An Ordered Logic Program Solver

Davy Van Nieuwenborgh[⋆], Stijn Heymans, and Dirk Vermeir[⋆⋆]

Dept. of Computer Science
Vrije Universiteit Brussel, VUB
Pleinlaan 2, B1050 Brussels, Belgium
Telephone: +32495939799, Fax: +3226293525
{dvnieuwe,sheymans,dvermeir}@vub.ac.be

Abstract. We describe the design of the OLPS system, an implementation of the preferred answer set semantics for ordered logic programs. The basic algorithm we propose computes the extended answer sets of a simple program using an intuitive 9-valued lattice, called $\mathbf{T}_9$. During the computation, this lattice is employed to keep track of the status of the literals and the rules while evolving to a solution. It turns out that the basic algorithm needs little modification in order to be able to compute the preferred answer sets of an ordered logic program. We illustrate the system using an example from diagnostic reasoning and we present some preliminary benchmark results comparing OLPS with existing answer set solvers such as SMODELS and DLV.

Keywords: Preference, Answer Set Programming, Implementation

1 Introduction

In *answer set programming* (see e.g. [2] and the references therein), one uses a logic program to modularly describe the requirements that must be fulfilled by the solutions to a problem. The solutions then correspond to the models (answer sets) of the program, which are usually defined through (a variant of) the stable model semantics [13]. The technique has been successfully applied in problem areas such as planning [14, 6, 7], configuration and verification [20], diagnosis [5, 17, 24], game theory [25], updates [8] and database repairs [1, 15].

The *extended answer set* semantics for, possibly inconsistent, simple programs (containing only classical negation) is defined by allowing rules to be *defeated* (not satisfied). An *ordered logic program* then consists of a simple program with a partial order on the rules, representing a preference for satisfying certain rules, possibly at the cost of violating less important ones. Such a rule preference relation induces an order on extended answer sets, the minimal elements of which are called *preferred answer sets*. It can be shown [18] that the resulting semantics has a similar expressiveness as disjunctive logic programming, e.g. the membership problem is Σ_2^P-complete. Ordered programs have natural applications in e.g. database repair [15] or diagnosis [17, 24].

[⋆] Supported by the FWO.

[⋆⋆] This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

This paper describes the design and implementation of the OLPS system that can be used to compute the preferred answer sets of ordered programs. It is organized as follows: after a brief overview of the preferred answer set semantics for ordered programs (Section 2), we present the OLPS system in Section 3. Section 4 discusses an algorithm, based on *partial interpretations*, to compute the extended answer sets of a simple (unordered) program. In Section 5, this algorithm is adapted to take into account the rule order, and compute only preferred answer sets. Finally, Section 6 contains the results of some preliminary experiments and directions for further research.

The OLPS system has been released under the GPL and is available for download from `http://tinf2.vub.ac.be/olp`.

2 Preferred Answer Sets for Ordered Programs

Preliminaries and Notation. A *literal* is an *atom* a or a negated atom $\neg a$. For a literal l we use $\neg l$ to denote its inverse, i.e. $\neg l = \neg a$ iff $l = a$ while $\neg l = a$ iff $l = \neg a$. For a set of literals X, we use $\neg X$ to denote $\{\neg l \mid l \in X\}$. Such a set is *consistent* iff $X \cap \neg X = \emptyset$. In addition, we also consider the special symbol $\perp$ denoting contradiction. Any set $X \cup \{\perp\}$, with X a set of literals, is inconsistent. For a set of atoms A, we use $\mathcal{L}_A$ to denote the set of literals over A and define $\mathcal{L}_A^{\perp} = \mathcal{L}_A \cup \{\perp\}$.

A *rule* r is of the form $h_r \leftarrow b_r$ where b_r, the *body* of the rule, is a set of literals and h_r, the rule's *head*, is a literal or $\perp$. In the latter case, the rule is called a *constraint*[1], in the former case, it is called a h_r-rule.

For a set of rules R we use $R^\star$ to denote the unique smallest Herbrand model, see [22], of the positive logic program obtained from P by considering all literals and $\perp$ as separate atoms.

Simple Logic Programs and Extended Answer Sets. A *simple logic program* (SLP) is a countable set of rules. For a SLP P, we use $\mathcal{B}_P$ to denote its *Herbrand base*, i.e. the set of atoms appearing in the rules of P. An *interpretation* for P is any consistent subset of $\mathcal{L}_{\mathcal{B}_P}$. For an interpretation I and a set of literals X we write $I \models X$ just when $X \subseteq I$.

A rule $r = h_r \leftarrow b_r$ is *satisfied* by I, denoted $I \models r$, iff $h_r \in I$ whenever $I \models b_r$, i.e. whenever r is *applicable* ($I \models b_r$), it must be *applied* ($I \models b_r \cup \{h_r\}$); r is *defeated* by I, denoted $I \models \neg r$ iff there is an applied *competing* rule $r' = \neg h_r \leftarrow b_{r'}$. Note that, consequently, constraint rules cannot be defeated.

The semantics defined below deals with possibly inconsistent programs in a simple, yet intuitive, way: when faced with contradictory applicable rules for l and $\neg l$, one selects one, e.g. the l-rules, for application and ignores (defeats) the contradicting $\neg l$-rules.

Let I be an interpretation for a SLP P. The *reduct* of P w.r.t. I, denoted P_I is the set of rules satisfied by I, i.e. $P_I = \{r \in P \mid I \models r\}$. An interpretation I is called an *extended answer set* of P iff I is *founded*, i.e. $P_I^\star = I$, and each rule r in P is either satisfied or defeated, i.e. $\forall r \in P \cdot I \models r \vee I \models \neg r$.

[1] To simplify the theoretical treatment we use an explicit contradiction symbol $\perp$ in the head of constraint rules. The concrete OLPS syntax employs the usual notation where the head of a constraint is empty.

