11th International Conference on Logic Programming and Nonmonotonic Reasoning

Vancouver, Canada, May 16–19 2011

1st Workshop on
Grounding and Transformation for Theories with Variables

May 16 2011

Proceedings

Pedro Cabalar, David Mitchell, David Pearce and Eugenia Ternovska (Eds.)

11th International Conference on Logic Programming and Nonmonotonic Reasoning

Vancouver, Canada, May 16–19 2011

1st Workshop on
Grounding and Transformation for Theories with Variables

May 16 2011

Proceedings

Contents

Tight Integration of Non-Ground Answer Set Programming and Satisfiability Modulo Theories

Tomi Janhunen, Guohua Liu, and Ilkka Niemelä

Aalto University School of Science
Department of Information and Computer Science
{Tomi.Janhunen,Guohua.Liu,Ilkka.Niemela}@aalto.fi

Abstract. Non-Boolean variables are important primitives in logical modeling. For instance, in Answer Set Programming (ASP), they are used as place holders for constants and more complex ground terms. This is essential for compact and uniform encodings used in ASP although variables are removed in a grounding phase preceding the search for answer sets. On the other hand, in theories in the Satisfiability Modulo Theories (SMT) framework, variables are realized as constants that have a free interpretation over a specific domain such as integers or reals. The goal of this paper is to propose an approach to integrating the languages employed in ASP and SMT so that non-Boolean variables of the kinds above can appear in the same program. The resulting formalism ASP(SMT) is rule-based and extended by theory atoms from SMT dialects. We illustrate the use of the new language and its advantages from the modeling perspective. Moreover, we show how existing off-the-shelf ASP and SMT technology can be used to implement grounding and search for answer sets for this class of programs.

1 Introduction

Non-Boolean variables are important primitives in logical modeling and they are exploited in a number of paradigms such as answer set programming (ASP) [9, 11], traditional constraint programming (CP) [15], and satisfiability modulo theories (SMT) [14]. In ASP, variables are used as place holders for constants and more complex ground terms which formalize the domain of interest. Hence, rules can be written in a more abstract way which is essential for compact and uniform encodings typically sought in ASP. A typical ASP system implements answer set computation in two steps: first variables are removed via *grounding* and then the actual *search* for answer sets is based on the resulting ground program. In CP, problems are encoded by introducing a set of variables over particular (non-Boolean) domains, and by restricting the values of the variables using constraints available in the language. Such representations can be very compact which becomes apparent when translating constraint satisfaction problems (CSPs) into pure Boolean representations [7]. Similar effects are expected if translations into ASP are of interest. Thus, in order to exploit the succinctness of domain variables in ASP, an integration of rules and constraints is justifiable [5, 10]. Then one can utilize rules, which are not directly available in CP, for knowledge representation.

The SMT framework generalizes Boolean satisfiability checking in terms of a background theory which is selected amongst a number of alternatives. In addition to propositional atoms, also theory atoms are allowed and they can be used to express various

constraints such as linear constraints, difference constraints, and so on. Such constraints can involve non-Boolean variables.[1] The relation of ASP and SMT has been studied and it was recently shown [8, 12] that logic programs under answer set semantics can be translated into a particular SMT fragment, namely *difference logic* (DL) [13]. The transformation is linear but quite sophisticated. Hence, it is not reasonable to expect that such theories are written by humans in order to express ASP primitives in SMT. Translations in the other direction, i.e., from SMT to ASP, do not seem feasible because of the potentially infinite domains of variables involved in theory atoms, and since current ASP solvers expect a finite ground logic program as their input. Thus, in order to take the best out of ASP and SMT in modeling, an integrated language is called for.

The goal of this paper is to integrate the languages employed in ASP and SMT so that non-Boolean variables available in these formalisms can be used together. We aim at a rule-based language ASP(SMT) which enriches rules in terms of theory atoms from a particular SMT dialect. To get a preliminary idea of the resulting language combining rules and theory atoms, consider the following rules which formalize a part of the famous n-queens problem, i.e., placing n queens in different rows:

$$\begin{array}{l}\mathsf{queen}(1..n).\\ \mathsf{int}(row(X)) \leftarrow \mathsf{queen}(X).\\ \leftarrow row(X) - row(Y) = 0,\ \mathsf{queen}(X),\ \mathsf{queen}(Y),\ X < Y.\end{array}$$

Here $row(X) - row(Y) = 0$ is a theory atom of difference logic involving term variables X and Y in the sense of ASP. The term $row(X)$, once grounded, will be treated as an integer variable on the SMT side. After grounding, there will be n facts $\mathsf{queen}(1)$, $\ldots$, $\mathsf{queen}(n)$, as well as facts $\mathsf{int}(row(1)), \ldots, \mathsf{int}(row(n))$ formalizing type information, and effectively $n(n-1)/2$ difference constraints $\leftarrow row(1) - row(2) = 0; \ldots;$ $\leftarrow row(1) - row(n) = 0; \ldots$, etc. All this information can be expressed in DL and for the purposes of this paper, we will mostly concentrate on enhancing ASP with DL.

There are related approaches [1, 5, 10] that integrate constraint solving techniques in ASP. In these approaches, answer sets are computed using both an ASP solver and a constraint solver. The former deals with ASP rules whereas the latter handles constraints only. In particular, the approach of [1] computes an answer set in two steps: At first, a partial answer set with a set of constraints is computed and then the partial answer set is completed by solving the constraints. In contrast, our goal is to treat the entire program in a uniform way, i.e., everything including difference constraints and ASP rules are translated into a difference logic formula and solved by a respective solver.

The rest of this paper is organized as follows. The main definitions and notions of ASP and DL are briefly reviewed in the next section. The tight integration of the ASP and SMT frameworks takes place in Section 3. The syntax and a semantics based on (extended) answer sets is laid out. The treatment of non-Boolean variables in the resulting language is of special interest. The modeling aspects are at glance in Section 4. A number of existing problem domains are used to illustrate the positive effects of theory atoms on traditional ASP encodings. On the other hand, we show how recursive definitions enabled by rules can be used to enhance representations in pure SMT. The

[1] However, variables in SMT are syntactically formalized as constants having a free interpretation over a specific domain such as integers or reals.

objective of Section 5 is to present our preliminary implementation of ASP(DL) using off-the-shelf ASP grounders and solvers for difference logic. A translator LP2DIFF that transforms ground SMODELS programs into theories of ASP(DL) is a key component in this implementation and we use delayed macro expansion with M4 to implement the grounding of theory variables. The resulting tool DINGO has been designed so that it is easy to extend for SMT dialects supporting difference constraints or equivalent. Then, in Section 6, we report the results from our preliminary experiments of using DINGO. Finally, Section 7 concludes the paper and presents some directions for future research.

2 Preliminaries

In this section, we briefly review the syntax and semantics of answer set programs. The class of normal logic programs is addressed first—followed by its extension in terms of choice rules, cardinality rules, and weight rules. Moreover, we outline difference logic which forms the target language for our preliminary implementation.

Normal Logic Programs As usual, a normal logic program P is a finite set of rules

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n. \tag{1}$$

where each a, b_i, and c_j is a propositional atom. Given a rule r of the form (1), we introduce the following abbreviations. The *head* and the *body* of r, are defined by $hd(r) = a$ and $bd(r) = \{b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n\}$. Quite similarly, we let $bd^+(r) = \{b_1, \ldots, b_m\}$ and $bd^-(r) = \{c_1, \ldots, c_n\}$ to distinguish the *positive* and the *negative* parts of $bd(r)$, respectively. The intuition behind a rule r is that the head $hd(r)$ can be inferred by r if $bd(r)$ is satisfied: all atoms in $bd^+(r)$ can be inferred whereas none from $bd^-(r)$. A rule without head is an *integrity constraint* enforcing the body to be false. A rule without body is a *fact* whose head is true unconditionally.

The Herbrand base of a (normal) program P, denoted $\mathrm{At}(P)$, is the set of atoms that appear in its rules. We define *interpretations* for P as subsets of $\mathrm{At}(P)$. An interpretation M satisfies an atom a if $a \in M$ and a negative literal not a if $a \notin M$, denoted $M \models a$ and $M \models \text{not } a$, respectively. The interpretation M satisfies a set of literals L, denoted $M \models L$, if it satisfies every literal in L and M satisfies a rule r of the form (1) if $M \models hd(r)$ holds whenever $M \models bd(r)$ holds. The interpretation M is a model of P, denoted $M \models P$, if M satisfies each rule of P. An *answer set* of a program is in a sense "justified" model of the program which is captured by the concept of a *reduct*.

Definition 1. *Let P be a normal program and M an interpretation. The* reduct *of P with respect to M is $P^M = \{hd(r) \leftarrow bd^+(r) \mid r \in P \text{ and } bd^-(r) \cap M = \emptyset\}$.*

The reduct P^M does not contain any negative literals and, hence, has a unique $\subseteq$-minimal model. If this model coincides with M, then M is an answer set of P.

Definition 2. *Let P be a normal program. An interpretation $M \subseteq \mathrm{At}(P)$ is an* answer set *of P iff M is the minimal model of the reduct P^M.*

Weight Constraint Programs In the context of SMODELS-compatible systems, certain extended rule types [16] are available and based on expressions of the following forms:

$$l\{b_1, \ldots, b_n, \text{not } c_1, \ldots, \text{not } c_m\}u \tag{2}$$

$$l\{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, \text{not } c_1 = w_{c_1}, \ldots, \text{not } c_m = w_{c_m}\}u \tag{3}$$

Here (2) is a *cardinality atom* that is satisfied if the number of satisfied literals is between the lower bound l and the upper bound u. A *weight atom* of the form (3) generalizes this idea by assigning arbitrary positive weights to literals (rather than 1s). If these expressions appear in the head of a rule, they effectively express a *choice*. It is possible to generalize answer sets for rules extended by cardinality and weight atoms but the reader is referred to [16] for details. The class of *weight constraint programs* (WCPs) based on this syntax is supported by SMODELS-compatible systems. If appropriate, WCPs can be translated back to normal programs (implemented by a tool called LP2NORMAL[2]) or directly into DL. For this reason, we will freely use these extensions.

Difference Logic Difference logic (DL) [13] extends the language of classical propositional logic in terms of *difference constraints* of the form[3]

$$x - y \leq k \tag{4}$$

where x and y are variables ranging over integers and and k is an integer constant. The respective language of DL is based on *atomic formulas*, i.e., either propositional atoms or difference constraints, and is closed under negation $\neg$ and conjunction $\wedge$. Other Boolean connectives $\vee$, $\rightarrow$, and $\leftrightarrow$ can be defined in the standard way using $\neg$ and $\wedge$ as basis. Given these conventions, e.g., the expression $(x-y \leq 2) \leftrightarrow (p \rightarrow \neg(y-x \leq 2))$ is a well-formed formula in DL. It was established in [13] that the variables in (4) can also range over rationals or reals and the relational operator can be any of $<$, $>$, $\geq$, $=$, and $\neq$. For the purposes of this paper, however, we focus on integer variables but do not restrict the relational operator. Moreover, when $k = 0$, we sometimes write $x \ op \ y$ for $x - y \ op \ 0$ for the relational operators op listed above.

In DL, an *interpretation* is a pair $\langle I, \tau \rangle$ where I is a set of propositional atoms assumed to be true and τ is a valuation function that maps each variable x to an element in $\mathbb{Z}$, i.e., the set of integers. A propositional atom p is true in $\langle I, \tau \rangle$, denoted $\langle I, \tau \rangle \models p$, iff $p \in I$. Difference constraints (4) are covered by setting

$$\langle I, \tau \rangle \models x - y \leq k \quad \text{iff} \quad \tau(x) - \tau(y) \leq k.$$

The truth value of any DL formula ϕ can be evaluated by applying the standard recursive rules for Boolean connectives. For example, given a function τ such that $\tau(x) = \tau(y) = 1$, we have that $\langle \emptyset, \tau \rangle \models ((x - y \leq 2) \leftrightarrow (p \rightarrow \neg(y - x \leq 2))$.

A DL formula ϕ is *satisfiable*, if there is a *satisfying* interpretation $\langle I, \tau \rangle$ such that $\langle I, \tau \rangle \models \phi$ and, in this setting, $\langle I, \tau \rangle$ is called a *model* of ϕ. Since DL has classical

[2] `http://www.tcs.hut.fi/Software/asptools/`

[3] The original form of difference constraints is $x \leq y + k$ as given in [13]. However, we use the form (4) to emphasize the difference between x and y.

propositional logic as its special case, it is straightforward to see that given a DL formula ϕ, the problem of deciding the satisfiability of ϕ is NP-complete. We refer the reader to [2, 13] for SMT based techniques for solving the satisfiability problem. Moreover, it is shown in [8, 12] that normal logic programs can be faithfully embedded into DL. This transformation is implemented by another translator, viz. LP2DIFF[4]. It is also worth pointing out that more general rule types involving cardinality and weight atoms can be translated into DL without substantial blow-up using new atoms and integer variables.

3 The Integrated Language

This section presents the language ASP(DL) which enriches ASP rules with DL formulas. A *logic program with difference constraints*, or a *program*, is a set of rules

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n, t_1, \ldots, t_l \tag{5}$$

where a, b_i, and c_i are propositional atoms. Each t_i, called *theory atom*, is a difference constraint. For a rule r of the form (5), the set of theory atoms $\{t_1, \ldots, t_l\}$ is called the *theory* part of the body of r and denoted by $bd^t(r)$. Comparing (5) to (1), the integrated language extends the pure ASP language in terms of theory atoms used as additional conditions. For a program P with difference constraints, an *interpretation* I is defined as a pair $\langle M, T\rangle$ where M is a set of propositional atoms and T is a set of theory atoms such that $T \cup \overline{T}$ is satisfiable in DL where $\overline{T}$ is the set of DL formulas $\neg t$ for each theory atom t appearing in P but not in T. For an interpretation $I = \langle M, T\rangle$, we denote M by I^p and T by I^t. An interpretation I satisfies an atom, a literal, or a rule if and only if the set $I^p \cup I^t$ satisfies them in the sense of Section 2, respectively. A *model* M of P, denoted $M \models P$, satisfies all rules of P. Definitions 1 and 2 generalize as follows.

Definition 3. *A model M of a program P is an* answer set *of P, if M^p is the minimal model of $P^M = \{hd(r) \leftarrow bd^+(r) \mid r \in P,\ bd^-(r) \cap M^p = \emptyset,$ and $M^t \models bd^t(r)\}$.*

Example 1. Let P be a program consisting of the five rules given below.

$$\leftarrow \text{not } s. \qquad s \leftarrow x > z. \qquad p \leftarrow x \leq y. \qquad p \leftarrow q. \qquad q \leftarrow p, y \leq z.$$

Here s, p, and q are propositional atoms whereas the symbols x, y, and z appearing in the theory atoms are treated as constants in the sense of ASP. Consider the interpretation $M_1 = (\{s\}, \{x > z\})$. It is an answer set of P, since the set $\{(x > z), \neg(x \leq y), \neg(y \leq z)\}$ is satisfiable in DL, $M_1 \models P$, and the set $\{s\}$ is the minimal model of the reduct $P^{M_1} = \{s \leftarrow;\ p \leftarrow q\}$. On the other hand, the interpretation $M_2 = (\{s, p, q\}, \{x > z, x \leq y, y \leq z\})$ is not an answer set, since $\{(x > z), (x \leq y), (y \leq z)\}$ is not satisfiable. Finally, consider $M_3 = (\{s, p, q\}, \{x > z, y \leq z\})$. It is not an answer set as $\{s, p, q\}$ is not the minimal model of the reduct $P^{M_3} = \{s \leftarrow;\ p \leftarrow q;\ q \leftarrow p\}$. □

It can be verified that the semantics given by Definition 3 coincides with the stable model semantics proposed in [6] in case no theory atoms appear in a program. In the

[4] `http://www.tcs.hut.fi/Software/lp2diff/`

non-ground case, the idea is to treat ASP variables appearing in rules as usual via Herbrand instantiation. As illustrated in Example 1, theory atoms typically refer to integer variables of DL. Such a variable, say x, can be viewed as a constant in the respective ASP language. In certain applications, we would like to flexibly use such variables and index them with ASP variables. This leads to using a term of the form $x(V_1, \ldots, V_m)$ where x is now a function symbol of arity m and each V_i is a regular ASP variable subject to Herbrand interpretation. Each ground instance $x(c_1, \ldots c_m)$ of this will be a ground term in the resulting ground program and treated as an integer variable in DL. This is an important distinction to which we want to draw the reader's attention at this point. By this arrangement, we can use both ASP variables and DL integer variables. A rule with non-ground variables is treated as a shorthand for its Herbrand instances. E.g., in the constraint $\leftarrow \mathsf{occurs}(a, S_1),\ \mathsf{occurs}(b, S_2),\ t(S_2) - t(S_1) > 7$, the variables S_1 and S_2 are replaced by appropriate combinations of ground terms and the respective instances of $t(S_1)$ and $t(S_2)$ are treated as integer variables in DL.

4 Problem Modeling in ASP(DL)

In this section, we present a number of examples to illustrate the use and advantages of ASP(DL) for problem modeling. As we shall see, the language ASP(DL) can result in much smaller ground programs than pure ASP language. The reason is that ASP(DL) may avoid grounding the variables that have large domains.

4.1 Scheduling Problem

Scheduling problem is to find a schedule for a number of tasks to guarantee them to be finished before some time limit. In scheduling problems, the time range is usually large compared to the number of tasks. Using ASP(DL) can avoid grounding the variables in the time domain in the encoding of these problems. Below, we study a typical scheduling problem, the newspaper problem. Note that similar problems are encountered in many real-life applications, e.g., resource allocation, university timetabling, and complex manufacturing with multiple lines of products.

Example 2. We are given a number of persons and newspapers. Each person spends different amount of time when reading different newspapers, due to his or her interests. The goal of is to find a schedule for the persons to read the newspapers such that:

1. each person has enough time to read a newspaper and read it as quickly as possible;
2. no person can read more than one newspaper simultaneously;
3. no newspaper can be read simultaneously by more than one person; and
4. each person finishes all the newspapers before some deadline. □

To model the problem in ASP(DL), we use the predicate $\mathsf{read}(P, N, D)$ to represent that the duration that person P needs to read newspaper N is D and the constant $deadline$ to denote the total allowed time. We introduce the integer variables $s(P, N)$

and $e(P, N)$ to denote the time when person P starts to read and finishes reading newspaper N respectively. Then the above four constraints are expressed as follows:

$$\leftarrow e(P,N) - s(P,N) \neq D, \mathsf{read}(P,N,D). \tag{6}$$

$$\leftarrow s(P,N_1) < s(P,N_2), s(P,N_2) - s(P,N_1) < D_1, \mathsf{read}(P,N_1,D_1), \mathsf{read}(P,N_2,D_2), N_1 \neq N_2. \tag{7}$$

$$\leftarrow s(P_1,N) < s(P_2,N), s(P_2,N) - s(P_1,N) < D_1, \mathsf{read}(P_1,N,D_1), \mathsf{read}(P_2,N,D_2), P_1 \neq P_2. \tag{8}$$

$$\leftarrow e(P,N) > \mathit{deadline}, \mathsf{read}(P,N,D). \tag{9}$$

To encode the same constraints in pure ASP, we introduce predicates $\mathsf{start}(P, N, T)$ and $\mathsf{end}(P, N, T)$ to denote that person P starts to read and finishes reading newspaper N at time T, respectively. Then the main constraints are encoded as

$$\leftarrow \mathsf{start}(P,N,T_1), \mathsf{end}(P,N,T_2), T_1 - T_2 \neq D. \tag{10}$$

$$\leftarrow T_1 < T_2, T_2 - T_1 < D_1, \mathsf{start}(P,N_1,T_1), \mathsf{start}(P,N_2,T_2), \mathsf{read}(P,N_1,D_1), \mathsf{read}(P,N_2,D_2), N_1 \neq N_2. \tag{11}$$

$$\leftarrow T_1 < T_2, T_2 - T_1 < D_1, \mathsf{start}(P_1,N,T_1), \mathsf{start}(P_2,N,T_2), \mathsf{read}(P_1,N,D_1), \mathsf{read}(P_2,N,D_2), P_1 \neq P_2. \tag{12}$$

$$\leftarrow \mathsf{end}(P,N,T) > \mathit{deadline}, \mathsf{start}(P,N,T), \mathsf{read}(P,N,D). \tag{13}$$

Let us calculate the number of ground instances of the rules (7) and (11) to demonstrate the advantage of ASP(DL). The number of ground instances of the rule (7) is $n_1 = |P| \times |N|^2$ and that of the rule (11) is $n_2 = |P| \times |N|^2 \times |T|^2$, where $|P|$ and $|N|$ are the respective numbers of persons and newspapers and $|T|$ is the maximum allowed time for the persons to finish reading, i.e., the size of the time domain. The factor $|T|^2$ in n_2 comes from the inequalities in the rule (11). In scheduling problems, the time range $|T|$ is usually much larger than other domains, thus the encoding in ASP(DL) results to a much simpler ground program than that in pure ASP language.

4.2 Routing with Time Constraints

Besides scheduling problems, network routing problems involve reasoning with time domains. The language ASP(DL) is also suitable to model them.

