Answer Set Programming with Constraints using Lazy Grounding

A. Dal Palù[1], A. Dovier[2], E. Pontelli[3], and G. Rossi[1]

[1] Dip. Matematica, Univ. Parma,
{alessandro.dalpalu|gianfranco.rossi}@unipr.it
[2] Dip. Matematica e Informatica, Univ. Udine, dovier@dimi.uniud.it
[3] Dept. Computer Science, New Mexico State Univ., epontell@cs.nmsu.edu

Abstract. The paper describes a novel methodology to compute stable models in Answer Set Programming. The proposed approach relies on a bottom-up computation that does not require a preliminary grounding phase. The implementation of the framework can be completely realized within the framework of Constraint Logic Programming over finite domains. The use of a high level language for the implementation and the clean structure of the computation offer an ideal framework for the implementation of extensions of Answer Set Programming. In this work, we demonstrate how non-ground arithmetic constraints can be easily introduced in the computation model. The paper provides preliminary experimental results which confirm the potential for this approach.

1 Introduction

The recent literature has shown a booming interest towards the *Answer Set Programming (ASP)* paradigm [2]. ASP builds on the theoretical foundations of normal logic programs under stable model semantics, and it provides a programming paradigm that elegantly integrates traditional logic programming, non-monotonic reasoning, and some forms of constraint-based reasoning.

The popularity of ASP has been fueled by the realization that ASP offers compact, elegant, and provably correct solutions for problems in a variety of application domains (e.g., phylogenetic inference [4], planning [13], bioinformatics [9]); significant effort has also been invested in the design of knowledge building blocks and methodologies (e.g., [2]). The development of novel applications has also stretched to the limits both the traditional *languages* supported by ASP as well as system implementations, emphasizing some of the limitations of the currently used technology. This has been, for example, highlighted in a recent study concerning the use of ASP to solve complex planning problems (drawn from recent international planning competitions) [23]. A problem like Pipeline (from IPC-5), whose first 9 instances can be effectively solved by state-of-the-art planners like FF, can be solved only in its first instance using Lparse and Smodels; instances 2 through 4 do not terminate within several hours of execution, while instance 5 leads Lparse to generate a ground image that is beyond the input capabilities of Smodels.

We have witnessed a flourishing of new proposals for language extensions (e.g., aggregates, domain-specific constraints, functions), to enable the declarative encoding of complex relationships. In turn, also these extensions have proved challenging for the implementors, often leading to unnecessarily complex machineries to integrate extensions within the rigid framework of existing ASP solvers (e.g., [7, 19]).

The majority of the existing ASP systems rely on a two-stage computation model. The actual computation of the answer set is performed only on propositional programs—either directly (as in SMODELS [21], DLV [12] and CLASP [8]) or appealing to the use of a SAT solver (as in ASSAT [14] and CMODELS [1]). On the other hand, the convenience of ASP vitally builds on the use of *first-order* constructs. This introduces the need of a grounding phase, typically performed by a grounding module (e.g., a separate program, like LPARSE or GRINGO, or an integrated module as in DLV). The presence of grounding represents a significant obstacle to applications and extensions—it has the potential (often observed in practice) of leading to extremely large ground programs and it may force developers to unnatural solutions to circumvent the grounding of certain components of the program (e.g., as observed in some implementations of aggregates [7] and domain-specific constraints [19]).

In this manuscript, we propose a different perspective on this problem, aimed at creating a framework which executes ASP programs *without* preliminary grounding and which enables ease integration of extensions like domain-specific constraints. The proposed framework is called *Grounding-lazy ASP (*GASP*)*. The spirit of our effort can be summarized as follows:

- The framework is completely developed in a declarative language (Constraint Logic Programming over finite domains)—where finite domain sets are employed for the compact representation of predicates in an ASP program.
- The execution model is bottom-up and does not require preliminary grounding of the program.

This combination of ideas provides a novel system with significant potentials:

- It enables the simple integration of new features in the solver, such as domain-specific constraints (e.g., numerical constraints). With a preliminary grounding stage, these features would have to be encoded as ground programs, thus reducing the capability to devise general strategies to optimize the search, and often leading to exponential growth in the size of the ground program.
- The adoption of a non-ground search allows the system to effectively control the search process at a higher level, enabling the adoption of Prolog-level implementations of search strategies and the use of static analysis techniques.
- It reduces the negative impact of grounding the whole program before execution; grounding is lazily applied to the rules being considered during the construction of an answer set, and the ground rules are not kept beyond their needed use.

GASP has been implemented in a prototype, implemented in SICStus Prolog (using the `clpfd` library) and available at `www.dimi.uniud.it/dovier/GASP`.

GASP supports the use of numerical constraints in the ASP programs (providing language capabilities comparable to that of the system presented in [19]). In spite of the overheads introduced by the intermediate Prolog layer, GASP is performance-wise competitive; it is capable of outperforming systems like SMODELS and CLASP especially in benchmarks where the ground image is large.

The ideas presented in this paper expand our preliminary work on comparing ASP and CLP methodologies [6] and development of computation-based characterizations of answer sets [15]. The work is also similar in spirit to the concurrently proposed ASPeRiX system [11]. Both GASP and ASPeRiX have their theoretical roots in the same notion of computation-based characterization of answer sets [15]. ASPeRiX is implemented in C++ and develops heuristics aimed at enhancing the choice of the rules when more of them are applicable. Models for non-ground computation based on alternative execution schemes (e.g., top-down computations) have also been recently proposed (e.g., [3]).

