Answer Set Programming – a Domain in Need of Explanation
A Position Paper

Martin Brain and Marina De Vos

Department of Computer Science,
University of Bath,
United Kingdom
{mjb,mdv}@cs.bath.ac.uk

Abstract. This paper describes the problems with debugging tools for answer set programming, a declarative programming paradigm. Current approaches are difficult to use on most applications due to the considerable bottlenecks in communicating the reasons to the user. In this paper we examine the reasons for this and suggest some possible future directions.

1 Introduction

One of the long term goals of computer science is creating efficient declarative programming systems. In an ideal world, the user would create a description of what the problem is and the programming system would work out how to solve it, and return the answer. However, as any student of software engineering will note, what the user *wants* and what the user *asks for* are often different. In the context of declarative programming languages and other 'intelligent' systems, this results in the user trying to understand why the system gave the answer it did. Thus, most practical declarative programming systems rapidly develop tools for explaining why they gave a particular answer. These are normally referred to as 'debugging' tools; although they are only superficially related to most procedural debugging tools. Given the strong formal semantics underpinning most declarative programming systems, building a tool to compute (online or offline) the formal argument behind an answer is relatively straight-forward. Although the names vary with the problem domain; proof trees, refutations, traces, arguments and so on, all follow a similar methodology of building a formal structure to explain their answers based on the rules that give the semantics of the language. This is, of course, widely known and implemented, what is much rarely discussed is how to explain this formal, structured information to the user. As the complexity of the declarative program rises, the resultant proof structure grows, often polynomially but in some cases exponentially; resulting in a considerable bottleneck in conveying this information to the user, which often renders these tools unusable on anything more than toy examples.

This paper discusses the problems resulting from using these 'explanations' for declarative debugging in the context of Answer Set Programming (ASP); a logical, model based, declarative programming paradigm. ASP is a particularly strong example of these issues as the semantics make it easy to compute the (formal) reasons for a given model, the syntax supports a number of natural ways of outputting this information and the models in most typical applications are large enough to make the obvious approaches to communicating explanations impractical. Section 2 describes the logical language, *AnsProlog*, used in ASP, Section 3 describes how it is used, Section 4 outlines the literature on debugging tools and the computation of reasons and Section 5 discusses the practicalities of using these, what the problems are and proposes possible approaches to creating a solution.

2 Answer Set Semantics

Answer set semantics [13] is a model based semantics for logic programs. Programs written in *AnsProlog*[1] are unordered[2] sets of rules. Rules are made from *atoms*, indivisible propositions which can either be known or not known, and take the form of basic causal laws. For example:

$$a \leftarrow b, \text{not } c.$$

is interpreted as "if b is known and c is not known then a is known". The atom a is the *head* of the rule, denoted $H(r)$ and $\{b, \text{not } c\}$ is the *body* of the rule, written $B(r)$. The body is then divided into the *negative body*, $B^-(r)$, the atoms that are negated and the *positive body*, $B^+(r)$, the normal atoms in the body. In the preceding example $B^-(r) = \{c\}$ and $B^+(r) = \{b\}$. Rules with empty bodies are referred to as *facts*.

The semantics of positive programs (programs Π which do not contain negation, i.e. $\forall r \in \Pi \,.\, B^-(r) = \emptyset$), are relatively straight-forward and uncontroversial. Starting with the empty set, an immediate consequence operator is applied until a fixed point is reached. This matches the intuition of starting with no knowledge/assumptions and then only 'learning' information when the conditions (body) of a rule are met. More formally, this is defined with the T_p operator, a generalisation of the principle of *modus ponens*.

Definition 1. *Given a positive program Π and the set of atoms it contains, $HB(\Pi)$, the immediate consequence operator, $T_p : \mathscr{P}(HB(\Pi)) \rightarrow \mathscr{P}(HB(\Pi))$, is defined as:*

$$T_p(A) = A \cup \{H(r) | r \in \Pi, B^+(r) \subset A\} \tag{1}$$

[1] Here we use the notation of [3].

[2] This contrasts with `Prolog` and related operational semantics where the order of rules and the ordering within rules affects the semantics.

The model of Π is the fixpoint of applying T_p to $\emptyset$.