Example 1. The program P_1 shown below has 2 extended answer sets $\{\neg a, b\}$ and $\{a, \neg b\}$ corresponding to the reducts $\{c, r_{\neg a}, r_a, r_b\}$ and $\{c, r_{\neg b}, r_a, r_b\}$, respectively.

$$\begin{array}{lll} r_{\neg a} : \neg a \leftarrow & r_a : \ a \leftarrow \neg b & r_{\neg b} : \neg b \leftarrow \\ r_b : \quad b \leftarrow \neg a & c : \bot \leftarrow \neg a, \neg b & \end{array}$$

Ordered Programs and Preferred Answer Sets. An *ordered logic program* (OLP) is a pair $\langle R, < \rangle$ where R is a simple program and $<$ is a well-founded strict[2] partial order on the rules in R[3].

Intuitively, $r_1 < r_2$ indicates thats r_1 is preferred over r_2. The notation is extended to sets of rules, e.g. $R_1 < R_2$ abbreviates $\bigwedge_{r_1 \in R_1 \wedge r_2 \in R_2} r_1 < r_2$.

The preference $<$ on rules in $\langle R, < \rangle$ will be translated to a preference relation on the extended answer sets of R via an ordering on reducts: a reduct R_1 is preferred over a reduct R_2, denoted $R_1 \sqsubseteq R_2$ iff $\forall r_2 \in R_2 \backslash R_1 \cdot \exists r_1 \in R_1 \backslash R_2 \cdot r_1 < r_2$, i.e. each rule from $R_2 \backslash R_1$ is "countered" by a rule in $R_1 \backslash R_2$. It can be shown (Theorem 6 in [15]) that $\sqsubseteq$ is a partial order on 2^R. Consequently, we write $R_1 \sqsubset R_2$ just when $R_1 \sqsubseteq R_2$ but $R_1 \neq R_2$. The $\sqsubseteq$-order on reducts induces a preference order on the extended answer sets of R: for extended answer sets M_1 and M_2, $M_1 \sqsubseteq M_2$ iff $R_{M_1} \sqsubseteq R_{M_2}$. Minimal (according to $\sqsubset$) extended answer sets of R are called *preferred answer sets* of $\langle R, < \rangle$. An extended answer set is called *proper* iff it satisfies all minimal elements from R.

Example 2. Consider the ordered program below, which is written using the OLPS-syntax: $\neg$ is written as "`-`" and rules are grouped in modules that are partially ordered using statements of the form "`A < B`".

```
Avoid { pass :- study .   study .   }
 Prefer { -study . }
ForSure { -pass :- -study .   pass :- -pass .   }
ForSure < Prefer < Avoid
```

The program expresses the dilemma of a person preferring not to study but aware of the fact that not studying leads to not passing (`-pass :- -study`) which is unacceptable (`pass :- -pass`). It is straightforward to verify that the single (proper) preferred answer set is $\{study, pass\}$ which satisfies all rules in `ForSure` and `Avoid`, but not the rules in `Prefer`.

In [15, 16] it is shown that OLP can simulate negation as failure (i.e. adding negation as failure does not increase the expressiveness of the formalism) as well as disjunction (under the minimal possible model semantics) and that e.g. membership is Σ_2^P-complete. This makes OLP as expressive as disjunctive logic programming under its normal semantics. However, as with logic programming with ordered disjunction[4], no effective translation is known in either direction.

[2] A strict partial order $<$ on a set X is a binary relation on X that is antisymmetric, anti-reflexive and transitive. The relation $<$ is well-founded if every nonempty subset of X has a $<$-minimal element.

[3] Strictly speaking, we should allow R to be a multiset or, equivalently, have labeled rules, so that the same rule can appear in several positions in the order. For the sake of simplicity of notation, we will ignore this issue: all results also hold for the general multiset case.

3 The Ordered Logic Program Solver (OLPS) System

OLPS computes (a selection of) the proper preferred answer sets of a finite ordered program which is described using a sequence of module definitions and order assertions. A module is specified using a module name followed by a set of rules, enclosed in braces while an order assertion is of the form $m_0 < m_1 < \ldots < m_n$, $n > 0$, where each m_i, $0 \leq i \leq n$ is a module name.

Rules are written as usual in datalog, with a few exceptions: variables must start with an uppercase letter and classical negation ($\neg$) is represented by a "`-`" in front of a literal, e.g. `-p(X,a)`. In addition, some convenient syntactic sugar constructs can be used in non-grounded programs. E.g. rules such as `t({1,2-4,a}).` abbreviate `t(1).t(2).t(3).t(4).t(a).` and variables can be "typed", where a type is a unary predicate: e.g. `p(X:t) :- q(Y:r,Z).` abbreviates `p(X) :- q(Y,Z), t(X), r(Y).`

The example program in Figure 1 describes the operation of a unary adder, as shown in Figure 2 [12]. It illustrates how ordered programs can be used to implement diagnostic systems [17].

```
Error { fault (N:gate , F: fault ). } % May be needed to explain observation.
Default { − fault (N:gate , F: fault ).  % By default, gates are not faulty.
  −adder(X:bit , Y:bit , Z: bit , Sum:bit , Carry: bit ). % Naf for adder/5.
  }
Model { bit ({0, 1}).   gate({xor1 , xor2 , and1 , and2 , or1 }).
   fault ({ stuck_at_0 , stuck_at_1 }).
  xor(N:gate ,0,0,1) :−  fault (N, stuck_at_1 ). xor(N:gate ,1,1,1) :−  fault (N, stuck_at_1 ).
  xor(N:gate ,0,1,0) :−  fault (N, stuck_at_0 ). xor(N:gate ,1,0,0) :−  fault (N, stuck_at_0 ).
  and(N:gate ,1,1,0) :−  fault (N, stuck_at_0 ). and(N:gate ,1,0,1) :−  fault (N, stuck_at_1 ).
  and(N:gate ,0,1,1) :−  fault (N, stuck_at_1 ). and(N:gate ,0,0,1) :−  fault (N, stuck_at_1 ).
  or(N:gate ,1,1,0) :−  fault (N, stuck_at_0 ). or(N:gate ,1,0,0) :−  fault (N, stuck_at_0 ).
  or(N:gate ,0,1,0) :−  fault (N, stuck_at_0 ). or(N:gate ,0,0,1) :−  fault (N, stuck_at_1 ).
  % Normal model
  adder(X: bit , Y:bit , Z: bit , Sum:bit , Carry: bit ) :−
    xor(xor1 , X,Y,S ), xor(xor2 , Z,S,Sum), and(and1 , X,Y,C1), and(and2 , Z,S,C2),
    or(or1 , C1,C2,Carry).
  % Normal behaviour of gates.
  xor(N:gate , 1,1,0).   xor(N:gate , 0,1,1).   xor(N:gate , 1,0,1).   xor(N:gate , 0,0,0).
  and(N:gate , 1,1,1).   and(N:gate , 1,0,0).   and(N:gate , 0,1,0).   and(N:gate , 0,0,0).
  or(N:gate , 1,1,1).    or(N:gate , 1,0,1).    or(N:gate , 0,1,1).    or(N:gate , 0,0,0).
}
Observations { :− −adder (0,0,1,0,1).  }
Model < Default < Error
```

