seller@giocas:1099/JADE'),long_name('fs/fruitSeller:
                                                          fruitSellerClass')),
  %%% Definition of the agent reader %%%
  agent_reader(short_name('buyer@giocas:1099/JADE'),long_name('fb/fruitBuyer:
                                                       fruitBuyerClass')),
  %%% Definition of the AIP activities %%%
  main_fragment(
   %%% Translation of a while activity %%%
   while(no_guard,
    %%% Translation of a sequence activity %%%
    seq([
    %%% Translation of a send activity %%%
    send(msg('REQUEST','availability_and_price(fruit(F))')),
    %%% Translation of an alternative activity %%%
    switch([
     case(no_guard,
     seq([
```

```
%%% Translation of a receive activity %%%
receive(msg('INFORM','available(fruit(F))')),
receive(msg('PROPOSE','buy(fruit(F),price(EuroForKg))')),
switch([
 case(no_guard,
 seq([
 send(msg('REQUEST','delivery_modes')),
 receive(msg('INFORM','delivery_mode(ListOfModes)')),
 send(msg('REQUEST','accepted_payment_methods')),
 receive(msg('INFORM','accepted_payment_methods(ListOfMethods)')),
 switch([
  case(no_guard,
  send(msg('ACCEPT-PROPOSAL','sell(fruit(F),amount(A),payment_method(M),
                                                    delivery_mode(D))'))),
  case(no_guard,
  send(msg('REJECT-PROPOSAL','no_thanks')))])])),
 case(no_guard,
 seq([
 send(msg('CFP','make_a_discount(fruit(F),amount(Kg))')),
 receive(msg('PROPOSE','buy(fruit(F),amount(Kg),discounted_price(EuroForKg),
                                    payment_method(M),delivery_mode(D))')),
 switch([
  case(no_guard,
  send(msg('ACCEPT-PROPOSAL','sell(fruit(F),amount(A),payment_method(M),
                                                    delivery_mode(D))'))),
  case(no_guard,
 send(msg('REJECT-PROPOSAL','no_thanks')))])])])])),
case(no_guard,
seq([
receive(msg('INFORM','sorry_not_available(fruit(F))')),
%%% Translation of an option activity %%%
if_then(no_guard,
receive(msg('INFORM','i_may_sell(fruit(F1))')))])])])))))
```

5 Generating the WS-BPEL-compliant Prolog Program

The philosophy behind the generation of a Prolog program starting from the Prolog representation of the AIP, is that a finite state machine is simulated by the generated clauses. States are meaningless terms only used to enforce the correct transitions, and transitions correspond either to communicative actions (sending or receiving messages), or to check of conditions. In some cases, empty transitions that just move from one state to another, are used. Thus, the transitions can be of four kinds: *send transition*, *receive transition*, *check transition* and *null transition*. They are exemplified below, where the initial and final fragments of the Prolog code corresponding to the fruit marketplace protocol depicted in Figure 1 are shown (the clause numbers written in italic are not part of the code). Note that we did not take care of efficiency in the development of this prototype: states can become very long terms, and no optimisations are made in implementing the transitions.

clause 1:
```
exec(s('sd FruitMarket',0)) :-
  check_guard(s('sd FruitMarket',0), no_guard), exec(s(s('sd FruitMarket',0),0)).
```

clause 2:
```
exec(s('sd FruitMarket',0)) :- exec(s('sd FruitMarket',final)).
```

clause 3:
```
exec(s('sd FruitMarket',1)) :- exec(s('sd FruitMarket',0)).
```

clause 4:
```
exec(s(s('sd FruitMarket',0),0)) :-
  exec(s(s(s('sd FruitMarket',0),0),0)).
```

```
clause 5:  exec(s(s(s('sd FruitMarket',0),0),0)) :-
             send('REQUEST','availability_and_price(fruit(F))','seller@giocas:1099/JADE'),
             exec(s(s(s('sd FruitMarket',0),0),1)).

clause 6:  exec(s(s(s('sd FruitMarket',0),0),1)) :-
             check_guard(s(s(s(s('sd FruitMarket',0),0),1),0), no_guard),
             exec(s(s(s(s('sd FruitMarket',0),0),1),0)).

clause 7:  exec(s(s(s(s('sd FruitMarket',0),0),1),0)) :-
             exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),0)).

clause 8:  exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),0)) :-
             receive('INFORM','available(fruit(F))','seller@giocas:1099/JADE'),
             exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),1)).

clause 9:  exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),1)) :-
             receive('PROPOSE','buy(fruit(F),price(Euro))','seller@giocas:1099/JADE'),
             exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),2)).

...................

clause n-3:  exec(s(s(s(s(s('sd FruitMarket',0),0),1),1),1)) :-
check_guard(s(s(s(s(s('sd FruitMarket',0),0),1),1),1),no_guard),
exec(s(s(s(s(s('sd FruitMarket',0),0),1),1),1),0)).

clause n-2:  exec(s(s(s(s(s('sd FruitMarket',0),0),1),1),1)) :-
exec(s('sd FruitMarket',1)).

clause n-1:  exec(s(s(s(s(s('sd FruitMarket',0),0),1),1),1),0)) :-
receive('INFORM','i_may_sell(fruit(F1))','seller@giocas:1099/JADE'),
exec(s('sd FruitMarket',1)).

clause n:  exec(s('sd FruitMarket',final)) :- true.
```

The predicate `check_guard(State,Guard)` (clauses *1*, *6* and *n-3*) characterises a *check transition*, and succeeds if `call(Guard)` succeeds. Since no guards are specified by the WS-BPEL AIP, their translation is always `no_guard`, and `call(no_guard)` succeeds. The `check_guard(State,Guard)` atom may be manually edited by the developer for inserting the application-dependent conditions that the agent might want to check.

The predicate `send` in clause *5* (resp., `receive`, in clauses *8*, *9* and *n-1*) characterises a *send transition* (resp., a `receive transition`). It is implemented by the DCaseLP libraries, and provides an interface between tuProlog and the communication facilities offered by JADE. It takes the message's FIPA performative [19], the content, and the JADE address of the receiver (resp., sender), as its arguments.

The predicate that performs the generation of the code, takes the initial and final states of the transition, the identifier of the agent, and the term representing the structured activity to translate, and returns a list of clauses that implement the transition. The state that represents the end of the protocol is identified by the constant `final`. The demonstration of `exec(s(ProtocolId,final))` always succeeds (clause *n*).

– *Translating cycles.* A `while(Guard,WhileActivities)` action performed in the state `s(S,I)` for reaching the state `SFinal`, is translated into

```
clause a:  exec(s(S,I)) :- check_guard(s(S,I),Guard), exec(s(s(S,I),0)).
clause b:  exec(s(S,I)) :- exec(SFinal).
clause c:  exec(s(S,I1)) :- exec(s(S,I)).
   Clauses that translate the WhileActivities from s(s(S,I),0) to s(S,I1)
```

Our fruit market AIP starts with a while activity performed in the state `s('sd Fru-`

itMarket',0) for reaching the state s('sd FruitMarket',final). Clause *0* of our code fragment corresponds to the first clause of the translation of the while activity (clause *a*); clause *2* corresponds to clause *b*; and clause *3* to clause *c*. All the remaining activities of the protocol correspond to the *WhileActivities*, and they will need to end with reach state s('sd FruitMarket',1) (corresponding to s(S,I1) in clause *c*). When discussing the translation of options, we will see that this truly happens.

– *Translating communication actions.* A *communication* action (where *communication* may be either send or receive) performed in the state s(S,I) for reaching the state SFinal, is translated into the clause

```
clause a:   exec(s(S, I)) :- communication(Perform,Cont,Addr), exec(SFinal).
```

Examples of this translation are clauses *5*, *8*, *9*, and *n-1*.

– *Translating sequences.* A seq([Activity0, ..., ActivityN]) action performed in the state s(S,I) for reaching the state SFinal, is translated into

```
clause a:   exec(s(S,I)) :- exec(s(s(S,I),0)).
   Clause that translates Activity0 from s(s(S, I),0) to s(s(S,I),1)
   Clause that translates Activity1 from s(s(S, I),1) to s(s(S,I),2)
   ....
   Clause that translates ActivityN from s(s(S,I),N) to SFinal
```

An example of this translation are clauses *4* and *5*. The state s(S, I) from which the translation starts is s(s('sd FruitMarket',0),0) (S = s('sd FruitMarket',0); I = 0) and the state to reach is s('sd FruitMarket',1). Clause *4* corresponds to clause *a*, while clause *5* corresponds to the translation of the first activity within the sequence, send('REQUEST', 'availability_and_price(fruit(F))','seller-@giocas:1099/JADE'), from s(s(S,I),0) to s(s(S,I),1). Clause *n-1* corresponds to the very last activity in the sequence. Also clauses *8* and *9* translate two items of a sequence, started in clause *7*.

– *Translating alternatives.* A switch([case(Guard0,Activity0), ..., case(GuardN,ActivityN)]) action performed in the state s(S,I) for reaching the state SFinal, is translated into

```
clause a:   exec(s(S,I)) :- check_guard(s(s(S,I),0),Guard0), exec(s(s(S,I),0)).
   Clauses that translates Activity0 from s(s(S,I),0) to SFinal
....
clause z:   exec(s(S,I)) :- check_guard(s(s(S,I),N),GuardN), exec(s(s(S,I),N)).
   Clauses that translates ActivityN from s(s(S,I),N) to SFinal
```

Clause *6* is an example of translation of a switch activity, and corresponds to clause *a*. Clauses *7*, *8*, *9* and successive ones correspond to the alternative branch where the fruit is available; another clause with the same head exec(s(s(s('sd FruitMarket',0),0),1)) as clause *6*, not shown in the program fragment, corresponds to clause *z*, namely to the alternative branch where the fruit is not available.

– *Translating options.* An if_then(Guard,Then_activities) action performed in the state s(S,I) for reaching the state SFinal, is translated into

```
clause a:   exec(s(S,I)) :- check_guard(s(S,I),Guard), exec(s(s(S,I),0)).
clause b:   exec(s(S,I)) :- exec(SFinal).
   Clauses that translate ThenActivities from s(s(S,I),0) to SFinal
```

Clauses *m-3* and *m-2* correspond to clauses *a* and *b* respectively, where s(S,I) corresponds to s(s(s(s('sd FruitMarket',0),0),1),1),1) and SFinal to s('sd

FruitMarket',1). *ThenActivities* correspond to the reception of the message 'i_may_sell(fruit(F1))', after which the agent moves to s('sd FruitMarket',1) (clause *m-1*), as anticipated when we introduced the translation of cycles.

6 Is This Protocol the Right One for Me?

In order to complete the protocol execution with a success, the Prolog program of the WS-aware agent (the fruit buyer in our running example) must contain the fact condition_to_check(Condition), where Condition must be instantiated with exists(Action), forall(Action), or no_cond. The default for this fact is condition_to_check(no_cond), meaning that no checks on the protocol are made, but it may be manually edited by the MAS developer if he/she wants to verify some conditions. Action may correspond to one of the transition types, *send*, *receive*, *check*: the WS-aware agent may thus check that there is one possible path where (resp. in any possible path) a send(Performative,Content,Receiver), or a receive(Performative,Content,Sender), or a check_guard(State,Guard), is executed.

Since, as already explained, for the moment we do not instantiate guards, and since the WS-aware agent should have control over its own communicative actions, the most interesting type of condition to check involves the reception of a message from the WS publisher (the fruit seller of our example).

To go on with our fruit market example, let as consider a "restrictive" fruit buyer agent that accepts to interact with the fruit seller agent only if it provides a bunch of payment methods among which the buyer can choose. Assuming the existence of an ontology shared between the fruit buyer and the fruit seller, that allows them to exchange messages whose content has been previously agreed upon, the fruit buyer agent's code should contain the fact: condition_to_check(forall(receive('INFORM','accepted_payment_methods(ListOfMethods)',Sender))). This condition is not verified by the protocol depicted in Figure 1 and discussed throughout the paper, as it can be easily seen. Thus, the "restrictive" fruit buyer agent does not even start the protocol execution.

However, a "flexible" fruit buyer might just want to check if, in the best case, the fruit seller would allow it to choose among more than one payment method. The buyer might force the execution of the protocol branch where this possibility takes place, as long as the choice is up to it, but it might also be ready to accept that the seller, at some point in the protocol execution, acts in such a way that the execution of the desired action can no longer take place. The condition to check for the "flexible" agent would be exists(receive('INFORM','accepted_payment_methods(ListOfMethods)', Sender)). The protocol does verify this condition, and the agent would start the protocol execution.

Now, three situations may take place

1. The actual protocol branch executed is the one expected by the fruit buyer: w.r.t. our protocol, this means that the fruit seller had enough fruit of the required type to sell, and sends an inform(available(fruit(F))) message to the buyer. The buyer then requests the delivery and payment modes, and the seller provides the expected answer. From now on, the protocol can follow whatever branch.

2. The branch executed does not allow to verify the condition: the seller has not enough fruit to sell, and thus the branch where the buyer could perform a `receive ('INFORM','accepted_payment_methods(ListOfMethods)',Sender)` action can no longer be taken. The fruit buyer gives up with its hope of receiving this message, and goes on following the protocol branch determined by the seller.
3. The fruit seller sends some message that was not foreseen at all by the protocol. The buyer stops to interact with the seller, and the protocol execution fails.

While executing, the WS-aware explains its actions by printing them on a log. A fragment of this explanation in the case that everything goes as expected, is:

```
I will try to enforce the following path, as long as the choice is up to me

check_guard(s('sd FruitMarket',0),no_guard)
send('REQUEST','availability_and_price(fruit(F))','seller@giocas:1099/JADE')
check_guard(s(s(s('sd FruitMarket',0),0),1),0),no_guard)
receive('INFORM','available(fruit(F))','seller@giocas:1099/JADE')
receive('PROPOSE','buy(fruit(F),price(EuroForKg))','seller@giocas:1099/JADE')
check_guard(s(s(s(s(s('sd FruitMarket',0),0),1),0),2),0),no_guard)
send('REQUEST',delivery_modes,'seller@giocas:1099/JADE')
receive('INFORM','delivery_mode(ListOfModes)','seller@giocas:1099/JADE')
send('REQUEST',accepted_payment_methods,'seller@giocas:1099/JADE')
receive('INFORM','accepted_payment_methods(ListOfMethods)','seller@giocas:1099/JADE')

***************************** ACTUAL EXECUTION ******************************

I executed the statement check_guard(s('sd FruitMarket',0),no_guard)
I am still following the desired path!

.........

Nondeterministic action: I hoped to receive
('INFORM','delivery_mode(ListOfModes)','seller@giocas:1099/JADE')
The message that I received is the one I was waiting for!

I executed the statement
send('REQUEST',accepted_payment_methods,'seller@giocas:1099/JADE')
I am still following the desired path!

Nondeterministic action: I hoped to receive
('INFORM','accepted_payment_methods(ListOfMethods)','seller@giocas:1099/JADE')
The message that I received is the one I was waiting for!

