Extensions of Answer Set Programming

V. W. Marek
Department of Computer Science
University of Kentucky
Lexington, KY 40506

J.B. Remmel
Departments of Computer Science and Mathematics
University of California
La Jolla, CA 92093

Abstract

We discuss a number of possible extensions of Answer Set Programming. The four formalisms we investigate are:

1. logic programs where the negative parts of the bodies in clauses can be replaced by arbitrary constraints which we call Arbitrary Constraint Logic Programming (ACLP),
2. logic programs where we are allowed arbitrary set constraint atoms,
3. logic programs where atoms represent sets from some fixed set X and the one-step provability operator is composed with a monotone idempotent operator on 2^X which we call Set-based Logic Programming (SBLP), and
4. logic programming where the clauses (rules) have embedded algorithms which we call Hybrid Answer Set Programming (H-ASP).

1 Introduction

Past research has demonstrated that logic programming with the answer-set semantics, known as *answer-set programming* or *ASP*, for short, is an expressive knowledge-representation formalism [MT99, Nie99, Lif99, GL02, Bar03, MR04]. The availability of the non-classical negation operator *not* allows the user to model incomplete information, frame axioms, and default assumptions such as normality assumptions and the closed-world assumption (CWA). Modeling these concepts in classical propositional logic is less direct [GL02] and typically requires much larger representations. In addition, current implementations of ASP support aggregate operations over finite sets or, more generally, constraints over finite sets.

A fundamental methodological principle behind ASP, which was identified in [MT99, Nie99], is that to model a problem, one designs a program so that its answer sets *encode* or *represent* problem solutions. This is in contrast with the traditional way automated reasoning is used in knowledge representation, which relies on proof-theoretic methods of resolution with unification. Niemelä [Nie99] has argued that logic programming with the stable-model semantics should be thought of as a language for representing constraint satisfaction problems. Thought of from this point of view, ASP systems are ideal logic-based systems to reason about a variety of types of data and integrate quantitative and qualitative reasoning. ASP systems allow the users to describe solutions by giving a series of constraints and letting an ASP solver search for solutions. In [MR03] it is shown that such systems can, in principle, solve any NP-search problem [MR03], i.e. any FNP problem as described in [BG94].

In this paper, we shall consider several ways to extend answer set programming. One of our main motivations for considering extension of answer set programming is that current *solvers* such as *cmodels* [BL02], *smodels* [SNS02], *assat* [LZ02], *clasp* [GKNS07], *dlv* [LPF06], *pbmodels* [LT05b] and *aspps* [ET06, EIMT06] have no systematic way to reason about infinite sets. Of course, there is one obvious way that one can use to reason about infinite sets in logic programming, namely, we can add function symbols to the language. However, adding function symbols to the language has significant drawbacks, especially with regard to complexity. For example, finding the least model of a finite Horn program with no function symbols can be done in linear time [DG84] while the least model of a finite predicate logic Horn program with function symbols can be an arbitrary recursively enumerable set [Smu68]. If we consider logic programs with negation, Marek and Truszczyński [MT89] showed that the question of whether a finite propositional logic program has a stable model is NP-complete. However Marek, Nerode, and Remmel [MNR92] showed that the question of whether a finite predicate logic program with function symbols possesses a stable model is Σ_1^1 complete. Similarly, the stable models of logic programs that contain function symbols can be quite complex. Starting with [AB90] and continuing with [BMS95] and [MNR92], a number of results showed that the stable models of logic programs that allow function symbols can be exceedingly complex, even in the case where the program has a unique stable model. For example, Marek, Nerode and Remmel [MNR92] showed that there exist finite predicate logic programs which have stable models but which have no hyperarithmetic stable models. Thus there is no hope to have general processing engines that will handle normal logic programs with function symbols.

These complexity results for logic programs with function symbols may seem quite negative, but they had a positive effect in the long run in that they forced researchers and designers to limit themselves to cases where programs can be actually processed. The effect was that processing programs called *solvers* such as *cmodels* [BL02], *smodels* [SNS02], *assat* [LZ02], *lasp* [GKNS07], and *dlv* [LPF06], *pbmodels* [LT05b] and *aspps* [ET06, EIMT06], had to focus on finite programs that do not admit function symbols. The *dlv* system does allow for some limited use of functions symbols, with the idea which is common in Computer Science that it is programmer's responsibility to write programs that the system can process. But at this point, none of the existing solvers have good ways to deal with infinite sets.

Why do we need to reason about infinite sets? Clearly, if one wants to reason about regions in Euclidean space or time intervals,then one is implicitly reasoning about infinite sets although one often can get by with finite descriptions or approximations. However, we believe that another source of need for reasoning about infinite sets comes from the many interactions with the Internet that are required for modern applications which gives rise to the need to reason about sets and databases which are extremely large and/or are constantly changing and evolving. We claim that infinite sets offer an effective way to address problems involving large finite sets that do not have a clear structure and may change rapidly. Such finite sets often do not have concise representations, and manipulating them based on their explicit enumerations is impractical. On the other hand, infinite approximations to these large finite sets, if chosen appropriately, may have structure that makes concise finite representations possible and possibly allow for effective reasoning and processing. For instance, a large database of documents and the set of WWW pages are examples of very large sets, interesting subsets of which can be thought of conceptually as infinite sets, e.g., "all documents containing a given string". Often such sets can be described as regular languages and hence have a finite description. In addition, a set of localities that might be affected by a tornado or the scope of a battlefield provide examples of finite sets that change rapidly. Thus it may be more convenient to find approximations such as polygons covering the affected areas that lend themselves to easy manipulation. In each case, while the finite sets of interest may have no small representations, the infinite sets used as approximations do - a feature that can be exploited in automated reasoning.

For yet another example, consider the problem of controlling an unmanned underwater vehicle V. Given parameters such as position, velocity and direction of motion, as well as the model of the environment in which the vehicle moves, we can describe constraints on various subsets of the set of possible trajectories of the vehicle that maintain the vehicle V in stable condition. Here, not only is each trajectory infinite, but the set of trajectories that keep V stable may also be infinite. The many-dimensional space describing the vehicle status, and other features of the vehicle is a region $X \subseteq R^n$, and the desired regions, where the vehicle needs to be can be treated as subsets of the same R^n. There are, potentially many such regions. One reasonable way to describing them is by means of constraints on subsets of R^n.

The designers of the solvers have also focused on the issues of both improving the processing of the logic programs, i.e. finding more efficient ways to search for a stable models, and improving the use of logic programs as a programming language. The latter task consists of extending the constructs available to the programmer to make programming easier and more readable. The extensions of ASP that we shall talk about in this paper can be viewed as part of this latter task.

The basic idea behind all the extensions that we shall discuss in this paper is to carefully consider the definition of the stable model or answer set semantics via the Gelfond-Lifschitz transform and to consider ways in which that general mechanism can be extended. To make our ideas more precise, we shall briefly review the definition of stable models for propositional and predicate logic programs.

A (propositional) *logic programming clause* is an expression of the form

$$C = p \leftarrow q_1, \ldots, q_m,\ \mathit{not}\, r_1, \ldots, \mathit{not}\, r_n \quad (1)$$

where $p, q_1, \ldots, q_m, r_1, \ldots, r_n$ are atoms from a fixed set of atoms At. The atom p in the clause above is called the *head* of C ($head(C)$), and the expression

$$q_1, \ldots, q_m, \textit{not}\, r_1, \ldots, \textit{not}\, r_n,$$

with ',' interpreted as the conjunction, is called the *body* of C ($body(C)$). The set $\{q_1, \ldots, q_n\}$ is called the *positive part of the body* of C ($posBody(C)$) and the set $\{r_1, \ldots, r_m\}$ is called the *negative part of the body* of C ($negBody(C)$). Given any set $M \subseteq At$ and atom a, we say that M satisfies a (*not* a), written $M \models a$ ($M \models$ *not* a), if $a \in M$ ($a \notin M$). We say that M satisfies C, written $M \models C$, if whenever M satisfies the body of C, then M satisfies the head of C. A normal logic program P is set of clauses of the form of (1). We say that $M \subseteq At$ is a model of P, written $M \models P$, if M satisfies every clause in C.

A (propositional) Horn clause is a logic programming clause of the form

$$H = p \leftarrow q_1, \ldots, q_m \tag{2}$$

where $p, q_1, \ldots, q_m \in At$. Thus in a Horn clause, the negative part of its body is empty. A Horn program P is a set of Horn clauses. Each Horn program P has a least model under inclusion relation, LM_P, which can defined using the one-step provability operator T_P. For any set A, let 2^A denote the set of all subsets of A. The one-step provability operator $T_P : 2^A \rightarrow 2^A$ associated with the Horn program P [vEK76] is defined by setting:

$$T_P(M) = \{p : \exists C \in P(p = head(C) \wedge M \models body(C))\} \tag{3}$$

for any $M \in 2^A$. We define $T_P^n(M)$ by induction by setting $T_P^1(M) = T_P(M)$ and $T_P^{n+1}(M) = T_P(T_P^n(M))$. Then the least model LM_P can be computed as

$$LM_P = T_P(\emptyset)^{\mathbb{N}} = \bigcup_{n \geq 1} T_P^n(\emptyset).$$

If P is a normal logic program and $M \subseteq A$, then the Gelfond-Lifschitz transform of P with respect to M [GL88] is the Horn program $GL_P(M)$ which results by eliminating those clauses C of the form (1) such that $r_i \in M$ for some i and replacing C by $p \leftarrow q_1, \ldots, q_n$ otherwise. We then say that M is a *stable model* or an *answer set* for P if M equals the least model of $GL_P(M)$.

We should note that the operator T_P makes perfectly good sense for any normal logic program [AvE82]. The fixpoints of the operator T_P are called *supported models* of P. One can prove that every answer set of P is a supported model. Supported models of P can be shown to coincide with models of the completion of P, $comp(P)$ [Cla78]. As $comp(P)$ is a propositional theory, one can use SAT solvers to compute its models and so, the supported models of P. By pruning those supported models that are not answer sets, one can also compute answer sets by means of SAT solvers. This possibility was successfully used in systems such as *assat* [LZ02] and *cmodels* [BL02]. Moreover *assat* and *cmodels* implement pruning by expanding the input program with the so-called *loop formulas* [LZ02]. The process can be viewed as a version of clause learning used in SAT solvers. Recent solvers which are improvements on *assat* and *cmodels* such as *clasp* are very efficient.

One can extend the notion of stable models to predicate logic programs as follows. A (predicate) *logic programming clause* is an expression of the form

$$C = p \leftarrow q_1, \ldots, q_m,\ \textit{not}\, r_1, \ldots, \textit{not}\, r_n \tag{4}$$

where $p, q_1, \ldots, q_m, r_1, \ldots, r_n$ are atoms from some fixed first order language $\mathcal{L}$. As in the case of propositional logic clauses, the atom p in the clause above is called the *head* of C ($head(C)$), and the expression $q_1, \ldots, q_m, \textit{not}\, r_1, \ldots, \textit{not}\, r_n$, with ',' interpreted as the conjunction, is called the *body* of C ($body(C)$). The set $\{q_1, \ldots, q_n\}$ is called the *positive body* of C ($posBody(C)$) and the set $\{r_1, \ldots, r_m\}$ is called the negative body of C ($negBody(C)$). A ground instance of the clause C is a substitution instance of C where we have uniformly replaced the free variables in C with ground terms, i.e. terms with no free variables, so that resulting substitution instance has no free variables. A predicate logic program P is a collection of clauses of the form (4). We then let $ground(P)$ denote the set of all ground instances of clauses in P. Thus $ground(P)$ can be thought of as propositional logic program. We then say that a collection of ground atoms M, i.e. a subset of atoms of $\mathcal{L}$ with no free variables, is a *stable model* or an *answer set* of P if and only if M is a stable model of $ground(P)$.

In this paper, we shall consider four different extensions of the basic stable model paradigm described above.

Extension 1. *Arbitrary Constraint Logic Programming.*
Our first extension is to follow the paper of Marek, Nerode, and Remmel [MNR95] and consider logic programs with arbitrary constraints. Notice that in the definition of stable model of P, the negative bodies of the clauses of P only play a role in determining which Horn clauses end up in $GL_P(M)$. Thus the idea of [MNR95] is to replace these negative bodies by arbitrary constraints so that we end up with clauses of the form

$$p \longleftarrow q_1, \ldots, q_n : \Psi. \tag{5}$$

Here Ψ is *any* type of constraint such that given $M \subseteq At$, we can decide whether M satisfies Ψ. Thus Ψ does not even have to be in the original language of the program and it could express an infinite constraint such as the ones studied by Marek, Nerode, and Remmel in [MNR97]. Thus replacing negative bodies by arbitrary constraints provides a rich way to reason about all sorts of infinite constraints in ASP which we call Arbitrary Constraint Logic Programming (ACLP).

Extension 2. *Adding set constraint atoms to logic programming.*
A powerful extension of Answer Set Programming stems out of the work of Niemelä et.al. [SNS02]. The idea was to use as building blocks of programs not only atoms and negated atoms, but expressions of the form kXl where X is a finite set of atoms, and k, l are nonnegative integers, smaller or equal than the size of X. The interpretation of such constraint is "at least k but not more than l of atoms from X are true in a putative model M".

Later Marek and Remmel [MR03] introduced set constraint atoms of the form $\langle X, \mathcal{F}\rangle$ where X is a set and $\mathcal{F}$ is a finite set of subsets of X. Subsequent research of many authors [MR03, GL02, MNT08, LT05a] led to significant progress in understanding such constraints. On the concrete level, arbitrary set constraints atoms include weight constraints, SQL constraints, parity constraints, and other kinds of common constraints, and, on the abstract level, include monotone, antimonotone, and convex constraints. Adding arbitrary set constraint atoms to logic programming is a natural mechanism that allows the user to reason about large variety of constraints in ASP solvers and SAT solvers.

Set constraint atoms $\langle X, \mathcal{F} \rangle$ where X is an infinite set and $\mathcal{F}$ is a finite set of 2^X can also be used to reason about infinite sets. For example, Cenzer, Marek, and Remmel [CRM05] studied constraints of the form $\langle X, \mathcal{F} \rangle$ where X is an infinite recursive set and $\mathcal{F}$ is a finite set of indices for certain recursive or recursively enumerable subsets of X.

