Satisfiability checking for PC(ID) ⋆

Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{maartenm,mitra,marcd,maurice}@cs.kuleuven.be

Abstract. The logic FO(ID) extends classical first order logic with inductive definitions. This paper studies the satisifiability problem for PC(ID), its propositional fragment. We develop a framework for model generation in this logic, present an algorithm and prove its correctness. As FO(ID) is an integration of classical logic and logic programming, our algorithm integrates techniques from SAT and ASP. We report on a prototype system, called MidL, experimentally validating our approach.

1 Introduction

The logic FO(ID), or Inductive Definition Logic (ID-logic) [7], extends classical first order logic (FO) with a language primitive that allows a uniform representation of inductive definitions. In general, inductive definitions cannot be represented in first order logic (FO). The semantics of this primitive is based on the well-founded semantics of logic programming [28]; indeed, as argued in [6, 8], it correctly formalizes the semantics of inductive definitions.

While definitions are common in mathematics, they are also crucial in declarative Knowledge Representation. Not only non-inductive definitions are frequent in common-sense reasoning as argued in the seminal paper [2], also inductive definitions are. In [10], the *situation calculus* is given a very natural and general representation as an iterated inductive definition in the well-ordered set of situations and [16] observes that inductive definitions are present in many applications of Answer Set Programming (ASP) [12]. In short, definitions are a distinctive and important form of knowledge that can be naturally represented in FO(ID).

The goal of this paper is to present algorithms to solve SAT(PC(ID)), the satisfiability problem for PC(ID)[1], the propositional fragment of FO(ID), or equivalently, generate models for theories in this fragment. This problem is an extension of SAT, the satisfiability problem of propositional CNF formulas. Current SAT solvers exhibit impressive performance on many industrial instances. Unfortunately, SAT is a rather poor modeling language, and substantial effort is often required to encode a problem. PC(ID) is a major enhancement of the expressivity [19]. Solvers for SAT(PC(ID)) are also strongly related to ASP solvers such as Smodels [21] and DLV [4]. These solvers use the fixpoint operator of the well-founded semantics as a boolean propagation mechanism for rule sets.

⋆ Works supported by FWO-Vlaanderen, IWT-Vlaanderen, European Framework 5 Project WASP, and by GOA/2003/08.

[1] PC(ID) stands for "Propositional Calculus, extended with Inductive Definitions".

Viable approaches for building a solver for SAT(PC(ID)) are:

1. A native approach that integrates inference techniques for inductive definitions with SAT inference techniques.
2. Mapping PC(ID) theories to CNF theories and applying off-the-shelf SAT solvers as in [23].
3. Mapping PC(ID) theories to equivalent general logic programs and then applying off-the-shelf ASP solvers. Using results from [16], it is easy to accomplish, but it completely bypasses PC(ID)'s relation with SAT.

The first approach, explored in this paper, is the most promising in the long run. It will improve our understanding of how to compute with inductive definitions and is best suited to integrate language extensions such as aggregates, constraints and open functions.

Despite the semantical differences, the computational tasks of our algorithm are very similar to those of algorithms like Smodels' Expand [21] and DLV's DetCons [4]. Its novelty lies in the use of justification semantics [9], which offers a different view on the computational task involved:

– One can locally test whether the well-founded operator is required.
– One can use a watched literal technique for propagations, very similar to the Two Watched Literal technique used in SAT [20].

We introduce PC(ID) in Section 2 and the semantic foundations of our algorithm in Section 3. We present the algorithm and argue its correctness in Section 4. Its implementation, MidL, and a comparison with other approaches are described in Section 5. We finish with conclusions and related work.

2 Preliminaries

2.1 PC(ID)

The semantics of FO(ID) ([7]) is here redefined for the propositional fragment PC(ID). A vocabulary Σ is a set of atom symbols. A definition D is a set of rules of the form $(P \leftarrow \varphi)$, where $P \in \Sigma$ is called the head of the rule, and the body of the rule, φ, is a propositional formula in Σ. We denote by $Def(D)$ the set of atoms that appear in the heads of the rules of D. The set $\Sigma \setminus Def(D)$ is called the set of open symbols of D and is denoted by $Open(D)$. A literal is an atom P or its negation $\neg P$; it is *defined* if $P \in Def(D)$, otherwise it is *open*. A PC(ID) theory T in Σ is a set of propositional formulas and definitions in Σ.