Finally, I have reached my goal!
```

7 Let's Run!

For supporting the interaction between the Service Provider Agent and the WS-Aware Agent, we have designed and implemented the system depicted in Figure 3. In this section, we will name "Agent Services" (ASs) both the general management services offered by JADE's Directory Facilitator and by the Protocol Manager Agent that we implemented, and the application specific services such as the `fruit-selling` service offered by the fruit seller and advertised by publishing the WS-BPEL document discussed in Section 2.

The Directory Facilitator agent (DF) is provided by JADE, and offers "yellow pages" allowing agents to publish ASs, so that other agents can find and exploit them. An agent

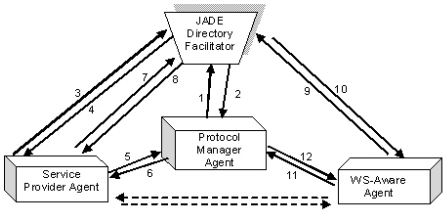


Fig. 3. Our multiagent system implemented in JADE

wishing to publish an AS must send information about itself and about the AS it provides, to the DF.

We have developed a Protocol Manager Agent (PMA) that allows agents to publish and retrieve WS-BPEL specifications representing AIPs. In other words, the PMA offers an AS that consists in the advertisement and retrieval of WSs' specifications. The PMA stores the WS-BPEL specifications received on a MySql DBMS[7].

When the JADE platform is started, the PMA registers the `protocol-publishing` and the `protocol-reading` ASs to the DF (arrows 1 and 2 in the figure 3).

When a Web Service Provider Agent (WSPA) wants to advertise an AS specified by means of WS-BPEL, it

1. looks in the DF to find the `protocol-publishing` AS (arrows 3, 4);
2. sends a message to the PMA with the WS-BPEL document, and waits to receive the protocol identifier (PID) assigned by the PMA to it (arrows 5, 6);
3. registers the AS specified by the WS-BPEL document to the JADE DF (arrows 7, 8), adding the PID obtained by the PMA and the PMA address to the AS properties.

When a WS-Aware Agents (WSAA) looks for an AS, it

1. queries the DF to find the required AS (arrows 9, 10). If a WSPA had previously registered the AS, then the WSAA obtains the name and address of the service provider, the address of the PMA, and the PID that the WSPA assigned to the AS;
2. sends a message to the PMA to obtain the WS-BPEL document representing the AIP that it must follow to obtain the AS (arrows 11, 12);
3. from the WS-BPEL specification, generates a corresponding Prolog term as discussed in Section 4;
4. generates the Prolog program from the Prolog term, as discussed in Section 5;
5. reasons about the Prolog program corresponding to the WS-BPEL AIP as discussed in Section 6; and
6. according to the reasoning outcome, eventually starts the protocol-compliant communication in order to obtain the AS (direct communication between the WSPA and the WSAA, represented by dashed arrows in Figure 3).

Figure 4 refers to an execution run of the multiagent system composed by one PMA, one WSAA, one WSPA and the DF. In this figure, the WSPA plays the role

[7] `http://www.mysql.com/`

of `fruitSeller` while the WSAA plays the role of `fruitBuyer`. The first twelve messages correspond to the communication represented by the twelve solid arrows in Figure 3, while the other messages are exchanged during the execution of the fruitMarket AIP, aimed at allowing the `fruitBuyer` to obtain the `fruit-selling` AS from the `fruitSeller`. The execution run shown here corresponds to the situation where the conditions on the protocol execution put by the buyer are all met, and the seller's proposal is accepted.

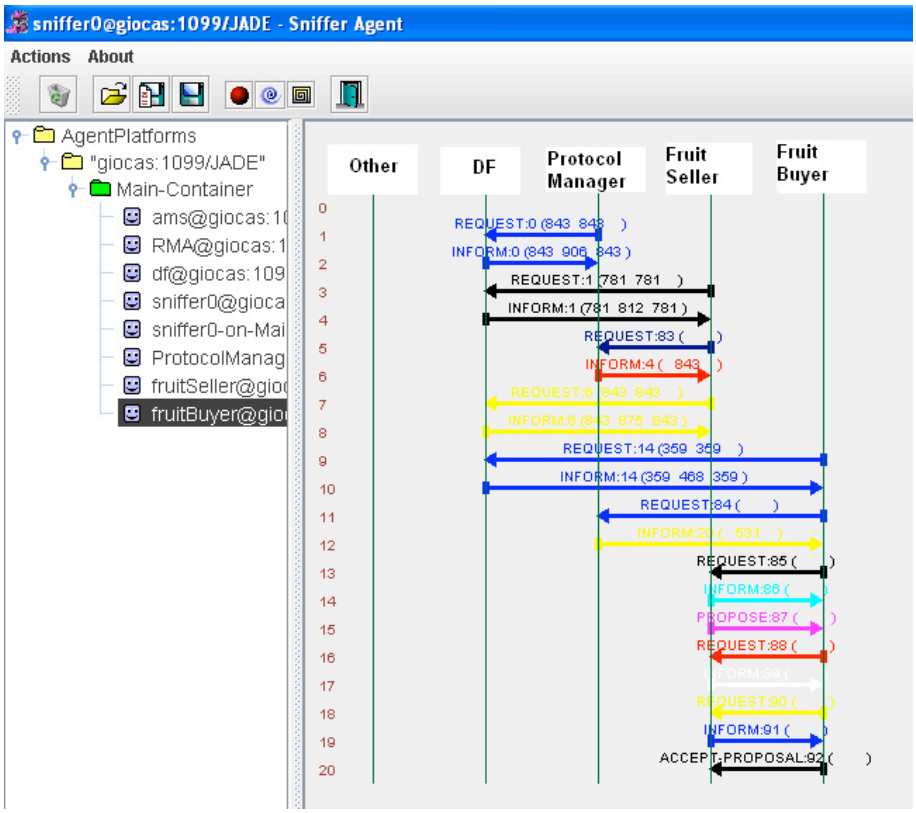


Fig. 4. Execution run in JADE

8 Conclusions

WSs are defined as heterogeneous, distributed, loosely coupled software applications. These characteristics make WSs a very flexible and scalable technology; standard languages for describing (WSDL), coordinating (WSCI, [3]), and defining business processes based on WSs (WS-BPEL) exist, and are known by most designers and develop-

ers of distributed web applications. This is one of the main reasons of the WSs' success. Despite to these advantages, WSs still remain something "static". In fact, a WS can be considered as a set of operations with predefined input and output types, unable to show any reasoning, learning, or other sophisticated capability that might allow it to dynamically adapt to the changing environment where it is situated.

On the other hand, agents are defined as autonomous, interactive, context-aware, distributed entities, working in heterogeneous, dynamic, unpredictable and open environments. Though agents are definitely more "dynamic" then WSs, their characterising features remain at a very high level, and no final agreement on standard languages for their specification and implementation has still been reached.

Many researchers today view agents either as a coordination framework for WSs or as software entities that can use WSs. In our opinion, instead, there are many applications where agents should not just use WSs, but should *extend and substitute them at all*. In fact, as pointed out by Dickinson and Wooldridge in [14], the key conceptual difference between agents and WSs is that only agents can be described in terms of human-like mental attitudes, such as Beliefs, Desires and Intentions. In those web applications where mimicking the human way of thinking may make a difference, intelligent agents built on top of the WS technology might be used instead of WSs.

This paper proposes our first step towards the usage of well established WS-related technologies to make agent-related technologies and models more concrete and effective.

The exploitation of CL makes the implementation of an "intelligent" and "human-like" behaviour easier than with other programming approaches. Although the reasoning capabilities of our WS-aware agents are currently pretty limited, the usage of CL offers to us a great potential of improvement in the near future. In fact, we are actively collaborating with other researchers involved in the Italian project MIUR PRIN 2005 "Specification and verification of agent interaction protocols" [16], aimed at proving the applicability of declarative approaches for 1) defining suitable formalisms for specifying and verifying interaction protocols, 2) developing techniques for automatic property verification and reasoning about web services, and 3) translating modelling languages into the formal languages developed in the project. We are exploring how our WS-aware agents could be implemented in dynamic linear time and/or abductive logic programming, in order to take advantage of the results already obtained by other partners in the project, in the areas of on- and off-line verification of the compliance of an agent to a protocol.

References

1. Web Services Business Process Execution Language (WS-BPEL) Version 2.0, 2005. Oasis Committee Draft 01 Sep.2005.
2. M. Alberti, D. Daolio, P. Torroni, M. Gavanelli, E. Lamma, and P. Mello. Specification and verification of agent interaction protocols in a logic-based system. In *Proc. of SAC'04*, pages 72–78. ACM, 2004.
3. A. Arkin and et al. Web service choreography interface (WSCI) 1.0. W3C Note 2002-08-08.
4. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about conversation protocols in a logic-based agent language. In *Proce. of AIIA'03*. Springer-Verlag, 2003. LNAI 2829.

5. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for customizing web service selection and composition. *Journal of Logic and Algebraic Programming, special issue on Web Services and Formal Methods*, 2006. To appear.
6. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *Proc. of EPEW'05 and WS-FM'05*, pages 257–271. Springer-Verlag, 2005.
7. M. Bozzano and G. Delzanno. Automated protocol verification in linear logic. In *Proc. of PPDP '02*, pages 38–49. ACM Press, 2002.
8. M. Bozzano and G. Delzanno. Automatic verification of secrecy properties for linear logic specifications of cryptographic protocols. *J. Symb. Comput.*, 38(5):1375–1415, 2004.
9. L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. CooWS: Adaptive BDI agents meet service-oriented programming. In *Proc. of ICWI'05*, pages 205–209. IADIS Press, 2005.
10. P. A. Buhler and J. M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management*, 6:61–87, 2005.
11. G. Casella and V. Mascardi. AUML e WS-BPEL: Ingegnerizzare e pubblicare su web protocolli di interazione tra agenti. *Intelligenza Artificiale*, 2006. To appear.
12. P. Ciancarini and M. Wooldridge. Agent-oriented software engineering: The state of the art. In *Proc. of AOSE'00*, pages 1–28. Springer-Verlag, 2000.
13. E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.
14. I. Dickinson and M. Wooldridge. Agents are not (just) web services: considering BDI agents and web services. In *Proc. of SOCABE'2005*, 2005.
15. J. Dix, F. Sadri, and K. Satoh, editors. Annals of Mathematics and Artificial Intelligence, 37(1-2). 2003.
16. A. Martelli et al. Modeling, verifying and reasoning about web services. Submitted to ALPSWS'06, 2006.
17. E. Christensen et al. Web Services Description Language (WSDL) 1.1, 2001. W3C Note 15 Mar. 2001.
18. M-P. Huget et al. FIPA modeling: Interaction diagrams. First proposal, 2003-07-02, 2003.
19. Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Standard 2002-12-06. `http://www.fipa.org/specs/fipa00061/`, 2002.
20. I. Gungui, M. Martelli, and V. Mascardi. DCaseLP: a prototyping environment for multilingual agent systems. Technical report, DISI, Univ. of Genova, Italy, 2005. DISI-TR-05-20.
21. H. Haas and A. Brown. Web Services Glossary – W3C Working Group Note 2004-02-11, 2004.
22. JADE Home Page. `http://jade.tilab.com/`.
23. M. Luck, P. McBurney, O. Shehory, S. Willmott, and the AL Community. *Agent Technology: Computing as Interaction – A Roadmap for Agent-Based Computing*. AgentLink III, 2005.
24. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *TPLP Journal*, 4(4):429–494, 2004.
25. S. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. A. Williams, editors, *Proc. of KR'02*, pages 482–496. Morgan Kaufmann, 2002.
26. S. A. McIlraith. Towards declarative programming for web services. In *Proc. of SAS'04*, page 21. Springer, 2004. LNCS 3148.
27. J. Rao, P. Küngas, and M. Matskin. Logic-based web services composition: From service description to process model. In *Proc. of ICWS'04*, pages 446–453. IEEE, 2004.
28. F. Sadri and F. Toni. Computational Logic and Multi-Agent Systems: a Roadmap. Technical report, Department of Computing, Imperial College, London, 1999.
29. C. Walton. Uniting agents and web services. *AgentLink News*, 18:26–28, 2005.

Efficient Web Service Discovery and Composition using Constraint Logic Programming

Srividya Kona, Ajay Bansal, Gopal Gupta[1]
and Thomas D. Hite[2]

[1] Department of Computer Science
The University of Texas at Dallas
[2] Metallect Corp.
2400 Dallas Parkway, Plano, TX 75093

Abstract. Service-oriented computing is gaining wider acceptance. For Web services to become practical, an infrastructure needs to be supported that allows users and applications to discover, deploy, compose and synthesize services automatically. This automation can take place effectively only if formal semantic descriptions of Web services are available. In this paper we present an approach for automatic service discovery and composition with both syntactic and semantic description of Web services. In syntactic case, we use a repository of services described using WSDL (Web Service Description Language). In the semantic case, the services are described using USDL (Universal Service-Semantics Description Language), a language we have developed for formally describing the semantics of Web services. In this paper we show how the challenging task of building service discovery and composition engines can be easily implemented and efficiently solved via (Constraint) Logic programming techniques. We evaluate the algorithms on repositories of different sizes and show the results.

1 Introduction

A Web service is a program accessible over the web that may effect some action or change in the world (i.e., causes a side-effect). Examples of such side-effects include a web-base being updated because of a plane reservation made over the Internet, a device being controlled, etc. The next milestone in the Web's evolution is making *services* ubiquitously available. As automation increases, these Web services will be accessed directly by the applications rather than by humans [8]. In this context, a Web service can be regarded as a "programmatic interface" that makes application to application communication possible. An infrastructure that allows users to discover, deploy, synthesize and compose services automatically is needed in order to make Web services more practical.

To make services ubiquitously available we need a semantics-based approach such that applications can reason about a service's capability to a level of detail that permits their discovery, deployment, composition and synthesis [3]. Several

efforts are underway to build such an infrastructure. These efforts include approaches based on the semantic web (such as USDL [1], OWL-S [4], WSML [5], WSDL-S [6]) as well as those based on XML, such as Web Services Description Language (WSDL [7]). Approaches such as WSDL are purely syntactic in nature, that is, it only addresses the syntactical aspects of a Web service [17].

Given a formal description of the context in which a service is needed, the service(s) that will precisely fulfill that need can be automatically determined. This task is called discovery. If the service is not found, the directory can be searched for two or more services that can be composed to synthesize the required service. This task is called composition. In this paper we present an approach for discovery and composition of Web services. We show how these tasks can be performed using both syntactic and semantic descriptions of Web services.

The rest of the paper is organized as follows. We present different approaches to the description of Web services in section 2 with brief description of WSDL and USDL. Section 3 describes the two major Web services tasks namely discovery and composition with their formal definitions. In section 4, we present our multi-step narrowing based solution for automatic service discovery and composition. Then we show the high-level design of our system with brief descriptions of the different components in section 5. Various efficiency and scalability issues are discussed in section 6. Then we show performance results of our discovery and composition algorithm in section 7. Finally we present our conclusions.

2 Description of Web Services

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface that is described in a machine-processible format so that other systems can interact with the Web service through its interface using messages. Currently WSDL (Web Services Description Language) [7] is used to describe Web services, but it is only syntactic in nature. The automation of Web service tasks (discovery, composition, etc.) can take place effectively only if formal semantic descriptions of Web services are available. For formally describing the semantics of Web services we have developed a language called USDL (Universal Service-Semantics Description Language). The motivation and details of USDL can be found in [1]. This section presents an overview of both the syntactic approach (WSDL) and the semantic approach (USDL) for description of Web services.

2.1 WSDL: Web Services Description Language

WSDL is an XML-based language used for describing the interface of a Web service. It describes services as a set of operations (grouped into ports) operating on messages containing either document-oriented or procedure-oriented information. WSDL descriptions are purely syntactic in nature, that is, they merely specify the format of messages to be exchanged by invocable operations.

Below is an example WSDL description of a *FlightReservation* service, similar to a service in the SAP ABAP Workbench Interface Repository for flight reservations [9], that takes in a *CustomerName*, *FlightNumber*, and *DepartureDate* as inputs and produces a *FlightReservation* as the output.

```
<definitions ...>
  <portType name="ReserveFlight_Service">
    <operation name="ReserveFlight">
      <input message="ReserveFlight_Request"/>
      <output message="ReserveFlight_Response"/>
    </operation>
  </portType>
  <message name="#ReserveFlight_Request">
    <part name="#CustomerName" type="xsd:string">
    <part name="#FlightNumber" type="xsd:string">
    <part name="#DepartureDate" type="xsd:date">
  </message>
  <message name="ReserveFlight_Response">
    <part name="FlightReservation" type="xsd:string"/>
  </message>
</definitions>
```

In order to allow interoperability and machine-readability of web documents, a common conceptual ground must be agreed upon. The first step towards this common ground are standard languages such as WSDL and OWL [15]. However, these do not go far enough, as for any given type of service there are numerous distinct representations in WSDL and for high-level concepts (e.g., a ternary predicate), there are numerous disparate representations in terms of OWL, representations that are distinct in terms of OWL's formal semantics, yet equal in the actual concepts they model. This is known as the semantic aliasing problem: distinct syntactic representations with distinct formal semantics yet equal conceptual semantics. For the semantics to equate things that are conceptually equal, we need to standardize a sufficiently comprehensive set of basic concepts, i.e., a universal ontology, along with a restricted set of connectives.

Industry specific ontologies along with OWL can also be used to formally describe Web services. This is the approach taken by the OWL-S language [4]. The problem with this approach is that it requires standardization and undue foresight. Standardization is a slow, bitter process, and industry specific ontologies would require this process to be iterated for each specific industry. Furthermore, reaching a industry specific standard ontology that is comprehensive and free of semantic aliasing is even more difficult. Undue foresight is required because many useful Web services will address innovative applications and industries that don't currently exist. Standardizing an ontology for travel and finances is easy, as these industries are well established, but new innovative services in new upcoming industries also need to be ascribed formal meaning. A universal ontology will have no difficulty in describing such new services.

2.2 USDL: Universal Service-Semantics Description Language

USDL is a language that service developers can use to specify formal semantics of Web services [1]. We need an ontology that is somewhat coarse-grained yet universal, and at a similar conceptual level to common real world concepts. WordNet [10] is a sufficiently comprehensive ontology that meets these criteria. USDL uses OWL WordNet ontology [11] thus providing a universal, complete, and tractable framework, which lacks the semantic aliasing problem, to which Web service messages and operations are mapped. As long as this mapping is precise and sufficiently expressive, reasoning can be done within the realm of OWL by using automated inference systems (such as, one based on description logic), and thus automatically reaping the wealth of semantic information in the OWL WordNet ontology that describes relations between ontological concepts, like subsumption (hyponym-hypernym) and equivalence (synonym) relations.

USDL can be regarded as providing semantics to WSDL statements. Thus, if WSDL can be regarded as a language for formally specifying the syntax of Web services, USDL can be regarded as a language for formally specifying their semantics. USDL allows sophisticated conceptual modeling and searching of available Web services, automated composition, and other forms of automated service integration. For example, the WSDL syntax and USDL semantics of Web services can be published in a directory which applications can access to automatically discover services. USDL is perhaps the first attempt to capture the semantics of Web services in a universal, yet decidable manner. Instead of documenting the function of a service as comments in English, one can write USDL statements that describe the function of that service. USDL relies on a universal ontology (OWL WordNet Ontology) to specify the semantics of atomic services.

USDL describes a service in terms of *portType* and *messages*, similar to WSDL. The formal class definitions and properties of USDL in OWL are available at [12]. The semantics of a service is given using the OWL WordNet ontology: portType (operations provided by the service) and messages (operation parameters) are mapped to disjunctions of conjunctions of (possibly negated) concepts in the OWL WordNet ontology. The semantics is given in terms of how a service *affects* the external world. USDL assumes that each side-effect is one of following four operations: *create, update, delete,* or *find*. A generic *affects* side-effect is used when none of the four apply. An application that wishes to use a service automatically should be able to reason with WordNet atoms using the OWL WordNet ontology. The syntactic terms describing portType and messages are mapped to disjunctions of conjunctions of (possibly negated) OWL WordNet ontological terms. A service is then formally defined as a function, labeled by the side-effect. Using USDL, conditions/constraints on the service can also be described. Below is the USDL description of the *FlightReservation* service.