Example 3. In a routing problem, we are given a network consisting of nodes connected by edges. Each edge is assigned a weight which indicates the minimum delay in transmitting a network packet along the edge from one end to the other. Some nodes are critical and each of critical nodes is associated a time indicating the deadline that the node can receive a packet (otherwise, the node rejects to receive and relay the packet). The goal of the problem is to find routes for packets subject to following constraints:

1. a packet can only be sent from and received by one node at a time;
2. the travel time of a packet over an edge is at least the minimum delay; and
3. a packet has to reach each critical node within its deadline. □

We use the predicate $\mathsf{node}(X)$ to represent that X is a node; $\mathsf{edge}(X, Y, W)$ to represent that the delay on the edge (X, Y) is W; $\mathsf{critical}(X, T)$ to represent that X is a critical node and its associated deadline is T; $\mathsf{route}(X, Y)$ to represent that edge (X, Y) is selected to be in the route and the packet is transmitted from node X to Y in the route; $\mathsf{reach}(X)$ to represent that a packet reaches node X. We use the integer variable $at(X)$ to denote the time when a packet reaches node X.

The main constraints in the routing problem can be encoded in ASP(DL) as:

$$\{\mathsf{route}(X, Y)\} \leftarrow \mathsf{edge}(X, Y, W). \tag{14}$$

$$\mathsf{reach}(Y) \leftarrow \mathsf{reach}(X), \mathsf{route}(X, Y). \tag{15}$$

$$\leftarrow 2\{\mathsf{route}(X, Y) : \mathsf{edge}(X, Y, W)\}, \mathsf{node}(X). \tag{16}$$

$$\leftarrow 2\{\mathsf{route}(X, Y) : \mathsf{edge}(X, Y, W)\}, \mathsf{node}(Y). \tag{17}$$

$$\leftarrow \mathsf{route}(X, Y), \mathsf{edge}(X, Y, W), at(Y) - at(X) < W. \tag{18}$$

$$\mathsf{missing\_critical} \leftarrow \mathsf{critical}(X, T), \text{not } \mathsf{route}(X). \tag{19}$$

$$\mathsf{missing\_critical} \leftarrow \mathsf{critical}(X, T), \mathsf{reach}(X), at(X) > T. \tag{20}$$

$$\leftarrow \mathsf{missing\_critical}. \tag{21}$$

The rule (14) says that any edge could be selected in a route; the rule (15) states that if one end of an edge is reached by a packet and the edge is selected in the route then the other end of the edge is also reached by the packet; the rule (16) and (17) enforces the first constraint; the rule (18) specifies the second constraint and the rules (19), (20), and (21) together encode the third constraint.

The difference logic formulas in the rules (18) and (20) reduce the size of the ground programs. For example, an encoding of (18) in pure ASP language could be

$$\leftarrow \mathsf{route}(X, Y), \mathsf{edge}(X, Y, W), \mathsf{reach\_at}(X, T_1), \\ \mathsf{reach\_at}(Y, T_2), T_2 - T_1 < W. \tag{22}$$

where the predicate $\mathsf{reach\_at}(X, T)$ represents that a packet reach the node X at time T. It can be seen that the number of ground instances of the rule (18) is $|E|$ and that of the rule (22) is $|E| \times |T|^2$, where $|E|$ is the number of edges and $|T|$ is the maximum allowed time for a packet to travel in the network.

4.3 Trip Planning

In [10], a trip planning problem that involves timing constraints is presented to demonstrate the advantage of integrating constraint logic programming with ASP. We show that encoding these timing constraints in ASP(DL) is similarly advantageous.

Example 4. John, who is currently at work, needs to be in his doctor's office in one hour carrying the insurance card and money to pay for the visit. The card is at home and money can be obtained from the nearby ATM. John knows the minimum time (in minutes) needed to travel between the relevant locations, e.g., twenty minutes are needed to go from his home to his office. Can he find a plan to make his trip on time? The timing constraints relevant for this problem are:

1. time should be a monotonic function of steps[5];
2. the whole trip does not take more than an hour; and
3. sufficient amount of time is reserved for trips between any two locations. □

Following the notations in [10], we use the predicate $\mathsf{next}(S_1, S_0)$ to represent that step S_1 is the next step of S_0 and introduce the integer variable $t(S)$ to denote the time when step S happens. We use predicate $\mathsf{goal}(S)$ to represent that S is the goal step, i.e., John arrives his doctor's office in time at step S. The fluent $go\_to(P, L)$ represents that person P goes to location L and $at\_loc(P, L)$ represents that person P is at location L. The predicate $\mathsf{occurs}(A, S)$ represents that action A occurs at step S and $\mathsf{holds}(F, S)$ represents that fluent F holds at step S. Then the timing constraints can be encoded by:

$$\leftarrow \mathsf{next}(S_1, S_0), t(S_1) - t(S_0) < 0. \tag{23}$$

$$\leftarrow \mathsf{goal}(S), t(S) - t(0) > 60. \tag{24}$$

$$\leftarrow \mathsf{next}(S_1, S_0), \mathsf{occurs}(go\_to(john, home), S_0) \tag{25}$$

$$\mathsf{holds}(at\_loc(john, \mathit{office}), s_0), t(S_1) - t(S_0) < 20. \tag{26}$$

The advantage of the above encoding is similar to that discussed in [10], i.e., the size of the ground program of the encoding is smaller than that of the encoding in pure ASP by the factor of $|T|^2$ where $|T|$ is the size of the time domain.

4.4 Sorting Problem

Sorting is a frequently used operation in real applications. Thus, we are interested in the capability of ASP(DL) to model sorting problems.

Example 5. We are given a sequence of distinct numbers. The goal is to sort the numbers in increasing order, i.e., the resulting sequence must satisfy the constraints:

1. each number has one and only one place in the resulting sequence; and
2. a number is greater than any number before it in the resulting sequence. □

We use the predicate $\mathsf{number}(X)$ to represent that X is a number and the integer variable $p(X)$ to denote the position of X in the ordered sequence. The two constraints given above are encoded by the following rules:

$$\leftarrow p(X_1) = p(X_2), X_1 \neq X_2, \mathsf{number}(X_1), \mathsf{number}(X_2). \tag{27}$$

$$\leftarrow p(X_1), p(X_2), X_1 > X_2, p(X_1) < p(X_2), \mathsf{number}(X_1), \mathsf{number}(X_2). \tag{28}$$

To encode these constraints in pure ASP, we introduce a predicate $\mathsf{position}(Y)$ to represent that Y is a position in the ordered sequence and $\mathsf{place}(X, Y)$ to represent that the position of X is Y in the ordered sequence. Then the following program results:

$$1\{\mathsf{place}(X, Y) : \mathsf{position}(Y)\}1 \leftarrow \mathsf{number}(X). \tag{29}$$

$$1\{\mathsf{place}(X, Y) : \mathsf{number}(X)\}1 \leftarrow \mathsf{position}(Y). \tag{30}$$

$$\leftarrow X_1 > X_2, Y_1 < Y_2, \mathsf{place}(X_1, Y_1), \mathsf{place}(X_2, Y_2), \mathsf{number}(X_1), \mathsf{number}(X_2). \tag{31}$$

[5] The steps of the planning part of this problem can be found in [10].

The rules (29) and (30) together specify that each number is assigned to one and only one position. The rule (31) guarantees the increasing order of the sequence. We can see that the number of ground instances of the rule (27) (also (28)) is $|N|^2$ and that of the rule (31) is $|N|^4$ where $|N|$ is the length of the sequence.

5 Implementation

In this section, we present a preliminary implementation of the language ASP(DL) using existing off-the-shelf ASP and SMT tools, and other Unix/Linux tools. For simplicity, we discuss the implementation in the case of difference logic, but any other SMT fragment can be similarly dealt with by introducing suitable predicates for the kinds of theory atoms involved in that particular fragment. The current implementation has been designed to be flexible in this sense, i.e., it is easy to modify the representation of theory atoms in order to meet the users' needs. However, to avoid an implementation of a complex parser or a entirely new grounder for any such language extensions, we decided to exploit macros in the treatment of theory atoms. These requirements led us to implement the following architecture for grounding, macro evaluation, and solving:

(i) We use GRINGO *unmodified* to ground a logic program in its input language where theory atoms are represented by special predicates with reserved names. For instance, a ternary predicate $\mathsf{dl\_lt}(X, Y, D)$ could be introduced for DL and $<$. The grounder is instructed to treat such as *externally defined* predicates. In addition, some arguments to such predicates must have their types declared using rules. Special domain predicates are reserved for this purpose such as $\mathsf{int}(V)$ in case of DL. (ii) We translate the resulting ground program into a theory of difference logic using LP2DIFF. The outcome consists of Clark's completion of the program enhanced with ranking constraints—as explained in [8, 12]. Theory atoms can be recognized by their names based on reserved predicate names. (iii) We extract the relevant type information from the symbol table of the ground program, i.e., the instances of the predicate $\mathsf{int}(\cdot)$ for DL, and incorporate the respective declarations to the prologue of the DL theory produced in the previous step. (iv) We separate ground instances of theory atoms from the symbol table, treat them as macro instances, and expand them into respective expressions of DL using M4 which is a standard macro processor available in Unix environments. (v) We invoke an SMT solver such as Z3 to compute a satisfying assignment (if any) that can be mapped back to an answer set given symbolic information stored after grounding. In addition, the values of integer variables are also of interest.

The steps above have been integrated into one shell script DINGO[4] that accepts a file in the syntax of GRINGO as its input and prints an answer set as its output.

Example 6. For the sake of illustration, consider the rule (18) and the occurrence of the theory atom $at(X) - at(Y) < W$ in it. This can be expressed using a predicate $\mathsf{dl\_lt}(at(X), at(Y), W)$ in the respective logic program. In addition to this, we insist on the declaration of SMT theory constants using separate rules:

$$\mathsf{int}(at(X)) \leftarrow \mathsf{edge}(X, Y, W).$$
$$\mathsf{int}(at(Y)) \leftarrow \mathsf{edge}(X, Y, W).$$
$$\leftarrow \mathsf{route}(X, Y), \mathsf{edge}(X, Y, W), \mathsf{dl\_lt}(at(Y), at(Y), W).$$

The first two rules state that terms $at(X)$ associated with nodes X denote integer variables. This is not necessary for the third arguments W of the edge predicate under the assumption that W will be literally substituted by a concrete integer value such as 15. The third rule is a rewrite of (18) using the predicate $\mathsf{dl\_lt}(\cdot,\cdot,\cdot)$.

Let us then justify the correctness of the implementation, for the sake of simplicity, in the case of propositional normal programs. As shown by Janhunen et al. [8], an interpretation $M \subseteq \mathrm{At}(P)$ is a stable model of a normal program P iff there is a model $\langle M, \tau\rangle \models \mathrm{Comp}(P) \cup \mathrm{Rank}(P)$ where $\mathrm{Comp}(P)$ is the standard *completion* of P and $\mathrm{Rank}(P)$ contains the (strong) *ranking constraints* of P (see [8] for details). To address rules of the form (5), we first translate them into rules $a \leftarrow b_1,\ldots,b_m, d_1,\ldots,d_l, \text{not } c_1,\ldots,\text{not } c_n$, accompanied by defining rules $d_1 \leftarrow t_1; \ldots; d_l \leftarrow t_l$ for the new atoms $d_1,\ldots,d_l$. These two forms of rules give rise to the respective parts for the translation, i.e., the *normal form* $\mathrm{NForm}(P)$ of P and the *theory part* $\mathrm{TPart}(P)$ of P. Then $\langle M, T\rangle$ is an answer set of a program P iff $\langle M \cup D, T\rangle$ is an answer set of $\mathrm{NForm}(P) \cup \mathrm{TPart}(P)$ where $D = \{d \mid d \leftarrow t \in \mathrm{TPart}(P) \text{ and } t \in T\}$.

We use a translation $\mathrm{Comp}(\mathrm{NForm}(P)) \cup \mathrm{Rank}(\mathrm{NForm}(P)) \cup \mathrm{Comp}(\mathrm{TPart}(P))$ into DL in our implementation. The part $\mathrm{Comp}(\mathrm{TPart}(P))$ consists of an equivalence $d \leftrightarrow t$ for each $d \leftarrow t$ in $\mathrm{TPart}(P)$. Then, using the definition of D above, $\langle M, T\rangle$ is an answer set of $P \iff M \cup D$ is an answer set of $\mathrm{NForm}(P)$ and $\langle D, T\rangle$ is an answer set of $\mathrm{TPart}(P) \iff$ there are models $\langle M \cup D, \tau_1\rangle \models \mathrm{Comp}(\mathrm{NForm}(P)) \cup \mathrm{Rank}(\mathrm{NForm}(P))$ and $\langle D, \tau_2\rangle \models \mathrm{Comp}(\mathrm{TPart}(P)) \iff$ there is a combined model $\langle M \cup D, \tau\rangle \models \mathrm{Comp}(\mathrm{NForm}(P)) \cup \mathrm{Rank}(\mathrm{NForm}(P)) \cup \mathrm{Comp}(\mathrm{TPart}(P))$. Thus, given any model $\langle M \cup D, \tau\rangle$ for the translation, an answer set of P can be extracted.

6 Preliminary Experiments

We compare the running time and the increasing of the size of ground instances for ASP(DL) and pure ASP programs. For benchmarks, we selected the newspaper problem as a representative for problems involving a time domain and the sorting problem that does not. We ran DINGO for ASP(DL) encodings and CLINGO (version 2.0.3) [4] for pure ASP encodings. All experiments were run on Ubuntu 9.10 workstation with two 2GHz CPUs and 4GB RAM. For each problem, we experimented on 10 groups of instances and each group contains 100 randomly generated instances. The results are shown in Table 1 and Table 2. The first column of Table 1 gives the deadline of each group, i.e., the time in which all the persons have to finish reading all the newspapers and the first column of Table 2 gives how many numbers are to be sorted in the instances of each group. The running times are reported in seconds. We set the cut off time to 300 seconds and running times greater than that are indicated by '–' in the tables. The increasing sizes of ground instances are reported as ratios with respect to the size of the smallest instance, for which the ratio is 1.0 by definition.

As it is easy to inspect from Table 1 for the newspaper problem, DINGO is around 2 orders of magnitude faster than CLINGO for the instances solvable by both of them. In addition, DINGO can solve the instances unsolvable to CLINGO in a very small amount of time. We think the reason lies in the sizes of the resulting ground instances: the size

Table 1. Newspaper

Deadline	DINGO		CLINGO	
	time	size ratio	time	size ratio
100	0.09	1.0	2.10	1.0
200	0.11	1.1	9.00	3.1
300	0.11	1.3	21.32	6.3
400	0.10	1.4	36.68	15
500	0.12	1.5	61.15	23
600	0.12	1.7	93.51	34
700	0.11	1.8	–	44
800	0.11	1.9	–	60
900	0.12	2.1	–	74
1000	0.13	2.2	–	81

Table 2. Sorting

Numbers	DINGO		CLINGO	
	time	size ratio	time	size ratio
60	0.59	1.0	13.12	1.0
70	0.77	1.3	25.50	2.1
80	1.01	1.8	49.70	2.7
90	1.26	2.3	76.14	4.9
100	1.54	2.8	145.43	7.8
110	1.84	3.4	–	12
120	2.25	4.1	–	17
130	2.71	4.8	–	28
140	3.17	5.6	–	34
150	3.56	6.4	–	38

of the encodings in ASP(DL) increases significantly slower than that in pure ASP, e.g., the maximum ratio of ASP(DL) encoding is 2.2 while that of pure ASP encoding is 81. Similar observations can be found for the sorting problem.

We also tried out 516 benchmark instances from the category of NP-complete problems that were used in the Second ASP Competition [3]. The idea was to check the performance of DINGO for programs not involving difference constraints. It turned out that a level of performance that is comparable to using LP2DIFF with Z3 as a back-end solver [8] can be achieved. However, this presumes the use of SMODELS for simplifying ground programs before DL translation. Further work is required to implement similar simplification in DINGO because SMODELS does not natively support external atoms.

7 Conclusion

In this paper, we present an approach to integrate the languages used in ASP and SMT. At the syntactic level, the idea is to enrich rules with extra conditions which together form an SMT theory pertaining to a particular SMT fragment. This leads to a straightforward generalization of the answer set semantics for programs extended in this way. The class of normal programs is formally handled in the current paper but a similar approach carries over to its extensions. Such possibilities are already illustrated in terms of cardinality constraints and difference constraints on the modeling side. These syntactic elements are also fully supported by our preliminary implementation, viz. DINGO, that exploits off-the-shelf ASP and SMT components for grounding (GRINGO) and the search for answer sets (Z3). As regards grounding the theory part, it is realized by first grounding macros corresponding to theory atoms which are expanded after grounding to full SMT syntax using a standard macro processor (M4). Following such a delayed macro expansion strategy, we are able to apply standard ASP methodology in the creation of SMT theories of interest. As illustrated in the paper, the combined language ASP(DL) enables more concise ways to encode problems from various domains. Some of the encodings are clearly more verbose in plain ASP or SMT. Our first experiments using these encodings also suggest positive effects on solving time.

As regards future work and lines of research, we think that also other SMT dialects should be taken into consideration. The current implementation is basically easy to extend for new fragments and back-end SMT solvers. The main effort in this respect is to identify required SMT primitives, to introduce macros for them, to provide definitions for such macros. If the output of LP2DIFF[4] cannot be exploited as part of the SMT translation, it may be necessary to modify LP2DIFF for the SMT dialect in question.[6] There is also potential for combining SMT dialects as long as there is a suitable target language and a back-end SMT solver available. It is also feasible to develop ASP(SMT) encodings in a modular way. The intermediate ground programs produced by GRINGO can be linked together using LPCAT from the ASPTOOLS[2] collection. At least for the moment, it is essential to do this before macro expansion (and SMT translation) because we are not aware of any linkers for SMT theories themselves.

References

1. M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *Proc. ASPOCP*, 2009.
2. S. Cotton and O. Maler. Fast and flexible difference constraint propagation for DPLL(T). In *Proc. SAT'06, pages 12-15*, pages 170–183, 2006.
3. M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczynski. The second answer set programming competition. In *Proc. LPNMR'09*, pages 637–654, 2009.
4. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In *Proc. LPNMR'07*, pages 386–392, 2007.
5. M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proc. ICLP'09*, pages 235–249, 2009.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP'88*, pages 1070–1080, 1988.
7. J. Huang. Universal Booleanization of constraint models. In *Proc. CP'08*, pages 144–158, 2008.
8. T. Janhunen, I. Niemelä, and M. Sevalnev. Computing stable models via reductions to difference logic. In *Proc. LPNMR'09*, pages 142–154, 2009.
9. V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. 1999.
10. V. S. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *AMAI*, 53(1-4):251–287, 2008.
11. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *AMAI*, 25(3-4):241–273, 1999.
12. I. Niemelä. Stable models and difference logic. *AMAI*, 53(1-4):313–329, 2008.
13. R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Proc. CAV'05*, pages 321–334, 2005.
14. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, 2006.
15. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.
16. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[6] This should be straightforward and we have already done this for theories involving bit vectors.

An Experiment on Tabled Evaluation for Hidden Predicates*

Pedro Cabalar and Martín Diéguez

Department of Computer Science,
University of Corunna (Spain)
{cabalar,martin.dieguez}@udc.es

Abstract. Most Answer Set Programming grounders allow filtering the information shown in each answer set by selecting which predicates must be shown or become hidden instead. Full grounding of hidden predicates (usually introduced for auxiliary computations) is in many cases not necessary. In the paper, we consider the possibility of avoiding grounding of hidden (stratified) predicates by using tabled evaluation: a top-down strategy combined with an extension table that records the previously solved goals.

1 Introduction

Answer Set Programming (ASP) [1, 2] has recently become one of the most popular and successful paradigms for Nonmonotonic Reasoning (NMR) and Knowledge Representation (KR). Part of this success is due to the simplicity and robustness of its theoretical foundations which rely on the notion of *answer set* or *stable model* [3]. Originally defined for normal logic programs, this semantics proved to be flexible enough to end up covering the general syntax of arbitrary first order theories [4, 5] and accommodating useful operators for practical KR like weight constraints [6] or aggregates [7, 8]. A second, but not less important factor for the success of ASP has been the availability of efficient solvers that were boosted both in number and performance, thanks to the establishment of a competition [9] with public benchmarks and results. Most ASP solvers divide their computation into two differentiate steps: in a first phase, called *grounding*, variables in each rule are replaced by all their possible combinations of ground instances (the constants in the program). In a second step, the solver computes answer sets for the resulting ground program. Some tools like `lparse`[1] or `gringo`[2] are distributed as separate grounders that must be combined with solvers that exclusively accept propositional programs as an input, like `smodels`[3], `clasp`[4],

* This research was partially supported by Spanish MEC project TIN2009-14562-C05-04 and Xunta de Galicia project INCITE08-PXIB105159PR.

[1] `http://www.tcs.hut.fi/Software/smodels/`

[2] `http://sourceforge.net/projects/potassco/files/gringo/`

[3] `http://www.tcs.hut.fi/Software/smodels/`

[4] `http://sourceforge.net/projects/potassco/files/clasp/`

`cmodels`[5] or `assat`[6]. Other tools like `DLV`[7] embody the grounder and the propositional solver in a same package.

Although the input languages of `lparse`, `gringo` and `DLV` have some differences, the three tools allow selecting which predicates are shown in the obtained stable models and which ones can be hidden. Typically, these hidden predicates contain intermediate or auxiliary information that is actually irrelevant for describing the final solutions of the problem we are interested in. The three mentioned ASP grounders use this feature as a simple output choice, so that when a predicate is hidden, it is just filtered out from the information displayed in an answer set, but its grounding and computation is not actually affected. However, the following question arises: once we know that the information provided by a predicate will be irrelevant, is it always necessary to compute its full extent?

