Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

asprin: Customizing Answer Set Preferences without a Headache

Gerhard Brewka
Universität Leipzig
Leipzig, Germany
brewka@informatik.uni-leipzig.de

James Delgrande
Simon Fraser University
Burnaby, B.C., Canada
jim@cs.sfu.ca

Javier Romero **Torsten Schaub***
Universität Potsdam
Potsdam, Germany
{javier,torsten}@cs.uni-potsdam.de

Abstract

In this paper we describe *asprin*[1], a general, flexible, and extensible framework for handling preferences among the stable models of a logic program. We show how complex preference relations can be specified through user-defined preference types and their arguments. We describe how preference specifications are handled internally by so-called preference programs, which are used for dominance testing. We also give algorithms for computing one, or all, optimal stable models of a logic program. Notably, our algorithms depend on the complexity of the dominance tests and make use of multi-shot answer set solving technology.

Introduction

Preferences are pervasive. The computation of preferred, or optimal, solutions is required in many real-world applications. Often this involves combining various qualitative and quantitative preferences. Although preferences have been widely studied in Answer Set Programming (ASP; (Baral 2003)) in various contexts (cf. (Delgrande et al. 2004)), current ASP systems are limited to optimization statements over sum or count aggregates, as specified either by $\#minimize$ statements or weak constraints. As well, these systems generally handle only a fixed, predefined, set of preferences.

We address these limitations in the system *asprin*, a general, flexible, and extensible framework for implementing preferences among the stable models of a logic program. Our framework is *general* and captures the major existing approaches to preference. Also, it is *extensible* and allows for an easy implementation of new or extended approaches to preference handling. Our framework builds on recent control capabilities for multi-shot ASP solving (providing successive yet operational grounding and solving of changing logic programs). This technology allows us to direct the search for specific preferred solutions without modifying the ASP solver. As well, it significantly reduces redundancies found in an iterated setting. Finally, this technology paves the way for the high customizability of our framework by offering an implementation of preferences via ordinary ASP encodings.

From an abstract point of view, the aim is to determine the *preferred* stable models of a given logic program and preference relation. That is, we determine a strict partial order $\succ$ on the stable models of a logic program P, representing a *preference relation*, such that $X \succ Y$ means that X is preferred to Y. Then, a stable model X of P is $\succ$*-preferred* if there is no other stable model Y such that $Y \succ X$. Restricting preferences to (strict) partial orders has the advantage that a satisfiable program has preferred stable models.[2]

The next section gives an overview of the *asprin* system, explaining its general functionality. We then describe in more detail how preferences are represented and handled in *asprin* via repeated calls to so-called preference programs. The following section describes some of the underlying algorithms. We then report results of evaluations we have performed. We conclude with a discussion of related work.

We assume some familiarity with ASP. For a comprehensive introduction, see (Gebser et al. 2012), from which we also take our notation. Our ASP encodings rely upon the new ASP language standard ASPCore2.0 (Calimeri et al. 2012). Other notation is introduced when first used.

Overview of *asprin*

In a nutshell, *asprin* allows a user to declare and evaluate preference relations among the stable models of an ASP program. Preferences are declared *within* a program by so-called *preference statements*. A preference statement is composed of an identifier, a type, and an argument set. The identifier names the preference relation, whereas the type and argument define the relation. Here is an example:

$$\#preference(costs, less(weight))\{40\colon sauna, 70\colon dive\} \quad (1)$$

This statement declares a preference relation named $costs$ with type $less(weight)$ and argument set $\{40 : sauna, 70 : dive\}$, and which minimizes the sum of weights of the atoms mentioned in the argument. Hence, stable models with neither $sauna$ nor $dive$ are preferred over those with only $sauna$. Stable models with only $dive$ are still less preferred, while those with both $sauna$ and $dive$ are least preferred.

*Affiliated with Inria Rennes, France, and SFU, Canada.

[1] *asprin* stands for "*ASP* for *pr*eference handl*in*g".

[2] *asprin* can also reason with prioritized classical models: programs having these models as answer sets are easily designed.

While the arguments of preference types are sets, we see later on that the elements contained in these sets—which we call *preference elements*—can be more complex than in the example. In the most general case we even admit conditional elements, which are used to capture conditional preferences. Moreover, preference types may refer to other preferences in their arguments, for example:

$$\#preference(all, pareto) \\ \{name(costs), name(fun), name(temps)\} \quad (2)$$

This defines a preference relation all which is the Pareto ordering of 3 preference relations $costs$, fun and $temps$. That is, stable model S_1 is strictly preferred to S_2 iff it is at least as good as S_2 with respect to the 3 argument orderings, and strictly better with respect to at least one of them.

Since preference statements may be needed for the specification of other relations, the user must explicitly specify which preference relation is to be used for optimization. This is done via an $\#optimize$ statement with the name of the respective preference statement as argument.

Once the preference and optimize statements are given, the computation of preferred stable models is done via a *preference program*. Such programs, which need to be defined for each preference type, take as input a stable model S_1 of the original program P and produce a stable model of P *strictly better* than S_1, according to the optimize statement, if such a stable model exists. An optimal stable model is computed stepwise, by repeated calls to the ASP solver: first, an arbitrary stable model of P is generated; then this stable model is "fed" to the preference program to produce a better one, etc. Once the preference program becomes unsatisfiable, the last stable model obtained is an optimal one.

