A Glimpse of Answer Set Programming

Christian Anger and Kathrin Konczak and Thomas Linke and Torsten Schaub

Answer Set Programming (ASP) is a declarative paradigm for solving search problems appearing in knowledge representation and reasoning. To solve a problem, a programmer designs a logic program so that models of the program determine solutions to the problem. ASP has been identified in the late 1990s as a subarea of logic programming and is becoming one of the fastest growing fields in knowledge representation and declarative programming. Major advantages of ASP are (1) its simplicity, (2) its ability to model effectively incomplete specifications and closure constraints, and (3) its relation to constraint satisfaction and propositional satisfiability, which allows one to take advantage of advances in these areas when designing solvers for ASP systems.

1 Introduction

Answer Set Programming (ASP) emerged in the late 1990s as a new logic programming paradigm [22, 34, 35, 28], having its roots in nonmonotonic reasoning, deductive databases and logic programming with negation as failure. Since its inception, it has been regarded as the computational embodiment of nonmonotonic reasoning and a primary candidate for an effective knowledge representation tool. This view has been boosted by the emergence of highly efficient solvers for ASP [50, 17]. It now seems hard to dispute that ASP brought new life to logic programming and nonmonotonic reasoning research and has become a major driving force for these two fields, helping to dispel gloomy prophecies of their impending demise.

The basic idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver for finding answer sets of the program. This approach is closely related to the one pursued in propositional satisfiability checking (SAT), where problems are encoded as propositional theories whose models represent the solutions to the given problem. Even though syntactically, ASP programs look like Prolog programs, they are treated by rather different computational mechanisms. Indeed, the usage of model generation instead of query evaluation can be seen as a recent trend in the encompassing field of knowledge representation and reasoning. ASP is particularly suited for solving difficult combinatorial search problems. Among these, we find applications to plan generation, product configuration, diagnosis, and graph-theoretical problems.

2 Background

We restrict the formal development of ASP to propositional (normal) *logic programs* consisting of rules of the form

$$p_0 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\ , \qquad (1)$$

where $n \geq m \geq 0$, and each p_i $(0 \leq i \leq n)$ is an *atom*. Given such a rule r, we let $head(r)$ denote the *head*, p_0, of r and $body(r)$ the *body*, $\{p_1, \ldots, p_m,\ not\ p_{m+1}, \ldots, not\ p_n\}$, of r. Also, let $body^+(r) = \{p_1, \ldots, p_m\}$ and $body^-(r) = \{p_{m+1}, \ldots, p_n\}$. The intuitive reading of such a rule is that of a constraint on an answer set: If all atoms in $body^+(r)$ are included in the set and no atom in $body^-(r)$ is in it, then $head(r)$ must be included in the answer set.

Answer sets as such are defined via a reduction to negation-as-failure-free programs: A logic program is called *basic* if $body^-(r) = \emptyset$ for all its rules. A set of atoms X is *closed under* a basic program Π if for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$. The smallest set of atoms which is closed under a basic program Π is denoted by $Cn(\Pi)$ and constitutes the *answer set* of Π.

For the general case, we need the concept of a *reduct* of a program Π *relative to* a set X of atoms:

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}.$$

With these formalities at hand, we can define *answer set semantics* for logic programs: A set X of atoms is an *answer set* of a program Π if $Cn(\Pi^X) = X$. This definition is due to [22], where the term *stable model* is used; the idea traces back to [45]. In fact, one may regard an answer set as a model of a program Π that is somehow "stable" under Π. In other words, an answer set is closed under the rules of Π, and it is "grounded in Π", that is, each of its atoms has a derivation using "applicable" rules from Π.

For illustration, consider the program

$$\Pi = \{p \leftarrow p,\ q \leftarrow not\ p\}\ .$$

Among the four candidate sets, we find a single answer set, $\{q\}$, as can be verified by means of the following table:

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p\}$	$p \leftarrow p$	$\emptyset$
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$

A noteworthy fact is that posing the query p or q to Π in a Prolog system leads to non-terminating situation due to its top-down approach.