We shall also briefly outline the work of Brik and Remmel [BR11a] that shows how one can use arbitrary set constraint atoms to reason about preferences in ASP. The ability to express preferences and to reason about them effectively has many important applications to problems in planning and negotiations. Recent work by Brik and Remmel [BR11a] has shown that set constraint atoms can be a very convenient and compact way to express a wide variety of such preferences. That is, suppose that we are given a set constraint atom $\langle X, \mathcal{F} \rangle$ and weight function $wt : 2^X \rightarrow \mathbb{Q}$ where $\mathbb{Q}$ is the set of rational numbers. Then our idea is that the weight function wt is defined in such a way so that we prefer those $F \in \mathcal{F}$ which have the smallest weight. For example, suppose that Dr. X is buying a car and the dealer offers several option packages such as you can have a red car with an automatic transmission with a high end CD player or you can have a blue car with standard transmission and a standard CD player. Suppose that the blue car costs \$25,000 and the red car costs \$35,000. Let B stand for blue, R stand for red, A stand for automatic transmission, S stand for standard transmission, HCD stand for high end CD player, and SCD stand for standard CD player. Suppose that the prices of the cars can be \$20,000, \$25,000, \$30,000, \$35,000. Then we let $X = \{B, R, A, S, HCD, SCD, 20000, 25000, 30000, 35000\}$ We can then view the set $F_1 = \{B, S, SCD, 20000\}$, $F_2 = \{R, A, HCD, 35000\}$, and $F_3 = \{B, S, HCD, 25000\}$ as option packages available from the car dealer. While an individual may prefer red cars to blue cars, standard transmissions to automatic transmission, and high end CD players to standard CD players and to get the car at the lowest possible price, there may be no such package as $\{R, S, HCD, 20000\}$. Thus the buyer has to choose from one the three packages F_1, F_2, or F_3 so that we may have a set constraint atom $\langle X, \{F_1, F_2, F_3\} \rangle$. Now we can insist that a model M satisfies the set constraint $\langle X, \mathcal{F} \rangle$ by adding a clause of the form

$$\langle X, \mathcal{F} \rangle \leftarrow . \tag{6}$$

One can use an auxiliary weighting function to expresses preference in this case. For example, we might define $wt(F_1) = 2$, $wt(F_2) = 1.5$ and $wt(F_3) = 1$. Of course, the buyer's spouse may have a different set of preferences so that we might want to create two copies of the $\langle X, \{F_1, F_2, F_3\} \rangle$, one for the husband and one for the wife. Thus we might want to consider programs which have several clauses of the form (6). This leads to a natural weighting on models M of the program defined to be the sum of $wt(M \cap X)$ for all such clauses. The idea is that lower weighted models satisfy more of the preferences incorporated by clauses of the form (6).

Often times in such situations, it is impossible to meet all individual preferences. This can lead to programs that do not have stable models. One way to handle this problem is to specify hard preferences, those that have to be satisfied, and soft preferences, those that do not necessarily have to be satisfied. Another approach is to look for looks for subsets of preferences which can be satisfied and this naturally leads one to search for maximal subprograms of a program P which do have stable models. To date, none of the ASP solvers have the ability to find such maximal subprograms. However, there is an algorithm called the forward chaining algorithm developed by Marek, Nerode, and Remmel [MNR94b] which does allow one to find such maximal subprograms when the original program does not have a stable model. Recently, Brik and Remmel [BR10] have combined the forward chaining algorithm with the Metropolis algorithm [Met53] to produce a novel Monte Carlo type algorithm to find such maximal subprograms.

Extension 3. *Set-based logic programming.*
Blair, Marek, and Remmel [BMR01] observed that the ASP formalism can be significantly extended by allowing atoms to represent sets in some fixed universe X. That is, instead having the intended underlying universe be the Herbrand base of the program, one replaces the underlying Herbrand universe by some fixed space X and has the atoms of the program specify subsets of X, i.e. elements of the set 2^X, the set of all subsets of X.

If we reflect for a moment on the basic aspects of logic programming with an Herbrand model interpretation, a slight change in our point of view shows that interpreting atoms as subsets of the Herbrand base is quite natural. In normal logic programming, we determine the truth value of an atom p in an Herbrand interpretation I by declaring $I \models p$ if and only if $p \in I$. However, this is equivalent to defining the sense, $[\![p]\!]$, of a ground atom p to be the set $\{p\}$ and declaring that $I \models p$ if and only if $[\![p]\!] \subseteq I$. By this simple move, we have permitted ourselves to interpret the sense of an atom as a subset, rather than the literal atom itself.

This given, Blair, Marek, and Remmel developed a system that they called *spatial logic programming* in [BMR01] in which they showed that it is a natural step to take the *sense* $[\![p]\!]$ of a ground atom p to be a fixed assigned subset of some nonempty set X and to define a $I \subseteq X$ to be a model of p, written $I \models P$, if and only if $[\![p]\!] \subseteq I$. This type of model theoretic semantics makes available, in a natural way, multiple truth values, intensional constructs, and interpreted relationships among the elements and subsets of X. Observe that the assignment $[\![\cdot]\!]$ of a *sense* to ground atoms is intrinsically intensional. Interpreted relationships among the elements and subsets of X allow the programs that use this approach, which was called *spatial logic programing* in [BMR01], to serve as front-ends for existing systems and still have a seamless model-theoretic semantics for the system as a whole.

In [BMR08], Blair, Marek, and Remmel showed that if the underlying space X has structure such as a topology or an algebraic structure such as a group, ring, field, or vector space, then a number of natural options present themselves. For example, if we are dealing with a topological space, one can compose the one step consequence operator T_P with an operator that produces topological closures of sets or interiors of sets. In such a situation, one ensures that the the extended T_P operator always produces closed sets or always produces open sets. Similarly, if the underlying space X is a vector space, one might insist that the extended T_P operator always produces a subspace of X or a subset of X which is convex closed. Notice that each of the operators: *closure, interior, span* and *convex-closure* is a *monotone idempotent operator*. That is, an operator $op : 2^X \rightarrow 2^X$ is an monotone operator if $I \subseteq J \Rightarrow op(I) \subseteq op(J)$ for all $I \subseteq J \subseteq X$ and is an idempotent operator if $op(op(I)) = op(I)$ for all $I \subseteq X$. We call such an operator a *miop* (pronounced "my op").

Unlike the situation in Extensions 1 and 2, there is a variety of options for how to interpret negation in spatial logic programming. In normal logic programming, a model M satisfies *not* p if $p \notin M$. From the set-based point of view when p is interpreted as a singleton $\{p\}$, this would be equivalent to saying that M satisfies *not* p if (i) $\{p\} \cap M = \emptyset$, or (equivalently) (ii) $\{p\} \not\subseteq M$. When the sense of p is a set with more than one element it is easy to see that saying that M satisfies *not* p if $[\![p]\!] \cap M = \emptyset$ which we call strong negation is different from saying that M satisfies *not* p if $[\![p]\!] \not\subseteq M$ which we call weak negation. There are thus two natural interpretations of the negation symbol. Again, when the underlying space has structure, one can get even more subsidiary types of negation by taking M to satisfy *not* p if $cl([\![p]\!]) \cap M = cl(\emptyset)$, or by taking M to satisfy *not* p if $cl([\![p]\!]) \not\subseteq M$ where cl is some natural miop. By composing the one-step provability operator with a miop, one naturally produced only those stable models which have desired properties such a being closed or being a subspace of a vector space. The familiar T_P operator corresponds to the case where the underlying miop operator is the simplest possible monotone idempotent operator, namely, the identity.

Blair, Marek, and Remmel [BMR08] called the extension of spatial logic programming with miops *set-based logic programming* (SBLP). Set-based logic programming provides yet another powerful way to reason about infinite sets as one is allowed to have the sense $[\![a]\!]$ of an atom a be an infinite subset of X. Indeed, Marek and Remmel [MR09] showed that one can effectively reason about infinite sets in SBLP provided that infinite sets have an indexing scheme with certain decision properties. For example, if the sense of all atoms are regular languages over some fixed finite alphabet Σ and $X = \Sigma^*$, then Marek and Remmel [MR09] proved that the stable models of a finite SBLP program P are always regular languages over Σ and that one can effectively decide whether a given regular language $L \subseteq \Sigma^*$ is a stable model of P.

Extension 4. *Hybrid Answer Set Programming*.

In [BR11b], Brik and Remmel introduced an extension of the ASP formalism in which one can reason about continuous trajectories which they called Hybrid Answer Set Programming (H-ASP). This extension is different than the other three in that the notion of a clause is greatly extended. To motivate this extension, consider the following situation where James Bond wants to take his Aston-Martin from point A to point B where the underlying trajectory his divided up into three regions: Region I which consists of ice and snow on a mountain, Region II which consists of lake, and Region III which consists of desert. With a push of button, Bond's Aston-Martin can change its configuration so that it can run on snow and ice, run as a boat, or run as a high performance car. This situation is pictured in Figure 1 where the rectangle in Region I is some building which must be avoided, the circles in Region II are some islands that must be avoided, and the hexagon is region III is some fort which must be avoided. We imagine that Bond makes certain decisions at regular intervals of length Δ as to what to do depending on his position $x(k\Delta)$, his velocity $v(k\Delta)$, his acceleration $a(k\Delta)$ and other requirements such as surface conditions, wind velocity and other logical conditions such as "I am being chased" or "I am at a minimum safe distance from an obstacle." In Figure 1, we have indicated Bond's position's at times $0, \Delta, 2\Delta, \ldots, 11\Delta$ by placing the $k\Delta$ at the position he has reached at time $k\Delta$.

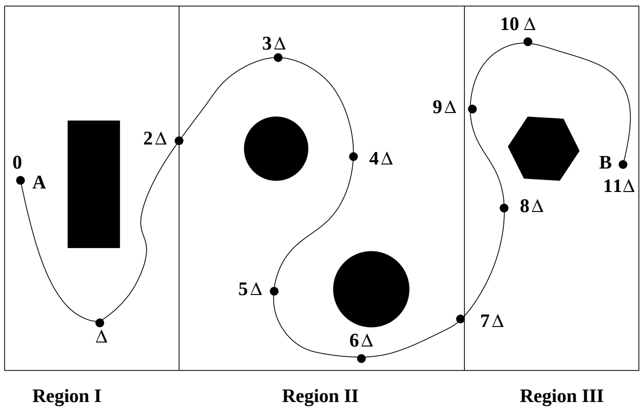

Figure 1: Picture of Bond's trajectory.

In [BR11b], Brik and Remmel discuss two systems of hybrid ASP. We shall briefly describe their simplified system of hybrid ASP in this introduction which they called *basic hybrid ASP* (BH-ASP) and discuss a more extended version of hybrid ASP in Section 4. In basic hybrid ASP, one specifies a parameter space S and a set of atoms At. The intended universe of an BH-ASP program is $At \times S$. That is, one thinks of the position and situation at time $k\Delta$ as being specified by a sequence of parameters $\vec{x}(k\Delta) = (x_1(k\Delta), x_2(k\Delta), \ldots, x_n(k\Delta)) \in S$ that specify such things as time, position, velocity, acceleration, etc. which are needed to compute the next position and the data base $M(k\Delta)_{\vec{x}(k\Delta)}$ of atoms a in At such that $(a, \vec{x}(k\Delta))$ are true at time $k\Delta$.

There are two types of clauses in a BH-ASP program.

1. *Stationary clauses* which are of the form

$$a \leftarrow a_1, ..., a_n, \textit{not } b_1, ..., \textit{not } b_m : O$$

where $a, a_1, ..., a_n, b_1, ..., b_m \in At$, $O \subseteq S$. The idea is that if $\vec{x}(k\Delta) \in O$, $a_i \in M(k\Delta)_{\vec{x}(k\Delta)}$ for $i = 1, ..., n$, and $b_j \notin M(k\Delta)_{\vec{x}(k\Delta)}$ for $j = 1, ..., m$, then $a \in M(k\Delta)_{\vec{x}(k\Delta)}$.

2. *Advancing clauses* which are of the form

$$a \leftarrow a_1, ..., a_n, \textit{not } b_1, ..., \textit{not } b_m : A, O$$

where $a, a_1, ..., a_n, b_1, ..., b_m \in A$, $O \subseteq S$, and A is an algorithm. The idea is that if $\vec{x}(k\Delta) \in O$, $a_i \in M(k\Delta)_{\vec{x}(k\Delta)}$ for $i = 1, ..., n$, and $b_j \notin M(k\Delta)_{\vec{x}(k\Delta)}$

for $j = 1, ..., m$, then we can apply the algorithm A to the set of parameters $\vec{x}(k\Delta)$ to compute the set of parameters $\vec{x}((k+1)\Delta)$ at the next time step and the clause specifies that $(a, \vec{x}((k+1)\Delta))$ holds.

Here for advancing clauses, we envision that algorithm A could require that one solve a differential or integral equation to get the next set of parameters or it could require solving some system of linear equations or some linear programming problem to get the next set of parameters, etc.. From this point of view, we can think of an advancing clause as input-output device. Of course, classical logic rules also can be thought of as input-output devices, but one rarely thinks in this sort of terms.

The outline of this paper is as follows. In Section 2 we shall discuss ACLP of Extension 1. In Section 3, we shall discuss various extensions of ASP that use set constraint atoms. In Section 4, we shall discuss set-based logic programming and how it can be used to reason about certain classes of infinite set effectively. In Section 5, we shall briefly introduce basic hybrid ASP and its more general extension called Hybrid ASP (H-ASP) as described by Brik and Remmel [BR11a]. Finally, in Section 6 we discuss conclusions and further research.

2 Arbitrary Constraint Logic Programming

In this section we discuss the variation of programs obtained by treating the negative part of the body of a clause as a constraint on applicability of a clause. That is, we shall give a detailed description of Arbitrary Constraint Logic Programming (ACLP) as described in Extension 1 of the introduction.

The basic Gelfond-Lifschitz transform mechanism of Answer Set Programming can be expressed as follows. The negative literals in the body of a clause serve as a "semaphore". Namely, when we guess a set of atoms M (a putative answer set), the negative part of the clause C tells us if the Horn part of C can be used in the computation or not. To make this intuition a bit more precise, given a clause

$$C = p \leftarrow q_1, \ldots, q_n, \textit{not}\ r_1, \ldots, \textit{not}\ r_n, \tag{7}$$

let $h(C)$ be the Horn clause $p \leftarrow q_1, \ldots, q_n$. Then $P_M = \{h(C) : M \models \neg negBody(C)\}$. Thus $\neg negBody(C)$ is a constraint on usability of $h(C)$. There is no reason why such constraints should be restricted to be only conjunctions of negative literals. Motivated by this observation, we introduce the concept of ACLP clause and of ACLP program. An ACLP clause is a string C:

$$p \leftarrow q_1, \ldots, q_m : \Phi_C$$

where $p, q_1, \ldots, q_m$ are atoms, and Φ_C is a formula of some language $\mathcal{L}$ for which we have a satisfaction relation $\models$ which allows us to test if $M \models \Phi_C$. An ACLP program is a set of ACLP clauses. Now, the idea is to generalize the "semaphore" as defined above. Namely, we first guess a set M of atoms and then we test if the constraint Φ_C is satisfied by M or not. If it is satisfied, $h(C)$ is placed in P_M, otherwise it is eliminated. Then we compute the least model of P_M and check if it coincides with M. In such situation, we call M a *constraint answer set* for P.

