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 $T \cup \bar{T}$ is satisfiable in DL where $\bar{T}$ 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:

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

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

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

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

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

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

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

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

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

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

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

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.