We also provide an *asprin library* which contains a number of predefined, common, preference types along with the necessary preference programs. Users happy with what is available in the library can thus use the available types without having to bother with preference programs at all. However, if the predefined preference types are insufficient, users may define their own relations (and so become preference engineers). In this case, they also have to provide the preference programs *asprin* needs to cope with the new preference relations. How this works is explained in the following.

Expressing Preferences

We provide a generic preference language for expressing a wide range of preference relations. To keep our framework open for extensions, we do not fix a set of predefined preferences. Rather we give examples of how well-known preferences can be expressed and implemented. Many of these are included in *asprin*'s preference library, which provides basic building blocks for defining new preferences.

Syntax Our language is made up of the following elements: *weighted formulas*, *preference elements*, *preference statements*, and *preference specifications*.

Let $\mathcal{A}$ be a fixed alphabet. A *weighted (propositional) formula* is of the form[3] $w_1, \ldots, w_l : \phi$ where each w_i is a term and ϕ is a Boolean expression over $\mathcal{A}$ with logical connectives $\top$, $\neg$, $\wedge$, and $\vee$. We write ϕ whenever $l = 0$. For expressing composite preferences, we use a dedicated unary naming predicate $name$ that allows us to refer to auxiliary preferences. That is, a *naming atom*, $name(s)$, refers to relations associated with a preference statement s (see below).

A *preference element* is of the form[4]

$$\Phi_1 > \cdots > \Phi_m \parallel \phi \quad (3)$$

where ϕ is a non-weighted formula giving the context, and each Φ_r is a set of weighted formulas for $r = 1, \ldots, m$ and $m \geq 1$. Intuitively, r gives the rank of the respective set of weighted formulas. Preference elements provide a (possible) structure to a set of weighted formulas by giving a means of conditionalization and a symbolic way of defining pre-orders (in addition to using weights). For convenience, we may drop the surrounding braces of such sets and omit "$\parallel \phi$" if ϕ is tautological. Also, we drop "$>$" if $m = 1$. Hence $\{a, \neg b\} > c$ stands for $\{a, \neg b\} > \{c\} \parallel \top$. Similarly, $40 : sauna$ in (1) stands for $\{40 : sauna\} \parallel \top$.

A *preference statement* p is of the form

$$\#preference(s, t)\{e_1, \ldots, e_n\} \quad (4)$$

where s and t are ground terms giving the preference name and its type, respectively, and each e_j is a preference element. In what follows, we use $id(p)$ to denote p's identifier s and refer to its type by t_s. The preference type determines the set of admissible preference elements. For instance, $less(weight)$ in (1) is restricted to weighted literals. A preference type may or may not require naming atoms, depending on whether it is *composite* or *primitive*. For instance, $pareto$ in (2) is composite.

A set of preference statements is accompanied by a single *optimization directive*, $\#optimize(s)$, telling a solver to restrict its reasoning mode to the preference relation declared by s. The collection of preference statements in a program has to satisfy certain requirements to be useful. We say a set of preference statements S is

- *closed*, if $p \in S$ for some p with $id(p) = s$ whenever $name(s)$ occurs in S, and
- *acyclic*, if the dependency relation induced among preference statements in S by naming atoms is acyclic.

A *preference specification* is a set of preference statements S along with a single directive $\#optimize(s)$ such that S is acyclic and closed and, for some $p \in S$, $id(p) = s$. We call p the *primary* preference statement in S and refer to statements in $S \setminus \{p\}$ as *secondary*.

Semantics A preference statement like (4) declares a preference relation with associated preference type and preference elements. More formally, a preference type, t, is a function mapping a set of preference elements, E, to a (strict) preference relation $t(E)$ on stable models. The full generality of preference elements is not always needed. For this reason, we will specify for each preference type t its domain $dom(t)$. A preference statement $\#preference(s, t)E$ is said

[3] This syntax follows *aggregate elements* (Calimeri et al. 2012).

[4] Following (Bienvenu et al. 2010).

to be *admissible*, if $E \in dom(t)$. For simplicity, we also denote the relation $t(E)$ by $\succ_s$.

Here is an example of defining a preference type:

- Define $less(card)$ by: $dom(less(card)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$, where $\mathcal{P}(S)$ is the power set of S, and $(X, Y) \in less(card)(E)$ iff
$$|\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$$

Our approach centres on the implementation of the decision problem: $(X, Y) \in t(E)$? The resulting algorithmic framework requires that the induced preference relations be strict partial orders. We can still define strict preference relations in terms of non-strict or other auxiliary preference relations, as long as the preference relation subject to optimization is irreflexive and transitive.

Here is a further example from *asprin*'s library, going beyond existing preferences in ASP solvers.

- Define $subset$ by: $dom(subset) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$, and $(X, Y) \in subset(E)$ iff
$$\{\ell \in E \mid X \models \ell\} \subset \{\ell \in E \mid Y \models \ell\}$$

Composite preferences are formed by aggregation. The naming predicate $name$ is used to refer to auxiliary preferences. For a naming atom $name(s')$, we let $\succeq_{s'}$ be the non-strict preference relation associated with preference statement s'. Relations $\succ_{s'}$, $=_{s'}$, $\prec_{s'}$, $\preceq_{s'}$ are similarly defined.

