Augmenting Knowledge Representation and Reasoning Languages with Customizable Metalogic Features *

Stefania Costantini[1] and Andrea Formisano[2]

[1] Università di L'Aquila, stefania.costantini@univaq.it
[2] Università di Perugia, INdAM-GNCS, andrea.formisano@unipg.it

Abstract. In this paper we present a methodology for introducing customizable metalogic features in logic-based knowledge representation and reasoning languages. The proposed approach is based on concepts of introspection and reflection previously introduced and discussed by various authors in relevant literature. This allows a knowledge engineer to specify enhanced reasoning engines by defining properties and meta-properties of relations as expressible for instance in OWL. We employ meta-level axiom schemata based upon a naming (reification) device. We propose general principles for extending the semantics of "host" formalisms accordingly. Suitable pre-defined libraries of properties can be made available, while user-defined new schemata are also allowed. We make the specific case of Answer Set Programming (ASP), where such features may be part of software engineering toolkits for this programming paradigm. We show how to extend the Answer Set Programming principles and practice to accommodate the proposed methodology, so as to perform meta-reasoning within the plain ASP semantics (where we mention and discuss approaches where significant extensions are required).

Keywords: Web-ontologies · Meta-reasoning · Answer Set Programming

1 Introduction

The OWL ontology language [1, 2] provides a powerful data modeling language, and automated reasoning abilities based upon Description Logics [3]. There are, as is well known, different versions of OWL depending upon the kind of reasoning one wishes to use and can computationally afford; in fact, such versions range from polynomial-time complexity, with reasoning capabilities equivalent to SQL, to very complex reasoning capabilities even leading to undecidability. Relevant

* This work is partially supported by INdAM-GNCS-17 and INdAM-GNCS-19 projects, by project B.I.M.–2018.0419.021, by Univ.of Perugia (projects "ricerca-di-base-2016", YASMIN, CLTP, and RACRA). Supported by Action COST CA17124 "DigForASP".

aspects concerning knowledge representation and reasoning that can be found in OWL are: (i) properties of relations: e.g., symmetry/asymmetry, transitivity, functionality, reflexivity/irreflexivity, domain/range; (ii) relations between relations (meta-properties): e.g., inverse-of, equivalence, disjointness, subclass; (iii) cardinality of relations.

Such features have widely demonstrated their usefulness in the definition and use of ontologies in the Semantic Web, and in our opinion they would find useful application in many knowledge engineering and automated reasoning languages. In fact, ontologies are pervading many areas of knowledge and data representation and management, and a lot of effort has been spent on the development of sufficiently expressive languages for the representation and querying of ontologies.

In this paper we present a methodology for introducing OWL-like features in logic-based knowledge engineering, representation and reasoning languages. In fact, ontology languages such as OWL have been devised in the context of the Semantic Web to help automated processes ("intelligent agents") to access information in a uniform and principled way [1]. Therefore ontologies (expressed in such languages) are expected to provide structured vocabularies and definitions that specify the relationships between terms and properties, thus allowing intelligent agents (and, possibly, also humans) to interpret their meaning flexibly yet unambiguously. This, however, is useful and important also within knowledge representation languages that define knowledge bases which are at the "core" of intelligent systems and support their reasoning processes. Thus, here we introduce an approach based on concepts of introspection and reflection discussed, among others, in [4–6]. In order to implement an engine realizing properties and meta-properties of relations inspired by those expressible in OWL, we employ meta-level axiom schemata based upon a naming (reification) device. We propose a method for extending the semantics accordingly. Practically, such schemata should be added by default to any program/theory. We do not claim to reproduce all OWL features and maybe not even most of them. In fact we could not, as we stay within decidable frameworks based upon CWA (Closed-World Assumption). However, the features that we reproduce are widely used, and they are useful in many practical contexts. Moreover, we improve over OWL as user-defined new properties are allowed in our proposal.

As a proof of concept we consider Answer Set Programming (ASP, cf. [7] and the references therein), where the proposed techniques can be seen as a software engineering method to be employed, together with others, in the development of ASP programs. The ASP programming paradigm in fact, though general, powerful and widely used, is at present employed in a quite basic way. The available software engineering constructs include aggregates and weight and cardinality constraints (discussed below in the section on ASP), and tools for modularity of ASP programs (cf. [8, 9] and the references therein) and for "templates" [10], where one can define subprograms/macros. We will try to demonstrate that our techniques can usefully enrich the available toolkit.

The possibility of improving knowledge engineering capabilities by means of metaprogramming and metareasoning has been explored in the past [11]. In logic settings such as Prolog [12] the aim was to enlarge representation and reasoning possibilities while avoiding to resort to a higher-level setting, mainly by using meta-interpreters [13] and trying to equip them with a logical semantics [14, 15], or by devising specialized language extensions [16, 6]. Recently, the Rulelog language [17] features some seemingly higher-order characteristics inspired by the HiLog second-order language [18], though transposed into a simple (though limited under the point of view of reasoning capabilities) first-order representation.