```
<definitions>
  <portType rdf:about="#Flight_Reservation">
    <hasOperation rdf:resource="#ReserveFlight">
  </portType>
  <operation rdf:about="#ReserveFlight">
```

```
    <hasInput rdf:resource="#ReserveFlight_Request"/>
    <hasOutput rdf:resource="#ReserveFlight_Response"/>
    <creates rdf:resource="#FlightReservation" />
  </operation>
  <Message rdf:about="#ReserveFlight_Request">
    <hasPart rdf:resource="#CustomerName"/>
    <hasPart rdf:resource="#FlightNumber"/>
    <hasPart rdf:resource="#DepartureDate"/>
  </Message>
  <QualifiedConcept rdf:about="#CustomerName">
    <isA rdf:resource="#Name"/>
    <ofKind rdf:resource="#Customer"/>
  </QualifiedConcept>
  <BasicConcept rdf:about="#Name">
    <isA rdf:resource="&wn;name"/>
  </BasicConcept>
  <BasicConcept rdf:about="#Customer">
    <isA rdf:resource="&wn;customer"/>
  </BasicConcept>
  <!-- Similarly FlightNumber, DepartureDate are defined -->
</definitions>
```

3 Automated Web service Discovery and Composition

Discovery and Composition are two of the major tasks related to Web services. In this section we formally describe these tasks as *The Discovery Problem* and *The Composition Problem*. Both these problems have a syntactic and a semantic version which are also described below.

3.1 The Discovery Problem

Given a repository of Web services, and a query (i.e., the requirements of the requested service; we refer to it as the query service in the rest of the text), automatically finding a service from the repository that matches these requirements is the Web service Discovery problem. This problem comes in two flavors: syntactic and semantic, depending on the type of service descriptions provided in the repository. All those services that produce at least the requested output parameters and use only from the provided input parameters can be valid solutions. Some of the solutions may be a little over-qualified, but they are still considered as long as they fulfill the input and output parameter requirements.

Definition: Let $\mathcal{R}$ be the set of services in a Web services repository. For simplicity, a service is represented as a pair of its input and output sets. Then let $\mathcal{Q} = (\mathcal{I}', \mathcal{O}')$ be a query service. The Discovery problem can be defined as automatically finding a set $\mathcal{S}$ of services such that $\mathcal{S} = \{s \mid s = (\mathcal{I}, \mathcal{O}), \mathrm{s} \in \mathcal{R}, \mathcal{I} \sqsubseteq \mathcal{I}',$

$\mathcal{O} \sqsupseteq \mathcal{O}'\}$. The meaning of the $\sqsubseteq$ relation depends on whether it is syntactic or semantic discovery. For syntactic discovery the $\sqsubseteq$ relation is the subset relation and for semantic discovery it is the subsumption (subsumes) relation. Figure 1 explains the discovery problem pictorially.

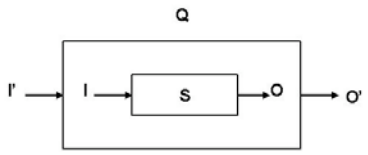


Fig. 1. Substitutable Service

Syntactic Discovery: WSDL provides syntactic description of Web services which can be provided in a repository. Given a query with requirements of the requested service, the discovery problem involves finding a specific service that can fulfill the given input and output criteria in the query based on a syntactical equivalence of the input and output names.

Semantic Discovery: We assume that a directory of services has already been compiled, and that this directory includes a USDL description document for each service. Inclusion of the USDL description, makes service directly "semantically" searchable. However, we still need a query language to search this directory, i.e., we need a language to frame the requirements on the service that an application developer is seeking. USDL itself can be used as such a query language. A USDL description of the desired service can be written, a query processor can then search the service directory for a "matching" service.

3.2 The Composition Problem

Given a repository of service descriptions, and a query with the requirements of the requested service, in case a matching service is not found, the composition problem involves automatically finding a chain of services that can be put together in correct order of execution to obtain the desired service. This problem also can be either syntactic or semantic depending on the kind of service descriptions provided in the repository. Web service discovery problem can be treated as a special case of the Web service composition problem where the length of the chain of services is one.

Definition: Let $\mathcal{R}$ be the set of services in a Web services repository. For simplicity, a service is represented as a pair of its input and output sets. Then let $\mathcal{Q} = (\mathcal{I}', \mathcal{O}')$ be a query service. The Composition problem can be defined as automatically finding a sequence $\mathcal{S}$ of services such that $\mathcal{S} = (\ \mathcal{S}_1,\ \mathcal{S}_2,\ ...,\ \mathcal{S}_n\ )$ where for all i, $\mathcal{S}_i \in \mathcal{R}$, $\mathcal{S}_i = (\mathcal{I}_i,\ \mathcal{O}_i)$ and $\mathcal{I}' \sqsupseteq \mathcal{I}_1,\ \mathcal{O}_1 \sqsupseteq \mathcal{I}_2,\ ...,\ \mathcal{O}_n \sqsupseteq \mathcal{O}'$.

The meaning of the $\sqsubseteq$ relation depends on whether it is syntactic or semantic composition. For syntactic composition the $\sqsubseteq$ relation is the subset relation and for semantic composition it is the subsumption (subsumes) relation. Figure 2 explains the composition problem pictorially.

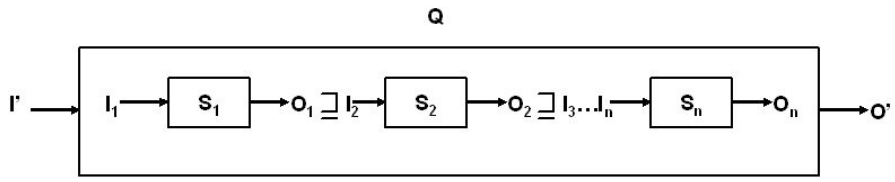


Fig. 2. Composite Service

Syntactic Composition: WSDL descriptions are provided in the repository. The Composition problem involves deriving a possible sequence of services where only the provided input parameters are used for the services and at least the required output parameter is provided as an output by the chained services. The goal is to derive a single solution, where the aim is to keep the list of involved services minimal. In the sequence of services, the outputs of a service are fed in as inputs of the next subsequent service.

Semantic Composition: USDL descriptions are provided in the repository. For service composition, the first step is finding the set of composable services. USDL itself can be used to specify the requirements of the composed service that an application developer is seeking. Using the discovery engine, individual services that make up the composed service can be selected. Part substitution technique [2] can be used to find the different parts of a whole task and the selected services can be composed into one by applying the correct sequence of their execution. The correct sequence of execution can be determined by the pre-conditions and post-conditions of the individual services. That is, if a subservice $\mathcal{S}_1$ is composed with subservice $\mathcal{S}_2$, then the post-conditions of $\mathcal{S}_1$ must imply the pre-conditions of $\mathcal{S}_2$.

4 A Multi-step Narrowing based Solution

With the formal definition of the Discovery and Composition problem, presented in the previous section, one can see that there can be many approaches to solving the problem. Our approach is based on a multi-step narrowing of the list of candidate services using various constraints at each step. In this section we discuss our Discovery and Composition algorithms in detail.

4.1 Discovery Algorithm:

The Discovery routine takes in the query parameters and produces a list of matching services. Our algorithm first uses the query output parameters to narrow down the list of services in the repository. It gets all those services that produce at least the query outputs. In case of the semantic approach, the output parameters provided by a service must be equivalent to or be subsumed by the required output in the query. From the list of services obtained, we find the set of all inputs parameters of all services in the list, say *I*. Then a set of wrong/bad inputs, say *WI* is obtained by computing the set difference of *I* and the query inputs *QI*. Then the list of services is further narrowed down by removing any service that has even one of the inputs from the set *WI*. After all such services are removed, the remaining list is our final list of services called *Result*. Figure 3 shows a pictorial representation of our discovery engine.

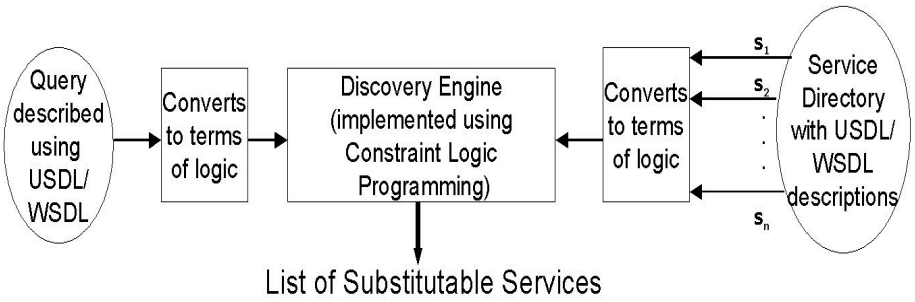


Fig. 3. Discovery Engine

Algorithm: Discovery
Input: QI - QueryInputs, QO - QueryOutputs
Output: Result - ListOfServices
1. L ← NarrowServiceList(QO);
2. I ← GetAllInputParameters(L);
3. WI ← GetWrongInputs(I, QI); i.e., WI = I - QI
4. Result ← FilterServicesWithWrongInputs(WI, L);
5. Return Result;

4.2 Composition Algorithm:

The composition routine also starts with the query output parameters. It first finds a list of all those services which produce outputs such that they are equivalent to or are subsumed by the required output in the query. From the list obtained, for each service the algorithm fetches their input parameters, say I' and tries to find all those services from the repository that produce I' as outputs. The goal is to derive a single solution, which is a list of services that can

be composed together to produce the requested service in the query. The aim is also to keep the list of involved services minimal. Figure 4 shows a pictorial representation of our composition engine.

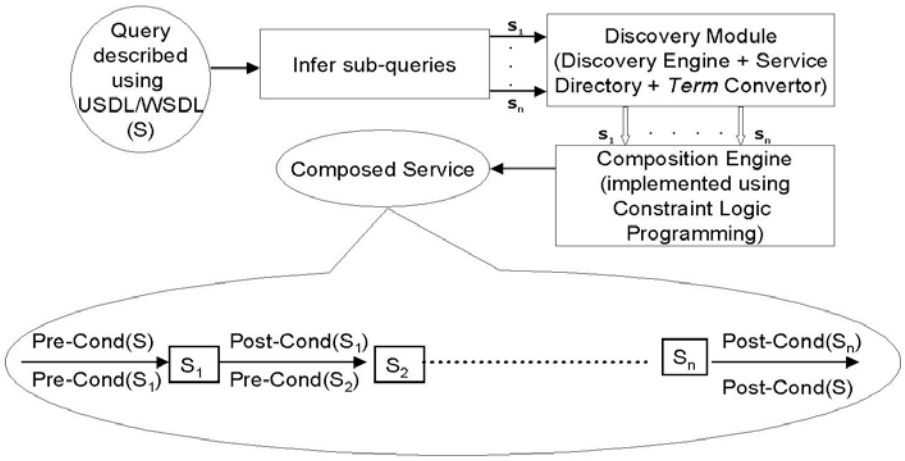


Fig. 4. Composition Engine

Algorithm: Composition
Input: QI - QueryInputs, QO - QueryOutputs
Output: Result - ListOfServices
1. L ← NarrowServiceList(QO);
2. For each service S in L
3. Add S to the Result List;
4. I ← GetAllInputParameters(S);
5. L′ ← NarrowServiceList(I); i.e. find services which produce I as output
6. Repeat the loop lines 2-5 on the new List L′;
7. End For
8. Return Result;

5 Implementation

Our discovery and composition engine is implemented using Prolog [14] with Constraint Logic Programming over finite domain [13], referred to as CLP(FD) hereafter. The high-level design of the Discovery and Composition engines is shown in Figure 5. The software system is made up of the following components.

5.1 Triple Generator

The triple generator module converts each service description into a triple. In syntactic approach WSDL descriptions are converted to triples like:

(null, affects(null, I, O), null).

WSDL being syntactic in nature, does not provide any information about Pre/Post-Conditions and side-effects. So we use the generic *affects* for all services. In the semantic approach the USDL descriptions are converted to triples like:

(Pre-Conditions, affect-type(affected-object, I, O), Post-Conditions).

The function symbol *affect-type* is the side-effect of the service and *affected object* is the object that changed due to the side-effect. I is the list of inputs and O is the list of outputs. *Pre-Conditions* are the conditions on the input parameters and *Post-Conditions* are the conditions on the output parameters. Services are converted to triples so that they can be treated as terms in first-order logic and specialized unification algorithms can be applied to obtain exact, generic, specific, part and whole substitutions [2]. In case conditions on a service are not provided, the *Pre-Conditions* and *Post-Conditions* in the triple will be null. Similarly if the affect-type is not available, this module assigns a generic affect to the service.

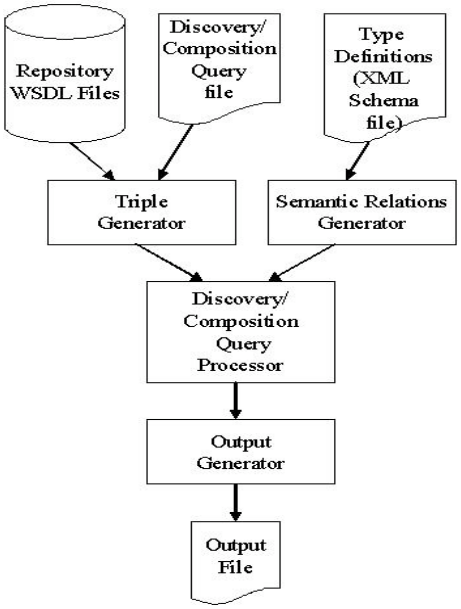


Fig. 5. High-level Design

5.2 Query Reader

This module reads the query file and passes it on to the Triple Generator. The query file can be any pre-decided format. For example, the following XML snippet shows an example of a query file we use for testing our system.