2 The Language GASP

Syntax. The signature $\Sigma = \langle \Pi_C \cup \Pi_U, \mathcal{F}, \mathcal{V} \rangle$ of the language is defined as follows. $\mathcal{V}$ is a denumerable set of variables. $\mathcal{F} = \mathbb{Z} \cup F_Z \cup F_U \cup \{'..'/2\}$ is the set of constant and function symbols of the language, where

- $\mathbb{Z} = \{0, -1, 1, -2, 2, \ldots\}$ is a set of constants for the integer numbers
- F_Z is a set of function symbols representing operations over integer numbers, such as $+, -, *, \mathsf{div}, \mathsf{mod}$, etc.;
- F_U is a (possibly empty) set of user-defined constant symbols, with the property that $F_U \cap (\mathbb{Z} \cup F_Z \cup \{\text{'..'}\}) = \emptyset$.
- '..' is a binary function symbol used to build intervals.

Π_U is the set of user-defined predicate symbols, while Π_C is the set of constraint predicate symbols (we assume that $\Pi_C \cap \Pi_U = \emptyset$).

Each Σ-term of the form $a..b$ is well-formed iff a, b are integer constants and $a \leq b$; we will refer to this type of terms as *interval terms*. Each Σ-term of the form $f(t_1, \ldots, t_k)$, $f \in F_Z$, is well-formed iff $t_1, \ldots, t_k$ are either variables, or integer constants, or (recursively) well-formed compound terms of the same form. Well-formed Σ-terms of the above form are called *compound integer terms*. User-defined function symbols are not allowed in our language.

$\langle \Pi_U, \mathcal{F}, \mathcal{V} \rangle$-atoms are *user-defined atoms*, while $\langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$-atoms are *constraint atoms* (or *primitive constraints*). We assume that interval terms can occur only in user-defined atoms (namely, in rule's head atoms—see below), while compound integer terms can occur only in constraint atoms. Negated literals have the form **not** A, where A is a (positive) Σ-atom.

The set of constraint predicate symbols Π_C of our language includes $=$, $\neq$ and $\leq$. $t_1 = t_2$, $t_1 \neq t_2$ are well-formed iff t_1 and t_2 are Σ-terms, while $t_1 \leq t_2$ is well-formed iff t_1 and t_2 are integer terms (either constants, variables, or compound integer terms). These symbols represent, respectively, the (syntactic) equality and inequality over $\mathbb{Z} \cup F_U$ and the natural order relation over $\mathbb{Z}$.

A GASP-*constraint* is a conjunction of constraint atoms. Other integer predicates (e.g., $<$, $\geq$, and $>$) can be defined as GASP-constraints using $=$, $\neq$, and $\leq$. Examples of well-formed terms and atoms are: `p(1..10)`, `p` $\in \Pi_U$ and `X` $\neq$ `Y + 1`. Hereafter, we will consider only well-formed terms and atoms.

Let us observe that our approach is parametric w.r.t. the constraint domain considered. In the paper, however, we focus on integer constraints.

A GASP-*rule* has the form $H \leftarrow B_1, \ldots, B_k$, where H is a user-defined atom or false, and $B_1, \ldots, B_k$ are either user-defined literals or constraint atoms or true. A GASP-rule of the form $H \leftarrow$ true (abbreviated H) is called a *fact*. A GASP-rule of the form false $\leftarrow B_1, \ldots, B_k$ (abbreviated $\leftarrow B_1, \ldots, B_k$) is called an *integrity constraint*. Intuitively, an integrity constraint $B_1, \ldots, B_k$ expresses the fact that we want to discard all models of the given program that entail $B_1 \wedge \ldots \wedge B_k$. A GASP-*program* is a collection of GASP-rules.

Given a GASP-rule $H \leftarrow B_1, \ldots, B_k$, let us denote with $\mathcal{U}\_body$ the collection of user-defined literals in $B_1, \ldots, B_k$, and with $\mathcal{C}\_body$ the collection of constraint atoms in $B_1, \ldots, B_k$. Hence, $H \leftarrow B_1, \ldots, B_k$ can be written as $H \leftarrow \mathcal{U}\_body, \mathcal{C}\_body$. Moreover we define with $body^+$ the collection of positive literals in $\mathcal{U}\_body \cup \mathcal{C}\_body$ and with $body^-$ the collection of atoms that appear in negative literals in $\mathcal{U}\_body$.

We assume that our language provides also special atoms called *cardinality constraints* [22]. Accordingly, Π_C includes the symbol $\{\}/3$ which is used to construct cardinality constraints of the form $h\{\varphi\}k$. $h\{\varphi\}k$ is well-formed iff h and k are integer s.t. $0 \leq h \leq k$, and φ is a sequence of atoms of the form $A : B_1, \ldots, B_n$, $n \geq 0$ (written A, if $n = 0$), where $A, B_1, \ldots, B_n$ are user-defined atoms and $B_1, \ldots, B_n$ occur as head atoms in some rules of the program. Furthermore, $vars(B_1, \ldots, B_n) \subseteq vars(A)$. Cardinality constraints can occur both in the head and in the body of a rule. When occurring in the head, their intuitive meaning is the following. $h\{A[\bar{X}, \bar{Y}_1, \ldots, \bar{Y}_n] : B_1[\bar{Y}_1], \ldots, B_n[\bar{Y}_n]\}k$ forces models of the given program to contain, for each tuple of ground terms $\bar{t}$ for $\bar{X}$, a set R such that $R \subseteq \{A : \bar{X} = \bar{t}, \exists \bar{Y}_1 \ldots \exists \bar{Y}_n (B_1, \ldots, B_n)\} \wedge h \leq |R| \leq k$. For example, given the program

$$r(1..3). \quad q(a). \quad q(b). \quad 1\{p(X, Y) : r(Y)\}1 \leftarrow q(X).$$

its models are required to contain exactly one among $p(a, 1)$, $p(a, 2)$, $p(a, 3)$ and exactly one among $p(b, 1)$, $p(b, 2)$, $p(b, 3)$.