For example, let N be the set of naming atoms. Then:

- Define $pareto$ by: $dom(pareto) = \mathcal{P}(\{n \mid n \in N\})$, and $(X, Y) \in pareto(E)$ iff
$$\bigwedge_{name(s)\in E}(X \succeq_s Y) \wedge \bigvee_{name(s)\in E}(X \succ_s Y)$$
- Define lex by: $dom(lex) = \mathcal{P}(\{w{:}n \mid n \in N, w \in \mathbb{N}\})$ and $(X, Y) \in lex(E)$ iff
$$\bigvee_{w:name(s)\in E}((X \succ_s Y) \wedge \bigwedge_{v:name(s')\in E, v<w}(X =_{s'} Y))$$

Other composite preference types are easily defined.

Here are examples with specific preference elements.

- $\#preference(1, less(card))\{a, \neg b, c\}$ declares $X \succ_1 Y$ as
$$|\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|$$
- $\#preference(2, more(weight))\{1{:}a, 2{:}\neg b, 3{:}c\}$ declares $X \succ_2 Y$ as
$$\sum_{(w:\ell)\in\{1:a,2:\neg b,3:c\}, X\models\ell} w > \sum_{(w:\ell)\in\{1:a,2:\neg b,3:c\}, Y\models\ell} w$$
- $\#preference(3, subset)\{a, \neg b, c\}$ declares $X \succ_3 Y$ as
$$\{\ell \in \{a, \neg b, c\} \mid X \models \ell\} \subset \{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}$$
- $\#preference(4, pareto)\{name(1), name(2), name(3)\}$ declares $X \succ_4 Y$ as $(X \succeq_1 Y) \wedge (X \succeq_2 Y) \wedge (X \succeq_3 Y) \wedge ((X \succ_1 Y) \vee (X \succ_2 Y) \vee (X \succ_3 Y))$
- $\#preference(5, lex)\{1 : name(1), 2 : name(2), 3 : name(3)\}$ declares $X \succ_5 Y$ as $(X \succ_1 Y) \vee ((X =_1 Y) \wedge (X \succ_2 Y)) \vee ((X =_1 Y) \wedge (X =_2 Y) \wedge (X \succ_3 Y))$

Handling Preferences

We now discuss how preferences in *asprin* are handled through *preference programs*.

Instance format Preference statements are represented internally as a collection of facts. A weighted formula of form $w_1, \ldots, w_l : \phi$ occurring in some set Φ_r of a preference element e_j in a preference statement s as in (4) is represented as a fact of form

$$\texttt{preference}(\texttt{s}, \texttt{j}, \texttt{r}, \texttt{for}(\texttt{t}_\phi), (\texttt{w}_1, \ldots, \texttt{w}_1)).$$

where each $\texttt{w}_\texttt{i}$ represents w_i for $i = 1, \ldots, l$ and $\texttt{t}_\phi$ is a term representing ϕ by using function symbols $\texttt{neg}/2$, $\texttt{and}/2$, and $\texttt{or}/2$. For simplicity, we use indexes, r and j, for identifying the respective structural components. For representing the condition of e_j, we set $\texttt{r}$ to $\texttt{0}$. A naming atom $name(s)$ is represented analogously, except that $\texttt{for}(\texttt{t}_\phi)$ is replaced by $\texttt{name}(\texttt{s})$.

We let $F_{s,j}$ denote the set of all facts obtained for all weighted formulas and naming atoms contained in preference element e_j belonging to preference statement s. With this, we define the translation of a preference statement $\#preference(s, t)\{e_1, \ldots, e_n\}$ as

$$F_s = \{\texttt{preference}(\texttt{s}, \texttt{t}_\texttt{s}).\} \cup \bigcup_{j=1,\ldots,n} F_{s,j}$$

For example, some of the previous preference statements are translated as follows.

- $\#preference(1, less(card))\{a, \neg b, c\})$ yields

```
preference(1,less(cardinality)).
preference(1,1,1,for(a),()).
preference(1,2,1,for(neg(b)),()).
preference(1,3,1,for(c),()).
```

- $\#preference(2, more(weight))\{1 : a, 2 : \neg b, 3 : c\})$ yields

```
preference(2,more(weight)).
preference(2,1,1,for(a),(1)).
preference(2,2,1,for(neg(b)),(2)).
preference(2,3,1,for(c),(3)).
```

- $\#preference(5, lex)\{1 : name(1), 2 : name(2), 3 : name(3)\}$ yields

```
preference(5,lexico).
preference(5,1,1,name(1),(1)).
preference(5,2,1,name(2),(2)).
preference(5,3,1,name(3),(3)).
```

Encoding We consider logic programs over a set $\mathcal{A}$ of atoms and refer to them as *base programs*. We assume that a preference program is formed over a set $\mathcal{E} \cup \mathcal{F}$ of atoms disjoint from $\mathcal{A}$. While $\mathcal{F}$ provides a fixed set of internal atoms (formed from dedicated predicate symbols, such as `preference`), $\mathcal{E}$ can be customized.

For deciding whether one stable model is preferred to another, we implement each preference type t by an ASP encoding E_t. E_t defines under which conditions a (new) stable model is strictly better than an old one, given preference type t and facts F_s. This is expressed using a unary predicate `better`. In addition, preference programs use auxiliary rules, C, describing problem-independent internal concepts like satisfaction of Boolean expressions. C also contains the integrity constraint:

```
:- not better(P), optimize(P).
```

This constraint ensures the preference program is unsatisfiable whenever no strictly better stable model can be found. To implement the obligatory optimization directive $\#optimize(s)$, we add '`optimize(s).`' to the facts F_s of the primary preference s.