A three-valued Σ-interpretation I is a function from Σ to the set $\{\boldsymbol{f}, \boldsymbol{u}, \boldsymbol{t}\}$ of truth values; a two-valued interpretation maps to $\{\boldsymbol{f}, \boldsymbol{t}\}$ instead. Truth values are ordered by the truth order, defined as $\boldsymbol{f} < \boldsymbol{u} < \boldsymbol{t}$, and the precision order, given by $\boldsymbol{u} <_p \boldsymbol{t}$ and $\boldsymbol{u} <_p \boldsymbol{f}$. We also have $\boldsymbol{f}^{-1} = \boldsymbol{t}$, $\boldsymbol{u}^{-1} = \boldsymbol{u}$ and $\boldsymbol{t}^{-1} = \boldsymbol{f}$. We denote the projection of I on the atoms in a set $\mathcal{S}$ by $I|_{\mathcal{S}}$.

The semantics can be defined by means of the well-founded semantics [28]: Given a definition D and an interpretation $I|_{Open(D)}$ of the open atoms of D, there is a unique well-founded model of D extending $I|_{Open(D)}$, which we denote by $\mathrm{wfm}_D(I|_{Open(D)})$. An interpretation I is a model of definition D, denoted $I \models D$, iff I is two-valued and $I = \mathrm{wfm}_D(I|_{Open(D)})$; I is a model of a PC(ID) theory T iff I is two-valued and is a model of every definition and every propositional formula in T. An equivalent characterisation is provided by Corollary 1 below.

Example 1. Consider the definition $D_1 = \{P \leftarrow Q\}$. Then $P \in Def(D_1), Q \in Open(D_1)$. The models of D_1 are $\{P, Q\}$ and $\emptyset$. They are also the models of $D_2 = \{Q \leftarrow P\}$. Hence, $T_1 = \{\{P \leftarrow Q\}, \{Q \leftarrow P\}\}$ which consists of two definitions has models $\{P, Q\}$ and $\emptyset$. However, $T_2 = \{\{P \leftarrow Q, Q \leftarrow P\}\}$ consists of a single definition and has only $\emptyset$ as model.

The theory $T_3 = \{\{P \leftarrow Q\}, \{P \leftarrow\}, \neg Q \vee R\}$ has two definitions for P and one propositional formula; the first definition has two models (as D_1), while the second one has only $\{P\}$ as its model and $\{P, Q, R\}$ is the only model of T_3.

2.2 MidL normal form

Definition 1 (MidL normal form). *A PC(ID) theory T is in* MidL normal form *iff $T = C \cup \{D\} \cup E$ where C is a set of clauses without defined literals, E is a set of equivalences $P \equiv Q$ where P is an open and Q a defined atom, and D is a definition (a set of rules), with for each atom Q in $Def(D)$ exactly one rule with Q in the head. Moreover for all rules R in D, R is in the form $Q \leftarrow L_1 \wedge \ldots \wedge L_n$ or $Q \leftarrow L_1 \vee \ldots \vee L_n$, where $n \geq 1$ and L_i are literals.*

Example 2. $T_3^{MidL} = \{\{P^{D_A} \leftarrow Q, P^{D_B} \leftarrow\}, \neg Q \vee R, P \equiv P^{D_A}, P \equiv P^{D_B}\}$ is the MidL normal form equivalent to T_3 in Example 1: the models of T_3^{MidL}, restricted to T_3's vocabulary, are the models of T_3.

Defined atoms whose body is a disjunction respectively conjunction are called *disjunctive* respectively *conjunctive* atoms. Let $\mathcal{S}$ be a set of literals; then we abbreviate $\bigwedge_{L \in \mathcal{S}} L$ by $\bigwedge \mathcal{S}$ and $\bigvee_{L \in \mathcal{S}} L$ by $\bigvee \mathcal{S}$.

A straightforward transformation (time linear in the size of the input) that maps a PC(ID) theory T to an equivalent theory in MidL normal form by introducing new atoms, is given in [17]. As the above example shows, the head of a definition $P \leftarrow \ldots$ becomes a defined atom P^D which is linked to the original atom P through an equivalence.

3 Semantic Background

This section introduces semantical concepts borrowed from [9].