When cardinality constraints occur in the body of a rule, they will be entailed by models that meet the above- mentioned property.

We assume, as done in several existing ASP systems, that programs satisfy the *range restriction* property, suitably adapted to account for constraints. A GASP-rule is range restricted if all variables occurring in its head (except those which are "local" to cardinality constraints) occur also in at least one positive atom of $\mathcal{U}\_body$. A GASP-program is range restricted iff every rule in it is range restricted. In this way, all variables in the program are guaranteed to have a finite set of possible values associated with.

Semantics. A GASP-program can be seen as a syntactic shorthand for an ASP program where any non-ground GASP-rule represents a family of ground ASP rules. Let $\mathcal{A}$ be a collection of propositional atoms. An *ASP rule* has the form:

$$p \leftarrow p_0, \ldots, p_n, \mathbf{not}\, p_{n+1}, \ldots, \mathbf{not}\, p_m$$

where $\{p, p_0, \ldots, p_n, p_{n+1}, \ldots, p_m\} \subseteq \mathcal{A}$. An ASP program is a collection of ASP rules. The process of replacing each non-ground rule with an equivalent finite set of ground rules is called *grounding*.

A *ground instance* of a rule R of P is obtained from R by replacing all variables in it by ground terms built using the symbols in $\mathcal{F} \setminus (\{'..'\} \cup F_Z)$, respecting well-formedness of the resulting ground atoms. In addition, each variable $v \in \mathcal{V}$ that appears in a Σ-term whose functor is in $F_Z \cup \{'..'\}$ or that appears in GASP-constraints based on $\leq$ has to be grounded using an element of $\mathbb{Z}$. Additionally, note that:

- We omit compound integer terms from the grounding process—as these are meant to be evaluated and replaced with the constants representing the values of the compound terms (elements of $\mathbb{Z}$);
- We omit intervals. Instead, we expect the grounding process to replace each rule of the form $p(\bar{t}, a..b, \bar{s}) \leftarrow body$, with the set of rules

$$p(\bar{t}, a, \bar{s}) \leftarrow body \quad p(\bar{t}, a+1, \bar{s}) \leftarrow body \cdots p(\bar{t}, b, \bar{s}) \leftarrow body$$

- each ground constraint atom C is replaced with true or false depending on whether C is entailed or not in the traditional theory of integer arithmetic.

Let us note that $\mathcal{C}\_body$ disappears as soon as the program is grounded.

A ground program ground(P) is obtained from P by replacing all rules in P by all ground instances of all rules in P.

Integrity constraints are always removed from the generated program: an ASP integrity constraint $\leftarrow p_0, \ldots, p_n, \mathbf{not}\, p_{n+1}, \ldots, \mathbf{not}\, p_m$ is equivalent, for stable models, to $p \leftarrow \mathbf{not}\, p, p_0, \ldots, p_n, \mathbf{not}\, p_{n+1}, \ldots, \mathbf{not}\, p_m$, where p is a new propositional atom. Similarly, rules containing cardinality constraints are replaced by a collection of rules that precisely capture their semantics.

Therefore we can use all definitions and results usually adopted in ASP to provide a semantics characterization of GASP-programs. In particular, we consider here the semantics based on the notion of *well-founded model* [24]. The *well-founded model* [24] of a general program P is a *3-interpretation* I, i.e., a pair $\langle I^+, I^- \rangle$ such $I^+ \cup I^- \subseteq \mathcal{A}$ and $I^+ \cap I^- = \emptyset$. I^+ denotes the atoms that are known to be true while I^- denotes those atoms that are known to be false.

The union between two 3-interpretations $I \cup J$, where $I = \langle I^+, I^- \rangle$ and $J = \langle J^+, J^- \rangle$, is defined as $\langle I^+ \cup J^+, I^- \cup J^- \rangle$. The intersection is defined similarly. If $I^+ \cup I^- = \mathcal{A}$, then the interpretation I is said to be *complete*. Given two 3-interpretations I, J, we use $I \subseteq J$ to denote the fact that $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. The notion of entailment for 3-interpretations can be defined as follows. If $p \in \mathcal{A}$, then $I \models p$ iff $p \in I^+$; $I \models \mathbf{not}\, p$ iff $p \in I^-$; $I \models A \wedge B$ iff $I \models A$ and $I \models B$, and $I \models H \leftarrow A_1 \wedge \ldots \wedge A_n$ iff $I \models H$ or there is $i \in \{1, \ldots, n\}$ such that $I \models \mathbf{not}\, A_i$.

Intuitively, the well-founded model of P contains only (possibly not all) literals that are necessarily true and the ones that are necessarily false in all stable models of P. The remaining literals are undefined. It is well-known that a general program P has a unique well-founded model $\mathsf{wf}(P)$ [24]. If $\mathsf{wf}(P)$ is complete then it is also a stable model (and it is the unique stable model of P).

