Comments on Modeling Languages for Answer-Set Programming

Mirosław Truszczyński

Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA, mirek@cs.uky.edu

Abstract. Strong emphasis on intuitive and direct modeling of application domains is one of the distinguishing features and major strengths of the answer-set programming paradigm. It leads naturally to several key questions. Is there a need for standardizing such languages? What functionality should these languages support? Are there any general design requirements for them? This note attempts to propose some answers.

1 Introduction

Answer-set programming (ASP) is a paradigm for declarative programming. Speaking informally, in ASP a problem is modeled as a theory in some language of logic. This representation is designed so that once expanded with an encoding of particular instance of the problem, it results in a theory whose *models* correspond to solutions to the problem for this instance [13, 14].

Thus, the main automated reasoning task in support of the ASP paradigm is computing models of theories. A typical approach is to *ground* a theory representing a problem and its particular instance or, in other words, to *compile* the "program" and the "data" into a low-level representation. The result of this step is a propositional theory that has the same models as the original one. They are computed in the last step of the process by programs called *solvers*.

This overview of the ASP process shows that when solving a problem one deals with a theory in several different formats. First, there is a format determined by the modeling language. Second, there is a format of the grounded (propositional) version of this theory. Finally, there is "solver" format, a version of the ground theory in a format accepted by solvers. A central issue to the design and development of software tools in support of ASP is that of standards for theory formats at every stage of the process.

My goal in this note is to address the matter of standards for ASP modeling languages. I argue that no specific standards are necessary. Instead I present several "desiderata" that should be taken into account when designing ASP modeling languages.

2 What is Answer-Set Programming?

In most general terms, ASP is a paradigm for modeling and solving search problems. In order to talk about ASP and issues related to software tools for ASP, it will be convenient to introduce first a formal setting, in which search problems can be defined and studied.

2.1 Search problems

In formal definitions of search problems, one typically assumes a fixed infinite countable set U, referred to throughout the note as the *universe*. A *signature* is a nonempty set σ of relation symbols r, each with a positive integer arity k_r. An *instance* of a signature σ is a pair $I = \langle D, R \rangle$, where $D \subseteq U$ is a finite set called the *domain* of I, $dom(I)$ in symbols, and

$$R = \{r^I : r \in \sigma \text{ and } r^I \subseteq D^{k_r} \text{ is a relation of arity } r\}.$$

Throughout the note, $Inst_\sigma$ will stand for the set of all instances of σ.

For two *disjoint* signatures σ^i and σ^s, a *search problem over* (σ^i, σ^s) is a *recursive* set $\Pi \subseteq Inst_{\sigma^i} \times Inst_{\sigma^s}$ such that for every $(I, S) \in \Pi$, $dom(I) = dom(S)$. Elements of $Inst_{\sigma^i}$ are *instances* of Π. If $I \in Inst_{\sigma^i}$ then every $S \in Inst_{\sigma^s}$ such that $(I, S) \in \Pi$ is a *solution* to Π for I.

Typically, given a search problem $\Pi \in Inst_{\sigma^i} \times Inst_{\sigma^o}$ and its instance $I \in Inst_{\sigma^i}$, the objective is to find a solution to Π for I. From the practical point of view, there are two crucial issues: how to *model* search problems — one must be able to specify them in order to solve them, and how to *find a solution* given a problem specification and an instance. Answer-set programming is a paradigm that addresses both issues.

2.2 Modeling search problems

Let $\mathcal{L}_\sigma$ be some logic language over σ. For now, I specify neither the set of boolean connectives of the language nor its set of well-formed formulas. The only assumption I make is that there is a *recursive* relation $\models$ that holds between instances in $Inst_\sigma$ and formulas in $\mathcal{L}_\sigma$. For example, if $\mathcal{L}_\sigma$ is the language of first-order logic (under our definition of σ — with no constant or function symbols), one could choose for $\models$ the standard satisfiability relation between a structure and a formula. If $\mathcal{L}_\sigma$ is the language of logic programs, where formulas are conjunctions of program rules, one might define $I \models \varphi$ to hold if I is a stable model (an answer set) of φ.