Fig. 1. A program for circuit diagnosis.

Intuitively, observations are represented using constraints, and rules describing the normal operation of the system are preferred over "fault rules" that specify possible ab-

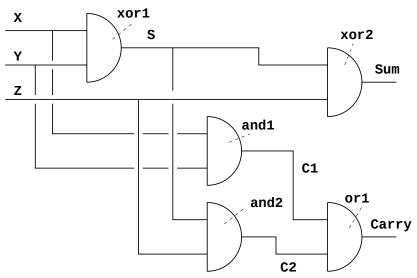


Fig. 2. Unary adder [12] described in the program of Figure 1.

normal behaviors. Here, the *adder*-rule in *Model* describes the normal operation of the circuit where variables correspond to the connections between the gates, which are named in the *gate/1*-predicate. It is assumed that a broken gate may have a fixed output, whatever its inputs. This leads to the introduction of two constants *stuck_at_0* and *stuck_at_1* (defined in the *fault/1* rules) and a specification of the behavior of the various gate types when they are stuck using rules such as *xor(N:gate, 0, 0, 1) :- fault(N, stuck_at_1).* The *Default* module specifies that *fault/2* and *adder/5* are false by default (*Model* < *Default*).

To add diagnostic capabilities, it suffices to add another weaker module *Error* that contains rules that should only be used "as a last resort".

The observation of a malfunctioning circuit is described using a constraint, e.g. *:- -adder(0,0,1,0,1)* forces OLPS to find an explanation for *adder(0,0,1, 0,1)*. To this end, some rules in *Default* will need to be defeated by applying some weaker rules from *Error*. As shown in [17], each preferred answer set will contain a (subset) minimal set of *fault/2* literals.

Running OLPS on the example using the command[4]

```
    olps -p 'fault/2' -n 0 circuit.olp
```

will compute the possible minimal explanations shown below.

```
{ + fault (xor1 ,  stuck_at_1 ) }
{ + fault (or1 ,  stuck_at_1 ) + fault (xor2 ,  stuck_at_0 ) }
{ + fault (and2 ,  stuck_at_1 ) + fault (xor2 ,  stuck_at_0 ) }
{ + fault (and1 ,  stuck_at_1 ) + fault (xor2 ,  stuck_at_0 ) }
```

Like SMODELS[21], OLPS first produces a grounded version of the program that then serves as input to the solver proper. The default grounding[5], *olpg*, produces all (some are, however, optimized away) the instances of rules that are used in the computation of the minimal answer set of the positive program, obtained by considering all literals as separate atoms.

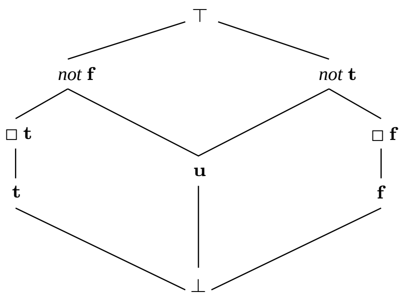
[4] The "`-p`" option is used to print only the *fault/2* predicate, "`-n 0`" will cause the system to compute all proper preferred answer sets.

[5] The grounding program runs as a separate process and can be selected at run time.

4 Computing Extended Answer Sets for Simple Programs

Partial Interpretations

OLPS searches for answer sets by building and extending *partial interpretations* that carry intermediate information regarding the status of literals and rules. To represent such information on literals, we use the lattice $\mathbf{T}_9$ of truth values depicted in Figure 3. Intuitively, $\mathbf{T}_9$ can be considered as an extension of FOUR from [3, 19] with approximations[6] $\Box\,\mathbf{t}$ and $\Box\,\mathbf{f}$ of resp. $\mathbf{t}$ and $\mathbf{f}$, denoting that a literal must eventually become resp. true or false at the end of the computation in order for a partial interpretation to result in an extended answer set. Further, we use *not* $\mathbf{t}$ and *not* $\mathbf{f}$ as explicit representations of the complements of $\mathbf{t}$ and $\mathbf{f}$. Clearly, the order $\sqsubset$ in $\mathbf{T}_9$ corresponds to the "knowledge" ordering[3, 19], i.e. $t_1 \sqsubset t_2$ indicates that t_1 is more determined than t_2.



	Intuition
$\bot$	Contradiction.
$\top$	No information.
$\mathbf{t}$	True.
$\mathbf{f}$	False.
$\Box\,\mathbf{t}$	Eventually true.
$\Box\,\mathbf{f}$	Eventually false.
not $\mathbf{t}$	Not true.
not $\mathbf{f}$	Not false.
$\mathbf{u}$	Neither true nor false.

t	$\neg t$
$\bot$	$\bot$
$\top$	$\top$
$\mathbf{t}$	$\mathbf{f}$
$\mathbf{f}$	$\mathbf{t}$
$\Box\,\mathbf{t}$	$\Box\,\mathbf{f}$
$\Box\,\mathbf{f}$	$\Box\,\mathbf{t}$
not $\mathbf{t}$	*not* $\mathbf{f}$
not $\mathbf{f}$	*not* $\mathbf{t}$
$\mathbf{u}$	$\mathbf{u}$

Fig. 3. Truth value lattice $\mathbf{T}_9$.