Definition 2 (Direct justification). *Let D be a definition in MidL normal form and J_d a set of literals. J_d is a* direct justification *for a defined atom P iff:*

- *either $P \leftarrow L_1 \wedge \ldots \wedge L_n \in D$ and $J_d = \{L_1, \ldots, L_n\}$;*
- *or $P \leftarrow L_1 \vee \ldots \vee L_n \in D$ and $J_d = \{L_i\}$ for some $i \in 1 \ldots n$.*

J_d is a direct justification *for a defined literal $\neg P$ iff:*
- *either $P \leftarrow L_1 \wedge \ldots \wedge L_n \in D$ and $J_d = \{\neg L_i\}$ for some $i \in 1 \ldots n$;*
- *or $P \leftarrow L_1 \vee \ldots \vee L_n \in D$ and $J_d = \{\neg L_1, \ldots, \neg L_n\}$.*

Example 3. Consider $D_1 = \{P \leftarrow Q \wedge \neg R, Q \leftarrow \neg P \vee R\}$. The set $\{Q, \neg R\}$ is a direct justification for P and $\{P, \neg R\}$ for $\neg Q$; both $\{\neg P\}$ and $\{R\}$ are direct justifications for Q and both $\{\neg Q\}$ and $\{R\}$ for $\neg P$.

Consider $D_2 = \{P \leftarrow Q, Q \leftarrow P \vee R, R \leftarrow \neg P \wedge S\}$. Though $\{Q\}$ is a direct justification for P and $\{P\}$ for Q, the truth of P doesn't justify the truth of Q or vice versa, as illustrated by the model $\mathrm{wfm}_D(\{S^{\boldsymbol{f}}\}) = \{P^{\boldsymbol{f}}, Q^{\boldsymbol{f}}, R^{\boldsymbol{f}}, S^{\boldsymbol{f}}\}$.

A direct justification is insufficient to infer the truth of a literal in the well-founded model. We need to consider graphs of direct justifications. A *leaf* of a graph is a node without outgoing edges.

Definition 3 (Justification). *A* justification *J of a definition in* M<span style="font-variant:small-caps">id</span>*L normal form is a directed graph where the nodes are literals, such that for each internal node L, L is a defined literal and the set of its children, $Ch_J(L)$, is a direct justification for L. A justification is* total *if none of its leaves are defined literals.*

Example 4. D_2 from Example 3 has many justifications. Examples are:

$$J_0 = \emptyset,\ J_1 = \left[\begin{tikzcd} P \arrow[d, bend left] \\ Q \arrow[u, bend left] \end{tikzcd}\right],\ J_2 = \left[\begin{tikzcd} P \arrow[r] & Q \arrow[r] & R \arrow[r] \arrow[d] & S \\ \neg R \arrow[u] & \neg Q \arrow[l] \arrow[r, bend left] & \neg P \arrow[l, bend left] & \end{tikzcd}\right],\ J_3 = \left[\begin{tikzcd} \neg P \arrow[d, bend left] & \\ \neg Q \arrow[u, bend left] \arrow[r] & \neg R \end{tikzcd}\right].$$

Here J_0, J_1 and J_2 are total, but J_3 is not since $\neg R$ is a defined leaf.

As the example shows, justifications can contain cycles. We distinguish between *positive. negative* and *mixed* cycles. They consist of repectively only positive (as in J_1), only negative (as in J_3) and both kind of literals (as in J_2).

A justification is an argument for the truth value of its literals. Its *value* depends on the structure of the justification and the truth value of its leaves.

Definition 4 ($\mathcal{V}_I(J)$, the value of a justification). *Let J be a justification, and I an interpretation.*

- *$\mathcal{V}_I(J) = \boldsymbol{f}$ if J contains either a leaf L with $I(L) = \boldsymbol{f}$ or a positive cycle.*
- *$\mathcal{V}_I(J) = \boldsymbol{u}$ if $\mathcal{V}_I(J) \neq \boldsymbol{f}$ and J contains either a leaf L with $I(L) = \boldsymbol{u}$ or a mixed cycle (or both).*
- *$\mathcal{V}_I(J) = \boldsymbol{t}$ otherwise (all leaves are $\boldsymbol{t}$ and cycles, if any, are negative).*

Example 5. Continuing Example 4. Independent of I, $\mathcal{V}_I(J_0) = \boldsymbol{t}$ and, because J_1 has a positive cycle, $\mathcal{V}_I(J_1) = \boldsymbol{f}$. Now, let $I = \{P^{\boldsymbol{t}}, Q^{\boldsymbol{t}}, R^{\boldsymbol{f}}, S^{\boldsymbol{t}}\}$. $\mathcal{V}_I(J_2) = \boldsymbol{u}$, indeed, although the only leaf (S) is $\boldsymbol{t}$, J_2 contains a mixed cycle; and finally $\mathcal{V}_I(J_3) = \boldsymbol{t}$, because the only leaf is $\boldsymbol{t}$ and the only cycle is negative.