The mechanisms that we propose allow programmers to define relations (also) in terms of their properties and meta-properties; this on the basis of metalevel definitions that should be part of any program using them. Many such definitions (namely, those concerning the most commonly-used properties) might be predefined and imported by a program upon need, e.g. via libraries. However, the approach also allows programmers to define their own new (meta) properties. So, the average programmer does not define and does not need to see the definition of properties of relations, but rather just uses predefined ones though the skilled programmer may optionally define new properties.

So, a given knowledge representation language becomes in fact extensible, where an extension finds an immediate semantic and computational counterpart. We stay within the realm of knowledge representation languages based upon computational logic and logic programming [12], syntactically based upon some first-order language. Among them are Prolog, ASP, and many agent-oriented programming languages [19]. As mentioned we discuss in particular the case of ASP, which is at present a well-known successful logic programming and knowledge representation language paradigm. For ASP, we will show in detail how the approach might be practically and easily implemented.

The paper is organized as follows. We first shortly summarize the basic principles of the OWL language in Section 2, and we recall the concept of *reification* (naming) of first-order terms and atoms (Section 3). Then, we present our approach in Sections 4-5. Later, after shortly recalling ASP (Section 6), in Section 7 we show (also by exploiting significant examples) how the approach can be customized to the case of ASP. Finally, in Section 8 we discuss related work on meta-reasoning in ASP and draw conclusions.

2 Background: OWL

OWL is a language for the definition of *ontologies* (the reader may refer to `www.w3.org/TR/2012/REC-owl2-primer` for an introduction). The term ontology has a complex history in Philosophy, and recently in Computer Science. In Knowledge Representation, an ontology is a set of formal statements aimed to describe some part of the world (often referred to as the "domain of interest" or the "subject matter" of the ontology). Precise descriptions satisfy several purposes, among which: they prevent misunderstandings in human communication

and they ensure a better software behavior, especially when different software modules interact.

In order to precisely describe a domain of interest, the OWL language is based upon a *vocabulary*. The meanings of terms is established by stating how each term is interrelated to the other terms (and similarly for classes, properties, and individuals). A terminology, providing a vocabulary together with such interrelation information constitutes an essential part of an OWL ontology. Besides this "terminological" knowledge, usually called TBOX, an ontology might also contain so called "assertional knowledge" (ABOX) that introduces concrete objects of the considered domain. The TBOX part is the analogous of the set of rules of a Prolog program, while the ABOX is the analogous of the set of facts.

OWL 2 is not a programming language, rather it provides a declarative way to describe knowledge in a logical way. For the decidable fragments of OWL, appropriate tools (so-called *reasoners*) can then be used to infer further information from a given TBOX+ABOX description. How these inferences are realized algorithmically depends on the specific implementations and on the fragments of OWL considered. Still, the correct answer to any of such question is predetermined by the OWL formal semantics.

In OWL it is possible to define classes of objects/individuals, membership to classes, class inclusion, equivalence and disjointness, class hierarchies. Concerning object properties, they correspond to binary predicates, i.e., to relations, and are expressed concerning specific objects which are related by each property. It is also possible to express negative assertions, concerning individuals *not* enjoying some property. It is possible to specify hierarchies of properties (e.g., to state that some properties are sub-properties of other ones) and to define domain and range of each property. Among the "Advanced Use of Properties", one can state that certain properties are reflexive or irreflexive, symmetric or asymmetric, transitive, equivalent to some other properties, or disjoint from them. It can be stated that a property is functional, or that its inverse is functional.

3 Background: Naming Mechanisms

A *reification mechanism*, also known as "naming relation", is a method for representing within a first-order language expressions of the language itself, without resorting to higher-order features. Naming relations can be introduced in several manners. For a discussion of different possibilities, with their advantages and disadvantages, see, e.g., [20–23]. However, all of them are based upon introducing distinguished constants, function symbols (if available) and predicates, devised to construct names. For instance, given atom $p(a, b, c)$ a name might be $atom(pred(p'), args([a', b', c'])$ where p' and a', b', c' are new constants intended as names for the syntactic elements p and a, b, c and notice that: p is a predicate symbol (which is not a first-class object in first-order settings), *atom* is a distinguished predicate symbol, *args* a distinguished function symbol and [...] is a list.

More formally, let us consider a standard first-order language $\mathcal{L}$ including sets of *predicate*, *constant* and (possibly) *function* symbols, and a (possibly denumerable) set of symbols of *variables*. As usual, well-formed formulas have *atoms* as their basic constituents, where an atom is built via the application of a predicate to a number n (according to the predicate arity) of *terms*. The latter can be variables, constants, or compound terms built by using function symbols (if available). We augment $\mathcal{L}$ with new symbols, namely a new constant (say of the form p') for each predicate symbol p, a new constant (say f') for each function symbol f, a new constant (say c') for each constant symbol c, and a denumerable set of meta-variables, that we assume to have the form X' so as to distinguish them syntactically from "plain" variables X. The new constants are intended to act as names, where we will say that, syntactically, p' denotes p, f' denotes f and c' denotes c, respectively. The new variables can be instantiated to *meta-level formulas*, i.e., to terms involving names, where we assume that plain variables can be instantiated only to terms *not* involving names. We assume an underlying mechanism managing the naming relation (however defined), so we can indicate the name of, e.g., atom $p(a, b, c)$ as $p'(a', b', c')$ and the name of a generic atom A as $\uparrow A$.