The general idea behind the usage of $\mathbf{T}_9$ is to start with $\top$ ("no information") for each literal and evolve during the computation towards either $\mathbf{t}$, $\mathbf{f}$, $\mathbf{u}$ or $\bot$, taking the knowledge ordering $\sqsubset$ into account. When, at the end of the computation, a partial interpretation assigns either $\mathbf{t}$, $\mathbf{f}$or $\mathbf{u}$ to each literal, we have found an extended answer set.

Definition 1. *A $\mathbf{T}_9$-**valuation** on a set of atoms A is a total function ϕ assigning a truth value $\phi(a)$ to each $a \in A$; it is extended to literals over A by defining $\phi(\neg a) = \neg\phi(a)$, for $a \in A$. A valuation ϕ is **consistent** iff $\phi^{-1}(\bot) = \emptyset$. It is **final** iff it assigns only to truth values that cannot be improved without introducing contradiction, i.e. $\forall t \notin \{\mathbf{t}, \mathbf{f}, \mathbf{u}\} \cdot \phi^{-1}(t) = \emptyset$.*

The order in $\mathbf{T}_9$ induces a partial ordering on valuations: ϕ_1 *extends* ϕ_2, denoted $\phi_1 \sqsubseteq \phi_2$, iff $\phi_1(a) \sqsubseteq \phi_2(a)$ for all $a \in A$. Intuitively, $\phi_1 \sqsubset \phi_2$ (i.e. $\phi_1 \sqsubseteq \phi_2$ and $\phi_1 \neq \phi_2$) if ϕ_1 is more determined than ϕ_2.

$\mathbf{T}_9$-valuations will be represented as sets of extended literals where an *extended literal* is a literal or of one of the forms $\Box\,l$ or *not* l, with l an ordinary literal. For an

[6] The notation $\Box\,t$ should not be confused with the "always" modality from modal logic.

extended literal e, we use $\widehat{e}$ to denote the underlying atom, i.e. $\widehat{\neg a} = \widehat{a} = a$, while $\widehat{\Box\, l} = \widehat{\mathit{not}\, l} = \widehat{l}$. For a set of extended literals E, $\widehat{E}$ abbreviates $\{\widehat{e} \mid e \in E\}$. The set of all extended literals over a set of atoms A is denoted $\mathcal{E}_A$ while $\mathcal{E}_A^{\perp} = \mathcal{E}_A \cup \{\perp\}$. For a set of literals X, $\Box\, X$ abbreviates $\{\Box\, x \mid x \in X\}$.

We associate a truth value $v(e)$ from $\mathbf{T}_9$ with an extended literal e, where $\widehat{e} = a$, in the obvious way: $v(a) = \mathbf{t}$, $v(\mathit{not}\ a) = \mathit{not}\,\mathbf{t}$ and $v(\Box\, a) = \Box\,\mathbf{t}$ while $v(\neg a) = \neg(v(a)) = \mathbf{f}$, $v(\mathit{not}\,\neg a) = \neg v(\mathit{not}\, a) = \mathit{not}\,\mathbf{f}$ and $v(\Box\,\neg a) = \neg v(\Box\, a) = \Box\,\mathbf{f}$. Using v, we can interpret a set E of extended literals as a valuation ϕ_E by defining

$$\phi_E(a) = \sqcap\{v(e) \mid e \in E\ \wedge\ \widehat{e} = a\}$$

where, by definition, $\sqcap\emptyset = \top$. E.g., if $E = \{\Box\, a, a, \mathit{not}\, b, \mathit{not}\,\neg b, \Box\, c, \mathit{not}\, c\}$ is a set of extended literals over $\{a, b, c, d\}$ then $\phi_E(a) = \mathbf{t}$, $\phi_E(b) = \mathbf{u}$, $\phi_E(c) = \perp$ and $\phi_E(d) = \top$. A set of extended literals E is consistent and/or final iff ϕ_E is consistent and/or final. Obviously, if $E_1 \subseteq E_2$, then $\phi_{E_2} \sqsubseteq \phi_{E_1}$.

A set of extended literals E_1 *extends* a set E_2, denoted $E_1 \sqsubseteq E_2$ iff $\phi_{E_1} \sqsubseteq \phi_{E_2}$. A *conservative extension* of a set of extended literals E is any superset $E' \supseteq E$ that preserves the associated valuation, i.e. $\phi_{E'} = \phi_E$. Since the set of conservative extensions of a set of extended literals is closed under union, we can define the *closure* $\overline{E}$ of a set of extended literals E as the unique maximal conservative extension of E. E.g., the closure of $E = \{\Box\, a, a, \mathit{not}\, b, \mathit{not}\,\neg b, \Box\, c, \mathit{not}\, c\}$ is $\overline{E} = \{\Box\, a, a, \mathit{not}\,\neg a, \mathit{not}\, b, \mathit{not}\,\neg b, \Box\, c, \Box\,\neg c, c, \neg c, \mathit{not}\, c, \mathit{not}\,\neg c\}$. It can be shown that, for sets of extended literals E_1 and E_2, $E_1 \sqsubseteq E_2$ iff $\overline{E_2} \subseteq \overline{E_1}$.

For a set of extended literals E we write that $E \models F$, with F a set of extended literals, iff $F \subseteq \overline{E}$.

In the sequel, we will often abuse notation by considering a set of rules R also as a set of atoms (disjoint from $\mathcal{B}_R$), one for each rule $r \in R$, thus defining e.g. $\mathcal{L}_R$.

