Complex Optimization in Answer Set Programming — Extended Version —

Martin Gebser and Roland Kaminski and Torsten Schaub∗
Institut für Informatik, Universität Potsdam

submitted [n/a]; revised [n/a]; accepted [n/a]

Abstract

Preference handling and optimization are indispensable means for addressing non-trivial applications in Answer Set Programming (ASP). However, their implementation becomes difficult whenever they bring about a significant increase in computational complexity. As a consequence, existing ASP systems do not offer complex optimization capacities, supporting, for instance, inclusion-based minimization or Pareto efficiency. Rather, such complex criteria are typically addressed by resorting to dedicated modeling techniques, like *saturation*. Unlike the ease of common ASP modeling, however, these techniques are rather involved and hardly usable by ASP laymen. We address this problem by developing a general implementation technique by means of meta-programming, thus reusing existing ASP systems to capture various forms of qualitative preferences among answer sets. In this way, complex preferences and optimization capacities become readily available for ASP applications.

1 Introduction

Preferences are often an indispensable means in modeling since they allow for identifying preferred solutions among all feasible ones. Accordingly, many forms of preferences have already found their way into systems for Answer Set Programming (ASP; (Baral 2003)). For instance, *smodels* provides optimization statements for expressing cost functions on sets of weighted literals (Simons et al. 2002), and *dlv* (Leone et al. 2006) offers weak constraints for the same purpose. Further approaches (Delgrande et al. 2003; Eiter et al. 2003) allow for expressing various types of preferences among rules. Unlike this, no readily applicable implementation techniques are available for qualitative preferences among answer sets, like inclusion minimality, Pareto-based preferences as used in (Sakama and Inoue 2000; Brewka et al. 2004), or more complex combinations as proposed in (Brewka 2004). This shortcoming is due to their higher expressiveness leading to a significant increase in computational complexity, lifting decision problems (for normal logic programs) from the first to the second level of the polynomial time hierarchy (cf. (Garey and Johnson 1979)). Roughly speaking, preferences among answer sets combine an NP with a $coNP$ problem. The first one defines feasible solutions, while the second one ensures that there are no better solutions according to the preferences at hand. For implementing such problems, Eiter and Gottlob invented in (1995) the *saturation* technique, using the elevated complexity of disjunctive logic programming. In stark contrast to the ease of common ASP modeling (e.g.,

∗ Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

strategic companies can be "naturally" encoded (Leone et al. 2006) in disjunctive ASP), however, the saturation technique is rather involved and hardly usable by ASP laymen.

For taking this burden of intricate modeling off the user, we propose a general, saturation-based implementation technique capturing various forms of qualitative preferences among answer sets. This is driven by the desire to guarantee immediate availability and thus to stay within the realm of ASP rather than to build separate (imperative) components. To this end, we take advantage of recent advances in ASP grounding technology, admitting an easy use of meta-modeling techniques. The idea is to reinterpret existing optimization statements in order to express complex preferences among answer sets. While, for instance in *smodels*, the meaning of `#minimize` is to compute answer sets incurring minimum costs, we may alternatively use it for selecting inclusion-minimal ones. In contrast to the identification of minimal models, investigated by Janhunen and Oikarinen in (2004; 2008), a major challenge lies in guaranteeing the stability property of implicit counterexamples, which must be more preferred answer sets rather than (arbitrary) models. For this purpose, we develop a refined meta-program qualifying answer sets as viable counterexamples. Unlike the approach of Eiter and Polleres (2006), our encoding avoids "guessing" a level mapping to describe the formation of a counterexample, but directly denies models for which there is no such construction. Notably, our meta-programs apply to (reified) *extended* logic programs (Simons et al. 2002), possibly including choice rules and `#sum` constraints, and we are unaware of any existing meta-encoding of their answer sets, neither as candidates nor as counterexamples refuting optimality.

2 Background

We consider extended logic programs (Simons et al. 2002) allowing for (proper) disjunctions in heads of rules (Gelfond and Lifschitz 1991). A *rule* r is of the following form:

$$H \leftarrow B_1, \dots, B_m, {\sim}B_{m+1}, \dots, {\sim}B_n.$$

By $\mathit{head}(r) = H$ and $\mathit{body}(r) = \{B_1, \dots, B_m, {\sim}B_{m+1}, \dots, {\sim}B_n\}$, we denote the *head* and the *body* of r, respectively, where "$\sim$" stands for default negation. The head H is a disjunction $a_1 \vee \dots \vee a_k$ over *atoms* $a_1, \dots, a_k$, belonging to some alphabet $\mathcal{A}$, or a `#sum` constraint $L\,\#\mathtt{sum}[\ell_1 = w_1, \dots, \ell_k = w_k]\,U$. In the latter, $\ell_i = a_i$ or $\ell_i = {\sim}a_i$ is a *literal* and w_i a *non-negative* integer *weight* for $a_i \in \mathcal{A}$ and $1 \leq i \leq k$; L and U are integers providing a lower and an upper bound. Either or both of L and U can be omitted, in which case they are identified with the (trivial) bounds 0 and ∞, respectively. A rule r such that $\mathit{head}(r) = \bot$ (H is the empty disjunction) is an *integrity constraint*. Each body component B_i is either an atom or a `#sum` constraint for $1 \leq i \leq n$. If $\mathit{body}(r) = \emptyset$, r is called a *fact*, and we skip "$\leftarrow$" when writing facts below. For a set $\{B_1, \dots, B_m, {\sim}B_{m+1}, \dots, {\sim}B_n\}$, a disjunction $a_1 \vee \dots \vee a_k$, and a `#sum` constraint $L\,\#\mathtt{sum}[\ell_1 = w_1, \dots, \ell_k = w_k]\,U$, we let $\{B_1, \dots, B_m, {\sim}B_{m+1}, \dots, {\sim}B_n\}^+ = \{B_1, \dots, B_m\}$, $(a_1 \vee \dots \vee a_k)^+ = \{a_1, \dots, a_k\}$, and $(L\,\#\mathtt{sum}[\ell_1 = w_1, \dots, \ell_k = w_k]\,U)^+ = [\ell_i = w_i \mid 1 \leq i \leq k, \ell_i \in \mathcal{A}]$. Note that the elements of a `#sum` constraint form a multiset, possibly containing duplicates. For some $S = \{a_1, \dots, a_k\}$ or $S = [a_1 = w_1, \dots, a_k = w_k]$, we define $\mathit{atom}(S) = \{a_1, \dots, a_k\}$.

A (Herbrand) *interpretation* is represented by the set $X \subseteq \mathcal{A}$ of its entailed atoms. The satisfaction relation "$\models$" on rules r is inductively defined as follows:

- $X \models {\sim}B$ if $X \not\models B$,
- $X \models (a_1 \vee \cdots \vee a_k)$ if $\{a_1, \ldots, a_k\} \cap X \neq \emptyset$,
- $X \models (L\,\#\mathtt{sum}[\ell_1 = w_1, \ldots, \ell_k = w_k]\,U)$ if $L \leq \sum_{1\leq i\leq k, X\models \ell_i} w_i \leq U$,
- $X \models \mathit{body}(r)$ if $X \models \ell$ for all $\ell \in \mathit{body}(r)$, and
- $X \models r$ if $X \models \mathit{head}(r)$ or $X \not\models \mathit{body}(r)$.

A *logic program* Π is a set of rules r, and X is a *model* of Π if $X \models r$ for every $r \in \Pi$. The reduct of the head H of a rule r wrt X is $H^X = \{a_1 \vee \cdots \vee a_k\}$ if $H = a_1 \vee \cdots \vee a_k$, and $H^X = \mathit{atom}(H^+) \cap X$ if $H = L\,\#\mathtt{sum}[\ell_1 = w_1, \ldots, \ell_k = w_k]\,U$. Furthermore, the reduct of some (positive) body element $B \in \mathit{body}(r)^+$ is $B^X = B$ if $B \in \mathcal{A}$, and $B^X = \big(L - \sum_{1\leq i\leq k, \ell_i = {\sim}a_i, a_i \notin X} w_i\big)\,\#\mathtt{sum}\,B^+$ if $B = L\,\#\mathtt{sum}[\ell_1 = w_1, \ldots, \ell_k = w_k]\,U$. The *reduct* of Π wrt X is the following logic program:

$$\Pi^X = \big\{H \leftarrow B_1^X, \ldots, B_m^X \mid r \in \Pi, X \models \mathit{body}(r), H \in \mathit{head}(r)^X, \mathit{body}(r)^+ = \{B_1, \ldots, B_m\}\big\}.$$

That is, for all rules $r \in \Pi$ whose bodies are satisfied wrt X, the reduct is obtained by replacing $\#\mathtt{sum}$ constraints in heads with individual atoms belonging to X and by eliminating negative components in bodies, where lower bounds of residual $\#\mathtt{sum}$ constraints (with trivial upper bounds) are reduced accordingly. Finally, X is an *answer set* of Π if X is a model of Π such that no proper subset of X is a model of Π^X. In view of the latter condition, note that an answer set is a *minimal* model of its own reduct.

The definition of answer sets provided above applies to logic programs containing extended constructs ($\#\mathtt{sum}$ constraints) under "choice semantics" (Simons et al. 2002), while additionally allowing for disjunctions under minimal-model semantics (wrt a reduct). We use these features to embed extended constructs of an object program into a disjunctive meta-program, so that their combination yields optimal answer sets of the object program. To this end, we reinterpret $\#\mathtt{minimize}$ statements of the following form:

$$\#\mathtt{minimize}[\ell_1 = w_1@J_1, \ldots, \ell_k = w_k@J_k]. \tag{1}$$

Like with $\#\mathtt{sum}$ constraints, every ℓ_i is a literal and every w_i an integer weight for $1 \leq i \leq k$, while J_i additionally provides an integer *priority level*.[1] Priorities allow for representing a sequence of lexicographically ordered $\#\mathtt{minimize}$ objectives, where greater levels are more significant than smaller ones. By default, a $\#\mathtt{minimize}$ statement distinguishes optimal answer sets of a program Π in the following way. For any $X \subseteq \mathcal{A}$ and integer J, let Σ_J^X denote the sum of weights w over all occurrences of weighted literals $\ell = w@J$ in (1) such that $X \models \ell$. An answer set X of Π is dominated if there is an answer set Y of Π such that $\Sigma_J^Y < \Sigma_J^X$ and $\Sigma_{J'}^Y = \Sigma_{J'}^X$ for all $J' > J$, and optimal otherwise.