Analogously, we may check that the program

$$\Pi_1 = \{p \leftarrow not\ q,\ q \leftarrow not\ p\}$$

has the two answer sets $\{p\}$ and $\{q\}$.

X	Π^X	$Cn(\Pi^X)$
$\emptyset$	$p \leftarrow$	$\{p, q\}$
	$q \leftarrow$	
$\{p\}$	$p \leftarrow$	$\{p\}$
$\{q\}$	$q \leftarrow$	$\{q\}$
$\{p, q\}$		$\emptyset$

The two rules in Π_1 are mutually exclusive and capture an indefinite situation: p can be added unless q has been added and vice versa. We show in the next section how this can be exploited in modeling problems in ASP.

Unlike the previous examples,

$$\Pi_2 = \{p \leftarrow not\ p\}$$

admits no answer set. Interestingly, Π_2 offers a straightforward way to model *integrity constraints* of form

$$\leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\ . \qquad (2)$$

This can be done by introducing a new atom f and replacing (2) by rule $f \leftarrow not\ f, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$. Whenever the integrity constraint in (2) is violated by a candidate set this set is eliminated by the effect observed on program Π_2. The usefulness of integrity constraints can be observed by adding $\leftarrow p$ to program Π_1. In fact, the integrity constraints $\leftarrow p$ eliminates the original answer set $\{p\}$ of Π_1, so that $\Pi_1 \cup \{\leftarrow p\}$ yields a single answer set $\{q\}$.

Although general ASP is principally computationally complete, that is, it can simulate arbitrary Turing machines [6], one usually deals with decidable fragments. Commonly, we consider rules with variables that are taken as abbreviations for all ground instances over a finite set of constants. The propositional fragment of ASP defined above allows for encoding all decision and search problems within NP [48, 33].

3 Modeling

The basic approach to writing programs in ASP follows a "generate-and-test" strategy. First, one writes a group of rules whose answer sets would correspond to candidate solutions. Then, one adds a second group of rules, mainly consisting of integrity constraints, that eliminates candidates representing invalid solutions.

As an example consider the well-known n-queens problem. The goal is to place n queens on an $n \times n$ chessboard so that no two queens appear on the same row, column, or diagonal. Let us represent the positioning of the queens by atoms of the form $q(i, j)$, where $1 \leq i, j \leq n$. That is, an atom $q(i, j)$ represents that a queen is at position (i, j).

Let us first give a "generator". To this end, we build upon the non-deterministic behavior observed on program Π_1. Also, we introduce an auxiliary atom $q'(i, j)$ indicating that there is no queen at (i, j). Finally, we need a "domain predicate" d indicating the dimension of the chessboard. Accordingly, we obtain:

$$q(X, Y) \leftarrow d(X), d(Y), not\ q'(X, Y) \qquad (3)$$
$$q'(X, Y) \leftarrow d(X), d(Y), not\ q(X, Y) \qquad (4)$$

So, taking a 1×1 chessboard by adding $d(1)$, we obtain from (3) and (4) two answer sets, $\{q(1, 1), d(1)\}$ and $\{q'(1, 1), d(1)\}$, just as with program Π_1. Accordingly, we obtain by adding $d(1)$ and $d(2)$ a 2×2 chessboard, generating 16 answer sets, among which we find:

$\{q'(1,1), q'(1,2), q'(2,1), q'(2,2), d(1), d(2)\}$  (5)

$\{q(1,1), q'(1,2), q'(2,1), q'(2,2), d(1), d(2)\}$  (6)

$\{q(1,1), q'(1,2), q'(2,1), q(2,2), d(1), d(2)\}$  (7)

$\{q(1,1), q(1,2), q(2,1), q(2,2), d(1), d(2)\}$  (8)

Now, the "tester" must eliminate candidate sets in which queens are positioned on the same row, column, or diagonal. This can be done by means of the following integrity constraints:

$$\leftarrow q(X, Y), q(X', Y), X' \neq X, d(X), d(Y), d(X'), d(Y') \qquad (9)$$
$$\leftarrow q(X, Y), q(X, Y'), Y' \neq Y, d(X), d(Y), d(X'), d(Y') \qquad (10)$$
$$\leftarrow q(X, Y), q(X', Y'), |X - X'| = |Y - Y'|, X' \neq X, Y' \neq Y, d(X), d(Y), d(X'), d(Y') \qquad (11)$$

In fact, all rules rule out candidate set (8). Rule (11) eliminates set (7). However, none of them accounts for the requirement that n queens must be put on the board. This can be achieved by the following pair of rules.

$$\leftarrow d(X), hasq(X)$$
$$hasq(X) \leftarrow d(X), d(Y), q(X, Y)$$

The two latter rules eliminate candidate sets (5) and (6).