Definition 2. *A* ***partial interpretation*** *of a simple program R is a set $I \subseteq \mathcal{L}_R \cup \mathcal{E}_{\mathcal{B}_R}^{\perp}$. Intuitively, the rule literals $I_R = I \cap \mathcal{L}_R$ represent the desired status of the rules from R: if $r \in I_R$ then r should be satisfied while $\neg r \in I_R$ indicates that r should be defeated. $I_L = I \cap \mathcal{E}_{\mathcal{B}_R}$ represents a valuation of $\mathcal{B}_R$. The* ***reduct*** *of R w.r.t. I, denoted R_I is defined by $R_I = \{r \mid r \in I_R\}$. A partial interpretation I is*

- ***complete*** *iff $\widehat{I_R} = R$, i.e. each rule has a desired status;*
- ***consistent*** *iff $\perp \notin I$, both I_R and ϕ_{I_L} are consistent and, moreover, there exists a final consistent extension $F \sqsubseteq I_L$ such that $\forall l \in I_R \cdot F \cap \mathcal{L}_{\mathcal{B}_R} \models l$, i.e. I_R is consistent with I_L;*
- ***final*** *iff ϕ_{I_L} is final; and*
- ***founded*** *iff $(R_I)^\star = I_L \cap \mathcal{L}_{\mathcal{B}_R}^{\perp}$.*

A partial interpretation J ***extends*** *a partial interpretation I, denoted $I \sqsubseteq J$ iff $I_R \cup \overline{I_L} \subseteq J_R \cup \overline{J_L}$.*

Note that a partial interpretation need not be consistent. It is easily seen that $\sqsubseteq$ defines a partial order on partial interpretations and that all extensions of an inconsistent partial interpretation are themselves inconsistent.

Extended answer sets correspond to partial interpretations that are complete, consistent, final and founded.

Proposition 1. *Let R be a simple program. If M is an extended answer set of R then*

$$\Pi_R(M) = R_M \cup \neg(R \setminus R_M) \cup M \cup \bigcup_{a \in \mathcal{B}_R \setminus \widehat{M}} \{not\, a, not\, \neg a\}$$

is a partial interpretation that is complete, consistent, final and founded. Conversely, $I \cap \mathcal{L}_{\mathcal{B}_R}$ is an extended answer set for any partial interpretation I that is complete, consistent, final and founded.

Note that the last component of $\Pi_R(M)$ corresponds to a version of the closed world assumption: any literal l for which no information is available is assumed to be "necessarily unknown", i.e. $\phi_{\Pi_R(M)}(l) = \mathbf{u}$.

A rule r is *blocked* w.r.t. a set of extended literals E iff $\exists l \in b_r \cdot E \models not\, l$. If r is not blocked w.r.t. E, it is said to be *open*. We use $R_h(E)$ to denote the sets of h-rules from R that are open w.r.t. E. An open rule r is *applicable* w.r.t. E iff $E \models \Box\, b_r$, it is *applied* iff it is applicable, $h_r \neq \bot$, and, moreover, $E \models \Box\, h_r$.

For a given partial interpretation I, we need an operator $\Phi^\star_R(I)$ to compute the maximal deterministic extension of I. This operator is based on the primitive notion of forcing, that, for a partial interpretation I and a single rule r, defines which new information can be deterministically derived from I and r.

Definition 3. *A partial interpretation I for an SLP R* ***forces*** *a set of (extended or rule) literals J, denoted $I \Vdash J$ iff $X \Vdash J$ (and I fulfills the extra condition, if any) for some $X \subseteq I_R \cup \overline{I_L}$ where $\Vdash$ is defined below.*

$$\Vdash \{r\} \qquad \textit{if } h_r = \bot \tag{1}$$
$$\{\neg r\} \Vdash \Box\, b_r \cup \{\Box\, \neg h_r\} \qquad \textit{if } h_r \neq \bot \tag{2}$$
$$\{not\, \neg h_r\} \Vdash \{r\} \tag{3}$$
$$\{r\} \cup \Box\, b_r \Vdash \{\Box\, h_r\} \qquad \textit{if } h_r \neq \bot \tag{4}$$
$$\{r\} \cup \Box\, b_r \Vdash \{\bot\} \qquad \textit{if } h_r = \bot \tag{5}$$
$$\{r\} \cup b_r \Vdash \{h_r\} \tag{6}$$
$$\Box\, b_r \cup \{not\, h_r\} \Vdash \{\neg r\} \tag{7}$$
$$\{\Box\, h_r\} \Vdash \{r\} \cup \Box\, b_r \qquad \textit{if } R_{h_r}(I_L = I \cap \mathcal{E}_{\mathcal{B}_R}) = \{r\} \tag{8}$$
$$\Box\,(b_r \setminus \{b\}) \cup \{r, not\, h_r\} \Vdash \{not\, b\} \qquad \textit{if } b \in b_r \tag{9}$$
$$\Box\,(b_r \setminus \{b\}) \cup \{r\} \Vdash \{not\, b\} \qquad \textit{if } b \in b_r \textit{ and } h_r = \bot \tag{10}$$
$$\Vdash \{not\, h_r\} \qquad \textit{if } h_r \neq \bot \textit{ and } R_{h_r}(I_L) = \emptyset \tag{11}$$
$$\{not\, b\} \Vdash \{r\} \qquad \textit{if } b \in b_r \tag{12}$$

Intuitively, (1) asserts that constraints cannot be defeated while (2) encodes the definition of defeat: $\neg r$, i.e. r is defeated, iff r is applicable but $\neg h_r$ is implied by some defeating rule. Consequently, if $\neg h_r$ cannot be true, the rule r must be satisfied (3). Definitions (4,5) and (6) encode (satisfied) rule application while (7) expresses that an applicable rule that cannot be applied must be defeated. Definition (8) indicates that if only a single rule is available to motivate a needed literal, it must eventually become applied. On the other hand, an almost applicable satisfied rule with a conclusion that is inconsistent with the interpretation must be blocked (9,10). If there are no open rules for a literal a, then *not* a must hold (11). Finally, a blocked rule must be satisfied (12).

Definition 4. *Let R be a finite simple program. The operator Φ_R is defined by $\Phi_R(I) = I \cup \bigcup_{I \Vdash X} X$, for any partial interpretation I. The closure $\Phi^\star_R$ of Φ_R is defined by $\Phi^\star_R(I) = \bigcup_{n>0} \Phi^n_R(I)$.*

It can be shown that $\Phi^\star_R(I)$ is unique and extends I, i.e. $I \sqsubseteq \Phi^\star_R(I)$.

Clearly, $\Phi^\star_R(I)$ computes the maximal deterministic extension of a partial interpretation I. It encompasses the Fitting operator[11] and plays a similar role as does the function *det_cons* in DLV[9], or *expand* in SMODELS[21].