The question to be addressed now is: How do we represent stable models in order to compare them? For this we rely on ASP's meta-interpretation capacities and reify atoms to constants by using unary predicates $holds$ and $holds'$. We define for $X \subseteq \mathcal{A}$ the following sets.

$$\begin{array}{lcl} H_X & = & \{holds(a) \mid a \in X\} \\ H'_X & = & \{holds'(a) \mid a \in X\} \\ R_X & = & \{holds(a) \leftarrow a \mid a \in X\} \\ R'_X & = & \{holds'(a) \leftarrow a \mid a \in X\} \\ G_X & = & \{\{h\} \leftarrow \mid h \in H_X\} \\ G'_X & = & \{\{h'\} \leftarrow \mid h' \in H'_X\} \end{array}$$

R_X provides the dynamic counterpart of H_X relative to an encompassing model X. Atoms formed by using predicates $holds$ and $holds'$ are internal and thus belong to $\mathcal{F}$.

The next results form the theoretical foundation of our approach.

Definition 1 *Let s be a preference statement declaring preference relation $\succ_s$ and let the programs E_{t_s}, F_s, and A be as described above. We call $E_{t_s} \cup F_s \cup C$ a* preference program *for s, if for all sets $X, Y \subseteq \mathcal{A}$, we have*

$$X \succ_s Y \text{ iff } E_{t_s} \cup F_s \cup C \cup H_X \cup H'_Y \text{ is satisfiable.}$$

Note that preference programs refer only to atom sets and are thus initially independent of any base programs. This changes once a program P together with reifying or generating rules, like R_X or G_X, is considered (instead of H_X).

For illustration, consider the implementation of a preference program for $\#preference(3, subset)\{a, \neg b, c\}$ that allows us to check whether $\{a, b\} \succ_3 \{a\}$ holds.

$E_{subset} =$
```
better(P) :- preference(P,subset),
 1 #sum{ 1,X : not holds(X), holds'(X),
             preference(P,_,_,for(X),_) },
 holds'(X) : preference(P,_,_,for(X),_),
             holds(X).
```

$F_3 =$
```
preference(3,subset).       optimize(3).
preference(3,1,1,for(a),()).
preference(3,2,1,for(neg(b)),()).
preference(3,3,1,for(c),()).
```

$C =$
```
:- not better(P), optimize(P).
holds(neg(A))  :- not holds(A),
        preference(_,_,_,for(neg(A)),_).
holds'(neg(A)) :- not holds'(A),
        preference(_,_,_,for(neg(A)),_).
```

$H_{\{a,b\}} =$
```
{ holds(a).  holds(b). }
```

$H'_{\{a\}} =$
```
{ holds'(a). }
```

Grounding and solving the above programs yields a stable model containing `better(3)` indicating that $\{a, b\} \succ_3 \{a\}$, or $\{a\} \subset \{a, \neg b\}$, holds.

Base and preference programs are formed over disjoint sets of atoms. Interactions among them are controlled by mapping atoms in $\mathcal{A}$ to $H_\mathcal{A}$ and $H'_\mathcal{A}$, respectively. The next proposition makes precise how preference programs capture the semantics of preference statements.

Proposition 1 *Let $E_{t_s} \cup F_s \cup C$ be a preference program for preference statement s.*

1. *If Z is a stable model of $E_{t_s} \cup F_s \cup C \cup G_\mathcal{A} \cup G'_\mathcal{A}$, then $\{a \mid holds(a) \in Z\} \succ_s \{a \mid holds'(a) \in Z\}$.*
2. *If $X \succ_s Y$, then there is a stable model Z of $E_{t_s} \cup F_s \cup C \cup G_\mathcal{A} \cup G'_\mathcal{A}$ such that*

$$X = \{a \mid holds(a) \in Z\} \text{ and } Y = \{a \mid holds'(a) \in Z\}.$$

The above implies that $\succ_s = \{(X, Y) \mid X, Y \subseteq \mathcal{A}, (E_{t_s} \cup F_s \cup C \cup H_X \cup H'_Y) \text{ is satisfiable}\}$.

Now, replacing $H_{\{a,b\}}$ and $H_{\{a\}}$ among the above programs by

$G_{\{a,b,c\}} =$
```
{ { holds(a); holds(b); holds(c) }. }
```

$G'_{\{a,b,c\}} =$
```
{ { holds'(a); holds'(b); holds'(c) }. }
```

yields 19 stable models capturing preference relation $\succ_3$.

Next, we show how preference programs can be used for deciding whether a stable model of a base program is preferred, and how a dominating model is obtained.

Proposition 2 *Let P be a program over $\mathcal{A}$ and let s be a preference statement.*

1. *If X is a stable model of P, then X is $\succ_s$-preferred iff $\left(P \cup E_{t_s} \cup F_s \cup C \cup R_\mathcal{A} \cup H'_X\right)$ is unsatisfiable.*
2. *If Y is a stable model of $\left(P \cup E_{t_s} \cup F_s \cup C \cup R_\mathcal{A} \cup H'_X\right)$ for some $X \subseteq \mathcal{A}$, then $Y \cap \mathcal{A}$ is a stable model of P such that $(Y \cap \mathcal{A}) \succ_s X$.*

We use $(P \cup E_{t_s} \cup F_s \cup C \cup R_\mathcal{A} \cup H'_X)$ for checking whether there is a model dominating X. Note how the usage of program $P \cup R_\mathcal{A}$ restricts candidates to stable models of P (unlike arbitrary subsets of $\mathcal{A}$ as in Proposition 1).