4 Language Extensions

Classical negation. Normal logic programs provide negative information implicitly through the closed world assumption [44]. Consider the following rule: $cross \leftarrow not\ train$. If $train$ is not derivable, the atom $cross$ becomes true. But this may lead to a disaster because you have no explicit information that there really is no train. An alternative would be to use an explicit negation operator $\neg$. Then we can express the previous rule as follows: $cross \leftarrow \neg train$.

An atom p or a negated atom $\neg p$ is called a *literal*. Logic programs with literals are called *extended logic programs* [22]. An extended logic program is *contradictory* [5] if complementary literals, e.g. $train$ and $\neg train$, are derivable. In that case, one obtains exactly one answer set, viz the set of all literals. If a program is not contradictory, the definition of answer sets of extended logic programs carries over from normal programs.

Classical negation can be eliminated by a polynomial transformation, replacing each negated atom $\neg p$ by a new atom p' and adding the rules,

$$q \leftarrow p, p' \quad \text{and} \quad q' \leftarrow p, p' \ , \tag{12}$$

for each atom p and q. The rules in (12) generate the set of all literals in case of contradictory programs. For preserving only consistent answer sets, we may add constraints $\leftarrow p, p'$ for each atom p, instead of the rules in (12).

Disjunctive logic programs extend normal programs by disjunctive information in the head of a rule [22]. More precisely, the head of a rule is a disjunction $q_0; \ldots; q_k$ for atoms q_i, where $0 \leq i \leq k$. E.g. $p; q$ expresses that "p is *true* or q is *true*". Letting $head(r) = \{q_0, \ldots, q_k\}$, a set of atoms X is closed under a basic program Π if for any $r \in \Pi$, $head(r) \cap X \neq \emptyset$ whenever $body^+(r) \subseteq X$. The definition of Π^X carries over from normal programs. An answer set X of a disjunctive logic program is a $\subseteq$- minimal set of atoms being closed under Π^X. For example, the disjunctive logic program $\Pi = \{p; q \leftarrow\}$ has the answer sets $\{p\}$ and $\{q\}$. The set $\{p, q\}$ is closed under $\Pi^{\{p,q\}}$, but it is not an answer set of Π since it is not $\subseteq$-minimal. Observe that adding the rules $p \leftarrow$ and $q \leftarrow$ to Π makes $\{p, q\}$ the only answer set of $\Pi \cup \{p \leftarrow, q \leftarrow\}$. If we use disjunction in the n-queens problem, rules (3) and (4) can be replaced by one rule:

$$q(X, Y);\ q'(X, Y) \quad \leftarrow \quad d(X), d(Y)$$

The usage of disjunction raises the complexity of the underlying decision problems, e.g. deciding whether there exists an answer set X for a given atom p such that $p \in X$ is Σ_2^P-complete [18].

Nested logic programs are logic programs where the bodies and heads of rules may contain arbitrary boolean expressions formed from propositional atoms and the symbols $\top$ (true) and $\bot$ (false) using negation-as-failure (not), conjunction (,), and disjunction (;) [30]. Answer sets are similarly defined as for disjunctive programs under regard that every boolean expression must be satisfied in the sense of classical logic and that the reduct Π^X is operating on boolean expressions. For illustration, consider nested program $\Pi = \{(p; not\ p) \leftarrow\}$. Taking $X = \emptyset$, we obtain $\Pi^\emptyset = \{(p; \top) \leftarrow\}$ as reduct and $\emptyset$ as the only $\subseteq$- minimal set being closed under $\Pi^\emptyset$, that is, the boolean expression $(p; \top)$ is trivially satisfied by $X = \emptyset$. For $X = \{p\}$, we get $\Pi^{\{p\}} = \{(p; \bot) \leftarrow\}$ and $\{p\}$ as the $\subseteq$-minimal set satisfying the expression $(p; \bot)$. Hence, $\emptyset$ and $\{p\}$ are the answer sets of nested program $\{(p; not\ p) \leftarrow\}$. In the n-queens problem, rules (3) and (4) can be replaced by $q(X, Y);\ not\ q(X, Y) \ \leftarrow \ d(X), d(Y)$, where the usage of nested expressions avoids using auxiliary predicate q'.