Example 3. Reconsider program P_1 from Example 1 and the interpretation $I = \{r_{\neg a}\}$. The table below illustrates a possible computation of $\Phi^\star_{P_1}(I)$.

$\{r_{\neg a}\} \Vdash \{\neg a\}$	(6)	$\{\neg r_{\neg b}\} \Vdash \{\Box\, b\}$	(2)
$\Vdash \{c\}$	(1)	$\{\Box\, b\} \Vdash \{r_b, \Box\, \neg a\}$	(8)
$\{c, \neg a\} \Vdash \{\textit{not}\ \neg b\}$	(10)	$\{\textit{not}\ \neg b\} \Vdash \{r_a\}$	(12)
$\{\textit{not}\ \neg b\} \Vdash \{\neg r_{\neg b}\}$	(7) on $r_{\neg b}$	$\{r_b, \neg a\} \Vdash \{b\}$	(6) on r_b

Thus, $\Phi^\star_{P_1}(I) = \{r_{\neg a}, \neg r_{\neg b}, r_a, r_b, c, \neg a, b\} = \Pi_{P_1}(\{\neg a, b\})$.

Consistency is easy to check for fixpoints of Φ_R.

Proposition 2. *Let I be a partial interpretation of a simple program R such that $\Phi_R(I) = I$. Then I is consistent iff $\bot \notin I$ and both I_R and I_L are consistent.*

The following is an easy consequence of (6) in Definition 3.

Proposition 3. *Let I be a partial interpretation of a simple program R. If I is founded then so is $\Phi^\star_R(I)$.*

Complete founded fixpoints of Φ_R have no consistent founded proper extensions.

Proposition 4. *Let I be a consistent complete founded partial interpretation of a simple program R such that $\Phi_R(I) = I$. Then $J = I$ for all $I \sqsubseteq J$ such that J is consistent and founded.*

Replacing an interpretation I by $\Phi^\star_R(I)$ does not loose any answer sets.

Proposition 5. *Let I be an interpretation of a simple program R. Any extended answer set M of R that extends I, i.e. $I \sqsubseteq \Pi_R(M)$, also extends $\Phi^\star_R(I)$, i.e. $\Phi^\star_R(I) \sqsubseteq \Pi_R(M)$.*

The following example shows that consistent maximal (and thus complete) founded extensions are not necessarily final.

Example 4. Consider the simple program P_2 shown below and the empty partial interpretation.

$$\begin{array}{lll} r_0 : \neg a \leftarrow & r_1 : \ b \leftarrow a & r_2 : c \leftarrow b \\ r_3 : \ \ a \leftarrow c & r_4 : \bot \leftarrow \neg a & \end{array}$$

It is straightforward to verify that $\Phi^\star_{P_2}(\emptyset) = \{r_4, \neg r_0, r_1, r_2, r_3, \textit{not}\ \neg a, \Box\, a, \Box\, b, \Box\, c\}$ does not correspond to an extended answer set (in fact, P_2 does not have any extended answer sets). Intuitively, r_0 cannot be defeated because the only possible motivation for a is based on a circularity.

A set such as $\{\Box a, \Box b, \Box c\}$ in Example 4 is called *unfounded*[7]. Formally, a set X, $X \subseteq \Box \mathcal{L}_{\mathcal{B}_P}$, is unfounded w.r.t. a partial interpretation I iff, for any $\Box l \in X$, each non-blocked (w.r.t. I) l-rule r contains a literal $d \in b_r$ such that $\Box d \in X$. It can be shown that if I contains an unfounded set, then there are no extended answer sets among its extensions. This result is used in the *prune* function from Figure 4.

The *aset* Procedure

The main procedure for enumerating extended answer sets that are extensions of a given partial interpretation is shown in Figure 4. Note that the $select$ function returns an arbitrary rule from its argument set.

```
 PartialInterpretation
prune(const Program& R, PartialInterpretation $I$) {
J = $\Phi^{\star}_{R}(I)$;
if (J contains an unfounded set)
  J = J $\cup \{\bot\}$;
return J;
}

set< Interpretation >
aset (const Program& R, PartialInterpretation $I$) {
// Precondition: $I$ is founded and $\Phi_R(I) = I$.
if (! $I$ is consistent ) // Easy to check because of Proposition 2.
  return $\emptyset$; // There are no answer sets extending $I$.
if ( $I$ is complete) {
  if ( $I$ is final ) // $I$ corresponds to an answer set by Proposition 1.
    return $\{I \cap (\mathcal{B}_R \cup \neg\mathcal{B}_R)\}$;
  else return $\emptyset$; // By Proposition 4, there are no answer sets extending $I$.
  }
else {
  Rule r = select ($R \backslash \widehat{I_R}$);
  // The preconditions for the calls are assured by Proposition 3..
  return aset(R, prune($I \cup \{r\}$) $\cup$ aset(R, prune($I \cup \{\neg r\}$);
  }
}
```

Fig. 4. The *aset* function for simple programs.

Proposition 6. *Let R be a simple program and let I be a founded fixpoint of Φ_R. Then $aset(R, I)$ will return all extended answer sets of R that extend I.*

From Proposition 6, it follows that all extended answer sets of R can be obtained using the call $aset(R, \Phi^{\star}_{R}(\emptyset))$.

[7] We use the term “unfounded” in this context since the intuition behind it is similar to unfounded sets in the well-founded semantics[23].

The implementation of $\Phi^{\star}_{R}$ uses a queue of pattern occurrences, each pattern corresponding to the left hand side of one of the rules in Definition 3. The queue is processed by adding the right hand side of the pattern to the partial interpretation, thus possibly generating further patterns for the queue. The computation finishes when an inconsistency is detected or the queue becomes empty. This design is sound because a pattern remains applicable in any consistent extension of the partial interpretation where it was first detected. Detection is facilitated by keeping some derived information such as the number of "open" literals in rule bodies, the number of open rules for a given literal etc.

5 Computing Preferred Answer Sets

A naive way to compute preferred answer sets would be to compute all extended answer sets and then retrieve the minimal (according to $\sqsubset$) elements.