The value of a total justification J depends only on its cycles and the interpretation of its open symbols. J_L, the *restriction of J to L*, denotes the subgraph with L as root. A defined literal L is *justified* in a justification J if J_L

is non-empty and total. The *supported value* $SV_I(L)$ of a defined literal L in an interpretation I is the maximal value in the truth order of its justifications i.e., $SV_I(L) = \max_{\leq}\{\mathcal{V}_I(J) | J$ is a justification and L is justified in $J\}$.

It was proven in [9] that in the well-founded model M, the interpretation of defined literals agrees with the supported values, i.e., $M(L) = SV_M(L)$. Furthermore, [5] proved that $SV_M(L) = SV_M(\neg L)^{-1}$ for any defined literal L.

Example 6. Continuing from Example 4, let I' be an interpretation with $I'(S) = \boldsymbol{f}$. Then $SV_{I'}(P) = \boldsymbol{f}$, since for any justification J in which P is justified, either S is a leaf in J_P, or $J_P = J_1$. And indeed, let J' be the total justification obtained from J_3 by adding the edge $(\neg R, \neg S)$; then $SV_{I'}(\neg P) \geq \mathcal{V}_{I'}(J'_{\neg P}) = \mathcal{V}_{I'}(J') = \boldsymbol{t}$.

Hence one can compute the well-founded model by computing for each literal its supported value. However, searching over all justifications for the best one is infeasible. Fortunately, attention can be restricted to a subclass of justifications.

Definition 5 ((strict) support, positive residue). *Let J be a justification and I an interpretation. J* supports *I if $I(L) \leq_p I(\bigwedge Ch_J(L))$ and J* strictly supports *I if $I(L) = I(\bigwedge Ch_J(L))$ for each internal node L of J.*

The positive residue *of J in I consists of the atoms that would be strictly supported if true but are undefined, i.e., $I(L) = \boldsymbol{u}$ and $I(\bigwedge Ch_J(L)) = \boldsymbol{t}$.*

Definition 6 (Cycle-safe). *Let J be a justification and I an interpretation. J is* cycle-safe *in I if J contains neither mixed cycles with literals that are true in I nor positive cycles.*

Definition 7 ($\boldsymbol{v}$-total). *Let J be a justification, I an interpretation, and $\boldsymbol{v}$ a truth value, i.e., $\boldsymbol{v} \in \{\boldsymbol{f}, \boldsymbol{u}, \boldsymbol{t}\}$. Then J is* $\boldsymbol{v}$-total *in I if for each defined literal L such that $I(L) = \boldsymbol{v}$, L is justified in J.*

Note that if a justification J is $\boldsymbol{f}$-, $\boldsymbol{u}$- and $\boldsymbol{t}$-total w.r.t. an interpretation I, then J is total, but a total justification might not be $\boldsymbol{f}$-, $\boldsymbol{u}$- or $\boldsymbol{t}$-total w.r.t. any interpretation, because not every literal occurs in J.

Theorem 1. *Let I be an interpretation and J a justification. If the following conditions hold:*

(i1) J strictly supports I;
(i2) J is cycle-safe in I;
(i3) J is $\boldsymbol{t}$-total in I;
(i4) J is $\boldsymbol{u}$-total in I,

then for every defined literal L it holds that $I(L) = SV_I(L)$.

Corollary 1. *Let I and J satisfy (i1)-(i4). Then* $\text{wfm}_D(I|_{Open(D)}) = I$.

In the next section we introduce an algorithm that incrementally constructs and maintains a 3-valued interpretation I and a justification J that satisfy the conditions (i1)-(i4).

```
1  Initialize $I$ and $J$ (see Section 4.4);
2  while there is an open atom $A$ with $I(A) = \boldsymbol{u}$ do
3      Select open atom $A$ with $I(A) = \boldsymbol{u}$;
4      Choose $\mathcal{S} := \{A\}$ or $\mathcal{S} := \{\neg A\}$;
       % Boolean Propagation:
5      while true do
6          if $\mathcal{S} \neq \emptyset$ then
7              Direct_Propagation($I, J, \mathcal{S}$);  % Can initiate backtracking
8          if there is an $L$ with $I(L) = \boldsymbol{u}$ and $Ch_J(L) = \emptyset$ then  select such a $L$;
9          else  Break;
10         $\mathcal{S}$ := Justify($I, J, L$);
11 if $I$ is 2-valued then  Return SATISFIABLE;
12 else  Backtrack;
```

Algorithm 1: MIDL(T)