3 Computation-based characterization of stable models

The computation model adopted in GASP has been derived from recent investigations about alternative models to characterize answer set semantics for various extensions of ASP—e.g., programs with *abstract constraint atoms* [17].

The work described in [15] provides a *computation-based* characterization of answer sets for programs with abstract constraints. One of the side-effects of that research is the development of a computation-based view of answer sets for general logic programs. The original definition of answer sets [10] requires guessing an interpretation and successively validating it—through the notion of reduct (P^I) and the ability to compute minimal models of a definite program (e.g., via repeated iterations of the immediate consequence operator [16]).

The characterization of answer sets derived from [15] does not require the initial guessing of a complete interpretation; instead it combines the guessing process with the construction of the answer set.

We provide our formalization of computation in terms of GASP-*computation* a nd show that it is an instance of [15]. We begin with the following notion:

Definition 1 (Applicable rule). *We say that a ground rule $a \leftarrow body$ is* applicable *w.r.t. an interpretation I, if $body^+ \subseteq I^+$ and $body^- \cap I^+ = \emptyset$.*

We extend the definition of applicable to a non-ground rule R w.r.t. I iff there exists a grounding r of R that is applicable w.r.t. I. Note that $\mathcal{C}\_body$ is replaced by true during the local grounding stage.

Given a program P and an interpretation I, we denote with $P \cup I$ the program

$$P \cup I = (P \setminus \{r \in P \mid head(r) \in I^-\}) \cup I^+.$$

Intuitively, $P \cup I$ is the program P modified in such a way to guarantee that all elements in I^+ are true and all elements in I^- are false.

Definition 2 (GASP-computation). *A* GASP-computation *of a program P is a sequence of 3-interpretations $I_0, I_1, I_2, \ldots$ that satisfies the following properties:*

- $I_0 = \mathsf{wf}(P)$
- $I_i \subseteq I_{i+1}$ *for all* $i \geq 0$ *(*Persistence of Beliefs*)*
- *if* $I = \bigcup_{i=0}^{\infty} I_i$, *then* $\langle I^+, \mathcal{A} \setminus I^+ \rangle$ *is a model of P (*Convergence*)*
- *for each* $i \geq 0$ *there exists a rule* $a \leftarrow body$ *in P that is applicable w.r.t. I_i and* $I_{i+1} = \mathsf{wf}(P \cup I_i \cup \langle body^+, body^- \rangle)$ *(*Revision*)*
- *if* $a \in I_{i+1}^+ \setminus I_i^+$ *then there is a rule* $a \leftarrow body$ *in P which is applicable w.r.t. I_j, for each* $j \geq i$ *(*Persistence of Reason*).*

The proofs of correctness and completeness of GASP-computation w.r.t. the answer sets of a program P can be found at `http://sole.dimi.uniud.it/~agostino.dovier/GASP/`.

4 A CLP approach for Stable Models computation

In this section we show how the GASP-computation can be implemented within a CLP(FD) framework. The use of a Prolog based implementation allows fast prototyping of the search techniques and heuristics. The CLP environment allows non-ground computation of arithmetic constraints to be easily embedded into the implemented system.

4.1 Representation of Interpretations based on FDSETs.

Instances of a predicate that are true and false within an interpretation are encoded as sets of tuples, and handled using FD techniques. In order to compute the set of ground applicable rules, a local grounding phase is performed according to the definition of applicable rule, i.e. only compatible assignments of the rule w.r.t. the current interpretations are considered. During the construction of a model, the effect of this strategy is to ground only those rules that effectively contribute in supporting each stable model. Moreover, when arithmetic constraints are present in a rule, the ability to treat them in their non-ground version, allows to save on the enumeration of all possible admissible combinations of their ground instances.

The computation of applicable rules is at the basis of the GASP-computation and it is performed very frequently, i.e. at every node of the computation tree. From a relational algebra point of view, this can be seen as a set of join and projection operations on a set of tuples. If performed naively, these operations may become inefficient, especially when the number of tuples increases. We cope with this problem by introducing a *compact* and dynamic representation of the interpretations based on FDSETs. This allows us to efficiently handle large sets of tuples with respect to memory usage and query time. The compact representation is withdrawn to build a CSP whose solutions correspond to the applicable ground rules.

FDSETs are a data structure available in the clpfd library of SICStus Prolog that allows to efficiently store and compute on sets of integer numbers. Basically, a set $\{a_1, a_2, \ldots, a_n\}$ is interpreted as the union of a set of intervals $[a_{b_1}..a_{e_1}], \ldots, [a_{b_k}..a_{e_k}]$ and stored consequently as $[[a_{b_1}|a_{e_1}], \ldots, [a_{b_k}|a_{e_k}]]$. A library of built-in predicates for dealing with this data structure is made available.