OLPS tries instead to detect (and prune) partial interpretations that cannot lead to preferred answer sets as soon as possible. This is done by (a) always extending a partial interpretation I using a minimal rule (among the "open" rules), and (b) checking, for each previously found preferred answer set M, whether it is still possible to find a set of rules $N \subseteq R$ such that $\{r \in R \mid r \in I_R\} \subseteq N$ and $R_M \not\sqsubset N$.

In this context, a module[8] $X \subseteq R$ is said to be *decided* by a partial interpretation I when, by abuse of notation, $X \subseteq \hat{I_R}$, i.e. each rule $r \in X$ has a status in I. Further, two partial interpretations I and J are *equal* w.r.t. a module X iff $I_X = J_X$, i.e. they have the same status for the rules in X.

For a complete partial interpretation I and an arbitrary partial interpretation J, we say that I is *incomparable* w.r.t. J iff there exist a module $X \subseteq R$ such that

- $(J_X \cap X) \backslash (I_X \cap X) \neq \emptyset$, i.e. J has at least one satisfied rule in X that is defeated by I; and
- every module $Y \subseteq R$ with $Y < X$ is decided by J and, moreover, I and J are equal w.r.t. Y.

On the other hand, I is *stronger* than J iff for each module X which is such that I and J are equal w.r.t. all more preferred modules $Y < X$[9], it holds that

- $(X \cap J_X) \subsetneq (X \cap I_X)$ i.e. I satisfies strictly more rules in X than does J; and
- $(X \backslash \hat{J}) \subseteq I$, i.e. all rules in X that have not yet a status in J are satisfied w.r.t. I.

The following are easy consequences of the above definitions.

Proposition 7. *Let I be a complete partial interpretation and let J be a partial interpretation. I incomparable w.r.t. J implies that $R_{I_L} \not\sqsubset R_{K_L}$ for every extension K of J.*

Proposition 8. *Let I be a complete partial interpretation and let J be a partial interpretation. I stronger than J implies that $R_{I_L} \sqsubset R_{K_L}$ for every extension K of J.*

[8] We use the term *module*, just as in the syntax of OLPS, to denote a maximal set of rules $X \subseteq R$ that are all at the same position in the well-founded strict partial order on R. Clearly, this order on rules induces an equivalent order on the modules.

[9] This implies that Y is decided by J.

Clearly, checking incomparability or being stronger can be performed, even in the absence of optimization, in linear time and space (w.r.t. the size of the program).

Importing these checks into an adapted version of the prune function, as shown in Figure 5 ensures an early detection of a situation where no extended answer sets that extend I can be minimal.

```
⟨PartialInterpretation , $set < CompletePartialInterpretation >$⟩) {
 preferred_prune (const Program& R,
                  ⟨PartialInterpretation $I, set < CompletePartialInterpretation > P$⟩) {
J = $\Phi^{\star}_{R}(I)$;
if (J contains an unfounded set ) {
  J = J $\cup \{\bot\}$;
  return $\langle J, P\rangle$;
}
for each $T \in P$ {
  if $T$ incomparable w.r.t. $J$
        P = $P \backslash \{T\}$; // Due to Proposition 7.
  else if $T$ stronger than $J$ {
        J = J $\cup \{\bot\}$; // Due to Proposition 8.
        return $\langle J, P\rangle$;
  }
}
return $\langle J, P\rangle$;
}
```

Fig. 5. The *preferred_prune* function for ordered programs.

The procedure for finding preferred answer sets is shown in Figure 6. It can be shown that, if I is founded and $\Phi_R(I) = I$, then *preferred_aset(R,*$\langle I, P\rangle$*)* will return the set of all minimal (according to $\sqsubset$) extended answer sets M of R that extend I and such that no $T \in P$ exists for which $T \sqsubset M$ holds. It follows that *preferred_aset(R,*$\langle \Phi^{\star}_{R}(\emptyset), \emptyset\rangle$*)* computes all preferred answer sets of $\langle R, <\rangle$[10].

6 Conclusions and Directions for Further Research

Some preliminary tests of the current implementation have been conducted on a 2GHz Linux PC. The results are shown in Table 1: *circuit* refers to the program of Figure 1 while *ham-N* and *ham-dN* refer to programs that solve the Hamiltonian circuit problem on a randomly generated graph with N nodes and $N^2/10$, resp. $N^2/2$, edges. Note that the latter problem is Σ^P_1-complete and thus directly solvable by both SMODELS, DLV and OLPS. In [15] a transformation is presented of non-disjunctive seminegative programs into ordered programs where the preferred answer sets of the latter coincide with the classical subset minimal answer sets of the former. We have used this transformation to conduct our experiments with OLPS. It is clear from the table that the current naive

[10] Proper preferred answer sets are obtained by *preferred_aset(R,*$\langle \Phi^{\star}_{R}(R_{min}), \emptyset\rangle$*)*, with R_{min} contains the (atoms corresponding to the) $<$-minimal elements of R.

```
set< Interpretation >
 preferred_aset ( const Program& $R$,
                 ⟨PartialInterpretation $I$, $set < CompletePartialInterpretation > P$⟩) {
// Precondition: $I$ is founded and $\Phi_R(I) = I$.
if (! $I$ is consistent ) // Easy to check because of Proposition 2.
  return $\emptyset$; // There are no answer sets extending $I$.
if ( $I$ is complete) {
  if ( $I$ is final ) // $I$ corresponds to an answer set by Proposition 1.
    return $\{I \cap (\mathcal{B}_R \cup \neg\mathcal{B}_R)\}$;
  else return $\emptyset$; // By Proposition 4, there are no answer sets extending $I$.
  }
else {
  Rule r = select_min ($R \backslash \widehat{I_R}$);
  // Note that $n \not\sqsubset m$ for any $n \supseteq (R \cup \{\neg r\}), m \supseteq (R \cup \{r\})$.
  set< Interpretation > M = preferred_aset (R, preferred_prune ($\langle I \cup \{r\}, P\rangle$));
  return M $\cup$ preferred_aset(R, preferred_prune($\langle I \cup \{\neg r\}, P \cup M\rangle$));
  }
}
```

Fig. 6. The *preferred_aset* function for ordered programs.