Nested programs can be polynomially translated into disjunctive programs [39].

Cardinality constraints are extended literals [49]. They are of the form $l\ \{q_1, \ldots, q_m\}\ u$, for $m \geq 1$, where l, u are lower and upper bounds on the cardinality of subsets of $\{q_1, \ldots, q_m\}$ satisfied in an encompassing answer set. They can appear in the head or in the body of a rule. A cardinality constraint is satisfied in an answer set X, if the number of atoms from $\{q_1, \ldots, q_m\}$ belonging to X is between l and u. To ensure in the n-queens problem that exactly one queen is in every column j, we use the expression $1\{q(1, j), \ldots, q(n, j)\}1$, which can be abbreviated by $1\{q(X, j) : d(X)\}1$, a so-called *conditional literal* [49]. Hence, the n-queens problem can be encoded with three rules, namely rule (11) and

$$1\{q(X, Y) : d(X)\}1 \quad \leftarrow \quad d(Y) \tag{13}$$

$$1\{q(X, Y) : d(Y)\}1 \quad \leftarrow \quad d(X), \tag{14}$$

where rules (13) and (14) ensure that there is exactly one queen in every column and row, respectively.

Deciding whether a normal program with cardinality constraints has an answer set is NP-complete [49].

Preferences. The notion of preference is pervasive in commonsense reasoning, e.g. in decision making, in part because preferences constitute a very natural and effective way of resolving indeterminate situations. In the following, we will exemplarily consider preferences among rules and so-called ordered disjunctions.

Rule preferences. A logic program with (static) preferences is a pair $(\Pi, <)$ where Π is a logic program and $<$ is a strict partial order among rules of Π expressing that one rule has higher priority than another rule. Also, dynamic preferences can be modeled by taking a special purpose predicate *prec* instead of an external order $<$. In both cases, the idea is to apply a rule r only if the "question of applicability" has been settled for all higher preferred rules r'. See [47, 14] for a survey on strategies for rule preferences.

Ordered disjunctions allow to represent alternative, ranked options in the head of rules [10]. $p \times q$ is an ordered disjunction which means: if possible p, but if p is impossible then at least q. Ordered disjunctions can be used, e.g. for the problem of composing a menu. You can choose between $meat$ or $vegetarian$ food. In case of choosing meat you would prefer $wine$ over mineral $water$, otherwise you prefer mineral $water$ over $wine$. These preferences can be modeled with ordered disjunctions as follows:

$$\begin{aligned} meat \times vegetarian \quad &\leftarrow \\ wine \times water \quad &\leftarrow \quad meat \\ water \times wine \quad &\leftarrow \quad not\ meat \end{aligned}$$

This program has answer set $\{meat, wine\}$. But in the case where $meat$ is not possible, e.g. adding the integrity constraint $\leftarrow\ meat$, the answer set is $\{vegetarian, water\}$.

Further preference handling approaches in answer set programming are preferences among literals [46], ordered

choice logic programs [8], the preference description language *PDL* [9] for specifying complex preferences structures in optimization problems, and consistency-restoring rules [4].

Other language extensions include aggregate functions [15], like *sum*, *count*, or *min*, weak constraints [27] as a variant of integrity constraints, and weight constraints [49] as an extension of cardinality constraints.

5 Systems

In recent years, several systems for answer sets computation have become available, including `smodels` [50, 49], `dlv` [17, 27], `noMoRe` [38, 32], `assat` [2, 31], and `cmodels` [12]. Most of the currently known ASP systems are designed to compute answer sets for propositional logic programs. In order to deal with programs containing variables, the systems rely on a two phase implementation consisting of:

1. elimination of variables for obtaining propositional programs and handling special system dependent language extensions (see Section 4); and
2. computation of answer sets for propositional programs.