In this paper we consider a partial grounding of hidden predicates, so that their extent is computed depending on the values of variables fixed by other predicates. In this way, the hidden predicate is called as a top-down query rather than computing its full extent using a bottom-up strategy. In order to avoid repeating subgoals due to multiple recursive calls, instead of a pure top-down strategy, we propose applying a technique called *tabled evaluation* [10] that combines top-down queries with an extension table that keeps record of previously solved goals. In fact, this technique has been implemented and extensively used in the `XSB`[8] Prolog interpreter, which has been used as a back-end in our preliminary experiments.

The rest of the paper is organised as follows. In the next section we present a motivating example that is followed in Section 3 by a description of the prototype we have implemented and a preliminary experiment, commenting the obtained results. Section 4 contains a brief discussion and concludes the paper.

2 A Motivating Example

To illustrate our main purpose, consider the following motivating example.

Example 1. Suppose we have a directed graph where some nodes are marked as *source* and that we want to generate arbitrary sets of nodes that are reachable from a source. To this aim, we include a predicate $in(X)$ that points out that node X is in our current selection.

[5] `http://www.cs.utexas.edu/users/tag/cmodels.html`
[6] `http://assat.cs.ust.hk/`
[7] `http:www.dbai.tuwien.ac.at/proj/dlv/`
[8] `http://xsb.sourceforge.net/`

A possible program that solves this problem and is syntactically accepted by the three grounders mentioned before is the following one:

$$reach(X,Y) \leftarrow edge(X,Y) \tag{1}$$
$$reach(X,Z) \leftarrow node(X), reach(X,Y), edge(Y,Z) \tag{2}$$
$$in(Y) \leftarrow source(X), reach(X,Y), not\ out(Y) \tag{3}$$
$$out(Y) \leftarrow source(X), reach(X,Y), not\ in(Y) \tag{4}$$

As usual in ASP, the cyclic rules (3)-(4) involving an auxiliary predicate $out(X)$, complement of $in(X)$, are used to generate possible solutions. Each instance of this problem would be given as a set of facts for predicates $node(X)$, $edge(X,Z)$ and $source(X)$. For instance, the graph depicted in Figure 1 (double circled nodes are sources) would be represented by the set of facts:

$$node(1..9).\ source(3).\ source(5).$$
$$edge(1,2).\ edge(2,3).\ edge(1,4).\ edge(2,5).$$
$$edge(3,6).\ edge(4,5).\ edge(5,6).\ edge(4,7).$$
$$edge(5,8).\ edge(6,9).\ edge(7,8).\ edge(8,9).$$

As only three nodes, 6, 8 and 9, are reachable from source nodes, it is easy to check that the resulting program has eight answer sets, that correspond to all solutions, i.e., all possible subsets of $\{in(6), in(8), in(9)\}$. The complete answer sets, however, will contain much more information than the extent for predicate $in(X)$. In particular, each answer set will contain the facts we entered to describe the graph plus the extent of predicates $reach(X,Y)$ and $out(X)$. For instance, all answer sets will include the following 27 facts for predicate $reach(X,Y)$:

$$reach(1,2), reach(1,3), reach(1,4), reach(1,5), reach(1,6), reach(1,7),$$
$$reach(1,8), reach(1,9), reach(2,3), reach(2,5), reach(2,6), reach(2,8),$$
$$reach(2,9), reach(3,6), reach(3,9), reach(4,5), reach(4,6), reach(4,7),$$
$$reach(4,8), reach(4,9), reach(5,6), reach(5,8), reach(5,9), reach(6,9),$$
$$reach(7,8), reach(7,9), reach(8,9)$$

Since our interest is focused on the solutions in terms of $in(X)$, we would typically hide the rest of predicates. In `lparse` and `gringo` this is done by including the clauses:

$$hide.$$
$$show\ in(X).$$

whereas in `DLV` we would include the command line option `-filter=in` for the same purpose. In the three grounders, hiding the extent of a predicate does not have any real implication on the grounding or computation of the answer sets: it just filters the facts that are shown in the output, but all the information is computed anyway. However, if we look carefully at our example, it can be noticed that computing reachable nodes from $1, 2, 4$ and 7 could be avoided if we

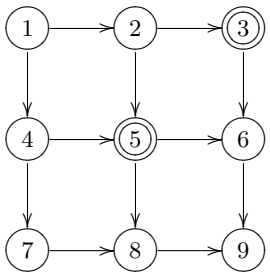


Fig. 1. An example of directed graph with two "*source*" nodes.

just started from source nodes 3 and 5 and followed predicate *reach* on demand. For instance, once we fix $X = 3$ in rule (3) as one of the possible values obtained from $source(X)$, we would make a call to $reach(3, Y)$ and try to solve it using a top-down strategy. By recursive applications of (1)-(2) we would then obtain the values $Y = 6$ and $Y = 9$, which would finally allow us generating the ground rules:

$$in(6) \leftarrow not\ out(6)$$
$$in(9) \leftarrow not\ out(9)$$

Using this strategy, we would never need computing facts for $reach(1, Y)$, $reach(2, Y)$, $reach(4, Y)$ or $reach(7, Y)$, something that reduces the 27 facts obtained before to only the following 7:

$$reach(3, 6), reach(3, 9), reach(5, 6), reach(5, 8),$$
$$reach(5, 9), reach(6, 9), reach(8, 9)$$

A pure top-down evaluation, however, introduces a new problem, not present in bottom-up computation: some goals are called more than once and its computation must be repeated. As a simple example, suppose we marked node 1 as source, making $source(1)$ true. Then, we would be continuously repeating goals: for instance, $reach(6, 9)$ would also be called in the computation of each $reach(Y, 9)$ for all nodes $Y = 1, \ldots, 5$, that is, those that pass through node 6 to reach 9. In an example like that, the bottom-up procedure would be much more efficient since, after all, there is no way to avoid the full computation of $reach(X, Y)$ (all nodes would be reachable from 1).

To avoid this goal repetition, we have considered an alternative strategy called *tabled evaluation* [10] that can be seen as a midpoint between bottom-up and top-down. It consists in maintaining a table with previously solved goals so that they can be reused without repeating their computation. In our worst case version of the example when $source(1)$, if we compare it to the bottom-up strategy, this technique would only introduce the cost of checking the existence of each goal in the extension table.

2.1 Stratified predicates

Making a partial grounding for hidden predicates is not always applicable, as the non-computed part may affect the existence of answer sets for a given program. For instance, suppose we want to rule out solutions where we pick nodes that are reachable from 7. This is done using the constraint:

$$\leftarrow node(X), in(X), reach(7, X)$$

A constraint like this is usually implemented by introducing a new fresh auxiliary predicate $aux(X)$ and modifying the rule as follows:

$$aux(X) \leftarrow node(X), in(X), reach(7, X), not\ aux(X) \tag{5}$$

Since $aux(X)$ will be a hidden predicate and *is never used* in the rest of the program, a partial grounding of $aux(X)$ would just ignore rule (5). However, this constraint should be grounded at least for values $X = 8$ and $X = 9$ (that is, the accessible nodes from 7):

$$aux(8) \leftarrow in(8), not\ aux(8)$$
$$aux(9) \leftarrow in(9), not\ aux(9)$$

and this should rule out all previous solutions that contained atoms $in(8)$ or $in(9)$. Furthermore, a simple rule like:

$$aux'(X) \leftarrow node(X), not\ aux'(X)$$

would make the program to have no answer sets (provided that we have at least one node) whereas partial grounding would ignore predicate $aux'(X)$ if it is not used in other rules.

These examples show that, in order to ignore some ground instances of a hidden predicate, we must be sure that this will not affect the set of answer sets of the program. As a simple sufficient condition, we have considered the case in which the hidden predicate is *stratified* [11]. The dependence graph of a program is formed by taking a node per each predicate and an arc $p \rightarrow q$ (meaning that p *directly depends on* q) when p is in the head of some rule in which q occurs in the body. The dependence is said to be *negative* when q is in the scope of default negation *not* , and said to be *positive* otherwise. A program is *stratified* if it contains no dependence loop that involves some negative dependence.

We say that p *depends on* a rule r, if p occurs in the head of r or there exists some path in the dependence graph from p to a predicate q occurring in the head of r. Given a program Π and a predicate p, let Π_p denote the set of rules on which p depends. A predicate p is *stratified* iff Π_p is a stratified program. Let us take now the set of stratified hidden predicates P and let Π_P the union of all Π_p for each hidden predicate $p \in P$. It is clear that we can split program Π into two parts, $bottom(\Pi) \stackrel{\text{def}}{=} \Pi_P$ and $top(\Pi) \stackrel{\text{def}}{=} \Pi \setminus \Pi_P$ that satisfy that no predicate in $bottom(\Pi)$ depends on rules in $top(\Pi)$. In this situation, we can

apply the *splitting theorem* from [12] to show that the answer sets of Π can be computed in two stages: first obtain an answer set I of $bottom(\Pi)$ (in our case, I is unique) and second, simplify program $top(\Pi)$ with the information in I for predicates defined in $bottom(\Pi)$, getting answer sets for the rest of facts for remaining predicates.

In the previous example, $bottom(\Pi)$ consists of rules (1) and (2) that define predicate *reach* plus the following facts (i.e., those on which these rules depend):

$$node(1..9).\ edge(1,2).\ edge(2,3).$$
$$edge(1,4).\ edge(2,5).\ edge(3,6).\ edge(4,5).\ edge(5,6).$$
$$edge(4,7).\ edge(5,8).\ edge(6,9).\ edge(7,8).\ edge(8,9).$$

On the other hand, $top(\Pi) = \Pi \setminus bottom(\Pi)$ contains the rest of rules and facts in Π. Note that no atom of $bottom(\Pi)$ occurs in any rule head of $top(\Pi)$.

As a result, if part of the unique answer set I for the stratified bottom program Π_P is not actually computed, this will not affect the answer sets of the program Π provided that we actually generate the full ground program for $top(\Pi)$.

3 The experiment

As a first prototype, we have implemented an XSB Prolog program that reads an ASP program as an input. After selecting a set P of intensional hidden predicates that are stratified, we call DLV with a modification of the rest of rules $\Pi \setminus \Pi_P$ to ground all variables that can be fixed by the rest of predicates not in P. For instance, in our previous example, we would take $P = \{reach\}$ and transform rules (3)-(4) so that we capture variables that can be independently fixed without *reach* predicate:

$$aux_1(X) \leftarrow source(X)$$
$$aux_2(X) \leftarrow source(X)$$

In this way, predicate aux_1 will capture instances for variable X in (3) and aux_2 instances for X in (4). As a result, we get the partially ground program:

$$in(Y) \leftarrow reach(3,Y), not\ out(Y) \quad (6)$$
$$in(Y) \leftarrow reach(5,Y), not\ out(Y) \quad (7)$$
$$out(Y) \leftarrow reach(3,Y), not\ in(Y) \quad (8)$$
$$out(Y) \leftarrow reach(5,Y), not\ in(Y) \quad (9)$$

After this first step, we use the XSB `assert` mechanism to include the rules (1)-(2) for predicate *reach* so that any call to this predicate will be solved using the built-in tabled evaluation algorithm of XSB. Finally, for each partially

grounded rule, we take non-ground atoms for hidden predicates and make the corresponding call to XSB. For instance, for (6), we make the call $reach(3, Y)$ obtaining the possible solutions $Y = 6$ and $Y = 9$ which eventually generate the ground rules:

$$in(6) \leftarrow not\ out(6)$$
$$in(9) \leftarrow not\ out(9)$$

We have performed some preliminary experiments with some random graph instances for problem in Example 1. For each random graph, we have varied the number of nodes, the *connectivity* (the number of outgoing arcs per node) and we have tried with one or two random source nodes. Results are shown in Table 1 where we compare the time obtained by a direct single call to DLV to the proposed combined execution of DLV and XSB that makes a partial grounding of hidden predicates. For graphs that are very connected, most nodes can be reached from all the rest. Thus, we can notice that the time for DLV+XSB is not significantly better or can be even worse due to additional processing and the checkings in the extension table. However, for small connectivity ratios, the time is considerably reduced. The smaller the relevant part of the graph to be traversed, the better performance obtained.

Nodes	**Connectivity**	**DLV**	**DLV+XSB**
1 source node			
500	10	0m34.581s	2m44.064s
500	20	5m14.837s	2m14.965s
500	30	9m22.146s	4m42.853s
500	40	12m20.197s	8m13.163s
500	60	17m40.125s	17m51.523s
500	100	24m2.454s	48m32.006s
2 source nodes			
600	20	4m54.504s	1m20.178s
620	20	10m32.675s	3m47.708s
640	20	12m7.537s	4m0.469s
660	20	14m3.903s	4m16.870s
700	20	16m39.363s	4m58.550s

Table 1. Running times for randomly generated graphs for problem in Example 1.

As an exaggerated case, we have constructed a graph similar to Figure 1 but of 100×100 nodes instead of 3×3, fixing 9850 as the only source node. This means that only 101 nodes are reachable from the source whereas the bottom-up computation will compute more than 10000 facts for $reach(X, Y)$ for the whole graph. DLV was able to solve the problem in 903 minutes versus DLV+XSB that took slightly more than 8 minutes.

Of course, it may be objected that the particular example we chose could have been implemented in a different way much more favorable to an efficient (complete) grounding for the problem to be considered. For instance, if we write the alternative encoding:

$$\begin{aligned} reach_s(Y) &\leftarrow source(X), edge(X,Y) \\ reach_s(Y) &\leftarrow reach_s(X), edge(X,Y) \\ in(X) &\leftarrow node(X), reach_s(X), not\ out(X) \\ out(X) &\leftarrow node(X), reach_s(X), not\ in(X) \end{aligned}$$

where this time, predicate $reach_s$ means reachable from some source node, we would obviously reduce the number of ground instances for $reach_s$ to be considered, so that partial grounding would become much less interesting. On the other hand, we cannot always expect that the program we get as an input is the best encoding for an efficient grounding. In fact, we claim that adapting each problem encoding to reduce the grounding effort usually reduces flexibility: for instance, it could be the case that rules for predicate $reach(X, Y)$ came from a general module for reachability that could be reused for other different problems.

4 Conclusions

We have considered the partial grounding of hidden predicates by treating body atoms for them as queries to be solved using tabled evaluation. We have implemented a first prototype that combines the use of an existing grounder (DLV) with the use of the XSB Prolog interpreter, which has a built-in tabled evaluation procedure. First results are promising but the application of this technique is still in a preliminary stage.

The combination of a partial grounding process plus an external tool for completing the remaining non-ground information is very similar to the technique applied in [13] where, in that case, the non-ground atoms obtained after the partial grounding correspond to constraints to be solved by a Constraint Logic Programming (CLP) tool. A relevant difference between both approaches is that, in our case, our technique does not require any syntactic or semantic extension and can be transparently applied to existing programs with the only extra requirement of specifying hidden predicates, something already present in all the existing grounders, but not currently exploited for reducing grounding.

In a future extended version of this document, we will include formal proofs for the correctness of the method. Future work also includes encoding a specialised tabled evaluation algorithm on top of the existing open source grounder `gringo`, for applying this technique on (stratified) hidden predicates.

References

1. Marek, V., Truszczyński, M. In: Stable models and an alternative logic programming paradigm. Springer-Verlag (1999) 169–181
2. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25** (1999) 241–273
3. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R.A., Bowen, K.A., eds.: Logic Programming: Proc. of the Fifth International Conference and Symposium (Volume 2). MIT Press, Cambridge, MA (1988) 1070–1080
4. Pearce, D., Valverde, A.: Quantified equilibrium logic and foundations for answer set programs. In: Proc. of the 24th Intl. Conf. on Logic Programming (ICLP'08). (2008) 547–560
5. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07). (2007) 372–379
6. Niemelä, I., Simons, P.: Extending the smodels system with cardinality and weight constraints. In: Logic-Based Artificial Intelligence, Kluwer Academic Publishers (2000) 491–521
7. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate well-founded and stable semantics for logic programs with aggregates. In: 17th International Conference on Logic Programming (ICLP'01). (2001) 212–226
8. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Proc. of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04). (2004)
9. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second Answer Set Programming competition. In: Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). LNAI (5753), Springer-Verlag (2009) 637–654
10. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM **43**(1) (1996) 20–74
11. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987)
12. Lifschitz, V., Turner, H.: Splitting a logic program. In: Proceedings of the 11th International Conference on Logic programming (ICLP'94). (1994) 23–37
13. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence **53**(1-4) (2008) 251–287

Handling Function Symbols in the DLV Grounder

Francesco Calimeri[1], Susanna Cozza[2], and Simona Perri[1]

[1] Department of Mathematics, University of Calabria,
P.te P. Bucci, Cubo 30B, I-87036 Rende, Italy
[2] DEIS, University of Calabria,
P.te P. Bucci, Cubo 42C, I-87036 Rende, Italy
{calimeri,cozza,perri}@mat.unical.it

Abstract. This paper describes how the grounder of the ASP system DLV handles programs with function symbols, briefly illustrating two different strategies alongside the results of a preliminary experimental activity. The grounder is able to deal with ASP programs making an unrestricted use of function symbols, and it can perform an on-demand check in order to ensure that the input falls into the class of *argument-restricted* programs, thus guaranteeing the termination.

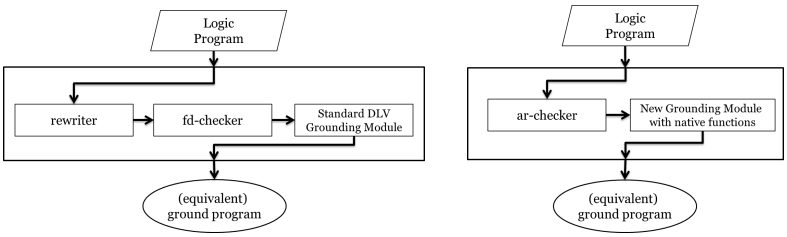
1 Introduction

Answer Set Programming (ASP) is a highly expressive language, which, due to the presence of disjunction and negation, allows the use of nondeterministic definitions for modelling complex problems in computer science, in particular in Artificial Intelligence. Traditionally, ASP has often been used as a first-order language without function symbols, similar to Datalog, in order to deal with finite structures only. More recently, also uninterpreted function symbols have been frequently considered, thus enabling a natural representation of complex and recursive structures, such as strings, lists, stacks, trees and many other. In the latest years, ASP became widely used for knowledge representation and reasoning in many application scenarios, also thanks to the blooming of robust and efficient systems. Some of these systems started to support function symbols to some extent. Unfortunately, the presence of function symbols within ASP programs has a strong impact on the grounding process, which might even not terminate: the common reasoning tasks are indeed undecidable, in the general case.

In this paper we discuss some advances recently introduced in the grounder of the ASP system DLV [9], concerning the management of programs with functions. In particular, a new approach, relying on functional terms as native structures, has been implemented, which brings relevant performance improvements w.r.t. the previous one, which was based on rewriting. Moreover, a new checker has been implemented, that allows the user to statically recognize if the input program belongs to a class for which the termination of the grounding process can be guaranteed: the class of *argument-restricted* programs [10], which is larger than the class recognized by DLV so far. The results of an experimental analysis are also reported, confirming a significant gain in terms of efficiency w.r.t. the old approach.

2 Preliminaries

This Section shortly reports some preliminaries about the language herein considered, i.e. ASP with functions.



(a) old grounding framework (b) new grounding framework

Fig. 1. New and old System Architectures

A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant or a variable. If $t_1 \dots t_n$ are terms and f is a function symbol (*functor*) of arity n, then $f(t_1, \dots, t_n)$ is a *functional term*.

If $t_1, \dots, t_k$ are terms, then $p(t_1, \dots, t_k)$ is an *atom*, where p is a predicate of fixed arity $k \geq 0$; by $p[i]$ we denote its i-th argument. Atoms prefixed by # are instances of *built-in* predicates: such kind of atoms are evaluated as true or false by means of operations performed on their arguments, according to some predefined semantics.[3]

A *literal* l is of the form a or `not` a, where a is an atom; in the former case l is *positive*, and in the latter case *negative*. A *rule* r is of the form $\alpha_1 \vee \dots \vee \alpha_k$:– $\beta_1, \dots, \beta_n$, `not` $\beta_{n+1}, \dots,$ `not` β_m. ($m \geq 0, k \geq 0$; $\alpha_1, \dots, \alpha_k$ and $\beta_1, \dots, \beta_m$ being atoms). We define $H(r) = \{\alpha_1, \dots, \alpha_k\}$ (the *head* of r) and $B(r) = B^+(r) \cup B^-(r)$ (the *body* of r), where $B^+(r) = \{\beta_1, \dots, \beta_n\}$ (the *positive body* of r) and $B^-(r) = \{$`not` $\beta_{n+1}, \dots,$ `not` $\beta_m\}$ (the *negative body* of r). If $H(r) = \emptyset$ then r is a *constraint*; if $B(r) = \emptyset$ and $|H(r)| = 1$ then r is referred to as a *fact*.

A rule is safe if each variable in it appears in at least one positive literal in the body. For instance, the rule $p(X, f(Y, Z))$:– $q(Y)$, `not` $s(X)$. is not safe, because of both X and Z. From now on we assume to deal only with safe rules. An ASP program is a finite set P of rules. As usual, a program (a rule, a literal) is said to be *ground* if it contains no variables.