For example, given the following positive program:

$$\begin{aligned}
a &\leftarrow b. \\
b &\leftarrow . \\
c &\leftarrow a, b. \\
d &\leftarrow b, c. \\
e &\leftarrow f. \\
f &\leftarrow e.
\end{aligned}$$

the model is given by the computation:

$$\begin{aligned}
T_p(\emptyset) &= \{b\} \\
T_p(\{b\}) &= \{b, a\} \\
T_p(\{b, a\}) &= \{b, a, c\} \\
T_p(\{b, a, c\}) &= \{b, a, c, d\} \\
T_p(\{b, a, c, d\}) &= \{b, a, c, d\}
\end{aligned}$$

note that e and f do not appear in the model. Although $\{a, b, c, d, e, f\}$ would be a model if the rules where interpreted as propositions in classical logic, e and f are not included as there is no independent 'reason' why they should be regarded as being known (to conclude e we have to know f but this is only known if e is known).

Clearly, the model of a positive program can be visualised as a graph, with nodes corresponding to atoms, and a directed link expressing inference. More formally the node corresponding to a links to the node corresponding to $b \iff \exists r \in \Pi \centerdot H(r) = b, a \in B(r)$. These are referred to as a *support graph*, figure 1 shows the graph for the proceeding example. In this case the arc have been labelled with the corresponding rules.

The natural mechanism for computing negation in logic programs in *negation by failure*, which tends to be characterised as epistemic negation ("we do not known this is true"), rather than classical negation ("we know that this is not true"). This correspondence is motivated by the intuition that we should only claim to know things that can be proven, thus anything that can not be proven is not known. To extend the semantics to support this type of negation, the *Gelfond-Lifschitz reduct* is used. This takes a set of proposed atoms and gives a reduced, positive program by removing any rule which depends on the negation of any atom in the set and dropping all other negative dependencies.

Definition 2. *The Gelfond-Lifschitz reduct of an AnsProlog program Π with respect to a set A, written as Π^A is given by:*

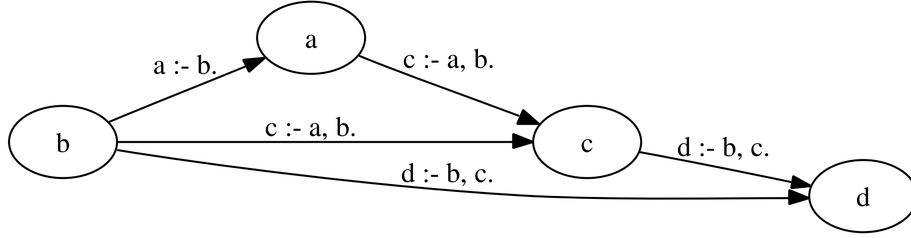

Fig. 1. The support graph for a simple positive program

$$\Pi^A = \{H(r) \leftarrow B^+(r) | r \in \Pi, B^-(r) \cap A = \emptyset\} \tag{2}$$

For example, if Π denotes the following program:

$$\begin{aligned} a &\leftarrow \text{not } b. \\ b &\leftarrow \text{not } a. \\ c &\leftarrow \text{not } d. \\ e &\leftarrow a, c, \text{not } b. \\ f &\leftarrow \text{not } g, e. \\ g &\leftarrow \text{not } f, e. \end{aligned}$$

Thus the set $\{b, c\}$ gives the reduced program $\Pi^{\{b,c\}}$

$$\begin{aligned} b &\leftarrow . \\ c &\leftarrow . \\ f &\leftarrow e. \\ g &\leftarrow e. \end{aligned}$$

As all rules that depend on not b or not c are ignored and the remaining negations are removed. This naturally leads to the definition of *answer sets*:

Definition 3. *If Π is an AnsProlog program, then A, a set of atoms is an answer set of $\Pi \Leftrightarrow A$ is the model of the program Π^A.*

So in the previous example, $\{b, c\}$ is an answer set of Π as it is clearly the least fixpoint of T_p applied to the reduced program. $\{a, c, e, f\}$ and $\{a, c, e, g\}$ are also answer sets of Π. A simple but significant corollary of this definition is that