In the following, we assume that every logic program is accompanied with one (possibly empty) $\#\mathtt{minimize}$ statement of the form (1). Instead of the default semantics, we consider Pareto efficiency wrt priority levels J, weights w, and several distinct optimization

[1] Explicit priority levels are supported in recent versions of the grounder *gringo* (Gebser et al.). This avoids a dependency of priorities on input order, which is considered by *lparse* (Syrjänen) if several $\#\mathtt{minimize}$ statements are provided. Priority levels are also supported by *dlv* (Leone et al. 2006) in weak constraints. Furthermore, we admit negative weights in $\#\mathtt{minimize}$ statements, where they cannot raise semantic problems (cf. (Ferraris 2005)) going along with the rewriting of $\#\mathtt{sum}$ constraints suggested in (Simons et al. 2002).

criteria. In view of this, we use levels for inducing a lexicographic order, while weights are used for grouping literals (rather than summation). Pareto improvement then builds upon a two-dimensional structure of orderings among answer sets, induced by J and w. In turn, each such pairing is associated with some of the following orderings. By $Y \leq^w_J X$, we denote that the cardinality of the multiset of occurrences of $\ell = w@J$ in (1) such that $Y \models \ell$ is not greater than the one of the corresponding multiset for $X \models \ell$. Furthermore, we write $Y \subseteq^w_J X$ if, for any weighted literal $\ell = w@J$ occurring in (1), $Y \models \ell$ implies $X \models \ell$. Finally, we denote by $Y \preceq^w_J X$ that Y is preferable to X (Sakama and Inoue 2000) according to a (given) preference relation $\preceq$ among literals ℓ such that $\ell = w@J$ occurs in (1). Given a logic program Π and a collection M of relations of the form $\diamond^w_J$ for priority levels J, weights w, and $\diamond \in \{\leq, \subseteq, \preceq\}$, an answer set Y of Π dominates an answer set X of Π wrt M if there are a priority level J and a weight w such that $X \diamond^w_J Y$ does not hold for $\diamond^w_J \in M$, while $Y \diamond^{w'}_{J'} X$ holds for all $\diamond^{w'}_{J'} \in M$ where $J' \geq J$. In turn, an answer set X of Π is *optimal* wrt M if there is no answer set Y of Π that dominates X wrt M.

As an example, consider the following program, referred to by Π_0:

$$1\,\{p,t\} \leftarrow 1\,\{r,s,{\sim}t\}\,2. \tag{2}$$

$$\{q,r\}\,1 \leftarrow 1\,\{p,t\}. \tag{3}$$

$$s \leftarrow {\sim}q, {\sim}r. \tag{4}$$

This program has five answer sets, viz. $\{p,q\}$, $\{p,r\}$, $\{p,s\}$, $\{p,s,t\}$, and $\{s,t\}$. (Sets $\{a_1,\dots,a_k\}$ in (2) and (3) are used as shorthands for $\#\mathtt{sum}[a_1 = 1,\dots,a_k = 1]$.) In addition, let Π_1 denote the union of Π_0 with the following $\#\mathtt{minimize}$ statement:

$$\#\mathtt{minimize}[p = 1@1, q = 1@1, r = 1@1, s = 1@1]. \tag{5}$$

This statement specifies that all atoms of Π_0 except for t are subject to minimization. Passing Π_1 to *gringo* and an answer set solver like *smodels* yields the single $\leq^1_1$-minimal answer set $\{s,t\}$. Note, however, that Π_0 has three $\subseteq^1_1$-minimal answer sets, namely $\{p,q\}$, $\{p,r\}$, and $\{s,t\}$. They cannot be computed directly from Π_1 via any available ASP system.

We implement the complex optimization criteria described above by meta-interpretation in disjunctive ASP. For transparency, we provide meta-programs as true ASP code in the first-order input language of *gringo* (Gebser et al.), including `not` and `|` as tokens for $\sim$ and $\vee$, respectively, as well as $\{a_1,\dots,a_k\}$ as shorthand for $\#\mathtt{sum[}a_1\mathtt{=1,}\dots\mathtt{,}a_k\mathtt{=1]}$. Further constructs are informally introduced by need in the remainder of this paper. Note that our (disjunctive) meta-programs apply to an extended object program that does not include proper disjunctions (over more than one atom). Unless stated otherwise, we below use the term *extended* program to refer to a logic program without proper disjunctions.

3 Basic Meta-Modeling

For reinterpreting $\#\mathtt{minimize}$ statements by means of ASP, we take advantage of recent advances in ASP grounding, admitting an easy use of meta-modeling techniques. To be precise, we rely upon the unrestricted usage of function symbols and program reification as provided by *gringo* (Gebser et al.). The latter allows for turning an input program along with a $\#\mathtt{minimize}$ statement into facts representing the structure of their ground instantiation via fixed sets of predicates and function symbols.

```
1  rule(pos(sum(1,0,2)),pos(conjunction(0))). % 1 { p, t } :- 1 { r, s, not t } 2.
2  wlist(0,0,pos(atom(p)),1). wlist(0,1,pos(atom(t)),1).
3  set(0,pos(sum(1,1,2))).
4  wlist(1,0,pos(atom(r)),1). wlist(1,1,pos(atom(s)),1). wlist(1,2,neg(atom(t)),1).

6  rule(pos(sum(0,2,1)),pos(conjunction(1))). % { q, r } 1 :- 1 { p, t }.
7  wlist(2,0,pos(atom(q)),1). wlist(2,1,pos(atom(r)),1).
8  set(1,pos(sum(1,0,2))).

10 rule(pos(atom(s)),pos(conjunction(2))).    % s :- not q, not r.
11 set(2,neg(atom(q))). set(2,neg(atom(r))).

13 scc(0,pos(atom(p))). scc(0,pos(atom(r))). scc(0,pos(atom(t))).
14 scc(0,pos(conjunction(0))). scc(0,pos(sum(1,1,2))).
15 scc(0,pos(conjunction(1))). scc(0,pos(sum(1,0,2))).

17 minimize(1,3). % #minimize [ p = 1 @ 1, q = 1 @ 1, r = 1 @ 1, s = 1 @ 1 ].
18 wlist(3,0,pos(atom(p)),1). wlist(3,1,pos(atom(q)),1).
19 wlist(3,2,pos(atom(r)),1). wlist(3,3,pos(atom(s)),1).
```

Listing 1. Facts describing a reified extended logic program.

For illustrating the format output by *gringo*, consider the facts in Line 1–15 of Listing 1, obtained by calling *gringo* with option `--reify` on program Π_0. Let us detail the representation of the rule in (2) inducing the facts in Line 1–4. The predicate `rule/2` is used to link the rule head and body. By convention, both are positive rule elements, as indicated via the functor `pos/1`. Furthermore, the term `sum(1,0,2)` tells us that the head is a `#sum` constraint with lower bound `1` and (trivial) upper bound `2` over a list labeled `0` of weighted literals. In fact, the included literals are provided via the facts over `wlist/4` given in Line 2, whose first arguments are `0`. While the second arguments, `0` and `1`, are simply indexes (enabling the representation of duplicates in multisets), the third ones provide literals, `p` and `t`, each having the (default) weight `1`, as given in the fourth arguments. Again by convention, the body of each rule is a conjunction, where the term `conjunction(0)` in Line 1 refers to the set labeled `0`. Its single element, a positive `#sum` constraint with lower bound `1` and upper bound `2` over a list labeled `1`, is provided by the fact in Line 3. The corresponding weighted literals are described by the facts in Line 4; observe that the negative literal `not t` is represented in terms of the functor `neg/1`, applied to `atom(t)`. The rules in (3) and (4) are represented analogously in Line 6–8 and 10–11, respectively. It is still interesting to note that recurrences of lists of weighted literals (and sets) can reuse labels introduced before, as done in Line 8 by referring to `0`. In fact, *gringo* identifies repetitions of structural entities and reuses labels. In addition to the rules of Π_0, the elements of non-trivial strongly connected components of its positive dependency graph (cf. (6) below) are provided in Line 13–15. Albeit their usage is explained in the next section, note already that the members of the only such component, labeled `0`, include atoms as well as (positive) body elements, i.e., conjunctions and `#sum` constraints, connecting the component. Indeed, the existence of facts over `scc/2` tells us that Π_0 is not tight (cf. (Fages 1994)).

Now, we may compute all five answer sets of Π_0 (given in `p0.lp`) by combining the facts in Line 1–15 of Listing 1 with the basic meta-program in Listing 2 (`meta.lp`):[2]

```
gringo --reify p0.lp | gringo meta.lp - | clasp 0
```

[2] Following Unix customs, the minus symbol "-" stands for the output of "`gringo --reify p0.lp`."

```
1  % extract rule elements

3  litb(B) :- rule(_,B).
4  litb(E) :- litb(pos(conjunction(S))), set(S,E).
5  litb(E) :- eleb(sum(_,S,_)), wlist(S,_,E,_).

7  eleb(P) :- litb(pos(P)).
8  eleb(N) :- litb(neg(N)).

10 elem(E) :- eleb(E).
11 elem(E) :- rule(pos(E),_).
12 elem(P) :- rule(pos(sum(_,S,_)),_), wlist(S,_,pos(P),_).
13 elem(N) :- rule(pos(sum(_,S,_)),_), wlist(S,_,neg(N),_).

15 % generate answer set from reified rules

17 hold(conjunction(S)) :- eleb(conjunction(S)),
18                hold(P)      : set(S,pos(P)),
19            not hold(N)      : set(S,neg(N)).
20 hold(sum(L,S,U))     :- eleb(sum(L,S,U)),
21   L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
22            not hold(N) = W : wlist(S,Q,neg(N),W) ] U.

24 hold(atom(A))        :- rule(pos(atom(A)),   pos(B)), hold(B).
25 L #sum [     hold(P) = W : wlist(S,Q,pos(P),W),
26          not hold(N) = W : wlist(S,Q,neg(N),W) ] U
27                      :- rule(pos(sum(L,S,U)),pos(B)), hold(B).
28                      :- rule(pos(false),     pos(B)), hold(B).

30 % project output to atoms of answer set

32 #hide. #show hold(atom(A)).
```

Listing 2. Basic meta-program (`meta.lp`) for reified extended logic programs.

Each answer set of the meta-program applied to a reified program corresponds to an answer set of the reified program. More precisely, a set X of atoms is an answer set of the reified program iff the meta-program yields an answer set Y such that $X = \{a \mid \texttt{hold(atom(}a\texttt{))} \in Y\}$, e.g., `hold(atom(`$q$`))` stands for q. As indicated in the comments (preceded by `%`), our meta-program consists of three parts. Among the rule elements extracted in Line 3–13, only those occurring within bodies, identified via `eleb/1`, are relevant to the generation of answer sets specified in Line 17–28. (Additional head elements, given by `elem/1`, are of interest in the next section.) In fact, answer set generation follows the structure of reified programs, identifying conjunctions and `#sum` constraints that hold[3] to further derive atoms occurring in rule heads, either singular or within `#sum` constraints (cf. Line 24–27). Line 28 deals with integrity constraints represented via the constant `false` in heads of reified rules. The last part in Line 32 restricts the output of the meta-program's answer sets to the representations of original input atoms.

Finally, note that `meta.lp` does not inspect facts representing a reified `#minimize` statement, such as the ones in Line 17–19 of Listing 1 stemming from the statement in (5). Such facts over `minimize/2` provide a priority level as the first argument and the label of a list of weighted literals, like the ones referred to from within terms of functor `sum/3`, as the second argument. Rather than simply mirroring the standard meaning of `#minimize` statements (by encoding them analogously to rules; cf. Line 17–28 of Listing 2), we support

[3] The "`:`" connective expands to the list of all instances of its left-hand side such that corresponding instances of literals on the right-hand side hold (cf. (Syrjänen) and (Gebser et al.)).

flexible customizations. In fact, the next section presents our meta-programs implementing preference relations and Pareto efficiency, as described in the background.

4 Advanced Meta-Modeling

Given the reification of extended logic programs and the encoding of their answer sets in `meta.lp`, our approach to complex optimization is based on the idea that an answer set generated via `meta.lp` is optimal (and thus acceptable) only if it is not dominated by any other answer set. For implementing our approach, we exploit the capabilities of disjunctive ASP to compactly represent the space of all potential counterexamples, viz. answer sets dominating a candidate answer set at hand. To this end, we encode the subtasks of

1. guessing an answer set as a potential counterexample and
2. verifying that the counterexample dominates a candidate answer set.