4 Algorithm

4.1 Structure of the algorithm

The progenitor of our algorithm is the DLL algorithm [3], which is also the basis of most of today's SAT solvers. The DLL algorithm incrementally constructs an interpretation I that satisfies an input CNF theory T. In each stage, a choice literal is selected and is made true, after which *boolean propagation* is applied on the current assignment to infer other true literals. In particular, *unit propagation* makes the last non-false literal of a clause true. In case of conflict, backtracking returns to the last choice point. There, the alternative choice is made.[2]

Algorithm 1, the backbone of our algorithm is similar to the DLL algorithm. The input is a PC(ID) theory in MIDL normal form. However, the choice literal is an open one and boolean propagation is extended to include propagations according to the well-founded semantics by maintaining a 3-valued assignment I and a justification J. The aim of Steps 1 and 5 is to establish the conditions (i1)-(i4), as well as the following invariant:

(i5) For each clause $\mathcal{C} \in T$, $I(\mathcal{C}) \geq \boldsymbol{u}$;

If so, obviously, I is a model when Step 11 returns SATISFIABLE. The "Boolean Propagation" (Step 5) is the core of the algorithm; it consists of two components:

Direct Propagation The input consists of I and J, the current interpretation and justification and the set of literals $\mathcal{S}$ to be made true. The latter are made true, unit propagation on the clauses and propagation from body to head in the rules of the definitions are applied. This establishes the invariants (i1)-(i3) and (i5) unless a conflict is detected and backtracking is initiated.

[2] UNSATISFIABLE is returned when no more backtrackings are possible.

Justify The input consists of the current I and J, and a literal L that violates invariant (i4) ($\boldsymbol{u}$-totality) (L is a unknown defined literal that has no justification). The procedure tries to adjust J into a justification J' in which all literals of J'_L are strictly supported. In case this fails, an *unfounded set* [28] is found. The involved atoms must become false, their negation is returned in the set $\mathcal{S}$ for processing by the direct propagation.

The $\boldsymbol{u}$-totality invariant is satisfied when the while loop exits in Step 9. It terminates because Direct_Propagation extends the interpretation I and Justify either reduces the number of unknown defined literals without justification or finds a unfounded set that leads to an extension of I in the next iteration. Hence:

Theorem 2. *Steps 5-10 of Algorithm 1 terminate; when they do via Step 9 then invariants (i1)-(i5) are satisfied.*

4.2 Direct Propagation (Algorithm 2)

The Direct_Propagation algorithm uses the following data structures:

- A data structure *dj* which associates with each defined literal L a direct justification of L. While $dj(L)$ is constant for conjunctive atoms and the negation of disjunctive atom, it consists of a selected literal (the *watched* literal) in the other cases.
- A boolean data structure *just* over defined literals, indicating whether or not the literal's direct justification belongs to the current justification[3].

The data structures *dj* and *just* together determine a justification J in the following way: if $just(L)$, then $Ch_J(L) = dj(L)$; otherwise, $Ch_J(L) = \emptyset$.

Making a literal in $\mathcal{S}$ true (Step 5) invalidates invariant (i1); however, the following weaker invariants are maintained:

(i6) J supports I;
(i7) the positive residue of J in I is a subset of $\mathcal{S}$; for all defined literals $L \in \mathcal{S}$: $\mathcal{V}_{J_L}(I) = \boldsymbol{t}$; i.e., there exists a justification for making L true.

From invariant (i7), it follows that there is no positive residue when $\mathcal{S}$ becomes empty, hence (i1) is restored. Also (i2) and (i3) are preserved:

$\boldsymbol{t}$-totality (i3): A defined literal can only be made true in step Step 5 of Algorithm 2.[4] Therefore, if a defined literal L has $I(L) = \boldsymbol{t}$, we know by (i6) that it has children in J that are all also true. By induction over all literals in J_L, J_L must be total and invariant (i3) holds.

[3] Hence L is justified if $just(L)$ and the literals in $Ch_J(L)$ are justified