Before detailing the second phase, we shortly report on the main tools for variable elimination, so-called *grounders*. Currently, most ASP systems utilize `lparse` (the grounder coming with the `smodels` system) or the grounder included in `dlv`. Both grounders allow for parsing disjunctive logic programs with classical negation along with their specific language extensions.

The remainder of this section describes the second phase, dealing with answer set generation of propositional logic programs. For simplicity, we deal with normal programs only. In this case, answer sets are (sub)sets of atoms occurring in a given program as heads of rules. A *partial model* is defined as a 3-valued truth assignment for the atoms of a program with truth values true, false and undefined. Such a 3-valued model is *total* if it contains no undefined atoms. Generally, the answer set computation in the second phase aims at extending a given partial model to a total one which is an answer set (or determining that this is impossible). The computation can be decomposed into alternating deterministic and non-deterministic parts. We start with the partial model where all atoms are undefined. Then we try to extend it to an answer set by computing its deterministic consequences by means of propagation techniques. These techniques extend the partial model by increasing the number of atoms being true or false, respectively. If the obtained model is total, then it is returned as an answer set. If the model is contradictory, in the sense that an atom has been assigned both true and false, then we have detected a situation admitting no answer sets. Otherwise, we must non-deterministically choose an undefined atom and "branch" on its possible truth values true and false. This is done by a case analysis which recursively applies the described procedure once to the partial model where the chosen atom is assigned true and again to the partial model where the chosen atom is false[1].

[1]The above procedure is similar to the Davis-Putnam-Logemann-Loveland procedure [13] for SAT solvers.

Observe that all ASP solvers additionally utilize some heuristics to guide their choices. The actual heuristics is crucial for the overall system performance, since the number of choices determines the depth of the (exponential) search tree. For a detailed discussion of different ASP heuristics including experimental results see [20].

The `smodels` system. The basic `smodels` algorithm can be described as follows:

smodels(L, U**)**
1 **expand(**L, U**)**
2 **if** $L \not\subseteq U$ **then return** fail
3 **if** $L = U$ **then exit with** L
4 $A \leftarrow$ **select(**$U \setminus L$**)**
5 **smodels(**$L \cup \{A\}, U$**)**
6 **smodels(**$L, U \setminus \{A\}$**)**

It computes an answer set between lower bound L and upper bound U, or determines that this is impossible. That is, the two sets L and U aim at capturing all answer sets X such that $L \subseteq X \subseteq U$. Observe that partial models are represented through two sets of atoms L and U, where L contains all true atoms and U contains all atoms not yet known to be false (that is, true or undefined); false atoms are implicitly given by $Atm(\Pi) \setminus U$.[2] First, `smodels` computes deterministic consequences of a partial model at hand by calling its propagation procedure `expand` (Line 1). Observe that `expand` is based on propagation rules which generalize the well-founded semantics [53]. In fact, if a call to `expand` produces sets L and U such that L is not a subset of U, then it follows that there is no answer set between the initially given bounds (Line 2). On the other hand, if $L = U$ after a call to `expand` (Line 3), then L is an answer set of the underlying logic program. More precisely, `expand` tries to enlarge lower bound L and, at the same time, it tries to make upper bound U smaller in such a way that no answer set is lost. That is, for all answer sets X of a program we have if condition $L \subseteq X \subseteq U$ holds before calling `expand`, then it also holds after the execution of `expand`. If necessary, `smodels` then chooses an undefined atom A (Line 4) and calls itself recursively, first (Line 5) to try to find an answer set between $L \cup \{A\}$ and U, that is, setting A true, and second (Line 6) to find an answer set between L and $U \setminus \{A\}$, that is, setting A false. Initially, `smodels` is called with $\emptyset$ and $Atm(\Pi)$. This guarantees that all answer sets of Π are found via backtracking.