A candidate answer set passes both phases if it turns out to be infeasible to guess a counterexample that dominates it. For expressing the non-existence of counterexamples, we make use of an error-indicating atom `bot` and saturation (Eiter and Gottlob 1995), deriving all atoms representing the space of counterexamples from `bot`. Since the semantics of disjunctive ASP is based on minimization, saturation makes sure that `bot` is derived only if it is inevitable, i.e., if it is impossible to construct a counterexample. However, via an integrity constraint, we can stipulate `bot` (and thus the non-existence of counterexamples) to hold, yet without providing any derivation of `bot`. In view of such a constraint and saturation, a successful candidate answer set is accompanied by all atoms representing counterexamples. Given that the reduct drops negative literals, the necessity that all atoms representing counterexamples are true implies that we cannot use their default negation in any meaningful way. Hence, we below encode potential counterexamples, i.e., answer sets of extended programs, and (non-)dominance of a candidate answer set in disjunctive ASP without taking advantage of default negation (used in `meta.lp`).

For encoding the first subtask of guessing a counterexample, we rely on a characterization of answer sets in terms of an *immediate consequence operator* $\mathcal{T}$ (cf. (Lloyd 1987)), defined as follows for a logic program Π and a set $X \subseteq \mathcal{A}$ of atoms: $\mathcal{T}_\Pi(X) = \{\mathit{head}(r) \mid r \in \Pi, X \models \mathit{body}(r)\}$. Furthermore, an iterative version of $\mathcal{T}$ can be defined in the following way: $\mathcal{T}_\Pi^0(X) = X$ and $\mathcal{T}_\Pi^{i+1}(X) = \mathcal{T}_\Pi^i(X) \cup \mathcal{T}_\Pi(\mathcal{T}_\Pi^i(X))$. In the context of an extended program Π, possibly including choice rules, default negation, and upper bounds of weight constraints, we are interested in the least fixpoint of $\mathcal{T}$ applied wrt the reduct Π^X. Since a fixpoint is reached in at most $|atom(\Pi)|$ applications of $\mathcal{T}$, where $atom(\Pi) \subseteq \mathcal{A}$ denotes the set of atoms occurring in Π, the least fixpoint is given by $\mathcal{T}_{\Pi^X}^{|atom(\Pi)|}(\emptyset)$. As pointed out in (Liu and You 2010), a model X of an extended program Π is an answer set of Π iff $\mathcal{T}_{\Pi^X}^{|atom(\Pi)|}(\emptyset) = X$. Furthermore, Liu and You (2010) show that X violates the loop formula of some atom or loop if X is a model, but not an answer set of Π. This property motivates a "localization" of $\mathcal{T}$ on the basis of (circular) positive dependencies.

The *(positive) dependency graph* of an extended program Π is given by the following pair of nodes and directed edges:

$$\big(atom(\Pi), \{(a, b) \mid r \in \Pi, a \in atom(\mathit{head}(r)^+), B \in \mathit{body}(r)^+, b \in atom(B^+)\}\big). \quad (6)$$

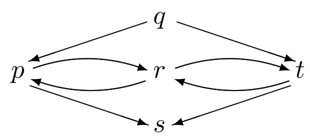
A strongly connected component (SCC) is a maximal subgraph of the dependency graph of Π such that all nodes are pairwisely connected via paths. An SCC is trivial if it does not contain any edge, and non-trivial otherwise. Note that the SCCs of the dependency graph of Π induce a partition of $atom(\Pi)$ such that every atom and every loop of Π is contained in some part. Hence, we can make use of the partition to apply $\mathcal{T}$ separately to each part.

Proposition 1
Let Π be an extended logic program, $C_1, \dots, C_k$ be the sets of atoms belonging to the SCCs of the dependency graph of Π, and $X \subseteq atom(\Pi)$.

Then, we have that $\mathcal{T}_{\Pi^X}^{|atom(\Pi)|}(\emptyset) = X$ iff $\bigcup_{1 \leq j \leq k}(\mathcal{T}_{\Pi^X}^{|C_j|}(X \setminus C_j) \cap C_j) = X$.

Proof (Sketch)
It is clear that $\mathcal{T}_{\Pi^X}^{|atom(\Pi)|}(\emptyset) = X$ implies $\mathcal{T}_{\Pi^X}^{|C_j|}(X \setminus C_j) \cap C_j = X \cap C_j$ for every $1 \leq j \leq k$. Hence, we only need to show that $\bigcup_{1 \leq j \leq k}(\mathcal{T}_{\Pi^X}^{|C_j|}(X \setminus C_j) \cap C_j) = X$ implies $\mathcal{T}_{\Pi^X}^{|atom(\Pi)|}(\emptyset) = X$. To this end, assume that $\bigcup_{1 \leq j \leq k}(\mathcal{T}_{\Pi^X}^{|C_j|}(X \setminus C_j) \cap C_j) = X$ and that $C_1, \dots, C_k$ are topologically ordered such that the dependency graph of Π does not contain any edge from an atom in C_j to atoms in $C_{j+1} \cup \dots \cup C_k$ for $1 \leq j \leq k$. That is, atoms in $C_{j+1} \cup \dots \cup C_k$ do not occur in rules $r \in \Pi^X$ such that $head(r) \in C_j$, so that $\mathcal{T}_{\Pi^X}^{|C_j|}(X \setminus C_j) \cap C_j = \mathcal{T}_{\Pi^X}^{|C_j|}(X \cap (C_1 \cup \dots \cup C_{j-1})) \cap C_j$ for $1 \leq j \leq k$. From $X = (X \cap C_1) \cup \dots \cup (X \cap C_k) = (\mathcal{T}_{\Pi^X}^{|C_1|}(\emptyset) \cap C_1) \cup \dots \cup (\mathcal{T}_{\Pi^X}^{|C_k|}(X \cap (C_1 \cup \dots \cup C_{k-1})) \cap C_k) \subseteq \mathcal{T}_{\Pi^X}^{|C_1| + \dots + |C_k|}(\emptyset) = \mathcal{T}_{\Pi^X}^{|atom(\Pi)|}(\emptyset) \subseteq X$, we then conclude that $\mathcal{T}_{\Pi^X}^{|atom(\Pi)|}(\emptyset) = X$. $\square$

For illustration, reconsider Π_0 in (2)–(4) and its dependency graph, looking as follows:



Observe that the sets $\{s\}$, $\{p, r, t\}$, and $\{q\}$ of atoms belong to SCCs (ordered topologically). Furthermore, let us take the following reducts into account:

$$\Pi_0^{\{p,r\}} = \begin{Bmatrix} p \leftarrow 0 \ \#\mathtt{sum}[r = 1, s = 1]. \\ r \leftarrow 1 \ \#\mathtt{sum}[p = 1, t = 1]. \end{Bmatrix} \quad \Pi_0^{\{r,t\}} = \begin{Bmatrix} t \leftarrow 1 \ \#\mathtt{sum}[r = 1, s = 1]. \\ r \leftarrow 1 \ \#\mathtt{sum}[p = 1, t = 1]. \end{Bmatrix}$$

We have that $\mathcal{T}_{\Pi_0^{\{p,r\}}}^{1}(\emptyset) = \{p\}$ and $\mathcal{T}_{\Pi_0^{\{p,r\}}}^{2}(\emptyset) = \{p\} \cup \mathcal{T}_{\Pi_0^{\{p,r\}}}(\{p\}) = \{p, r\} = \mathcal{T}_{\Pi_0^{\{p,r\}}}^{3}(\emptyset)$. Along with $\mathcal{T}_{\Pi_0^{\{p,r\}}}^{1}(\{p, r\}) \cap \{s\} = \mathcal{T}_{\Pi_0^{\{p,r\}}}^{1}(\{p, r\}) \cap \{q\} = \emptyset$, we obtain $(\mathcal{T}_{\Pi_0^{\{p,r\}}}^{1}(\{p, r\}) \cap \{s\}) \cup (\mathcal{T}_{\Pi_0^{\{p,r\}}}^{3}(\emptyset) \cap \{p, r, t\}) \cup (\mathcal{T}_{\Pi_0^{\{p,r\}}}^{1}(\{p, r\}) \cap \{q\}) = \{p, r\}$. In view of Proposition 1, this confirms that the model $\{p, r\}$ of Π_0 is an answer set of Π_0. On the other hand, $(\mathcal{T}_{\Pi_0^{\{r,t\}}}^{1}(\{r, t\}) \cap \{s\}) \cup (\mathcal{T}_{\Pi_0^{\{r,t\}}}^{3}(\emptyset) \cap \{p, r, t\}) \cup (\mathcal{T}_{\Pi_0^{\{r,t\}}}^{1}(\{r, t\}) \cap \{q\}) = \emptyset$ yields that the model $\{r, t\}$ of Π_0 is not an answer set of Π_0.

In a nutshell, our encoding of answer sets (as counterexamples) in disjunctive ASP combines the following parts:

1. guessing an interpretation,
2. deriving the error-indicating atom `bot` if the interpretation is not a supported model (where each true atom occurs positively in the head of some rule whose body holds),

3. deriving `bot` if the true atoms of some *non-trivial* SCC are not acyclicly derivable (checked via determining the complement of a fixpoint of $\mathcal{T}$), and
4. saturating interpretations that do not correspond to answer sets by deriving all truth assignments (for atoms) from `bot`.

Note that the third part, checking acyclic derivability, concentrates on atoms of non-trivial SCCs, while checking support in the second part is already sufficient for trivial SCCs.

The meta-program in Listing 3 implements the sketched idea. In the following, we concentrate on describing its crucial features. For evaluating support, the meta-rules in Line 3 and 4 collect atoms having a positive occurrence in the head of a rule along with the rule's body. Note that, for atoms contained in a `#sum` constraint in the head, the associated bounds and weights are inessential in the context of support. On the other hand, the meta-rule in Line 6 sums the weights of all literals in a `#sum` constraint; this is needed to evaluate bounds in the sequel, where (non-reified) default negation and upper bounds (acting negatively) are inapplicable in view of saturation.

The meta-rules in Line 10–29 generate an interpretation by guessing some truth value for each atom (Line 10) and evaluating further constructs occurring in a reified program accordingly (Line 12–29). While the special constant `false` (used as head of integrity constraints) holds in no interpretation (`fail(false)` is a fact) and the evaluation of conjunctions is straightforward, more care is required for evaluating `#sum` constraints. For instance, the case that a `#sum` constraint holds is in the meta-rule in Line 19–23 identified via sufficiently many literals that hold to achieve the lower bound `L` and also sufficiently many literals that do not hold to fill the gap between the upper bound `U` and the sum `T` of all weights. Note that the latter condition is encoded by the lower bound `T-U`, rather than taking `U` as an upper bound (as done in `meta.lp`). The complementary cases that a `#sum` constraint does not hold are described in the same manner in Line 24–29, where the lower bound `T-L+1` (or `U+1`) for weights of literals that do not hold (or hold) is used to indicate a violated lower (or upper) bound of the reified `#sum` constraint.

Given an interpretation of atoms and the corresponding truth values of further constructs in an extended program, the meta-rules in Line 33 and 34 are used to derive `bot` if the interpretation does not provide us with a supported model. To avoid such a derivation of `bot`, every rule of the reified program must be satisfied, and every true atom must have a positive occurrence in the head of some rule whose body holds.