[4] This explains both the restriction in Step 3 of Algorithm 1 to *open* literals, and the requirement in Definition 1 that the CNF formula cannot contain defined literals.

```
    Assert: (i1), (i2), (i3) and (i5)
 1  repeat
 2      Pop $L$ from $\mathcal{S}$;
 3      if $I(L) = \boldsymbol{f}$ then backtrack (to choice point in Alg. 1);
 4      if $I(L) \neq \boldsymbol{t}$ then
 5          $I(L) := \boldsymbol{t}$;
 6          if $L$ is defined then Push its original on $\mathcal{S}$ (using the equivalences);
 7          if $L$ is open then for every clause $c$ of $T$ containing $\neg L$ do
 8              if $c$ has exactly one non-false literal $L'$ then Push $L'$ on $\mathcal{S}$;
 9          for every unknown literal $L'$ for which $\{L\}$ is a direct justification do
10              $just(L') := \boldsymbol{t}$; $dj(L') := \{L\}$; Push $L'$ on $\mathcal{S}$ ;
11          for every unknown literal $L'$ such that $\neg L \in dj(L')$ do
12              case $L' = P$ with $P \leftarrow \neg L \wedge \cdots \in T$, or $L' = \neg P$ with
                $P \leftarrow L \vee \cdots \in T$
13                  $dj(\neg L') := \{L\}$; $just(\neg L) := \boldsymbol{t}$; Push $\neg L'$ on $\mathcal{S}$;
14              case $L' = \neg P$ with $P \leftarrow L \wedge \cdots \in T$
15                  if $P$ has no unknown body literals left then Push $P$ on $\mathcal{S}$;
16                  else
17                      Select an unknown body literal $L''$ of $P$; $dj(L') := \{\neg L''\}$;
                        $just(L') := \boldsymbol{t}$ ;
18              case $L' = P$ with $P \leftarrow \neg L \vee \cdots \in T$
19                  if $P$ has no unknown body literals left then Push $\neg P$ on $\mathcal{S}$;
20                  else
21                      if $P$ has unknown open or negative body literal $L''$ then
22                          $dj(L') := \{L''\}$; $just(L') := \boldsymbol{t}$;
23                      else
24                          Select an unknown defined body atom $Q$;
25                          $dj(L') := \{Q\}$; $just(L') := \boldsymbol{f}$;
26  until $\mathcal{S} = \emptyset$;
```

Algorithm 2: Direct Propagation($I, J, \mathcal{S}$)

Cycle-safeness (i2): By (i3) and (i6), true literals are justified, hence taking a true literal as the direct justiciation of a unknown literal cannot create a mixed cycle containing true literals.We must also verify that none of the steps in Algorithm 2 creates a positive cycle in J. Steps 2-10 don't change J. In the case of Step 12 $Ch_J(L')$ is only changed for a *negative* literal L', in the case of Step 14 J is either not changed, or only for a negative literal. Finally, in the case of Step 18 a positive cycle might be created in J when the direct justification of the atom P is changed to a positive defined atom. This is prevented by removing it from J ($just(L') := \boldsymbol{f}$ implies $Ch_J(L') := \emptyset$). Consequently, $\boldsymbol{u}$-totality, invariant (i4), is violated.

Making an unknown literal true can violate invariant (i5) on the clauses. Step 8 not only performs unit propagation (not all clauses containing $\neg L$ are inspected but the Two Watched Literal technique is used) but also restoration of the invariant. By pushing the last non-false literal on the stack, a violation of the invariant will finally result in the uncovering of a conflict in Step 3.

Example 7. Let $T = \{\{P^D \leftarrow Q^D \vee A,\ Q^D \leftarrow P^D \wedge R^D \wedge \neg B,\ R^D \leftarrow C\}, P^D \equiv P,\ Q^D \equiv Q,\ R^D \equiv R\}$, $I = \emptyset$, and

$$J = \left[\begin{tikzcd} P^D \arrow[d] & Q^D \arrow[l] \arrow[r] \arrow[d] & R^D \arrow[d] & \neg P^D \arrow[d] \arrow[r, bend left] & \neg Q^D \arrow[l, bend left] & \neg R \arrow[d] \\ A & \neg B & C & \neg A & & \neg C \end{tikzcd} \right]$$

. Suppose Step 3 in Algorithm 1 chooses $\mathcal{S} := \{B\}$. Then Step 5 makes B true; subsequently Step 13 changes $Ch_J(\neg Q^D)$ from $\neg P^D$ to B, and pushes $\neg Q^D$ on $\mathcal{S}$. In a next iteration of the repeat loop, $\neg Q^D$ is made true, and $\neg Q$ is pushed on $\mathcal{S}$ in Step 6.

Suppose that orginally $\mathcal{S} := \{\neg A\}$ is chosen. After making A false, the algorithm ends up in Step 23. $dj(P^D)$ is changed from A to Q^D. In addition $just(P^D)$ is set to $\boldsymbol{f}$. Note that otherwise a positive cycle $P^D \leftrightarrow Q^D$ would have been created.

The Direct_Propagation algorithm can be understood as performing a watched literal technique for rules, similar to the Two Watched Literals (2WL) technique in SAT [20]. Every literal in a singleton direct justification (i.e., of a positive disjunctive atom, or of a negative conjunctive atom) is "watched": when the literal becomes false, the corresponding rule has to be visited. When any other literal in that body becomes false, we don't need to visit the rule yet, since the "watched literal" is still unknown and thus might still justify the head.