The simplest case is when Φ_C consists of conjunctions of negative literals only. In that case, we get nothing new. Indeed, let us assign to a clause C of the form (7), a constraint clause:

$$C' = p \leftarrow q_1, \ldots, q_n : \neg r_1, \ldots, \neg r_n$$

and $P' = \{C' : C \in P\}$. Then we have

Proposition 2.1. *A set of atoms M is an answer set for P if and only if M is a constraint answer set for P'.*

We note that the notion of answer set for constraint programs includes the notion of a supported model via the following construction. Given a clause C we assign to C a constraint clause C'' as follows

$$C'' = p \leftarrow : q_1, \ldots, q_n, \neg r_1, \ldots, \neg r_n$$

and set $P' = \{C'' : C \in P\}$. Then we have

Proposition 2.2. *A set of atoms M is a supported model for P if and only if M is a constraint answer set for P''.*

As long as the constraints Φ are taken from the propositional language generated by the set of atoms of the program P, the expressive power of the concept of constraint answer set does not increase. We have the following fact.

Proposition 2.3. *Let $\mathcal{PC}$ be class of constraint programs where all the constraints are propositional formulas. Then the existence problem for constraint answer sets of programs in $\mathcal{PC}$ is an NP-complete problem.*

In [PR96], Pollett and Remmel looked at a class of constraint programs where the constraints were quantified Boolean formulas over the set of atoms occurring in the program. That is, Pollett and Remmel consider programs whose clauses are of the form

$$p \leftarrow a_1, \ldots, a_n : B_1(\vec{b}_1), \ldots, B_n(\vec{b}_m) \qquad (*)$$

where $p, a_1, \ldots, a_n$ are propositional variables and B is a quantified Boolean formula and $\vec{b}_i$'s represents the free propositional variables in each B_i.

Let Σ^q_k denote the set of quantified Boolean formulas with at most k-alternations of quantifier type and whose outermost quantifier is an $\exists$. Similarly, let Π^q_k denote the set of quantified Boolean formulas with at most k-alternations of quantifier type and whose outermost quantifier is an $\forall$. In both cases, unless we say we are dealing with only sentences, we assume our formulas have free variables. Lastly, we write QBF_k to denote Boolean combinations of these two classes. In the $k = 0$ case, all of the above classes are the same. They each define the class of propositional formulas. We recall that the problem of determining whether a Σ^q_k-sentence is true is Σ^P_k-complete and the problem of determining whether a Π^q_k-sentence is true is Π^p_k-complete. Given an assignment to the free variables of a QBF_k formula, we can determine whether or not it is true in Δ^P_{k+1}.

We now define LP_k is the class of *finite* arbitrary constraint logic programs whose constraints are all in QBF_k and
$LP_\infty = \bigcup_{k \geq 0} LP_k$.

It turns out that Proposition 2.3 generalizes in a straightforward way. That is, Pollett and Remmel proved the following theorem.

Theorem 2.1.

1. *The problem of determining whether an LP_k program has an answer set is Σ^P_{k+1}-complete.*
2. *The problem of determining whether a finite LP_∞ program has an answer set is $PSPACE$-complete.*
3. *The problem of deciding whether a given variable a is in an answer set of an LP_k program is Σ^P_{k+1}-complete.*
4. *The problem of deciding whether a given variable a is in an answer set of an LP_∞ program is $PSPACE$-complete.*

Pollett and Remmel pointed out that there are several ways to generalize the notion of logic programming with quantified Boolean constraints that fits the paradigm of ACLP programs. For example, rather than take our atoms of quantified Boolean formulas to be just propositional variables, we could let them be propositional variables and expressions of the form $a_1a_2\ldots a_n \in A$. That is, checking if the concatenation of some string propositional variables is in an oracle A. Given a variable assignment ν, we say $\bar{\nu}(a_1a_2\ldots a_n \in A) = 1$ if and only if the string $\nu(a_1)\nu(a_2)\ldots\nu(a_n)$ is in the set A. Thus, there is a well defined semantics for such formulas. We can thus define the classes $\Sigma^q_k(A)$, $\Pi^q_k(A)$, and $QBF_k(A)$ and use them in our logic programming theories. Hence, we can define $LP_k(A)$ to be those finite logic programs with $QBF_k(A)$ constraints. $SAT_k(A)(y)$ is Σ^P_{k+1}-complete [GJ79]. Then Pollett and Remmel proved the following generalization of Theorem 2.1

Theorem 2.2.

1. *The problem of deciding whether an $LP_k(A)$ program has an answer set is $\Sigma^P_{k+1}(A)$-complete.*
2. *The problem of deciding whether an $LP_\infty(A)$ program has an answer set is $PSPACE(A)$-complete.*
3. *The problem of deciding whether a given variable a is in an answer set of an $LP_k(A)$ program is $\Sigma^P_{k+1}(A)$-complete.*
4. *The problem of deciding whether a given variable a is in an answer set of an $LP_\infty(A)$ program is $PSPACE(A)$-complete.*

Going beyond propositional logic and quantified Boolean formulas leads to interesting but not investigated class of constraint programs. Specifically, for integers $j \geq 2$ and $0 \leq i < j$, we define a new formula $imodj$ and stipulate, for a finite set of atoms M, $M \models imodj$ if $|M| \equiv i \bmod j$. It should be clear that due to the localization properties of propositional logic and quantified Boolean formulas, the formulas $i \bmod j$ are not definable in the languages defined above. But once we defined satisfaction for expressions of the form $imodj$, we have immediately satisfaction relation for the language of propositional formulas with atoms of the form $imodj$. Let us call such constraints *mod-constraints*. We can then consider programs that use mod-constraints. Parity constraints as considered in [MNR95] are from this language. Complexity issues for the constraint programs in this language have not been studied. We illustrate the programs with mod-constraints with a simple example.

Example 2.1. Let P be a mod-constraint program consisting of the following clauses.

1. $p \leftarrow q, u, w$
2. $r \leftarrow s, v$
3. $u \leftarrow r$
4. $v \leftarrow r$
5. $s \leftarrow: 2mod3$
6. $t \leftarrow: 2mod3$
7. $q \leftarrow: 1mod3$
8. $w \leftarrow: 1mod3$
9. $u \leftarrow: 1mod3$

We then check that $M_1 = \{p, s, t, u\}$ is a mod-constraint answer set for P. Indeed, the reduct of P by M_1 yields the program:
$p \leftarrow q, u, w$
$r \leftarrow s, v$
$u \leftarrow r$
$v \leftarrow r$
$q \leftarrow$
$w \leftarrow$
$u \leftarrow$
with M_1 as the least model.
Likewise, we leave to the reader the task of testing that $M_2 = \{s, t\}$ is another mod-constraint answer set for P.

The family of answer sets for a constraint program does not need to form an antichain. Minimal answer set of constraint programs have not been studied.

3 Logic Programming with Set Constraint Atoms

In this section we define a number of generalizations of cardinality and weight constraints.

One can think about propositional atoms as very simple constraints on assignments. Specifically, an atom p is just a requirement that the intended model M satisfies p. Likewise, a clause

$$C = p \leftarrow q_1, \ldots, q_m, \textit{not}\, r_1, \ldots \textit{not}\, r_n$$

propagates constraints and can be informally interpreted as a constraint on the intended model M. That is, once M satisfies the body of C, it has to satisfy the head of C as well. This point of view has been proposed in [Nie99, MT99] with the idea that the program specifies constraints of the problem at hand, while the answer sets encode intended solutions of the problem.

Once such point of view is adopted, it is natural to ask whether the ASP mechanism could be adopted to propagation of more complex constraints. In [SNS02] Niemelä and his collaborators have shown that, indeed one can adopt ASP formalism to a situation where the constraints are more complex. Specifically, in [SNS02] its authors show a construction dealing with two specific types of constraints: cardinality constraints and its generalization, weight constraints. These constraints, under the name of pseudo-Boolean constraints are used in logic design and also in combinatorial optimization. Next, we describe the case of cardinality constraints which are the simple case of pseudo-Boolean constraints where all the weights on the atoms are equal to 1. A cardinality constraint is a string C of the form kXl where X is a finite set of atoms and $k \leq l \leq |X|$. When $k = 0$ we drop it from the description of C, and similarly we drop l when it is $|X|$. When M is a set of atoms, we write $M \models kXl$ if $k \leq |X \cap M| \leq l$. We observe that $M \models p$ if and only if $M \models 1\{p\}$, and $M \models \mathit{not}\, p$ if and only if $M \models \{p\}0$. Thus cardinality constraints generalize atoms. The satisfaction relation $\models$ can be extended to the language treating cardinality constraints as new atoms. We can also write program clauses where the heads of clauses and elements of bodies are cardinality constraints. The notion of a model of such clause generalizes the usual notion of a model of a program. Specifically, a set of atoms M satisfies a clause

$$kXl \leftarrow k_1Y_1l_1, \ldots k_mY_ml_m \tag{8}$$

if for some $j \leq m$, $M \not\models k_jY_jl_j$, or if $M \models kXl$. A set M is a model of a program if it is a model of each clause of the program. The cardinality constraints considered here concern the sets of atoms; the original definition in [SNS02] used literals, not only atoms.

The notion of an answer set of a cardinality constraint program P involves a significant modification of the usual Gelfond-Lifschitz transform (GL-transform). We will call it the NSS-transform. A number of steps are performed. As we shall see, some of these step are different from the steps used to define the GL-transform. Let us guess M, a putative answer set. First, we require that M is a model of P. Next, one eliminates all clauses C in P of the form of (8) such that $M \not\models \mathit{body}(C)$. The final step transforms each remaining clause C of form (8) of the program P in two ways.

(a) First one eliminates the upper bounds of all constraints in the body of C. Let us call the resulting clause C'

(b) Second, one replaces the clause C' by all clauses of the form $p \leftarrow \mathit{body}(C')$ such that $p \in M \cap X$.

The NSS-transform of P is the set of clauses produced from P by this process. The program P'' has two key properties. First, the heads of clauses of P'' are atoms. Second, for every clause C'' of the program P'', the collection of sets N that satisfy the body of the clause C'' is upper-closed. That is, if $N \models \mathit{body}(C'')$ and $N \subseteq N'$, then $N' \models \mathit{body}(C'')$. This implies that the one-step provability operator associated with P'' and M is monotone. It is also continuous, but, since in this section we shall limit ourselves to finite cardinality constraints programs, we shall not discuss this issue further. Thus by Knaster-Tarski theorem, this operator $T_{P,M}$ possesses a least fixpoint.

If that fixpoint coincides with M, we call M an answer set of P. Notice that by the construction, an answer set must be a model of P.

We observe that for normal logic programs, the SMS-transform of a program eliminates more clauses than the GL-transform. That is, the GL-transform tests only the negative part of the clauses, not the entire body. Yet, for normal programs, the fixpoints obtained via GL-transform and via SMS-transform are the same. The reason is that if a clause C has the property that $posBody(C) \setminus M \neq \emptyset$ and C survives the GL-test, then C will fire only if an atom $q_i \in posBody(C) \setminus M$ is computed. This guarantees, however, that M is different from the fixpoint and, hence, is not an answer set [MT98].

A very similar construction can be done for weight constraints where atoms are weighted and the bounds k and l are on cumulative weight of $M \cap X$. We should note that if one allows weight functions which admit negative values, then the results are not always intuitive and alternative approaches have been proposed [LPST07, FPL11].

Recall that we defined a set constraint to be a pair $\langle X, \mathcal{F}\rangle$ where X is a finite set and $\mathcal{F} \subseteq 2^X$. A *set constraint* (SC) *clause* is a string of the form

$$\langle X, \mathcal{F}\rangle \leftarrow \langle Y_1, \mathcal{G}_1\rangle, \ldots, \langle Y_n, \mathcal{G}_n\rangle.$$

A set constraint program is a finite set of SC clauses.

We note that there are interesting constraints that are not cardinality constraints nor weight constraints. An example of one such constraint is a constraint analogous to $1mod3$ discussed above, specifically, $M \models \langle X, 1mod3\rangle$ if $|M \cap X| \equiv 1 \mathrm{mod} 3$. Many other natural constraints can be defined as set-constraints. In fact generalized quantifiers over finite sets of atoms [Lib04] can be expressed this way.

To see how the NSS-transform can be utilized for SC programs, we first need to define the satisfaction relation like we did in case of cardinality constraints. Let M be a set of atoms and $K = \langle X, \mathcal{F}\rangle$ be a set constraint atom. We say that $M \models K$ if $X \cap M \in \mathcal{F}$. This is an abstract version of the satisfaction relation defined above. Let us notice that going to the abstract version of the constraint may significantly increase the size of the representation. For example, the cardinality constraint $1\{p, q, r\}2$ becomes $\langle \{p, q, r\}, \{\{p\}, \{q\}, \{r\}, \{p, q\}, \{p, r\}, \{q, r\}\}\rangle$.

We now show how the NSS-transform can be adopted for SC programs. The following observation is easy.

Proposition 3.1. *For every set X and a family $\mathcal{F}$ of subsets of X there exists a $\subseteq$-least family $\mathcal{G}$ of subsets of X such that*

1. *$\mathcal{F} \subseteq \mathcal{G}$*
2. *$\mathcal{G}$ is upper-closed that is, whenever $A \in \mathcal{G}$ and $A \subseteq B \subseteq X$ then $B \in \mathcal{G}$.*

Since the family $\mathcal{G}$ is $\subseteq$-least, it is unique. Hence we call the unique family $\mathcal{G}$ whose existence is established by Proposition 3.1, the closure of $\mathcal{F}$ and denote it $\overline{\mathcal{F}}$. It is easy to see that when $\langle X, \mathcal{F}\rangle$ is equivalent to the cardinality constraint kXl, then $\langle X, \overline{\mathcal{F}}\rangle$ is equivalent to the cardinality constraint kX.

At a cost of a possible large representation, we can describe answer sets for programs that include arbitrary set-constraints. Again the process of defining a stable model for SC programs is based on some form of "Horn" programs, GL-reduction, and least fixpoints of the one-step provability operators for Horn programs.

We will call an SC-clause *Horn* if

1. the head of that clause is a single atom (recall that atoms are represented as set constraints) and
2. whenever $\langle X_i, \mathcal{F}_i \rangle$ appears in the body, then $\mathcal{F}_i$ is an upper closed family of subsets of X_i.