A thorough discussion about the semantics, based on the notion of Answer Set proposed in [8], can be found in [2].

3 Advances in Grounding Programs with Functional Terms

In the following, we present a new version of the DLV grounder which is able to deal with ASP programs written in the language of Section 2, which has been conceived in order to improve the old one. The two system architectures are (at a very high level) depicted in Fig. 1.

3.1 Native Functional Terms vs Rewriting

As reported in [3], the support for functional terms has firstly been added to the state-of-the-art ASP system DLV [9] by means of a proper rewriting strategy. The resulting system, named DLV-COMPLEX [6], includes a rewriting module that removes all functional terms, introducing a number of instances of proper predefined built-in predicates.

[3] The system herein presented provides some simple built-in predicates, such as the comparative predicates equality, less-than, and greater-than ($=, <, >$).

The grounding module is fed with a new program without functional terms: no actual change is then needed in the grounding process already implemented in DLV.

More in detail, any functional term $t = f(X_1, \ldots, X_n)$ appearing in some rule $r \in P$ is replaced by a fresh variable F, and one of the following atom is then added to $B(r)$:

- $\#function\_pack(f, X_1, \ldots, X_n, F)$ if either t appears in $H(r)$ or t appears in $B^-(r)$;
- $\#function\_unpack(F, f, X_1, \ldots, X_n)$ if t appears in $B^+(r)$.

Example 1. The rule: $p(f(f(X))) :- q(X, g(X, Y))$. will be rewritten as follow:

$$p(F_1) :- \#function\_pack(F_1, f, F_2), \#function\_pack(F_2, f, X), q(X, F_3), \#function\_unpack(F_3, g, X, Y).$$

Note that rewriting the nested functional term $f(f(X))$ generates two instances of $\#function\_pack$ in the body: (i) for the inner f function having X as argument and (ii) for the outer f function having as argument the fresh variable F_2, representing the inner functional term. □

In practice, both $\#function\_pack$ and $\#function\_unpack$ built-in predicates act on strings; the former builds a new string concatenating a functor with its arguments, while the latter unfolds the functional term (again, represented by a string) identifying the functor and the list of its arguments, in order to provide values to the variables of the $\#function\_unpack$ built-in.

On the one hand, such rewriting strategy allowed, at the time of implementing, an effective support to functions without any drastic change in the grounding process; on the other hand, it suffers from many drawbacks, if efficiency is taken into account: apart from the rewriting time, which is negligible, the approach clearly pays a price while manipulating strings (storing, tokenizing, comparing, etc.), which grows more and more in case of programs that make a massive use of functional terms.

Experiences with such drop of performances pointed out the need for a re-engineering of the grounding process while dealing with functions. First of all, we redesigned the involved DLV internal data structure for managing functional terms as native data, in order to avoid expensive string manipulation. We roughly describe next the most relevant changes introduced into the grounder; going too much into implementation details is out of the scope of this work.

The DLV grounder originally supported three kind of terms: strings constants, integer constants, and variables; we introduced functions. For performance reasons, each term was internally associated with an integer value representing the index of the position in a proper table, in which its actual value is stored. Clearly, such a table were not suitable for storing a functional term, because of the need to store also the list of its arguments. Note that arguments of a function are also (possibly functional) terms: thus, a sort of recursive structure is needed. For representing functional terms, we exploited a new table storing functors, in order to associate each functor with an integer; then, an additional table (for functional terms) is used, where each entry is a pair of a functor index and a list of object terms representing its arguments (which, in turn, might be functional terms). More in detail, each object term features a flag telling its type (functional, variable, etc.) and an index; the table referred by the index depends on the flag. Note that, if the argument of a functional term is, again, functional, the index "recursively" refers to another entry in the functional term table. It is also worth noting that a functional term might be both a variable and a constant, but this makes no difference in terms of its representation. In particular, a functional term is considered as a variable if at least one argument in its list is a variable (or a variable functional term); it is treated as a constant otherwise.

The introduction of a new type of term has a strong impact on the whole grounding process: it requires relevant changes to many subtasks such as body reordering, indexing, backjumping, matching. However, it brings also significant performance improvements (see Section 4). In particular, the matching procedure takes great advantage from the possibility to use integers instead of strings while comparing functional terms.

3.2 Argument-Restricted vs Finite-Domain Programs

Since the grounding process for an ASP program with functional terms might not terminate [4], checking if termination can be "a priori" guaranteed is a useful feature in many cases.

The DLV-COMPLEX [3] system implemented a module for recognizing the membership to a class of programs that is known to be computable, namely the class of *finite-domain* programs, introduced in [2]. Other computable classes of ASP programs with functions have been identified from then on; in particular, the class of *argument-restricted* programs [10] significantly extends both the class of finite-domain and the class of λ-*restricted* [7] programs. According to [10], a program is argument-restricted if it has an argument ranking: basically, if it is possible to find a mapping from each argument of each predicate appearing in the program to an integer, such that a set of given conditions is satisfied. Such a condition is constrained for each variable in each predicate argument appearing in the head of each rule, and takes into account the head-body differences between nesting levels and values of argument ranking. Intuitively, it might be seen as a sort of stratification, but applied to argument predicates instead of atoms.

Example 2. The following program is not a finite-domain program, but it is argument-restricted.

$$p(f(X)) \text{ :- } q(X). \qquad q(X) \text{ :- } p(X), r(X). \qquad r(0). \qquad p(0).$$

Roughly (see [10] for a formal definition), let nl(var, atom) denote the nesting level of a variable in an atom; then, the inequalities to fulfill are:

$$rank(p[1]) - rank(q[1]) \geq nl(X, p(f(X))) - nl(X, q(X)) \quad \text{(for the first rule)}$$
$$rank(q[1]) - rank(r[1]) \geq nl(X, q(X)) - nl(X, r(X)) \quad \text{(for the second rule)}$$

An argument ranking exists in this case: indeed, the mapping:

$$rank(p[1]) = 1 \qquad rank(q[1]) = rank(r[1]) = 0$$

satisfies both inequalities above. □

Example 3. The following program is not argument-restricted.

$$p(f(X)) \text{ :- } p(X). \qquad p(0).$$

For this program the inequality to be fulfilled is:

$$rank(p[1]) - rank(p[1]) \geq nl(X, p(f(X))) - nl(X, p(X))$$

but there cannot exist any argument ranking in this case, since every assignment for $rank(p[1])$ would fail (the inequality above becomes $0 \geq 1$). □

test	DLV-rewriting	DLV-native	clingo
cycles.1	313,3	51,0	308,1
cycles.2	225,1	96,7	187,6
cycles.3	26,6	5,0	76,7
cycles.4	19,0	6,5	134,2
cycles.5	20,1	8,7	166,6
hanoi.7discs.01	155,3	8,3	4,6
hanoi.7discs.02	194,7	10,0	5,4
hanoi.7discs.03	242,7	11,5	6,2
hanoi.7discs.04	7,5	0,4	0,2
hanoi.7discs.05	73,3	3,7	2,1
hanoi.7discs.06	98,0	4,5	2,6
hanoi.7discs.07	57,2	3,0	1,7
hanoi.7discs.08	82,5	4,3	2,5
hanoi.7discs.09	122,4	5,9	3,3
hanoi.7discs.10	87,4	4,8	2,7
palindromes.2	1031,3	1,3	1,1
palindromes.3	–	43,9	31,8
palindromes.4	–	16,9	10,5
palindromes.5	–	19,4	10,5
palindromes.6	–	200,9	98,5
paths.1	172,7	145,8	129,5
paths.2	172,7	145,6	129,4
paths.3	57,2	14,6	158,3
paths.4	15,2	11,7	220,8
paths.5	47,0	0,3	0,2

Table 1. Benchmark Results (times are reported in seconds)

The new prototype has been equipped with a proper module able to statically decide wether an input program is argument restricted, or not. The module, enabled by default and named "ar-checker", substitutes the "fd-checker" module of DLV-COMPLEX (Figure 1). If the user is confident that the program can be grounded in finite time (even if it does not belong to the class of argument-restricted programs), then she can disable the ar-checker; this can be done by specifying the command-line option `-nofinitecheck`. Another way for guaranteeing termination is the choice of a maximum allowed nesting level N for functional terms (command-line option `-maxnestinglevel=<N>`).

The implementation of the ar-checker exploits the idea depicted in [10]. Values for a ranking are computed as the least fixpoint of a monotone operator; since such a ranking, if it exists, cannot exceed an upper bound M (statically computable as the maximum nesting level times the total number of predicate arguments), our algorithm answers YES as soon as the fixpoint is reached, and NO in case some value becomes greater than M.

4 Experiments and Conclusions

We report in this Section the experimental activities we carried out with the aim to assess the impact of the new implementation approach on the DLV grounder performances. We start by briefly describing the benchmark problems considered.

Hanoi Towers. The classic Towers of Hanoi puzzle problem with three pegs and n disks of increasing size.
Paths in a Graph. Given a directed graph, find all paths of a maximum given length connecting each pair of nodes.
Cycles in a Graph. Given a directed graph, find all cycles involving at most a given number of nodes. A cycle has to be intended as a path (as in the previous problem) where the starting node and the ending node coincide.

Palindromic Words. Given a predefined set of chars, find all sequences of a given length that are equal to their reverse, and represent a meaningful word according to a given dictionary.

In order to properly test the system modules mainly involved in our work, we considered benchmark problems whose encodings heavily rely on function symbols, that is with many rules exploiting function symbols and involving the generation of a large number of ground atoms with functional terms. Moreover, for all benchmark the most part of computation time is spent in the grounding phase; in particular, all encodings but Hanoi Towers are normal and stratified. In order to meet space constraints, encodings are not shown here; for the sake of full reproducibility, encodings, instances and binary files can be found at a proper web link.[4] For each problem, we have randomly generated five instances; for hanoi towers, we took ten 7-disc instances from the second ASP Competition [5], and suitably adapted them in order to take advantage of functions.

The system tested are: DLV-*rewriting* (official DLV release 2010-10-14), wich consists of DLV featuring the rewriting strategy; DLV-*native* (the new developed prototype), that features native function terms; and *clingo* (version 3.0.3), a very efficient ASP solver [1]. For every instance, we allowed a maximum running (CPU) time of 900 seconds (15 minutes).

The machine exploited, featuring two Intel Xeon "Woodcrest" (quad-core) processors with 4MB of L2 cache, equipped with 4GB of RAM, runs GNU Linux Debian 4.1.1-21 (kernel 2.6.23.9 amd64 smp).

Results are reported in Table 1; a dash mark ("–") means that a system was not able to solve that instance within the allowed amount of time. In general, results confirm the intuition that the integration of function terms as native data, thus avoiding the rewriting phase involving heavy string manipulation, has a very positive impact on the efficiency of DLV. In particular, the new approach regularly outperforms the rewriting-based one. In many cases we noticed impressive speedups: for instance, DLV-rewriting was able to solve just one out of five instances of palindromic words, while DLV-native solved them all (in less than one minute in most cases).

Notably, the new approach allows DLV to bridge the performance gap with clingo, when a heavy usage of functional terms is required.

References

1. `clingo` homepage, `http://potassco.sourceforge.net/`
2. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: Proceedings of the 24th International Conference on Logic Programming (ICLP 2008). Lecture Notes in Computer Science, vol. 5366, pp. 407–424. Springer, Udine, Italy (Dec 2008)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: An ASP System with Functions, Lists, and Sets. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning — 10th International Conference (LPNMR 2009). Lecture Notes in Computer Science, vol. 5753, pp. 483–489. Springer Verlag (Sep 2009)
4. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys 33(3), 374–425 (2001)
5. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The Second Answer Set Programming Competition. In: Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, vol. 5753, pp. 637–654. Springer Berlin / Heidelberg (2009)

[4] `http://www.mat.unical.it/calimeri/downloads/gttv-2011-calimeri-etal.zip`

6. Faber, W., Pfeifer, G.: DLV homepage (since 1996), http://www.dlvsystem.com/
7. Gebser, M., Schaub, T., Thiele, S.: Gringo : A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07. Lecture Notes in Computer Science, vol. 4483, pp. 266–271. Springer Verlag, Tempe, Arizona (May 2007)
8. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
9. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (Jul 2006)
10. Lierler, Y., Lifschitz, V.: One More Decidable Class of Finitely Ground Programs. In: Proceedings of the 25th International Conference on Logic Programming (ICLP 2009). Lecture Notes in Computer Science, vol. 5649, pp. 489–493. Springer, Pasadena, CA, USA (Jul 2009)

Strong Equivalence of RASP Programs

Stefania Costantini[1], Andrea Formisano[2], and David Pearce[3]

[1] Università di L'Aquila, Italy stefania.costantini@univaq.it
[2] Università di Perugia, Italy formis@dmi.unipg.it
[3] Universidad Politècnica de Madrid, Spain david.pearce@upm.es

Abstract. RASP is a recent extension of Answer Set Programming that permits declarative specification and reasoning on consumption and production of resources. In this paper, we extend the concept of strong equivalence (which, as widely recognized, provides an important conceptual and practical tool for program simplification, transformation and optimization) from ASP to RASP programs and discuss its usefulness in this wider context.

Introduction

Answer Set Programming (ASP) is a paradigm of logic programming based on the answer set semantics [1], where solutions to a given problem are represented in terms of selected models (answer sets) of the corresponding logic program [2]. ASP is nowadays applied in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see, among many others, [3, 4]).

In recent work [5, 6, 7], an extension to ASP, called RASP (standing for ASP with Resources), has been proposed so as to support declarative reasoning on consumption and production of resources. RASP is a proper extension to ASP, in the sense that useful theoretical and practical results developed for ASP can be extended to RASP. In this paper in fact, we extend the concept of strong equivalence to RASP programs and discuss its usefulness.

Strong equivalence [8, 9], as widely recognized, provides an important conceptual and practical tool for program simplification, transformation and optimization. Even in the case where two theories are formulated in the same vocabulary, they may have the same answer sets yet behave very differently once they are embedded in some larger context. For a robust or modular notion of equivalence one should require that programs behave similarly when extended by any further programs. This leads to the concept of strong equivalence, where programs P_1 and P_2 are strongly equivalent if and only if for any S, $P_1 \cup S$ is equivalent to (has the same answer sets as) $P_2 \cup S$. It is easy to get convinced that, whenever P_1 and P_2 are different formulation of a RASP program involving consumption and production of resources, their behaving equivalently in different contexts is of particular importance related to reliability in resource usage. Moreover, a designer might be able to evaluate, in terms of strong equivalence, different though

analogous ways of producing certain resources, and thus be able to choose one rather than the other in terms of suitable criteria.

In order to extend the notion of strong equivalence to RASP, we need to reformulate the semantics of RASP programs so as to be similar to that of plain ASP programs. This is done in Sect. 2 after an introduction to RASP provided in Sect. 1. After that, the definition of strong equivalence developed in [9] can be fairly easily extended to RASP (Sect. 3). However, it turns out that RASP programs behave quite differently from ASP programs as far as strong equivalence is concerned, as interferences in resource usage among rules of P_1, P_2 and S can easily arise. Then, we will argue (Sect. 4) that a significant notion of strong equivalence for RASP programs requires to state some constraints on S. So done, strong equivalence becomes a more effective notion in the RASP context. Finally, in Sect. 5 we conclude and outline some possible future directions. Proofs of theorems can be found in Appendix A. We assume a reader to be familiar with both ASP and strong equivalence. The reader may refer for the former to [10] and to the references therein, and for the latter to [11].

1 Background on RASP

RASP [5, 6, 7] is an extension of the ASP framework obtained by explicitly introducing the notion of *resource*. It supports both formalization and quantitative reasoning on consumption and production of amounts of resources. These are modeled by *amount-atoms* of the form $q\!:\!a$, where q represents a specific type of resource and a denotes the corresponding amount. Resources can be produced or consumed (or declared available from the beginning).

The processes that transform some amounts of resources into other resources are specified by *r-rules*, for instance, as in this simple example:

$$computer\!:\!1 \leftarrow cpu\!:\!1, hd\!:\!2, motherboard\!:\!1, ram\_module\!:\!2.$$

where we model the fact that an instance of the resource *computer* can be obtained by "consuming" some other resources, in the indicated amounts.

In their most general form, r-rules may involve plain ASP literals together with amount-atoms. Semantics for RASP programs is given by combining stable model semantics with a notion of *allocation*. While stable models are used to deal with usual ASP literals, allocations are exploited to take care of amounts and resources. Intuitively, an allocation assigns to each amount-atom a (possibly null) quantity. Quantities are interpreted in an auxiliary algebraic structure that supports comparisons and operations on amounts. Thus, one has to choose a collection Q of *quantities*, the operations to combine and compare quantities, and a mapping that associates quantities to amount-symbols. Admissible allocations are those satisfying, for all resources, the requirement that one can consume only what has been produced. Alternative allocations might be possible. They correspond to different ways of using the same resources. A simple natural choice for Q is the set of integer numbers. In all the examples proposed in the rest of the paper, we implicitly make this choice.

Syntax and semantics of RASP are presented in [5]. Syntax is summarized below, semantics is reported shortly in Appendix B. Various extensions, presented in [5] and in [6, 7] (which discuss preferences and complex preferences in RASP) are not considered here. An implementation of RASP is discussed in [7] and is available at `http://www.dmi.unipg.it/~formis/raspberry/`.

RASP syntax is based upon partitioning the symbols of the underlying language into program symbols and resource symbols. Precisely, let $\langle \Pi, \mathcal{C}, \mathcal{V} \rangle$ be a structure where $\Pi = \Pi_{\mathcal{P}} \cup \Pi_{\mathcal{R}}$ is a set of predicate symbols such that $\Pi_{\mathcal{P}} \cap \Pi_{\mathcal{R}} = \emptyset$, $\mathcal{C} = \mathcal{C}_{\mathcal{P}} \cup \mathcal{C}_{\mathcal{R}}$ is a set of constant symbols such that $\mathcal{C}_{\mathcal{P}} \cap \mathcal{C}_{\mathcal{R}} = \emptyset$, and $\mathcal{V}$ is a set of variable symbols. The elements of $\mathcal{C}_{\mathcal{R}}$ are said *amount-symbols*, while the elements of $\Pi_{\mathcal{R}}$ are said *resource-predicates*. A *program-term* is either a variable or a constant symbol. An *amount-term* is either a variable or an amount-symbol.

The second step is that of introducing amount-atoms in addition to plain ASP atoms, called program-atoms. Let $\mathcal{A}(X, Y)$ denote the collection of all atoms of the form $p(t_1, \ldots, t_n)$, with $p \in X$ and $\{t_1, \ldots, t_n\} \subseteq Y$. Then, a *program-atom* is an element of $\mathcal{A}(\Pi_{\mathcal{P}}, \mathcal{C} \cup \mathcal{V})$. An *amount-atom* is an expression of the form $q{:}a$ where $q \in \Pi_{\mathcal{R}} \cup \mathcal{A}(\Pi_{\mathcal{R}}, \mathcal{C} \cup \mathcal{V})$ and a is an amount-term. Let $\tau_{\mathcal{R}} = \Pi_{\mathcal{R}} \cup \mathcal{A}(\Pi_{\mathcal{R}}, \mathcal{C})$. We call elements of τ_R *resource-symbols*.

Expressions such as $p(X){:}V$ where V, X are variable symbols are allowed, as resources can be either directly specified as constants or derived. Notice that the set of variables is not partitioned, as the same variable may occur both as a program term and as an amount-term. *Ground* amount- or program-atoms contain no variables. As usual, a *program-literal* L is a program-atom A or the negation *not* A of a program-atom (intended as negation-as-failure).[4] If $L = A$ (resp., $L = \mathit{not}\ A$) then $\overline{L}$ denotes *not* A (resp., A). A *resource-literal* is either a program-literal or an amount-atom. Notice that, for reasons discussed in [5], we do not admit negation of amount-atoms.

Finally, we distinguish between plain rules and rules that involve amount-atoms. In particular, a *program-rule* is defined as a plain ASP rule. Besides program-rules we introduce resource-rules which differ from program rules (which are usual ASP rules) in that they may contain amount-atoms. A *resource-proper-rule* has the form $H \leftarrow B_1, \ldots, B_k.$ where $B_1, \ldots, B_k$, are resource-literals and H is either a program-atom or a (non-empty) list of amount-atoms. If H is an amount-atom of the form $q{:}a$ where a is a constant and the body is empty then the rule is called a *resource-fact*. According to the definition then, the amount of an initially available resource has to be explicitly stated. As usual, we often denote the resource-fact $H \leftarrow$ simply by writing H.

In general, we admit several amount-atoms in the head of a rule where the case in which a rule γ has an empty head is admitted only if γ is a program-rule (i.e., γ is an ASP *constraint*). The list of amount-atoms composing the head of

[4] In this paper we will only deal with negation-as-failure. Nevertheless, classical negation of program literals could be used in RASP programs and treated as usually done in ASP.

an resource-rule has to be understood conjunctively, i.e., as a collection of those resources that are all produced at the same time by *firing*, i.e. applying, the rule.

A *resource-rule* (*r-rule*, for short) can be either a resource-proper-rule or a resource-fact. A RASP program may involve both program rules and resource-rules, i.e., a *RASP-rule* (*rule*, for short) γ is either a program-rule or a resource-rule and a RASP program (*r-program*) is a finite set of RASP-rules.