Now, adding P and replacing $H_{\{a,b\}}$ by $R_{\{a,b,c\}}$ among the above programs, where

$P =$
```
{ 2 { a; b; c } 2. }
```

$R_{\{a,b,c\}} =$
```
holds(a) :- a.
holds(b) :- b.
holds(c) :- c.
```

allows us to check whether a set X is dominated by some stable model of P. For instance, checking whether $\{a\}$ is dominated is done by keeping $H'_{\{a\}}$. This yields a stable model containing `holds(a)` and `holds(b)` and tells us that $\{a\}$ is dominated by $\{a, b\}$. Now, replacing $H'_{\{a\}}$ by $H'_{\{a,b\}}$ yields an unsatisfiable program, indicating that $\{a, b\}$ is a $\succ_3$-preferred stable model of P.

Both primitive and composite preference types are implemented by preference programs. For the latter, the definition of `better` depends on the respective definitions of the constituent preference types. Note that for some cases definitions of non-strict versions of the intended preference relations, respectively equality, need to be provided as well.

Thus, given a preference specification S, we let the preference program $E_{t_s} \cup F_s$ of the primary preference s in S also account for the specifications of all secondary preferences.

In the rest of this section we show how selected preference types are implemented in *asprin*.

- $less(card)$ is implemented by

```
better(P) :- preference(P,less(cardinality)),
 1 #sum{ -1,X: holds(X),  preference(P,_,_,for(X),_);
          1,X: holds'(X), preference(P,_,_,for(X),_) }.
```

- $more(weight)$ is implemented by

```
better(P) :- preference(P,more(weight)),
 1 #sum{  W,X: holds(X), preference(P,_,_,for(X),(W));
         -W,X: holds'(X),preference(P,_,_,for(X),(W))}.
```

The preference type *pareto* uses a non-strict relation $bettereq$; for $less(cardinality)$ and $more(weight)$ this is implemented by simply replacing the above lower bound `1` by `0`. Given this, *pareto* is implemented by

```
better(P) :- preference(P,pareto),
             better(P''), preference(P,_,_,name(P''),_),
             bettereq(P') : preference(P,_,_,name(P'),_).
```

In general the correctness of a preference program is the responsibility of the implementer, just as with regular ASP encodings. However, for *asprin*'s preference library, we can provide correctness results.

Computing Preferences

Our algorithms rely upon successive calls to a (multi-shot) ASP solver (for this we use *clingo* 4). For a (normal or disjunctive) program P, define

$$solve(P) = \begin{cases} X & \text{if } X \text{ is (some) stable model of } P \\ \bot & \text{if } P \text{ is unsatisfiable} \end{cases}$$

Computing one preferred model Given a program P and a preference statement s, Algorithm 1[5] computes a $\succ_s$-preferred stable model of P. We put no restrictions on the program P or on the preference program; both may even be disjunctive programs. Note that the non-dominance test for candidate models is implemented as prescribed by Proposition 1. This is done in Line 5 where we check whether there is a Y such that $Y \succ_s X$.[6]

Theorem 1 *Given a program P and preference statement s, Algorithm 1 computes a $\succ_s$-preferred stable model of P if P is satisfiable, and $\bot$ otherwise.*

Computing all preferred models While base programs remain unrestricted, we limit ourselves here to preferences for which we can decide whether $X \succ Y$ holds for sets X, Y in polynomial time. Given this, we assume without loss of generality that preference programs are stratified (Apt et al. 1987), since each problem decidable in polynomial time can be represented as a stratified logic program.

[5]This algorithm is inspired by ideas in (Brewka et al. 2003; Giunchiglia and Maratea 2012), see the discussion for details.

[6]Note that Algorithm 1 can easily be turned into an anytime algorithm returning the best answer set computed so far.

Algorithm 1: $solveOpt(P, s)$

Input : A program P over $\mathcal{A}$ and preference statement s.
Output : A $\succ_s$-preferred stable model of P, if P is satisfiable, and $\bot$ otherwise.

1 $Y \leftarrow solve(P)$
2 **if** $Y = \bot$ **then return** $\bot$
3
4 **repeat**
5 $\quad X \leftarrow Y$
6 $\quad Y \leftarrow solve(P \cup E_{t_s} \cup F_s \cup C \cup R_{\mathcal{A}} \cup H'_X) \cap \mathcal{A}$
7 **until** $Y = \bot$
8 **return** X

Given a program P and a preference statement s, Algorithm 2 computes all $\succ_s$-preferred stable models of P. The idea is to collect preferred models computed in analogy to Algorithm 1. To see this, observe that Lines 3-8 correspond to Lines 1-6 in Algorithm 1. That is, starting from an initial model Y in Line 3 a preferred model, X, is obtained after the repeat loop via successive non-dominance tests. Preferred models are accumulated in the indexed set $\mathcal{X}$.

Algorithm 2: $solveOptAll(P, s)$

Input : A program P over $\mathcal{A}$ and preference statement s.
Output : The set of $\succ_s$-preferred stable models of P.