A set-constraint Horn program P is an SC-program which consists entirely of Horn clauses. There is a natural one-step provability operator associated to an SC-Horn program P, $T_P : 2^X \rightarrow 2^X$ where X is the underlying set of atoms of the program. Specifically, $T_P(S)$ consists of all p such that there is clause

$$C = p \leftarrow \langle X_1, \mathcal{F}_1 \rangle, \ldots, \langle X_m, \mathcal{F}_m \rangle \in P$$

such that S satisfies the body of C. Our definitions ensure that T_P is a monotone operator and hence each SC-Horn program P has a least model M^P. M^P can be computed in a manner analogous to the computation of the least model of a definite Horn program as $T_P^\omega(\emptyset)$. The NSS transform $\mathbf{NSS}_M(P)$ of the set-constraint program P for a given set of atoms M which is a model of P is defined as follows. First eliminate all clauses with bodies not satisfied by M. Next, for each remaining clause

$$\langle X, \mathcal{F} \rangle \leftarrow \langle X_1, \mathcal{F}_1 \rangle, \ldots, \langle X_m, \mathcal{F}_m \rangle$$

and each $p \in M \cap X$, put the clause

$$p \leftarrow \langle X_1, \overline{\mathcal{F}}_1 \rangle, \ldots, \langle X_m, \overline{\mathcal{F}}_m \rangle$$

into $\mathbf{NSS}_M(P)$. Clearly the resulting program $\mathbf{NSS}_M(P)$ is an SC-Horn program and hence has a least model $M^{\mathbf{NSS}_M(P)}$. M is a stable model of P if M is a model of P and $M = M^{\mathbf{NSS}_M(P)}$. It can be shown that this construction corresponds to the same notion of Gelfond-Lifschitz stable models when we restrict ourselves to ordinary logic programs.

We should note that there are other semantics available for SC programs. For example, Son, Pontelli, and Tu [SPT07], observed that the stable models for an SC programs may be included one in another. That is, consider the following SC program P.

$a \leftarrow$
$b \leftarrow$
$c \leftarrow q$
$q \leftarrow \langle \{a, b, c\}, \{\{a, b, c\}\} \rangle$

One can easily checked that there are two stable models $M_1 = \{a, b, c, q\}$ and $M_2 = \{a, b\}$. An alternative semantics that does not allow for nested stable models was defined by Son, Pontelli, and Tu [SPT07].

We end this section with a few remarks on how set constraint atoms can allows us to reason about infinite sets and preferences.

3.1 Using set constraint atoms to reason about infinite sets.

Cenzer, Marek, and Remmel [CRM05] suggested a way to use set constraint atoms to reason about infinite sets. The basic idea is as follows. First we allow X to be an infinite recursive set and assume that we have a particular coding scheme for some family of subsets of a set X. Let $\mathcal{F}$ be a finite family of such codes. We will write C_e for the set with the code e. Then we can write two types of constraints. One constraint $\langle X, \mathcal{F}\rangle^{\subseteq}$ has the meaning that the putative set of integers M satisfies $\langle X, \mathcal{F}\rangle^{\subseteq}$ if and only if $M \cap X \supseteq C_e$ for some $e \in \mathcal{F}$. Similarly, we shall also consider constraints of the form $\langle X, \mathcal{F}\rangle^{=}$ where we say that M satisfies $\langle X, \mathcal{F}\rangle^{=}$ if and only if $M \cap X = C_e$ for some $e \in \mathcal{F}$. Observe that constraints of the form $\langle X, \mathcal{F}\rangle^{\subseteq}$ behave like atoms p in that they are preserved when the set grows while constraints of the form $\langle X, \mathcal{F}\rangle^{=}$ behave like constraints *not* p in that they are not always preserved as the set grows.

Now, it is clear that once we introduce these types of constraint schemes, we can consider various coding schemes for the set of indices. For example, Cenzer, Marek and Remmel [CRM05] used three such schemes: explicit indices of finite sets, recursive indices of recursive sets and recursively enumerable (r.e.) indices of recursively enumerable (r.e.) sets. They then defined extended set constraint clause C to be a clause of the form

$$\langle X, \mathcal{A}\rangle^{*} \leftarrow \langle Y_1, \mathcal{B}_1\rangle^{\subseteq}, \ldots, \langle Y_k, \mathcal{B}_k\rangle^{\subseteq}, \langle Z_1, \mathcal{C}_1\rangle^{=}, \ldots, \langle Z_l, \mathcal{C}_l\rangle^{=},$$

where $*$ is either $=$ or $\subseteq$.

Formally, Cenzer, Marek, and Remmel defined three types of indices three types of indices (i.e. codes) for certain subsets of the natural numbers $\mathbb{N}$.

(1) **Explicit indices of finite sets**. For each finite set $F \subseteq \mathbb{N}$, we define the explicit index of F as follows. The explicit index of the empty set is 0 and the explicit index of $\{x_1 < \cdots < x_n\}$ is $2^{x_1} + \cdots + 2^{x_n}$. We shall let F_n denote the finite set whose index is n.

(2) **Recursive indices of recursive sets**. Let $\phi_0, \phi_1, \ldots,$ be an effective list of all partial recursive functions. By a recursive index of a recursive set R, we mean an e such that ϕ_e is the characteristic function of R. If ϕ_e is a total $\{0,1\}$-valued function, then R_e will denote the set $\{x \in \mathbb{N} : \phi_e(x) = 1\}$.

(3) **R.e. indices of r.e. sets**. By a r.e. index of a r.e. set W, we mean an e such that W equals the domain of ϕ_e, that is, $W_e = \{x \in \mathbb{N} : \phi_e(x) \text{ converges}\}$.

Then for any subset $M \subseteq \mathbb{N}$, we shall say that M *is a model of* $\langle X, \mathcal{F}\rangle^{=}$, written $M \models \langle X, \mathcal{F}\rangle^{=}$, if there exists an $e \in \mathcal{F}$ such that $M \cap X$ equals that set with index e. Similarly, we shall say that M *is a model of* $\langle X, \mathcal{F}\rangle^{\subseteq}$, written $M \models \langle X, \mathcal{F}\rangle^{\subseteq}$, if there exists an $e \in \mathcal{F}$ such that $M \cap X$ contains the set with index e.

Based on these three different types of indices, Cenzer, Marek, and Remmel [CRM05] considered three different types of constraints.

(A) **Finite constraints**. Here we assume that we are given an explicit index x of a finite set X and a finite family $\mathcal{F}$ of explicit indices of finite subsets of X. We identify the finite constraints $\langle X, \mathcal{F}\rangle^=$ and $\langle X, \mathcal{F}\rangle^\subseteq$ with their codes, $ind(0,0,x,n)$ and $ind(0,1,x,n)$ respectively where $\mathcal{F} = F_n$, that is, the finite set with explicit index n. Here the first coordinate 0 tells us that the constraint is finite, the second coordinate is 0 or 1 depending on whether the constraint is $\langle X, \mathcal{F}\rangle^=$ or $\langle X, \mathcal{F}\rangle^\subseteq$, and the third and fourth coordinates are the codes of X and $\mathcal{F}$ respectively.

(B) **Recursive constraints**. Here we assume that we are given a recursive index x of a recursive set X and a finite family $\mathcal{R}$ of recursive indices of recursive subsets of X. Again we shall identify the recursive constraints $\langle X, \mathcal{R}\rangle^=$ and $\langle X, \mathcal{R}\rangle^\subseteq$ with their codes, $ind(1,0,x,n)$ and $ind(1,1,x,n)$ respectively, where $\mathcal{R} = F_n$. Here the first coordinate 1 tells us that the constraint is recursive, the second coordinate is 0 or 1 depending on whether the constraint is $\langle X, \mathcal{R}\rangle^=$ or $\langle X, \mathcal{R}\rangle^\subseteq$, and the third and fourth coordinates are the codes of X and $\mathcal{R}$ respectively.

(C) **R.e. constraints**. Here we are given a r.e. index x of a r.e. set X and a *finite* family $\mathcal{W}$ of r.e. indices of r.e. subsets of X. Again we identify the finite constraints $\langle X, \mathcal{W}\rangle^=$ and $\langle X, \mathcal{W}\rangle^\subseteq$ with their codes, $ind(2,0,x,n)$ and $ind(2,1,x,n)$ respectively, where $\mathcal{W} = F_n$. The first coordinate 2 tells us that the constraint is r.e., the second coordinate is 0 or 1 depending on whether the constraint is $\langle X, \mathcal{W}\rangle^=$ or $\langle X, \mathcal{W}\rangle^\subseteq$, and the third and fourth coordinates are the codes of X and $\mathcal{W}$.

An *extended set constraint* (ESC) *clause* is defined to be a clause of the form

$$\langle X, \mathcal{A}\rangle^* \leftarrow \langle Y_1, \mathcal{B}_1\rangle^\subseteq, \ldots, \langle Y_k, \mathcal{B}_k\rangle^\subseteq, \langle Z_1, \mathcal{C}_1\rangle^=, \ldots, \langle Z_l, \mathcal{C}_l\rangle^= \tag{9}$$

where $*$ is either $=$ or $\subseteq$. We shall refer to $\langle X, \mathcal{A}\rangle^*$ as the head of C, written $head(C)$, and $\langle Y_1, \mathcal{B}_1\rangle^\subseteq, \ldots,$ $\langle Y_k, \mathcal{B}_k\rangle^\subseteq$, $\langle Z_1, \mathcal{C}_1\rangle^=, \ldots, \langle Z_l, \mathcal{C}_l\rangle^=$ as the body of C, written $body(C)$. Here either k or l may be 0. M is said to be a model of C if either M does not model every constraint in $body(C)$ or $M \models head(C)$. *An extended set constraint* (ESC) program P is a set of clauses of the form of (1).

A (ESC) *Horn program* P is a set of clauses of the form

$$\langle X, \mathcal{A}\rangle^\subseteq \leftarrow \langle Y_1, \mathcal{B}_1\rangle^\subseteq, \ldots, \langle Y_k, \mathcal{B}_k\rangle^\subseteq. \tag{10}$$

where $\mathcal{A}$ is a singleton, that is $\mathcal{A}$ consists of a single index. We define the *one-step provability operator*, $T_P : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$, so that for any $S \subseteq \mathbb{N}$, $T_P(S)$ is the union of the set of all D_e such that there exists a clause $C \in P$ such that $S \models body(C)$, $head(C) = \langle X, \mathcal{A}\rangle^\subseteq$ and $A = \{e\}$ where $D_e = F_e$ if $head(C)$ is a finite constraint, $D_e = R_e$ if $head(C)$ is a recursive constraint, and D_e is W_e if $head(C)$ is an r.e. constraint. It is easy to see that T_P is a monotone operator and hence there is a least fixpoint of T_P which we denote by N^P. Moreover it is easy to check that N^P is a model of P.

If P is an ESC Horn program in which the body of every clause consists of *finite* constraints, then one can easily prove that the least fixpoint of T_P is reached in ω-steps, that is, $N^P = T_P^\omega(\emptyset)$. However, if we allow clauses whose bodies contain either recursive or r.e. constraints, then we can no longer guarantee that we reach the least fixpoint of T_P in ω steps. Here is an example.

Example 3.1. Let e_n be the explicit index of the set $\{n\}$ for all $n \geq 0$, let w be a recursive index of $\mathbb{N}$ and f be a recursive index of the set of even numbers E. Consider the following program.

$$\begin{array}{rcl} \langle\{0\},\{e_0\}\rangle^{\subseteq} & \leftarrow & \\ \langle\{2x+2\},\{e_{2x+2}\}\rangle^{\subseteq} & \leftarrow & \langle\{2x\},\{e_{2x}\}\rangle^{\subseteq} \text{ (for every number } x\text{)} \\ \langle\mathbb{N},\{w\}\rangle^{\subseteq} & \leftarrow & \langle E,\{f\}\rangle^{\subseteq} \end{array}$$

Clearly $\mathbb{N}$ is the least model of P but it takes $\omega+1$ steps to reach the fixpoint. That is, it is easy to check that $T_P^{\omega} = E$ and that $T_P^{\omega+1} = \mathbb{N}$.

Once we have a notion of ESC Horn program, we are in a position to define the analogue of stable models for ESC programs.

Definition 3.1. *Suppose that M is a model of an ESB program P.*

1. *We define the analogue of the NSS-transform by saying that* **NSS**$_M(C)$, *where $C \in P$ is a clause of the form (1), is nil if M does not satisfy the body of C. If M does satisfy the body of C, then since M is model of P, it must also be a model of the head of C, $\langle X,\mathcal{A}\rangle^*$ where $*$ is either $=$ or $\subseteq$. If $* = \subseteq$, there must be an explicit (recursive, r.e.) index in $\mathcal{A}$, of such that either $M \cap X$ contains the set with index e and for each such e, we add the clause*

$$\langle X,\{e\}\rangle^{\subseteq} \leftarrow \langle Y_1,\mathcal{B}_1\rangle^{\subseteq},\ldots,\langle Y_k,\mathcal{B}_k\rangle^{\subseteq},\langle Z_1,\mathcal{C}_1\rangle^{\subseteq},\ldots,\langle Z_l,\mathcal{C}_l\rangle^{\subseteq}. \quad (11)$$

   Similarly, if $*$ is $=$, there must be an index e such that $M \cap X$ is the set coded by e and again for each such e, we add the clause

$$\langle X,\{e\}\rangle^{\subseteq} \leftarrow \langle Y_1,\mathcal{B}_1\rangle^{\subseteq},\ldots,\langle Y_k,\mathcal{B}_k\rangle^{\subseteq},\langle Z_1,\mathcal{C}_1\rangle^{\subseteq},\ldots,\langle Z_l,\mathcal{C}_l\rangle^{\subseteq}. \quad (12)$$

   Then **NSS**$_M(P) = \{$**NSS**$_M(C) : C \in P\}$ *will be an ESB Horn program.*

2. *We then say that M is a* stable model *of P if M is a model of P and M equals the least model of* **NSS**$_M(P)$.

Cenzer, Marek, and Remmel explored the complexity of the least models of recursive ESC Horn programs and recursive ESC programs in [CRM05].

3.2 Using set constraint atoms to reason about preferences.

In this subsection, we briefly describe how we can use set constraint atoms to describe preferences based on ideas in a forthcoming paper by Brik and Remmel [BR11b]. The basic idea is to consider triples of the from $\langle X,\mathcal{F},wt\rangle$ or $\langle X,\mathcal{F},\preccurlyeq\rangle$ where

1. X is a finite set of atoms,
2. $\mathcal{F} \subseteq 2^X$,
3. $wt : \mathcal{F} \to [0,\infty) \subseteq \mathbb{R}$,

4. $\preccurlyeq$ is a partial order in $\mathcal{F}$.

We call triples of the form $\langle X, \mathcal{F}, wt \rangle$ *weight preference set constraint atoms* and triples of the form $\langle X, \mathcal{F}, \preccurlyeq \rangle$ *partially ordered preference set constraint atoms*. We say that a set of atoms M is satisfies $\langle X, \mathcal{F}, wt \rangle$ or $\langle X, \mathcal{F}, \preccurlyeq \rangle$ if and only if M satisfies $\langle X, \mathcal{F} \rangle$.