4 Metalogic for Properties of Relations

In this paper we mainly consider rule-based languages, where rules are typically represented in the form $Head \leftarrow Body$ where $\leftarrow$ indicates implication; other notations for this connective can alternatively be employed. In Prolog-like languages, $\leftarrow$ is indicated as $:-$, and *Body* is intended as a conjunction of literals (atoms or negated atoms) where $\wedge$ is conventionally indicated by a comma.

We will represent properties of relations in OWL style by means of metalevel rules. To define such rules, we assume to augment the language $\mathcal{L}$ at hand not only with names, but with the introduction of two distinguished predicates, *solve* and *solve_not*. An atom A is a *base atom* if it does not involve names and its predicate is neither *solve* nor *solve_not*. Distinguished predicates will allow us to respectively extend/restrict the meaning of the other predicates in a declarative way. In fact, *solve* and *solve_not* take as arguments (names of) atoms (involving any predicate excluding themselves), and thus they are able to express sentences about relations. Names of atoms, in particular, are allowed *only* as arguments of *solve* and *solve_not*. Also, *solve* and *solve_not* can occur in the body of a metarule *only if* the predicate of its head is in turn either *solve* or *solve_not*.

So, metalevel rules in general allow arguments of predicates to be names of predicates, function symbols and constants. A particular kind of metarules, that we call *metaevaluation rules*, have distinguished predicates *solve* and *solve_not* in their head, and possibly also in their body, taking as argument names of atoms.

Below is a simple example of the use of *solve* to specify which properties a reflexive predicate meets. Namely that $p(a, a)$ can be derived for any element a belonging to the predicate domain; here, this is elicited from a occurring in

the extensional definition of p. The first rule is a metaevaluation rule, featuring predicate *solve* in its head, taking as argument the name of an atom; the latter two rules are 'simple' metalevel rules not involving either *solve* or *solve_not*, and taking as arguments metalevel constants.

$solve(P'(X', X')) :- reflexive(P'), in\_domain(P', X').$
$in\_domain(P', X') :- solve(P'(X', Y')).$
$in\_domain(P', X') :- solve(P'(Y', X')).$

Our objective is to make it automatic to derive $p(a, a)$ whenever a program includes this definition, a fact $reflexive(p')$ occurs in the program, and a is in the domain of p. Vice versa, we can define:

$solve\_not(P'(X', X')) : - irreflexive(P').$

with the aim to prevent the derivation of $p(a, a)$ for any predicate p which have been declared to be irreflexive (i.e., for which a fact $irreflexive(p')$ occurs in the program).

Following [24], in general terms we understand a semantics *SEM* for logic knowledge representation languages/formalisms as a function which associates a theory/program with a set of sets of atoms, which constitute the intended meaning. When saying that Π is a program, we mean that it is a program/theory in the (here unspecified) logic language/formalism that one wishes to consider.

We start with the following restriction on sets of atoms that should be considered for the application of *SEM*. First, as customary, we only consider sets of atoms I composed of atoms occurring in the ground version of Π. The ground version of program Π is obtained by substituting in all possible ways variables occurring in Π by constants also occurring in Π. In our case, metavariables occurring in an atom must be substituted by metaconstants, with the following obvious restrictions: a metavariable occurring in the predicate position must be substituted by a metaconstant denoting a predicate; a metavariable occurring in the function position must be substituted by a metaconstant denoting a function; a metavariable occurring in the position corresponding to a constant must be substituted by a metaconstant denoting a constant. According to well-established terminology [12], we therefore require $I \subseteq B_\Pi$, where B_Π is the *Herbrand Base* of Π, given previously-stated limitations on variable substitution. Then, we pose some more substantial requirements. As said before, by $\uparrow A$ we intend a name of base atom A.

Definition 1 *Let Π be a program. $I \subseteq B_\Pi$ is a* potentially acceptable set of atoms *iff for every base atom A which belongs to I, $solve(\uparrow A)$ also belongs to I.*

Definition 2 *Let Π be a program, and I be a potentially acceptable set of atoms for Π. I is an* acceptable *set of atoms iff I satisfies the following axiom schemata for every base atom A:*

$A \leftarrow solve(\uparrow A)$
$\neg A \leftarrow solve\_not(\uparrow A)$

We restrict *SEM* to determine acceptable sets of atoms only, modulo bijection: i.e., *SEM* can be allowed to produce sets of atoms which are in one-to-one correspondence with acceptable sets of atoms. In this way, we obtain the implementation of properties of relations that have been defined via *solve* and *solve_not* rules without modifications to *SEM* for any formalism at hand. For clarity however, it is convenient to filter away *solve* and *solve_not* atoms from acceptable sets. Thus, given a program Π and an acceptable set of atoms I for Π, the *Base version* I^B of I is obtained by omitting from I all atoms of the form $solve(\uparrow A)$ and $solve\_not(\uparrow A)$.