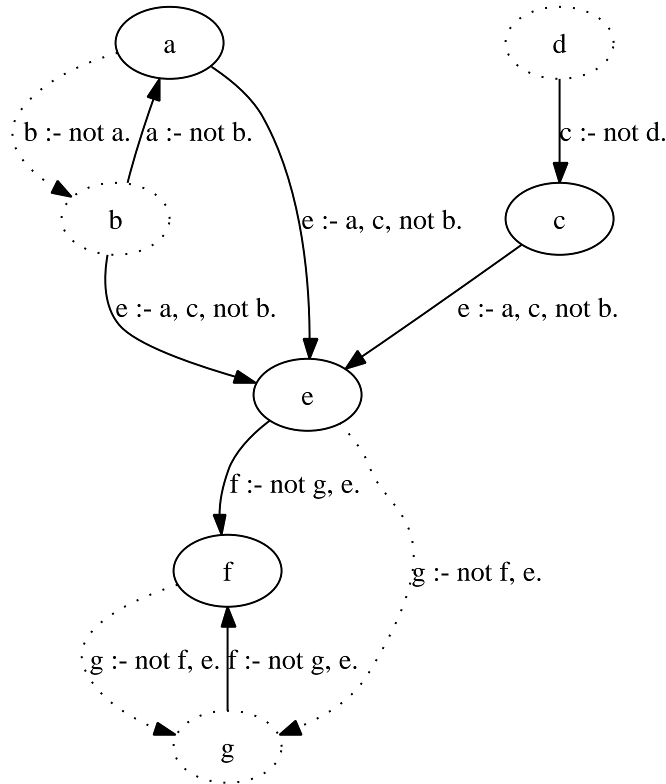

Fig. 2. The support graph for $\{a, c, e, f\}$

given an answer set A and an atom $d \notin A$, there are no rules with $H(r) = d$, $B^{+}(r) \subset A$ and $B^{-}(r) \cap A = \emptyset$, i.e. every rule that could conclude d has a reason why it is not applicable (one of the positive dependencies is not met or one of the negative dependencies is met).

There are a number of ways of displaying an answer set graphically. Clearly, it can be displayed as a directed acyclic graph as in the case of positive programs. However to understand why certain rules are/are not applicable, it is necessary to augment this with nodes corresponding to atoms not in the answer set. Figure 2 shows such a graph for the preceding program and the answer set $\{a, c, e, f\}$. Atoms that are not known and rules that are not applicable (some of the conditions in the body are not met) are marked by the use of dotted lines.

A program will have zero or more answer sets. Critically, computing an answer set of a program is an NP-complete task, which gives rise of a declarative programming paradigm for solving NP and NP-complete problems.

3 Answer Set Programming

Answer Set Programming (ASP) is a methodology for solving NP and NP-complete problems by representing the problem as an *AnsProlog* program, so that the answer sets of the program correspond to the solutions of the problem. It has been used to tackle a variety of problems, including planning and diagnosis [18], modelling and rescheduling of the propulsion system of the Space Shuttle [20], multi-agent systems [4, 8], semantic web and web-related technologies [22], super-optimisation [5], reasoning about biological networks [15], voting theory [17], and investigating the evolution of language [10].

The key advantages of answer set programming as a problem solving technique are that programs are very compact and fast to write, the programmer can focus on describing the problem rather than having to design the search algorithm and that the code can be ported to a variety of parallel architectures by simply using alternative computation tools.

The modelling languages based on *AnsProlog* tend to include a number of 'syntactic sugar' constructs to make it easier and cleaner to express certain common concepts. The most important of these is the use of variables in the bodies of atoms. These are handled at a theoretical level (and practically by the current generation of solvers) via instantiation. The variables must be quantified over a finite domain, allowing each rule to be translated to a set of rules with each of the possible combinations of variable instantiations. In implementations this is referred to as *grounding*. Less complex, but equally commonly used, most modelling languages support *constraints*, which prevent sets of atoms appear in any answer set and *choice rules*, which express a (non deterministic) choice between a number of variables. Constraints are typically written as a rule with no head atom, choice rules are written with a set (and often upper and lower limits) of atoms instead of the head atom. Both of these can be handled by polynomial, modular transforms on the ground program, although for reasons of performance, some implementations, handle these directly.

Figure 3 shows a program describing the Japanese number puzzle Sudoku. This is just a description of the rules of the puzzle, to solve a particular instance of given dimensions, domains for `numbers`, `row` and `col` must be given and facts giving the `sameSubSquare` relation and starting numbers must be added (Figure 4 on page 8 gives a simple example of this). The separation between the *encoding* of the general problem and the particular *instance* is a common feature of answer set programming. The first rule simply defines when two `X,Y` location pairs refer to the same location. The second rule is a choice rule and says that for every square (every row/column pair), exactly 1 (at least 1 and at most 1) number must be assigned to that square. The remaining three rules are constraints, the first saying that no two squares in the same row can be assigned the same number, the second saying that no two squares in the same column may be assigned the same number and the third saying that no two squares in the same subsquare (3 by 3 squares in the conventional puzzle) may be assigned the same number.