Now suppose that we have an SC program P which in addition has a finite set of clauses T of the form

$$\langle X_i, \mathcal{F}_i, wt_i \rangle \leftarrow$$

$i \in \{1, \ldots, n\}$. Now suppose that M is a stable model of $P \cup T$. Then we can define the weight of the model M as

$$W(M) = \sum_{i=1}^{n} wt_i(X_i \cap M).$$

As described in the introduction, we can use the weight functions to describe our preferences for what we want $M \cap X_i$ to be by saying that for $F_1, F_2 \in \mathcal{F}_i$, F_1 is preferred over F_2 if $wt_i(F_1) < wt_i(F_2)$. Then we say that a stable model M_1 of $P \cup T$ is preferred over the stable model M_2 of $P \cup T$ if $W(M_1) < W(M_2)$. Thus the introduction of weight preference set constraint atoms can lead to a natural weighting of stable models which can be used to model preferences.

Similarly, suppose that we have an SC program P which in addition has a finite set of clauses T of the form

$$\langle X_i, \mathcal{F}_i, \preccurlyeq_i \rangle \leftarrow$$

for $i \in \{1, \ldots, n\}$. Now suppose that we are given two stable models M_1 and M_2 of $P \cup T$. Then we say that $M_1 \preccurlyeq M_2$ if and only if $M_1 \cap X_i \preccurlyeq_i M_2 \cap X_i$ for $i = 1, \ldots, n$. Thus the introduction of partial order preference set constraint atoms can lead to a natural partial order on stable models which can be used to model preferences.

4 Set-Based Logic Programming

We start this section with a review the basic definitions of set-based logic programming as introduced by Blair, Marek, and Remmel [BMR08]. The syntax of set-based logic programs will essentially be the syntax of DATALOG programs with negation. We will then briefly discuss some results of Marek and Remmel [MR09] on conditions which ensure that we can effectively process set-based logic programs

Following [BMR08], we define a **set-based augmented first-order language** (**set-based language**, for short) $\mathcal{L}$ as a triple $(L, X, [\![\cdot]\!])$, where
(1) L is a language for first-order predicate logic (without function symbols other than constants),
(2) X is a nonempty (possibly infinite) set, called the **interpretation space**, and
(3) $[\![\cdot]\!]$ is a mapping from the ground atoms of L to the power set of X, called the *sense assignment*. If p is an atom, then $[\![p]\!]$ is called the *sense* of p.

Intuitively, one can treat the set of atoms A of $\mathcal{L}$ as a set of descriptions or codes of subsets of X. For example, if $X = \Sigma^*$ where Σ is a finite alphabet, then a description might be a regular expression for a language $A \subseteq X$ or a deterministic finite automaton (DFA) that accepts L. If $X = \mathbb{R}^n$, then a convex polygon of X can be described by the finite set of extreme points of X. We shall see later that the properties we need to effectively process set-based logic programs is that our set of atoms or descriptions come with algorithms which allow us to decide things like whether for any given atoms A and B, $[\![A]\!] \subseteq [\![B]\!]$ or $[\![A]\!] \cap [\![B]\!] = \emptyset$ holds and how to find an atom or code C such that $[\![C]\!] = [\![A]\!] \cup [\![B]\!]$.

For the rest of this section, we shall fix a set X and a first order language $\mathcal{L}$ with no function symbols except constants. We let HB_L denote the Herbrand base of L, i.e. the set of atoms of L. We omit the subscript L when the context is clear. We let 2^X be the power set of X. Given $[\![\cdot]\!] : \mathrm{HB}_L \longrightarrow 2^X$, an *interpretation* I of the set-based language $\mathcal{L} = (L, X, [\![\cdot]\!])$ is a subset of X.

A set-based logic programming clause is a clause of the form

$$\mathcal{C} = A \leftarrow B_1, \ldots, B_n, \textit{not}\, C_1, \ldots, \textit{not}\, C_m. \tag{13}$$

where A, B_i, and C_j are atoms for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. We let $head(\mathcal{C}) = A$, $Body(\mathcal{C}) = B_1, \ldots, B_n, \textit{not}\, C_1, \ldots, \textit{not}\, C_m$, and $posBody(\mathcal{C}) = \{B_1, \ldots, B_m\}$, and $negBody(\mathcal{C}) = \{C_1, \ldots, C_m\}$. A set-based program is a set of clauses of the form (13) and a set-based Horn program is a set of clauses of the form (13) which contain no occurrences of *not* .

A second component of a set-based logic program is one or more monotonic idempotent operators $O : 2^X \to 2^X$ that are associated with the program. Recall that an operator $O : 2^X \to 2^X$ is *monotonic* if for all $Y \subseteq Z \subseteq X$, we have $O(Y) \subseteq O(Z)$ and is *idempotent* if for all $Y \subseteq X$, $O(O(Y)) = O(Y)$. We call a monotonic idempotent operator a *miop* (pronounced "my op"). We say that a set Y is *closed* with respect to miop O if and only if $Y = O(Y)$.

For example, suppose that the interpretation space X is either $\mathbf{R}^n$ or $\mathbf{Q}^n$ where $\mathbf{R}$ is the reals and $\mathbf{Q}$ is the rationals. Then, X is a topological vector space under the usual topology so that we have a number of natural miop operators:

1. $op_{id}(A) = A$, i.e. the identity map is simplest miop operator,
2. $op_c(A) = \bar{A}$ where $\bar{A}$ is the smallest closed set containing A,
3. $op_{int}(A) = int(A)$ where $int(A)$ is the interior of A,
4. $op_{convex}(A) = K(A)$ where $K(A)$ is the convex closure of A, i.e. the smallest set $K \subseteq X$ such that $A \subseteq K$ and whenever $x_1, \ldots, x_n \in K$ and $\alpha_1, \ldots, \alpha_n$ are elements of the underlying field ($\mathbf{R}$ or $\mathbf{Q}$) such that $\sum_{i=1}^n \alpha_i = 1$, then $\sum_{i=1}^n \alpha_i x_i$ is in K, and
5. $op_{subsp}(A) = (A)^*$ where $(A)^*$ is the subspace of X generated by A.

We should note that (5) is a prototypical example if we start with an *algebraic* structure. That is, in such cases, we can let $op_{substr}(A) = (A)^*$ where $(A)^*$ is the substructure of X generated by A. Examples of such miops include the following:

(a) if X is a group, we can let $op_{subgrp}(A) = (A)^*$ where $(A)^*$ is the subgroup of X generated by A,

(b) if X is a ring, we can let $op_{subrg}(A) = (A)^*$ where $(A)^*$ is the subring of X generated by A,

(c) if X is a field, we can let $op_{subfld}(A) = (A)^*$ where $(A)^*$ is the subfield of X generated by A,

(d) if X is a Boolean algebra, we can let $op_{subalg}(A) = (A)^*$ where $(A)^*$ is the subalgebra of X generated by A or we can let $op_{ideal}(A) = Id(A)$ where $Id(A)$ is the ideal of X generated by A, and

(e) if $(X, \leq_X)$ is a partially ordered set, we can let $op_{uideal}(A) = Uid(A)$ where $Uid(A)$ is the upper order ideal of X, that is, the least subset S of X containing A such that whenever $x \in S$ and $x \leq_X y$, then $y \in S$.

For simplicity, for the rest of this section, we shall assume that all our miops O have the additional property that $Y \subseteq O(Y)$ for all $Y \in 2^X$. Now suppose that we are given a miop $op^+ : 2^X \rightarrow 2^X$ and Horn set-based logic program P over X. Blair, Marek, and Remmel [BMR08] generalized the one-step provability operator to set-based logic programs relative to a miop operator op^+ as follows. First, for any atom A and $I \subseteq X$, we say that $I \models_{[\![\cdot]\!],op^+} A$ if and only if $op^+([\![A]\!]) \subseteq I$. Then, given a set-based logic program P, let P' be the set of ground instances of a clauses in P and let

$$T_{P,op^+}(I) = op^+(I_1)$$

where $I_1 = \bigcup\{[\![A]\!] : A \leftarrow A_1, \ldots, A_n \ \in P' \ \& \ I \models_{[\![\cdot]\!],op^+} A_i, i = 1, \ldots, n\}$. We then say that a *supported model relative to* op^+ of P is a fixpoint of T_{P,op^+}.

We iterate T_{P,op^+} according to the following.

$$\begin{array}{rcl} T^0_{P,op^+}(I) & = & I \\ T^{\alpha+1}_{P,op^+}(I) & = & T_{P,op^+}(T^{\alpha}_{P,op^+}(I)) \\ T^{\lambda}_{P,op^+}(I) & = & op^+(\bigcup_{\alpha<\lambda}\{T^{\alpha}_{P,op^+}(I)\}), \ \lambda \text{ limit} \end{array}$$

It is easy to see that if P is a set-based Horn program and op^+ is a miop, then T_{P,op^+} is monotonic. Blair, Marek, and Remmel [BMR08] proved the following.

Theorem 4.1. Given a miop op^+, the least model of a Horn set-based logic program P exists and is closed under op^+ , is supported relative op^+, and is given by $\mathbf{T}^{\alpha}_{P,op^+}(\emptyset)$ for the least ordinal α at which a fixpoint is obtained.

We note, however, that if the underlying universe X universe of a set-based logic program is infinite, then, unlike the situation with ordinary Horn programs, T_{P,op^+} will not in general be upward continuous even in the case where $op^+(A) = A$ for all $A \subseteq X$. That is, consider the following example which was given in [BMR08].

Example 4.1. Assume that op^+ is the identity operator on 2^X. Let $\mathcal{L} = (L, X, [\![\cdot]\!])$ where L has four unary predicate symbols: p, q, r and s, and countably many constants $e_0, e_1, \ldots,$. X is the set $\mathbb{N} \bigcup \{\mathbb{N}\}$. $[\![\cdot]\!]$ is specified by $[\![q(e_n)]\!] = \{0, \ldots, n\}$, $[\![p(e_n)]\!] = \{0, \ldots, n+1\}$, $[\![r(e_n)]\!] = \mathbb{N}$, and $[\![s(e_n)]\!] = \{\mathbb{N}\}$.

The set-based program P consists of the following three clauses:

$q(e_0) \leftarrow$
$p(X) \leftarrow q(X)$ and
$s(e_0) \leftarrow r(e_0)$.

It is then easy to see that after ω iterations of the T_P operator starting from the empty set, $r(e_0)$ becomes satisfied. One more iteration is required to reach an interpretation that satisfies $s(e_0)$ which is the least fixpoint of T_P.

Next, we consider how we should deal with negation in the setting of miop operators. Suppose that we have a miop operator op^- on the space X. We do not require that op^- is the same as the miop op^+, but it may be. Our goal is to define two different satisfaction relations for negative literals relative to the miop operator op^- which are called strong and weak negation in [BMR08] [1].

Definition 4.1. Suppose that P is a set-based logic program over X and op^+ and op^- are miops on X and $a \in \{s, w\}$.

(I) Given any atom A and set $J \subseteq X$, we say
$J \models^a_{[\![\cdot]\!], op^+, op^-} A$ if and only if $op^+([\![A]\!]) \subseteq J$.

$(\mathrm{II})_s$ (Strong negation) Given any atom A and set $J \subseteq X$, we say
$J \models^s_{[\![\cdot]\!], op^+, op^-}$ *not* A if and only if $op^-([\![A]\!]) \cap J \subseteq op^-(\emptyset)$.

$(\mathrm{II})_w$ (Weak negation) Given any atom A and set $J \subseteq X$, we say
$J \models^w_{[\![\cdot]\!], op^+, op^-}$ *not* A if and only if $op^-([\![A]\!]) \not\subseteq J$.

This given, we can naturally define two analogues of the Gelfond-Lifschitz transform and two analogues of stable models depending on whether we want to use strong or weak negation to definition the satisfaction of *not* A.

Definition 4.2. Given a set $J \subseteq X$, we define the *strong Gelfond-Lifschitz transform*, $GL^s_{J,[\![\cdot]\!],op^+,op^-}(P)$, of a program P with respect to miops op^+ and op^- on 2^X, in two steps. First, we consider all clauses in P,

$$\mathcal{C} = A \leftarrow B_1, \ldots, B_n, \mathit{not}\, C_1, \ldots, \mathit{not}\, C_m \tag{14}$$

where $A, B_1, \ldots, B_n, C_1, \ldots, C_m$ are atoms. If for some i, it is *not* the case that $J \models^s_{[\![\cdot]\!], op^+, op^-}$ *not* C_i, then we eliminate clause $\mathcal{C}$. Otherwise we replace $\mathcal{C}$ by the Horn clause

$$A \leftarrow B_1, \ldots, B_n. \tag{15}$$

[1] Lifschitz [Lif94] observed that different modalities, thus different operators, can be used to evaluate positive and negative part of bodies of clauses of normal programs.

Then, $GL^s_{J,[\![\cdot]\!],op^+,\mathcal{R}}(P)$ consists of the set of all Horn clauses produced by this two step process.

We define the *weak Gelfond-Lifschitz transform*, $GL^w_{J,[\![\cdot]\!],op^+,op^-}(P)$, of a program P with respect to miops op^+ and op^- on 2^X in a similar manner except that we use $\models^w_{[\![\cdot]\!],op^+,op^-}$ in place of $\models^s_{[\![\cdot]\!],op^+,op^-}$ in the definition.

Notice that since $GL^a_{J,[\![\cdot]\!],op^+,op^-}(P)$ is a Horn set-based logic program for either $a = s$ or $a = w$, the least model of $GL^a_{J,[\![\cdot]\!],op^+,op^-}(P)$ relative to op^+ is defined. We then define the a-stable model semantics for a set-based logic program P over X relative to the miops op^+ and op^- on X for $a \in \{s, w\}$ as follows.

Definition 4.3. J is an a*-stable* model of P *relative* to op^+ and op^- if and only if J is the least fixpoint of $T_{GL^a_{J,[\![\cdot]\!],op^+,op^-}(P),op^+}$.

Next we give a simple example to show that there is a difference between s-stable and w-stable models.

Example 4.2. Suppose that the space $X = \mathbb{R}^2$ is the real plane. Our program will have two atoms $\{a, b\}, \{c, d\}$ where a, b, c and d are reals. We let $[a, b]$ and $[c, d]$ denote the line segments connecting a to b and c to d respectively. We let the sense of the these atoms be the corresponding subsets, i.e. we let $[\![\{a, b\}]\!] = \{a, b\}$ and $[\![\{c, d\}]\!] = \{c, d\}$. We let $op^+ = op^- = op_{convex}$. The consider the following program $\mathcal{P}$.