The `dlv` system [17] extents the computation of answer sets to disjunctive logic programs and `dlv` specific language extensions, e.g. aggregate functions and weak constraints. Its core algorithm is similar to the one of `smodels`. The propagation of `dlv` also relies on computing well-founded semantics plus back-propagation mechanisms that allow for additionally marking atoms as being "eventually true".

The systems `assat` and `cmodels`. These two ASP solvers operate by reducing the problem of answer set computation to the satisfiability problem of propositional formulas (via Clark's completion [11]) and by invoking a SAT solver

[2]$Atm(\Pi)$ denotes the set of all atoms occurring in Π.

to generate answer sets. The fact that answer sets of the syntactically restricted class of *tight* programs correspond to classical models of its completion was discovered by Fages [21] and used for answer set computation in [2, 12]. For handling the general case correctly, so-called *loop formulas* have to be added [31]. However, complexity considerations show that an exponential growth in size is likely to be unavoidable, whenever logic programs are mapped into equivalent propositional formulas [29].

The `noMoRe` **system** [38] relies on a graph-based approach. It has its roots in default logic [45], where extensions are often characterized through their (unique) set of *generating default rules*. Accordingly, `noMoRe` characterizes answer sets by means of their set of generating rules. For determining whether a rule belongs to this set, we must verify that each positive body atom is derivable and that no negative body atom is derivable. In fact, an atom is derivable if the set of generating rules includes a rule having the atom as its head; or conversely, an atom is not derivable if there is no rule among the generating rules that has the atom as its head. Consequently, the formation of the set of generating rules amounts to resolving positive and negative dependencies among rules. For capturing these dependencies, `noMoRe` takes advantage of the concept of a *rule dependency graph*, wherein each node represents a rule of the underlying program and two types of edges stand for the aforementioned positive and negative rule dependencies, respectively. Answer sets now can be expressed by total non-standard 2-colorings of the rule dependency graph [32, 26], such that, whether a rule belongs to a set of generating rules or not, is indicated through its color.

Other systems. In addition to the above presented ASP systems there are other systems available, among them we find the `plp` system [40] for preference handling, `XSB` [43] for computing well-founded semantics, the `quip` [42] system dealing with ASP through quantified boolean formulas, `nlp` [36] a front-end for nested logic programs, `psmodels` [41] an implementation of ordered disjunctions, and `gnt` [23] an system for disjunctive logic programs based on `smodels`.

In order to foster further development of ASP the automated benchmarking system `asparagus` [1, 7] was launched. Its two principal goals are to provide an infrastructure for accumulating challenging benchmarks, and to facilitate executing ASP solvers under the same conditions, guaranteeing reproducible and reliable performance results.

6 Applications

ASP has been applied in multiple areas, e.g. product configuration [52, 51], planning [28, 16] and diagnosis [3].

In product configuration one derives a valid configuration from predefined components, some restrictions on these, and a set of customer requirements. One example is the configuration of PCs. Take program

$$computer \leftarrow \tag{15}$$

$$1\ \{ide\_disk, scsi\_disk\} \leftarrow computer \tag{16}$$

$$german\_layout; us\_layout \leftarrow computer \tag{17}$$

$$scsi\_controller \leftarrow scsi\_disk \tag{18}$$

expressing in rule (16) that a computer has an IDE *or* a SCSI disk and *either* a German *or* US layout keyboard in (17). If it does have a SCSI disk, rule (18) states that it also needs a SCSI controller. The answer sets of this program represent all possible configurations. If you now add the customer requirements (German keyboard, no SCSI disk)

$$\leftarrow\ scsi\_disk$$

$$german\_layout \leftarrow$$

the remaining, single answer set

$$\{computer, ide\_disk, german\_layout\}$$

represents the only valid configuration.

This easy, highly declarative way of dealing with configuration tasks is used e.g. for the configuration of Debian Linux [51].