We identify with p^n a predicate p with arity n. In the program, a predicate p^n appears as $p(X_1, \ldots, X_n)$ where, in place of some variables, a constant can occur (e.g., $p(a, X, Y, d)$). The interpretation of the predicate p^n can be modeled as a set of tuples $(a_1, a_2, \ldots, a_n)$, where $a_i \in \mathit{Consts}(P)$—where $\mathit{Consts}(P)$ denotes the set of constants in the language used by the program

P. The explicit representation of the set of tuples has the maximal cardinality $|Consts(P)|^n$. The idea is to use a more compact representation based on FDSETs, after a mapping of tuples to integers. Without loss of generality, we assume that $Consts(P) \subseteq \mathbb{N}$. Each tuple $\boldsymbol{a} = (a_0, \ldots, a_{n-1})$ is mapped to the *unique* number $\mathsf{map}(\boldsymbol{a}) = \sum_{i \in [0..n-1]} a_i \mathbb{M}^i$, where $\mathbb{M}$ is a "big number", $\mathbb{M} \geq |Consts(P)|$. In case of predicates without arguments (predicates of arity 0), for the empty tuple () we set $\mathsf{map}(()) = 0$. We also extend the map function to the case of non-ground tuples, using FD variables. If $\boldsymbol{Y} = (y_1, y_2, \ldots, y_n)$, where $y_i \in Vars(P) \cup Consts(P)$, then $\mathsf{map}(\boldsymbol{Y})$ is the FD expression that represent the sum defined above. For instance, if $\boldsymbol{Y} = (3, X, 1, Y)$ and $\mathbb{M} = 10$, then $\mathsf{map}(\boldsymbol{Y}) = 3 + X * 10 + 1 * 10^2 + Y * 10^3$. Moreover, all variables possibly occurring in $\boldsymbol{Y}$ are constrained to have domain $0..\mathbb{M} - 1$.
A 3-interpretation $\langle I^+, I^- \rangle$ is represented by a set of 4-tuples $(p, n, \mathtt{POS}_{p,n}, \mathtt{NEG}_{p,n})$, one for each predicate symbol, where p is the predicate name, n its arity, and

$$\mathtt{POS}_{p,n} = \{\mathsf{map}(\boldsymbol{x}) : I^+ \models p(\boldsymbol{x})\} \qquad \mathtt{NEG}_{p,n} = \{\mathsf{map}(\boldsymbol{x}) : I^- \models \mathbf{not}\ p(\boldsymbol{x})\}$$

If clear from the context, we drop the subscript n from the notation. These sets are represented and handled efficiently, by using FDSETs. For instance, if

$$\begin{aligned}\mathtt{POS}_{p,3} = \{&\mathsf{map}(0,0,1), \mathsf{map}(0,0,2), \mathsf{map}(0,0,3), \mathsf{map}(0,0,8),\\ &\mathsf{map}(0,0,9), \mathsf{map}(0,1,0), \mathsf{map}(0,1,1), \mathsf{map}(0,1,2)\}\end{aligned}$$

and $\mathbb{M} = 10$, then its representation as FDSETs is simply: $[[1|3], [8|12]]$, in other words, the disjunction of two intervals.

4.2 Computing applicable rules.

We briefly show now how local grounding is performed, highlighting the role of arithmetic constraints in the rule.

The idea is to build a CSP where the variables appearing in the rule correspond to FD variables. Solutions to the CSP correspond to ground rules that are applicable. According to our definition, the applicable rule has its $body^+$ that is completely contained in I^+ and its $body^-$ that has no ground predicate that appears in I^+. These requirements can be encoded in terms of FD constraints, by linking the FD variables appearing in a predicate to the values associated to the predicate by the function map and contained in the FDSET representation of I. More formally, let us assume a rule

$$r \equiv p_0(\boldsymbol{X}_0) \leftarrow C(\boldsymbol{X}), p_1(\boldsymbol{X}_{p_1}), \ldots, p_k(\boldsymbol{X}_{p_k}), \mathbf{not}\ \ p_{n+1}(\boldsymbol{X}_{p_{n+1}}), \ldots, \mathbf{not}\ \ p_m(\boldsymbol{X}_{p_m}),$$

where C is $\mathcal{C}\_body$ of r (namely a conjunction of arithmetic constraints), $\boldsymbol{X}_i$ is a list of variables and/or ground integers that are compatible to the arity of p_i. For each variable in the rule, a corresponding FD variable is defined. Moreover, for each predicate p_i another FD variable V_i is created.

Every variable V_i is bounded to the corresponding variables $\boldsymbol{X}_i$, according to the $\mathsf{map}(\boldsymbol{X}_i)$ function, i.e. $V_i = \mathsf{map}(\boldsymbol{X}_i)$. Moreover, in order to implement

the semantics of applicable rule, we require that each predicate p_i in $body^+$ has domain $\texttt{POS}_{p_i}$ and each predicate p_i in $body^-$ can not take values from $\texttt{POS}_{p_i}$. In addition we require that the head does not appear in I, since in that case it would be already supported and the rule would be applied redundantly. Finally, the constraint C is added to the CSP by introducing the corresponding FD constraints. This choice allows GASP to be easily extensible and to support a wide range of constraints in a modular way.

The solutions of this CSP are all the ground instances of V_0, computed through labeling, that represent the possible head values to be added to the model.

The same technique is adapted to compute T_P and the single steps of the alternating fixpoint procedure for computing the well-founded models.

4.3 The overall algorithm.

We describe now the methodology followed in the implementation of the GASP-computation. We distinguish among three cases: the program is positive, the program admits a well-founded model, the program does not admit a complete well-founded model (it can have zero or more stable models).

In the first case, the computation of T_P operator fixpoint is performed, and the resulting interpretation is the only stable model. The computation of the fixpoint uses similar techniques to the CSP-based computation of the applicable rule, and it makes use of a dependency graph in order to select the rules to activate during the fixpoint.