(1) $\{a, b\} \leftarrow$ *not* $\{c, d\}$

(2) $\{c, d\} \leftarrow$ *not* $\{a, b\}$

There are four possible candidates for stable models in this case, namely (i) $\emptyset$, (ii) $[a, b]$, (iii) $[c, d]$, and (iv) $op_{convex}\{a, b, c, d\}$. Let us recall that $op_{convex}(X)$ is the convex closure of X which, depending on $a, b, c,$ and d may be either a quadrilateral, triangle, or a line segment.

If we are considering s-stable models where $J \models^s_{[\![\cdot]\!],op^+,op^-}$ *not* C if and only if $op^-(C) \cap J = op^-(\emptyset) = \emptyset$, then the only case where there are s-stable models if $[a, b]$ and $[c, d]$ are disjoint in which (ii) case and (iii) are s-stable models.

If we are considering w-stable models where $J \models^w_{[\![\cdot]\!],op^+,op^-}$ *not* C if and only if $op^-(C) \not\subseteq J$, then there are no w-stable models if $[a, b] = [c, d]$, (ii) is a w-stable model if $[a, b] \not\subseteq [c, d]$, (iii) is w-stable model if $[c, d] \not\subseteq [a, b]$ and (ii) and (iii) are w-stable models if neither $[a, b] \subseteq [c, d]$ nor $[c, d] \subseteq [a, b]$. $\Box$

It is still the case that the a-stable models of a set-based logic program P form an antichain for $a \in \{s, w\}$. That is, Blair, Marek, and Remmel [BMR08] proved the following result.

Theorem 4.2. *Suppose that P is a set-based logic program over X, op^+ and op^- are miops on X, and $a \in \{s, w\}$. If M and N are a-stable models of P and $M \subseteq N$, then $M = N$.*

We end this section by considering the question of what conditions are required if one is to effectively process a finite set-based logic program where the sense of the underlying atoms are allowed be infinite sets. This question was considered by Marek and Remmel [MR09, MR11]. The idea of Marek and Remmel was to start with a finite set-based logic program P and let $\mathcal{S}_P$ denote set of fixpoints over all finite unions of sets represented by the atoms of a finite set-based logic program P of the miops associated with P. Here the elements of $\mathcal{S}_P$ may be finite or infinite. Marek and Remmel [MR11] showed that if there is a way of associating codes $c(A)$ to the elements of $A \in \mathcal{S}_P$ such that there are effective procedures which, given codes $c(A)$ and $c(B)$ for elements of $A, B \in \mathcal{S}_P$, will

(i) decide if $A \subseteq B$,

(ii) decide if $A \cap B = \emptyset$, and

(iii) produce of the codes of closures of $A \cup B$ and $A \cap B$ under miop operators associated with P,

then we can effectively decide whether a code $c(A)$ is the code of a stable model of P.

There are several examples where conditions (i), (ii), and (iii) can be realized.
(1) Let $X = \mathbb{N}$ and assume that the atoms are codes for finite sets. For instance, we can let the code of the finite set $\{x_1, \ldots, x_n\}$ be $\sum_i 2^{x_i}$ and the code of the empty set be 0. If the miops are just the identity operators, then clearly conditions (i)-(iii) are satisfied. Thus the scheme proposed above can be realized for programs using such codes.

(2) Another example consists of the finite dimensional subspaces of the space $\mathbb{Q}^n$. Such subspace can be coded by any of its bases. The miop in this case is the subspace generated by a given set of vectors. Clearly, given two bases B_1 and B_2 for subspaces S_1 and S_2 of $\mathbb{Q}^n$, respectively, we can generate effectively from B_1 and B_2 a basis for the least space containing the union of S_1 and S_2. We can test if one space is included in another, and see if there is any vector different from the 0-vector in their intersection. Thus again, we can reason about such spaces with normal logic programs and form weak and strong answer sets.

(3) The third example is one where one naturally wants to use non-trivial miops. Namely, the space is $\mathbb{Q}^2$ and the collection $\mathcal{X}$ consists of convex polygons in $\mathbb{Q}^2$ determined by lines with rational slopes. The codes are sets of the extreme points of polygons. The miop is cl_{convex}. There are effective procedures for computation of the code of the closure of the union of two polygons, as well as for testing inclusion and disjointness. Thus, we can reason about such polygons, and compute weak and strong answer sets for programs with atoms being codes for convex polygons.

(4) The fourth example of a situation where we can reason about infinite sets that are regular languages. Here, the codes are the regular expressions for the language or a DFA which accepts the language. The sense function assigns to the code the regular set it describes.

We shall expand on example 4 to illustrate how conditions (i), (ii), and (iii) can naturally be satisfied. It is well known that given two deterministic finite automata

(DFA) A_1 and A_2 one can effectively decide whether the languages $L(A_1)$ and $L(A_2)$ accepted by A_1 and A_2, respectively, satisfy $A_1 \subseteq A_2$, $A_1 = A_2$, or $A_1 \cap A_2 = \emptyset$. Similarly, one can effectively construct DFA's which accept $L(A_1) \cup L(A_2)$, $L(A_1) \cap L(A_2)$, and $\Sigma^* - L(A_1)$.

We say that a miop $op : 2^{\Sigma^*} \rightarrow 2^{\Sigma^*}$ is *effectively automata-preserving* if for any DFA M whose underlying alphabet of symbols is Σ, we can effectively construct a DFA N whose underlying alphabet of symbols is Σ such that $L(N) = op(L(M))$. For example, suppose that $\Sigma = \{0, 1, \ldots, m\}$. Then, the following are effectively automata-preserving operators.

1. If N is a DFA whose underlying set of symbols is Σ, then we can define $op : 2^{\Sigma^*} \rightarrow 2^{\Sigma^*}$ by setting $op(S) = S \cup L(N)$ for any $S \subseteq \Sigma^*$. Clearly if $S = L(M)$ for some DFA M whose underlying set of symbols is Σ, then $op(L(M)) = L(M \cup N)$ so op is effectively automata-preserving.

2. If N is a DFA whose underlying set of symbols is Σ, then we can define $op : 2^{\Sigma^*} \rightarrow 2^{\Sigma^*}$ by setting $op(S) = S \cap L(N)$ for any $S \subseteq \Sigma^*$. Clearly if $S = L(M)$ for some DFA M whose underlying set of symbols is Σ, then $op(L(M)) = L(M \cap N)$ so op is effectively automata-preserving.

3. If T is any subset of Σ, we can let $op(S) = S(T^*)$. Again op will be an effectively automata-preserving miop since if M is DFA whose underlying set of symbols is Σ, then let N be NFA constructed from M by adding loops on all the accepting states labeled with letters from T. It is easy to see that N accepts $L(M)T^*$ and then one can use the standard construction to find a DFA N' such that $L(N') = L(N)$. Notice that in the special case where T equals Σ, we can think of op as constructing the upper ideal of S in Σ^* relative to the partial order $\sqsubseteq$ where for any words $u, v \in \Sigma^*$, $u \sqsubseteq v$ if and only if u is prefix of v, i.e. v is of the form uw for some $w \in \Sigma^*$. For any poset $(P, \leq_P)$, we say that a set $U \subseteq P$ is an *upper ideal* in P, if whenever $x \leq_P y$ and $x \in P$, then $y \in P$. Clearly, for the poset $(\Sigma^*, \sqsubseteq)$, $op(S)$ is the upper ideal of $(\Sigma^*, \sqsubseteq)$ generated by S.

4. Let $\mathcal{P} = (\Sigma, \leq)$ be a partially-ordered set. For any $w, w' \in \Sigma^*$, we say that w' is a factor of w if there are words $u, v \in \Sigma^*$ with $w = uw'v$. Define the *generalized factor order* on P^* by letting $u \leq w$ if there is a factor w' of w having the same length as u such that $u \leq w'$, where the comparison of u and w' is done componentwise using the partial order in $\mathcal{P}$. Again we can show that if $op(S)$ is the upper ideal generated by S the generalized factor order relative to P^*, then op is an effectively automata-preserving miop. That is, if we start with a DFA $M = (Q, \Sigma, \delta, s, F)$, then we can modify M to an NFA that accepts $op(L(M))$ as follows. Think of M as a digraph with edges labeled by elements of Σ in the usual manner. First, we add a new start state s_0. There are loops from s_0 labeled with all letters in Σ. There is also a λ-transition from s_0 to the old start state s. We then modify the transitions in M so that if there is an edge from state q to q' labeled with symbol r, then we add an edge from q to q' with any symbol s such that $r \leq s$. Finally we add loops to all accepting states such that labeled with all letters in in Σ.

Then Marek and Remmel [MR11] proved the following theorem.

Theorem 4.3. *Suppose that P is a finite set-based logic program over $\mathcal{L} = (L, X, [\![\cdot]\!])$ where $X = \Sigma^*$ for some finite alphabet Σ and $op^+ : 2^{\Sigma^*} \to 2^{\Sigma^*}$ and $op^- : 2^{\Sigma^*} \to 2^{\Sigma^*}$ are effectively automata-preserving miops. Moreover, assume that for any atom A which appears in Q, $[\![A]\!]$ is a language accepted by a DFA whose underlying set of symbols is Σ. Then:*

1. *Every weak (strong) stable model of P is a language accepted by a DFA.*
2. *For any DFA M whose underlying set of symbols is Σ, we can effectively decide whether $L(M)$ is a weak or strong stable model of P.*

5 Hybrid ASP

In this section, we shall give the definition of Hybrid ASP programs and stable models as defined by Brik and Remmel [BR11b].

The goal of Hybrid ASP is to allow the user to reason about dynamical systems that exhibit both discrete and continuous aspects. The unique feature of Hybrid ASP is that Hybrid ASP rules can be thought of as general input-output devices. In particular, Hybrid ASP programs allow the user to include ASP type rules that act as controls for when to apply a given algorithm to advance the system to the next position.

Modern computational models and simulations such as the model of dog's heart described in [KNGBOM07] rely on existing PDE solvers and ODE solvers to determine the values of parameters. Such simulations proceed by invoking appropriate algorithms to advance a system to the next state, which is often distanced by a short time interval into the future from the current state. In this way, a simulation of continuously changing parameters is achieved, although the simulation itself is a discrete system. The parameter passing mechanisms and the logic for making decisions regarding what algorithms to invoke and when are part of the ad-hoc control algorithm. Thus the laws of a system are implicit in the ad-hoc control software.

On the other hand, action languages [GL98] which are also used to model dynamical systems allow the users to describe the laws of a system explicitly. Initially action languages did not allow simulation of the continuously changing parameters, which severely limited applicability of such languages. Recently, Chintabathina introduced an action language H [Ch10] where he proposed an elegant approach to modeling continuously changing parameters. That is, a program in H describes a state transition diagram of a system where each state models a time interval in which the parameter dynamics is a known function of time. However, the implementation of H discussed in [Ch10] cannot use PDE solvers nor ODE solvers.

Hybrid ASP is an extension of ASP that allows users to combine the strength of the ad-hoc approaches, i.e. the use of numerical methods to faithfully simulate physical processes, and the expressive power of ASP which provides the ability to elegantly model laws of a system. Hybrid ASP provides mechanisms to express the laws of the modeled system via hybrid ASP rules which can control execution of algorithms relevant for simulation.

We should note that any given dynamical system may have a single trajectory or have multiple trajectories if the system is non-deterministic. For example, in our James Bond model given in the introduction, our agent may have two possible trajectories which would get him to his desired destination as pictured in Figure 2.

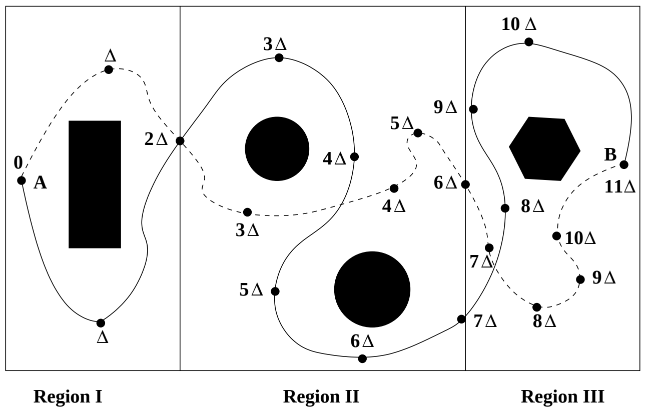

Figure 2: Multiple trajectories.

We shall start out this section by describing a simplified version of Hybrid ASP program which Brik and Remmel called Basic Hybrid ASP (BH-ASP) programs.

5.1 Basic Hybrid ASP

A BH-ASP program P will have an underling parameter space S. For example, in our secret agent example, imagine that we allow James Bond to make decision every Δ seconds where $\Delta > 0$. Then one can think of describing the position and situation at time $k\Delta$ by a sequence of parameters

$$\mathbf{x}(k\Delta) = (x_0(k\Delta), x_1(k\Delta), x_2(k\Delta), \ldots, x_m(k\Delta))$$

that specify both continuous parameters such as time, position, velocity, and acceleration as well as discrete parameters such as is the car configured as a car or as a boat. In a BH-ASP program, we shall always think of the parameter x_0 as specifying time and the range of x_0 is $\{k\Delta : k = 0, \ldots, n\}$ for some fixed n or of the form $\{k\Delta : k \in \mathbb{N}\}$. In particular, for finite BH-ASP programs, we shall assume that the range of x_0 is $\{k\Delta : k = 0, \ldots, n\}$ for some fixed n and $\Delta > 0$. Thus we shall always write an element of S in the form $\mathbf{x} = (k\Delta, x_1(k\Delta), \ldots, x_m(k\Delta))$ for some k. We refer to the elements of S as *generalized positions*. A BH-ASP program will also have an underlying set of atoms At. Then the underlying universe of the program will be $At \times S$.