If $\sigma' \subseteq \sigma$ are signatures, then $K \in Inst_\sigma$ *expands* $I \in Inst_{\sigma'}$, written as $I = K_{|\sigma'}$, if $dom(K) = dom(I)$ and for every $r \in \sigma'$, $r^I = r^K$. Let σ^i and σ^s be two disjoint signatures such that $\sigma^i \cup \sigma^o \subseteq \sigma$. Every formula $\varphi \in \mathcal{L}_\sigma$ gives rise to a search problem

$$\Pi_\varphi = \{(K_{|\sigma^i}, K_{|\sigma^s}) : K \in Inst_\sigma \text{ and } K \models \varphi\}.$$

Indeed, $dom(K_{|\sigma^i}) = dom(K_{|\sigma^s})$ (each is equal to $dom(K)$) and, since $\models$ is a recursive relation, Π_φ is a recursive set.

3 A minimal requirement for an ASP language

The discussion so far implies that every logical language, for which there is a recursive satisfiability relation $\models$ between instances and formulas provides a way to specify search problems. In other words, it can be regarded as an ASP modeling language.

How good a modeling tool an ASP language is depends to a large degree on the expressive power of the language — the class of search problems that are defined by formulas in the language in the way described above. One could argue that at the very least the expressive power of an ASP modeling language should be given by the class NPMV [18], as this class contains search problems of practical importance. In particular, all decision problems in the class NP (once they are recast as search problems) belong to the class NPMV. The class NPMV is also known as the class *NP-search*, the term I prefer as it makes a direct reference to search problems.

Of course, to be of practical use, the language also needs to be *implemented*, that is, come with a way to specify signatures, instances and formulas in terms of expressions that can be processed by computers, as well as with tools to compute solutions given a problem description as a formula and its input specified as an instance to the problem.

These comments suggest the following *minimal* requirement for ASP modeling languages.

> *An ASP modeling language is a language of logic with a recursive satisfiability relation $\models$ between signature instances (structures) and formulas, and with the expressive power equal at least to that of the class NP-search. The language comes with an implementation — software that allows one to code problem and instance specifications and, given the encodings, compute solutions (or determine that none exists).*

I do not think there is a need for any standardization of ASP modeling languages beyond this basic requirement. However, there are several considerations that should be taken into account when developing and evaluating ASP systems. Before discussing them, I introduce two examples of ASP modeling languages.

4 Two examples

The most studied and widely used ASP modeling language is the language of logic programming with the stable-model semantics [11, 13, 14]. In this formalism, problems are modeled as logic program. For example, the graph 3-colorability problem can be specified by the following program P_{col}:

```
b(X) :- vtx(X), not r(X), not g(X).
r(X) :- vtx(X), not b(X), not g(X).
g(X) :- vtx(X), not r(X), not b(X).
:- edge(X,Y), b(X), b(Y).
:- edge(X,Y), r(X), r(Y).
:- edge(X,Y), g(X), g(Y).
```

This program is presented in a format that is accepted by implementations of logic programming as an ASP system such as *lparse/smodels* [15, 19] and *dlv* [8, 12]. Each line lists a program rule — a single conjunct of the program. Commas in the bodies of rules stand for the conjunction and `not` represents the negation (to be exact, the negation-as-failure). The empty head stands for the contradiction. The program defines implicitly the signature σ (in this case consisting of relation symbols b, r, g, vtx and $edge$), as well as the signature σ^i, which consists of those symbols that do not appear in the heads of rules (symbols r, b and g),

An instance to the problem is a set of *ground* atoms of the form $vtx(x)$ and $edge(x, y)$ defining an input graph. The domain is defined implicitly as the set of all constants used in the description of the instance.

The program P_{col} is indeed an encoding of the graph 3-colorability problem due to the property that for every input instance I, instances of the signature $\{b, r, g, vtx, edge\}$ expanding the input instance and such that they are stable models of $P_{col} \cup I$ determine solutions to the problem. That is, extensions of the relations corresponding to b, r and g in a stable model of $P_{col} \cup I$ form a proper 3-coloring of the graph represented by I, and every proper 3-coloring has a representation as a stable model of $P_{col} \cup I$.

Another language that received some attention is based on the *logic of propositional schemata* [7]. In this logic, a basic formula is an implication with a conjunction of atoms in the antecedent and the disjunction of atoms (possibly existentially quantified) in the consequent. Search problems are represented as conjunctions (lists) of formulas in this elementary syntax. In particular, the graph 3-colorability problem can be specified as the conjunction of the following formulas:

```
vtx(X) -> r(X) | b(X) | g(X).
r(X), b(X) -> .
r(X), g(X) -> .
b(X), b(X) -> .
edge(X,Y), b(X), b(Y) -> .
edge(X,Y), r(X), r(Y) -> .
edge(X,Y), g(X), g(Y) -> .
```

Also in this case, the program is given in the format accepted by an implementation of the logic of propositional schemata [7]. In particular, commas in the antecedents represent the conjunction connective, `->` and `|` stand for the implication and disjunction, respectively. As before, the empty consequent represents the contradiction.