Procedural semantics and the specific naming relation that one intends to use remain to be defined. In fact, it is easy to see that the above-introduced semantics is independent of the naming mechanism. For approaches based upon (variants of) Resolution (like, e.g., Prolog) one can extend the procedure so as to be allowed to use rules with conclusion $solve(\uparrow A)$ to resolve a goal A and, vice versa, rules with conclusion A to resolve $solve(\uparrow A)$; if a goal G succeeds in this way with computed answer θ, then $solve\_not(\uparrow G\theta)$ should be attempted: if it succeeds, then G should be forced to fail; otherwise, success of G can be confirmed.

5 Expressing OWL-Like Properties of Relations

In the previous section we have shown the use of metalevel definitions involving *solve* and *solve_not* to define what it means of a predicate to be reflexive or, vice versa, irreflexive. These metalevel definitions are declarative yet executable, in that they suitably enlarge or restrict the involved predicates' extension. In this section we show the metalevel representation of other properties of relations that can be expressed in OWL. We concentrate in particular on properties which are relevant and widely used. The objective is to convince the reader that most such properties can be represented in our approach without resorting to the powerful though complex Description Logics.

Symmetry can be simply defined as follows:

$solve(P'(X', Y')) \text{:-} symmetric(P'), solve(P'(Y', X')).$
$symmetric(friend').$

This rule specifies in fact the meaning of symmetry for any predicate, stating (via the predicate *solve* applied over a generic atom name) that $p(X, Y)$ can be derived if $p(Y, X)$ holds; notice that in this rule predicate *solve* occurs not only in the head but also in the body of the rule. The fact $symmetric(friend')$ specifies that predicate *friend* is symmetric, via its name.

So, a programmer/knowledge designer behaves very much like in OWL, save that properties of relations must be specified on the names of the predicates. Notice that different metaevaluation rules (with their auxiliary metalevel rules) can be defined and expressed in a modular way, and they naturally interact and compose with each other.

Below we consider *transitivity*, that can be simply expressed in the following way:

$$solve(P'(X', Y')) :- transitive(P'), solve(P'(X', Z')), solve(P'(Z', Y')).$$

This rule specifies the meaning of transitivity for any predicate, where a fact of the form $transitive(P')$ declares that the predicate p' is transitive. The definition actually allows new facts to be derived. For example, if we have the following facts:

$transitive(same\_age')$.
$same\_age(ann, alice)$.
$same\_age(alice, chris)$.

via the previous rule we can derive $same\_age(ann, chris)$. A possible variation is the transitive closure.

Another very useful feature, that increases flexibility to a great extent, is equivalence between properties, obtained by the following definition.

$$solve(P'(X', Y')) :- equivalent(P', R'), solve(R'(X', Y')).$$

This rule defines two relations as equivalent if they have the same extension. For example, stating that predicate *friend* is equivalent to predicate *amico* (the latter is the translation into Italian of the former):

$equivalent(friend', amico')$.
$friend(ann, alice)$.
$symmetric(equivalent')$.

we can easily see that it becomes possible to derive $amico(ann, alice)$. The meta-meta statement $symmetric(equivalent')$ allows the translation to be applied in both ways. The concept of equivalence can be customized via other meta-rules.

Focusing the attention on the concept of inheritance, we may have the following:

$$solve(P'(X', Y')) :- hereditary(P', R'), \\ solve(R'(X', Z')), solve(P'(Z', Y')).$$

meaning that property P' is hereditary with respect to a relation R' if whenever an element of the domain of R' has property P' then also all the other elements have the same property. For example:

$hereditary(polygon', kind\_of')$.
$polygon(quadrilateral, four)$.
$kind\_of(square, quadrilateral)$.

Where $polygon(quadrilateral, four)$ indicates that a quadrilateral is a kind of polygon that has the property of having four sides; instead, $kind\_of(square, quadrilateral)$ indicates that a square is a kind of quadrilateral. Thanks to the *hereditary* rule we can derive $polygon(square, four)$. Other OWL properties, e.g., subclasses etc., can be represented in a similar way.

It is important to notice that in the present setting new properties of relations can be defined upon need and immediately employed, in combination with already existing ones, and meta-meta properties can be also expressed.

6 Background: Answer Set Programming

Answer Set Programming (ASP), is a well-known successful logic programming paradigm (cf. [7] and the references therein). Roughly speaking, an ASP program is a declarative Prolog-like (executable) specification of a problem to be solved. Such a program may have several "models", called "answer sets" (or also "stable models"), each one representing a possible interpretation of the situation described by the program (and, usually, encoding a solution to the problem at hand). ASP has been successfully applied in practice in many application domains.