Suppose that $M \subseteq At \times S$. Then we let $\widehat{M} = \{\mathbf{x} : (\exists a \in At)((a, \mathbf{x}) \in M)\}$. We will say that M satisfies $(a, \mathbf{x}) \in At \times S$, written $M \models (a, \mathbf{x})$, if $(a, \mathbf{x}) \in M$. For any element $(k\Delta, x_1, \ldots, x_m) \in S$, we let $W_M((k\Delta, x_1, \ldots, x_m)) = \{a \in At : (a, (k\Delta, x_1, \ldots, x_m)) \in M\}$ and we shall refer to $W_M(k\Delta, x_1, \ldots, x_m)$ as the *world* *of M at the generalized position* $(k\Delta, x_1, \ldots, x_m)$. We say that M is a **single trajectory model** if for each $k \in \{0, \ldots, n\}$, there is exactly one generalized position of the form $(k\Delta, x_1, \ldots, x_m)$ in $\widehat{M}$. If M is a single trajectory model, then we let $(k\Delta, x_1(k\Delta), \ldots, x_m(k\Delta))$ be the unique element of the form $(k\Delta, x_1, \ldots, x_m)$ in $\widehat{M}$ and we can write M as a disjoint union

$$M = \bigsqcup_{k=0}^{n} W_M(k\Delta, x_1(k\Delta), \ldots, x_m(k\Delta)) \times \{(k\Delta, x_1(k\Delta), \ldots, x_m(k\Delta))\}.$$

We will say that M is a **multiple trajectory model** if for each $k \in \{0, \ldots, n\}$, there is at least one generalized positions of the form $(k\Delta, x_1, \ldots, x_m)$ in $\widehat{M}$ and for some $0 \leq k_0 \leq n$, there are at least two generalized position of the form $(k\Delta, x_1, \ldots, x_m)$ in $\widehat{M}$. The reason for introducing multiple trajectory models is that we may want to reason about all possible trajectories of our secret agent rather than just reasoning about a single trajectory. If we drop the requirement that for each $k\Delta$, there is a generalized position $(k\Delta, x_1, \ldots, x_m) \in \widehat{M}$ in the definition of single trajectory or multiple trajectory models, we get what we call *partial single trajectory* and *partial multiple trajectory* models.

BH-ASP programs consist of collections of the following two types of clauses.

Stationary clauses are of the form

$$a \leftarrow a_1, ..., a_s, \textit{not}\ b_1, ..., \textit{not}\ b_t : O \tag{16}$$

where $a, a_1, ..., a_n, b_1, ..., b_m \in At$, O is a set of generalized positions in the parameter space S. The idea is that if for a generalized position $\mathbf{p} \in O$, if $(a_i, \mathbf{p})$ holds for $i = 1, ..., s$ and $(b_j, \mathbf{p})$ does not hold for $j = 1, ..., t$, then $(a, \mathbf{p})$ holds. Thus stationary clauses are typical normal logic programming clauses relative to a fixed world $W_M(\mathbf{p})$.

Advancing clauses are of the form

$$a \leftarrow a_1, ..., a_s, \textit{not}\ b_1, ..., \textit{not}\ b_t : A, O \tag{17}$$

where $a, a_1, ..., a_n, b_1, ..., b_m \in At$, O is a set of generalized positions in the parameter space S and A is an algorithm such that for any generalized position $\mathbf{p} \in O$, $A(\mathbf{p})$ is defined and is an element of S. Here A can be any sort of algorithm which might be the result of solving a differential or integral equation, solving a set of linear equations or linear programming equations, running a program or automaton, etc. The idea is that if for a generalized position $\mathbf{p} \in O$, if $(a_i, \mathbf{p})$ holds for $i = 1, ..., s$ and $(b_j, \mathbf{p})$ does not hold for $j = 1, ..., t$, then $(a, A(\mathbf{p}))$ holds. We will require that for all $\mathbf{p} \in O$, $A(\mathbf{p})$ always produces the same output. In a BH-ASP program, we will always assume that if $\mathbf{p} = (k\Delta, x_1, \ldots, x_m)$, then $A(\mathbf{p})$ is of the form $((k+1)\Delta, y_1, \ldots, y_m)$ for some $y_1, \ldots, y_m$. Thus advancing clauses are like input-output devices in that the algorithm A allows certain elements a which are to hold at the next generalized position.

In both advancing clauses and stationary clauses, we shall refer to the set O as the *constraint set* of the clause. The idea here is that O allows one to use a single clause to specify clauses that can be used at a variety of generalized positions. We shall refer to the algorithm A in advancing clause as the *advancing algorithm* of the clause.

A *BH-ASP Horn program* H is a collection of clauses of the form (16) and (17) such that there are no occurrences of *not* in any of its clauses. A *consistent BH-ASP Horn program* G is a BH-ASP Horn program such that if (A, O) and (A', O') appear in H, then $A \restriction_{O \cap O'} = A' \restriction_{O \cap O'}$, where for an algorithm B and set of positions K, $B \restriction_K$ is the restriction of the algorithm to the domain K.

Next we introduce the one-step provability operator for BH-ASP Horn programs. Let I be an initial generalized position in S, M be a subset of $At \times S$, and P be a basic hybrid Horn program. Then we define $T_{P,I}(M)$ to be the union of M and the set of all atoms $(a, \mathbf{p})$ such that either

1. there exists $C = \ a \leftarrow a_1, ..., a_n : O \in P$ and $\mathbf{p} \in (\widehat{M} \cup \{I\}) \cap O$ such that $(a_i, \mathbf{p}) \in M$ for $i = 1, ..., n$ or

2. there exists $C = a \ \leftarrow a_1, ..., a_n \ : A, O \in P$ and $\mathbf{q} \in (\widehat{M} \cup \{I\}) \cap O$ such that $(a_i, \mathbf{q}) \in M$ for $i = 1, ..., n$ and $\mathbf{p} = A\,(\mathbf{q})$.

It is easy to see that $T_{P,I}$ is a continuous monotone operator so that the least fixpoint of $T_{P,I}$ is

$$T_{P,I}(\emptyset)^{\omega} = \bigcup_{n \geq 0} T^n_{P,I}(\emptyset).$$

A *BH-ASP program* is a collection of clauses of the form (16) and (17). We shall define two types of stable models: partial multiple trajectory stable models and partial single trajectory stable models. Let P be a BH-ASP program over the parameter set S and sets of atoms At. Suppose that $I = (0, x_1, \ldots, x_m) \in S$ is an initial condition and $M \subseteq At \times S$ is a partial multiple trajectory model. Then the *Gelfond-Lifschitz reduct of P with respect to M and I* is denoted by $P^{M,I}$ and is defined by the following procedure.

1. Eliminate from P all clauses $C = a \leftarrow a_1, ..., a_n,$ *not* $b_1, ...,$ *not* $b_m \, : A, O$ such that $(\forall \mathbf{p} \in \left(\widehat{M} \cup \{I\}\right) \cap O)((\exists i)((b_i, \mathbf{p}) \in M)$ or $A\,(\mathbf{p}) \notin \widehat{M})$.

2. If a clause C is not eliminated in step (1), then replace it by the clause $a \leftarrow a_1, ..., a_n \, : A, O'$ where
O' equals the set of all $\mathbf{p}$ such that $\mathbf{p} \in \left(\widehat{M} \cup \{I\}\right) \cap O$ and $(b_i, \mathbf{p}) \notin M$ for $i = 1, \ldots, m$ and $A\,(\mathbf{p}) \in \widehat{M}$.

3. Eliminate from P all clauses $C = \ a \leftarrow a_1, ..., a_n,$ *not* $b_1, ...,$ *not* $b_m \, : O$ such that $(\forall \mathbf{p} \in \left(\widehat{M} \cup \{I\}\right) \cap O)(\exists i)((b_i, \mathbf{p}) \in M)$.

4. If a clause C in (3) is not eliminated, then replace it by the clause

$$a \leftarrow a_1, ..., a_n : O'$$

where $O' = \left\{\mathbf{p} | \ \mathbf{p} \in \left(\widehat{M} \cup \{I\}\right) \cap O \text{ and } \forall i = 1, ..., m \ ((b_i, \mathbf{p}) \notin M)\right\}$.

Clearly, $P^{M,I}$ is always a BH-ASP Horn program. If M is a partial single trajectory model, then it is easy to see that it must be the case that $P^{M,I}$ is a consistent BH-ASP Horn program. Then we say that M *is a stable model of* P *with initial condition* I if $T_{P^{M,I},I}(\emptyset)^{\omega} = M$. If, in addition, M is a partial single trajectory model so that $P^{M,I}$ is a consistent hybrid Horn program, then we say that M is a *partial single trajectory stable model of* P *with initial condition* I.

5.2 Hybrid ASP programs.

There are several features which are not available in BH-ASP programs that one would like to have in an ASP system that can reason about dynamic systems exhibiting a mixture of continuous and discrete phenomena. For example, here is a partial list of features that one would like to have.

1. The restriction that in BH-ASP programs, advancing clauses always have conclusions that represent information that occur at a fixed time interval Δ later than the current time is inconvenient. For example, in our James Bond example, the dynamics changes as we go from the mountain to the lake and as we go from the lake to the desert, but the time of such transitions may not be a multiple of of Δ.

2. It is often useful to specify that certain invariance properties hold over a set of times or generalized positions so that it may be desirable to have clauses whose hypothesis refer to two or more different generalized positions.

3. In a BH-ASP program, every algorithm is required to produce the values for all the parameters in a generalized position. As a number of parameters grows such a requirement could become a serious drawback. In many cases, it would be more convenient if an algorithm was allowed to specify values of only some of the parameters, letting other parameters be ”unspecified” and possibly allow unspecified values to be assigned by algorithms associated with other clauses.

4. There is the issue of how to deal with imprecise computations. That is, if our algorithm is to solve a partial differential equation numerically, we may not be able to get exact answers but only produce an answer that lies distance ϵ from the exact answer. Similarly, we may want to use randomized algorithms. For this reason, we might want to allow our algorithms to be set-valued rather than specify functions.

To deal with such issues, Brik and Remmel [BR11b] introduced an extension of BH-ASP which they called Hybrid ASP (H-ASP for short). As before, we start with a parameter space S consisting of tuples of parameters $\mathbf{p} = (v_t, v_1, ..., v_k)$ and a set of atoms At. If $\mathbf{p} = (v_t, v_1, ..., v_k) \in S$, we will assume that v_t is always the time parameter and we let $t(\mathbf{p})$ denote v_t and $v_i(\mathbf{p})$ denote v_i for $i = 1, \dots, k$. The universe U of a hybrid ASP program will equal $At \times S$.

Given $M \subseteq At \times S$, $B = a_1, \dots, a_n,$ *not* $b_1, ...,$ *not* b_m, and $\mathbf{p} \in S$, we say that M satisfies B at the generalized position $\mathbf{p}$, written $M \models (B_i, \mathbf{p})$, if $(a_i, \mathbf{p}) \in M$ for $i = 1, \dots, n$, and $(b_j, \mathbf{p}) \notin M$ for $j = 1, \dots, m$. Notice that if B_i is empty then $M \models (B_i, \mathbf{p})$ holds. We let $posBody(B_i) = a_1, \ldots, a_n$ and $negBody(B_i) =$ *not* $b_1, ...,$ *not* b_m. Let $\widehat{M} = \{\mathbf{x} : (\exists a \in At)((a, \mathbf{x}) \in M)\}$.

There are two types of clauses in H-ASP programs.

Extended stationary clauses are of the form

$$a \leftarrow B_1, B_2, \ldots, B_r : H, O \tag{18}$$

where each B_i is of the form $a_1^{(i)}, \ldots, a_{n_i}^{(i)}$, *not* $b_1^{(i)}, ...,$ *not* $b_{m_i}^{(i)}$ where $a_1^{(i)}, \ldots a_{n_i}^{(i)}$, $b_1^{(i)}, ..., b_{m_i}^{(i)}$ are atoms, a is an atom, $O \subseteq S^r$ is such that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \cdots < t(\mathbf{p}_r)$, and H is a Boolean valued algorithm. Here and in subsequent clauses, we allow n_i or m_i to be equal to 0 for any given i. Moreover, if $n_i = m_i = 0$, then B_i is empty and we automatically assume that B_i is satisfied by any $M \subseteq At \times S$.

The idea is that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$ and for each i, B_i is satisfied at the general position $\mathbf{p}_i$, and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ is true, then $(a, \mathbf{q})$ holds. Thus extended stationary clauses in H-ASP are similar to stationary clauses in BH-ASP except that we allow the clause to refer to generalized positions that occur at multiple times up to and including the time $t(\mathbf{p}_r)$ where we require the pair $(a, \mathbf{p}_r)$ to hold, and we allow a user to specify an additional constraint on the tuples of positions via an algorithm H. For example, H could involve such non-logical conditions as that the generalized position satisfies some system of linear equations or that there exist clauses in the program that could be used to advance position $\mathbf{p}_i$ to position $\mathbf{p}_{i+1}$ for all $i = 1, \ldots, r-1$. We shall refer to O as the *constraint set* of the clause and the algorithm H as the *Boolean algorithm* of the clause.

Extended advancing clauses are of the form

$$a \leftarrow B_1, B_2, \ldots, B_r : A, O \tag{19}$$

where A is an algorithm and each B_i is of the form $a_1^{(i)}, \ldots, a_{n_i}^{(i)}$, *not* $b_1^{(i)}, ...,$ *not* $b_{m_i}^{(i)}$ where $a_1^{(i)}, \ldots a_{n_i}^{(i)}$, $b_1^{(i)}, ..., b_{m_i}^{(i)}$ are atoms, a is atom, and $O \subseteq S^r$ is such that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, then $t(\mathbf{p}_1) < \ldots < t(\mathbf{p}_r)$ and for all $\mathbf{q} \in A(\mathbf{p}_1, \ldots, \mathbf{p}_r)$, $t(\mathbf{q}) > t(\mathbf{p}_r)$.

The idea is that if $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$ and for each i, B_i is satisfied at the general position $\mathbf{p}_i$, then the algorithm A can be applied to $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ to produce a set of generalized positions O' such that if $\mathbf{q} \in O'$, then $t(\mathbf{q}) > t(\mathbf{p}_r)$ and $(a, \mathbf{q})$ holds. Thus advancing clauses in H-ASP are similar to advancing clauses in BH-ASP except that we allow a clause to refer to generalized positions that occur at multiple times up to and including the time $t(\mathbf{p}_r)$ and our algorithm A is set-valued rather than single valued. As before, we shall refer to O as the *constraint set* of the clause and the algorithm A as the *advancing algorithm* of the clause.

An H-ASP program is a collection of clauses of the form (18) and (19). A *H-ASP Horn program* is a H-ASP program which does not contain any occurrences of *not* . A *consistent H-ASP Horn program P* is an H-ASP program such that if whenever two pairs of an advancing algorithm and a constraint set, (A, O) and (A', O'), appear in P and $O, O' \subseteq S^r$, then $A \restriction_{O \cap O'} = A' \restriction_{O \cap O'}$.

Let P be a H-ASP Horn program and $I \in S$ be an initial condition such that $t(I) = 0$. Then the one-step provability operator $T_{P,I}$ is defined so that given $M \subseteq At \times S$, $T_{P,I}(M)$ consists of M together with the set of all $(a, J) \in At \times S$ such that

1. there exists an extended stationary clause $C = a \leftarrow B_1, B_2, \ldots, B_r : H, O$ and $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 1$ and $(a, J) = (a, \mathbf{p}_r)$ and $M \models (B_i, \mathbf{p}_i)$ for $i = 1, \ldots, r$ or
2. there exists an advancing clause $C = a \leftarrow B_1, B_2, \ldots, B_r : A, O$ and $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $(a, J) \in A(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ and $M \models (B_i, \mathbf{p}_i)$ for $i = 1, \ldots, r$.