It remains to check the acyclic derivability of atoms belonging to non-trivial SCCs. To this end, the meta-rule in Line 38 determines the number `Z` of atoms in an SCC labeled `C` as the maximum step at which a fixpoint of $\mathcal{T}$, applied locally to `C`, is reached. Furthermore, the meta-rule in Line 40–41 derives `sccw(A)` if the atom referred to by `A` does not have a derivation external to `C`. (Recall that the positive body elements of rules internally connecting an SCC, i.e., rules contributing the SCC's edges to the dependency graph, are marked by facts over `scc/2`; cf. Listing 1.) The acyclic derivability of atoms indicated by `sccw(A)` is of particular interest in the sequel. In fact, our encoding identifies the complement of a fixpoint of $\mathcal{T}$ in terms of atoms `A` for which `wait(atom(A),Z)` is derived. To accomplish this, the meta-rule in Line 45 marks all atoms of `C` as underived at step `0`. As encoded via the meta-rule in Line 46–47, an atom `A` stays underived at a later step `D` if there is no external derivation of `A` (`sccw(A)` holds) and the bodies `B` of all component-

```
1  % extract supports of atoms and sums of weight lists' weights

3  supp(atom(A),B) :- rule(pos(atom(A)),   pos(B)).
4  supp(atom(A),B) :- rule(pos(sum(_,S,_)),pos(B)), wlist(S,_,pos(atom(A)),_).

6  sum(S,T)        :- elem(sum(_,S,_)), T = #sum [ wlist(S,Q,_,W) = W ].

8  % generate interpretation

10 true(atom(A)) | fail(atom(A)) :- elem(atom(A)).

12 fail(false).

14 true(conjunction(S)) :- elem(conjunction(S)),
15                         true(P) : set(S,pos(P)), fail(N) : set(S,neg(N)).
16 fail(conjunction(S)) :- elem(conjunction(S)), set(S,pos(P)), fail(P).
17 fail(conjunction(S)) :- elem(conjunction(S)), set(S,neg(N)), true(N).

19 true(sum(L,S,U))     :- elem(sum(L,S,U)), sum(S,T),
20                         L     #sum [ true(P) = W : wlist(S,Q,pos(P),W),
21                                      fail(N) = W : wlist(S,Q,neg(N),W) ],
22                         T-U   #sum [ fail(P) = W : wlist(S,Q,pos(P),W),
23                                      true(N) = W : wlist(S,Q,neg(N),W) ].
24 fail(sum(L,S,U))     :- elem(sum(L,S,U)), sum(S,T),
25                         T-L+1 #sum [ fail(P) = W : wlist(S,Q,pos(P),W),
26                                      true(N) = W : wlist(S,Q,neg(N),W) ].
27 fail(sum(L,S,U))     :- elem(sum(L,S,U)),
28                         U+1   #sum [ true(P) = W : wlist(S,Q,pos(P),W),
29                                      fail(N) = W : wlist(S,Q,neg(N),W) ].

31 % verify supported model properties

33 bot :- rule(pos(H),pos(B)), true(B), fail(H).
34 bot :- true(atom(A)), fail(B) : supp(atom(A),B).

36 % verify acyclic derivability

38 step(C,Z)                :- scc(C,_), Z = #sum [ scc(C,pos(atom(A))) ].

40 sccw(A)                  :- scc(C,pos(atom(A))),
41                             fail(B) : supp(atom(A),B) : not scc(C,pos(B)).

43 wait(E,D-1)              :- scc(C,pos(E)), fail(E), step(C,Z), D = 1..Z.

45 wait(atom(A),0)          :- scc(C,pos(atom(A))).
46 wait(atom(A),D)          :- scc(C,pos(atom(A))), sccw(A), step(C,Z), D = 1..Z,
47                             wait(B,D-1) : supp(atom(A),B) : scc(C,pos(B)).

49 wait(sum(L,S,U),D-1) :- scc(C,pos(sum(L,S,U))), sum(S,T), step(C,Z), D = 1..Z,
50        T-L+1 #sum [ fail(P) = W : wlist(S,Q,pos(P),W) : not scc(C,pos(P)),
51                 wait(P,D-1) = W : wlist(S,Q,pos(P),W) :     scc(C,pos(P)),
52                     true(N) = W : wlist(S,Q,neg(N),W) ].

54 wait(conjunction(S),D-1) :- scc(C,pos(conjunction(S))), set(S,pos(P)),
55                             scc(C,pos(P)), wait(P,D-1), step(C,Z), D = 1..Z.

57 bot :- scc(C,pos(atom(A))), true(atom(A)), wait(atom(A),Z), step(C,Z).

59 % saturate interpretations that are not answer sets

61 true(atom(A)) :- elem(atom(A)), bot.
62 fail(atom(A)) :- elem(atom(A)), bot.
```

Listing 3. Disjunctive meta-program (`metaD.lp`) for reified extended logic programs.

internal supports of `A` are yet underived at step `D-1` (`wait(B,D-1)` holds). The latter is checked via the meta-rules in Line 49–52 and 54–55, respectively. The former applies to `#sum` constraints and identifies cases where the weights of literals that do not hold along

```
1  step(0,3).  sccw(p). sccw(r). sccw(t).

3  wait(atom(p),        0;1;2) :- fail(atom(p)).
4  wait(atom(r),        0;1;2) :- fail(atom(r)).
5  wait(atom(t),        0;1;2) :- fail(atom(t)).
6  wait(conjunction(0),0;1;2) :- fail(conjunction(0)).
7  wait(sum(1,1,2),     0;1;2) :- fail(sum(1,1,2)).
8  wait(conjunction(1),0;1;2) :- fail(conjunction(1)).
9  wait(sum(1,0,2),     0;1;2) :- fail(sum(1,0,2)).

11 wait(atom(p),0).
12 wait(atom(p),1) :- sccw(p), wait(conjunction(0),0).
13 wait(atom(p),2) :- sccw(p), wait(conjunction(0),1).
14 wait(atom(p),3) :- sccw(p), wait(conjunction(0),2).

16 wait(atom(r),0).
17 wait(atom(r),1) :- sccw(r), wait(conjunction(1),0).
18 wait(atom(r),2) :- sccw(r), wait(conjunction(1),1).
19 wait(atom(r),3) :- sccw(r), wait(conjunction(1),2).

21 wait(atom(t),0).
22 wait(atom(t),1) :- sccw(t), wait(conjunction(0),0).
23 wait(atom(t),2) :- sccw(t), wait(conjunction(0),1).
24 wait(atom(t),3) :- sccw(t), wait(conjunction(0),2).

26 wait(sum(1,1,2),0) :- 3 #sum [ fail(atom(s)), wait(atom(r),0), true(atom(t)) ].
27 wait(sum(1,1,2),1) :- 3 #sum [ fail(atom(s)), wait(atom(r),1), true(atom(t)) ].
28 wait(sum(1,1,2),2) :- 3 #sum [ fail(atom(s)), wait(atom(r),2), true(atom(t)) ].

30 wait(sum(1,0,2),0) :- 2 #sum [ wait(atom(p),0), wait(atom(t),0) ].
31 wait(sum(1,0,2),1) :- 2 #sum [ wait(atom(p),1), wait(atom(t),1) ].
32 wait(sum(1,0,2),2) :- 2 #sum [ wait(atom(p),2), wait(atom(t),2) ].

34 wait(conjunction(0),0) :- wait(sum(1,1,2),0).
35 wait(conjunction(0),1) :- wait(sum(1,1,2),1).
36 wait(conjunction(0),2) :- wait(sum(1,1,2),2).

38 wait(conjunction(1),0) :- wait(sum(1,0,2),0).
39 wait(conjunction(1),1) :- wait(sum(1,0,2),1).
40 wait(conjunction(1),2) :- wait(sum(1,0,2),2).

42 bot :- true(atom(p)), wait(atom(p),3).
43 bot :- true(atom(r)), wait(atom(r),3).
44 bot :- true(atom(t)), wait(atom(t),3).
```

Listing 4. (Simplified) ground rules obtained from meta-rules in Line 38–57 of `metaD.lp`.

with the ones of yet underived atoms of `C` exceed `T-L`, so that the lower bound `L` is not yet established. Similarly, the underivability of a conjunction is recognized via a yet underived positive body element internal to the component `C`. Also note that the falsity of elements of `C` is propagated via the meta-rule in Line 43, so that false atoms, `#sum` constraints, and conjunctions do not contribute to derivations of atoms of `C`. As mentioned above, the complement of a fixpoint of $\mathcal{T}$ contains the atoms `A` such that `wait(atom(A),Z)` is eventually derived. If any such atom `A` is true, failure to construct an answer set is indicated by deriving `bot` via the meta-rule in Line 57.

For illustration, consider the ground rules shown in Listing 4, which are obtained for the SCC labeled `0` in Listing 1. For the answer set $\{p,r\}$ of Π_0, represented by the atoms `true(atom(p))`, `true(atom(r))`, `fail(atom(q))`, `fail(atom(s))`, and `fail(atom(t))`, we have that `wait(sum(1,1,2), 0;1;2)` and `wait(conjunction(0),0;1;2)` are underivable via the rules in Line 26–28 and 34–36, respectively. Thus, `wait(atom(p),1;2;3)` are not derived via the rules in Line 12–14, so that `wait(sum(1,0,2),1;2)` and wait(conjunction(1),1;2) are underivable in turn via the rules in Line 31–32 and 39–40, respectively. As a consequence, wait(atom(r),2;3) are not derived via the rules in Line 18–19. We have thus checked that none of the rules in Line 42–44 allows for deriving bot, which tells us that the true atoms p and r are acyclicly derivable. On the other hand, for the interpretation $\{r,t\}$, given by true(atom(r)), true(atom(t)), fail(atom(p)), fail(atom(q)), and fail(atom(s)), we have that the atoms wait(atom(p),0;1;2), wait(sum(1,1,2),0;1;2), wait(sum(1,0,2), 0;1;2), wait(conjunction(0),0;1;2), wait(conjunction(1),0;1;2), wait(atom(r),0;1;2;3), and wait(atom(t),0;1;2;3) are derived in turn via the rules in Line 3 and 16–40. From true(atom(r)) and wait(atom(r),3) as well as true(atom(t)) and wait(atom(t),3), we further derive the error-indicating atom bot via the rules in Line 43 and 44. This signals that the true atoms r and t are *not* acyclicly derivable, so that $\{r,t\}$ is not an answer set of Π_0.

Finally, saturation of interpretations that do not correspond to answer sets is accomplished via the meta-rules in Line 61 and 62 of Listing 3. They make sure that bot is included in an answer set of the meta-program only if it is inevitable wrt every interpretation. When considering the encoding part in Listing 3 in isolation, it like meta.lp describes answer sets of a reified program, and bot is derived only if there is no such answer set.