A number of 'off the shelf' reasoning engines exist that can compute the answer sets of an *AnsProlog* program. Most of these are divided into two com-

```
sameSquare(X,Y,X,Y) :- col(X), row(Y).
1 { assigned(X,Y,N) : number(N) } 1 :- col(X), row(Y).
 :- assigned(X1,Y,N), assigned(X2,Y,N), not sameSquare(X1,Y,X2,Y).
 :- assigned(X,Y1,N), assigned(X,Y2,N), not sameSquare(X,Y1,X,Y2).
 :- assigned(X1,Y1,N), assigned(X2,Y2,N), not sameSquare(X1,Y1,X2,Y2),
    sameSubSquare(X1,X2), sameSubSquare(Y1,Y2).
```

Fig. 3. A formalisation of Sudoku using *AnsProlog*

ponents, a *grounder* which handles the instantiation of rules containing variables and removal of other 'syntactic sugar' constructs, and a *solver* which takes the rules and computes the answer sets. `Gringo`[12] and `lparse`[24] are the grounders most commonly used and `clasp`[11], `smodels`[19], `cmodels`[14] and `dlv`[9] represent the state of the art of solver development. `Platypus`[16] is a solver that supports both multi-threaded shared memory and distributed memory parallel computation of answer sets.

4 Debugging *AnsProlog* Programs

Most of the work on explaining and illustrating the structure of answer sets has been from the perspective of debugging. A distinction is normally made between explaining why certain atoms occur in an answer set and explaining why no answers sets (or no answer sets containing a given subset) have been computed. In the former case, most of the techniques described compute the support graph or some function of it. In the later case, the computation is normally focused on creating refutations, effectively support graphs that show inconsistent support for a given set of atoms.

In [6] it is suggested that approaches to debugging based on modification of solver algorithms are unlikely to be effective and two procedural algorithms for investigating the support and refutation graphs are presented. The first of these attempts to provide an explanation for why a given set of atoms is contained in an answer set, by generating the subgraph of nodes that support atoms in the set. The second answers the converse question, why is a given set not in any answer set (a sub-case of this is the common problem of the solver returning no answer sets due to a contradiction arising from the basic facts), by inferring outwards from the given set until a contradiction is given.

[25] focuses on non-consistent programs; those without answer sets, and computes the (cardinality) minimal set of constraints that result in a contradiction. These are referred to as a *diagnosis*. From these *explanations* (essentially refutation trees) are built. The key innovation introduced by this work was the use of ASP and meta-programming to find the diagnosis sets. By 'abstracting' the rules it was possible to use an answer set solver to perform the searches required.

Focusing primarily on the reasons why a atoms appeared in an answer set, [21] applied the concepts of *justification* to *AnsProlog*, and derived a formal

1			
4	3		2

```
number(1..4).
row(1..4).
col(1..4).
sameSubSquare(1,2).
sameSubSquare(2,1).
sameSubSquare(3,4).
sameSubSquare(4,3)

assigned(1,1,1).
assigned(1,3,4).
assigned(2,3,3).
assigned(4,3,2).
```

Fig. 4. A 4x4 Sudoku and its representation in *AnsProlog*

framework for reasoning about the support graphs and refutation trees of programs. Procedural implementations for a number of the key questions where also created, with the aim of producing a debugging interface.

Most recently, [7] has extended the use of meta-programming for program analysis to create a generic framework in which debugging 'questions' can be implemented as programs. This allows exploring the rules and atoms required to support subsets of answer sets (essentially mapping out and exploring the support graph) and determining the reasons why certain sets result in inconsistency (exploring the refutations).

All of this work has been primarily focused on computation of the reasons why certain properties of the answer sets hold. A topic that has received little research or implementation attention is the question of how to present the resultant mass of symbolic information back to the user.

5 The Need for Explanation

Consider the 4 by 4 Sudoku puzzle given in Figure 4. Using the generic Sudoku description given in Figure 3, this can be represented using the *AnsProlog* code in Figure 4. This is probably about the smallest program that follows the same style of programming used in most real applications. A naïve instantiation will produce 64 `assigned` atoms. Figure 5 gives a simplified and idealised version of the support graph for one of the answer sets. Here only one 'reason' for each atom is given, atoms not present in the answer set are omitted and the links are abstracted a little from the rules so they do not require the omitted atoms. This would be difficult, but possible to create automatically. A similar graph for a 3 by 3 Sudoku would contain 81 nodes. Most applications that are large enough to actually need automated debugging/explanation tools will include tens, if not hundreds of thousands of atoms and potentially millions of rules. Clearly any

interface that outputs the whole support graph as graphics or as any form of text will not be usable for anything more than toy programs.