An answer set program Π (or simply "program") is a finite collection of *rules* of the form $H \leftarrow L_1, \ldots, L_n.$ where H is an atom, $n \geq 0$ and each literal L_i is either an atom A_i or its *default negation* $not\, A_i$. The left-hand side and the right-hand side of rules are called *head* and *body*, respectively. A rule can be rephrased as $H \leftarrow A_1, \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n.$ where $A_1, \ldots, A_m$ can be called *positive body* and $not\, A_{m+1}, \ldots, not\, A_n$ can be called *negative body*. Analogously to Prolog, practical programming environments often offer :− as a glyph for the symbol ←. A rule with empty body ($n = 0$) is called a *unit rule*, or *fact* (vs. non-unit rules). A rule with empty head, of the form $\leftarrow L_1, \ldots, L_n.$, is a *constraint*. It states that the literals $L_1, \ldots, L_n$ cannot be simultaneously true. A rule head can be a disjunction and a "classical negation" is provided; for lack of space we do not consider here such improvements to the basic paradigm.

A program may have several answer sets or may have no answer set (while in many semantics for logic programming a program admits exactly one "model", however defined). Whenever a program has no answer sets, we say that the program is *inconsistent*; so, checking for consistency means checking for the existence of answer sets. Answer sets of Π, if any exists, are consistently supported minimal classical models of the program (interpreted in the obvious way as a first-order theory).

In practice, answer sets can be found via inference engines called *ASP solvers* [25]. Several solvers have became available, each of them being characterized by its own prominent valuable features. As is well-known, most of the commonly available ASP solvers produce the grounding of the given program as a first step, as they are able to find the answer sets of ground programs only.

The ASP programming methodology can be called GCO, for "Guess & Check & Optimize", where: (i) Guess implies generating potential solutions via rules and cycles (ii) Check implies selecting admissible ones by defining suitable constraints; (iii) Optimize implies specifying preference criteria by exploiting *weak constraints*, indicated by connective :∼, that select among the admissible solutions those that satisfy such constraints at best.

ASP has been equipped in time with several additional features, representable via non-trivial ASP subprograms, and implemented directly in ASP solvers. One of these is the so-called *Cardinality Constraints* of the form (that we show for simplicity for the case of binary predicates, though it can be extended to predicates of any arity):

$$n\{p(X,Y) : d(X)\}m :\!-q(Y)$$

meaning that for every (constant value of) Y for which $q(Y)$ holds, every answer set must include no less than n and no more than m (where $n \leq m$) atoms of the form $p(X,Y)$ where $d(X)$ holds (of the constant value assigned to X). Given the possible answer sets originating from the basic definition, those that do not meet such a constraint are discarded. Below is an example of use, stating that every person takes residence in exactly one municipality:

$$1\{\mathit{residence}(X,\, Y) : \mathit{municipality}(X)\}1 :\!-\mathit{person}(\, Y).$$

Cardinality constraints can be seen as special cases of *aggregates*, which have the form

$$n\; op\, [L_1 = w_1, \ldots, L_n = w_n]\, m$$

where the L_i's are literals, the w_is are numerical weights, *op* is an operator (e.g. sum, average, min, max) to be applied to the weights of literals that are true in given set of atoms; in any answer set, the result of the application of the operator must stay within the bounds. For cardinality constraints, the weight of literals is implicitly set to 1 and the operator is implicitly set to sum. Cardinality constraints can reproduce some of the cardinality features of OWL; in particular they can represent the *functional* property of a predicate, basically (like in the above example) by *enforcing* the predicate to assume only one value by setting both n and m to 1. However, ASP has virtually no software engineering construct beyond such constraints; in particular other properties of relations should be specified, if needed, in an "ad hoc" manner, according to the skills of a programmer.

In the rest of the paper, we will discuss how to incorporate into ASP the mechanisms for definition and use of properties of relations that we have outlined before. In this way, predefined definitions are available to every ASP user, while the skilled one can represent her/his favorite new properties.

7 Properties of Relations in ASP

In order to be able to represent knowledge in ASP more easily and in a more understandable and flexible way, the methodology introduced in Sections 4-5 can be usefully employed. However, since ASP is not resolution-based, the methodology must be applied in a different way. In this section, we propose in particular, to compile a set S of metalevel and metaevaluation rules into a form that can be seamlessly added to a given ASP program Π, whose rules can also undergo some easy modifications; we thus obtain an augmented program Π^S where the requirements of Definitions 1 and 2 are satisfied in the answer sets of Π^S, which means that properties of relations specified in S are properly applied. The procedure is defined as follows.

Definition 3 *Given ASP program Π and a set of metalevel and metaevaluation rules S, and assuming that the metalevel constants occurring in S refer to (are names of) predicates and constants occurring in Π, we obtain a new ASP program Π^S from Π and S via the following steps (transformation τ_S).*

(i) For every atom of the form $solve(P'(Args))$ or $solve(p'(Args))$, where $Args$ denotes the set of arguments according to the cardinality of the predicate(s) the atom refers to, such atoms must be transformed into the form $solve(P', Args)$ or $solve(p', Args)$ respectively. The same is done for solve_not, where we obtain $solve\_not(P', Args)$ or $solve\_not(p', Args)$.

(ii) All metavariables are substituted with plain variables, and all metaconstants which are names of constants with the corresponding constants. Metaconstants which are names of predicates are kept untouched. All metarules and solve rules thus obtained are added to Π^S.