Our meta-programs meta.lp and metaD.lp in Listing 2 and 3 have not yet considered facts minimize(J,S) in reified programs, reflecting input #minimize statements. In fact, complex optimization is addressed by the meta-program metaO.lp, whose first part is shown in Listing 5. It allows for separate optimization criteria per priority level J and weight W (in facts wlist(S,Q,E,W)). Particular criteria can be provided via the user predicate optimize(J,W,O), where the values card, incl, and pref for O refer to minimality regarding cardinality, inclusion, and preference (Sakama and Inoue 2000), respectively, among the involved literals E. Such criteria are reflected via instances of cxopt(J,W,O), derived via the rules in Line 7 and 8–9, where card is taken by default if no criterion is provided by the user. At each priority level J, Pareto improvement of a counterexample (constructed via the rules in metaD.lp) over all weights W and criteria O such that cxopt(J,W,O) holds is used for deciding whether a candidate answer set (constructed via the rules in meta.lp) is optimal. To this end, similarity at a priority level J is indicated by deriving equal(J) from equal(J,W,O) over all instances of cxopt(J,W,O) via the rule in Line 13. Furthermore, the rules in Line 15–19 are used to chain successive priority levels, where a greater level J1 is more significant than its smaller neighbor J2, and to signal whether a priority level J2 is taken into account. The latter is the case if equal(J1) has been derived at all more significant priority levels J1. If it turns out that a candidate answer set is not refuted by a dominating counterexample, we derive bot via the rules in Line 21, 22, and 23: the first rule applies if there are no optimization criteria at all, the second one checks whether the counterexample is worse (or incomparable), as indicated by worse(J1) at an inspected priority level J1, and the third one detects lack of Pareto improvement from equality at the lowest priority level. Finally, the integrity constraint in Line 27 stipulates bot to hold. Along with saturation (in metaD.lp), this implies that a candidate answer set (constructed via the rules in meta.lp) is accepted only if there is no dominating counterexample, thus selecting exactly the optimal answer sets of an input program. The described rules serve the general

```
% extract (complex) optimization criteria per priority level and weight
% (relative to user predicate optimize/3; cardinality taken by default;
%  Pareto improvement over weights used for comparison at a priority level)

cxopt(card).  cxopt(incl).  cxopt(pref).

cxopt(J,W,O)    :- minimize(J,S), wlist(S,_,_,W), cxopt(O), optimize(J,W,O).
cxopt(J,W,card) :- minimize(J,S), wlist(S,_,_,W),
                   not optimize(J,W,O) : optimize(J,W,O).

% verify dominance

equal(J)     :- cxopt(J,_,_), equal(J,W,O) : cxopt(J,W,O).

chain(J1,J2) :- cxopt(J1;J2,_,_), J2 < J1,
                not cxopt(J3,W,O) : cxopt(J3,W,O) : J2 < J3 : J3 < J1.

check(J2)    :- cxopt(J2,_,_), not chain(J1,J2) : chain(J1,J2).
check(J2)    :- chain(J1,J2), check(J1), equal(J1).

bot          :- not cxopt(J,W,O) : cxopt(J,W,O).
bot          :- check(J1), worse(J1).
bot          :- check(J1), equal(J1), not chain(J1,J2) : chain(J1,J2).

% require non-existence of dominating answer set

:- not bot.

% check cardinality criteria

index(S,Q)         :- cxopt(J,_,card), minimize(J,S), wlist(S,Q,_,_),
                         not wlist(S,Q+1,E,W) : wlist(S,Q+1,E,W).

count(S,W,-1, 0)   :- cxopt(J,W,card), minimize(J,S).
count(S,W,Q+1,I)   :- count(S,W,Q,I), wlist(S,Q+1,_,_).
count(S,W,Q+1,I+1) :- count(S,W,Q,I), wlist(S,Q+1,pos(P),W),     hold(P).
count(S,W,Q+1,I+1) :- count(S,W,Q,I), wlist(S,Q+1,neg(N),W), not hold(N).

cdown(S,W,Q,  I)   :- count(S,W,Q,I), index(S,Q), not count(S,W,Q,I+1).
cdown(S,W,Q-1,I)   :- cdown(S,W,Q,I), wlist(S,Q,_,_).
cdown(S,W,Q-1,I-1) :- cdown(S,W,Q,I), wlist(S,Q,pos(P),W), true(P), 0 <= I.
cdown(S,W,Q-1,I-1) :- cdown(S,W,Q,I), wlist(S,Q,neg(N),W), fail(N), 0 <= I.

equal(J,W,card)    :- cxopt(J,W,card), minimize(J,S), cdown(S,W,-1,0).

worse(J)           :- cxopt(J,W,card), minimize(J,S), cdown(S,W,-1,-1).

% check inclusion criteria

ndiff(pos(P))   :- cxopt(J,W,incl), minimize(J,S), wlist(S,_,pos(P),W),
                   true(P).
ndiff(pos(P))   :- cxopt(J,W,incl), minimize(J,S), wlist(S,_,pos(P),W),
                            not hold(P).
ndiff(neg(N))   :- cxopt(J,W,incl), minimize(J,S), wlist(S,_,neg(N),W),
                   fail(N).
ndiff(neg(N))   :- cxopt(J,W,incl), minimize(J,S), wlist(S,_,neg(N),W),
                                hold(N).

equal(J,W,incl) :- cxopt(J,W,incl), minimize(J,S), ndiff(E) : wlist(S,_,E,W).

worse(J)        :- cxopt(J,W,incl), minimize(J,S), wlist(S,_,pos(P),W),
                   true(P), not hold(P).
worse(J)        :- cxopt(J,W,incl), minimize(J,S), wlist(S,_,neg(N),W),
                   fail(N),     hold(N).
```

Listing 5. Meta-program for complex optimization (`metaO.lp`) on reified logic programs.

purpose of identifying undominated answer sets, and the remainder of `metaO.lp` defines `equal(J,W,O)` and `worse(J)` relative to particular optimization criteria.

The rules in Line 31–46 implement the comparison of cardinalities between a candidate answer set and a counterexample at a priority level `J` over the literals indicated by a weight `W` (in facts `wlist(S,Q,E,W)`). To this end, we first count the number of such literals `E` that hold wrt the candidate answer set via the rules in Line 34–37. The outcome, i.e., the greatest number `I` such that `count(S,W,Q,I)` is derived at the maximum index `Q` of the list `S` given in `minimize(J,S)`, is identified via the rule in Line 39 and taken as the starting value for counting down the literals that hold wrt the counterexample. Note that this approach omits a (quadratic) comparison between the precise cardinalities obtained wrt the candidate answer set and the counterexample. The result of counting down, accomplished via the rules in Line 40–42, is given by the derived instances of `cdown(S,W,-1,I)`, where two outcomes deserve particular attention. In fact, `cdown(S,W,-1,0)` indicates that at least as many literals as wrt the candidate answer set hold wrt the counterexample, so that `equal(J,W,card)` is derived via the rule in Line 44. On the other hand, `cdown(S,W,-1,-1)` signals that more literals hold wrt the counterexample; as this makes a Pareto improvement at priority level `J` impossible, we derive `worse(J)` via the rule in Line 46. Finally, note that we did not directly encode minimization of cost functions with coefficients different from 1. In view of the multiset semantics supported by *gringo*, such coefficients can still be represented by including the desired number of duplicates of a literal in an input `#minimize` statement.

The second optimization criterion, inclusion (indicated via `cxopt(J,W,incl)`), is implemented by the rules in Line 50–64. The test for equality, attested by deriving `equal(J,W,incl)` via the rule in Line 59, is accomplished by checking whether a candidate answer set and a (comparable) counterexample agree on all involved literals `E`; otherwise, `ndiff(E)` is not derived via the rules in Line 50–57. Furthermore, the counterexample is incomparable to the candidate answer set if it includes some literal not shared by the latter; in such a case, `worse(J)` is derived via the rules in Line 61–62 and 63–64. In fact, the three $\subseteq_1^1$-minimal answer sets of Π_1 (given in `p1.lp`), consisting of the rules in (2)–(4) and the `#minimize` statement in (5) can now be computed in the following way:

```
gringo --reify p1.lp | gringo meta.lp metaD.lp metaO.lp \
  <(echo "optimize(1,1,incl).") - | claspD 0
```

Observe that *claspD* (Drescher et al. 2008), the disjunctive extension of *clasp* (Gebser et al. 2007), is used for solving the proper disjunctive ground program obtained from *gringo*.

The third optimization criterion implemented in `metaO.lp` (Listing 6) relies on literal preferences according to (Sakama and Inoue 2000), i.e., a relation $\ell_1 \preceq \ell_2$ provided via the user predicate `prefer/2`. In a nutshell, an answer set X_1 is preferable to another answer set X_2, i.e., $X_1 \preceq X_2$, if there is a preference $\ell_1 \preceq \ell_2$ such that ℓ_1 holds wrt X_1, but not wrt X_2, and ℓ_2 holds wrt X_2, but not wrt X_1, while $\ell_2' \not\preceq \ell_1$ or $\ell_1 \preceq \ell_2'$ applies to every literal ℓ_2' that holds wrt X_2, but not wrt X_1. The idea of the encoding part in Listing 6 is to derive `equal(J,W,pref)` if a candidate answer set X_1 is preferable to a counterexample X_2, and `worse(J)` if X_2 is not preferable to X_1. If both `equal(J,W,pref)` and `worse(J)` are underivable, it shows that X_2 is preferable to X_1, but not vice versa, so that X_1 is dominated by X_2 wrt the projection of the preference relation specified via `prefer/2` to literals `E` qualified by `minimize(J,S)` and `wlist(S,_,E,W)`. To implement these checks, the rules in Line 68–92 identify literals at which a candidate answer

```
66  % check preference criteria (relative to user predicate prefer/2)

68  cando(pos(P))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,pos(P),W),
69                      fail(P),     hold(P).
70  cando(neg(N))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,neg(N),W),
71                      true(N), not hold(N).
72  nocan(pos(P))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,pos(P),W),
73                      true(P).
74  nocan(pos(P))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,pos(P),W),
75                               not hold(P).
76  nocan(neg(N))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,neg(N),W),
77                      fail(N).
78  nocan(neg(N))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,neg(N),W),
79                                   hold(N).

81  condo(pos(P))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,pos(P),W),
82                      true(P), not hold(P).
83  condo(neg(N))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,neg(N),W),
84                      fail(N),     hold(N).
85  nocon(pos(P))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,pos(P),W),
86                      fail(P).
87  nocon(pos(P))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,pos(P),W),
88                                   hold(P).
89  nocon(neg(N))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,neg(N),W),
90                      true(N).
91  nocon(neg(N))    :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,neg(N),W),
92                               not hold(N).

94  cando(S,W,E)     :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,E;E1,W),
95                      prefer(E,E1), E1 != E, cando(E), condo(E1).

97  nocon(S,W,E)     :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,E,W), nocon(E).
98  nocon(S,W,E)     :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,E,W),
99                      nocan(E1) : wlist(S,_,E1,W) : E1 != E : prefer(E,E1).
100 nocon(S,W,E)     :- cxopt(J,W,pref), minimize(J,S), wlist(S,_,E;E2,W),
101                     prefer(E2,E), not prefer(E,E2), cando(E2).

103 equal(J,W,pref) :- cxopt(J,W,pref), minimize(J,S), cando(S,W,E),
104                     nocon(E2) : wlist(S,_,E2,W) : prefer(E2,E) : not prefer(E,E2).

106 worse(J)         :- cxopt(J,W,pref), minimize(J,S), nocon(S,W,E) : wlist(S,_,E,W).
```

Listing 6. Meta-program for literal preferences (`meta0.lp`) on reified logic programs.

set X_1 and a counterexample X_2 differ. The literals `E` that hold wrt X_1, but not wrt X_2, are indicated by `cando(E)`, while `nocan(E)` is derived otherwise. Similarly, `condo(E)` expresses that `E` holds wrt X_2, but not wrt X_1, whereas `nocon(E)` indicates that this is not the case. The rule in Line 94–95 further derives `cando(S,W,E)` if `E` is a literal that holds wrt X_1 only such that `E` $\preceq$ `E1` for a literal `E1` that holds wrt X_2 only. If such a literal `E` is not defeated by another literal `E2` satisfying `E2` $\preceq$ `E` as well as `E` $\not\preceq$ `E2` and holding wrt X_2 only, we derive `equal(J,W,pref)` via the rule in Line 103–104. On the other hand, a literal `E` provides no evidence for the counterexample X_2 being preferable to the candidate answer set X_1 if `E` does not hold wrt X_2 only, if no `E1` such that `E` $\preceq$ `E1` holds wrt X_1 only, or if some `E2` such that `E2` $\preceq$ `E` and `E` $\not\preceq$ `E2` holds wrt X_1 only. If some of these cases applies, `nocon(S,W,E)` is derived via the rules in Line 97–101. Given this, we further derive `worse(J)` via the rule in Line 106 if none of the literals `E` qualified by `wlist(S,_,E,W)` witnesses that X_2 is preferable to X_1. Finally, recall that the counterexample X_2 dominates the candidate answer set X_1 if X_2 is preferable to X_1, but not vice versa, i.e., if neither `equal(J,W,pref)` nor `worse(J)` is derivable in view of `cxopt(J,W,pref)`.