```
<DiscoveryRoutine name="discovery1">
  <Provided>
    StreetAddress,CityAddress,StateAddress,ZipAddress
  </Provided>
  <Resultant>
    hotelName,hotelID
  </Resultant>
</DiscoveryRoutine>
```

In the above snippet the *Provided* tag holds the list of input requirements and the *Resultant* tag holds the list of output requirements.

5.3 Semantic Relations Generator

For the semantic approach, matching is done based on the semantic relations between the parameters, conditions/constraints if provided and side-effects if provided. We obtain the semantic relations from the OWL WordNet ontology. OWL WordNet ontology provides a number of useful semantic relations like synonyms, antonyms, hyponyms, hypernyms, meronyms, holonyms and many more. USDL descriptions point to OWL WordNet for the meanings of concepts. A theory of service substitution is described in detail in [2] which uses the semantic relations between basic concepts of WordNet, to derive the semantic relations between services. This module extracts all the semantic relations and creates a list of Prolog facts.

5.4 Discovery Query Processor

This module compares the discovery query with all the services in the repository. The processor works as follows:

1. On the output parts of a service, the processor first looks for an *exact* substitutable. If it does not find one, then it looks for a parameter with hyponym relation [2], i.e., a *specific* substitutable.
2. On the input parts of a service, the processor first looks for an *exact* substitutable. If it does not find one, then it looks for a parameter with hypernym relation [2], i.e., a *generic* substitutable.

The discovery engine, written using Prolog with CLP(FD) library, uses a repository of facts, which contains a list of all the services, their input and output parameters and the semantic relations between the parameters. The following is the code snippet of our discovery engine:

```
discovery(sol(Qname,A)) :-
     dQuery(Qname,I,O), encodeParam(O,OL),
     /* Narrow candidate services(S) using output list(OL)*/
     narrowO(OL,S), fd_set(S,FDs), fdset_to_list(FDs,SL),
     /* Expand InputList(I) using semantic relations */
     getExtInpList(I, ExtInpList), encodeParam(ExtInpList,IL),
```

```
/* Narrow candidate services(SL) using input list (IL)*/
narrowI(IL,SL,SA), decodeS(SA,A).
```

The query is converted into a Prolog query that looks as follows:
discovery(sol(queryService, ListOfSolutionServices).
The engine will try to find a list of *SolutionServices* that match the *queryService*.

5.5 Composition Query Processor

For service composition, the first step is finding the set of composable services. If a subservice $\mathcal{S}_1$ is composed with subservice $\mathcal{S}_2$, then the output parts of $\mathcal{S}_1$ must be the input parts of $\mathcal{S}_2$. Thus the processor has to find a set of services such that the outputs of the first service are inputs to the next service and so on. These services are then stitched together to produce the desired service.

Similar to the discovery engine, composition engine is also written using Prolog with CLP(FD) library. It uses a repository of facts, which contains list of services, their input and output parameters and the semantic relations between the parameters. The following is the code snippet of our composition engine:

```
composition(Qname, A) :-
       dQuery(Qname,QI,QO), encodeParam(QO,OL),
       narrowO(OL,SL), fd_set(SL,Sset), fdset_member(S_Index,Sset),
       getExtInpList(QI,InpList), encodeParam(InpList,IL),
       list_to_fdset(IL,QIset), serv(S_Index,SI,_),
       list_to_fdset(SI,SIset), fdset_subtract(SIset,QIset,Iset),
       comp(QIset,Iset,[S_Index],SA), decodeS(SA,A).