The ground version (or "grounding") of an r-program P is the set of all ground instances of rules of P, obtained through ground substitutions over the constants occurring in P. As customary, in what follows we will implicitly refer to the ground version of P.

Intuitively, an interpretation of P is an answer set whenever it satisfies all the program rules in P and all the fired r-rules (in the usual way) as concerns their program-literals, and all consumed amounts either were available from resource-facts or have been produced by rule firings.

Example 1. Below is an example of RASP program.

$g{:}2 \leftarrow q{:}4.$
$f{:}3 \leftarrow q{:}2.$
$h{:}1 \leftarrow q{:}1.$
$q{:}4.$

This program has the following answer sets. The answer set $\{q{:}4, g{:}2, -q{:}4\}$, where we employ (consume) resource $q{:}4$ (as indicated by the notation $-q{:}4$) to *fire* (i.e., apply) the first rule and produce $g{:}2$, with no remainder (the full available amount of q is consumed). The answer set $\{q{:}4, f{:}3, h{:}1, -q{:}3\}$, where we employ (consume) part of resource $q{:}4$ (as indicated by the notation $-q{:}3$) to fire the second and third rule and produce $f{:}3$ and $h{:}1$. In this case, as we do not have consumed the full amount of q, there is a remainder $q{:}1$ that might have been potentially used by other rules. We cannot produce $g{:}2$ together with $f{:}3$ and/or $h{:}1$ because the available quantity of q is not sufficient. But, we also have the answer sets $\{q{:}4\}$, $\{q{:}4, f{:}3, -q{:}2\}$ and $\{q{:}4, h{:}1, -q{:}1\}$ where all of part of resource $q{:}4$, though available, is left unconsumed.

Notice that the given program does not involve negation. In plain ASP we may have several answer sets only if the program involves negation, and, in particular, cycles on negation. In RASP, we may have several answer sets also in positive (i.e., definite) programs because of different possible allocations of available resources.

Notice also that in the present setting each rule can be applied only once, i.e, we can't for instance use the third rule several times to produce several items of $h{:}1$ according to the available qs. Actually, multiple "firings" of rules can be allowed by a suitable specification but we do not consider this extension here. However, a rule might in principle occur more than once in a program: in this case, each "copy" of the rule can be separately applied.

In [5] we introduce *politics* for resource usage, by extending the semantics so as to allow a programmer to state, for each rule, if the firing is either optional

or mandatory. In the rest of this paper, to simplify the discussion, we make the assumption that the firing of rules is mandatory. In the above example, this politics excludes the answer sets $\{q{:}4\}$, $\{q{:}4, f{:}3, -q{:}2\}$ and $\{q{:}4, h{:}1, -q{:}1\}$. In the next section, we introduce a version of the semantics of RASP close to the one originally defined for plain ASP, i.e., in terms of a reduct (for ASP, the so-called Gelfond-Lifschitz reduct, or GL-reduct, introduced in [1]).

2 Reduct-based RASP Semantics

In order to extend the notion of strong equivalence to RASP, it is useful to devise a semantic definition for RASP as close as possible to the standard answer set semantics as defined in [1]. This will make it easier to extend to RASP the definitions and proofs provided in [9] for plain ASP.

To our aim, we exploit the definition of RASP semantics discussed in Appendix B. It is based upon generating, from a ground r-program, a version where resource-predicates occurring in bodies of rules are associated to the rule where they occur. For the sake of simplicity and without loss of generality, we assume that a resource predicate, say q, may occur more than once in the body of the same rule only if the amounts are different. It may occur instead with either the same or different amounts in the head of several rules. Below, let a "program" be a RASP program. The answer sets for program obtained with the formulation proposed in this section are called *reduct-answer sets* (for short, *r-answer sets*).

In order to cope with different instances of the same amount-atom, say $q{:}a$, occurring in the body of different rules, at this stage we "standardize apart" these occurrences, according to the following definition.

Definition 1. *Let P be a ground r-program, and let $\gamma_1, ..., \gamma_k$ be the rules in P containing amount-atoms in their body. The standardized-apart version P_s of P is obtained from P by renaming each amount-atom $q{:}a$ in the body of γ_j, $j \leqslant k$ as $q^j{:}a$. The q^j's are called the standardized-apart versions of q, or in general standardized-apart resource-predicates.*

Example 2. Let P be the following program.

$g{:}2 \leftarrow q{:}4.$ $\quad$ $a \leftarrow not\ b.$
$p{:}3 \leftarrow q{:}3, a.$ $\quad$ $b \leftarrow not\ a.$
$d{:}1 \leftarrow q{:}3, b.$ $\quad$ $c.$
$q{:}6 \leftarrow c.$

The standardized-apart version P_s of P is as follows.

$g{:}2 \leftarrow q^1{:}4.$ $\quad$ $a \leftarrow not\ b.$
$p{:}3 \leftarrow q^2{:}3, a.$ $\quad$ $b \leftarrow not\ a.$
$d{:}1 \leftarrow q^3{:}3, b.$ $\quad$ $c.$
$q{:}6 \leftarrow c.$

We let $\mathcal{A}_{P_s}$ be the set of all atoms (both program-atoms and amount-atoms) that can be built from predicate and constant symbols occurring in P_s.

Definition 2. *A candidate reduct-interpretation $\mathcal{I}_{P_s}$ for P_s is any multiset obtained from a subset of $\mathcal{A}_{P_s}$.*

Referring to Example 2, among the possible candidate interpretations are, e.g., $I_1 = \{p{:}3, q{:}6, q^2{:}3, a, c\}$ and $I_2 = \{p{:}3, q{:}6, q^1{:}4, q^3{:}3, a, c\}$. Standardized-apart amount-atoms occurring in a candidate r-interpretation represent resources that are *consumed*. Plain amount-atoms represent resources that have been produced, or were available from the beginning. For a candidate r-interpretation to be an admissible r-interpretation (or, simply, r-interpretation), consumption has not to exceed production.

Definition 3. *Given multi-set of atoms S and resource-predicate q (possibly standardized-apart) occurring in S, let $f(S)(q)$ be an amount-symbol obtained by summing the quantities related to the occurrences of q. If S contains $q{:}a_1$, ..., $q{:}a_k$ (or, respectively, $q^j{:}a_1$, ..., $q^j{:}a_k$) and $a = a_1 + \ldots + a_k$ we will have $f(S)(q) = a$ (or, respectively, $f(S)(q^j) = a$).*

Definition 4. *A candidate reduct-interpretation $\mathcal{I}_{P_s}$ is a reduct-interpretation (for short r-interpretation) if for every resource-predicate q occurring in $\mathcal{I}_{P_s}$, taken all its standardized-apart versions $q^{j_1}, \ldots, q^{j_h}$, $h \geq 0$, also occurring in $\mathcal{I}_{P_s}$, we have $f(\mathcal{I}_{P_s})(q) \geq \sum_{i=1}^{h} f(\mathcal{I}_{P_s})(q^{j_i})$*

Referring again to Example 2, it is easy to see that I_1 is an r-interpretation, as 6 items of q are produced and just 3 are consumed, while I_2 is not, as 6 items of q are produced but 7 are supposed to be consumed.

We now establish whether an r-interpretation $\mathcal{I}_{P_s}$ is an r-answer set for P_s, and we will then reconstruct from it an r-answer set for P. To this aim, we introduce the following extension to the Gelfond-Lifschitz reduct.

Definition 5 (RASP-reduct). *Given a reduct-interpretation $\mathcal{I}_{P_s}$ for the standardized-apart version P_s of RASP program P, the RASP-reduct $cfp(P_s, \mathcal{I}_{P_s})$ is a RASP program obtained as follows.*

1. *For every standardized-apart amount-atom $A \in \mathcal{I}_{P_s}$, add A to P_s as a fact, obtaining $P_s^+(\mathcal{I}_{P_s})$;*
2. *Compute the GL-reduct of $P_s^+(\mathcal{I}_{P_s})$.*

Let $LM(T)$ be the Least Herbrand Model of theory T. In case T is a RASP program, in computing $LM(T)$ amount-atoms are treated as plain atoms. We may state the main definition:

Definition 6. *Given reduct-interpretation $\mathcal{I}_{P_s}$ for the standardized-apart version P_s of RASP program P, $\mathcal{I}_{P_s}$ is a reduct-answer set (r-answer set) of P_s if $\mathcal{I}_{P_s} = LM(cfp(P_s, \mathcal{I}_{P_s}))$*

Referring again to Example 2, the r-answer sets of P_s are:
$M_1 = \{q{:}6, g{:}2, q^1{:}4, a, c\}$, $M_2 = \{q{:}6, g{:}2, q^1{:}4, b, c\}$,
$M_3 = \{q{:}6, p{:}3, q^2{:}3, a, c\}$, $M_4 = \{q{:}6, d{:}1, q^3{:}3, b, c\}$.

Notice that M_1 and M_2 have the same resource consumption and production but from the even cycle on negation they choose a different alternative (a w.r.t.

b). Producing g:2 excludes being able to produce p:3 or d:1 respectively, because the remaining quantity of q is not sufficient. Instead of producing g:2, one can produce either p:3 (answer set M_3) if choosing the alternative a or d:1 (answer set M_4) if choosing the alternative b.

We now have to prove the equivalence of the above-proposed semantic formulation with the one of [5], summarized in Appendix B. In the Appendix, we report the notion of RASP answer set, and, in Definition 15, the equivalent notion of admissible answer set of an adapted RASP program. We will prove that there is a bijection between the set of the r-answer sets of the standardized-apart version P_s of RASP program P and the set of the admissible answer sets of the corresponding adapted program. It follows that there exists a bijection between the set of RASP answer sets of ground RASP program P and the set of r-answer sets of the standardized-apart version P_s of P.

In order to compare r-answer sets with admissible answer sets we have to make their form compatible. In particular, in Definition 14 of Appendix B we consider interpretations of an adapted program obtained from the standardized-apart program P_s augmented by adding, for each q^j:a, an even cycle involving q^j:a and the fresh atom no_q^j:a. These interpretations will thus contain either q^j:a or no_q^j:a to signify availability or, respectively, unavailability of this resource. Below we transform an r-interpretation into this form.

Definition 7. *Given an r-interpretation $\mathcal{I}_{P_s}$ for the standardized-apart version P_s of RASP program P, $ad(\mathcal{I}_{P_s})$ is obtained from $\mathcal{I}_{P_s}$ by adding no_q^j:a for every amount symbol q^j:a which occurs in P_s but not in $\mathcal{I}_{P_s}$.*

We can now state the result we where looking for:

Theorem 1. *Given an r-interpretation $\mathcal{I}_{P_s}$ for the standardized-apart version P_s of RASP program P, $\mathcal{I}_{P_s}$ is an r-answer set of P_s if and only if $ad(\mathcal{I}_{P_s})$ is an admissible answer set of the adapted program corresponding to P_s.*

Lemma 1. *There exists a bijection between the set of RASP answer sets of ground RASP program P and the set of r-answer sets of the standardized-apart version P_s of P.*

At this stage, as we wish to obtain r-answer sets of P, we can get a more compact form by getting the global consumed/produced quantity of each resource q.

Definition 8. *Given an r-answer set $\mathcal{I}_{P_s}$ of P_s, an r-answer set $\mathcal{M}_P^R$ of P is obtained as follows. For each resource-predicate q occurring in $\mathcal{I}_{P_s}$:*
(i) replace its occurrences q:b_1, ..., q:b_s, $s > 0$, with q:b, for $b = b_1 + \ldots + b_s$;
(ii) replace its standardized-apart occurrences q^{j_1}:a_1, ..., q^{j_k}:a_k, $k \geq 0$, with $-q$:a, for $a = a_1 + \ldots + a_k$.

By some abuse of notation, we will interchangeably mention r-answer set of P or of P_s. Referring to Example 2, the r-answer sets of P are:
$M_1 = \{q{:}6, g{:}2, -q{:}4, a, c\}$, $M_2 = \{q{:}6, g{:}2, -q{:}4, b, c\}$,
$M_3 = \{q{:}6, p{:}3, -q{:}3, c\}$, $M_4 = \{q{:}6, d{:}1, -q{:}3, b, c\}$.

Notice that the above notation does not explicitly report about what is left. Actually, given each r-answer set we can establish which resources we have "in our hands" after the production/consumption process by computing the difference, for each resource, between what has been produced and what has been consumed. For instance, in M_1 and M_2 we are left with q:2 and g:2, in M_3 with q:3 and p:3, and in M_4 with q:3 and d:1.

3 Strong Equivalence of RASP programs

In this section, we extend the standard notion of strong equivalence to RASP programs, taking as a basis the definitions that can be found in [9], which provides a characterization of strong equivalence of ground programs in terms of the propositional logic of here-and-there (HT-logic). We remind the reader that the logic of here-and-there is an intermediate logic between intuitionistic logic and classical logic. Like intuitionistic logic it can be semantically characterized by Kripke models, in particular using just two worlds, namely *here* and *there*, assuming that the *here* world is ordered before the *there* world. Accordingly, interpretations (HT-interpretations) are pairs (X, Y) of sets of atoms from given language L, such that $X \subseteq Y$. An HT-interpretation is total if $X = Y$. The intuition is that atoms in X (the *here* part) are considered to be true, atoms not in Y (the *there* part) are considered to be false, while the remaining atoms (from $Y \setminus X$) are undefined. A total HT-interpretation (Y, Y) is called an equilibrium model of a theory T, iff $(Y, Y) \models T$ and for all HT-interpretations (X, Y), such that $X \subset Y$, it holds that $(X, Y) \not\models T$. For an answer set program P, it turns out that an interpretation Y is an answer set of P iff (Y, Y) is an equilibrium model of P when reinterpreted as an HT-theory.

We take as a basis the standardized-apart version P_s of RASP program P. To account for resource production and consumption, HT-logic must be extended as follows. The set of atoms must be augmented to admit amount-atoms, involving both plain and standardized-apart resource predicates. Like in the RASP semantics, we take for given the choice of an algebraic structure to represent amounts and support operations on them.

The satisfaction relation of HT-logic between an interpretation $I = \langle I^H, I^T \rangle$ and a formula F must be augmented by adding the following two new axioms (where w is the world, that can be either *here* or *there*).

The first new axiom "distributes" the available quantity of a resource q to the formulas that need to use it. Notice that the total quantity of q can be produced in various items, but the condition is that the quantity which is distributed cannot exceed production.

AR-1
$\langle I^H, I^T, w \rangle \models q^{j_1}{:}a_1 \wedge \ldots \wedge q^{j_k}{:}a_k$ where $k > 0$ and each $j_i > 0$ if
$\langle I^H, I^T, w \rangle \models q{:}b_1 \wedge \ldots \wedge q{:}b_s$, $s > 0$, and $b_1 + \ldots + b_s \geq a_1 + \ldots + a_k$

Notice that, for every HT-interpretation $\langle I_s^H, I_s^T \rangle$ of P_s, AR-1 ensures that the sets I_s^H and I_s^T are, in the terminology of previous section (Definition 4), r-interpretations of P_s.

The second new axiom accounts for the fact that, given the final total quantity, the "fragments" in which a resource amount is produced do not matter. To this aim, for each resource-predicate q we provisionally introduce a fresh corresponding resource-predicate q^T.

AR-2
$\langle I^H, I^T, w\rangle \models q^T{:}b$ if $\langle I^H, I^T, w\rangle \models q{:}a_1 \wedge \ldots \wedge q{:}a_k$ and
$\langle I^H, I^T, w\rangle \not\models q{:}a$, $a \neq a_1 \wedge \ldots \wedge a \neq a_k$, where $k > 0$ and $b = a_1 + \ldots + a_k$

Notice that AR-2 is able to "compute" the total produced quantity b of each resource q. In order to deal with different though equivalent production patterns that may occur in different RASP programs, we keep in HT-interpretations only the total quantities of produced resources.

Definition 9. *Given an HT-interpretation $\langle I_s^H, I_s^T\rangle$ of P_s, its* normalized version *is obtained by replacing in both I_s^H and I_s^T for each resource-predicate q the set of atoms $q^T{:}b, q{:}a_1, \ldots, q{:}a_k$ with $q : b$.*

In what follows, by some abuse of notation, by "HT-interpretation" we mean its normalized version (the same for HT-models). This stated, it is not difficult to suitably extend to RASP the results introduced in [9]. We have the following lemma and we can state the main theorem:

Lemma 2. *For any RASP program P and any set I of atoms, the HT-interpretation $\langle I, I\rangle$ is an equilibrium model of P iff I is an r-answer set of P_s.*

Theorem 2. *For any RASP programs P_1 and P_2, and for every RASP program S, $P_1 \cup S$ has the same answer sets of $P_2 \cup S$, i.e., P_1 is strongly equivalent to P_2, if and only if their standardized-apart versions P_{1s} and P_{2s} are equivalent in the extended logic of Here-and-There, i.e., have the same HT-models.*

As we have seen before, axiom AR-2 and normalized versions of HT-interpretations account for different production patterns. Thus, two RASP programs are candidates to be equivalent whenever the same total quantity of each resource q is produced. Instead, consumption must be performed in exactly the same way, which, as we will see, is a quite strong limitation. In the rest of this section in fact, we discuss a number of examples to illustrate the meaning of strong equivalence for RASP programs, and emphasize the problems which arise. We will then address these problems in the subsequent section.

Example 3. The following two RASP programs are strongly equivalent.

P_1	P_2
$q{:}6 \leftarrow$ *not* c.	$q{:}3 \leftarrow$ *not* c.
	$q{:}3 \leftarrow$ *not* c.

Their unique equilibrium model is $\langle I, I\rangle$ with $I = \{q : 6\}$ which is the unique r-answer set of both programs. The only difference between the two programs is that the same amount of resource q is produced in P_1 all in once, and in P_2 in two parts. This difference is taken into account by axiom AR-2.

However, in the terms illustrated up to now, it is very difficult for RASP programs to be not only strongly equivalent, but even simply equivalent, as demonstrated by the following examples.

Example 4. The following two standardized-apart RASP programs are not even equivalent, despite the fact that they produce and consume the same resources, though in a different way.

P_1
$q{:}1 \leftarrow p^1{:}2.$ $\quad p{:}6.$
$r{:}1 \leftarrow p^2{:}3.$ $\quad g{:}1.$
Unique answer set: $\{g{:}1, p{:}6, q{:}1, r{:}1, p^1{:}2, p^2{:}3\}$.

P_2
$q{:}1 \leftarrow p^1{:}3.$ $\quad p{:}6.$
$r{:}1 \leftarrow p^2{:}2.$ $\quad g{:}1.$
Unique answer set: $\{g{:}1, p{:}6, q{:}1, r{:}1, p^1{:}3, p^2{:}2\}$.

By summing quantities like in Definition 8 in previous section, one would obtain the same unique r-answer set for both, namely $\{g{:}1, p{:}6, q{:}1, r{:}1, -p{:}5\}$. Even considering the r-answer set, the two programs are not strongly equivalent, as we can see if one adds, e.g., as third rule $g{:}2 \leftarrow p{:}3$ (that, standardized-apart, becomes $g{:}2 \leftarrow p^3{:}3$). The reason is that the additional rule "competes" with the original ones for the use of amounts of resource $p{:}6$. Different choices for employing resource $p{:}6$ become thus possible. In fact, for the augmented former program the answer sets would be:
$M_1 = \{g{:}1, p{:}6, q{:}1, r{:}1, p^1{:}2, p^2{:}3\}$ $M_2 = \{g{:}1, p{:}6, q{:}1, g{:}2, p^1{:}2, p^3{:}3\}$ and
$M_3 = \{g{:}1, p{:}6, r{:}1, g{:}2, p^2{:}3, p^3{:}3\}$
For the latter program they would be instead:
$N_1 = \{g{:}1, p{:}6, q{:}1, r{:}1, p^1{:}3, p^2{:}2\}$ $N_2 = \{g{:}1, p{:}6, q{:}1, g{:}2, p^1{:}3, p^3{:}3\}$ and
$N_3 = \{g{:}1, p{:}6, r{:}1, g{:}2, p^2{:}2, p^3{:}3\}$

Example 5. The following two standardized-apart RASP programs are, differently from what would happen in plain ASP, not even equivalent. In fact, in the former program there is a number of plainly available resources, while in the latter producing $q{:}4$ requires consuming $p{:}2$.

P_1
$q{:}4.$ $\quad p{:}2.$
$r{:}4.$ $\quad c.$
Unique answer set: $\{c, q{:}4, p{:}2, r{:}4\}$

P_2
$q{:}4 \leftarrow p^1{:}2.$ $\quad p{:}2.$
$r{:}4.$ $\quad c.$
Unique answer set: $\{c, q{:}4, p{:}2, r{:}4, p^1{:}2\}$

However, in both examples one may notice that the two programs are equivalent w.r.t. what is produced, and differ w.r.t. resources that are consumed. A conceptual tool to recognize some kind of equivalence of the above programs is

in order, as a designer would be enabled to assess their similarity and choose the way of production deemed more appropriate in the application at hand. In the next section we will propose a weaker notion of equivalence and strong equivalence than those introduced so far.

4 Strong Equivalence of RASP programs revisited

Below we introduce a compact form of HT-models where produced and consumed resources occur in their total quantities, similarly to r-answer sets introduced in Definition 8.