For showing the correctness of our meta-programming approach, we make use of the property that any answer set generated via `meta.lp` and `metaD.lp` encapsulates a pair of answer sets of the reified input program.

Lemma 1
Let F be the set of facts in the reification of an extended logic program Π.

Then, X and Y are answer sets of Π iff there is a unique answer set Z of $F \cup \{\texttt{meta.lp}\} \cup \{\texttt{metaD.lp}\}$ such that $X = \{a \mid \texttt{hold(atom(}a\texttt{))} \in Z\}$ and $Y = \{a \mid \texttt{true(atom(}a\texttt{))} \in Z\}$.

Proof (Sketch)
Given that the meta-rules in `meta.lp` merely reconstruct Π from its reification, we have that X is an answer set of Π iff there is a unique answer set X' of $F \cup \{\texttt{meta.lp}\}$ such that $X = \{a \mid \texttt{hold(atom(}a\texttt{))} \in X'\}$, where X' additionally includes F, atoms over `hold/1` for body elements of Π that hold wrt X, as well as auxiliary atoms over `litb/1`, `eleb/1`, and `elem/1`. Along with the fact that the (ground instances of the) rules in `metaD.lp` are invariant under reduct, do not include any integrity constraint, and do not define any predicates occurring in $F \cup \{\texttt{meta.lp}\}$, any answer set X' of $F \cup \{\texttt{meta.lp}\}$ can be extended to an answer set $Z = X' \cup Y'$ of $F \cup \{\texttt{meta.lp}\} \cup \{\texttt{metaD.lp}\}$, where Y' is a minimal model of F, the auxiliary rules in Line 3–13 of `meta.lp`, and `metaD.lp`. In view of the (proper) disjunctive rule in Line 10 of `metaD.lp`, for each $a \in \mathit{atom}(\Pi)$, we have that $\texttt{true(atom(}a\texttt{))} \in Y'$ or $\texttt{fail(atom(}a\texttt{))} \in Y'$, while saturation in Line 61–62 of `metaD.lp` admits $\texttt{bot} \in Y'$ only if `bot` is necessarily derived via the rules in Line 33, 34, and 57 of `metaD.lp`. As a consequence, the rules in `metaD.lp` check for the existence of a model Y' such that $Y = \{a \mid \texttt{true(atom(}a\texttt{))} \in Y'\}$ is an answer set of Π. In particular, the rules in Line 33 and 34 of `metaD.lp` derive `bot` wrt interpretations that are not models of Π or include some atom that does not occur in the head of any rule of Π whose body holds. Moreover, the rule in Line 57 of `metaD.lp` derives `bot` wrt interpretations containing some atom of a non-trivial SCC of the dependency graph of Π that does not belong to the least fixpoint of $\mathcal{T}$ applied to the SCC. In view of Proposition 1, we conclude that Y' is a minimal model (more precisely, an answer set) of F, the rules in Line 3–13 of `meta.lp`, and `metaD.lp` such that $\texttt{bot} \notin Y'$ iff Y is an answer set of Π. Hence, we have that X and Y are answer sets of Π iff there is an answer set Z of $F \cup \{\texttt{meta.lp}\} \cup \{\texttt{metaD.lp}\}$ such that $X = \{a \mid \texttt{hold(atom(}a\texttt{))} \in Z\}$ and $Y = \{a \mid \texttt{true(atom(}a\texttt{))} \in Z\}$. Finally, any such answer set Z of $F \cup \{\texttt{meta.lp}\} \cup \{\texttt{metaD.lp}\}$ is unique since all atoms in Z that are not of the form $\texttt{hold(atom(}a\texttt{))}$ or $\texttt{true(atom(}a\texttt{))}$ follow deterministically once the interpretation of these atoms is fixed. $\square$

We are now ready to show our main correctness result wrt the specification of optimal answer sets given in the background.

Theorem 1
Let F be the set of facts in the reification of an extended logic program Π and a statement $\#\texttt{minimize}[\ell_1 = w_1@J_1, \dots, \ell_k = w_k@J_k]$ such that $\{\ell_1, \dots, \ell_k\} \subseteq \mathit{atom}(\Pi) \cup \{{\sim}a \mid a \in \mathit{atom}(\Pi)\}$, $M \subseteq \{\leq^{w_i}_{J_i}, \subseteq^{w_i}_{J_i}, \preceq^{w_i}_{J_i} \mid 1 \leq i \leq k\}$, and $\preceq\ \subseteq (\mathit{atom}(\Pi) \cup \{{\sim}a \mid a \in \mathit{atom}(\Pi)\}) \times (\mathit{atom}(\Pi) \cup \{{\sim}a \mid a \in \mathit{atom}(\Pi)\})$.

Then, X is an optimal answer set of Π wrt M iff there is a unique answer set Z of

$$\begin{aligned}\Pi_{\text{meta}} = F &\cup \{\texttt{meta.lp}\} \cup \{\texttt{metaD.lp}\} \cup \{\texttt{metaO.lp}\}\\ &\cup \{\texttt{optimize(}J\texttt{,}w\texttt{,card).} \mid \leq_J^w \in M\}\\ &\cup \{\texttt{optimize(}J\texttt{,}w\texttt{,incl).} \mid \subseteq_J^w \in M\}\\ &\cup \{\texttt{optimize(}J\texttt{,}w\texttt{,pref).} \mid \preceq_J^w \in M\}\\ &\cup \{\texttt{prefer(}\ell'_1\texttt{,}\ell'_2\texttt{).} \mid \ell_1 \preceq \ell_2\}\end{aligned}$$

such that $X = \{a \mid \texttt{hold(atom(}a\texttt{))} \in Z\}$, where $\ell' = \texttt{pos(atom(}\ell\texttt{))}$ if $\ell \in \mathit{atom}(\Pi)$ and, respectively, $\ell' = \texttt{neg(atom(}a\texttt{))}$ if $\ell = {\sim}a$ for $a \in \mathit{atom}(\Pi)$.

Proof (Sketch)
By Lemma 1, we have that X and Y are answer sets of Π iff there is a (unique) answer set Z' of $F \cup \{\texttt{meta.lp}\} \cup \{\texttt{metaD.lp}\}$ such that $X = \{a \mid \texttt{hold(atom(}a\texttt{))} \in Z'\}$ and $Y = \{a \mid \texttt{true(atom(}a\texttt{))} \in Z'\}$. Since $\{\texttt{metaO.lp}\} \setminus \{\texttt{:- not bot.}\}$ is stratified, there is a unique answer set Z of $Z' \cup (\{\texttt{metaO.lp}\} \setminus \{\texttt{:- not bot.}\}) \cup \{\texttt{optimize(}J\texttt{,}w\texttt{,card).} \mid \leq_J^w \in M\} \cup \{\texttt{optimize(}J\texttt{,}w\texttt{,incl).} \mid \subseteq_J^w \in M\} \cup \{\texttt{optimize(}J\texttt{,}w\texttt{,pref).} \mid \preceq_J^w \in M\} \cup \{\texttt{prefer(}\ell'_1\texttt{,}\ell'_2\texttt{).} \mid \ell_1 \preceq \ell_2\}$. In the following, we argue that, for any $\diamond_J^w \in M$ and $\texttt{O} = \texttt{card/incl/pref}$ if $\diamond = \leq/\subseteq/\preceq$, we have that $X \diamond_J^w Y$ holds iff $\texttt{equal(}J\texttt{,}w\texttt{,O)} \in Z$, while $Y \diamond_J^w X$ does not hold for some $\diamond_J^w \in M$ iff $\texttt{worse(}J\texttt{)} \in Z$. We thus consider the possible criteria for every $\diamond_J^w \in M$:

1. If $\diamond = \leq$, we have that atoms $\texttt{count(S,}w\texttt{,Q,0...I)}$ are derived via the rules in Line 7, 31–32, and 34–37 of $\texttt{metaO.lp}$, where $\texttt{Q}$ is the maximum index such that facts of the form $\texttt{minimize(}J\texttt{,S)}$ and $\texttt{wlist(S,Q,}\ell'\texttt{,W)}$ belong to F. Then, we have that $\texttt{I}$ is the cardinality of the multiset of occurrences of $\ell = w@J$ in $\#\texttt{minimize}[\ell_1 = w_1@J_1, \dots, \ell_k = w_k@J_k]$ such that $X \models \ell$. Starting from $\texttt{I}$ in Line 39, $\texttt{cdown(S,}w\texttt{,-1,0)}$ is derived via the rules in Line 40–42 iff the cardinality of the multiset of occurrences of $\ell = w@J$ in $\#\texttt{minimize}[\ell_1 = w_1@J_1, \dots, \ell_k = w_k@J_k]$ such that $Y \models \ell$ is at least $\texttt{I}$, and $\texttt{cdown(S,}w\texttt{,-1,-1)}$ is derived via the same rules iff the cardinality of this multiset is greater than $\texttt{I}$. That is, $X \leq_J^w Y$ holds iff $\texttt{equal(}J\texttt{,}w\texttt{,card)}$ is derived via the rule in Line 44, and $\texttt{worse(}J\texttt{)}$ is derived via the rule in Line 46 if $Y \not\leq_J^w X$.
2. If $\diamond = \subseteq$, an atom $\texttt{ndiff(}\ell'_i\texttt{)}$ such that $\ell_i = w@J$ for some $1 \leq i \leq k$ is derived via the rules in Line 7 and 50–57 of $\texttt{metaO.lp}$ iff $X \not\models \ell_i$ or $Y \models \ell_i$. Thus, $\texttt{equal(}J\texttt{,}w\texttt{,incl)}$ is in turn derived via the rule in Line 59 iff $X \subseteq_J^w Y$ holds. Moreover, $\texttt{worse(}J\texttt{)}$ is derived via the rules in Line 61–62 and 63–64 if there is some $\ell_i = w@J$ for $1 \leq i \leq k$ such that $Y \models \ell_i$ and $X \not\models \ell_i$, i.e., if $Y \not\subseteq_J^w X$.
3. If $\diamond = \preceq$, an atom $\texttt{cando(}\ell'_i\texttt{)}$, $\texttt{nocan(}\ell'_i\texttt{)}$, $\texttt{condo(}\ell'_i\texttt{)}$, or $\texttt{nocon(}\ell'_i\texttt{)}$ such that $\ell_i = w@J$ for some $1 \leq i \leq k$ is derived via the rules in Line 7 and 68–92 of $\texttt{metaO.lp}$ iff $X \models \ell_i$ and $Y \not\models \ell_i$, $X \not\models \ell_i$ or $Y \models \ell_i$, $X \not\models \ell_i$ and $Y \models \ell_i$, or, respectively, $X \models \ell_i$ or $Y \not\models \ell_i$. Via the rule in Line 94–95, we further derive $\texttt{cando(S,}w\texttt{,}\ell'\texttt{)}$ iff $\texttt{minimize(}J\texttt{,S)}$, $\texttt{wlist(S,\_,}\ell'\texttt{,}w\texttt{)}$, and $\texttt{wlist(S,\_,}\ell'_1\texttt{,}w\texttt{)}$ belong to F such that $\ell \preceq \ell_1$, $X \models \ell$, $Y \not\models \ell$, $X \not\models \ell_1$, and $Y \models \ell_1$. Then, $X \preceq_J^w Y$ holds if there is no literal ℓ_i such that $\ell_i = w@J$ for some $1 \leq i \leq k$, $\ell_i \preceq \ell$, $\ell \not\preceq \ell_i$, $Y \models \ell_i$, and $X \not\models \ell_i$, i.e., if $\texttt{nocon(}\ell'_i\texttt{)}$