comp(_, Iset, A, A) :- empty_fdset(Iset),!.
comp(QIset, Iset, A, SA) :-
       fdset_to_list(Iset,OL), narrowO(OL,SL), fd_set(SL,Sset),
       fdset_member(SO_Index,Sset), serv(SO_Index,SI,_),
       list_to_fdset(SI,SIset), fdset_subtract(SIset,QIset,DIset),
       comp(QIset,DIset,[SO_Index|A],SA).
```

The query is converted into a Prolog query that looks as follows:
composition(queryService, ListOfServices).
The engine will try to find a *ListOfServices* that can be composed into the requested *queryService*. Our composition engine uses the built-in, higher order predicate 'bagof' to return all possible *ListOfServices* that can be composed to get the requested *queryService*.

5.6 Output Generator

After the Discovery/Composition Query processor finds a matching service, or the list of atomic services for a composed service, the results are sent to the output generator in the form of triples. This module generates the output files in any desired XML format.

6 Efficiency and Scalability Issues

In this section we discuss the salient features of our system with respect to the efficiency and scalability issues related to the Web service discovery and composition problem. It is because of these features, we decided on the Multi-step narrowing based approach to solving these problems and implemented it using Constraint Logic Programming.

Pre-processing: Our system initially pre-processes the repository and converts all service descriptions into Prolog terms. In case of semantic approach, the semantic relations are also processed and loaded as Prolog terms in memory. Once the pre-processing is done, then discovery or composition queries are run against all these Prolog terms and hence we obtain results quickly and efficiently. The built-in indexing scheme and constraints in CLP(FD) facilitate the fast execution of queries. During the pre-processing phase, we use the term representations of services to set up constraints on services and the individual input and output parameters. This further helped us in getting optimized results.

Execution Efficiency: The use of CLP(FD) helped significantly in rapidly obtaining answers to the discovery and composition queries. We tabulated processing times for different size repositories and the results are shown in Section 7. As one can see, after pre-processing the repository, our system is quite efficient in processing the query. The query execution time is insignificant.

Programming Efficiency: The use of Constraint Logic Programming helped us in coming up with a simple and elegant code. We used a number of built-in features such as indexing, set operations, and constraints and hence did not have to spend time coding these ourselves. This made our approach efficient in terms of programming time as well. Not only the whole system is about 200 lines of code, but we also managed to develop it in less than 2 weeks.

Scalability: Our system allows for incremental updates on the repository, i.e., once the pre-processing of a repository is done, adding a new service or updating an existing one will not need re-execution of the entire pre-processing phase. Instead we can easily update the existing list of CLP(FD) terms loaded in the memory and run discovery and composition queries. Our estimate is that this update time will be negligible, perhaps a few milliseconds. With real-world services, it is likely that new services will get added often or updates might be made on existing services. In such a case, avoiding repeated pre-processing of the entire repository will definitely be needed and incremental update will be of great practical use. The efficiency of the incremental update operation makes our system highly scalable.

Use of external Database: In case the repository grow extremely large in size, then saving off results from the pre-processing phase into some external database might be useful. This is part of our future work. With extremely large repositories, holding all the results of pre-processing in the main memory may not be feasible. In such a case we can query a database where all the information is stored. Applying incremental updates to the database will be easily possible thus avoiding recomputation of the pre-processed data.

Searching for Optimal Solution: If there are any properties with respect to which the solutions can be ranked, then setting up global constraints to get the optimal solution is relatively easy with the constraint based approach. For example, if each service has an associated cost, then the discovery and the composition problem can be redefined to find the solutions with the minimal cost. Our system can be easily extended to take these global constraints into account.

7 Performance

We evaluated our approach on different size repositories and tabulated the Preprocessing time and the Query Execution time. We noticed that there was a significant difference in the pre-processing time between the first and the subsequent runs (after deleting all the previous pre-processed data) on the same repository. What we found is that the repository was cached after the first run and that explained the difference in the pre-processing time for the subsequent runs. We used repositories from the WS-Challenge web site [16].

Table 1 shows performance results for our Discovery Algorithm and table 2 shows results for Composition. The times shown in the tables are the wall clock times. The actual CPU time to pre-process the repository and execute the query should be less than or equal to the wall clock time. The results are plotted in figure 6 and 7 respectively. The graphs exhibit behavior consistent with our expectations: for a fixed repository size, the preprocessing time increases with the increase in number of input/output parameters. Similarly, for fixed input/output sizes, the preprocessing time is directly proportional to the size of the service repository. However, what is surprising is the efficiency of service query processing, which is negligible (just 1 to 3 milliseconds) even for complex queries with large service repositories.

Repository Size (number of services)	*Number of I/O parameters*	*Non-Cached Pre-processing Time (in secs)*	*Cached Pre-processing Time (in secs)*	*Query Execution Time (in msecs)*
2000	4-8	36.5	7.3	1
2000	16-20	45.8	13.4	1
2000	32-36	57.8	23.3	2
2500	4-8	47.7	8.7	1
2500	16-20	58.7	16.6	1
2500	32-36	71.6	29.2	2
3000	4-8	56.8	12.1	1
3000	16-20	77.1	19.4	1
3000	32-36	88.2	33.7	3

Table 1. Performance of our Discovery Algorithm

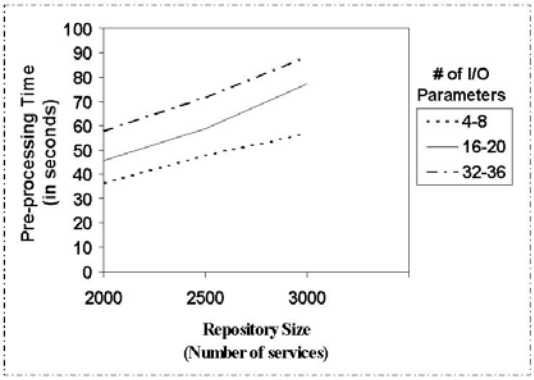


Fig. 6. Performance of Discovery Algorithm

Repository Size (number of services)	*Number of I/O parameters*	*Non-Cached Pre-processing Time (in secs)*	*Cached Pre-processing Time (in secs)*	*Query Execution Time (in msecs)*
2000	4-8	36.1	7.2	1
2000	16-20	47.1	15.1	1
2000	32-36	60.2	24.7	1
3000	4-8	58.4	11.0	1
3000	16-20	60.1	17.8	1
3000	32-36	102.0	42.1	1
4000	4-8	71.2	13.4	1
4000	16-20	87.9	25.3	1
4000	32-36	129.2	43.1	1

Table 2. Performance of our Composition Algorithm

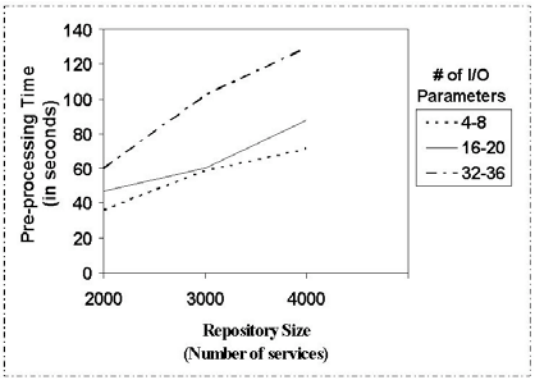


Fig. 7. Performance of Composition Algorithm

8 Conclusion

To catalogue, search and compose Web services in a semi-automatic to fully-automatic manner we need infrastructure to publish Web services, document them and query repositories for matching services. Our syntactic approach uses WSDL descriptions and applies the discovery and composition routines on first-order logic terms obtained from these descriptions. Our semantic approach uses USDL to formally document the semantics of Web services and our discovery and composition engines find substitutable and composite services that best match the desired service.

Our solution produces accurate and quick results with both syntactic and semantic description of Web services. We are able to apply many optimization techniques to our system so that it works efficiently even on large repositories. Use of Constraint Logic Programming helped greatly in obtaining an efficient implementation of this system.

References

1. A. Bansal, S. Kona, L. Simon, A. Mallya, G. Gupta, and T. Hite. A Universal Service-Semantics Description Language. In *European Conference On Web Services*, pp. 214-225, 2005.
2. S. Kona, A. Bansal, L. Simon, A. Mallya, G. Gupta, and T. Hite. USDL: A Service-Semantics Description Language for Automatic Service Discovery and Composition. Tech. Report UTDCS-18-06. `www.utdallas.edu/~sxk038200/USDL.pdf`.
3. S. McIlraith, T.C. Son, H. Zeng. Semantic Web Services. In IEEE Intelligent Systems Vol. 16, Issue 2, pp. 46-53, Mar. 2001.
4. OWL-S: Semantic markup for Web services. `www.daml.org/services/owl-s/1.0/owl-s.html`.
5. WSML: Web Service Modeling Language. `www.wsmo.org/wsml/`.
6. WSDL-S: Web Service Semantics. `http://www.w3.org/Submission/WSDL-S`.
7. WSDL: Web Services Description Language. `http://www.w3.org/TR/wsdl`.
8. A. Bansal, K. Patel, G. Gupta, B. Raghavachari, E. D. Harris, and J. C. Staves. Towards Intelligent Services: A case study in chemical emergency response. In *International Conference on Web Services*, pp. 751-758, 2005.
9. SAP Interface Repository. `http://ifr.sap.com/catalog/query.asp`.
10. WordNet: A Lexical Database for the English Language. `http://www.cogsci.princeton.edu/~wn`.
11. OWL WordNet: Ontology-based information management system. `http://taurus.unine.ch/knowler/wordnet.html`.
12. S. Kona, A. Bansal, G. Gupta, and T. Hite. USDL - Formal Definitions in OWL. Internal Report, University of Texas, Dallas, 2006. Available at `http://www.utdallas.edu/~srividya.kona/USDLFormalDefinitions.pdf`.
13. K. Marriott and P. J. Stuckey. Programming with Constraints: An Introduction. *MIT Press, 1998.*
14. L. Sterling and S. Shapiro. The Art of Prolog. *MIT Press, 1994.*
15. OWL: Web Ontology Language Reference. `http://www.w3.org/TR/owl-ref`.
16. WS Challenge 2006. `http://insel.flp.cs.tu-berlin.de/wsc06`.
17. U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel. Automatic Location of Services. In *European Semantic Web Conference*, May 2005.

Policy-based reasoning for smart web service interaction*

Marco Alberti[1], Federico Chesani[2], Marco Gavanelli[1], Evelina Lamma[1], Paola Mello[2], Marco Montali[2], and Paolo Torroni[2]

[1] ENDIF, Università di Ferrara
{marco.alberti|marco.gavanelli|evelina.lamma}@unife.it
[2] DEIS, Università di Bologna
{fchesani|pmello|mmontali|ptorroni}@deis.unibo.it

Abstract. We present a vision of smart, goal-oriented web services that reason about other services' policies and evaluate the possibility of future interactions. To achieve our vision, we propose a proof theoretic approach. We assume web services whose interface behaviour is specified in terms of reactive rules. Such rules can be made public, in order for other web services to answer the following question: "is it possible to inter-operate with a given web service and achieve a given goal?" In this article we focus on the underlying reasoning process, and we propose a declarative and operational abductive logic programming-based framework, called WAV^e.

1 Introduction

Service Oriented Computing (SOC) is rapidly emerging as a new programming paradigm, propelled by the wide availability of network infrastructures, such as the Internet, and by the success of its predecessor, Object Oriented programming paradigm. Web service-based technologies are an implementation of SOC, aimed at overcoming the intrinsic difficulties of integrating different platforms, operating systems, languages, etc., into new applications. It is then in the spirit of SOC to take off-the-shelf solutions, like web services, and compose them into new applications. Service composition is very attractive for its support to rapid prototyping and possibility to create complex applications from simple elements. It is the philosophy followed, e.g., by BPEL [1]: composing new applications through existing web services.

On the upside, the recent popularity of these new technologies developed into a growing presence of web services, made available through the Internet, and we can foresee a steady increase of such services also for the near future. On the downside, the lifetime of software developed with the classical methodologies of composition of existing services at design-time gets shorter and shorter. It quickly

* We thank the anonymous referees for their valuable feedback and pointers. This work has been partially supported by the MIUR PRIN 2005 project *Specification and verification of agent interaction protocols*.

becomes a suboptimal choice, blind to the exploitation of new opportunities. In highly competitive markets, this can be a severe drawback.

If we adopt the SOC programming paradigm, how to exploit the potential of a growing base of web services becomes one of our strategic issue. In a domain in which being more competitive means knowing more and using all available information at best, how shall we cope with the proliferation of new services? How shall we decide to use a web service rather than another one? when new ones becomes available, shall we go for them? are there new opportunities that were not there before? It is a necessary, never-ending, heavy and thus potentially very costly decision process, but it could also be very rewarding, if we had the proper tools.

A partial answer to these questions is given by service discovery. As new services become available, they are published, for instance by registration on some yellow-pages server; existing services can then become aware of the new ones and exploit them. This solves part of the problem: as through discovery we only know that there are some some services, which possibly follow some standards, but understanding whether interacting with them will be profitable or detrimental, is far from being a trivial question. For one, it is not possible to think to try and invoke all newly discovered services and analyze the results. Beside being highly error-prone, such a method would require expensive rollbacks that are often unaffordable at run-time. Thus, alternative approaches have to be developed. This is what we intend to address in this article.

The focus of this article is the following problem: how to dynamically understand if two web services can inter-operate, without them having a-priori knowledge of each other's capabilities, but by reasoning about policies exchanged at run-time.

We present a vision of smart, goal-oriented web services that reason about other services' specifications, with the aim to separate out those that can lead to a fruitful interaction, without resorting to trial and error. We envisage a two-phase discovery activity on the side of web services. First, web services collect information about other web services, and try and understand by reasoning which ones can lead to a fruitful interaction. This activity is carried out off-line, beforehand. Then they use the available information to interact with each other. It is the same philosophy of search engines: before, collect information through web spiders, then use it when requested by the user.

In this article we focus on the reasoning involved in the off-line phase, assuming that a new web service has been found, and we must decide about the possibility to interact with it. We assume that each web service publishes, alongside with its WSDL, its *interface behaviour specifications*. By reasoning on the information available about other web services' interface behaviour, each web service can verify which goals can be reached by interaction.

To achieve our vision, we propose a proof theoretic approach, based on computational logic – in fact, on abductive logic programming. In particular, we formalise policies for web services in a declarative language which is a modification of the $\mathcal{S}$CIFF language originally defined in the context of the UE IST-2001-

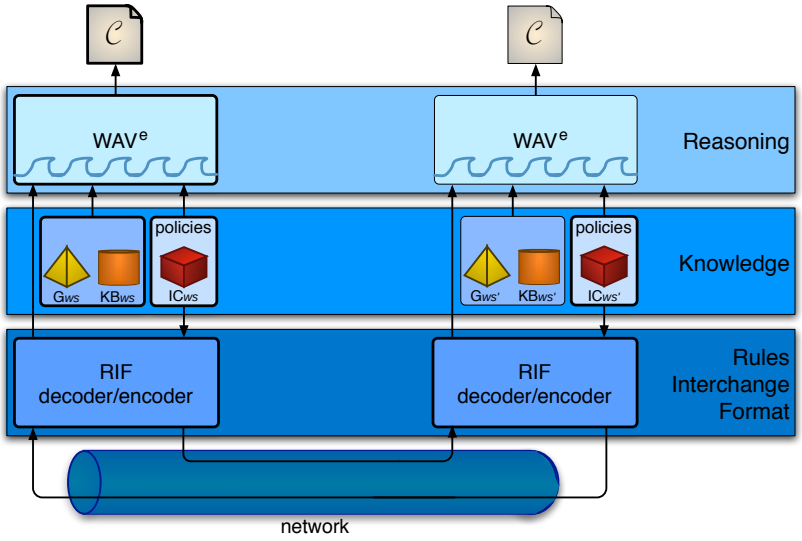


Fig. 1. The architecture of WAV^e

32530 project, to specify and verify social-level agent interaction. In this new language, policies can be defined by way of *social integrity constraints* (*IC*s): a sort of reactive rules used to generate and reason about expectations about possible evolutions of a given interaction setting. Based on the $\mathcal{S}$CIFF framework we propose a new declarative semantics and a new proof-procedure that combines forward, reactive reasoning with backward, goal-oriented reasoning, and is tailored to the discovery activity's off-line phase's verification problem. We have called this new framework WAV^e(Web-service Abductive Verification).

We start by showing the abstract architecture of WAV^e. In Sec. 3 we introduce a running on-line shopping scenario. In Sect. 4, we briefly introduce the language used in the framework, and in Sect. 5 we show how the scenario can be modeled in WAV^e in terms of *ICs*. Sect. 6 presents the declarative and operational semantics of WAV^e, and Sect. 7 proposes the application of WAV^e to the verification problem in the reference scenario. A brief discussion follows.

2 The Architecture of WAV^e

Fig. 1 depicts our general reference architecture. Arrows indicate the flow of policies between web services. The layered architecture of a web service, e.g. ws, has WAV^e at the top of the stack, performing reasoning based on its own knowledge and on the policies obtained from other web services, e.g. ws'. The functionalities of the various elements of the knowledge will be explained in Sect. 4. For the moment, we say that policies are identified with the $\mathcal{IC}_{ws}$ component. The architecture is symmetric. We represented with thick borders the modules involved in the operations carried out by ws, and its output. In order for ws' to pass

$\mathcal{IC}_{ws'}$ on to ws (and vice versa), a Rule Interchange Format (RIF) is adopted. One possibility for such a RIF could be RuleML [2]. Finally, as a result of the reasoning activity, ws produces an answer $\mathcal{C}$ to the question: "is it possible to inter-operate with ws' and achieve goal $\mathcal{G}_{ws}$?"

Fig. 1 does not show control elements, but only information flows. We assume that suitable interaction protocols are defined to control the flow of information (e.g. policies) between the web services. In particular, in a more comprehensive setting, ws and ws' could negotiate the exchange of policies in an incremental way, or could use the result $\mathcal{C}$ of this reasoning activity to perform the second, on-line phase of service interaction we mentioned in the introduction. All this is outside of this picture, and of this article's scope.

3 The *alice* & *eShop* Scenario

This scenario is inspired to the one described by the Working Group on Rule Interchange Format [3]. A similar scenario is also in [4]. We consider two entities, which we call *alice* and *eShop*.[3] *eShop* is a web service which sells devices. *alice* is another web service which instead needs to obtain a device, and which is considering buying it from *eShop*. *alice* and *eShop* describe their behaviour concerning sales/payment/... of items through policies, specified as rules, which they publish using some RIF.

Before *alice* buys any item from *eShop*, *alice* checks whether her policies and *eShop*'s policies are compatible, i.e., if they allow a successful transaction regarding the sales. During this process, it turns out that *eShop* accepts credit card payments, besides other payment methods, and that *alice* can only pay by credit card; in this case, in order to proceed with the payment, she requires evidence of the shop's membership to some trusted "Better Business Bureau" ($\mathcal{BBB}$) association. We assume that the shop is able and ready to provide such a piece of evidence. We can thus define *eShop*'s and *alice*'s policies as follows:

(shop1) if a customer wishes to buy an item, then (s)he should pay it either by credit card, or by cash, or by cheque;
(shop2) if a customer wishes to buy an item, and (s)he has paid it either by credit card, or by cash, or by cheque, then *eShop* will deliver the item;
(shop3) if a customer wishes to receive a certificate about *eShop*'s membership to the $\mathcal{BBB}$, then the shop will send it;
(alice1) if a shop requires that *alice* pays by credit card, *alice* expects that the shop provides evidence of its membership to the $\mathcal{BBB}$;
(alice2) if a shop requires that *alice* pays by credit card, and the shop has provided evidence of its membership to the $\mathcal{BBB}$, then *alice* will pay by credit card;

In this example, we can identify two kinds of policy rules. shop1 and alice1 express requirements, i.e., what is needed in order to proceed with accomplishing

[3] In this simplified scenario, we identify *alice* and *eShop* with their representative software counterparts which will carry out transactions on their behalf.

some request. shop2, shop3 and alice2 represent the effect of requests, i.e., they tell what has to be expected if some conditions hold and some request is received.

Using this scenario, we want to demonstrate the possibility of reaching an agreement through rules exchange. Besides, we want to show how policies support backward and forward reasoning, in the following way. Backward, pro-active reasoning starts from goals to produce (expectations about) actions or events that should be generated in order to achieve the goals. Forward, reactive reasoning starts from events and is used to generate (expectations about) actions that represent reactions to such events.

In this scenario, the goal of *alice* interacting with *eShop* is to obtain an item from *eShop*. Actions are all the messages exchanged between the two web services.

The steps that we envisage are as follows:

1. *alice* wants to obtain a device. She knows that she can have it if *eShop* delivers it to her. Thus, she sends *eShop* a request, by which she wants to know *eShop*'s policies regarding the delivery of that device;
2. *eShop* considers *alice*'s request, and composes a set of rules related to *alice*'s request (its policies), possibly deriving/filtering them from a larger set. In this example, the set contains shop1, shop2, and shop3. Once such a set is put together, *eShop* communicates it to *alice*;
3. *alice* reasons on (1) her goal, (2) her own policies (alice1 and alice2), and (3) *eShop*'s policies. Two are the possible outcomes:
   - either *alice* infers that she and *eShop* can have a successful transaction that satisfies each other's policies and that achieves her goal,
   - or *alice* infers that there is no such a possibility.
4. possibly, at a later point, *alice* and *eShop* may engage in a transaction which (hopefully) makes *alice* achieve her goal.

Points (1) through (3) represent the off-line phase of service discovery/interaction, whereas point (4) represent the actual transaction occurring between *alice* and *eShop*. The reasoning involved in (3) is the subject of this article.

4 The WAV^e Framework

In WAV^e, the observable behaviour of the web services is represented by *events*. Since we focus on (explicit) interaction between web services, events always represent exchanged messages.

WAV^e considers two types of events: those that one can control and those that one cannot. Typically, from the standpoint of a web service ws, an event such as a message generated by ws himself will fall into the first category, a message that ws is expecting from another fellow web service ws' will fall instead into the second one. We use two different functors to keep these two categories of messages distinct from each other. Atoms denoted by functor **H** will stand for events that a web service expects to be producing itself; atoms denoted by functor **E** will stand for events that a web service is expecting, and over which

it does not have any control. Since WAV^e is about reasoning on possible future courses of events, both kinds of events represent *hypotheses* that a web service can make on possibly happening events. The notation is: $\mathbf{H}(ws, ws', M, T)$, for messages ($M$) that a web service ws is expecting to send to ws' at time T, and $\mathbf{E}(ws', ws, M, T)$ for messages (M) expected by ws from ws' for time T.