Definition 10. *Given an HT-model $\langle I_s^H, I_s^T\rangle$ of P_s, a* compact *HT-model (c-HT-model) $\langle I^H, I^T\rangle$ of P is obtained by replacing, for each resource-predicate q occurring in $\langle I_s^H, I_s^T\rangle$, its standardized-apart occurrences $q^{j_1}{:}a_1$, ..., $q^{j_k}{:}a_k$, $k \geq 0$, with $-q{:}a$, where $a = a_1 + \ldots + a_k$.*

The following definition make a further simplification by eliminating consumed quantities from r-answers sets and c-HT-models.

Definition 11. *Given an r-answer set A of P or given a c-HT-model $\langle I_c^H, I_c^T\rangle$ of P_s, a* production *answer set (p-answer set) A' of P (resp., a* production *HT-model, or p-HT-model) $\langle I^H, I^T\rangle$ of P_s) is obtained as by removing from A (resp., from I_c^H and I_c^T) all atoms of the form $-q{:}a$ for any q and a.*

In Example 4, the unique equilibrium c-HT-model of both programs is $\langle I, I\rangle$ with $I = \{g{:}1, p{:}6, q{:}1, r{:}1, -p{:}5\}$ where I is the unique r-answer set. The corresponding p-answer set is $I' = \{g{:}1, p{:}6, q{:}1, r{:}1\}$, and the unique equilibrium p-HT-model is $\langle I', I'\rangle$.

In Example 5, the unique r-answer set of P_1 is $M = \{c, q{:}4, p{:}2, r{:}4\}$, which coincides with the p-answer set. The unique equilibrium c- and p-HT-model is $\langle M, M\rangle$. For P_2, the unique r-answer set is $N = \{c, q{:}4, p{:}2, r{:}4, -p{:}2\}$, and the unique equilibrium c-HT-model is $\langle N, N\rangle$. The corresponding p-answer set is $N' = \{c, q{:}4, p{:}2, r{:}4\}$, and the unique equilibrium p-HT-model is $\langle N', N'\rangle$. Thus, in both Example 4 and Example 5 the two given programs are "equivalent"w.r.t. equilibrium p-HT-models (or, equivalently, p-answer sets). We formalize this notion of equivalence below.

Definition 12. *Two RASP theories (standardized-apart programs) are* equivalent on production *(p-equivalent) if their p-HT-models (and, consequently, their p-answer sets) coincide and are* strongly equivalent on production *(p-strongly equivalent) if, after adding whatever RASP theory S to both, they are still equivalent on production.*

This definition much enlarges the set of RASP programs that can be considered to be equivalent. However, strong equivalence remains problematic.

Example 6. Consider the two programs of Example 5, that are equivalent on production. Assume to add rule: $r{:}1 \leftarrow p^4{:}3$. The c-HT-models and the p-HT-models remain unchanged, and thus the two programs are still p-equivalent. In

fact the added rule cannot fire, not being available the needed quantity of p. Assume instead to add rule $r{:}1 \leftarrow p^4{:}2$, which is able to exploit the available resource amount $p{:}2$. In this case, for the augmented P_1 we get the unique r-answer set $\{c, q{:}4, p{:}2, r{:}1, -p{:}2\}$, but for the augmented P_2 the new rule "competes" with pre-existing ones on the use of $p{:}2$: thus, we get the two r-answer sets $\{c, p{:}2, r{:}4, r{:}1, -p{:}2\}$ and $\{c, q{:}4, p{:}2, r{:}4, -p{:}2\}$. Consequently, the updated programs are not p-equivalent.

We may notice that the problems that we have discussed arise when S is a *proper RASP program*, i.e., a RASP program containing amount-atoms. In order to make strong equivalence possible, we may introduce the constraint that whenever the addition S is a proper RASP program, it does not compete on resources with the original program. This appears quite reasonable: in fact, if one intends to enlarge a production/consumption process, preservation of existing processes should be guaranteed. If not, it should be clear that one obtains a different process, with hardly predictable properties.

Definition 13. *A* rational addition *S to a RASP program P is RASP program such that in program $P \cup S$ rules of S do not consume resources produced by rules of P.*

It is easy to get convinced that:

Proposition 1. *RASP proper program S is a rational addition if and only if one of the following conditions hold.*

1. *Resources consumed in S are not available in P.*
2. *Resources consumed in S are produced in S.*
3. *Resources consumed in S are produced in $P \cup S$.*
4. *S does not consume resources.*

Referring to Example 5: rule $r{:}1 \leftarrow p^4{:}3$ is a rational addition of kind (1); RASP proper program $r{:}1 \leftarrow p^4{:}3.\ p{:}5.$ is a rational addition of kind (2); RASP proper program $r{:}1 \leftarrow p^4{:}3.\ p{:}5 \leftarrow c.$ is a rational addition of kind (3); rule $r{:}1 \leftarrow c$ is a rational addition of kind (4).

If we restrict p-strong equivalence to p-strong equivalence w.r.t. rational addition, then the notion becomes much more usable in practical cases. In particular for instance, programs P_1 and P_2 of Examples 4 and 5 are p-strongly equivalent w.r.t. rational addition. Notice however that p-strong equivalence between two programs does not guarantee that we are left with the same resources in the two cases. A subject of future work will be that of studying other forms of strong equivalence, e.g. w.r.t. what is left or more particularly what is left for a certain set of resources.

5 Conclusions and Future Directions

In this paper, we have extended the notion of strong equivalence from answer set programs to RASP programs. We have seen that this notion takes a quite

peculiar flavor in RASP, where strong equivalence (apart from trivial cases) can be ensured at the condition of imposing some requirements on the theory which is added to given one. In fact, resource usages in the two components may interact in non-trivial ways. Nonetheless, strong equivalence may help a designer to reason about different though equivalent or partially equivalent processes.

As future direction, we may notice that, in the RASP context, an extension of strong equivalence, i.e., synonymy [12], can find an interesting application. Synonymy extends strong equivalence in the sense that two theories corresponding to different descriptions of a piece of knowledge may be equivalent even if they are expressed in different languages, if each is bijectively interpretable in the other answer set. Moreover, in a suitable sense, they can remain equivalent or synonymous when extended by the addition of new formulas.

In RASP, this may allow one to compare different processes formulated in seemingly different ways in order to establish whether they are "in essence" equivalent. Then, in practical applications one will be able to choose between the different formulations according to suitable parameters.

References

[1] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proc. of the 5th Intl. Conference and Symposium on Logic Programming, The MIT Press (1988) 1070–1080

[2] Marek, V.W., Truszczyński, M. In: Stable logic programming - an alternative logic programming paradigm. Springer (1999) 375–398

[3] Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)

[4] Anger, C., Schaub, T., Truszczyński, M.: ASPARAGUS – the Dagstuhl Initiative. ALP Newsletter **17**(3) (2004) See `http://asparagus.cs.uni-potsdam.de`.

[5] Costantini, S., Formisano, A.: Answer set programming with resources. Journal of Logic and Computation **20**(2) (2010) 533–571

[6] Costantini, S., Formisano, A.: Modeling preferences and conditional preferences on resource consumption and production in ASP. Journal of Algorithms in Cognition, Informatics and Logic **64**(1) (2009) 3–15

[7] Costantini, S., Formisano, A., Petturiti, D.: Extending and implementing RASP. Fundamenta Informaticae **105**(1-2) (2010) 1–33

[8] Pearce, D.: A new logical characterization of stable models and answer sets. In: Non-Monotonic Extensions of Logic Programming. Number 1216 in LNAI. Springer (1997) 55–70

[9] Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic **2** (2001) 526–541

[10] Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation. Chapter 7. Elsevier (2007)

[11] Pearce, D., Valverde, A.: Quantified equilibrium logic and the first order logic of here-and-there. Technical report, Univ. Rey Juan Carlos (2006) available at http://www.satd.uma.es/matap/investig/tr/ma06_02.pdf.

[12] Pearce, D., Valverde, A.: Synonymous theories in answer set programming and equilibrium logic. Proc. of ECAI04, 16th Europ. Conf. on Art. Intell. (2004) 388–390

A Proofs of Theorems

Proof of Theorem 1

Proof. If part.
It is easy to see that the r-answer sets obtained by means of the RASP-reduct can equivalently be obtained by applying the GL-reduct to a version of P_s where, instead of asserting the $q^j{:}a$'s as facts, one would have added to P_s for each of them an even cycle as specified in Definition 14, thus obtaining the adapted program corresponding to P_s. In fact, one would get exactly $ad(\mathcal{I}_{P_s})$ that, by Definition 4, obeys the condition of Definition 15.
Only-if part.
If one removes the $no\_q^j{:}a$'s from $ad(\mathcal{I}_{P_s})$ one obtains a candidate r-interpretation $\mathcal{I}_{P_s}$ for P_s that, being an admissible answer set for the adapted program corresponding to P_s, is an r-answer set for P_s because: (i) admissibility of $ad(\mathcal{I}_{P_s})$ guarantees that the condition specified in Definition 4 is fulfilled, and thus the candidate r-interpretation is actually an r-interpretation; (ii) except for the additional even cycles, rules of P_s and of the corresponding adapted program are the same; (iii) the RASP-reduct exploits the $q^j{:}a$'s present in an r-interpretation as though they had been generated by even cycles;

Proof of Theorem 2 (Main Theorem)

The proof is sketched, as it can be obtained as a variation of the proof provided in [9] for plain ASP program. Therefore, we will not repeat all the steps of their proof (that takes many pages), rather we emphasize the variations. Below, by "HT-logic" we take the extended HT-logic defined in Sect. 3.

First of all it is useful to extend Lemma 3 of [9] to RASP programs. To this aim, we need:
Fact 1: Given RASP program P, for every HT-model $\langle H, T\rangle$ of its standardized-apart version P_s both H and T are r-interpretations for P.

In fact, axiom AR-1 of extended HT-logic ensures that from a resource q available in a certain total quantity ("computed" by axiom AR-2 of the logic) one can obtain standardized-apart portions of that resource in an amount not exceeding the available one, as required by Def. 4 of Sect. 2. This allows us to restate Lemma 3 of [9] as Lemma 2, proved below. So done, we have all the elements to conclude by applying the proof of [9] to P_{1s} and P_{2s}.

Proof of Lemma 2

Proof. To prove this lemma we simply have to notice that, from Definitions 5 and 6, an r-interpretation I of P is r-answer set I of P_s iff it is an answer set of $P_s^+(I)$, obtained by adding to P_s the standardized-apart atoms occurring in I as facts. Then, we can apply the proof of [9] to $P_s^+(I)$ given that axiom AR-2 accounts for different production patterns.

B RASP Semantics

In this section we summarize the semantics of RASP. In [5], there are two formulations of the RASP semantics, the the first one more "abstract" and the second one more "practical", i.e., closer to standard ASP. In this context we focus on the latter, that was originally provided in order to evaluate RASP complexity (which b.t.w. is the same as plain ASP) and has proved also useful for defining strong equivalence.

In the more abstract semantic definition, an interpretation/answer set of a RASP program is defined as a couple $\langle I, \mu \rangle$ where I takes care of program atoms, and μ is a function which assigns quantities to all occurrences of resource symbols, where the balance must be positive, in the sense that only what has been produced (or was available from the beginning) can be consumed. In the more practical semantic formulation, resource-predicates occurring in the body of rules are standardized-apart, by substituting them by means of fresh resource-predicates. This kind of elaboration is formalized in Definition 1, seen earlier.

By considering ground amount-atoms as plain atoms, we can now add, for each standardized-apart version of an amount-atom, an even cycle which simulates this resource to be allocated to the r-rule where it occurs or not (we can add such an even cycle only if that resource can potentially be produced by some other r-rule).

Definition 14. *Let P be a (ground) r-program, and let P_s be the standardized-apart version of P. The* adapted program *P' for P is obtained by adding to P_s for each $q^j{:}a$ occurring in the body of some r-rule of P_s and such that $q{:}b$ (for some b) occurs in the head of some r-rule of P_s the following pair of rules (where $no\_q^j{:}a$ is a fresh atom):*

$$q^j{:}a \leftarrow not\ no\_q^j{:}a.$$
$$not\ no\_q^j{:}a \leftarrow not\ q^j{:}a.$$

We can thus obtain the answer sets of the adapted program P', that we call "classical answer sets" of P'.

Below we state that a classical answer set M of the adapted program is *admissible* if the resources that have been used does not exceed the resources that were available. Notice that for each resource-predicate q, the available quantity t_a is obtained by summing all amounts of atoms of the form $q{:}a$ (that in the adapted program are found in the head of rules), while the total consumed quantity is obtained by summing all amounts of their standardized-apart versions (that in the adapted program are found in the body of rules).

Definition 15. *Let P be a (ground) r-program and P' the corresponding adapted program P'. A classical answer set M of P' is* admissible *if the* resource balance relative to M *is non-negative, i.e.:*

$$\forall q \in \tau_{\mathcal{R}} \left(\sum \{\!\{ a \mid q{:}a \in M \}\!\} \right) - \left(\sum \{\!\{ a \mid q^j{:}a \in M \ for\ some\ j \}\!\} \right) \geqslant 0$$

In [5] we have formally stated the relationship between RASP answer sets of program P and admissible answer sets of the adapted program P'.

Informally, given an admissible answer set M of the adapted program P', we have to identify in M the two components. Let $\mathcal{P}(M)$ be obtained from M as the subset including the program-atoms only. The particular quantities-assignment function $\mu = \nu(M)$ is elicited from M by collecting the quantities associated to amount-atoms occurring in rules which are fired in M, i.e., such that the head is in M and the body literals are satisfied in M.

It is thus possible to state the equivalence of the two semantic formulations.

Theorem 3. *$\langle I, \mu \rangle$ is a RASP answer set of a ground r-program P iff $I = \mathcal{P}(M)$ and $\mu = \nu(M)$ where M is an admissible answer set of the adapted program P' for P.*

Instantiation to Support the Integration of Logical and Probabilistic Knowledge

Jingsong Wang and Marco Valtorta

Department of Computer Science and Engineering
University of South Carolina, Columbia SC 29208, USA
{wang82, mgv}@cse.sc.edu

Abstract. Integrating the expressive power of first-order logic with the ability of probabilistic reasoning of Bayesian networks has attracted the interest of many researchers for decades. We present an approach to integration that translates logical knowledge into Bayesian networks and uses Bayesian network composition to build a uniform representation that supports both logical and probabilistic reasoning. In particular, we propose a new way of translation of logical knowledge, relation search, that is easy to understand, simple to implement, and efficient to execute. Grounding is required to generate a Bayesian network in the first-order case. In order to limit the size of the generated Bayesian networks, our prototype restricts the knowledge base to be function-free.

Keywords: Bayesian networks, First-order logic, Hybrid logical-probabilistic systems, Implicative normal form

1 Introduction

1.1 Automated Reasoning and the Integration Problem

Knowledge-based systems normally comprise two components: a knowledge base (KB) and an inference engine. The former encodes what we know about the world, while the latter acts on the knowledge base to answer queries [11, 2]. Traditionally, the knowledge bases consist of a set of logical rules, and the reasoning engine is based on logical deduction. Because of the unavoidable need of probabilistic reasoning, Bayesian networks (BNs) and the laws of probability theory play important roles in building knowledge-based systems.

However, a system based purely on logic or Bayesian networks is not practical for many advanced applications. Classical first-order logic (FOL) has great expressive power but cannot handle uncertainty. Bayesian networks can represent probabilistic information well but scale poorly and require users to have specialized expertise for effective modeling. This situation led many researchers to work on the integration problems of these two types of inference and various approaches have been proposed.

For the relation between logical reasoning and probabilistic reasoning, in case of propositional logic, logical reasoning is just one special case of probabilistic

reasoning. For FOL, whose reasoning is mostly based on instantiation and deduction (although there are also lifted approaches), it could be translated into propositional logic through instantiations (Herbrand Expansion), and therefore, in some sense, FOL could be considered a special case of probabilistic reasoning.

1.2 Using Logical Knowledge in Probabilistic Reasoning

Probabilistic models have been widely used in AI. Because of various data sources and ways of modeling, conflicts might appear in probabilistic reasoning of large systems, especially when comparing its result with commonsense conclusion from traditional logical reasoning. Purely probabilistic knowledge and modeling may not be accurate or strong enough, and we believe that the pre-existing logical knowledge could be a very important complement that should never be ignored in probabilistic reasoning. We can use a simple example to show the influence. Figure 1 shows a Bayesian network with two nodes A and B[1]. This Bayesian network represents our knowledge of probabilistic dependency of proposition A and proposition B. $\mathbf{Pr}(A)$ and $\mathbf{Pr}(B|A)$ quantify these dependencies. Then an abductive query for A's probability of being true given B is true is to calculate $\mathbf{Pr}(A|B)$, which has a unique value. In Figure 1, $\mathbf{Pr}(A = true|B = false) = 0.27$. However, suppose we also have some logical knowledge about the relation between A and B in KB, such as $A \rightarrow B$. Then there might be a disagreement with the posterior probability of A, as the logical knowledge requires us to conclude that A should be false, with 100% certainty. Which value should we choose? In some cases, we might like to accept the result from logic reasoning, as traditionally we have more experience with logical knowledge and we believe that logic reasoning is more mature in AI, while probabilistic knowledge is normally hard to quantify and therefore error-prone. This kind of case is possible especially in many design areas. The example shows that sometimes our knowledge might be inconsistent when modeling a problem from the logical view and the probabilistic view individually. Or we can say, the previous probabilistic model has not combined all our knowledge of A and B, and reasoning based on such a Bayesian network is insufficient.

If we use the logical relation to supplement the probabilistic model in a way as shown in Figure 2, i.e., we add an extra node representing the logical relationship, for the same evidence plus the additional logical knowledge, we get a different result of the posterior probability of A from the Bayesian network.

Things get more complex when we can have the uncertainty over logical rules from KB. However, the complexity also highlights the necessity of putting logical knowledge into consideration of probabilistic reasoning.

The previous example is very simple, as there are only two nodes in the original Bayesian network that are directly connected. For nodes with such direct connections, people should have better knowledge about their relations and the

[1] The probabilities shown in Figure 1 are computed using the commercial Bayesian network shell Hugin (www.hugin.com), which uses the junction tree algorithm for probabilistic inference [6].

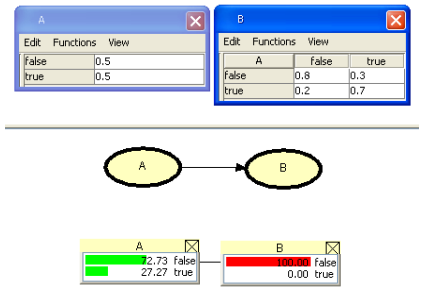


Fig. 1. A Bayesian network with its CPTs and reasoning result given evidence.

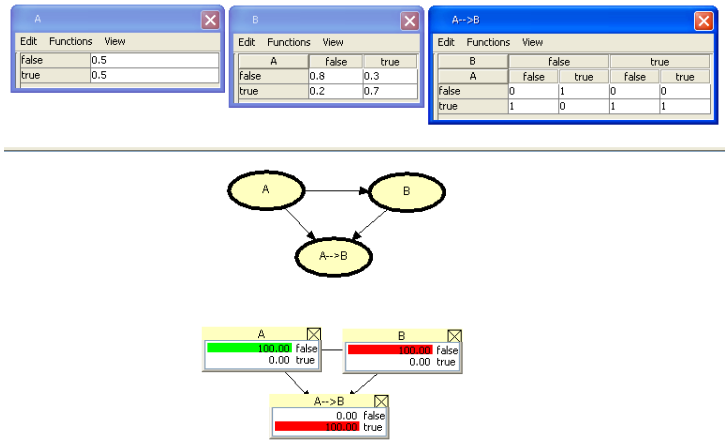


Fig. 2. Based on the Bayesian network in Figure 1, an extra node representing logical relations is added.

probabilistic dependency should be modeled consistently with the existing logical knowledge. However, we can think of a more complex example. We still have the same logic knowledge about A and B in the KB: $A \rightarrow B$. The original Bayesian network still contains node A and B, but there are also other nodes between them. Figure 3 shows a Bayesian network based on three nodes, one of which is the extra node C. Suppose now we are more interested in the posterior probability of C being true, given evidence B being true, which is shown to be is 0.14 in Figure 3. Now we integrate the logical relationship $A \rightarrow B$ into the new Bayesian network the same way as before, as shown in Figure 4. The posterior probability of C being true is 0.27, which is apparently increased, compared with the previous 0.14. Particularly, this shows the probabilistic influence of the logical relation of A and B to the proposition C even if there is no logical knowledge of C involved in the KB.

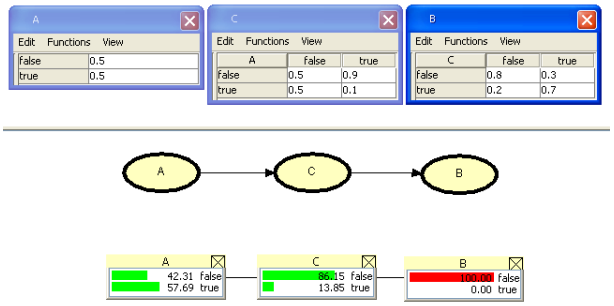


Fig. 3. A Bayesian network with its CPTs and reasoning result given evidence.