Interpreting the head of each rule as a second watched literal, it can be seen that the Direct_Propagation algorithm actually has the same behaviour as the 2WL scheme has on the completion of the definition.

4.3 Justify (Algorithm 3)

Algorithm 3 tries to adjust J so that J_P is a total justification for the unknown atom P. If $dj(P)$, the direct justification of P, can be added to the current justification J (by setting $Just(P)$ to $\boldsymbol{t}$) without creating a positive cycle (involving P) then we are done. This is tested in Step 1. This is a fairly simple test; however, if it fails, we will have to adjust the direct justification of P to construct a total justification.

The next step then is to find an overestimation of all unknown defined atoms that can potentially contribute to some J_P that is a total justification of P. Note that the unknown open literals and unknown negative literals can be used as valid leaves in a total justification, so they need not be included. This overestimation is computed in Step 2 and stored in $\mathcal{E}$. It consists of all atoms *reachable from* P in the dynamic positive dependency graph. The latter is defined as follows:

Definition 8 (dynamic positive dependency graph). *Let D be a definition in* MIDL *normal form, I a partial interpretation. The* dynamic positive dependency graph *of D in I is a directed graph (V, E) of atoms. $P \in V$ iff P is defined*

```
   **Assert**: (i1), (i2), (i3) and $Ch_J(P) = \emptyset$
   **Result**: a set of negative literals $\mathcal{S}$;
1  if $IsCycleSafe(P, dj)$ then $just(P) := \boldsymbol{t}$; Return $\emptyset$;
2  $\mathcal{E}$ := FindSet($I, P$);
3  $\mathcal{B}$ := FindBottomSeeds($I, \mathcal{E}$); $\mathcal{E} := \mathcal{E} \setminus \mathcal{B}$;
4  while $\mathcal{B} \neq \emptyset$ do
5      Select $Q$ from $\mathcal{B}$; $\mathcal{B} := \mathcal{B} \setminus \{Q\}$;
6      for disjunctive atoms $Q' \in \mathcal{E}$ such that $Q$ occurs in the body of $Q'$ do
7          $dj(Q') := \{Q\}$; $just(Q') := \boldsymbol{t}$; Move $Q'$ from $\mathcal{E}$ to $\mathcal{B}$;
8      for conjunctive atoms $Q' \in \mathcal{E}$ such that $dj(\neg Q') = \{\neg Q\}$ do
9          if there exists a $Q'' \in \mathcal{E}$ in the rule of $Q'$ then $dj(\neg Q') := \{\neg Q''\}$;
10         else $just(Q') := \boldsymbol{t}$; Move $Q'$ from $\mathcal{E}$ to $\mathcal{B}$;
11 Return $\{\neg Q | Q \in \mathcal{E}\}$;
```
Algorithm 3: Justify(I, J, P)

in D and $I(P) = \boldsymbol{u}$. $(P, Q) \in E$ iff $\{P, Q\} \subseteq V$ and P is the head of a rule and the atom Q occurs in its body.

Observe that $\mathcal{E}$ not only includes atoms without direct justification such as P itself, but also atoms with direct justification. In order to justify P, however, it may be necessary to update the direct justification of such atoms.In other words, $\mathcal{E}$ is the set of *endangered atoms*: atoms, whose direct justification may have to be revised.

In the following steps, the algorithm tries to construct justifications for elements in $\mathcal{E}$. In Step 3, it collects in $\mathcal{B}$ elements from $\mathcal{E}$ that have a trivial justification:

- conjunctive atoms $C \in \mathcal{E}$ which have no elements from $\mathcal{E}$ in their body (the unknown literals in their body are either open or negative)
- disjunctive atoms $D \in \mathcal{E}$ which have an unknown open or negative literal L in their body (their direct justification is set to that literal).

In the while loop, elements from $\mathcal{B}$ are used one by one as seeds to construct justifications for other elements in $\mathcal{E}$. In Step 6, the direct justification of a disjunctive atom is set to the seed and in Step 8, a conjunctive atom is justified and removed from $\mathcal{E}$ once it has no other body atoms in $\mathcal{E}$. In both steps, the newly justified atoms are added to the seeds. Observe that negative literals returned in Step 11 are justified; their direct justifications have been set in Step 9.

When exiting the while loop, all seeds have been used and what remains is an unfounded set of atoms. They have to be made false, hence their negation is returned by Justify.

This algorithm is an adaptation from [15] to the context of justifications.