1 $\mathcal{X} \leftarrow \emptyset$
2 **loop**
3 $\quad Y \leftarrow solve\big(P \cup \bigcup_{X_i \in \mathcal{X}} \big(N_{X_i} \cup (\overline{E_{t_s}} \cup F_s \cup C \cup H_{X_i})^i \cup R'^i_{\mathcal{A}}\big)\big) \cap \mathcal{A}$
4 $\quad$ **if** $Y = \bot$ **then return** $\mathcal{X}$
5
6 $\quad$ **repeat**
7 $\quad\quad X \leftarrow Y$
8 $\quad\quad Y \leftarrow solve\big(P \cup E_{t_s} \cup F_s \cup C \cup R_{\mathcal{A}} \cup H'_X\big) \cap \mathcal{A}$
9 $\quad$ **until** $Y = \bot$
10 $\quad \mathcal{X} \leftarrow \mathcal{X} \cup \{X_{|\mathcal{X}|+1}\}$

The most intricate part of Algorithm 2 is Line 3. The goal is to compute a stable model of P that is neither dominated by nor equal to any preferred model in $\mathcal{X}$. Line 3 checks whether there is a stable model Y of P such that $X_i \neq Y$ and $X_i \not\succ_s Y$ for all $i \in I$. (We already have $Y \not\succ_s X_i$ since each $X_i \in \mathcal{X}$ is $\succ_s$-preferred.) The condition $X_i \neq Y$ is guaranteed by the integrity constraint N_{X_i} of form $N_X = \{\leftarrow X \cup \{\sim a \mid a \in \mathcal{A} \setminus X\}\}$ for each $i \in I$. Although such solution recording is exponential in space, it is non-intrusive to the solver. For the condition $X_i \not\succ_s Y$, preference programs are not directly applicable since they result in an unsatisfiability problem according to Definition 1. Instead, we need to encode the condition as a satisfiability problem in order to obtain a stable model as a

starting point for the subsequent search. Due to our restriction to stratified preference programs, this is accomplished as follows: Given a program P, define $\overline{P}$ as the program

$$(P \setminus \{r \in P \mid head(r) = \emptyset\}) \cup \{u \leftarrow body(r) \mid r \in P, head(r) = \emptyset\} \cup \{\leftarrow \sim u\},$$

where u is a new atom. Now, if program P is stratified, P is satisfiable iff $\overline{P}$ is unsatisfiable. Moreover, let $E_{t_s} \cup F_s \cup C$ be a stratified preference program for preference statement s. Then, for all sets X, Y of atoms over $\mathcal{A}$, we have

$$X \not\succ_s Y \text{ iff } \overline{E_{t_s}} \cup F_s \cup C \cup H_X \cup H'_Y \text{ is satisfiable.}$$

Based on this, we can prove soundness and completeness.

Theorem 2 *Given a program P and a preference statement s, Algorithm 2 computes the set of all $\succ_s$-preferred stable models of P.*

In addition to the algorithms described above, *asprin* also provides algorithms for more complex preferences and for an axiomatic approach to preferences. Due to space restrictions we cannot go into detail here. In brief: Our algorithms for complex preferences (described elsewhere) build upon the *saturation* technique of (Eiter and Gottlob 1995) and the meta-interpretation-based approach in (Gebser et al. 2011). In the axiomatic approach, a program is extended so that the stable models of the extended program correspond to the preferred stable models of the original one. This extension is again formulated via saturation (cf. (Gebser et al. 2011)).

The *asprin* system

We implemented *asprin* by means of the ASP-Python integration of *clingo* 4.4 (Gebser et al. 2014); it is publicly available at (potassco). In fact, Algorithm 1 and 2 are implemented in Python and interact with a continuously running *clingo* object through `ground` and `solve` functions. The current *asprin* library includes `more` and `less` as regards `cardinality` and `weight`, respectively, `subset` and `superset`, `aso` (Brewka et al. 2003), partial orders of (Giunchiglia and Maratea 2012), `lexico`, `pareto`, `and` and `neg` (Son and Pontelli 2006).

Although *asprin*'s unique characteristics lie in its flexibility and generality, let us investigate how much this costs confronted with dedicated implementations, and also analyze the effect of preference composition. To this end, we have gathered 193 benchmark instances from eight different classes: 15-*Puzzle*, *Crossing*, and *Valves* stem from the ASP competitions[7] of 2009 and 2013; *Ricochet* Robots from (Gebser et al. 2013a), Circuit *Diagnosis* from (Siddiqi 2011) and adapted to ASP in (Gebser et al. 2013b), Metabolic Network *Expansion* from (Schaub and Thiele 2009), Transcription Network *Repair* from (Gebser et al. 2010), and *Timetabling* from (Banbara et al. 2013). All classes involve a single optimization statement; *Valves* and *Timetabling* deal with weight summation, all others with cardinality. We selected from each class (if possible) the 30 most runtime-consuming instances solvable in 300s by *clingo* 4.4.[8] The number of instances is given in parentheses in the first column of Table 1, followed by the average number of ground atoms comprised in the optimization statements. Each other entry (above the last line) provides the average runtime per class with the system indicated in the column header; the last line gives the overall average and timeouts. Each run was restricted to 900s CPU time (on Intel Xeon dual-core processors with 3.4 GHz and 4 GB RAM under Linux). Timeouts account for 900s; their respective number is given in parentheses.