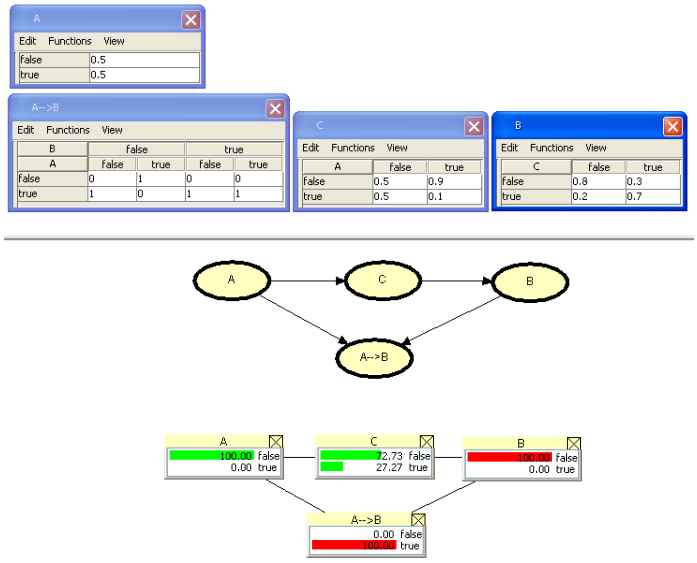


Fig. 4. An extra logical node is added to the Bayesian network in Figure 3.

This way of combining logical knowledge into probabilistic reasoning could be more meaningful when the original Bayesian networks are more complex. For example, just assume that nodes A and B in Figure 3 are now connected through ten other nodes. Note that the connection is not necessarily a line, but a network. In this case, because the distance from A to B is too long, people cannot ensure that, when they are modeling the probabilistic dependencies using Bayesian networks, they have put into enough consideration logical knowledge, which could have been gained from an existing KB. Under this situation, a simple

logical partial model, such as the ones in Figure 2 and Figure 4, can achieve this easily.

A special case is that when we have no knowledge over the logical rules, i.e., no evidence over logical rules, the composite model just resumes to the basic probabilistic model. This is because of the way the logical model is generated: it is a two level Bayesian network (in propositional case) where d-separations will appear when no evidence over logical rule nodes, which always serve as children of atomic formula nodes.

1.3 Our Framework for Integration

We pursue an approach that allows modelers to 1) specify special knowledge using the most suitable languages, while reasoning in an uniform engine; 2) make use of pre-existing logical knowledge bases for probabilistic reasoning (to complete the model or minimize potential inconsistencies).

In our framework, the user is assumed to use FOL formulas to specify logical knowledge stored in a KB, and a Bayesian network, which we call *Probabilistic Bayesian Network (PBN)*, to specify probabilistic knowledge. The basic idea is to convert related knowledge from the logical KB into a logical model represented through a Bayesian network, which we call *Logical Bayesian Network (LBN)*. Then the LBN is composed with the PBN. The final output of the composition is a Bayesian network, *Composite Bayesian Network (CBN)*, that integrates both the logical knowledge and the probabilistic knowledge related to the query. Figure 5 summarizes the process. Note that we still follow the knowledge-based approach for reasoning, but our knowledge base is neither only the traditional logical knowledge base nor simply the Bayesian networks, but their combination. However reasoning is still based on the laws of probability theory.

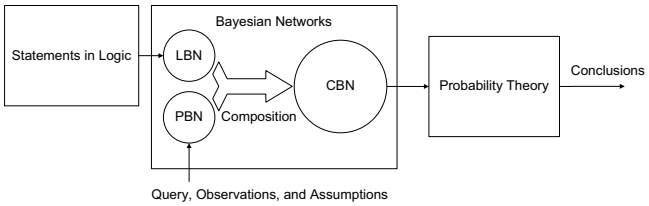


Fig. 5. Reasoning systems with the combination of logic and Bayesian networks.

The main focus of this paper is the way the logical model is generated. More details about the composition process and a comparison with related work can be found in [12]. Here we improve the work of [12] by removing the need for a theorem prover. The remainder of this paper is organized as follows. In section 2, we introduce the relation search algorithm in the propositional case at first

and then we extend it into the FOL case. A proof of correctness follows and an instantiation problem is discussed. Then we conclude this paper in section 3.

2 Logical Bayesian Network Generation

2.1 Propositional Case Algorithm

The end purpose is to build a composite model from logical knowledge and probabilistic knowledge. The PBN contains most of probabilistic knowledge, the nodes of which consist of the query, observations, and assumptions. We call them *probabilistic atoms*. When there is no direct probabilistic dependency specified among probabilistic atoms, the Bayesian network contains a set of isolated nodes.

The logical knowledge is assumed to be from a logical KB comprising a set of logical formulas. We will translate the KB directly into a LBN through a search algorithm, called *relation search*. The basic idea of relation search is very simple. We just look through the KB and extract all the logical formulas that are related to the probabilistic atoms. A formula has a *relation* with an atom if that atom appears directly or indirectly in this formula. The indirect appearance is defined the following way. If atom A_1 and A_2 both appear in a formula R_1, and A_1 also appears in a formula R_2 while A_2 does not, we say that A_2 appears indirectly in the formula R_2. Algorithm 1 depicts the pseudocode for our relation search algorithm.

Note that in Algorithm 1, we only explicitly generate the CPTs for the formulas from KB. These CPTs are determined by the formulas' logical structure. For example, for formula $A \vee B$, the value true has probability 1 if and only if either A or B is true. $BN(\mathcal{V}, \mathcal{E}, \mathcal{L})$ denotes the Bayesian network built based on $(\mathcal{V}, \mathcal{E}, \mathcal{L})$, where $\mathcal{V}$ represents the set of nodes (variables), $\mathcal{E}$ represents the set of directed edges that connect nodes together, and $\mathcal{L}$ represents the set of conditional probability tables.

2.2 FOL Case Modification of Algorithm

In the FOL case, instantiation and quantified formula node generation are needed to build the LBN. Even in the FOL case, queries, observations and assumptions should mostly be ground formulas. Thus we only consider ground atomic formulas from PBN and the set of such atoms is denoted by $\mathcal{P}$. We assume that all the formulas in KB are closed (i.e., no occurrence of free variables) and in Skolem form, which allows only universal quantifiers. Also, we assume that constants in KB are all the individuals in the domain, and we restrict the domain to be function-free.

1. We do the same search and find all the related formulas $\mathcal{R}$ from KB.
2. We use the available constants in $\mathcal{R} \cup \mathcal{P}$ to get all the possible ground instantiations of quantified formulas in $\mathcal{R}$, and add these instantiations to $\mathcal{R}$. This new set is named $\mathcal{R}'$.

Algorithm 1 Relation Search

Require: the probabilistic atom set $\mathcal{P}$ from PBN containing query, observations, and assumptions, the set of logical formulas KB $= \{R_1, R_2, ..., R_z\}$, $z = |\text{KB}|$, and the sets $\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_z$. $\mathcal{A}_i$ comprises all the atoms appearing in its corresponding logical formula $R_i \in$ KB, where $1 \leq i \leq z$.

$\mathcal{V} = \mathcal{P}$;
$\mathcal{E} = \emptyset$;
$\mathcal{L} = \emptyset$;
for $i = 1$ to z **do**
  Tag R_i as not visited;
end for
while true do
  $Changed \Leftarrow$ **false**;
  for $i = 1$ to z **do**
    if R_i is not visited **then**
      if $\mathcal{A}_i \cap \mathcal{V} \neq \emptyset$ **then**
        $\mathcal{V} = \mathcal{V} \cup \mathcal{A}_i \cup \{R_i\}$;
        for all $A \in \mathcal{A}_i$ **do**
          $\mathcal{E} = \mathcal{E} \cup \{(A, R_i)\}$;
        end for
        Build CPT Θ_i for R_i based on its logical structure;
        $\mathcal{L} = \mathcal{L} \cup \{\Theta_i\}$;
        Tag R_i as visited;
        $Changed \Leftarrow$ **true**;
      end if
    end if
  end for
  if $Changed =$ **false then**
    break;
  end if
end while
return $BN(\mathcal{V}, \mathcal{E}, \mathcal{L})$;

3. We build a Bayesian network based on $\mathcal{R}'$. We follow exactly the same procedure as in the propositional case for generating nodes, edges, and CPTs for the propositional formulas in $\mathcal{R}'$. For a quantified formula, we put it as the child of its ground instantiations (groundings) plus an extra node O, which represents a proposition that all the other instantiations that are based on constants appearing in KB but not in $\mathcal{R}'$ hold. The CPT for such a quantified formula is an AND table, i.e., the value true has probability 1 if all parents are true and probability 0 otherwise.

Thus for the FOL case, the generated Bayesian network will usually be a three level network if quantified formulas exist in $\mathcal{R}'$ [2]. The nodes corresponding

[2] One extreme case is that the quantified formula itself is an atomic formula. The parents of the corresponding node are root nodes in **G** directly. However, such formulas do not make sense in a normal KB. We exclude this case.

to them are in the third level. One important change for the FOL case relation search output is an O node for each quantified formula as one additional parent, whose value reflects the actual influence of some constants, which seems to be unrelated, through the related formulas.

Notice that any instantiation of the root nodes (including O nodes) of the resulting Bayesian network has defined one truth value for any formula appearing in the Bayesian network, as the Bayesian network has explicitly or implicitly contained all the constants from KB and we have assumed that the constants available in KB have been all the constants in the domain. For a ground atomic formula (including O node), which appears as the root node, its truth value is directly decided by its state value in the instantiation. For a ground compound formula, its truth value is decided by its logical evaluation based on its parents' truth values. For a quantified formula, its truth value is decided by the logical AND evaluation based on its parents' truth values.

We can use one example to illustrate how instantiation works. Suppose we have a logical KB containing formulas as shown in Table 1. $\mathcal{P} = \{P(a), P(b)\}$ are from PBN [3]. Then after the relation search,

$$\mathcal{R} = \{\forall x P(x) \rightarrow Q(x)\}.$$

Here constants are a, b, and c. We note that the second and the third formula in KB are ignored as they cannot be related. The output Bayesian network $\mathbf{G}$ is shown in Figure 6.

Table 1. The original FOL KB's formulas.

1 $\forall x P(x) \rightarrow Q(x)$,
2 $\forall x H(x) \rightarrow L(x)$,
3 $H(c)$.

2.3 Correctness

We want to show that models constructed in relation search behave according to our logical intuitions.

Theorem 1. *For a BN resulting from relation search,* $\mathbf{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$*, for any* $U \in \mathcal{V}$ *and any* $\mathcal{V}' \subseteq \mathcal{V}$*, if* $\mathcal{V}' \models U$*, then* $\mathbf{Pr}(U = true | \mathcal{V}' = true) = 1$ *in* $\mathbf{G}$*, where* $\mathcal{V}' = true$ *means that all the nodes in* $\mathcal{V}'$ *of* $\mathbf{G}$ *are set to true.*

Proof of Theorem 1 The idea is to use weighted model counting [2], over the Bayesian network $\mathbf{G}$, to conclude the joint probability of evidence $\mathbf{e} = \{U = false\} \cup \mathcal{V}' = true$ to be 0, and then follow Bayes' rule to conclude the posterior

[3] Note that the PBN can be the result of any modeling technique, such as MEBN [9, 10] or BLP [7], that outputs a Bayesian network.

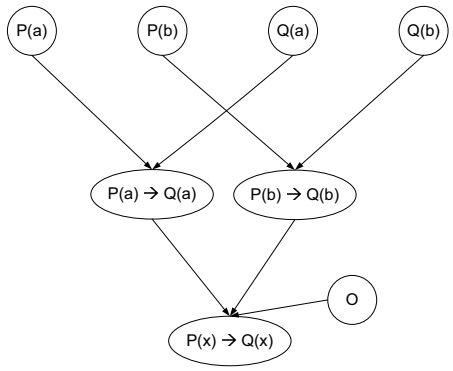


Fig. 6. A FOL case LBN example.

probability $\mathbf{Pr}(U = true|\mathcal{V}' = true) = 1$. We can assume $\mathcal{V}' = \{S_1, S_2, ..., S_k\}$, where $k \leq |\mathcal{V}|$. Then evidence can be represented by $\mathbf{e} = \{\overline{u}, s_1, s_2, ..., s_k\}$.

If $\mathcal{V}' \models U$, then FOL theory $\{\neg U\} \cup \mathcal{V}' = \{\neg U, S_1, S_2, ..., S_k\}$ is logically unsatisfiable. We define $F = U \wedge S_1 \wedge S_2 \wedge ... \wedge S_k$ and $F' = \neg U \wedge S_1 \wedge S_2 \wedge ... \wedge S_k$. Then a Herbrand structure of F is also the one of F' and vice versa. Note that every subformula of F has a corresponding node in $\mathbf{G}$. We know that F' is unsatisfiable. This means that all interpretations (more commonly called structures in FOL) of F', including its Herbrand structures, that evaluate formula U to false will also evaluate at least one formula S_x $(1 \leq x \leq k)$ to false.[4]

For BN $\mathbf{G}$, each instantiation includes one instantiation of $\mathbf{G}$'s root nodes, which are all ground atomic formulas because of the way $\mathbf{G}$ is built. (The root nodes also include the O nodes, which are also atomic propositions.) Such an instantiation of root nodes in $\mathbf{G}$ has defined a Herbrand structure for the language of $\mathbf{G}$. Thus given an instantiation $\mathbf{G}$, there is a corresponding evaluation for any formula appearing in $\mathbf{G}$ and this evaluation result is fixed. For example, the ground logical constituents of nodes $\{U, S_1, S_2, ..., S_k\} \subseteq \mathcal{V}$ are root nodes in $\mathbf{G}$. Each instantiation of the root nodes of $\mathbf{G}$ contains one Herbrand structure for formula $F = U \wedge S_1 \wedge S_2 \wedge ... \wedge S_k$ and its subformulas.

In addition, for each quantified formula appearing in $\mathbf{G}$, all its possible groundings based on constants available in the relation search result $(\mathcal{R} \cup \mathcal{P})$ also appear in $\mathbf{G}$.

Consider the special CNF encoding introduced in Section 11.6.1 [2] of BN $\mathbf{G}$. Any instantiation of $\mathbf{G}$ corresponds to a model of such a CNF encoding. To compute $\mathbf{Pr(e)}$ through the weighted model count, we only consider instantiations of $\mathbf{G}$ that are compatible with evidence $\mathbf{e} = \{\overline{u}, s_1, s_2, ..., s_k\}$.

[4] For a closed formula in Skolem form, it is satisfiable if and only if it has a Herbrand model. In other words, unsatisfiability ≡ no Herbrand model.

We choose one arbitrary instantiation of $\mathbf{G}$ compatible with $\mathbf{e}$. The Herbrand structure contained in the current instantiation is denoted by H. We know that formula U can evaluate to true or false under H. We consider these two cases separately.

1. U evaluates to false in H.
   Thus U is false in H. Because of the unsatisfiability of F', we know that F' does not have a Herbrand model. Therefore if U evaluates to false in H, there must be at least a S_x that evaluates to false in H. As we know that the nodes of $\mathbf{G}$ are in three levels, node S_x might appear in the first level (being an atomic formula node), in the second level (being a ground formula), or in the third level (being a quantified formula). Note that atomic formulas, appearing as root nodes in $\mathbf{G}$, definitely evaluate to the same values as their state values in the instantiation. Remember that all the nodes $S_1, S_2, ..., S_k$ are in state true, as the current instantiation is compatible with evidence $\mathbf{e}$. Since S_x is false in H, S_x could not be in the first level. This also excludes the case that S_x is an O node. If S_x is in the second level, then there is an inconsistency between its logical evaluation, which is false based on its parents' state values, and its state value, which is true as determined by $\mathbf{e}$. If node S_x is in the third level, i.e., it is a quantified formula, there are two cases for possible instantiations of its parent nodes, which are groundings of S_x, in $\mathbf{G}$:
   (a) All the parent nodes of S_x are in state value true.
   Consider the Herbrand structure H for F contained in the current instantiation. Based on the definition of the truth value of quantified formulas with universal quantifier, because H evaluates S_x (which has universal quantifier) to false, there must be at least one grounding of S_x, denoted by S_{xg}, that evaluates to false in H. Based on the way $\mathbf{G}$ is built, S_{xg} is a parent node of S_x and it is in the second level of $\mathbf{G}$. (Note that the O node associated with S_x is excluded from being S_{xg}, because it is a root node, which should have the same evaluation in H as its state value, and a parent of S_x, and we have assumed that S_x's parent nodes are all in state value true) However, we have assumed that S_x's parent nodes are all in state value true. Thus for S_{xg}, there is a similar inconsistency as before between its logical evaluation, which is false based on its parents' state values, and its state value, which is true.
   (b) There is at least one parent node of S_x in state false, including its O node.
   The CPT for node S_x, which represents a formula with universal quantifier, is an AND table. There is still a similar inconsistency as discussed before between S_x's logical evaluation, which is false based on its parents' state values, and its real state value, which is true in order to be compatible with $\mathbf{e}$.
   Therefore, if node S_x is in the third level of $\mathbf{G}$, for any possible instantiation compatible with $\mathbf{e}$, we can always find a node with logical inconsistency. This node might be S_x itself, or one of its parents S_{xg}.

2. U evaluates to true in H.
   There are three cases for the appearance of U: U is an O node, a ground formula that is not an O node, and a quantified formula.
   (a) U is an O node. This case is impossible.
   Being a root node, an O node should have the same evaluation as its state value false, which is compatible with evidence $\mathbf{e}$.
   (b) U is a ground formula but not an O node.
   Still, U cannot be a root node. Thus U is a second level grounding. U has a state value false, which is compatible with evidence $\mathbf{e}$, and however it evaluates to true. Thus U itself has an inconsistency.
   (c) U is a quantified formula. We have a similar analysis as before.
   If all the parent nodes of U are in state value true, then there is an inconsistency in U, as we know that the CPT of U is an AND table, and however its state value is false to be compatible with $\mathbf{e}$.
   Otherwise, there is at least one parent in state value false. Based on the definition of the truth value of a quantified formula, all the parent formulas (node O or not) of U in $\mathbf{G}$ need to evaluate to true in H. Being a root node, the O node cannot be in state value false, as its state value should be the same value as its evaluation in H. Therefore there must be a grounding in state value false, which however evaluates to true. Then this grounding has an inconsistency.

Because of the way CPTs are built in $\mathbf{G}$, as shown in the propositional case analysis, the entries of a CPT of a non-root node in $\mathbf{G}$ all have value 0, if the logical evaluation of its logical formula, given its parents' state values, is inconsistent with the node's real state value. For a node with such inconsistency, denoted by S, if we use c to represent the state values of its parent nodes in the current instantiation, and the corresponding truth assignment of CNF encoding is denoted by ω, we know that

$$Wt(P_{s|c}) = \theta_{s|c} = 0,$$

and the weight of the model ω is

$$Wt(\omega) = Wt(P_{s|c}) * \prod_{i=1}^{t-1} Wt(P_i) = 0,$$

where t is the number of nodes in $\mathbf{G}$, i.e., $|\mathbf{G}| = t$, and P_i represents the other parameter literal of the model ω that is not $P_{s|c}$.

This conclusion holds for all the other models. Thus

$$\mathbf{Pr}(\mathbf{e}) = \mathbf{Pr}(\overline{u}, \mathcal{V}' = true) = \mathbf{Pr}(\overline{u}, s_1, s_2, ..., s_k) = \sum_{i=1}^{2^{t-k-1}} Wt(\omega_i) = 0,$$

where ω_i represents the model corresponding to the different instantiation of $\mathbf{G}$ compatible with evidence $\{\overline{u}, s_1, s_2, ..., s_k\}$. Thus, as claimed in the statement of the theorem,

$$\mathbf{Pr}(u|\mathcal{V}' = true) = \frac{\mathbf{Pr}(u, \mathcal{V}' = true)}{\mathbf{Pr}(u, \mathcal{V}' = true) + \mathbf{Pr}(\overline{u}, \mathcal{V}' = true)} = \frac{\mathbf{Pr}(u, \mathcal{V}' = true)}{\mathbf{Pr}(u, \mathcal{V}' = true) + 0} = 1.$$

2.4 A Partial Instantiation

In the FOL case, as the number of atoms containing different variables in a quantified formula increases, the number of groundings could increase exponentially. This might make the BN **G** very large. However, many nodes, including atoms and groundings, in this BN are not meaningful in real use.

In a well-defined KB, besides formulas representing rules, there are some facts that represent basic and fixed features of a few context constants in KB. For atoms describing such features, any groundings of them that are not explicitly specified in KB are regarded impossible. This is often referred as closed world assumption, i.e., for a feature F of a sequence of constants $\mathcal{C}$ in the domain, if $F(\mathcal{C})$ is not listed as a fact, then we believe $F(\mathcal{C})$ is false. We call the predicates like F *context predicates* and the set of related facts *context facts*.[5] For example, consider two atoms $Person(a)$ and $Table(b)$ as existing facts in KB. For the constants specified by these two atoms, their features are not exchangeable, as we know that $Person(b)$ and $Table(a)$ are meaningless. There should not be uncertainty over them any time as they are always false. Therefore, we can control the number of groundings of quantified formulas by discarding meaningless instantiations. In addition, we assume that all the rules in KB are in Implicative Normal Form (INF). Note that any formula can be easily translated into INF. For example, $(A_1 \wedge A_2) \vee (B_1 \wedge B_2) \rightarrow C$, where A_1, A_2, B_1, and B_2 are literals and C is a subformula, can be converted into $A_1 \wedge A_2 \rightarrow C$ and $B_1 \wedge B_2 \rightarrow C$. Then we add one extra step for the instantiation process of quantified formulas, which are in INF. We will still try all the possible instantiations of a quantified formula that contains context predicates. We discard a grounding unless all of the context predicates appearing in its body are context facts. The idea is simple. Each rule in KB naturally adds one constraint to the set of possible worlds. However, when one part of the body of the rule, which is in INF, is deterministically false, the rule will not constrain anything, based on the definition of logical implication. Then adding the rule is truly meaningless. Thus we can just ignore such groundings when building the BN. In applications, the users can define their context predicates and context facts flexibly for controlling the size of resulting BN.