(iii) For every predicate p different from solve and solve_not occurring in Π such that p' also occurs in Π, the following pair of rules is added to Π^S:

$$p(X_1, \ldots, X_n) :- solve(p', X_1, \ldots, X_n), \\ not\ solve\_not(p', X_1, \ldots, X_n). \quad (1)$$

$$solve(p', X_1, \ldots, X_n) :- p(X_1, \ldots, X_n), \\ not\ solve\_not(p', X_1, \ldots, X_n). \quad (2)$$

where $X_1, \ldots, X_n$ are variables, n being the arity of predicate p.

(iv) All facts (unit rules) of Π are added to Π^S. For every non-unit rule in Π of the form

$$p(X_1, \ldots, X_n) :- Body.$$

the rule is replaced in Π^S by the modified version:

$$p(X_1, \ldots, X_n) :- Body, not\ solve\_not(p', X_1, \ldots, X_n).$$

Remark 1. The above transformation may need to add auxiliary predicates to ensure *safety* of rules, which is a technical condition required by ASP solvers in order to make the grounding of programs easier. This aspect presents no conceptual or practical problems and so, for the sake of simplicity, is not treated here. ⋄

We are able to prove the following:

Theorem 1 *The answer sets of Π^S correspond to acceptable sets of atoms for Π^S.*

Proof (sketch): All rules in S occur in Π^S, though in a format suitably modified so as to be compatible with ASP syntax and semantics. Apart from the modified notation, the addition of rules (1) guarantees the satisfaction, in every answer set, of the condition specified in Definition 1; the addition of rules (2) and of the additional condition $solve\_not(p', X_1, \ldots, X_n)$ in all the other rules enforce the satisfaction, in every answer set, of the conditions specified in Definition 2. ⋄

For the sake of clarity let us apply the above definition to the previous example (also considering symmetry). To summarize, the set S is the following:

$solve(P'(X', X')) :- reflexive(P'), in\_domain(P', X').$
$in\_domain(P', X') :- solve(P'(X', Y')).$
$in\_domain(P', X') :- solve(P'(Y', X')).$
$solve(P'(X', Y')) :- symmetric(P'), solve(P'(Y', X')).$
$solve\_not(P'(X', X')) : - irreflexive(P').$
$reflexive(same\_age').$
$irreflexive(friend').$
$symmetric(friend').$

Program Π is simply the following:

$friend(george, ann).$
$same\_age(ann, alice).$

From steps (i) and (ii) we obtain the following S':

$solve(P, X, X) :- reflexive(P), in\_domain(P, X).$
$in\_domain(P, X) :- solve(P, X, Y).$
$in\_domain(P, X) :- solve(P, Y, X).$
$solve(P, X, Y) :- symmetric(P), solve(P, Y, X).$
$solve\_not(P, X, X) : - irreflexive(P).$
$reflexive(same\_age').$
$irreflexive(friend').$
$symmetric(friend').$

Finally, we obtain S'' by adding the following rules to S':

$same\_age(X, Y) :- solve(same\_age', X, Y),$
$\quad not\ solve\_not(same\_age', X, Y).$
$solve(same\_age', X, Y) :- same\_age(X, Y),$
$\quad not\ solve\_not(same\_age', X, Y).$
$friend(X, Y) :- solve(friend', X, Y),$
$\quad not\ solve\_not(friend', X, Y).$
$solve(friend', X, Y) :- friend(X, Y),$
$\quad not\ solve\_not(friend', X, Y).$

Π^S is obtained by adding the rules in S'' to the given program Π, that in this case does not need modifications, as it is composed of facts only. It can be verified, by running Π^S via any ASP solver, that its answer sets bring the desired results (to perform experiments, metaconstants of the form p' must be given a syntax compatible with ASP solver, for instance $p1$). The resulting (in this case unique) answer set is:

$\{in\_domain(same\_age', alice), in\_domain(friend', ann),$
$in\_domain(same\_age', ann), in\_domain(friend', george),$
$symmetric(friend'), irreflexive(friend'),$
$reflexive(same\_age'), solve(friend', george, ann),$
$solve(same\_age', ann, alice), solve(friend', ann, george),$
$solve(same\_age', ann, ann), solve(same\_age', alice, alice),$
$same\_age(ann, alice),$
$same\_age(alice, alice), same\_age(ann, ann),$
$friend(george, ann), friend(ann, george)\}$

Let us notice explicitly that the above formulation is applicable to every program Π, not just to programs composed of facts only, but to any program, even including cycles, and thus featuring several answer sets. For example, if we add to the above sample program the rules and facts:

$friend(X, Y) :- nice(Y), not\ enemy(X, Y).$
$enemy(X, Y) :- not\ friend(X, Y).$
$nice(alice).$
$irreflexive(enemy').$

The resulting program, as it is easy to verify, has several answer sets, where *george*, *ann* and *alice* are either mutual enemies or friends of *alice*, the only one declared to be nice; however, due to the stated irreflexivity of predicate *enemy* (in addition of what already done for *friend*) none of them three is either friend or enemy of her/himself.