In the logic of propositional schemata signatures and input instances need to be defined explicitly. Similarly as for the logic programming representation, instances expanding an input instance and such that they are models of T_{col} correspond to 3-colorings of the graph represented by the instance.

Both logic programming and the language PS can express the whole class NP-search and each has been implemented. Thus, they satisfy the basic requirement identified above. I note that a variant of logic programming, *disjunctive* logic programming, captures a wider class of problems — the class Σ_2^P-search (wider, assuming the polynomial hierarchy does not collapse) and also has an implementation (for instance, system *dlv* [9, 12]).

5 Other requirements for ASP languages

These two examples of ASP modeling languages examples are simple and presented here without much detail. Nevertheless they bring up several important points.

Definitions. Due to the KR roots of logic programming with the answer-set semantics, ASP languages based on this formalism can handle effectively the *problem of definitions*. Let us suppose, that p holds precisely when both q and r hold or when both s and t hold. In LP languages, this definition can be stated in terms of two clauses:

```
p :- q, r.
p :- s, t.
```

In the language of propositional schemata, specifying this simple definition is much less concise — one needs to express in a CNF representation the formula $p \leftrightarrow (q_1 \wedge q_2) \vee (r_1 \wedge r_2)$. It can be done, for instance, as follows:

```
q, r -> p.
s, t -> p.
p -> q | s.
p -> q | t.
p -> r | s.
p -> r | t.
```

This is a more complex representation. Moreover, as the number of cases under which p holds grows, the complexity of the CNF representation may grow exponentially. To control this growth one typically introduces new symbols to the language.

In the case when p has a recursive definition matters get still more interesting. The definition of an answer-set involves a fixpoint construction and so LP languages support concise and direct definitions of relations that are closures of other relations. For instance, the following simple program defines the closure `path` of a relation `arc`,

```
path(X,X) :- arc(X,Y).
path(X,X) :- arc(Y,X).
path(X,Y) :- arc(X,Z), path(Z,Y).
```

No such simple definitions of the closure of a relation is known in terms of the logic of propositional schemata, where one needs to introduce several auxiliary predicates in order to build a representation [7].

The importance of definitions in knowledge representation is broadly recognized. Recent work on *ID-logic* [2, 4] demonstrates convincingly that providing means to model definitions is central to effective knowledge representation. These arguments extend to ASP and give rise to the following requirement.

> *An ASP modeling language should offer means for concise and direct representations of definitions and inductive definitions.*

With respect to this postulate, LP languages score well and the language PS scores poorly. Extending the language PS and, more generally, other languages based on first-order logic, with inductive definitions [4, 6, 3] addresses the shortcoming. I claim that this "definition-based" approach to ASP has substantial promise and deserves attention. On the one hand, it explicitly subsumes the language PS , on the other hand, it allows for straightforward and direct encodings of logic programs.

Basic syntax. There are two major considerations one needs to have in mind when deciding on the basic syntax of ASP languages. First, operators supported by the language should reflect typical structure of problem statements given in natural language. This is a "modeling" consideration. The second consideration is "computational". The syntax of an ASP language must be attuned to available tools for processing programs and, most importantly, computing solutions. Currently, these tools are based on DPLL-type backtracking search. In some cases they actually are SAT-solvers implementing the DPLL procedure. The effectiveness of DPLL-type backtracking search depends to a large degree on the effective unit propagation. The simpler the syntax of rules, the stronger propagation methods one can apply, leading to better performance of solvers. These considerations suggest the following postulate:

> *The basic syntax of ASP languages should be rooted in the notion of a clause — a conjunction of literals implying a disjunction of atoms.*

All LP languages and the language PS support formulas that are conjunctions of clauses. The restriction to clauses does not pose any major problems for LP languages. However, for the language PS, the restriction to clauses may make modeling even non-recursive definitions difficult. One could alleviate the problem to some degree by allowing additional connectives to the language, specifically, the "if and only if " connective. This approach does not address the problem in general (in particular, the problem of inductive definitions). Thus, extensions of the language PS with Horn rules or logic programs, as discussed above, may be a better solution.