Web service specifications in WAV^e are relations among expected events, expressed by an Abductive Logic Program (ALP). In general, an ALP [5] is a triplet $\langle P, A, IC\rangle$, where P is a logic program, A is a set of predicates named *abducibles*, and IC is a set of integrity constraints. Roughly speaking, the role of P is to define predicates, the role of A is to fill-in the parts of P which are unknown, and the role of IC is to constrain the ways elements of A are hypothesised, or "abduced". Reasoning in abductive logic programming is usually goal-directed (being G a goal), and it accounts to finding a set of abduced hypotheses Δ built from predicates in A such that $P \cup \Delta \models G$ and $P \cup \Delta \models IC$. In the past, a number of proof-procedures have been proposed to compute Δ (see Kakas and Mancarella [6], Fung and Kowalski [7], Denecker and De Schreye [8], etc.).

Definition 1 (Web service interface behaviour specification). *Given a web service ws, its* web service interface behaviour *specification $\mathcal{P}_{ws}$ is an ALP, represented by the triplet*

$$\mathcal{P}_{ws} \equiv \langle \mathcal{KB}_{ws}, \mathcal{E}_{ws}, \mathcal{IC}_{ws}\rangle$$

where:

- $\mathcal{KB}_{ws}$ *is ws's Knowledge Base,*
- $\mathcal{E}_{ws}$ *is ws's set of* abducible predicates*, and*
- $\mathcal{IC}_{ws}$ *is ws's set of Integrity Constraints.*

$\mathcal{KB}_{ws}$ is a set of clauses which declaratively specifies pieces of knowledge of the web service. Note that the body of $\mathcal{KB}_{ws}$'s clauses may contain $\mathbf{E}$ expectations about the behaviour of the web services, as defined above. $\mathcal{KB}_{ws}$'s syntax is summarised in Eq. (1).

$$\begin{aligned} \mathcal{KB}_{ws} &::= [\ Clause\ ]^\star \\ Clause &::= Atom \leftarrow Cond \\ Cond &::= ExtLiteral\ [\ \wedge ExtLiteral\ ]^\star \\ ExtLiteral &::= Atom \mid true \mid Expect \mid Constr \\ Expect &::= \mathbf{E}(Atom, Atom, Atom, Atom) \end{aligned} \tag{1}$$

$\mathcal{E}_{ws}$ includes $\mathbf{E}$ expectations, $\mathbf{H}$ events, and predicates not defined in $\mathcal{KB}_{ws}$.

$$\begin{aligned} \mathcal{IC}_{ws} &::= [\ IC\ ]^\star \\ IC &::= Body \rightarrow Head \\ Body &::= (Event \mid Expect)\ [\wedge BodyLit]^\star \\ BodyLit &::= Event \mid Expect \mid Atom \mid Constr \\ Head &::= Disjunct\ [\ \vee Disjunct\ ]^\star \mid false \\ Disjunct &::= (Expect \mid Event \mid Constr)[\ \wedge (Expect \mid Event \mid Constr)]^\star \\ Expect &::= \mathbf{E}(Atom, Atom, Atom, Atom) \\ Event &::= \mathbf{H}(Atom, Atom, Atom, Atom) \end{aligned} \tag{2}$$

Integrity Constraints (*ICs*) are forward rules, of the form *Body* $\rightarrow$ *Head* (Eq. (2)). The *Body* of *ICs* is a conjunction of literals and expected events; the *Head* instead is a disjunction of conjunctions of expectations, events and literals, or *false*. The syntax of $\mathcal{IC}_{ws}$ is a modification of that defined in [9]. In particular, unlike $\mathcal{S}$CIFF, WAV^e treats **H** events as abducible predicates, and as such it allows them to occur in the *Head* of integrity constraints; however, this initial version of WAV^e does not yet accommodate negative expectations nor negation ($\neg$). We intend to consider these two features in future extensions of WAV^e.

Intuitively, the operational behaviour of integrity constraints is similar to forward rules: whenever the body becomes true, the head is also made true.

5 Modeling in WAV^e

In this section, we demonstrate web service policy modelling in WAV^e by showing the specification of *alice* and *eShop*. The first three rules represent *eShop*'s policies.

$$\begin{aligned}&\mathbf{E}(eShop, alice, deliver(Item), T_s)\\ \rightarrow&\mathbf{E}(alice, eShop, pay(Item, cc), T_{cc}) \wedge T_{cc} < T_s\\ \vee&\mathbf{E}(alice, eShop, pay(Item, cash), T_{ca}) \wedge T_{ca} < T_s\\ \vee&\mathbf{E}(alice, eShop, pay(Item, cheque), T_{ch}) \wedge T_{ch} < T_s\end{aligned} \quad \text{(shop1)}$$

IC shop1 says that, if *alice* expects *eShop* to deliver an Item, then *eShop* expects *alice* to pay by credit card, cash, or cheque, and that payment must be made before delivery.[4] In that case, the abducibles in the head are expectations, because they represent actions that should be performed by *alice*: from *eShop*'s viewpoint, they can only be expected.

$$\begin{aligned}&\mathbf{E}(eShop, alice, deliver(Item), T_s)\\ &\wedge \mathbf{H}(alice, eShop, pay(Item, How), T_p) \wedge T_p < T_s\\ &\wedge How{::}[cc, cash, cheque])\\ \rightarrow&\mathbf{H}(eShop, alice, deliver(Item), T_s).\end{aligned} \quad \text{(shop2)}$$

IC shop2 says that, if *alice* expects *eShop* to deliver the Item, and *alice* has paid for it, then *eShop* will actually deliver it to *alice*. In that case, the abducible in the head is an event, because it represents an action that *eShop* should perform, and therefore it assumes that it will indeed happen (since it is its own responsibility).

$$\begin{aligned}&\mathbf{E}(eShop, alice, give\_guarantee, T_g)\\ \rightarrow&\mathbf{H}(eShop, alice, give\_guarantee, T_g).\end{aligned} \quad \text{(shop3)}$$

IC shop3 says that if *alice* expects to receive a guarantee, then *eShop* will send it. The following two rules represent *alice*'s policies.

$$\begin{aligned}&\mathbf{E}(alice, eShop, pay(Item, cc), T_p)\\ \rightarrow&\mathbf{E}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_p.\end{aligned} \quad \text{(alice1)}$$

[4] The alternative in the head could alternatively be expressed via a variable with domain: $\mathbf{E}(alice, eShop, pay(Item, How), T) \wedge How{::}[cc, cash, cheque] \wedge T < T_s$, where "::" represents a domain constraint.

IC alice1 says that, if *eShop* expects *alice* to pay for an Item by credit card, then *alice* expects that *eShop* will have provided a guarantee by the time she pays.

$$\begin{aligned}&\mathbf{E}(alice, eShop, pay(Item, cc), T_p)\\ &\wedge \mathbf{H}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_p\\ \rightarrow &\mathbf{H}(alice, eShop, pay(Item, cc), T_p).\end{aligned} \qquad \text{(alice2)}$$

IC alice2 says that, if *eShop* expects *alice* to pay for an Item by credit card, and *eShop* has provided *alice* with a guarantee, then *alice* will pay the Item by credit card. Finally, the following clause is part of $\mathcal{KB}_{alice}$

$$\begin{aligned}&have(alice, Item, T) \leftarrow\\ &\mathbf{E}(eShop, alice, deliver(Item), T_d) \wedge T_d \leq T.\end{aligned} \qquad \text{(alice3)}$$

Clause alice3 says that, in order for *alice* to have an Item at time T, then *alice* expects *eShop* to deliver the Item by time T.

6 Declarative and Operational Semantics

We have assumed that all web services have their own interface behaviour specified in the language of $\mathcal{IC}$s. This behaviour could be thought of as an extension of WSDL, that could be used by other fellow web services to reason about the specifications, or to check if inter-operability is possible. We are currently studying an XML-like extension of RuleML [2] to represent $\mathcal{IC}$s.

Another approach would be to obtain web services' interface behaviour through an appropriate request protocol, in which $\mathcal{IC}$s are (interactively) exchanged so that each web service may disclose *ad hoc*, customised information on demand.

In this work, we make the simplifying assumption that all information regarding the interface behaviour is provided at once. The web service will then try and prove that a fruitful interaction is possible based on what it receives.

The web service initiating the interaction has a goal $\mathcal{G}$, which is a given state of affairs. A typical goal could be to access a resource, to retrieve some information, or to obtain a service from another web service. $\mathcal{G}$ will often be an expectation (of obtaining a service, accessing a resource, or gathering information), but in general it can be any conjunction of expectations, CLP constraints, and any other literals, in the syntax of $\mathcal{IC}_{ws}$ *Head Disjunct*s (Eq. 2).

The verification of a web service ws about the possibility to achieve a goal $\mathcal{G}$ by interacting with another fellow web service ws' makes use of $\mathcal{KB}_{ws}$, $\mathcal{IC}_{ws}$, $\mathcal{G}$, and of the information obtained about ws''s policies, $\mathcal{IC}_{ws'}$ (see Fig. 1). The idea is to obtain, through abductive reasoning, a set of expectations about a possible course of events that together with $\mathcal{KB}_{ws}$ entails $\mathcal{IC}_{ws} \cup \mathcal{IC}_{ws'}$ and $\mathcal{G}$.

Note that we do not assume that ws knows $\mathcal{KB}_{ws'}$, as the $\mathcal{KB}$ is not part of the interface. However, in general integrity constraints can involve predicates defined in the knowledge base. For example, they can contain predicates defining parameters, deadlines, coefficients, etc., or other knowledge only available to ws'. If the interface behaviour provided by ws' involves predicates defined in $\mathcal{KB}_{ws'}$, unknown to ws, we have two alternatives:

– either ws' provides ws with the necessary information, e.g. with (part of) its $\mathcal{KB}_{ws'}$;
– or ws will have to make assumptions about such unknown predicates.

We take the second option, and consider unknowns that are neither **H** events nor **E** expectations as literals that can be abduced, and we keep them in a set Δ. We then have the following two equations that define the set of abductive answers representing possible interaction between ws and ws' achieving $\mathcal{G}$:

$$\mathcal{KB}_{ws} \cup \mathbf{HAP} \cup \mathbf{EXP} \cup \Delta \models \mathcal{G} \tag{3}$$

$$\mathcal{KB}_{ws} \cup \mathbf{HAP} \cup \mathbf{EXP} \cup \Delta \models \mathcal{IC}_{ws} \cup \mathcal{IC}_{ws'}. \tag{4}$$

where **HAP** is a conjunction of **H** atoms, **EXP** is a conjunction of **E** atoms, and Δ a conjunction of abducible atoms.

We ground the notion of entailment on a model theoretic semantics defined for Abductive Disjunctive Logic Programs [10]. Different semantics have been proposed for logic programs with disjunctions. Among them, answer set semantics [11] adopts an exclusive interpretation of disjunction, whereas possible model semantics [10] adopts an inclusive one (and recovers the former by additional constraints imposing mutual exclusion among the literals in the disjunctive head of a clause).

In the possible model semantics, the disjunctive program generates several (non-disjunctive) split programs, obtained by separating the disjuncts in the head of rules. Given a disjunctive logic program P, a *split program* is defined as a (ground) logic program obtained from P by replacing every (ground) rule

$$r : L_1 \vee \cdots \vee L_l \leftarrow \Gamma$$

from P with the rules in a non-empty subset of $Split_r$, where

$$Split_r = \{L_i \leftarrow \Gamma \mid i = 1, \ldots, l\}.$$

By definition, P has multiple split programs in general.

Example 1. The following program can be split into three split programs

$$\begin{array}{rcl} \mathbf{E}(p) \vee \mathbf{H}(p) & \leftarrow & \mathbf{E}(p). \\ goal & \leftarrow & \mathbf{E}(p). \end{array}$$

where the first clause is respectively substituted by $\{\mathbf{E}(p) \leftarrow \mathbf{E}(p)\}$, $\{\mathbf{H}(p) \leftarrow \mathbf{E}(p)\}$ and $\{\mathbf{E}(p) \leftarrow \mathbf{E}(p), \mathbf{H}(p) \leftarrow \mathbf{E}(p)\}$.

A *possible model* for a disjunctive logic program P is then defined as an answer set of a split program of P.

The inclusive interpretation of disjunctions adopted by the possible model semantics fits better with our case, since more than one disjunct in the head of an integrity constraint can be true at the same time, as in the following example.[5]

[5] For the sake of simplicity, in this example and in the following one, we specify a single argument for expectations and events.

Example 2. Let us consider the program:

$$\begin{array}{rcl} \mathbf{E}(p) \vee \mathbf{H}(p) & \leftarrow & \mathit{true}. \\ \mathbf{H}(p) & \leftarrow & \mathbf{E}(p). \\ \mathit{goal} & \leftarrow & \mathbf{E}(p). \end{array}$$

We would like to have an explanation for *goal*, where $\mathbf{E}(p)$ is assumed, and $\mathbf{H}(p)$ is also true because of clause $\mathbf{H}(p) \leftarrow \mathbf{E}(p)$. This is, instead, avoided by following an answer set approach, which adopts an exclusive interpretation of disjunctions.

Furthermore, in [10] possible model semantics was also applied to provide a model theoretic semantics for Abductive Extended Disjunctive Logic Programs (AEDP), which is our case. Semantics is given to AEDP in terms of *possible belief sets*. Given an AEDP $\Pi = \langle P, \mathcal{A} \rangle$ (where P is a disjunctive logic program and $\mathcal{A}$ is the set of abducible literals), a possible belief set S of Π is a possible model of the disjunctive program $P \cup E$, where P is extended with a set E of abducible literals ($E \subseteq \mathcal{A}$).

Definition 2 (Answer to a goal $\mathcal{G}$). *An* answer E to a (ground) goal $\mathcal{G}$ *is a set E of abducible literals constituting the abductive portion of a possible belief set S (i.e., $E = S \cap \mathcal{A}$) that entails $\mathcal{G}$.*

We rely upon possible belief set semantics, but we adopt a new notion for minimality with respect to abducible literals. In [10], a possible belief set S is $\mathcal{A}$-minimal if there is no possible belief set T such that $T \cap \mathcal{A} \subset S \cap \mathcal{A}$.

Example 3. Consider, again, the program of Example 1. The possible belief sets are the belief sets of each of the split programs: the first gives $\{\mathit{goal}, \mathbf{E}(p)\}$, and the others give $\{\mathit{goal}, \mathbf{E}(p), \mathbf{H}(p)\}$. Only the first explanation is $\mathcal{A}$-minimal under set inclusion, according to [10], but we cannot rely upon such definition for minimality since we would discard explanations which are, instead, correct.

We restate, then, the notion of $\mathcal{A}$-minimality as follows.

Definition 3 ($\mathcal{A}$-minimal possible belief set). *A possible belief set S is* $\mathcal{A}$-minimal *iff there is no possible belief set T* for the same split program *such that $T \cap \mathcal{A} \subset S \cap \mathcal{A}$.*

More intuitively, the notion of minimality with respect to hypotheses that we introduce is checked by considering the *same* split program, and by checking whether with a smaller set of abducibles the same consequences can be made true, but in the same split program. For the case depicted in Example 3, then, both the two possible belief sets are $\mathcal{A}$-minimal, according to Definition 3.

Finally, we provide a model-theoretic notion of explanation to an observation, in terms of answer to a goal, as follows.

Definition 4 ($\mathcal{A}$-minimal answer to a goal). *E is an* $\mathcal{A}$-minimal answer to a goal $\mathcal{G}$ *iff $E = S \cap \mathcal{A}$ for some possible $\mathcal{A}$-minimal belief set S that entails $\mathcal{G}$.*

We can now proceed with defining what kind of interaction is possible/fruitful, given two web services and a goal.

Definition 5 (Possible interaction about $\mathcal{G}$). *A* possible interaction about a goal $\mathcal{G}$ *between two web services ws and ws' is an $\mathcal{A}$-minimal set* $\mathbf{HAP} \cup \mathbf{EXP} \cup \Delta$ *such that Eq. 3 and 4 hold.*

Among all possible interactions about $\mathcal{G}$, some of them are fruitful, and some are not. An interaction only based on expectations which will not be matched by corresponding events is not a fruitful one: for example, the goal of ws might not have a corresponding event, thus the goal is not actually reached, but only *expected*. Or, one of the web services could be waiting for a message from the other fellow, which will never arrive, thus undermining the inter-operability.

We select, among the possible interactions, those whose history satisfies all the expectations of both the web services. After the abductive phase, we have a verification phase in which there are no abducibles, and in which the previously abduced predicates **H** and **E** are now considered as defined by atoms in **HAP** and **EXP**, and they have to match. If among the possible interactions there exists one satisfying

$$\mathbf{HAP} \cup \mathbf{EXP} \models \mathbf{E}(X, Y, Action, T) \leftrightarrow \mathbf{H}(X, Y, Action, T) \tag{5}$$

then ws has found a sequence of actions that obtains the goal $\mathcal{G}$.

Definition 6 (Possible interaction achieving $\mathcal{G}$). *Given two web services, ws and ws', and a goal $\mathcal{G}$, a* possible interaction achieving $\mathcal{G}$ *is a possible interaction about $\mathcal{G}$ satisfying Eq. 5.*

Intuitively, the "$\rightarrow$" implication in Eq. 5 is there to avoid situations in which a web service waits forever for an event that the other web service will never produce. The "$\leftarrow$" implication is there to avoid that one web service sends unexpected messages, which in the best case may not be understood (and in the worst scenarios it may lead to faulty, unpredictable behaviour of the parties involved).

6.1 Operational Semantics

The operational semantics is a modification of the $\mathcal{S}$CIFF proof-procedure [12]. $\mathcal{S}$CIFF is a transition system, whose state is given by the following tuple:

$$T \equiv \langle R, CS, PSIC, \Delta A, \mathbf{PEND}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle$$

The set of expectations **EXP** is partitioned into the fulfilled (**FULF**), violating (**VIOL**), and pending (**PEND**) expectations. The other elements are: the resolvent (R), the abduced literals that are not expectations (ΔA), the constraint store (CS), a set of implications, inherited from the IFF [7], called *partially solved integrity constraints* ($PSIC$), and the history of happened events (**HAP**).

A classical application of $\mathcal{S}$CIFF is on-line checking of compliance of agent interaction to protocols. in fact, $\mathcal{S}$CIFF was initially developed to specify and verify agent interaction protocols on-the-fly, under the assumption of open agent environments adopted by other noteworthy agent research work [13]. $\mathcal{S}$CIFF processes events drawing from **HAP** and generates (abduces) expectations; then it checks that all expectation are fulfilled by at least one happened event. The declarative semantics of $\mathcal{S}$CIFF contains in fact a requirement $\mathbf{E}(X) \rightarrow \mathbf{H}(X)$ – differently from WAV^e, which has a double implication (Eq. 5). In $\mathcal{S}$CIFF, as soon as new **H** events are processed, a transition *fulfilment* labels the relevant matching expectations as *fulfilled* and moves them to the set **FULF**. At the end of the derivation, if some expectation remains in the set **PEND**, a failure node is generated, and other alternative branches will be explored in backtracking, if there exist any.

WAV^e extends $\mathcal{S}$CIFF and abduces **H** events as well as expectations. The events history is not taken as input, but all possible interactions are hypothesised. Moreover, in WAV^e events not matched by an expectation (which are perfectly acceptable in the multi-agent scenario addressed by $\mathcal{S}$CIFF) cannot be part of a *possible interaction achieving* the goal.

The two phases in the declarative semantics (generation of possible interactions and their test for conformance) are condensed into one single derivation process, thanks to a new transition adopted in WAV^e. The *expected* transition, symmetrical to *fulfilment*, labels each **H** events with an *expected* flag as soon as an expectation matching it is abduced. At the end of the derivation, **H** with *expected* status = false will cause failure.

Otherwise, if the WAV^e derivation in a program $\mathcal{P}$ for a goal $\mathcal{G}$ succeeds with set of expectation $\mathbf{EXP} \cup \mathbf{HAP} \cup \Delta$, we write $\mathcal{P} \vdash_{\mathbf{EXP} \cup \mathbf{HAP} \cup \Delta} \mathcal{G}$.

6.2 Soundness and completeness results

WAV^e is a conservative modification of the $\mathcal{S}$CIFF proof-procedure, which is sound and complete under reasonable assumptions [14]. In the following, we give the soundness and completeness statements, and briefly explain why the soundness and completeness results proven for $\mathcal{S}$CIFF also hold with the new declarative and operational semantics of WAV^e.

Theorem 1 (Soundness). *If* $\mathcal{P} \vdash_{\mathbf{EXP} \cup \mathbf{HAP} \cup \Delta} \mathcal{G}$ *then* $\mathbf{EXP} \cup \mathbf{HAP} \cup \Delta$ *is a possible interaction achieving* $\mathcal{G}$*.*

Theorem 2 (Completeness). *If there exists a possible interaction* $\mathbf{EXP} \cup \mathbf{HAP} \cup \Delta$ *achieving a goal* $\mathcal{G}$*, then* $\exists \mathbf{EXP}' \cup \mathbf{HAP}' \cup \Delta' \subseteq \mathbf{EXP} \cup \mathbf{HAP} \cup \Delta$ *such that* $\mathcal{P} \vdash_{\mathbf{EXP} \cup \mathbf{HAP} \cup \Delta} \mathcal{G}$*.*

Note that WAV^e introduces two main additions to $\mathcal{S}$CIFF. The first one is the "$\rightarrow$" implication of Eq. 5, which makes the declarative relation between events and expectations symmetric. The operational semantics introduces a new transition, completely symmetric to that devoted to check fulfilment; the extension of the proofs for this are therefore straightforward.

The second 'important' addition is the notion of $\mathcal{A}$-minimality introduced in the declarative semantics. By choosing $\mathcal{A}$-minimal answers, therefore restricting the set of models considered, we do not affect completeness, which still holds by virtue of the completeness of $\mathcal{S}$CIFF. Concerning soundness, instead, we have to prove that, given a goal $\mathcal{G}$, for each abductive WAV^e computation:

$$\mathcal{KB}_{ws} \cup \mathcal{IC}_{ws} \cup \mathcal{IC}_{ws'} \vdash_{\mathbf{HAP} \cup \mathbf{EXP} \cup \Delta} \mathcal{G} \tag{6}$$

the computed set $\mathbf{HAP} \cup \mathbf{EXP} \cup \Delta$ corresponds to a possible interaction achieving $\mathcal{G}$. Again, the proof can exploit the soundness result of $\mathcal{S}$CIFF [15], apart from the $\mathcal{A}$-minimality requirement introduced here. This further condition requires to prove that the computed set $E = \mathbf{HAP} \cup \mathbf{EXP} \cup \Delta$ for goal $\mathcal{G}$ corresponds to an ($\mathcal{A}$-minimal) possible interaction for $\mathcal{G}$. First, it can easily be proven that a WAV^e computation corresponds to a (WAV^e) computation into a *single* split program obtained from the original one. Furthermore, it can be proven that a WAV^e computation corresponds to a (WAV^e) computation into a *renamed* split program obtained from the former by considering only the applied clauses and the fired social integrity constraints, and by duplicating and properly renaming them as many times as each of them is applied or fires. Let us denote $P^{split} = \mathcal{KB}_{ws}^{split} \cup \mathcal{IC}_{ws}^{split} \cup \mathcal{IC}_{ws'}^{split}$ such a *renamed* split program.

Example 4. Let us consider the following program:

$$\begin{array}{rcl} \mathbf{E}(X) \ \vee \ \mathbf{H}(X) & \leftarrow & \mathbf{E}(X). \\ \mathit{goal} & \leftarrow & \mathbf{E}(p). \\ \mathit{goal} & \leftarrow & \mathbf{E}(q). \end{array}$$

This program has two different successful derivations for *goal*, originated, respectively, by the following two renamed split programs:[6]

P_1^{split}		
$\mathbf{H}(p)$	$\leftarrow$	$\mathbf{E}(p).$
goal	$\leftarrow$	$\mathbf{E}(p).$

P_2^{split}		
$\mathbf{H}(q)$	$\leftarrow$	$\mathbf{E}(q).$
goal	$\leftarrow$	$\mathbf{E}(q).$

Thanks to the soundness of $\mathcal{S}$CIFF we have that E is a possible interaction for $\mathcal{G}$, given the considered *renamed* split program. We still have to prove that this set is $\mathcal{A}$-minimal. This part can be proven by reductio ad absurdum. Suppose there exists a smaller set $E' \subset E$ and that E' is a possible interaction for $\mathcal{G}$ in the *renamed* split program. Due to $\mathcal{S}$CIFF completeness, then there also exists a (WAV^e) computation which computes a set $E'' \subseteq E'$ for $\mathcal{G}$. But this is not possible, by the way the *renamed* split program has been built. Contradiction!

We will next demonstrate the operational functioning of verification in WAV^e in the *alice* & *eShop* scenario.

[6] For the sake of keeping a lightweight notation, we do not show renamed variables.

7 Verification in WAV^e

In the following, the sets $\mathbf{EXP}_a^N$ and $\mathbf{HAP}_a^N$ represent the evolution of *alice*'s expectations and events as WAV^e's derivation progresses; N is an incremental index. Let g be the following goal of *alice*'s:

$$g \leftarrow have(alice, device, 50). \qquad \text{(goal)}$$

Then, by unfolding of clause alice3,

$$\mathbf{EXP}_a^0 = \{ \quad \mathbf{E}(eShop, alice, deliver(device), T_s) \wedge T_s < 50 \ \} \qquad (by\ \text{(alice3)})$$