First, let us consider mono-objective cardinality and weight optimization (c/w) to contrast *asprin*'s outer solver optimization with the inner solver one of *clingo* (using branch-and-bound in its default setting). Comparing the results given in the columns headed by *clingo* and '*asprin*$_{c/w}$', we observe that on average *clingo* 4.4 takes only one third of the time needed by *asprin* (based on the same *clingo* 4.4 configuration), which moreover times out in 11 cases. This difference is mainly caused by the respective performance on *Diagnosis*, *Expansion*, and *Timetabling* instances. Although all three have optimization statements involving large sets of atoms, this cannot be the major cause in view of the moderate difference observed on *Repair* instances. In fact, a closer look reveals that all three classes exhibit the longest convergence to the optimum. That is, both *asprin* as well as *clingo* enumerate a large number of candidate models before they find the optimal one. While *asprin* and *clingo* inspect on average 123.87 and 167.35 stable models, they probe 362.64, 291.86, 236 and 361.37, 301.23, 565.58, respectively, on the aforementioned classes. Although *asprin* converges globally even better than *clingo*, each unsuccessful model makes it ground further rules. As a result, on these classes *asprin* spends 20-50% of its runtime on intermediate grounding, while this is negligible on all other classes. This is obviously a bottleneck of our approach; however, it can be remedied by improving the convergence to the optimum. An effective way is to suppress memorized phase assignments (Pipatsrisawat and Darwiche 2007) among consecutive `solve` calls, as offered by *clingo*'s option `--forget`. The result can be seen in the column headed '*asprin*$_{c/w}$ `-f`'. This strategy results in an enumeration of only 74.98 models on average and leads to a corresponding improvement in runtime, although in some cases we get more timeouts.[9]

Consider now mono-objective subset optimization (s). In ASP, this is done via saturation-based encodings using disjunctive logic programs. The *metasp* system reflects this approach by compiling a normal logic program along with a subset-oriented optimization statement into such a disjunctive logic program, which can then be solved with *clingo* 4.4. The results in the last three columns of Table 1 show that *asprin* (using normal logic programs) outperforms *metasp*, and moreover that finding subset-minimal models is even easier than cardinality-minimal ones (ignoring weights).

Next, let us briefly summarize our experiments on the impact of preference composition. For this, we turned the above mono-objective problems into multi-objective

[7] Other competition classes were either too easy or too difficult.

[8] A cutoff at 900s brought only a handful of additional instances.

[9] We get a higher average of 358.90 models with *Timetabling*.

Benchmark \ System		*clingo*	*asprin*$_{c/w}$	*asprin*$_{c/w}$ `-f`	*metasp*	*asprin*$_s$	*asprin*$_s$ `-f`
Ricochet	(30) 20.00	104.74 (0)	174.26 (0)	113.45 (0)	811.32 (24)	175.71 (0)	109.91 (0)
Timetabling	(12) 23687.75	35.82 (0)	490.39 (5)	694.92 (8)	798.75 (10)	142.03 (0)	12.01 (0)
Puzzle	(7) 580.57	77.00 (0)	77.39 (0)	96.70 (0)	34.79 (0)	17.06 (0)	17.22 (0)
Crossing	(24) 211.92	48.43 (0)	105.64 (1)	67.50 (0)	62.33 (0)	0.50 (0)	0.46 (0)
Valves	(30) 56.63	52.53 (0)	72.85 (0)	78.11 (0)	900.00 (30)	45.01 (0)	39.31 (0)
Expansion	(30) 7501.87	91.53 (0)	373.56 (2)	241.05 (7)	900.00 (30)	292.57 (0)	21.12 (0)
Repair	(30) 6750.73	71.78 (0)	102.19 (0)	43.94 (0)	900.00 (30)	6.88 (0)	2.19 (0)
Diagnosis	(30) 1669.00	84.96 (0)	254.19 (3)	101.33 (0)	181.71 (6)	41.55 (0)	1.56 (0)
Average		70.85 (0)	206.31 (11)	179.63 (15)	573.61 (130)	90.16 (0)	25.47 (0)

Table 1: Comparing different *asprin* settings with *clingo* 4.4 and *metasp*

ones by splitting their optimization statements, which were then composed in different ways. First, we distribute the atoms in an optimization statement to obtain 16 statements of the same type. We then use 1,3,7,15 `pareto` (or `lexico`) statements to compose the 16 ones in a tree-like fashion. Note that all four representations are equivalent (and `lexico`- implies `pareto`-optimality). As regards `pareto`, we observe a similar performance on all four configurations. An exception is observed on *Expansion* and *Repair* where a single `pareto` led to much shorter convergence and hence better performance. Globally, the addition of `pareto` led to a worse performance than obtained in Table 1; and it gets even worse when using `lexico`, although as before all four representations show similar results. Second, we divided the atoms in an optimization statement into $1, 2, 4, 8, 16$ statements of the same type and connected them with a single `pareto` (or `lexico`) statement. This results in different preferences with a decreasing number of preferred models. This is also reflected by *asprin*'s performance that improves with an increasing number of optimization statements. A similar behavior with worse performance is observed with `lexico`.

Finally, we compared *asprin* with the system *satpref* (di Rosa and Giunchiglia 2013) computing optimal models of satisfiability problems with partially ordered qualitative preferences. We ran all the benchmarks of (di Rosa and Giunchiglia 2013) in the same machine and with the same time limit as before. For *asprin* we translated the satisfiability problems to ASP and used option `--forget`, and for *satpref* we used the approach *optsat-bf*.[10] On random benchmarks *asprin* is faster, it takes 59.54 seconds average time and 99 timeouts, while *satpref* needs 101.38 seconds and timeouts 182 times. On structured benchmarks the picture is different and *satpref* is faster: *asprin* spends 103.16 seconds on average and does 195 timeouts, while *satpref* average time goes down to 15.16 seconds and timeouts only 26 times.