In the second case, the idea of alternating fixpoint [25] is coded in Prolog. The implementation boils down to controlling the alternating fixpoint computation and to encode the $T_{P,J}$ operator (see e.g., [25]). Once again, similar CSPs to applicable rule computations are added in order to compute the $T_{P,J}$ operators.

In the third case, the GASP-computation is launched starting from that model. Instead of starting from an empty model, literals that are necessarily true and false respectively in each stable model are included in the starting model and lesser application of rules are required.

The GASP-computation is implemented through a chronological backtracking search where choice points contain the option whether to apply an applicable rule or not. The key ingredients of the main loop are: the computation of an extension of the T_P operator fixpoint, the handling of some specific cardinality constraint and the implementation of some rule-based propagators.

In Figure 1, we summarize in pseudocode the algorithm. Each applicable rule represents a non-deterministic choice in the computation of a stable model. The computation explores the first of these choices (line 4), and acts depending on the head a of the rule. In case the head is a cardinality constraint (we currently support exactly one, but this can be extended in the future), a non-deterministic assignment is added to the model, where one literal out of the possible candidates is added to I^+ and all the remaining to I^- (line 6). After the assignment, a fixpoint over the computation of T_P and propagators is performed before entering the recursive call. The propagation phase will be discussed in the next section.

```
(1)    rec_search(P,I)
(2)       R = applicable_rules(I)
(3)       if (R = ∅ and I is a model) output: I is a stable model
(4)       else select a ← body⁺, not body⁻ ∈ R
(5)          if (a = 1{...}1)
(6)             ND-choice: I = ⟨assignment, body⁻⟩ ∪ I
(7)             I = fixpoint(propagation(P,T_P(P∪(⟨∅, body⁻⟩ ∪ I))))
(8)          else
(9)             ( I = ⟨{a}, body⁻⟩ ∪ I,
(10)              I = fixpoint(propagation(P,T_P(P∪(⟨∅, body⁻⟩∪I))))
(11)            OR (Non-deterministic)
(12)              P= P ∪{← body⁻},
(13)              I = fixpoint(propagation(P,T_P(P∪I))))
(14)         if (I not failed) rec_search(P,I)
```

Fig. 1. The answer set computation

If the head is a normal literal (line 9) then a non-deterministic choice is opened (lines 9 or 12). In the first part, the rule is applied and thus a and $body^-$ are added to I. After the fixpoint (line 10) the recursive call (line 14) is performed. In the second part, we consider the case in which the rule is never chosen in the subtree and to ensure this a new integrity constraint is added to the program (line 12). After the fixpoint the recursive call is made.

Let us recall that every time the local grounding in invoked, a CSP is built. We believe that the enhancement of this step (e.g., building CSPs less often and/or incremental CSP) could reduce the search time significantly. In line 14 "I not failed" means that $I^+ \cap I^- \neq \emptyset$.

The process may encounter a contradiction while adding new facts to the interpretation, and consequently the computation may encounter failures. Whenever there are no more applicable rules, a leaf in the search tree is reached (line 3) and the corresponding stable model is obtained (convergence property).

From the implementation point of view, it turns out that computing well-founded models at every non-deterministic application of a rule is time consuming. In particular, the computation of the extension of P with new facts from the positive interpretation is inefficient.

To gain efficiency, we substitute the call to the well-founded computation with a variant of the T_P operator. The extension of T_P to ASP considers rules where $body^+ \in I^+$ and $body^- \in I^-$. The T_P operator adds new positive atoms as stated by the head of the rule. Using well-founded computation involves the alternating fixpoint procedure which is not efficient enough to be included at each level of the search. The combination of T_P fixpoint and our propagators provide better results. In future work we plan to improve the well-founded model computation algorithm and to use it in place of the T_P fixpoint.

4.4 Non-Ground Propagation.

A propagation step is launched before each leaf expansion, in order to deduce additional literals that can be safely introduced in the current interpretation and that neither T_P nor well-founded fixpoints are able to infer.

The ideas presented below represent a generalization of some of the techniques that drive the search in SMODELS. In particular, we deal with non-ground rules and therefore we introduce a CSP-based analysis similar to the computation of the applicability of rules. The resolution of the CSP is designed to avoid the complete grounding of the rules involved. We address three settings where negative literals can be deduced: inferring a literal that appears (i) in the body, (ii) in the head and (iii) in a cardinality constraint in the head.

Let $I = \langle I^+, I^- \rangle$ be the current model, R be a non-ground ASP rule of the form $head(R) : -body^+(R), \mathbf{not}\ \ body^-(R)$ and R' a grounding of R.

The case (i) applies when there exists a grounding R' such that $head(R') \in I^-$. In this case the rule R' should not become applicable, otherwise $head(R')$ would be added in I^+ and generate a failure. We consider the specific situation in which $body(R')$ is completely satisfied except for exactly one undetermined literal $l \in body^+$ ($l \notin I^+ \cup I^-$). To prevent the rule R' to fire, the literal l is added to I^-.

The case (ii) applies when it is possible to deduce that an undetermined literal $l \notin I^+ \cup I^-$ may not be introduced in I^+ in any subsequent computation. The (ground) literal l can be introduced in I^+ only if there is at least one (potentially) applicable rule R' such that $head(R') = l$. If some literals $p \in body(R')$ are undetermined, we assume that they can potentially contribute to satisfy the body: i.e., if $p \in body^+(R')$ then p is assumed to be true and if $p \in body^-(R')$ then p is assumed to be false. If there is no such rule R' then the literal l can be safely added to I^-.