It is easy to see that for all H-ASP Horn programs P and initial conditions $I \in S$ such that $t(I) = 0$, $T_{P,I}$ is a continuous monotone operator so that the least model of P relative to the initial condition I is given by $T^{\omega}_{P,I}(\emptyset)$.

We can then define the stable model semantics for general H-ASP programs as follows. Suppose that we are given a hybrid ASP program P, over a set of atoms At and a parameter space S, a set $M \subseteq At \times S$, and an initial condition $I \in S$ such that $t(I) = 0$. Then we form the Gelfond-Lifschitz reduct of P over M and I, $P^{M,I}$ as follows.

1. Eliminate from P all advancing clauses $C = a \leftarrow B_1, \ldots, B_r : A, O$ such that for all $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, there is an i such that $M \not\models (\mathit{negBody}(B_i), \mathbf{p}_i)$ or $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M} = \emptyset$.
2. If the advancing clause $C = a \leftarrow B_1, \ldots, B_r : A, O$ is not eliminated by (1), then replace it by $a \leftarrow B_1^+, \ldots, B_r^+ : A^+, O^+$ where for each i, $B_i^+ = \mathit{posBody}(B_i)$, O^+ is equal to the set of all $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ in $O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $M \models (\mathit{negBody}(B_i), \mathbf{p}_i)$ for $i = 1, \ldots, r$ and $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M} \neq \emptyset$, and A^+ is defined so that the domain of $A+$ is O^+ and $A^+(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ is $A(\mathbf{p}_1, \ldots, \mathbf{p}_r) \cap \widehat{M}$ for all $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O^+$.
3. Eliminate from P all extended stationary clauses $C = \leftarrow B_1, \ldots, B_r : H, O$ such that for all $(\mathbf{p}_1, \ldots, \mathbf{p}_r) \in O$, either there is an i such that $M \not\models (\mathit{negBody}(B_i), \mathbf{p}_i)$ or $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 0$.
4. If the extended stationary clause $C = a \leftarrow B_1, \ldots, B_r : H, O$ is not eliminated by (3), then replace it by $a \leftarrow B_1^+, \ldots, B_r^+ : H, O^+$ where for each i, $B_i^+ = \mathit{posBody}(B_i)$, O^+ is equal to the set of all $(\mathbf{p}_1, \ldots, \mathbf{p}_r)$ in $O \cap \left(\widehat{M} \cup \{I\}\right)^r$ such that $M \models (\mathit{negBody}(B_i), \mathbf{p}_i)$ for $i = 1, \ldots, r$ and $H(\mathbf{p}_1, \ldots, \mathbf{p}_r) = 1$. Let H^+ be the restriction of H to O^+.

We then say that M is a *general stable model of P with initial condition I* if $T_{P^{M,I}}(\emptyset)^{\omega} = M$.

We believe that the point of view of thinking of rules as general input-output devices has the potential for many new applications of ASP techniques. Thus we believe that one should view Brik and Remmel's work on H-ASP programs [BR11b] as a first step for further work that will lead to both theoretical tools used for the modeling and analysis of dynamic systems and for computer applications that simulate dynamical systems. There is considerable work to be done in developing a theory of such programs which would be similar to the theory that has been developed for ASP programs. For example, a careful analysis of the complexity of the stable models of a H-ASP programs as a function to the complexity of the advancing and Boolean algorithms in the program needs to be done. One should explore more extended sets of rules that allows for partial parameter passing or allow different rules to instantiate disjoint sets of parameters for the next time step. We need to develop extensions of ASP solvers that can process Hybrid ASP programs. That is, in action languages like H, the goal is to compile an H program into a variant ASP program that can be processed with current variant ASP solvers. The existence of Hybrid ASP solvers would allow us to develop Hybrid ASP type extensions of action languages like H that could be compiled to Hybrid ASP programs which, in turn, would be processed by Hybrid ASP solvers.

6 Conclusions

While there are several declarative formalisms that deal with finite-domain constraint-satisfaction, two of these, ASP and Satisfiability (SAT) [BHMW09] are *logic-based*. However, the motivations of these two technologies are different. ASP is extensively discussed above and is based on generalizations of Horn logic and knowledge representation. On the other hand, SAT has its roots in the theory of computation and has significant applications in electronic design automation. However, at least up until now, the SAT community did not pay much attention to the issue of constraint representation. The tools available for a programmer to prepare the input clausal theory for the solver as well as tools for decoding the results returned by the solver have traditionally been very limited. By contrast, ASP has its roots in Knowledge Representation as understood by the Artificial Intelligence community. Thus researchers in ASP have been much more sensitive to the issue of proper representation of constraints and providing a bigger repertoire of tools that could support the programmer. Example of such tools include *grounders* that support the use of variables and pseudo-Boolean constraints. The work of the `dlv` designers shows that solvers can also be tightly coupled with traditional database systems.

It is only natural to ask whether there are further steps that can be taken to increase the applicability of ASP and its underlying logic and universal algebra mechanisms. As ASP solvers such as *clasp* [GKNS07] have recently become fast enough to compete with SAT solvers, it is worth to ask whether the knowledge representation tools available to the ASP programmer can be further extended. Put slightly differently, one should ask if the mechanism of context-dependent reasoning as discovered by Gelfond and Lifschitz can be applied to a richer class of applications that go well beyond finite domain constraint satisfaction. As shown in a number of our papers quoted in this paper, the abstract mechanism of fixpoint computation, based on the abstract form of the Knaster-Tarski fixpoint theorem, can be extended to much richer programming environments by properly interpreting the Gelfond-Lifschitz construction of stable models. Conceptually, this is akin to Satisfiability-Modulo-Theories. This observation has been made in a variety of forms by a number of other authors, especially, in [Nie09] and [MGZ08]. We have presented four such extensions in this paper. However, we should note that all of these extensions are compatible in that one can incorporate all the features of these extensions into a single system. In other words, by choosing appropriate libraries for processing various classes of constraints, one could use an abstract Gelfond-Lifschitz mechanism for stable model computation as a single processing paradigm. A step in this direction was made by Lierler who proposed such abstract mechanism in [Lie08]. The next step is to develop efficient solvers for such extensions so that one can extend the range of applications of ASP systems. This is a highly complex task, but one that we think is worth the effort.

We hope our review of these four extensions of ASP will motivate other researchers in ASP to investigate new extensions of ASP. This is a topic that has interested us over the last 15 years and we feel that there is still much more work to be done on the theory of such extensions, the implementations of such extensions, and the applications of such extensions.

Acknowledgements

The authors acknowledge very helpful suggestions of the anonymous referee, as well as discussions with Alex Brik, Howard Blair, and Mirek Truszczyński.

References

[AB90] K. Apt and H.A. Blair. Arithmetic classification of perfect models of stratified programs. *Fundamenta Informaticae*, 13(1):1–17, 1990.

[AvE82] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[BL02] Y. Babovich and V. Lifschitz. *Cmodels package*, 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.

[BG94] M. Bellare and S. Goldwasser. The Complexity of Decision Versus Search. *SIAM Journal of Computing* 23:97–119. 1994.

[Bar03] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[BMR01] H. A. Blair, V. W. Marek, and J. B. Remmel. Spatial logic programming. In *Proceedings of SCI 2001*, 2001.

[BMR08] H. A. Blair, V. W. Marek, and J. B. Remmel. Set-based logic programming. *Annals of Mathematics and Artificial Intelligence*, 52:81–105, 2008.

[BMS95] H.A. Blair, W. Marek, and J. Schlipf. The expressiveness of locally stratified programs. *Annals of Mathematics and Artificial Intelligence*, 15(2):209–229, 1995.

[BHMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds). *Handbook of Satisfiability*, IOS Press, 2009.

[Bon04] P.A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence 156*:75–111, 2004.

[BNT03] G. Brewka, I. Niemelä, and M. Truszczyński. Answer set optimization. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 867–872, 2003.

[BR10] A. Brik and J.B. Remmel, Computing Stable Models of Logic Programs using the Metropolis algorithm, preprint, 2010.

[BR11a] A. Brik and J. B. Remmel, Expressing preferences in ASP using set constraint atoms, in preparation, 2011.

[BR11b] A. Brik and J.B. Remmel, Hybrid ASP, preprint, 2011.

[Ch10] Sandeep Chintabathina, Towards Answer Set Programming Based Architectures for Intelligent Agents, *PhD thesis, Texas Tech University*, 2010.

[CRM05] D. A. Cenzer, J. B. Remmel, and V. W. Marek. Logic programming with infinite sets. *Annals of Mathematics and Artificial Intelligence*, 44:309–339, 2005.

[Cla78] K. Clark. Negation as failure. In *Logic and data bases*, H. Gallaire and J. Minker, Eds. Plenum Press, pages 293–322, 1978.

[DG84] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[ET06] D. East and M. Truszczyński. Predicate-calculus based logics for modeling and solving search problems. *ACM Transactions on Computational Logic*, 7:38–83, 2006.

[EIMT06] D. East, M. Iakhiaev, A. Mikitiuk, and M. Truszczyński. Tools for modeling and solving search problems. *AI Comunications*, 19(4):301-312, 2006.

[FPL11] W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence* 175(1): 278-298, 2011.

[GKNS07] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 386–392, 2007.

[GL02] M. Gelfond and N. Leone. Logic programming and knowledge representation – the A-prolog perspective. *Artificial Intelligence*, 138:3–38, 2002.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the International Joint Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*, W.H. Freeman, 1979.

[GL98] M. Gelfond and V. Lifschitz, Action languages, *Electronic Transactions on AI*, 3(16), 1998.

[KNGBOM07] R. Kerckhoffs, M. Neal, Q. Gu, J.B. Bassingthwaighte, J.H. Omens, A.D. McCulloch. January 2007. Coupling of a 3D Finite Element Model of Cardiac Ventricular Mechanics to Lumped Systems Models of the Systemic and Pulmonic Circulation. *Annals of Biomedical Engineering*, Vol. 35, No. 1, pp. 1-18, 2007.

[LPF06] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[Lib04] L. Libkin. *Elements of Finite Model Theory*. Springer-Verlag, 2004.

[Lif94] V. Lifschitz: Minimal Belief and Negation as Failure. *Artificial Intelligence* 70(1-2): 53-72, 1994.

[Lif99] V. Lifschitz. Action languages, answer sets, and planning. In K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer-Verlag, 1999.

[Lie08] Y. Lierler. Abstract Answer Set Solving. In *International Conference on Logic Programming*, Springer Lecture Notes in Computer Science 5366, pages 377–391, 2008.

[LZ02] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of AAAI 2002*, pages 112–117. AAAI Press, 2002.

[LPST07] L. Liu, E. Pontelli, T.C. Son, and M. Truszczyński. Logic programs with abstract constraint atoms: The role of computations. In *International Conference on Logic Programming*, pages 286–301, 2007.

[LT05a] L. Liu and M. Truszczyński. Properties of programs with monotone and convex constraints. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, pages 701–706. AAAI Press, 2005.

[LT05b] L. Liu and M. Truszczyński. Pbmodels - software to compute stable models by pseudoboolean solvers. In *Proceedings of LPNMR 2005* Springer Lecture Notes in Computer Science 3662, pages 410–415, 2005.

[MNR92] W. Marek, A. Nerode, and J.B. Remmel. How Complicated is the Set of Stable Models of a Logic Program? *Annals of Pure and Applied Logic, 56*:119-136, 1992.

[MNR94a] W. Marek, A. Nerode, and J.B. Remmel. The stable models of predicate logic programs. *Journal of Logic Programming 21*:129-154, 1994.

[MNR94b] W. Marek, A. Nerode, and J.B. Remmel. Context for belief revision: Forward chaining-normal nonmonotonic rule systems, *Annals of Pure and Applied Logic 67*:269-324, 1994.

[MNR95] W. Marek, A. Nerode, and J. B. Remmel. On logical constraints in logic programming. In *Proceedings of LPNMR 1995*, Springer Lecture Notes in Computer Science 928, pages 43–56, 1995.

[MNR97] W. Marek, A. Nerode, and J.B. Remmel. Nonmonotonic rule systems with recursive sets of restraints. *Arch. Math. Logic*, 36 (1997), 339-384.

[MNT08] V.W. Marek, I. Niemelä, and M. Truszczyński. Logic programs with monotone abstract constraint atoms, *Theory and Practice of Logic Programming*, 8:167–199, 2008.

[MR02] V.W. Marek and J.B. Remmel. Logic programs with cardinality constraints. In *Proceedings of the 9th International Workshop on Nonmonotonic Reasoning*, pages 219–228, 2002.

[MR03] V.W. Marek and J.B. Remmel. On the expressibility of stable logic programming. *Theory and Practice of Logic Programming*, 3:551–567, 2003.

[MR04] V.W. Marek and J.B. Remmel. Set constraints in logic programming. In *Proceedings of LPNMR 2004*, Springer Lecture Notes in Computer Science 2923, pages 167–179, 2004.

[MR09] V.W. Marek and J.B. Remmel. Automata and Answer Set Programming. In *Logical Foundations of Computer Science*, 2009. Springer Lecture Notes in Computer Science 5407, pages 323–337, 2009.

[MR11] V.W. Marek and J.B. Remmel, Effectively Reasoning about Infinite Sets in Answer Set Programming. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, to appear, 2011.

[MT89] V.W. Marek and M. Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38:588-619, 1991.

[MT98] V.W. Marek and M. Truszczyński. Revision programming. *Theoretical Computer Science* 190(2):241–277, 1998.

[MT99] V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[MGZ08] V.S. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence* 53:251–287. 2008.

[Met53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, Equation of State Calculations by Fast Computing Machines, *J. Chem. Phys*, 21:1087-1092. 1953.

[Nie99] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[Nie09] I. Niemelä. Integrating Answer Set Programming and Satisfiability Modulo Theories, Proceedings of LPNMR 2009. Springer Lecture Notes in Computer Science 5753, page 3, 2009.

[NS96] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of JICSLP-96*, pages 289–303, MIT Press, 1996.

[PR96] Ch. Pollett and J.B. Remmel. Non-Monotonic Reasoning with Quantified Boolean Constraints. In *Proceedings of LPNMR 1997*, Springer Lecture Notes in Computer Science 1265, pages 18-39, 1997.

[PS93] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1993.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence 138*:181–234, 2002.

[Smu68] R.M. Smullyan. *First-order logic*. Springer New York, Inc., New York, 1968.

[SPT07] T. C. Son, E. Pontelli, P. H. Tu: Answer Sets for Logic Programs with Arbitrary Abstract Constraint Atoms. *Journal of Artificial Intelligence Research* 29:353-389, 2007.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.

Use your QR-barcode reader to get to the e-repository of my papers.