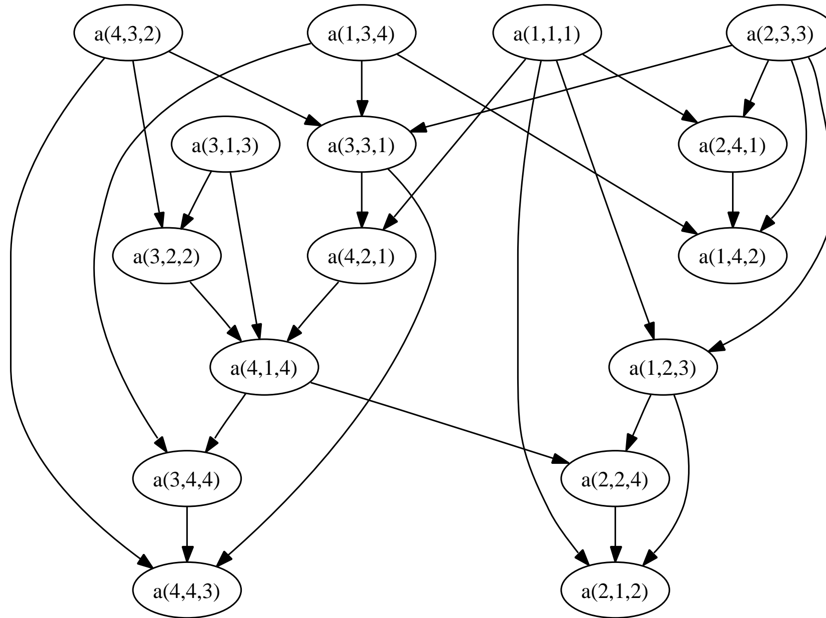

Fig. 5. An abstracted support graph for the program in figure 4

Thus any useful debugging/explanation interface will only output a section of the support graph or refutation at any given time. This leaves the problem of how to determine which section should be displayed. It is difficult to infer a suitable section of the graph automatically as in many cases[3] the program is semantically correct – just not what the programmer *meant*. Also, as the support graph corresponds to the structure, rather than the semantics of the argument, it is difficult to identify features of the graph that will correspond to 'bugs'.

This leads to the approach of most existing systems which require the programmer to annotate the rules and atoms to identify which areas are of interest and which areas can be 'assumed' to be correct. Although there is still scope for improvement in the interfaces used, this approach has a fundamental problem – providing useful annotation requires working out which rules and atoms are correct and which are not, which is largely the same task as finding the bug by hand.

Computing explanations and refutations for *AnsProlog* programs is easy, the problem is that the resulting arguments are large, causing a significant bottleneck

[3] Programs that can be shown to have no answer sets with minimal reasoning are possibly the only exception to this.

in communicating the explanation to the user. Thus in almost all cases, when the programs are large enough to need debugging support, it is faster to locate the bugs manually. Given the ease of extracting these explanations and their natural link with the syntax of the program, *AnsProlog* would seem an obvious application for explanation aware computing, but there remains an open question over how to build a suitable interface.

One possibility would be to integrate the explanation system into a development environment[23]. This would allow the programmer to mark sections of the program as 'correct' and thus have them ignored/assumed by all resultant explanation. Some kind of interface for controlling the marking and interactively exploring the graphs would also be needed. To reduce the cognitive load on the programmer it would be useful to be able to abstract the explanation from the ground instances of the rules in places where it would be meaningful for the explanation to contain variables. A continuation of this idea would be to (where possible) compute answer sets of the partial program during development, so that a programmer could ask hypothetical questions such as 'what would this rule do if added to the program'. Taken to its logical conclusion this could lead to an 'explanation centric' approach to development, where the central object of development was the explanation, with adding rules as the way of manipulating this until it became the explanation of the solution to the original problem.

An alternative to attempting to visualise the explanation/refutation graph would be to build a natural language translation of the rules with the aim of narrowing the gap between what the programmer intended and what the rules express. This would be focusing on explaining the rules rather than explaining the answer sets of the program. To make the text comprehensible it would probably be necessary to recognise some common idioms (transitive closure, assigning n objects to m locations). Such a system could be supported by annotations to the program to identify objects, state the correct way of expressing relations, etc. Although this does raise the question of whether the input language should be *AnsProlog* or a natural language subset that is translated to *AnsProlog*. One possible issue is that it would be natural to add plain text assertions to the program. For example, if the Sudoku description in figure 3 was converted to the following description:

- Every location (X,Y) contains one number.
- No two distinct locations in the same row contain the same number.
- No two distinct locations in the same column contain the same number.
- No two distinct locations in the same sub square contain the same number.