Table 1. Preliminary performance tests.

input	olpg	OLPS	lparse	SMODELS	DLV
circuit	0m02.536s	0m00.154s	NA	NA	NA
ham-50	0m00.176s	0m00.060s	0m00.072s	0m00.084s	0m00.084s
ham-d50	0m04.465s	0m01.537s	0m00.118s	0m00.670s	0m06.613s
ham-60	0m00.245s	0m00.081s	0m00.086s	0m00.135s	34m14.371s
ham-d60	0m07.807s	0m03.553s	0m00.202s	0m02.103s	0m23.051s
ham-70	0m00.368s	0m00.124s	0m00.109s	0m00.216s	0m00.815s
ham-d70	0m12.882s	0m11.030s	0m00.265s	0m04.513s	0m57.121s
ham-80	0m00.533s	0m00.162s	0m00.145s	0m00.313s	0m00.276s
ham-d80	0m20.078s	0m35.501s	0m00.418s	0m11.050s	1m55.984s
ham-90	0m00.788s	0m00.248s	0m00.175s	0m00.468s	0m00.512s
ham-d90	0m29.781s	1m36.511s	0m00.429s	0m17.944s	3m57.191s
ham-100	0m01.326s	0m00.395s	0m00.228s	0m00.764s	0m01.164s
ham-d100	2m20.249s	54m04.504s	0m00.881s	2m30.887s	47m08.002s
ham-200	0m01.190s	0m00.459s	0m00.684s	0m02.937s	570m12.123s

grounder program *olpg* should be improved considerably: it performs much worse than *lparse*[11]. On the other hand, on the sparser graphs, *olps* performs similarly or slightly better than *smodels*, while on the dense graphs OLPS performs worse. The reason for the latter is subject to further research. For *dlv* only total (grounding and solving) figures are shown. Clearly, these tests are anecdotal and only a wider comparison on a range of applications can lead to firm conclusions. Nevertheless, we believe that the prelim-

[11] The fact that *olpg* outputs source code while *lparse* uses an efficient binary format does not help.

inary results are encouraging. One could argue that a similar approach can be used to compare OLPS with DLV on Σ_2^P-complete problems, however, at the moment there is no known transformation between ordered programs and disjunctive programs in either direction, as is also the case with e.g. logic programming with ordered disjunction[4].

Future versions should investigate the use of heuristics[10]. Currently, *select_min* (Figure 6), simply picks a minimal "open" rule that has a minimal number of undecided body literals. The use of more sophisticated heuristics by *select_min* and the detection and exploitation of certain special cases in other parts of the system could improve performance considerably. Finally, adding support for negation as failure (directly or through the construction used in [16]) would make it easy to add new front-ends for e.g. LPOD[4].

References

1. M. Arenas, L. Bertossi, and J.Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Procs. of the 4th International Conference on Flexible Query Answering Systems*, pages 27–41. Springer-Verlag, 2000.
2. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
3. N. D. Belnap. A useful four-valued logic. In *Modern uses of multi-valued logic*, pages 8–37. D. Reidel Publ. Co., 1975.
4. G. Brewka. Logic programming with ordered disjunction. In *Proc. of the National Conference on Artificial Intelligence*, pages 100–105. AAAI Press, 2002.
5. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.
6. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In *Procs. of the International Conference on Computational Logic (CL2000)*, volume 1861 of *LNCS*, pages 807–821. Springer, 2000.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. The DLV^k planning system. In *Logic in Artificial Intelligence*, volume 2424 of *LNAI*, pages 541–544. Springer Verlag, 2002.
8. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In *Logic in Artificial Intelligence*, volume 1919 of *LNAI*, pages 2–20. Springer Verlag, 2000.
9. W. Faber, N. Leone, and G. Pfeifer. Pushing goal derivation in DLP computations. In *Logic Programming and Non-Monotonic Reasoning*, volume 1730 of *LNAI*, pages 177–191. Springer Verslag, 1999.
10. W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for answer set programming. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 635–640. Morgan Kaufmann, 2001.
11. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of logic programming*, 4:295–312, 1985.
12. P. Flach. *Simply Logical - Intelligent Reasoning by Example*. Wiley, 1994.
13. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Procs. of the Intl. Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.
14. V. Lifschitz. Answer set programming and plan generation. *Journal of Artificial Intelligence*, 138(1-2):39–54, 2002.
15. D. Van Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. In *Logic in Artificial Intelligence*, volume 2424 of *LNAI*, pages 432–443. Springer Verlag, 2002.

16. D. Van Nieuwenborgh and D. Vermeir. Order and negation as failure. In *Procs. of the Intl. Conference on Logic Programming*, volume 2916 of *LNCS*, pages 194–208. Springer Verlag, 2003.
17. D. Van Nieuwenborgh and D. Vermeir. Ordered diagnosis. In *Procs. of the Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850 of *LNAI*, pages 244–258. Springer Verlag, 2003.
18. Davy Van Nieuwenborgh and Dirk Vermeir. Preferred answer sets for ordered logic programs. *Theory and Practice of Logic Programming (TPLP)*, page Accepted for publication, 2004.
19. T. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, 13:445–463, 1990.
20. T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Procs. of the Intl. Workshop on Practical Aspects of Declarative Languages*, volume 1551 of *LNCS*, pages 305–319. Springer Verslag, 1999.
21. T. Syrjänen and I. Niemelä. The smodels system. In *Procs. of the Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, volume 2173 of *LNCS*, pages 434–438. Springer-Verlag, 2001.
22. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733–742, 1976.
23. Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, 1991.
24. Davy Van Nieuwenborgh and Dirk Vermeir. Ordered programs as abductive systems. In *Proceedings of the APPIA-GULP-PRODE Conference on Declarative Programming (AGP2003)*, pages 374–385, Regio di Calabria, Italy, 2003.
25. M. De Vos and D. Vermeir. Choice Logic Programs and Nash Equilibria in Strategic Games. In *Computer Science Logic*, volume 1683 of *LNCS*, pages 266–276. Springer Verslag, 1999.