On the other hand, the language PS is directly aligned with the syntax accepted by SAT-solvers. Due to dramatic advances of SAT-solver technology, it is a major advantage. Whenever specification of a search problem do not require modeling the closure operation, the language PS might be the right modeling tool.

A formalism that takes full advantage of the syntax of clauses as defined above is that of disjunctive logic programming. Disjunctive program rules are implications, where the antecedent is a conjunction of atoms and negation-as-failure literals, the consequent is a disjunctions of atoms. The two formalisms discussed above either do not allow negation in the antecedent or disjunctions in the consequent. With the semantics of answer sets, the disjunctive logic programming is an effective knowledge representation formalism and the basis for the *dlv* [9, 12], one of the most advanced ASP systems. Two important features of this formalism are explicit means to model indefinite information (through disjunctions) and its expressive power given by the class Σ_2^P-search.

Support for externally evaluated relations and functions. Most ASP languages support built-in integer arithmetic operations and integer arithmetic comparison relations. They also support the equality relation over the domain of problem instances. These modeling features of ASP languages turned out to be crucial for concise encodings of problems of practical interest.

The benefits of built-in functions and relations can be expanded to custom-built relations and functions coded in programming languages external to an ASP language. In this way programmers are able to delegate simple computational tasks that are hard to capture in a declarative fashion to much more effective procedural languages. Such functionality is, for example, available in the *lparse/smodels* system. This discussion brings up the following requirement.

An ASP modeling language should have support for external evaluation of relations and functions.

Aggregates. Aggregates in the form of cardinality and weight atoms were introduced to ASP by *lparse/smodels* system. Experiments demonstrated that constraints specifying search problems often involve aggregates. Expanding the syntax of an ASP language with aggregates often allows us to design representations of search problems that are direct, intuitive and concise. Importantly, it turns out that computational tools developed for programs without aggregates can be generalized to the case with aggregates. Moreover, due to significant decrease in the size of the representation and some new propagation methods, the overall performance improves substantially. I feel that providing the functionality of aggregate operations is one of the most crucial requirements for ASP:

An ASP modeling language must provide support for aggregate operations.

At present all ASP modeling languages provide some level of support for aggregates. However, there are significant differences in the syntax and, on the side of LP languages, some differences in the semantics of aggregates [19, 1, 10, 16, 17]. As for approaches stemming from the language PS and ID-logic, support for cardinality and weight atoms is provided by the implementation of the logic PS+ [5]. There are no semantic difficulties though, as long as aggregates do not appear in the definitions.

Optimization and preferences. Most problems of interest are not plain search problems, where *any* solution satisfying constraints will do. In most cases, there are preferences that users have and optimization criteria that they take into account.

An ASP modeling language must have means to specify user preferences, goal functions, and optimization criteria.

Some current ASP languages provide support for preferences and optimization. Most comprehensive approach is implemented by the *dlv* system. A more narrow approach, focusing on optimization of linear goal functions is available in *lparse/smodels*. Nevertheless, I feel this is an area where the field has not bridged the gap between theoretical studies of preferences (there is a vast literature on the subject, much of it devoted to preferences in logic programming) and practical implementations. Addressing the problem of preferences in ASP modeling languages is one of the main problems for the field.

Interoperability with databases. ASP languages can be regarded as query languages for deductive database systems. In fact, much of the interest in logic programs with negation came from the database community.

There are several reasons to do it. Let us consider a database of employees. The goal is to select a team of at most five with particular skills and satisfying some additional constraints (preventing some pairs of employees from being included together in a team, ensuring that some skills are adequately represented, etc.). It may be the case that the selection has to be repeated with some regularity and that the set of employees in the company changes with time, the changes being reflected in a database. In this scenario, an ASP modeling language should support accessing the company database, posing a query to extract tables specifying data needed for the team selection and, finally, modeling the constraints and criteria to be used in the selection. Other applications might concern data integration, and query processing in case of data inconsistency. These comments serve as a justification for the following postulate:

An ASP modeling language must provide support for interactions with database systems.

This postulate was one of the main principles guiding the development of the *dlv* system. As a result, the *dlv* has all the functionality needed for the effective interoperability with database systems.