To this expectation, *eShop* will react by expecting a payment:

$$\begin{aligned}\mathbf{EXP}_a^1 = \{ \quad & \mathbf{E}(eShop, alice, deliver(device), T_s) \wedge T_s < 50 \\ & \wedge (\mathbf{E}(alice, eShop, pay(device, cc), T_{cc}) \wedge T_{cc} < T_s \\ & \quad \vee \mathbf{E}(alice, eShop, pay(device, cash), T_{ca}) \wedge T_{ca} < T_s \\ & \quad \vee \mathbf{E}(alice, eShop, pay(device, cheque), T_{ch}) \wedge T_{ch} < T_s) \ \}\end{aligned} \qquad (by\ \text{(shop1)})$$

Since the expectation containing the payment by *cc* is the only one which generates an expectation matching a rule of *alice* ((alice1)), the first expectation among the three payment alternatives is selected (the other branches eventually fail by Eq. 5, because no matching $\mathbf{H}$ is abduced). This choice triggers (alice1):

$$\begin{aligned}\mathbf{EXP}_a^2 = \{ \quad & \mathbf{E}(eShop, alice, deliver(device), T_s) \wedge T_s < 50 \\ & \wedge \mathbf{E}(alice, eShop, pay(device, cc), T_{cc}) \wedge T_{cc} < T_s \\ & \wedge \mathbf{E}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_{cc} \ \}\end{aligned} \qquad (by\ \text{(alice1)})$$

Then (shop3) fires, and abduces the happening of *give_guarantee* event. We then have:

$$\begin{aligned}\mathbf{EXP}_a^3 = \{ \quad & \mathbf{E}(eShop, alice, deliver(device), T_s) \wedge T_s < 50 \\ & \wedge \mathbf{E}(alice, eShop, pay(device, cc), T_{cc}) \wedge T_{cc} < T_s \\ & \wedge \mathbf{E}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_{cc} \ \}\end{aligned} \qquad (by\ \text{(alice1)})$$

$$\mathbf{HAP}_a^3 = \{ \quad \mathbf{H}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_{cc} \ \} \qquad (by\ \text{(shop3)})$$

Given the guarantee, *alice* will pay by credit card (rule (alice2) fires):

$$\begin{aligned}\mathbf{EXP}_a^4 = \{ \quad & \mathbf{E}(eShop, alice, deliver(device), T_s) \wedge T_s < 50 \\ & \wedge \mathbf{E}(alice, eShop, pay(device, cc), T_{cc}) \wedge T_{cc} < T_s \\ & \wedge \mathbf{E}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_{cc} \ \}\end{aligned}$$

$$\begin{aligned}\mathbf{HAP}_a^4 = \{ \quad & \mathbf{H}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_{cc} \\ & \wedge \mathbf{H}(alice, eShop, pay(device, cc), T_{cc}) \wedge T_{cc} < T_s \ \}\end{aligned} \qquad (by\ \text{(alice2)})$$

Having received the payment, *eShop*'s policy would be to deliver the device:

$$\begin{aligned}\mathbf{EXP}_a^5 = \{ \quad & \mathbf{E}(eShop, alice, deliver(device), T_s) \wedge T_s < 50 \\ & \wedge \mathbf{E}(alice, eShop, pay(device, cc), T_{cc}) \wedge T_{cc} < T_s \\ & \wedge \mathbf{E}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_{cc} \ \}\end{aligned}$$

$$\begin{aligned}\mathbf{HAP}_a^5 = \{ \quad & \mathbf{H}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_{cc} \\ & \wedge \mathbf{H}(alice, eShop, pay(device, cc), T_{cc}) \wedge T_{cc} < T_s \\ & \wedge \mathbf{H}(eShop, alice, deliver(device), T_s) \wedge T_s < 50 \ \}\end{aligned} \qquad (by\ \text{(shop2)})$$

Summarising, *alice* devised the following set of events, which should let her achieve her goal (have the desired device) while respecting both of *alice*'s and *eShop*'s policies.

$$\begin{aligned}\mathcal{C}_a = \quad \{ \quad & \mathbf{H}(eShop, alice, give\_guarantee, T_g) \wedge T_g < T_p \\ & \wedge \mathbf{H}(alice, eShop, pay(device, cc), T_p) \wedge T_p < T_s \\ & \wedge \mathbf{H}(eShop, alice, deliver(device), T_s) \wedge T_s < 50) \ \}\end{aligned}$$

8 Discussion

WAV^e is a framework intended for describing declaratively the behavioural interface of web services, and for testing operationally the possibility of fruitful interaction between them. WAV^e answers the question "does there exist a viable interaction, between two given web services, which achieves a given goal $\mathcal{G}$?" In case of success, WAV^e produces a set of expectations about events. WAV^e is particularly suitable for highly dynamic environments, in which inter-operability is an unknown that has to be checked.

WAV^e uses and extends a technology initially developed for on-line compliance verification of agent interaction to protocols [9]. $\mathcal{S}$CIFF and the protocol specification language based on social integrity constraints were motivated and inspired by conspicuous work done in the context of agent interaction in open societies, notably work by Singh [13] and colleagues. The extension of such a work to the context of web services, centering around the concept of policies, as proposed in this work, seems to be very promising. The idea of policies for web services and policy-based reasoning is one that many other authors also adopt. We will cite work by Finin and colleagues [16], and by Bradshaw and colleagues [17], the first one with an emphasis on representation of actions, the latter on the deontic semantic aspects of web service interaction. We acknowledge the importance of action modelling and we point that the idea of expected behaviour of web services can have a deontic reading. In fact, previous work on $\mathcal{S}$CIFF has been devoted to investigating and clarifying the interesting links between deontic operators and expectation-based reasoning [18]. The distinguishing features of WAV^e, compare to most work of literature, are its logical underpinning and its sound and complete operational characterisation. It is in our agenda to carry out an extensive empiric evaluation of WAV^e based on interesting cases and scenarios such as those proposed in related work, and on the existing implementation of the $\mathcal{S}$CIFF framework.[7]

Another direction of current work relates to the actual use of the answers of WAV^e by web services after they manage a successful derivation. In principle, the sequence of events produced by WAV^e could be instantiated into a concrete sequence of messages, which will guarantee the achievement of $\mathcal{G}$, under ideal external conditions. But this is true only if the policies disclosed by both web services are a faithful representation of their actual behaviour. This may not be the case, as for example policies may depend on sensible data, and web services may be not allowed to disclose full information to the outside. In that case nothing warrants that the course of action produced by WAV^e will be satisfactory for either web service. We might then have to resort to further steps. For example both web services could "formally" agree that a certain course of events in an acceptable option, possibly after another mutual verification phase. This is subject of ongoing work. Finally, we are currently investigating the exchange of policies between web services, for which a suitable interaction

[7] See `http://lia.deis.unibo.it/research/sciff`.

protocol needs to be devised. We are thinking of specifying such a protocol for exchanging the policies in the same language WAV^e uses to specify policies.

References

1. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services version 1.1 (2003) `http://www.ibm.com/developerworks/library/ws-bpel/`.
2. Adi, A., Stoutenburg, S., Tabet, S., eds.: *Proc. 1st Int. Conference on Rules and Rule Markup Languages for the Semantic Web*, *LNCS* **3791**, Springer Verlag (2005)
3. Working Group on Rule Interchange Format: Use cases and requirements. `http://www.w3.org/2005/rules/wg/ucr/draft-20060323.html` (2006)
4. Bry, F., Eckert, M.: Twelve theses on reactive rules for the web. In: Proceedings of the Workshop on Reactivity on the Web, Munich, Germany (2006)
5. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation **2** (1993) 719–770
6. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In: *Proc. 1st Pacific Rim International Conference on Artificial Intelligence*, Ohmsha Ltd. (1990) 438–443
7. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33** (1997) 151–165
8. Denecker, M., Schreye, D.D.: SLDNFA: an abductive procedure for abductive logic programs. Journal of Logic Programming **34** (1998) 111–167
9. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: The SOCS computational logic approach for the specification and verification of agent societies. In: *Global Computing 2004*, *LNAI* **3267**, Springer-Verlag (2005) 324–339
10. Sakama, C., Inoue, K.: Abductive logic programming and disjunctive logic programming: their relationship and transferability. Journal of Logic Programming **44** (2000) 75–100
11. Gelfond, M., Lifschitz, V.: Classical negation in logic programming and disjunctive databases. New Generation Computing **9** (1991) 365–385
12. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: The $\mathcal{S}$CIFF abductive proof-procedure. In: *AI*IA*, *LNAI* **3673**, Springer-Verlag (2005) 135–147
13. Singh, M.: Agent communication language: rethinking the principles. IEEE Computer (1998) 40–47
14. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the $\mathcal{S}$CIFF proof-procedure. Technical Report DEIS-LIA-06-001, DEIS, University of Bologna (Italy) (2006)
15. Gavanelli, M., Lamma, E., Mello, P.: Proof of properties of the SCIFF proof-procedure. Technical Report CS-2005-01, DI, Università di Ferrara (2005) Available at `http://www.ing.unife.it/informatica/tr/CS-2005-01.pdf`.
16. Kagal, L., Finin, T.W., Joshi, A.: A policy based approach to security for the semantic web. In *Proc. 2nd ISWC. LNCS* **2870**, Springer (2003) 402–418
17. Uszok, A., Bradshaw, J.M., Jeffers, R., Tate, A., Dalton, J.: Applying KAoS services to ensure policy compliance for semantic web services workflow composition and enactment. In *Proc. 3rd ISWC. LNCS* **3298**, Springer (2004) 425–440
18. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Sartor, G., Torroni, P.: Mapping deontic operators to abductive expectations. Computational and Mathematical Organization Theory (2006) To appear.

Towards Automated Web Service Composition with the Abductive Event Calculus

Onur Aydın[1] Nihan Kesim Cicekli[2] Ilyas Cicekli[3]

[1]Microsoft Corporation, Seattle, U.S.A.
[2]Department of Computer Engineering, METU, Ankara , Turkey
[3]Department of Computer Engineering, Bilkent University, Ankara, Turkey
onura@microsoft.com, *nihan@ceng.metu.edu.tr, ilyas@cs.bilkent.edu.tr*

In this paper, the use of a logic programming framework, the event calculus [3], is discussed in the automated composition of web services. The web service discovery problem is beyond of the scope of the paper. Our goal is to show that the event calculus can be used for both definitions of web services composition. That is, it can be used to generate a composite process as the output of planning. It can also be used to define a generic composition and produce a user specific composition (plan) according to the user constraints. Abductive planning of event calculus is used to show that when atomic services are available, composition of services that would yield the desired effect is possible. An abductive planner implementation of the event calculus [4] is extended to be used for plan generation.

Event Calculus and Abductive Event Calculus

Event calculus [3] is a logical formalism which is used with domains where events affect and change the world. The formulation of the event calculus is defined in first order predicate calculus. There are actions and effected fluents. Fluents are changing their valuations according to effect axioms defined in the theory of the problem domain. Each event calculus theory is composed of axioms. The axioms that define whether a fluent holds starting from the initial state are as follows.

$HoldsAt(F, T) \leftarrow Initially(F) \wedge \neg Clipped(T_0, F, T)$

$HoldsAt(\neg F, T) \leftarrow Initially(\neg F) \wedge \neg Declipped(T_0, F, T)$

Axioms below are used to deduce whether a fluent holds or not at a specific time.

$HoldsAt(F, T) \leftarrow Happens(E, T_1), Initiates(E, F, T_1), T_1 < T , \neg Clipped(T_1, F, T)$

$HoldsAt(\neg F, T) \leftarrow Happens(E, T_1), Terminates(E, F, T_1), T_1 < T, \neg Declipped(T_1, F, T)$

The predicate *Clipped* defines a time frame for a fluent that is overlapping with the time of an event which terminates or releases this fluent. Similarly *Declipped* defines a time frame for a fluent which overlaps with the time of an event that initiates or releases this fluent.

Abduction is logically the inverse of deduction. It is used over the event calculus axioms to obtain partially ordered sets of events. Abduction is handled by a second order logical prover which is defined as an abductive theorem prover (ATP) in [4]. ATP tries to solve the goal list by proving the elements one by one. During the resolution, abducible predicates, < (temporal ordering) and *Happens*, are stored in a residue to keep the record of the narrative. The narrative is a sequence of time-stamped events, and the residue keeping a record of the narrative is the plan.

Web Services Composition with Abductive Planning

The event calculus can be used for planning as it is theoretically explained in [4]. The planning problem in the event calculus is formulated in simple terms as follows: Given the domain knowledge (i.e. a conjunction of *initiates, terminates*), the Event Calculus axioms (i.e. *HoldsAt*) and a goal state (e.g. *HoldsAt*(*f,t*)), the abductive theorem prover generates the plan which is a conjunction of *Happens* and temporal ordering predicates. ATP returns a valid sequence of time stamped events that leads to the resulting goal. Multiple solutions are thought to be as different braches of a more general plan and they are obtained with the help of backtracking.

In the event calculus framework, the web services are modeled as events with input and output parameters. For instance, a web service, which returns the availability of a flight between two locations, can be described as:

Happens(*getFlights*(*Orgn, Dest, FlDate, FNL*), T_1, T_1) ←
Ex_getFlights(*Orgn, Dest, FlDate, FNL*).

The predicate *Ex_getFlights* is used as a precondition for the event and it is invoked anytime it is added to the plan to populate the input parameters.

It is also possible to create generic compositions in the event calculus. ATP can then be used to generate a plan which corresponds to the user specific execution of the composite service. Composite services correspond to compound events in the Event Calculus [2]. An OWL-S to event calculus translation scheme is presented to show that OWL-S composition constructs can be expressed as event calculus axioms [1].

Conclusions

The Event Calculus framework can be used for the solution of web service composition problem. When a goal situation is given, the event calculus can find proper plans as web service compositions with the use of abduction technique. It is possible that the solutions that are generated by the event calculus can be compiled into a graph like composition for the satisfaction of the goal situation [1]. The Event Calculus can also be used to create generic compositions and ATP can be used to generate a plan which corresponds to the user specific execution of the composite service.

As a future work, the results expressed in this paper will be implemented in a real web environment. Common structures of compositions will be expressed as *meta* event calculus constructs. Another improvement might be on queries which are known a priori for the compositions. Queries can be entered in a natural language and then translated into the event calculus goals.

References

1. Aydin, O., Automated web service composition with the event calculus, M.S. Thesis, Dept. of Computer Engineering, METU, Ankara, 2005.
2. Cicekli N. K., Cicekli I. Formalizing the specification and execution of workflows using the event calculus, to appear in Information Sciences.
3. Kowalski R. A., Sergot M. J.A Logic-Based Calculus of Events. New Generation Computing, Vol. 4(1), pp. 67--95, 1986.
4. Shanahan M.P. An abductive event calculus planner. Journal of Logic Programming, Vol. 44(1-3), pp. 207--240, July 2000.

Modeling, verifying and reasoning about web services (Extended Abstract) ⋆

Alberto Martelli

co-ordinator of the Italian MIUR Project PRIN 2005
"*Specification and verification of agent interaction protocols*"
Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
E-mail: mrt@di.unito.it

This paper describes the research activities carried out in the context of the Italian MIUR Project PRIN 2005 "Specification and verification of agent interaction protocols", concerning the area of modeling, verification and reasoning about web services.

Web services are an emergent paradigm for implementing business collaborations over the web. Each service has an interface that is accessible through standard protocols and that describes the interaction capabilities of the service. It is possible to develop new applications by combining and integrating existing web services. In this scenario, various languages have been developed for modeling processes and their interaction protocols. In particular, the language WS-BPEL has emerged as the standard for specifying the business processes of single services, while the global view of the interaction is captured by the concept of choreography, expressed by using specific languages like WS-CDL. Nowadays, in many application domains it is getting more and more common describing and realizing the offered services by means of a set of communicating *agents*. Techniques for the specification and verification of the interactive behavior of open agent systems find an immediate application in web services.

The goal of our project is to prove the usefulness and the applicability of techniques based on declarative approaches for tackling issues typical of the web service application area. Our claim is that web service interactions should be represented according to some formalism which relies on well-founded models with a clear semantics. Furthermore, automated tools for reasoning about such a description and performing tasks of interest must be developed.

The goal of the project is pursued through three main steps:

- Definition of suitable formalisms for the specification and verification of interaction protocols/choreographies;
- Development of techniques for automatic property verification and reasoning about web services;
- Translation of modeling languages into the formal languages developed in the project.

⋆ Additional authors: M. Baldoni, C. Baroglio, V. Patti, C. Schifanella (Dipartimento di Informatica, Università di Torino), M. Alberti, M. Gavanelli, E. Lamma, F. Riguzzi, S. Storari (ENDIF, Università di Ferrara), F. Chesani, A. Ciampolini, P. Mello, M. Montali, P. Torroni (DEIS, Università di Bologna), A. Bottrighi, L. Giordano, V. Gliozzi, D. Theseider Dupré, P. Terenziani (Dipartimento di Informatica, Università del Piemonte Orientale), G. Casella, V. Mascardi (DISI, Università di Genova)

In particular, we are considering the following issues:

Specification and verification of interaction protocols. We are defining and comparing different formalisms. The first approach is based on *abductive logic programming*, and exploits the *S*CIFF framework, developed in the European project SOCS. Within the *S*CIFF framework, a language suitable for specifying global protocols has been provided, and an abductive proof procedure has been developed [2].

A different approach makes use of a formalism for *reasoning about actions*. Web services can be described by specifying their interaction protocols in an action theory based on a temporal logic. The proposed framework provides a simple formalization of the communicative actions in terms of their effects and preconditions, and the specification of an interaction protocol by means of temporal constraints [5].

Conformance verification. We are studying the issue of verifying whether the business process of some peers will produce interactions which are conformant to the agreed protocol (legality issue). Such issue is tackled by the so called *conformance test*, considered as a means for certifying the capability of interacting of the involved parts [4].

Reasoning about web service behavior. Formalisms for reasoning about actions are suitable for dealing with web service composition and selection. In particular, we have applied planning techniques to the problem of composing web services, and to the problem of *personalizing* web service selection and composition w.r.t. user preferences [3].

We have also tackled the problem of dynamically understanding if two web services can inter-operate, without having a-priori knowledge of each one capabilities, but reasoning on policies exchanged at run-time [1].

Implementation of Prolog-based agents. Another direction of research is aimed at factoring the three technologies of web services, intelligent agents, and Prolog, for implementing Prolog-based agents that reason about interaction protocols specified using WS-BPEL and WSDL [6].

References

1. M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Policy-based reasoning for smart web service interaction. In *ALPSWS'06*, 2006.
2. M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence*, 20(2-4):133–157, February-April 2006.
3. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for customizing web service selection and composition. *J. of Logic and Algebraic Programming, special issue on Web Services and Formal Methods*, 2006. To appear.
4. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *LNCS*. Springer, 2005.
5. Laura Giordano, Alberto Martelli, and Camilla Schwind. Specifying and verifying interaction protocols in a temporal action logic. *Journal of Applied Logic*, 2006. to appear.
6. V. Mascardi and G. Casella. Intelligent agents that reason about Web Services: a Logic Programming approach. In *ALPSWS'06*, 2006.

A RuleML Syntax for Answer-Set Programming*

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter, ianni, roman, tompits}@kr.tuwien.ac.at

1 Introduction

The need for sharing knowledge on the Web—and in particular rules in a standardized format—is an important issue. RuleML [1] has been the most prominent effort in this direction so far. Nonetheless, it turns out that none of the variants available is suitable for expressing nonmonotonic logic programs as used in the *answer-set programming* (ASP) paradigm, in which the former are assigned with a declarative semantics, known as *answer-set semantics* or *stable-model semantics* [3]. ASP in general is an important declarative problem-solving paradigm, gaining increasing attention in the recent years (see, e.g., [4]).

In this work, we present a new language variant in addition to the current RuleML proposal for expressing an ASP core language. Then we provide an extension to this core to accommodate different ASP dialects, substantially based on the notion of an *oracle atom*. Oracle atoms are rooted in the notion of an *external atom* [2]. A working translator from RuleML to ASP and vice versa is available. This way, RuleML specifications are made executable under the ASP semantics.

The framework we present here is supposed to be a starting point that should encourage both the Semantic Web and the ASP community to discuss and achieve a comprehensive RuleML interchange format for the ASP semantics. It is work in progress which, as we believe, will attract other participants after initial dissemination.

2 Description of the Work

Our approach towards extending RuleML to answer-set programs consists of several layers. First, we define a RuleML schema called *aspbase*, which encapsulates the syntax of traditional ASP. We then present an extension (*asporacle*) to this schema, facilitating the expression of a number of advanced constructs which are provided by current ASP solvers, such as aggregates, built-ins or external atoms, by a general syntactical element.

Figure 1 shows how an ASP specification written in RuleML can be processed. Translators rewrite an answer-set program in the general RuleML syntax into a textual representation suitable for a given reasoner. Each translator might accept a set of specific syntactic features. It enforces a specific meaning to each particular feature by rewriting it into the construct expected by the corresponding reasoner.

* This work was partially supported by the Austrian Science Fund (FWF) under grant P17212-N04, and by the European Commission through the IST Networks of Excellence REWERSE (IST-2003-506779).

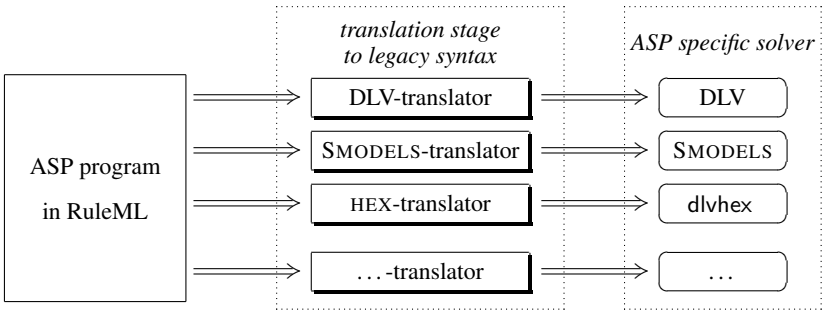


Fig. 1. ASP RuleML architecture

3 An Example

A complex construct like a *cardinality constraint*, featured by SMODELS, such as $1\ \{a, b, not\ c\}\ 2.$ can be modeled in our setting in the following way:

```
<Oracle>
  <Rel>cardCons</Rel>
  <Input>
    <Data xsi:type="xs:integer">1</Data>
    <Atom>
      <Rel>a</Rel>
    </Atom>
    <Atom>
      <Rel>b</Rel>
    </Atom>
    <Naf>
      <Atom>
        <Rel>c</Rel>
      </Atom>
    </Naf>
    <Data xsi:type="xs:integer">2</Data>
  </Input>
</Oracle>
```

For further examples of this language variant and the use of oracle atoms, we refer the reader to `http://www.kr.tuwien.ac.at/research/ruleml`.

References

1. H. Boley, S. Tabet, and G. Wagner. Design Rationale for RuleML: A Markup Language for Semantic Web Rules. In *Proc. SWWS 2001*, pages 381–401, 2001.
2. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *Proc. IJCAI 2005*, pp. 90–96.
3. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
4. S. Woltran (ed.). Answer Set Programming: Model Applications and Proofs-of-Concept. WASP, 2005. `http://www.kr.tuwien.ac.at/projects/WASP/report.html`.

Prolog Execution Engines for Description Logic Reasoners

Gergely Lukácsy, Zsolt Nagy, Péter Szeredi

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
{lukacsy, zsnagy, szeredi}@cs.bme.hu

Introduction. The goal of this poster is to present some Prolog-based [3] alternatives for Description Logic ABox-reasoning. We believe that Prolog can be used well for extending DL formalisms with rules. The top-down executional mechanism of Prolog suits the ABox-reasoning task by ignoring irrelevant data, and it is also worth to mention that the Prolog community has been developping Prolog for more than 30 years, providing a highly optimized logic programming environment. In our approaches, both the ABox-reasoning code and the attached rules can be executed in a Prolog framework. Queries are answered using the PTTP (Prolog Technology Theorem Proving) [4] approach. The aim of these reasoning algorithms is to efficiently answer instance-check and instance-retrieval queries when sizeable amounts of data are stored in the ABox. We describe two approaches of transforming a DL knowledge-base to Prolog clauses.

Soundness and completeness of both approaches are based on resolution and the PTTP technique. In the implementation of the ideas, procedural elements of Prolog such as the cut for filtering multiple solutions out are used for optimizing the code.

The Restricted Approach. [2] This approach provides ABox-reasoning services over an empty TBox. Let an extensionally reduced ABox $\mathcal{A}$ be given with the property that $\mathcal{A}$ is satisfiable[1]. The goal is to determine all the instances of an $\mathcal{ALC}$ query-concept C, or as a special case, determine if an individual o is an instance of the query-concept C. Reasoning is split into two parts: first a Prolog execution plan is produced, then the plan is executed on the ABox.

More specifically, we first transform the $\mathcal{ALC}$ query-concept into a union of tree-concepts. A *tree-concept* is an $\mathcal{ALC}$ concept formed using the intersection, existential restriction and atomic negation constructors only. Each tree-concept is transformed into a piece of Prolog-code individually. Prolog clauses belonging to each tree-concept have to be executed in a common namespace. We refer the reader to [2] on the details of the transformation.

The Intermediate Approach. [1] This approach provides ABox reasoning services over an extensionally reduced $\mathcal{ALC}$ knowledge-base containing an ABox and a restricted TBox. We exclude subsumption axioms $C \sqsubseteq D$ from the TBox

[1] We do not deal with ABoxes containing contradictions. When reading the assertions, we do not allow both $A(i)$ and $\neg A(i)$ to be simultaneously asserted for any A or i.

where $\forall R.E$ is a subconcept of the negation normal form of C or $\exists R.E$ is a subconcept of the negation normal form of D. This restriction is due to the fact that the current reasoning algorithm cannot handle Horn-clauses containing Skolem-functions. Let a TBox $\mathcal{T}$ conforming to the restrictions above and an extensionally reduced ABox $\mathcal{A}$ be given. The content of $\mathcal{T}$ is transformed to Prolog rules and the content of $\mathcal{A}$ is transformed to Prolog facts.

Comparison of the approaches. We have designed and experimented with a case-study on fault-tolerant behavior of systems. Using this case study, we compared the performance of the two approaches.

We summarize the basic differences between the two approaches in Table 1.

The *Restricted* approach presented in [2] focuses on ABox-reasoning over an empty TBox. Here, it is also possible to add non-DL Horn-clauses to the transformed set of clauses. These Horn-clauses make it possible for the knowledge-engineer to describe terminological knowledge regarding the instances of the knowledge-base. The main advantage of this approach is its performance and scalability.

On the other hand, the *Intermediate* approach [1] is able to provide ABox-inference services over a non-empty terminology box. The main advantage of this technique is that it provides reasoning services over a knowledge-base containing (a) a slightly restricted $\mathcal{ALC}$ TBox, (b) terminology level knowledge represented using Horn-clauses and (c) ABox-instances and relations. The content of the knowledge-base is transformed into Horn-clauses which are executed in Prolog using the PTTP technique. Execution is currently done by using our own PTTP Horn-clause interpreter. Preprocessing optimizations do not appear in this work, the solution is derived from the PTTP inference engine only. Although this approach is capable of solving ABox-inference problems over a non-empty TBox, we still refer to it as the *Intermediate Approach*, since we plan getting rid of the restrictions involving the TBox and the interpreted execution.

Table 1. Comparison of the two reasoning approaches.

Approach	*Restricted*	*Intermediate*
Expressive power	$\mathcal{ALC}$ ABox, no TBox	$\mathcal{ALC}$ ABox and restricted TBox
Preprocessing	all ABox-independent steps	none
Execution plan	runnable Prolog program	interpreted PTTP clauses

Summary, future work. We believe that Prolog-based ABox-reasoning can be well combined with Prolog rules and we found or initial results encouraging. In the future, we would like to combine the advantages of the two approaches and form an approach capable of efficiently handling ABox-reasoning over an arbitrary TBox and ABox.

References

1. Zsolt Nagy, Gergely Lukácsy, and Péter Szeredi. Description logic reasoning using the PTTP approach. to appear in the proceedings of dl2006. international description logic workshop, windermere, england., 2006.
2. Zsolt Nagy, Gergely Lukácsy, and Péter Szeredi. Translating description logic queries to Prolog. In *PADL*, volume 3819 of *Lecture Notes in Computer Science*, pages 168–182, 2006.
3. U. Nilsson and J. Maluszynski, editors. *Logic, Programming and Prolog.* John Wiley and Sons Ltd., 1990.
4. M. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog, Technical Note 464, SRI international, 1989.

Applying Prolog to Semantic Web Ontologies & Rules Moving Toward Description Logic Programs

K. Samuel[1], L. Obrst[1], S. Stoutenburg[2], K. Fox[2], P. Franklin[2], A. Johnson[2], K. Laskey[2], D. Nichols[1], S. Lopez[2], J. Peterson[2]

The MITRE Corporation*
[1] 7525 Colshire Drive, McLean, VA 22102-7508
[2] 1155 Academy Park Loop, Colorado Springs, CO 80910-3716
{samuel, lobrst, suzette, kfox, pfranklin, abjohnson, klaskey, dlnichols, slopez, jasonp}
@mitre.org

We are developing SWORIER (Semantic Web Ontologies and Rules for Interoperability with Efficient Reasoning), which is a system that uses Logic Programming to reason about ontologies and rules in order to answer queries. The system expects a human developer to create ontologies in the formalisms of OWL-DL (Web Ontology Language for Description Logic) along with rules in SWRL (the Semantic Web Rule Language) or RuleML (the Rule Markup Language). Then, at compile time, this information is translated into Prolog code using XSLTs (Extensible Stylesheet Language Transformations). In addition, a Prolog program called 'General Rules', which is meant to capture the semantics of OWL's primitives, is appended to the XSLT output to form a complete Prolog program. We then use knowledge compilation techniques to create an efficient version of the program. At run time, the system can answer queries and assimilate dynamic changes by reasoning over the given information.

For our prototype, we have developed ontologies and rules in a military command and control domain in which a supply convoy moves through an unsecured area. New information can become available at any time, such as an approaching sandstorm or the discovery of a new hostile theater object. Rules trigger alerts and recommendations to assist the commander in making decisions. For example, if an enemy unit is within the convoy's region of interest, the system reports that and recommends a new route.

Recent research has addressed issues similar to ours concerning combining logic programming with Semantic Web ontologies and rule technologies. Related work includes Description Logic Programming [2, 4, 6], answer set programming [1, 5, 7], and courteous logic programs [3]. In particular, we are building on the groundbreaking work of [8], addressing a number of problems that the paper suggested were unsolvable.

For example, Prolog typically has negation as (finite) failure, while OWL uses logical negation. So we created a new Prolog predicate called **`logicNot`**. Also, to

* The authors' affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the authors. We note that the views expressed in this paper are those of the authors alone and do not reflect the official policy or position of any other organization or individual.

address the fact that Prolog does not allow disjunction in the head, we will create another Prolog predicate, **`or`**. The **`logicNot`** predicate enables SWORIER to represent OWL's open world assumption and to reason about complementary and disjoint classes. And with the **`or`** predicate, SWORIER can handle enumerated classes (the **`owl:oneOf`** primitive).

We created a Prolog predicate for each OWL primitive, unlike [8], who made the ontology's (object-level) classes and properties into Prolog predicates. Our syntax makes it easier to enforce substitutivity of equivalent classes and to handle inconsistent cardinality restrictions.

Most inconsistencies can be addressed in multiple ways, such as by sending an error message to the developer or by creating a new unnamed individual to satisfy a constraint. In this way, we address constraints such as those imposed by cardinality and existential quantification.

At run time, SWORIER can reason about queries, switch from one rule set to another, and assimilate assertions and deletions of individuals.

We ran experiments with the convoy use case described above. SWORIER was too slow for practical use until we implemented three techniques: extensionalization, avoiding reanalysis, and code minimization. Now SWORIER's knowledge compilation phase takes less than seven hours, and at runtime, SWORIER can answer a query in less than a second and assimilate a dynamic change in a few milliseconds.

In the future, we intend to test our ideas that address issues involving disjunction, inconsistencies, enumerated classes, cardinality, etc. We will also implement more OWL primitives and enable SWORIER to handle more kinds of dynamic changes.

1. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining Answer Set Programming with Description Logics for the Semantic Web. In: The Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (2004).
2. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: DLV-HEX: Dealing with Semantic Web under Answer-Set Programming. In: The Proceedings of the 4th International Semantic Web Conference (2005).
3. Grosof, B., Labrou, Y., Chan, H. Y.: A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In: The Proceedings of the 1st ACM Conference on Electronic Commerce (1999).
4. Grosof, B., Horrocks, I., Volz, R., Decker, S.: Description Logic Programs: Combining Logic Programs with Description Logic. In: The Proceedings of the 12th International Conference on the World Wide Web (2003).
5. Heymans, S., Vermeir, D.: Integrating Description Logics and Answer Set Programming. In: The Proceedings of the International Workshop on Principles and Practice of Semantic Web Reasoning. Springer LNCS 2901 (2003), 146-159.
6. Motik, B., Rosati, R.: Closing Semantic Web Ontologies. University of Karlsruhe Technical Report. www.cs.man.ac.uk/~bmotik/publications/paper.pdf (2006).
7. Niemel, I., Simons, P.: Smodels — An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In: The Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (1997), 420-429.
8. Volz, R., Decker, S., Oberle, D.: Bubo — - Implementing OWL in Rule-Based Systems. www.daml.org/listarchive/joint-committee/att-1254/01-bubo.pdf (2003).

Author Index