The case (iii) applies when a positive ground literal in I^+ and the predicate matches the cardinality constraint ($1\{...\}1$) in the head of an applicable rule. In this case, every other literal in the same range can be safely set to false.

Note that the inference of positive literals is possible as well, however they can not be introduced in the model, unless a test for unfoundedness is performed (they must be supported by some chains of applicable rules). We plan as future work to investigate this kind of propagation that resembles a mixed top-down approach in the computation of stable models.

5 Experiments

The prototype implementing the ideas described above and all the tests described in this section are available at `www.dimi.uniud.it/dovier/GASP`. The prototype has been developed using SICStus Prolog 4.0 (`www.sics.se/isl/sicstuswww/site/`), chosen for its rich library of FDSET primitives. Although faster constraint solvers are available (e.g., Gecode), we prefer to stay in the realm of declarative programming.

Test	Param	LPARSE	SMODELS	CLASP	GASP
non_wf_graph	40	4.035	0.735	1.720	1.82
(all sol)	80	29.824	6.874	7.8	13.55
	160	235.039	61.568	45.6	120.81
	320	1,885	-	-	1,380.77
Send More Money (all)	none	55.69	0.01	0.01	3.43
Queens	22	0.310	172.5	0.05	0.62
(1st sol)	23	0.360	395.9	0.06	1.68
	24	0.415	220.0	0.08	1.20
	25	0.464	2,067.0	0.09	8.38
Squares	(19,7)	1.76	2.99	0.17	1.43
(1st sol)	(6,24)	88	371.71	17.37	0.49
	(6,45)	1,140	-	-	1.48
p2	100	0.84	0.286	.200	0.75
(all sol)	200	3.346	1.172	1.45	3.00
	300	8.338	2.691	5.54	7.72
	400	13.242	4.819	15.27	14.81

Table 1. Timings. '-' means that computation was killed after 24 hours

We performed some preliminary experiments, using different classes of ASP programs, and we report the execution times in Table 1. All the experiments have been performed on an AMD Opteron 2.2 GHz Linux Machine. For the ASP tests, we used LPARSE 1.1.5, SMODELS 2.33 (`www.tcs.hut.fi/Software/smodels/`), CLASP 1.1.0 (`www.cs.uni-potsdam.de/clasp`) and ASPeRiX 0.1 (`http://www.info.univ-angers.fr/pub/claire/asperix/`).

In Table 1 we report on the benchmarks we run to compare the performances of GASP and LPARSE+SMODELS and LPARSE+CLASP. Times are in seconds.

The first set of benchmarks (non_wf_graph) is based on a non well-founded program inspired by a graph problem, where the parameter determines the size of the graph. The program admits two distinct stable models and basically computes a transitive closure h of a binary predicate p, then add the predicate r: $r(X,Y)$:- $h(X,Y), \textbf{not}\ p(X,Y)$. Depending on the stable model, the predicate p is slightly modified. The preliminary computation of well-founded model returns a sub-model and the non-deterministic GASP computation procedure must be used. The grounding time (and size of the program—with $q = 320$ the file is 1.9GB) are not negligible. However, in large instances GASP outperforms LPARSE+SMODELS and LPARSE+CLASP even removing the time spent for grounding.

The second benchmark is the classical Send + More = Money problem, coded with ASP. Here constraint propagation performed by CLP in GASP is the key for solving the problem efficiently. Compared to LPARSE, it is interesting to note that even if the size of the ground program is small (53K and 1300 rules), it takes almost a minute to produce the file.

Test	Param	GASP	ASPeRiX	GASP nodes	ASPeRiX nodes
p2	100	0.75	0.22	0	10,000
(all sol)	200	3.00	1.20	0	40,000
	300	7.72	3.53	0	90,000
	400	14.81	7.92	0	160,000
non_wf_graph	40	1.82	0.096	2	1
(all sol)	80	13.55	0.671	2	1
	160	120.81	5.315	2	1
	320	1,380.77	42.918	2	1

Table 2. GASP and ASPeRiX

The third set of benchmarks is the N queens problem. Here, GASP outperforms SMODELS, while the performances of CLASP are only biased by the grounding time.

The fourth set is taken from the CSPLIB (`www.csplib.org`). The problem # 9 is the perfect square placement problem, where a set of non-overlapping squares must be placed inside a larger square. We designed 3 test sets, where (N, S) indicates the number N of squares and S the master side: the set (19,7) contains 1 square (side 4), 5 squares (side 2) and 13 squares (side 1); the set (6,24) contains 1 square (side 16) and 5 squares (side 8); the set (6,45) contains 1 square (side 30) and 5 squares (side 15). In this test, the performances of GASP are impressive, since the non-ground computation takes advantage of FD constraint solving during the search. The time spent by LPARSE increases dramatically with the size of the master square as well as the size of the ground program (for (6,24) we have 71MB and 3.5M rules, for (6,45) 945MB and 44M rules), thus making it impossible for the solvers to find a solution.

Finally we included last test p2 taken from [11] and used by authors to prove the effectiveness of ASPeRiX in a case with large grounding when one is interested in a single solution. The program admits a stable model that contains a complete graph of a number of nodes that is provided as parameter. We can see that the GASP is capable of finding the solutions in the time needed to ground the program. Note that the ground program for 400 nodes has 22MB of size and contains 880K rules.