holds for every ℓ'_i such that `wlist(S,_,`ℓ'_i`,`w`)` and `prefer(`ℓ'_i`,`ℓ'`)`, but not `prefer(`ℓ'`,`ℓ'_i`)`, belong to F. Hence, `equal(`J`,`w`,pref)` is derived via the rule in Line 103–104 iff $X \preceq^w_J Y$ holds. Conversely, a literal ℓ_i such that $\ell_i = w@J$ for some $1 \leq i \leq k$ provides no indication for $Y \preceq^w_J X$ if $Y \not\models \ell_i$ or $X \models \ell_i$, if there is no $\ell = w@J$ in $\#\mathtt{minimize}[\ell_1 = w_1@J_1, \ldots, \ell_k = w_k@J_k]$ such that $\ell_i \preceq \ell$, $X \models \ell$, and $Y \not\models \ell$, or if $\ell \preceq \ell_i$, $\ell_i \not\preceq \ell$, $X \models \ell$, and $Y \not\models \ell$ for some $\ell = w@J$ in $\#\mathtt{minimize}[\ell_1 = w_1@J_1, \ldots, \ell_k = w_k@J_k]$. Some of these cases applies iff `nocon(S,`w`,`ℓ'_i`)` is derived via the rules in Line 97–101, so that `worse(`J`)` is in turn derived via the rule in Line 106 if $Y \not\preceq^w_J X$.

As we have above investigated all rules that can possibly derive `worse(`J`)`, i.e., the rules in Line 46, 61–64, and 106 of `metaO.lp`, and since some of them applies iff $\diamond^w_J \in M$ such that $Y \diamond^w_J X$ does not hold, we now conclude that `worse(`J`)` $\in Z$ iff $Y \diamond^w_J X$ does not hold for some $\diamond^w_J \in M$. In addition, the above cases yield that `equal(`J`,`w`,O)` $\in Z$ for `O` $=$ `card/incl/pref` iff $X \diamond^w_J Y$ holds for $\diamond = \leq/\subseteq/\preceq$ such that $\diamond^w_J \in M$. In view of the rules in Line 13–23, we further conclude that `bot` $\in Z$ iff there are no priority level J and weight w such that $Y \diamond^{w'}_{J'} X$ holds for all $\diamond^{w'}_{J'} \in M$ where $J' \geq J$ and $X \diamond^w_J Y$ does not hold for $\diamond^w_J \in M$. That is, `bot` $\in Z$ iff X is an answer set of Π that is not dominated by the answer set Y of Π wrt M. However, for any answer set Z of Π_{meta}, saturation in Line 61–62 of `metaD.lp` admits `bot` $\in Z$ only if `bot` is derived wrt every answer set Y of Π, i.e., if there is no answer set Y of Π that dominates X wrt M. Hence, the integrity constraint in Line 27 of `metaO.lp` selects exactly the answer sets Z of $\Pi_{\text{meta}} \setminus \{$`:- not bot.`$\}$ such that $X = \{a \mid$ `hold(atom(`a`))` $\in Z\}$ is an undominated and thus optimal answer set of Π wrt M. Finally, any such answer set Z of Π_{meta} is unique since all atoms in Z that are not of the form `hold(atom(`a`))` follow deterministically once the interpretation of these atoms is fixed. In particular, `bot` and `true(atom(`a`))` as well as `fail(atom(`a`))` for all $a \in \mathit{atom}(\Pi)$ hold in view of the integrity constraint in Line 27 of `metaO.lp` along with saturation in Line 61–62 of `metaD.lp`. $\square$

Regarding the computational complexity of tasks that can be addressed using our meta-programming approach to optimization, we first note that deciding whether there is an optimal answer set is in *NP*, as the existence of some answer set (decidable by means of `meta.lp` only) is sufficient for concluding that there is also an optimal one. However, the inherent complexity becomes more sensible if we consider the question of whether some atom a belongs to an optimal answer set. To decide it, one can augment the reified input program (but not the input program itself), `meta.lp`, `metaD.lp`, and `metaO.lp` with the integrity constraint `:- not hold(atom(`a`))`. Then, several complex optimization criteria at a single priority level `1` lead to completeness for Σ^P_2, the second level of the polynomial time hierarchy, thus showing that disjunctive ASP is appropriate to implement them. To see this, note that deciding whether an atom a belongs to some answer set of a positive disjunctive logic program is Σ^P_2-complete (Eiter and Gottlob 1995). When disjunctions $a_1 \vee \cdots \vee a_k$ in the heads of rules are rewritten to $1\ \#\mathtt{sum}[a_1 = 1, \ldots, a_k = 1]$, the question of whether an atom a belongs to an answer set of the original program can be addressed by reifying the rewritten program, adding the integrity constraint `:- not hold(atom(`a`))`, and applying `meta.lp`, `metaD.lp`, and `metaO.lp` wrt several optimization criteria. For one, we can include a $\#\mathtt{minimize}$ statement over all atoms of the input program, each associated with a different weight, to exploit the Pareto improvement implemented in `metaO.lp` for refuting a candidate answer set including a if it does not correspond to a minimal model, i.e., an answer set of the original program. Alternatively, we can include a `#minimize` statement over all atoms of the input program, each having the weight `1`, and augment the meta-program with the fact `optimize(1,1,incl)`. We could also use a `#minimize` statement over all atoms a_i of the input program along with their negation, each having the weight `1`, and add the facts `optimize(1,1,pref)` as well as `prefer(neg(atom(`a_i`)),pos(atom(`a_i`)))`. In view of these reductions, we conclude that Pareto efficiency, inclusion, and literal preferences independently capture computational tasks located at the second level of the polynomial time hierarchy, and our meta-programs allow for addressing them via an extended program along with facts (and possibly also integrity constraints) steering optimization relative to its reification.

5 Applications: A Case Study

While the approach of Eiter and Polleres (2006) consists of combining two separate logic programs, one for "guessing" and a second one for "checking," into a disjunctive program addressing both tasks, our meta-programming technique applies to a single (reified) input program along with complex optimization criteria. In fact, we provide a generic implementation of such criteria on top of extended programs encoding solution spaces. Hence, our meta-programming technique allows for a convenient representation of reasoning tasks in which testing the optimality of solutions to an underlying problem in *NP* lifts the complexity to Σ_2^P-hardness. Respective formalisms include ordinary, parallel, as well as prioritized circumscription (McCarthy 1980; Lifschitz 1985), minimal consistency-based diagnosis (Reiter 1987), and preferred extensions of argumentation frameworks (Dung 1995). Similarly, Pareto efficiency is an important optimality condition in decision making (Chevaleyre et al. 2007) and system design (Gries 2004). In the following, we illustrate the application of our approach on the example of an existing real-world application: repair wrt large gene-regulatory networks (Gebser et al. 2010).

Listing 7 shows a simplified version of the repair encoding given in (Gebser et al. 2010). It applies to a regulatory network, a directed graph with (partially) labeled edges, represented by facts of the predicates `vertex/1`, `edge/2`, and `obs_elabel/3`, where a label `S` is `1` (activation) or `-1` (inhibition). In addition, the data of experiments labeled `P` are provided by facts of the predicates `exp/1`, `inp/2` denoting input vertices (subject to perturbations), and `obs_vlabel/3`, where a label `S` is again `1` (increase) or `-1` (decrease). The regulatory network is consistent with the experiment data if there are total labelings of edges and vertices (for each experiment `P`) such that the label of every non-input vertex `V` is explained by the influence of some of its regulators `U`, where the influence is the product `S*T` of the edge label `S` and the label `T` of `U` (in experiment `P`). In the practice of systems biology, regulatory networks and experiment data are often collected from heterogeneous sources, and likewise incomplete or noisy. In view of this, it is very likely that a regulatory network and experiment data are mutually inconsistent, which makes it highly non-trivial to draw biologically meaningful conclusions in an automated way. To address this shortage, several repair operations were devised in (Gebser et al. 2010), which can be enabled via facts of the form `repair(K,J,W)`, where `K` indicates a certain kind of admissible

```
1  % auxiliary concepts

3  sign(-1;1).
4  complement(S,-S) :- sign(S).

6  % construct candidate repair

8  pos(aedge(U,V),  J,W) :- repair(aedge,J,W), vertex(U;V), U != V.
9  pos(eflip(U,V,S),J,W) :- repair(eflip,J,W), obs_elabel(U,V,S).
10 pos(ivert(V),    J,W) :- repair(ivert,J,W), vertex(V).
11 pos(pvert(P,V),  J,W) :- repair(pvert,J,W), exp(P), vertex(V).
12 pos(vflip(P,V,S),J,W) :- repair(vflip,J,W), obs_vlabel(P,V,S).

14 { apply(R) } :- pos(R,_,_).

16 % construct consistent total labelings

18 elabel(U,V,S) :- not apply(eflip(U,V,S)), obs_elabel(U,V,S).
19 elabel(U,V,T) :-     apply(eflip(U,V,S)),                   complement(S,T).
20 elabel(U,V,S) :-     apply(aedge(U,V)),   not elabel(U,V,T), complement(S,T).
21 elabel(U,V,S) :-     edge(U,V),           not elabel(U,V,T), complement(S,T).

23 vlabel(P,V,S) :- not apply(vflip(P,V,S)), obs_vlabel(P,V,S).
24 vlabel(P,V,T) :-     apply(vflip(P,V,S)), complement(S,T).
25 vlabel(P,V,S) :- not vlabel(P,V,T),       complement(S,T),   exp(P), vertex(V).

27 inf(P,V,S*T)  :- elabel(U,V,S), vlabel(P,U,T), not inp(P,V).

29 :- vlabel(P,V,S), not inf(P,V,S), not inp(P,V), not apply(ivert(V);pvert(P,V)).

31 % optimize repair

33 #minimize [ apply(R) = W @ J : pos(R,J,W) ].
```

Listing 7. Encoding of repair wrt regulatory networks and experiment data (`repair.lp`).

repair operations, `J` a priority level, and `W` a weight. The repair operations `R` to apply are selected via the rule in Line 14 of Listing 7, and their effects are propagated via the rules in Line 18–29, thus obtaining total edge and vertex labelings witnessing the reestablishment of consistency. Given that applications of repair operations modify a regulatory network or experiment data (depending on the kind of operations), we are interested in applying few operations only, which is expressed by the `#minimize` statement in Line 33.