3 Conclusion

In this paper, we presented a new way of translating logical knowledge into Bayesian networks that supports a new approach to the integration problem of logical and probabilistic reasoning that is easy to understand, simple to implement, and efficient to execute.

The method presented in this paper can be used not only for the general integration problem but also the traditional BN learning problem. There have been

[5] *Context predicates* and *context facts* play similar roles in instantiation as *random variable declarations* in LBN [4] and *context terms* in MEBN [9, 10]; these are all similar to typed predicates and terms.

classical ways to learn BNs, which include parameter estimation and structure learning, such as the EM algorithm [3] for parameter estimation with incomplete data and score-based learning for BN structure [1]. Meanwhile, Inductive logic programming (ILP), which is concerned with relational data mining, has been greatly researched. The problem of learning logic programs is the first and still most commonly addressed problem in ILP [5]. Because our approach provides a way of translating logic programs into Bayesian networks, we can easily extend most existing ILP techniques to work on BN learning.

We also notice that one quantitive label of a formula could be produced from the logic program's learning process. This label represents, e.g., the formula's rate of successful matching in the training data. Thus it is ideal to be used as soft evidence [8] in the composite model BN resulting from the integration process for further probabilistic reasoning. How to use such soft evidence for the improvement of accurate reasoning for many problems, such as classification, is an interesting topic for future work .

References

1. Cooper, G.F., Herskovits, E.: A Bayesian method for the induction of probabilistic networks from data. Machine Learning 9, 309–347 (1992)
2. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)
3. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the em algorithm. Journal of the Royal Statistical Society, Series B 39(1), 1–38 (1977)
4. Fierens, D., Blockeel, H., Bruynooghe, M., Ramon, J.: Logical bayesian networks and their relation to other probabilistic logical models. In: Proceedings of the 15th International Conference on Inductive Logic Programming. pp. 121–135. Springer (2005)
5. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning). The MIT Press (2007)
6. Jensen, F.V., Nielsen, T.D.: Bayesian Networks and Decision Graphs, second edition. Springer, New York, NY (2007)
7. Kersting, K., Raedt, L.D.: Bayesian logic programs. CoRR cs.AI/0111058 (2001)
8. Kim, Y.G., Valtorta, M., Vomlel, J.: A prototypical system for soft evidential update. Applied Intelligence 21(1) (July–August 2004)
9. Laskey, K.B.: First-order Bayesian logic, Technical Report C4I06-01. Tech. rep., SEOR Department, George Mason University (February 2006)
10. Laskey, K.B.: MEBN: A language for first-order knowledge bases. Artificial Intelligence 172, 140–178 (2008)
11. McCarthy, J.: Programs with common sense. In: Semantic Information Processing. pp. 403–418. MIT Press (1959)
12. Wang, J., Byrnes, J., Valtorta, M., Huhns, M.: On the combination of logical and probabilistic models for information analysis. Applied Intelligence pp. 1–26 (January 2011)

The DLV parallel grounder

Simona Perri[1], Francesco Ricca[1], and Marco Sirianni[1]

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{perri,ricca,sirianni}@mat.unical.it

1 Introduction

Answer Set Programming (ASP) [1, 2], a purely declarative programming paradigm based on nonmonotonic reasoning and logic programming. The high expressive power of ASP has been profitably exploited for developing advanced applications belonging to several fields, from Artificial Intelligence [3–7] to Information Integration [8], and Knowledge Management [9]. Interestingly, these applications of ASP recently have stimulated some interest also in industry [10].

The idea of answer set programming is to represent a given computational problem by a logic program the answer sets of which correspond to solutions, and then, use an answer set solver to find such solutions [2]. The main construct of the language of ASP is the logic rule.[1] ASP rules allow (in general) both disjunction in the head and nonmonotonic negation in the body. Importantly, ASP is declarative, and the ASP encoding of a variety of problems is very concise, simple, and elegant [3, 9, 13, 7].

As an example, consider the well-known 3-colorability problem. Given a graph $G = (V, E)$, assign each node one of three colors (say, red, green, or blue) such that adjacent nodes always have distinct colors. First of all, we represent G by introducing suitable facts: $vertex(v).\ \forall v \in V$, and $edge(v_1, v_2).\ \forall (v_1, v_2) \in E$; then, an ASP program solving the 3-colorability problem is the following:

$$(r_1) \quad col(X, red) \vee col(X, green) \vee col(X, blue) \text{ :- } vertex(X).$$
$$(r_2) \quad \text{ :- } edge(X, Y), col(X, C), col(Y, C).$$

The " :- " symbol can be read as "if", thus rule r_1 expresses that each node must either be colored red, green, or blue; due to minimality of answer sets, a node cannot be assigned more than one color. Rule r_2 acts as an integrity constraint and disallows situations in which two vertices connected by an edge are associated with the same color. Intuitively, an empty head is false, thus rule r_2 has the effect of discarding models in which the conjunction is true.

The computation of an ASP program is a two step process; the first step of evaluation is called instantiation and amounts to computing a program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$, but not containing any variable; after, $\mathcal{P}'$ is evaluated by using propositional backtracking search techniques.

The instantiation of ASP program is a crucial task for efficiency, and is particularly relevant when huge input data has to be dealt with. In this scenario, significant perfor-

[1] For introductory material on ASP, we refer to [2, 11, 9, 12].

mance improvements can be obtained by exploiting modern multi-core/multi-processor SMP machines, featuring several CPU in the same case.

In this paper, we report on the implementation of DLV$^{par}_{Gr}$ a parallel instantiator based on the state of the art ASP system DLV [14]. The resulting instantiator features three levels of parallelism [15, 16], as well as load-balancing and granularity control heuristics, which allows for effectively exploiting the processing power offered by modern multi-core/multi-processor SMP machines. The result of an experimental analysis, which was carried out on publicly-available benchmarks already exploited for evaluating the performance of instantiation systems, are also reported.

2 Parallelization Techniques and Implementation Issues

In this Section we briefly present the main new features of DLV$^{par}_{Gr}$ and, then, we enlighten some pragmatically-relevant issues regarding its implementation.

System Features. The system enjoys a three level parallel instantiation technique, enhanced with granularity control and load balancing heuristics. More in detail, the first level of parallelism allows for instantiating in parallel independent subprograms of the program in input $\mathcal{P}$. The division of $\mathcal{P}$ is performed accordingly to the *Dependency Graph*, a graph that, intuitively, describes the dependencies among the predicates of $\mathcal{P}$. Clearly, this level of parallelism is useful when the input program contains parts that are independent.

The second level of parallelism allows for evaluating in parallel rules within a given subprogram. Among these rules, we distinguish two different types, recursive and non recursive ones (also called exit rules). First, all exit rules are concurrently instantiated; then, all recursive rules are processed in parallel performing several iterations according to a seminaive schema[17]. The second level of parallelism is particulary useful when the number of rules in the subprograms is large.

The third level, allows for the parallel evaluation of a single rule. In particular it allows for subdividing the extension of a body predicate and evaluating the "split" parts in parallel. This level is crucial for the parallelization of programs with few rules, where the first two levels are almost not applicable.

Clearly, it can happen that a rule is particulary easy to be instantiated; in this case, applying the third level of parallelism, can be useless, or in the worst case, it can even slow down the instantiation of the rule. To this aim, a *granularity control* heuristics has been defined, which is based on the dynamic evaluation of the "weight" of a rule *r*; if the heuristic value is less than a given threshold, the level of parallelism is not applied. The "weight" is evaluated combining two estimations: (i) the size of the join of the body predicates of *r* (which should give an idea of the number of ground rules produced); and, (ii) an estimation of the number of comparison done for instantiating *r*. The same heuristic values are exploited for defining a *load balancing* strategy which allows for keeping the size of each single split of the body predicate large enough, in order to not introduce delays, but also sufficiently small so that it is unlikely that one split requires an instantiation time significantly larger than the others. If a rule passes the granularity control, load balancing is applied: according to the computed heuristic values, the number of splits is determined for the given rule; if this number is larger than

a given threshold, a finer redistribution of the workload is performed in the very last part of the computation. A more detailed description of these techniques and heuristics can be found in [15, 16].

Implementation issues. The parallel instantiator DLV_{Gr}^{par} enhances the instantiator module of DLV [14] by introducing the three levels of parallelism and the heuristics described in the previous section. The system architecture is based on the repeated appliction of the traditional producer and consumers paradigm. Each level of parallelism receives tasks from the previous level (but the first one which receives its tasks directly from the main application process), and dispenses tasks to the subsequent one. In multi-threaded applications, the minimization of both mutual exclusive parts of code and lock contentions delays is fundamental for efficiency; also the number of threads used by the application should carefully managed, to both limit the costs of data structure initialization and avoid starvation problems. Our system makes use of a fixed number of threads. In particular, a certain number of threads is spawned at the beginning for each level of parallelism. Master threads (at each level) push tasks in the buffer of consumers. The dimension of the consumer's task buffer is important. Indeed, if it is not sufficiently large, there could be delays due to lock contention between consumer and producers, and also delays due to the fact that a producer has to search for a consumer buffer which is still not full. Clearly if the task buffer is too large, there will be a waste of computational resources. The optimum value for this buffer size depends on the machine at hand and, thus, it can be provided as an input parameter.

As the instantiation process is carried out in parallel, there will be the production of ground rules simultaneously; this may introduce lock contention on the output stream. To this aim we introduced output buffers (one for each thread) to store ground rules; when a given number of ground rules has been produced the output buffers are flushed. Also in this case, the output buffers size can be set by the user as an input parameter to be adapted for optimizing performance on the machine at hand.

3 System usage and options

In this Section we describe the usage of the grounder DLV_{Gr}^{par} and the available options. DLV_{Gr}^{par} can be invoked as follows:

```
./DLV^par_Gr  [-THC= <>] [-THR= <>]  [-THS= <>]
 [-FlushFactor= <>] [-TaskBufferSize= <>]
 [-SequentialJoinThreshold= <>] [-SequentialMatchThreshold= <>]
 [-redistributionThreshold= <>] [filename [filename [...]]]
```

Beside the standard DLV options, DLV_{Gr}^{par} introduces the ones that are relevant system parameters for optimizing the parallel instantiation process on the machine ad hand. In more details:

(i) "-THC","-THR", and "-THS" indicate the number of threads spawned for component parallelism, rule parallelism, and single rule parallelism, respectively. A rule of thumb for establishing these values is to keep the first two small and the third

significantly larger, but still at a reasonable rate (as too many threads may significantly slow down the computation). Default values are 8, 8, 256 suitable for a machine equipped with 8 cores.
(ii) "-FlushFactor" indicates to overll number of ground rules that can be stored in output buffers before printing them on standard output. The optimal value mostly depends on the number of processors available, default value is 100.
(iii) "-TaskBufferSize" is the maximum number of tasks that may be scheduled for the *component*, *rule*, and *single rule* level of parallelism at the same time. Default value is 8.
(iv) "-SequentialJoinThreshold" and "-SequentialMatchThreshold" are meant to tune the granularity control heuristics; the two values specify the thresholds used in the granularity control heuristic. Default values are respectively 100 and 10000.
(v) "-redistributionThreshold" is meant to tune the redistribution heuristics; the value specifies the number of tasks of single rule parallelism that may be scheduled for instantiating a rule, before applying the redistribution heuristics. Default value is 8.

System Availability. The system (32bit ELF executable) can be downloaded at `http://www.mat.unical.it/ricca/downloads/parallelground10.zip`.

4 Experimental analysis

We carried out an experimental analysis to assess the performance of the instantiator. We considered some well-known combinatorial problems which have been already used for assessing ASP instantiator performance [14, 3, 7], namely *N-Queens*, *Ramsey Number*, *Golomb Ruler*, *Max Clique* and *3Colorability*.[2] Note that these benchmarks are particulary difficult to parallelize because of the compactness of their encodings (a common property of ASP programs due to the declarative nature of the language). About intances, we considered five instance of increasing difficulty for each of the first 5 problem, whereas for 3Colorability we generated graphs having from 80 to 260 nodes. The machine used for the experiments is a two-processor Intel Xeon "Woodcrest" (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0. In order to obtain more trustworthy results, each single experiment has been repeated five times and the average of performance measures are reported. In order to study the performance of the system when the number of available processor increases, the system was run on the selected benchmarks with 1,2,3,4,5,6,7, and 8 CPU enabled.[3]

The results of the analysis are summarized in Fig. 1. In particular, the table in Fig. 1a reports the instantiation times and standard deviation elapsed, as the number of enabled CPUs grows from 1 to 8, for solving the problem instances of all the considered benchmarks; whereas the average efficiency[4] for the same problems is reported in Fig. 1b.

[2] Encodings and instances can be downloaded at `http://www.mat.unical.it/ricca/downloads/parallelground10.zip`.

[3] The CPU n was disabled/enabled by running the linux command $echo\, 0/1 >> /sys/devices/system/cpu/cpu-n/online$.

[4] Efficiency is given by the run time of the sequential algorithm divided by the runtime of the parallel one times the number of processors.

	Average instantiation time (standard deviation)							
Problem	serial	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queens_1$	4.64 (0.00)	2.53 (0.01)	1.71 (0.01)	1.31 (0.01)	1.07 (0.00)	0.91 (0.01)	0.78 (0.01)	0.69 (0.01)
$queens_2$	5.65 (0.00)	3.11 (0.00)	2.11 (0.01)	1.60 (0.00)	1.31 (0.01)	1.11 (0.01)	0.97 (0.00)	0.86 (0.02)
$queens_3$	6.83 (0.00)	3.79 (0.01)	2.57 (0.01)	1.97 (0.01)	1.60 (0.01)	1.35 (0.02)	1.17 (0.01)	1.03 (0.02)
$queens_4$	8.19 (0.00)	4.54 (0.00)	3.06 (0.01)	2.35 (0.01)	1.90 (0.01)	1.62 (0.02)	1.41 (0.01)	1.22 (0.00)
$queens_5$	9.96 (0.00)	5.57 (0.19)	3.68 (0.01)	2.81 (0.02)	2.26 (0.02)	1.92 (0.00)	1.69 (0.01)	1.43 (0.01)
$ramsey_1$	258.52 (0.00)	131.72 (0.08)	89.04 (0.41)	67.10 (0.46)	55.14 (0.93)	46.62 (0.19)	39.98 (0.12)	36.23 (0.34)
$ramsey_2$	328.68 (0.00)	167.47 (0.16)	112.97 (0.94)	85.90 (0.15)	70.64 (1.74)	58.70 (0.82)	51.21 (0.18)	46.09 (0.33)
$ramsey_3$	414.88 (0.00)	210.98 (0.38)	142.85 (0.68)	108.00 (0.38)	88.13 (0.51)	74.83 (0.22)	65.25 (0.59)	58.06 (0.20)
$ramsey_4$	518.28 (0.00)	264.69 (1.82)	178.67 (2.39)	137.42 (1.89)	111.09 (2.15)	95.27 (2.02)	81.45 (0.45)	75.19 (2.41)
$ramsey_5$	643.65 (0.00)	327.06 (0.36)	222.81 (0.20)	169.37 (0.86)	135.94 (0.17)	115.78 (0.92)	101.21 (1.33)	92.28 (0.65)
$clique_1$	16.06 (0.00)	8.51 (0.13)	5.84 (0.08)	4.45 (0.17)	3.64 (0.04)	3.08(0.1)	2.67 (0.03)	2.35 (0.01)
$clique_2$	29.98 (0.00)	15.92 (0.23)	10.69 (0.18)	8.27 (0.09)	6.77 (0.11)	5.71 (0.10)	4.94 (0.40)	4.34 (0.07)
$clique_3$	49.11 (0.00)	25.81 (0.41)	17.31 (0.06)	13.39 (0.20)	10.92 (0.20)	9.23 (0.02)	7.98 (0.03)	7.09 (0.02)
$clique_4$	78.05 (0.00)	41.68 (0.07)	27.91 (0.28)	21.10 (0.02)	17.33 (0.20)	14.60 (0.04)	12.76 (0.06)	11.29 (0.11)
$clique_5$	119.48 (0.00)	62.87 (0.13)	42.62 (0.15)	32.46 (0.04)	26.14 (0.21)	22.24 (0.00)	19.14 (0.00)	17.09 (0.16)
$golomb\_ruler_1$	6.58 (0.00)	3.34 (0.01)	2.26 (0.00)	1.73 (0.02)	1.42 (0.02)	1.24 (0.02)	1.06 (0.03)	0.94 (0.02)
$golomb\_ruler_2$	13.74 (0.00)	6.63 (0.02)	4.60 (0.18)	3.41 (0.04)	2.86 (0.10)	2.43 (0.04)	2.11 (0.02)	1.84 (0.09)
$golomb\_ruler_3$	24.13 (0.00)	12.11 (0.02)	8.15 (0.06)	6.34 (0.06)	5.06 (0.10)	4.34 (0.17)	3.79 (0.05)	3.25 (0.13)
$golomb\_ruler_4$	40.64 (0.00)	20.27 (0.05)	13.51 (0.11)	10.35 (0.10)	8.64 (0.19)	7.13 (0.25)	6.35 (0.31)	5.51 (0.10)
$golomb\_ruler_4$	62.23 (0.00)	31.54 (0.29)	21.30 (0.16)	16.03 (0.09)	12.95 (0.20)	11.03 (0.27)	9.67 (0.15)	8.36 (0.17)
$3 - col_1$	8.61 (0.00)	4.41 (0.02)	3.02 (0.02)	2.29 (0.03)	1.86 (0.03)	1.59 (0.02)	1.38 (0.03)	1.26 (0.03)
$3 - col_2$	30.92 (0.00)	15.74 (0.27)	10.78 (0.19)	7.97 (0.02)	6.35 (0.02)	5.37 (0.03)	4.77 (0.20)	4.28 (0.30)
$3 - col_3$	78.64 (0.00)	40.25 (0.19)	26.66 (0.42)	20.31 (0.19)	16.24 (0.34)	13.44 (0.13)	11.56 (0.10)	10.36 (0.15)
$3 - col_4$	177.28 (0.00)	89.29 (0.65)	60.65 (0.58)	44.91 (0.24)	36.34 (0.26)	30.54 (0.51)	26.35 (0.06)	22.70 (0.37)
$3 - col_5$	347.97 (0.00)	178.98 (4.07)	123.22 (2.62)	90.09 (1.79)	71.84 (1.48)	61.42 (2.13)	53.32 (0.52)	45.88 (1.12)

(a) Average instantiation times in seconds (standard deviation)

	Efficiency						
Problem	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queens$	0.98	0.97	0.96	0.94	0.92	0.92	0.91
$ramsey$	0.98	0.97	0.95	0.94	0.92	0.91	0.88
$clique$	0.94	0.93	0.91	0.90	0.89	0.87	0.86
$golomb\_ruler$	1.00	0.99	0.97	0.95	0.93	0.91	0.92
$3 - col$	0.98	0.96	0.97	0.96	0.95	0.94	0.93

(b) Mean efficiencies

Fig. 1: Impact of parallelization techniques

Looking at both the tables it is possible to see that the performance is nearly optimal, it slightly decreases as the number of processors increases, and rapidly increases when the difficulty of the input instance grows. The granularity control heuristics has been able to catch easy rules and perform their instantiation sequentially. The load balancing mechanism resulted to be effective and produced a well-balanced workload between CPUs.

Summarizing, the parallel instantiator behaved very well in all the considered instances, indeed its efficiency rapidly reaches good levels and remains stable when the sizes of the input problem grow. Importantly, the system offers a very good performance already when only two CPUs are enabled (i.e. for the largest majority of the commercially-available hardware at the time of this writing) and efficiency remains at a very good level when up to 8 CPUs are available.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
2. Lifschitz, V.: Answer Set Planning. In: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
3. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR'07. LNCS 4483, (2007) 3–17
4. Balduccini, M., Gelfond, M., Watson, R., Nogeira, M.: The USA-Advisor: A Case Study in Answer Set Planning. In: LPNMR 2001 . Volume 2173., (2001) 439–442
5. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Logic-Based Artificial Intelligence. Kluwer (2000) 257–279
6. Baral, C., Uyan, C.: Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In: LPNMR 2001. Volume 2173., (2001) 186–199
7. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, 14-18, 2009. Proceedings. LNCS 5753, (2009) 637–654
8. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005, Baltimore, Maryland, USA, ACM Press (2005) 915–917
9. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
10. Grasso, G., Leone, N., Manna, M., Ricca, F.: Asp at work: Spin-off and applications of the DLV system. In: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of M. Gelfond. (2011) 1–20
11. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm – A 25-Year Perspective. (1999) 375–398
12. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . AI **138**(1–2) (2002) 3–38
13. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In: Logic-Based Artificial Intelligence. Kluwer (2000) 79–103
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
15. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. Journal of Algorithms in Cognition, Informatics and Logics **63**(1–3) (2008) 34–54
16. Perri, S., Ricca, F., Sirianni, M.: A parallel asp instantiator based on DLV. In: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming. DAMP '10, New York, USA, ACM (2010) 73–82
17. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press (1988)