We introduce a simplified version for the answer sets of Π^S:

Definition 4 *Given an ASP program Π, a set of metalevel and metaevaluation rules (meta-definitions) S and the program Π^S obtained from the former according to Definition 3, the* Base version I^B *of an answer set I is obtained by omitting from I all atoms of the form $solve(p', Args)$ or $solve\_not(p', Args)$.*

As for the modality of use of the approach, a user should write program Π, and declare the desired properties of relations to exploit (in the previous example, facts $reflexive(same\_age')$, $irreflexive(friend')$, and $symmetric(friend')$) that by abuse of notation might be expressed directly on predicate names, with a pre-processor to be responsible of re-arranging the notation. A smart programmer/knowledge engineer might enrich the predefined set S by writing new metalevel and metaevaluation rules, in the user-friendly form illustrated in previous sections, where Π^S would then be generated by a pre-processor.

Remark 2. Transformation τ_S over program Π and set S of meta-definitions adds two new rules for each predicate p occurring in Π such that p' occurs in S. This would at worst multiply by three the size of Π when computing S'', and consequently also the size of the corresponding grounded program. This is however a pessimistic esteem, because an "ad hoc" definition of properties such as symmetry, transitivity, etc. for specific predicates would in any case imply adding at least one additional rule per property per predicate. Thus, there is

indeed an increase in size w.r.t. given program, but it is not dramatic and can be considered as a reasonable drawback in exchange for the extra expressiveness. Complexity remains the same, so there is no additional computational burden due to the approach. $\diamond$

8 Related Work and Conclusions

It is worth considering if the metaevaluation part S of given program, or more precisely the ASP transposition S' obtained according to Definition 3, might be encapsulated within either a template [10] or a module [8, 9]. This can be possible only if the specific approach to modular/template ASP allows recursive (direct or indirect) call of templates/modules because metalevel properties can be combined: see, e.g., the example where a symmetric predicate is equivalent to another one, where equivalence is itself symmetric. To the best of our understanding, [10] would require the specific definition of each predicate with its properties and their specification to be enclosed into a template. [8] would allow to augment the definition of a predicate via a module defining the specific metalevel part; [9] allows instead predicates as module arguments, so there might be a unique common definition.

The HEX framework [26, 27] allows to combine ASP with Description Logics (or other formalisms). Clearly, external reasoners and knowledge bases are assumed to be available. For properties of relations, their reification approach would require one so-called 'HEX rule' for each predicate p with arity n which might enjoy some properties, of the form: $p(T_1, \ldots, T_n) :- \&ext[p](T_1, \ldots, T_n)$. Then, for the external evaluation of "$\&ext[p]$", one should resort to an OWL external ontology where to define the properties of p and of other predicates. The HEX approach is certainly very useful for integrating various forms of knowledge representation and reasoning. For properties of predicates such as those discussed here it appears however over-dimensioned and certainly less efficient (having to resort to an OWL reasoner) than our simple implementation.

Moreover, in all the above-mentioned frameworks it would be required to know in advance if some properties can be applicable to a predicate, so as to invoke the relative module(s) or to write the related HEX rule. Instead, if S' is imported as a library and an associated pre-processor generates S'', metalevel properties smoothly enlarge/restrict the extension of each predicate to which they are applicable. Different properties are combined automatically with no effort required to the programmer.

In conclusion, we have introduced a methodology based on naming and metareasoning for enriching logic-based formalisms with the possibility of expressing and using metalevel properties of relations similarly to what done in the OWL ontology language. Computationally, the methodology has the advantage of not requiring higher-order features. From the knowledge representation point of view, it allows the introduction of both predefined and user-defined properties, so it can increase the usability and flexibility of virtually any knowledge representation and reasoning architecture at very little cost for implementation,

and little burden for knowledge engineers. We have shown that the methodology is usable in both resolution-based frameworks but also in formalisms such as ASP which are based on a very different computational engine.

Related work exists about ontologies and ASP. [28] shows that RDF(S) ontologies can be expressed, without loss of semantics, into Answer Set Programming. Then, based on a previous result showing that the SPARQL query language (a candidate W3C recommendation for querying ontologies) can be mapped to a rule based language with stable model semantics, it shows that efficient querying of big ontologies can be accomplished within an extension of the well known ASP system DLV [25]. The difference with our work is that we do not intend to query external ontologies, rather we show that ontological reasoning can be accomplished *within* a logic program. The DLVHEX system [29] is a logic-programming reasoner for computing the models of so-called HEX-programs. In this approach ASP programs are extended to become higher-order logic programs, which accommodate meta-reasoning through higher-order atoms, and with external atoms for software interoperability. For instance, a rule may look like the following one, with variables ranging over predicates:

$$C(X) :- subClassOf(D, C), D(X).$$

Although we are not able to query external ontologies (so far, as we might adopt for instance the method of [28]), we are able to perform the same kind of meta-reasoning within the traditional ASP semantics, which is in our opinion an added value.