Planning within ASP is another area that has been extensively studied during the last few years. Consider the blocks world domain where we want to move blocks from an initial situation to a goal situation. In ASP, we can express the problem in the following way. We start with the generator:

$$\{mv(B,L,T) : bl(B) : loc(L)\}\ 1 \leftarrow t(T), T < last \tag{19}$$

which generates possible moves (mv) of blocks (bl) to certain locations[3] (loc) for all discrete time steps (t) before time step $last$ (see rule (19)).

$$on(B,L,T+1) \leftarrow mv(B,L,T), bl(B), loc(L), t(T), T < last \tag{20}$$

$$on(B,L,T+1) \leftarrow on(B,L,T), not\ \neg on(B,L,T+1), bl(B), loc(L), t(T), T < last \tag{21}$$

$$\neg on(B,L_1,T) \leftarrow on(B,L,T), L \neq L_1, bl(B), loc(L), loc(L_1), t(T), T < last \tag{22}$$

We describe the effect of a move action in (20) by saying that if block B is moved to location L at time T, it will be on that location at the next time step $T+1$. Inertia (21) expresses that a block stays on a location, unless it does not ($\neg on$). The information that a block is not on a certain location is represented by rule (22), which expresses uniqueness of location (no block can be at two locations at the same time).

Now let us look at the tester part of our program.

$$\leftarrow 2\ \{on(B_1,B,T) : bl(B_1)\}, bl(B), t(T) \tag{23}$$

$$\leftarrow mv(B,L,T), on(B_1,B,T), bl(B), bl(B_1), loc(L), t(T), T < last \tag{24}$$

[3] Possible locations are the blocks themselves and the table.

Integrity constraint (23) expresses that no two blocks can be on top of the same block (though they can both be on the table). Via constraint (24) we express that a block can only be moved, if there is no block on top of it.

We have now described the blocks world domain with just six rules. Now we only need to specify our initial (i.e. at time step 0) and goal (at time step $last$) situations. E.g., we start with two towers with two blocks each:

$$on(1,2,0) \leftarrow \qquad on(2,table,0) \leftarrow$$
$$on(3,4,0) \leftarrow \qquad on(4,table,0) \leftarrow$$

To specify our goal situation of just one big tower, we use integrity constraints. We disallow all answer sets (possible plans) where the blocks are not in the desired position at the $last$ time step:

$$\leftarrow not\ on(4,3,last) \qquad \leftarrow not\ on(3,2,last)$$
$$\leftarrow not\ on(2,1,last) \qquad \leftarrow not\ on(1,table,last)$$

The last thing we need to do is to fix the macro $last$ to a specific number of time steps, representing the length of the plans to be investigated. For the situation above, we could e.g. use $last = 3$, which would give us all possible plans of length three, i.e., $\{mv(4,table,0), mv(3,2,1), mv(4,3,2)\}$.

We have refrained from giving explicit instantiations of the domain predicates bl and t, which would also be needed.

Other Applications of ASP include the Reaction Control System of the Space Shuttle [37], which has primary responsibility for maneuvering the aircraft while in space. Also, ASP has been employed e.g. for Constraint Programming [35], for certain issues pertaining Petri nets [24], for cryptanalysis [25] and for research in historical linguistics [19].

Acknowledgments. The authors were supported by DFG under grants FOR 375/1 and SCHA 550/6 as well as WASP project IST-2001-37004. We would like to thank G. Brewka, V. Lifschitz, I. Niemelä, and M. Truszczyński whose materials provided an extremely valuable source for this paper.

Literatur

[1] `http://asparagus.cs.uni-potsdam.de`.

[2] `http://assat.cs.ust.hk`.

[3] M. Balduccini and M. Gelfond. Diagnostic reasoning with A-prolog. *Theory and Practice of Logic Programming*, 3(4-5):425–461, 2003.

[4] M. Balduccini and M. Gelfond. Logic programs with consistency-restoring rules. In P. Doherty, J. McCarthy, and M.-A. Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI Spring Symposium Series*, 2003.

[5] C. Baral. *Knowledge representation, reasoning and declarative problem solving with Answer sets*. Cambridge University Press, 2003.

[6] P. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.

[7] P. Borchert, C. Anger, T. Schaub, and M. Truszczyński. Towards systematic benchmarking in answer set programming. In V. Lifschitz and I. Niemelä, editors, *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 3–7. Springer, 2004.