6 Summary

This note presents a personal look at the problem of designing ASP modeling languages. I identified one general fundamental requirement related to the fact that the main goal of ASP modeling languages is to offer ways to express search problems. I also put forth several other postulates, based on the current state-of-the-art in ASP systems.

There are several issues that I have not discussed here but that are of importance to ASP modeling languages. I will now mention two of them. First, there is a problem of ASP program development tools. As the complexity of applications grows, it becomes acutely clear that they are necessary. Second, there is a problem of expressing the syntax of ASP programs within the framework of the

Rule Markup Initiative (cf. http://www.ruleml.org/). The problem has received some some attention (cf. http://www.kr.tuwien.ac.at/staff/roman/aspruleml/). Nevertheless, it seems to me much remains to be done, especially that the effort I mentioned has focused only on ASP languages based on the logic programming formalism.

Acknowledgments

The author acknowledges the support of NSF grant IIS-0325063 and KSEF grant 1036-RDE-008.

References

1. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and Gerald Pfeifer, *Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV*, Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003), Morgan Kaufmann, 2003, pp. 847–852.
2. M. Denecker, *The well-founded semantics is the principle of inductive definition*, Logics in Artificial Intelligence (J. Dix, L. Fariñas del Cerro, and U. Furbach, eds.), vol. 1489, Springer, 1998, pp. 1–16.
3. M. Denecker and E. Ternovska, *A logic for non-monotone inductive definitions*, ACM Transactions on Computational Logic (2008), To appear.
4. Marc Denecker, *Extending classical logic with inductive definitions.*, Computational Logic - CL 2000, Lecture Notes in Computer Science, vol. 1861, Springer, 2000, pp. 703–717.
5. D. East, M. Iakhiaev, A. Mikitiuk, and M. Truszczyński, *Tools for modeling and solving search problems*, AI Comunications **19(4)** (2006), 301–312.
6. D. East and M. Truszczyński, *Datalog with constraints*, Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), AAAI Press, 2000, pp. 163–168.
7. D. East and M. Truszczyński, *Predicate-calculus based logics for modeling and solving search problems*, ACM Transactions on Computational Logic **7** (2006), 38–83.
8. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello, *A deductive system for non-monotonic reasoning*, Logic programming and nonmonotonic reasoning (Dagstuhl, Germany, 1997), Lecture Notes in Computer Science, vol. 1265, Springer, 1997, pp. 364–375.
9. ———, *A KR system* `dlv`*: Progress report, comparisons and benchmarks*, Proceeding of the 6th International Conference on Knowledge Representation and Reasoning (KR-1998), Morgan Kaufmann, 1998, pp. 406–417.
10. Wolfgang Faber, Nicola Leone, and Gerald Pfeifer, *Recursive aggregates in disjunctive logic programs: Semantics and complexity.*, Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004), LNAI, vol. 3229, Springer, 2004, pp. 200 – 212.
11. M. Gelfond and V. Lifschitz, *The stable semantics for logic programs*, Proceedings of the 5th International Conference on Logic Programming, MIT Press, 1988, pp. 1070–1080.

12. 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 **7(3)** (2006), 499–562.
13. V.W. Marek and M. Truszczyński, *Stable models and an alternative logic programming paradigm*, The Logic Programming Paradigm: a 25-Year Perspective (K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, eds.), Springer, Berlin, 1999, pp. 375–398.
14. I. Niemelä, *Logic programming with stable model semantics as a constraint programming paradigm*, Annals of Mathematics and Artificial Intelligence **25** (1999), no. 3-4, 241–273.
15. I. Niemelä, P. Simons, and T. Syrjänen, *SLP solver* smodels, 1997, http://www.tcs.hut.fi/Software/smodels/.
16. N. Pelov, *Semantics of logic programs with aggregates*, PhD Thesis. Department of Computer Science, K.U.Leuven, Leuven, Belgium (2004).
17. N. Pelov, M. Denecker, and M. Bruynooghe, *Well-founded and stable semantics of logic programs with aggregates*, Theory and Practice of Logic Programming (2006), Accepted (available at http://www.cs.kuleuven.ac.be/ dtai/projects/ALP/TPLP/).
18. A. Selman, *A taxonomy of complexity classes of functions*, Journal of Computer and System Sciences **48** (1994), no. 2, 357–381.
19. P. Simons, I. Niemelä, and T. Soininen, *Extending and implementing the stable model semantics*, Artificial Intelligence **138** (2002), 181–234.