In Table 2 we compare the performances of GASP and ASPeRiX on programs that are supported by the latter (ASPeRiX does not support cardinality constraints). Since the approach is similar, we can compare the size of the search trees (number of choice points nodes). For the time comparisons, recall that ASPeRiX has a C++ implementation, while GASP is written in Prolog.

In the p2 program, despite the penalty for running into a Prolog environment, GASP timings are comparable and linearly scaled to ASPeRiX. Moreover, the rule-based propagators of GASP are able to reduce the search tree to a single node, while ASPeRiX develops a quadratic sized tree. On the other hand, the pruning of the tree in GASP represents the principal cost of the search.

In the second test (non_wf_graph), GASP is much slower than ASPeRiX, suggesting that the propagators used by GASP perform unnecessary work. This issue will be considered in future work. In the current implementation, when propagation is performed, CSPs are built on the current interpretation and they ignore the partial work performed in previous runs. Considering incremental versions of these CSPs could save the largest fraction of time currently used.

6 Conclusions

In this paper, we provided the foundation for a bottom-up construction of stable models of a program P without preliminary program grounding. The notion of GASP-computation has been introduced; this model does not rely on the explicit grounding of the program. Instead, the grounding is local and performed on-demand during the computation of the answer sets. The GASP language handles cardinality constraints and arithmetic constraints that can be implemented in a non-ground fashion and provide significant enhancements in the computation. We believe this approach can provide an effective avenue to achieve greater efficiency in space and time w.r.t. a complete program grounding.

We illustrated our Prolog implementation of GASP using CSP on FD variables and FDSETs. The performances of GASP show that, notwithstanding the Prolog overhead and naive data structures used, the computations are comparable and often better than traditional ground-based approaches.

We plan to investigate how to handle incremental CSPs in order to save redundant work, to reimplement efficiently the well-founded computation and include it in the main loop, to study a mixed goal-driven resolution (top-down approach) that should guide the non-deterministic choices.

Acknowledgments. The work has been partially supported by MIUR FIRB RBNE03B8KK and PRIN projects, and NSF grants HRD0420407, CNS0220590 and IIS0812267. We really thank Andrea Formisano for the several useful discussions.

References

1. Y. Babovich and M. Maratea. Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs. *LPNMR*, LNCS 2923:346–350, Springer, 2004.
2. C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, 2003.
3. P. Bonatti, E. Pontelli, T. Son. Credulous Resolution for ASP. *AAAI*, 2008.
4. D. Brooks, E. Erdem, S. Erdogan, J. Minett, and D. Ringe. Inferring Phylogenetic Trees Using Answer Set Programming. *JAR*, 39(4):471–511, 2007.
5. P. Codognet and D. Diaz. A Minimal Extension of the WAM for `clp(fd)`. *ICLP*, pp 774–790, MIT Press, 1993.
6. A. Dovier, A. Formisano, and E. Pontelli. A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems. *ICLP*, LNCS 3668:67–82, Springer, 2005.

7. I. Elkabani, E. Pontelli, T. Son. A System for Computing Answer Sets of Logic Programs with Aggregates. *LPNRM*, LNCS 3662:427–431, Springer, 2005.
8. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Clasp: a Conflict-driven Answer Set Solver. *LPNMR*, LNCS 4483:260–265, Springer, 2007.
9. M. Gebser, T. Schaub, S. Thiele, B. Usadel, P. Veber. Detecting Inconsistencies in Large Biological Networks with ASP. *ICLP*, LNCS 5366:130–144, Springer, 2008.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. *ICLP*, pp 1070–1080, MIT Press, 1988.
11. C. Lefevre and P. Nicolas. Integrating Grounding in Search Process for Answer Set Computing. *Work. on Integrating ASP and Other Computing Paradigms*, 2008.
12. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. , S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006
13. V. Lifschitz. Answer Set Planning. *LPAR*, LNCS 1705:373–374, Springer, 1999.
14. F. Lin, and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence* 157(1-2): 115–137, 2004.
15. L. Liu, E. Pontelli, S. Tran, and M. Truszczynski. Logic Programs with Abstract Constraint Atoms: the Role of Computations. *ICLP*, LNCS 4670:286–301. Springer, 2007.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer, Heidelberg, 1987.
17. V. Marek and J. Remmel. Set Constraints in Logic Programming. LPNMR, LNCS 2923:167–179. Springer, 2004.
18. V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In K.R. Apt, V.W. Marek, M. Truszcziński, and D. S. Warren, editors, *The Logic Programming Paradigm*. Springer, 1999.
19. V. Mellarkod and M. Gelfond. Integrating Answer Set Reasoning with Constraint Solving Techniques. *FLOPS*, LNCS 4989:15–31, Springer, 2008.
20. I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and AI*, 25(3-4): 241–273, 1999.
21. I. Niemelä and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. *LPNMR*, LNCS 1265:421–430. Springer, 1997.
22. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1.2): 181–234, 2002.
23. T. Son and E. Pontelli. Planning for Biochemical Pathways: a Case Study of Answer Set Planning in Large Planning Problem Instances. *First International Workshop on Software Engineering for Answer Set Programming*, pp 116–130, 2007.
24. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
25. U. Zukowski, B. Freitag, and S. Brass. Improving the Alternating Fixpoint: The Transformation Approach. *LPNMR*, LNCS 1265:4–59, Springer, 1997.