[8] M. Brain and M. De Vos. Implementing oclp as a front-end for answer set solvers: From theory to practice. In M. De Vos and A. Provetti, editors, *Proceedings of the International Workshop on Answer Set Programming*, 2003. CEUR Workshop Proceedings.

[9] G. Brewka. Complex preferences for answer set optimization. In D. Dubois, Chr. Welty, and M.-A. Williams, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 213–223. AAAI Press, 2004.

[10] G. Brewka, I. Niemelä, and T. Syrjänen. Logic programs with ordered disjunction. *Computational Intelligence*, 20(2):335–357, May 2004.

[11] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[12] `http://www.cs.utexas.edu/users/tag/cmodels.html`.

[13] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[14] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334, 2004.

[15] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in dlv. In G. Gottlob and T. Walsh, editors, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 847–852. Morgan Kaufmann, 2003.

[16] Y. Dimopoulos, B. Nebel, and J. Köhler. Encoding planning problems in nonmonotonic logic programs. In S. Steel and R. Alami, editors, *Proceedings of the European Conference on Planning*, pages 169–181. Springer, 1997.

[17] `http://www.dbai.tuwien.ac.at/proj/dlv`.

[18] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.

[19] E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe. Reconstructing the evolutionary history of indo-european languages using answer set programming. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*, pages 160–176, 2003.

[20] W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for answer set programming. In B. Nebel, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 635–640. Morgan Kaufmann, 2001.

[21] F. Fages. Consistency of clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
[22] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
[23] `http://www.tcs.hut.fi/Software/gnt`.
[24] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 218–223, 1999.
[25] M. Hietalahti, F. Massacci, and I. Niemelä. Des: A challenge problem for nonmonotonic reasoning systems. In *Proceedings of the International Workshop on Non-Monotonic Reasoning*, 2000.
[26] K. Konczak, T. Linke, and T. Schaub. Graphs and colorings for answer set programming: Abridged report. In V. Lifschitz and I. Niemelä, editors, *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 127–140. Springer, 2004.
[27] N. Leone, W. Faber, G. Pfeifer, T. Eiter, G. Gottlob, C. Koch, C. Mateis, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 2004. To appear.
[28] V. Lifschitz. Answer set planning. *Artificial Intelligence*, 138(1-2):39–54, 2002.
[29] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*. To appear.
[30] V. Lifschitz, L. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.
[31] F. Lin and Y. Zhao. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157:115–137, 2004.
[32] T. Linke. Graph theoretical characterization and computation of answer sets. In B. Nebel, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 641–645. Morgan Kaufmann, 2001.
[33] V. Marek and J. Remmel. On the expressibility of stable logic programming. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 107–120. Springer, 2001.
[34] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, 1999.
[35] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
[36] `http://www.cs.uni-potsdam.de/~torsten/nlp`.
[37] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*, pages 169–183, 2001.
[38] `http://www.cs.uni-potsdam.de/~linke/nomore`.
[39] D. Pearce, V. Sarsakov, T. Schaub, H. Tompits, and S. Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs. In P. Stuckey, editor, *Proceedings of the International Conference on Logic Programming*, pages 405–420. Springer, 2002.
[40] `http://www.cs.uni-potsdam.de/~torsten/plp`.
[41] `http://www.tcs.hut.fi/Software/smodels/priority`.
[42] `http://www.kr.tuwien.ac.at/research/quip.html`.
[43] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for effciently computing wfs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 431–441. Springer, 1997.
[44] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, 1978.
[45] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.
[46] C. Sakama and K. Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1-2):185–222, 2000.
[47] T. Schaub and K. Wang. A semantic framework for preference handling in answer set programming. *Theory and Practice of Logic Programming*, 3(4-5):569–607, 2003.
[48] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and Systems Sciences*, 51:64–86, 1995.
[49] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
[50] `http://www.tcs.hut.fi/Software/smodels`.
[51] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319, 1999.
[52] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, 2001.
[53] A. van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Contact

Chr. Anger, K. Konczak, Th. Linke, and T. Schaub
Universität Potsdam, Institut für Informatik
August-Bebel-Straße 89, D-14482 Potsdam
{christian,konczak,linke,torsten}@cs.uni-potsdam.de
http://www.cs.uni-potsdam.de/wv/