It would be tempting to add (as documentation, explanation and assertions):

- Every row must contain each number exactly once.
- Every column must contain each number exactly once.
- Every subsquare must contain each number once.

This is potentially problematic as adding redundant choice rules or constraints simply to encode these ideas would result in a larger and slower program and having them as assertions about the program could be computationally expensive.

6 Conclusion

Answer set programming is a declarative methodology for solving NP and NP-complete search problems. In common with many declarative languages, explanations are useful as the basis of debugging tools. Given the logical, declarative basis of the syntax and semantics of *AnsProlog*, extracting these explanations is a relatively simple matter. However, the existing methods of communicating to the user and exploring these explanations do not scale and result in the paradoxical situation of manual debugging being faster than using the support tools. This leaves an open question of how to partially work with large, regularly structured explanations.

References

1. *Proceedings of the International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
2. *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*. Springer, 2007.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
4. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In *Logic-based artificial intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.
5. M. Brain, T. Crick, M. De Vos, and J. Fitch. Toast: Applying answer set programming to superoptimisation. In *International Conference on Logic Programming*, LNCS. Springer, Aug. 2006.
6. M. Brain and M. De Vos. Debugging Logic Programs under the Answer Set Semantics. In M. De Vos and A. Provetti, editors, *ASP05: Answer Set Programming: Advances in Theory and Implementation*, pages 142–152. Research Press International, July 2005.
7. M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. Debugging asp programs by means of asp. In *LPNMR* [2], pages 31–43.
8. O. Cliffe, M. De Vos, and J. Padget. Specifying and analysing agent-based social institutions using answer set programming. In *Selected revised papers from the workshops on Agent, Norms and Institutions for Regulated Multi-Agent Systems (ANIREM) and Organizations and Organization Oriented Programming (OOOP) at AAMAS'05*, volume 3913 of *LNCS*, pages 99–113. Springer Verlag, 2006.
9. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, 1998.
10. E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe. Reconstructing the evolutionary history of indo-european languages using answer set programming. In *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2003.

11. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.
12. M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In *LPNMR* [2], pages 266–271.
13. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, Seattle, Washington, August 1988. The MIT Press.
14. E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-04)*, pages 61–66, 2004.
15. S. Grell, T. Schaub, and J. Selbig. Modelling biological networks by action languages via answer set programming. In *Proceedings of the International Conference on Logic Programming (ICLP'06)* [1], pages 285–299.
16. J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A Platform for Distributed Answer Set Solving. In *LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, pages 227–239. Springer, 2005.
17. K. Konczak. Voting theory in answer set programming. In *Proceedings of the Twentieth Workshop on Logic Programmin (WLP'06)*, number INFSYS RR-1843-06-02 in Technical Report Series, pages 45–53. Technische Universität Wien, 2006.
18. V. Lifschitz. Answer set programming and plan generation. *Journal of Artificial Intelligence*, 138(1-2):39–54, 2002.
19. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.
20. M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. A A-Prolog Decision Support System for the Space Shuttle. In *Answer Set Programming: Towards Efficient and Scalable Knowledge Represenation and Reasoning*. American Association for Artificial Intelligence Press, Stanford (Palo Alto), California, US, Mar. 2001.
21. E. Pontelli and T. C. Son. *Justifications* for logic programs under answer set semantics. In *ICLP* [1], pages 196–210.
22. M. Ruffolo, N. Leone, M. Manna, D. Saccà, and A. Zavatto. Exploiting asp for semantic information extraction. In *Answer Set Programming*, volume 142 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.
23. A. Sureshkumar, M. De Vos, M. Brain, and J. Fitch. Ape: An ansprolog* environment. In *Proceedings of the First International Workshop on Software Engineering for Answer Set Programming*, volume 281 of *Workshop Proceedings*, pages 101–115, Tempe, Arizona, US, May 2007.
24. T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report Series, Digital Systems Laboratory, Helsinki University of Technology, Oct. 1998.
25. T. Syrjänen. Debugging inconsistent answer set programs. In *NMR06*, pages 77–83, 2006.