A reasonable repair configuration could consist of facts of the following form:

`repair(ivert,`$\mathtt{J}_1$`,`$\mathtt{W}_1$`).` admitting to turn vertices into inputs in *all* experiments.
`repair(eflip,`$\mathtt{J}_2$`,`$\mathtt{W}_2$`).` admitting network modifications by flipping edge labels.
`repair(pvert,`$\mathtt{J}_3$`,`$\mathtt{W}_3$`).` admitting to turn vertices into inputs in specific experiments.
`repair(vflip,`$\mathtt{J}_4$`,`$\mathtt{W}_4$`).` admitting data modifications by flipping vertex labels.

While the kinds of repair referred to by `ivert` and `eflip` operate primarily on a network (in view of incompleteness or incorrectness), the ones denoted by `pvert` and `vflip` mainly address the data (which can be noisy). If we penalize all repair operations uniformly via $\mathtt{J} = \mathtt{J}_1 = \mathtt{J}_2 = \mathtt{J}_3 = \mathtt{J}_4$ and $\mathtt{W} = \mathtt{W}_1 = \mathtt{W}_2 = \mathtt{W}_3 = \mathtt{W}_4$, the instantiation of the `#minimize` statement in Line 33 represents ordinary cardinality-based optimization, assembled in solvers like *clasp* and *smodels*. However, by adding `optimize(J,W,incl)` as a fact, we can easily switch to inclusion-based minimization and use a disjunctive solver like *claspD* to solve the more complex problem. While our meta-programs enable such a shift of optimization criteria by means of adding just one fact, a direct disjunctive encoding of inclusion-based minimization has been provided in (Gebser et al. 2010); note that the latter is by far more involved than the basic repair encoding in Listing 7. Furthermore, our meta-programming approach allows us to distinguish between different kinds of repair operations (without prioritizing them) and optimize wrt Pareto efficiency. To accomplish this, one only needs to pick unequal values for $\mathtt{W}_1, \ldots, \mathtt{W}_4$, where cardinality-based minimization wrt each $\mathtt{W}_i$ can selectively be replaced by inclusion via providing a fact `optimize(J,`$\mathtt{W}_i$`,incl)`. Finally, we can choose to rank kinds of repair operations by providing different priority levels $\mathtt{J}_1, \ldots, \mathtt{J}_4$. For instance, network repairs may be considered as "more drastic" operations than data repairs, which can be expressed by assigning levels such that $\mathtt{J}_1 = \mathtt{J}_2 > \mathtt{J}_3 = \mathtt{J}_4$. In fact, the kinds of admissible repair operations as well as appropriate penalties for applying them depend on the biological application and experience. In this respect, the flexibility gained due to meta-programming allows for deploying and comparing different optimization criteria, e.g., regarding the accuracy of resulting predictions (cf. (Gebser et al. 2010)).

For giving an account of the practical capabilities of our meta-programming approach, we empirically compared it to the direct encoding of inclusion-based minimization in (Gebser et al. 2010). To this end, we ran *gringo* version 3.0.3 and *claspD* version 1.1 on 100 instances (random samples from real data of two biological experiments, as also used in (Gebser et al. 2010)) wrt three kinds of admissible repair operations, resulting in 300 runs each with our meta-programs and with the direct encoding. All runs have been performed sequentially on a machine equipped with Intel Xeon E5520 processors and 48 GB main memory under Linux, imposing a time limit of 4000 sec per run. (In view of moderate memory consumption, a space limit was not needed.) To our own surprise, more runs were completed in time with the meta-programs than with the direct encoding: 219 versus 150.[4] The disadvantages of the direct encoding show that further gearing would be required to improve solving efficiency, which adds to the difficulty of furnishing a functional saturation-based encoding. In view of this, we conclude that our meta-programming approach to complex optimization is an eligible and viable alternative. However, enhancements of disjunctive ASP solvers boosting its performance would still be desirable.

6 Discussion

The major contribution of this work is the provision of easy modeling capacities for expressing complex preferences involving a higher computational complexity. We accomplish this by developing advanced meta-modeling techniques that allow us to reinterpret `#minimize` statements by means of ASP. This methodology offers a generic saturation-based implementation technique for testing various optimality conditions wrt answer sets of extended logic programs, as used with *lparse* and *gringo*. At (metasp), we provide enhanced versions of our meta-encodings, supporting additional features of *gringo*'s input language (Gebser et al.) as well as some basic optimizations with regard to grounding performance. These encodings provide a ready-to-use platform for implementing applications involving complex optimization criteria in ASP.

Our integral approach to modeling complex optimization criteria in ASP brings about a

[4] All instances and detailed results are available at (metasp).

number of individual contributions. To begin with, we introduce the reification capacities of our grounder *gringo* along with the associated meta-encoding, paving the way to the immediate availability of meta-modeling techniques. In fact, the full version of the basic meta-encoding in Listing 1 covers the complete language of *gringo*, including disjunctions and diverse aggregates. Moreover, our meta-modeling techniques provide a general account of saturation and, thus, abolish its compulsory replication for expressing complex preferences. Of particular interest is the stability property of answer sets serving as implicit counterexamples. Unlike the approach of Eiter and Polleres (2006), our encoding avoids "guessing" level mappings. Also, our target language involves choice rules and `#sum` constraints (Simons et al. 2002), and we are unaware of any pre-existing meta-encoding of corresponding answer sets, neither as candidates nor as counterexamples. Likewise, related meta-programming approaches for generating consequences of logic programs (Faber and Woltran 2009) or explanations wrt debugging queries (Oetsch et al. 2010) do not consider such aggregates (but disjunctions in object programs).

We exploit the two-dimensionality of `#minimize` statements by using levels and weights for combining a lexicographic ranking with Pareto efficiency. At each level, one may choose whether groups of literals sharing the same weight are compared wrt cardinality or inclusion. Also, summation can be modeled by exploiting the multiset property of `#minimize` statements. This is extended by the framework of (Sakama and Inoue 2000), relying on a preference relation among literals (given in addition to `#minimize` statements). In fact, the approach of Section 4 allows for capturing the special cases of parallel and prioritized circumscription, investigated by Janhunen and Oikarinen in (2004; 2008). An interesting future extension is the encoding of optimality conditions for logic programs with ordered disjunction (Brewka et al. 2004), whose custom-made implementation in the prototype *psmodels* interleaves two *smodels* oracles for accomplishing a generate-and-test approach similar to the idea of our meta-programs. Ultimately, our approach could serve as an implementation platform for answer set optimization (Brewka et al. 2003) and the preference description language proposed in (Brewka 2004). Last but not least, our meta-programs furnish a rich and readily available source of Σ_2^P-hard challenge problems, fostering the development of ASP solvers for disjunctive logic programming.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2. We are grateful to Tomi Janhunen, Ilkka Niemelä, and the referees for their helpful comments.

References

BARAL, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

BREWKA, G. 2004. Answer sets: From constraint programming towards qualitative optimization. In *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, V. Lifschitz and I. Niemelä, Eds. Lecture Notes in Artificial Intelligence, vol. 2923. Springer-Verlag, 34–46.

BREWKA, G., NIEMELÄ, I., AND SYRJÄNEN, T. 2004. Logic programs with ordered disjunction. *Computational Intelligence 20,* 2, 335–357.

BREWKA, G., NIEMELÄ, I., AND TRUSZCZYNSKI, M. 2003. Answer set optimization. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann Publishers, 867–872.

CHEVALEYRE, Y., ENDRISS, U., LANG, J., AND MAUDET, N. 2007. A short introduction to computational social choice. In *Proceedings of the Thirty-third Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, J. van Leeuwen, G. Italiano, W. van der Hoek, C. Meinel, H. Sack, and F. Plasil, Eds. Lecture Notes in Computer Science, vol. 4362. Springer-Verlag, 51–69.

DELGRANDE, J., SCHAUB, T., AND TOMPITS, H. 2003. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming 3,* 2, 129–187.

DRESCHER, C., GEBSER, M., GROTE, T., KAUFMANN, B., KÖNIG, A., OSTROWSKI, M., AND SCHAUB, T. 2008. Conflict-driven disjunctive answer set solving. In *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, G. Brewka and J. Lang, Eds. AAAI Press, 422–432.

DUNG, P. 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence 77,* 2, 321–357.

EITER, T., FABER, W., LEONE, N., AND PFEIFER, G. 2003. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory and Practice of Logic Programming 3,* 4-5, 463–498.

EITER, T. AND GOTTLOB, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence 15,* 3-4, 289–323.

EITER, T. AND POLLERES, A. 2006. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming 6,* 1-2, 23–60.

FABER, W. AND WOLTRAN, S. 2009. Manifold answer-set programs for meta-reasoning. In *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Artificial Intelligence, vol. 5753. Springer-Verlag, 115–128.

FAGES, F. 1994. Consistency of Clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science 1*, 51–60.

FERRARIS, P. 2005. Answer sets for propositional theories. In *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Artificial Intelligence, vol. 3662. Springer-Verlag, 119–131.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. Freeman and Co.

GEBSER, M., GUZIOLOWSKI, C., IVANCHEV, M., SCHAUB, T., SIEGEL, A., THIELE, S., AND VEBER, P. 2010. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In *Proceedings of the Twelfth International Conference on Principles* *of Knowledge Representation and Reasoning (KR'10)*, F. Lin and U. Sattler, Eds. AAAI Press, 497–507.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.
Available at `http://potassco.sourceforge.net`.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI Press/The MIT Press, 386–392.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

GRIES, M. 2004. Methods for evaluating and covering the design space during early design development. *Integration 38,* 2, 131–183.

JANHUNEN, T. AND OIKARINEN, E. 2004. Capturing parallel circumscription with disjunctive logic programs. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA'04)*, J. Alferes and J. Leite, Eds. Lecture Notes in Computer Science, vol. 3229. Springer-Verlag, 134–146.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

LIFSCHITZ, V. 1985. Computing circumscription. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI'85)*, A. Joshi, Ed. Morgan Kaufmann Publishers, 121–127.

LIU, G. AND YOU, J. 2010. Level mapping induced loop formulas for weight constraint and aggregate logic programs. *Fundamenta Informaticae 101,* 3, 237–255.

LLOYD, J. 1987. *Foundations of Logic Programming*, 2nd ed. Symbolic Computation. Springer-Verlag.

MCCARTHY, J. 1980. Circumscription — a form of nonmonotonic reasoning. *Artificial Intelligence 13,* 1-2, 27–39.

METASP. `http://www.cs.uni-potsdam.de/wv/metasp`.

OETSCH, J., PÜHRER, J., AND TOMPITS, H. 2010. Catching the ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming. Twenty-sixth International Conference on Logic Programming (ICLP'10) Special Issue 10,* 4-6, 513–529.

OIKARINEN, E. AND JANHUNEN, T. 2008. Implementing prioritized circumscription by computing disjunctive stable models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA'08)*, D. Dochev, M. Pistore, and P. Traverso, Eds. Lecture Notes in Artificial Intelligence, vol. 5253. Springer-Verlag, 167–180.

REITER, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence 32,* 1, 57–96.

SAKAMA, C. AND INOUE, K. 2000. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence 123,* 1-2, 185–222.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

SYRJÄNEN, T. Lparse 1.0 user's manual.
Available at `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.