Discussion

This paper introduces a general, flexible and extendable framework for preference handling in ASP. Our intention was not primarily to come up with new preference relations on stable models that have not been previously studied (although one can certainly introduce such new relations in *asprin*). Rather our goal was to provide ASP technology matching the substantial research on preference handling in ASP and beyond. Essentially, we want to put this research into practice. We believe that *asprin* may play a similar role for answer set optimization (Brewka et al. 2003) as the development of efficient ASP solvers had in boosting the basic answer set solving paradigm.

We expect two types of users: those who are happy using the preference relations in the *asprin* library, and those who want to exploit the extensibility of the system and define their own preference orderings. For the former, much of the technical capabilities of the system are not needed. In fact, they can use *asprin* as a preference handling system where all one needs to know are the available preference types and their arguments. For the latter type of users, let's call them *preference engineers*, the system provides all the additional functionality to define interesting new preference orderings.

There is a large body of work on preferences in logic programming. This literature is too large to be discussed here in detail; see (Delgrande et al. 2004) for a comprehensive (though somewhat dated) overview and (Costantini et al. 2010) for more recent work on preferences related to (quantitative) resources. A study of the computational complexity of the ASO approach with related algorithms can be found in (Zhu and Truszczyński 2013). Our algorithms are inspired by ideas in (Brewka et al. 2003; Giunchiglia and Maratea 2012; Gebser et al. 2011), but while the methods presented in those papers are defined for specific types of preferences, our algorithms can handle any preference type defined in ASP. In fact, most approaches one encounters in the literature can be modelled in our system. This includes established ASP optimization techniques like *#minimize* directives (Simons et al. 2002) and weak constraints (Leone et al. 2006), but also ASO (Brewka et al. 2003), the proposal in (Giunchiglia and Maratea 2012) and the language for specifying preferences in planning domains in (Son and Pontelli 2006). Arguably, the approach closest to ours is (Brewka 2004), in which a specific preference language with a predefined set of preference relations and combination methods is specified. However, this latter language is fixed and lacks the flexibility of our approach, and the only reasoning problem addressed is computing a single preferred model.

[10] A comparison between both systems modifying the heuristic of the underlying solver is left as future work.

Acknowledgement This work was partly supported by DFG research unit FOR 1513 (HYBRIS). The second author was partially supported by a Canadian NSERC Discovery Grant.

References

Apt, K.; Blair, H.; Walker, A. 1987. Towards a theory of declarative knowledge. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 89–148.

Banbara, M.; Soh, T.; Tamura, N.; Inoue, K.; Schaub, T. 2013. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming* 13(4-5):783–798.

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

Bienvenu, M.; Lang, J.; Wilson, N. 2010. From preference logics to preference languages, and back. In (Lin and Sattler 2010), 414–424.

Brewka, G.; Niemelä, I.; Truszczyński, M. 2003. Answer set optimization. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, 867–872. Morgan Kaufmann.

Brewka, G. 2004. Complex preferences for answer set optimization. *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, 213–223. AAAI Press.

Cabalar, P.; Son, T., eds. 2013. *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Springer.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Ricca, F.; Schaub, T. 2012. ASP-Core-2: Input language format.

Coste-Marquis, S.; Lang, J.; Liberatore, P.; Marquis, P. 2010. Expressive Power and Succinctness of Propositional Languages for Preference Representation. In (Lin and Sattler 2010), 203–212.

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

Delgrande, J.; Schaub, T.; Tompits, H.; Wang, K. 2004. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence* 20(2):308–334.

di Rosa, E.; Giunchiglia, E. 2013. Combining approaches for solving satisfiability problems with qualitative preferences. *AI Communications* 26(4):395–408.

Eiter, T.; Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence* 15(3-4):289–323.

Gebser, M.; Guziolowski, C.; Ivanchev, M.; Schaub, T.; Siegel, A.; Thiele, S.; Veber, P. 2010. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In (Lin and Sattler 2010), 497–507.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan and Claypool.

Gebser, M.; Jost, H.; Kaminski, R.; Obermeier, P.; Sabuncu, O.; Schaub, T.; Schneider, M. 2013a. Ricochet robots: A transverse ASP benchmark. In (Cabalar and Son 2013), 348–360.

Gebser, M.; Kaufmann, B.; Otero, R.; Romero, J.; Schaub, T.; Wanko, P. 2013b. Domain-specific heuristics in answer set programming. *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, 350–356. AAAI Press.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Schaub, T. 2014. *Clingo* = ASP + control: Preliminary report. *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*. Theory and Practice of Logic Programming, Online Supplement.

Gebser, M.; Kaminski, R.; Schaub, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming* 11(4-5):821–839.

Giunchiglia, E.; Maratea, M. 2012. Algorithms for solving satisfiability problems with qualitative preferences. *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, 327–344. Springer.

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

Lin, F.; Sattler, U., eds. 2010. *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*. AAAI Press.

Pipatsrisawat, K.; Darwiche, A. 2007. A lightweight component caching scheme for satisfiability solvers. *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, 294–299. Springer.

Potassco. http://potassco.sourceforge.net

Schaub, T.; Thiele, S. 2009. Metabolic network expansion with ASP. *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, 312–326. Springer.

Siddiqi, S. 2011. Computing minimum-cardinality diagnoses by model relaxation. *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI'11)*, 1087–1092. IJCAI/AAAI.

Simons, P.; Niemelä, I.; Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Son, T.; Pontelli, E. 2006. Planning with preferences using logic programming. *Theory and Practice of Logic Programming* 6(5):559–608.

Zhu, Y.; Truszczyński, M. 2013. On optimal solutions of answer set optimization problems. In (Cabalar and Son 2013), 556–568.