References

1. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: the making of a web ontology language. J. Web Sem. **1**(1) (2003) 7–26
2. Antoniou, G., van Harmelen, F.: Web ontology language: OWL. In: Handbook on Ontologies. International Handbooks on Information Systems. Springer (2009) 91–110
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge Univ. Press (2003)
4. Konolige, K.: Reasoning by introspection. In: Meta-Level Architectures and Reflection. North-Holland (1988) 61–74
5. van Harmelen, F., Wielinga, B., Bredeweg, B., Schreiber, G., Karbach, W., Reinders, M., Voss, A., Akkermans, H., Bartsch-Spörl, B., Vinkhuyzen, E.: Knowledge-level reflection. In: Enhancing the Knowledge Engineering Process – Contributions from ESPRIT. Elsevier Science (1992) 175–204
6. Barklund, J., Dell'Acqua, P., Costantini, S., Lanzarone, G.A.: Reflection principles in computational logic. J. Log. Comput. **10**(6) (2000) 743–786
7. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12) (2011) 92–103
8. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In Procòf ICLP'06, Vol.4079 of LNCS., Springer (2006) 376–390

9. Krennwallner, T.: Promoting modular nonmonotonic logic programs. In ICLP'11 Tech. Comm. Vol.11 of LIPIcs, Schloss Dagstuhl Leibniz-Zentrum für Informatik (2011) 274–279
10. Calimeri, F., Ianni, G., Ielpa, G., Pietramala, A., Santoro, M.C.: A system with template answer set programs. In: Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Pr. Vol.3229 of LNCS., Springer (2004) 693–697
11. Costantini, S.: Meta-reasoning: A survey. In: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. Vol.2408 of LNCS., Springer (2002) 253–288
12. Lloyd, J.W.: Foundations of Logic Programming, Second Edition. Springer (1987)
13. Bowen, K.A., Kowalski, R.A.: Amalgamating language and metalanguage in logic programming. In: Logic Programming. Academic Press, London (1982) 153–172
14. Carlucci Aiello, L., Levi, G.: The uses of metaknowledge in AI systems. In: Meta-Level Architectures and Reflection. North-Holland (1988) 243–254
15. Bonatti, P.: Model theoretic semantics for demo. In: Meta-Programming in Logic. LNCS 649, Berlin, Springer (1992) 220–234
16. Hill, P.M., Lloyd, J.W.: The Gödel Programming Language. The MIT Press (1994)
17. Grosof, B.N., Kifer, M., Fodor, P.: Rulelog: Highly expressive semantic rules with scalable deep reasoning. In Doctoral Consortium, Challenge, Industry Track, Tutorials and Posters@RuleML+RR'17 hosted by RuleML+RR'17. Vol.1875 of CEUR Workshop Pr., CEUR-WS.org (2017)
18. Chen, W., Kifer, M., Warren, D.S.: HILOG: A foundation for higher-order logic programming. J. Log. Program. **15**(3) (1993) 187–230
19. Bordini, R.H., Braubach, L., Dastani, M., El Fallah-Seghrouchni, A., Gomez-Sanz, J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica (Slovenia) **30**(1) (2006)
20. van Harmelen, F.: Definable naming relations in meta-level systems. In: Meta-Programming in Logic. LNCS 649, Berlin, Springer (1992) 89–104
21. Barklund, J.: What is a meta-variable in Prolog? In: Meta-Programming in Logic Programming. The MIT Press (1989) 383–98
22. Hill, P.M., Lloyd, J.W.: Analysis of metaprograms. In: Meta-Programming in Logic Programming, THE MIT Press (1988) 23–51
23. Barklund, J., Costantini, S., Dell'Acqua, P., Lanzarone, G.A.: Semantical properties of encodings in logic programming. In: Logic Programming – Proc. 1995 Intl. Symp., MIT Press (1995) 288–302
24. Dix, J.: A classification theory of semantics of normal logic programs: I. Strong properties. Fundam. Inform. **22**(3) (1995) 227–255
25. ASP: Answer set programming solvers (incomplete list) (2018) DLV: `www.dlvsystem.com`. WASP: `www.mat.unical.it/DLV2/wasp`. clingo: `www.potassco.org`. lparse+smodels: `www.tcs.hut.fi/Software/smodels`.
26. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Redl, C., Schüller, P.: A model building framework for answer set programming with external computations. TPLP **16**(4) (2016) 418–464
27. Eiter, T., Kaminski, T., Redl, C., Schüller, P., Weinzierl, A.: Answer set programming with external source access. In: Reasoning Web. Semantic Interoperability on the Web - 13th Int. Summer School 2017, Tutorial Lectures. Vol.10370 of LNCS., Springer (2017) 204–275
28. Ianni, G., Martello, A., Panetta, C., Terracina, G.: Efficiently querying RDF(S) ontologies with answer set programming. J. Log. Comput. **19**(4) (2009) 671–695
29. Eiter, T., Germano, S., Ianni, G., Kaminski, T., Redl, C., Schüller, P., Weinzierl, A.: The DLVHEX system. KI **32**(2-3) (2018) 187–189