Example 8. Continuing from Example 7, where we initially chose $\mathcal{S} := \{\neg A\}$. In Step 10 of Algorithm 1, Justify(I, J, Q^D) will be called. Since $dj(Q^D) = \{P^D\}$ and $dj(P^D) = \{Q^D\}$, $IsCycleSafe(Q^D, dj)$ fails. $\mathcal{E} = \{P^D, Q^D, R^D\}$ is computed

in Step 2. In the next step we find $\mathcal{B} = \{R^D\}$ and therefore $\mathcal{E} = \{P^D, Q^D\}$, meaning that R^D might still justify P^D and/or Q^D becoming true.
Since R^D only occurs in a conjunctive body (Q^D), but $dj(\neg Q^D) \neq \{\neg R^D\}$, R^D cannot justify anything, and the while-loop stops. Finally Justify returns $\{\neg P^D, \neg Q^D\}$ which are the negated literals from the unfounded set $\{P^D, Q^D\}$.

Let J be a justification, then we define $\mathcal{T}_J$ as the set $\{L | L \in J \wedge Ch_J(L) = \emptyset \wedge L \text{ is defined}\}$, i.e. the defined leaves of J.

Proposition 1. *Let I, J and P satisfy the requirements in Algorithm 3. Let J' be the justification after termination of Algorithm 3 and $\mathcal{S}$ the resulting set. Then either $\mathcal{S}$ is empty and $\mathcal{T}_{J'} \subseteq \mathcal{T}_J \setminus \{P\}$, or $\mathcal{S}$ is not empty and contains the negated literals from an unfounded set: for each $\neg Q \in \mathcal{S}$, $\neg Q$ is justified and $\mathcal{V}_{J_{\neg Q}}(I) = \boldsymbol{t}$.*

Proposition 2. *Algorithm 3 preserves invariants (i1), (i2), (i3) and (i5).*

These propositions formalize our claim in Section 4.1 that after each call to Justify either I is constant and the number of unknown defined leaves of J decreases, or I can be extended by the set returned by Justify.

4.4 Correctness, initialization and optimization issues

Soundness and Completeness. Soundness has been argued in Section 4.1: I is a model when Algorithm 1 returns SATISFIABLE. Completeness follows from the completeness of the DLL algorithm and the observation that our Boolean Propagation extends unit propagation of DLL and computes after each choice point the well-founded model extending the interpretation of the open literals.

Initialization. Step 1 of Algorithm 1 should initialize an interpretation I and a partial justification J such that invariants (i1)-(i5) are satisfied. To do this, we initialize I to the empty interpretation and $dj(L)$ to an arbitrary direct justification of L, for every defined literal. To avoid positive cycles, we set $just(P) := \boldsymbol{f}$ for every defined atom P. Then the set of unit clauses is collected in $\mathcal{S}$, and the initialisation executes the same while loop as described in Step 5 of Algorithm 1.

Backtracking. Backtracking restores I and J to some old value, say I_b and J_b. The invariants (i1)-(i5) hold for I_b and J_b. In fact, with exception of (i4), they also hold for I_b and J. Indeed, a conflict arises at a moment when (i2), (i3) and (i6) all hold. After restoring I to a previous level, for every defined literal L it holds that $I(L) = I(\bigwedge dj(L))$, and hence (i6) still holds, i.e., J supports I. Since (i1)-(i3), (i5) are sufficient pre-conditions for Boolean Propagation, the justification need not be restored upon backtracking.

Endangered Atoms. After an unsuccessful cycle-safety check (which is in fact optional), the first step of Justify is a call to FindSet to produce $\mathcal{E}$, the set of atoms endangered by the modified direct justification of P. There are a number of alternative possibilities. For example, the algorithm would remain correct if FindSet would return all unknown atoms in $Def(D)$ instead. However, one could also compute a smaller set of endangered atoms: Those in the strongly connected component (SCC) of P in the dynamic positive dependency graph. An SCC is a set $\mathcal{S}$ of atoms, such that every element of $\mathcal{S}$ can reach every other element of $\mathcal{S}$ via the directed edges. Tarjan's algorithm searches for strongly connected components in time linear in the size of the graph [26, 22]. Also the SCC of the static positive dependency graph, which we define as the dynamic positive dependency graph after the initialization (Step 1 in Algorithm 1), could be used. The smaller the set of endangered atoms, the faster it is processed. However, there is a trade-off between the work spent to compute the endangered atoms and that spent on processing them.

5 Experimental results

We have made a prototype implementation, called MidL, of the algorithm described in the previous section. We have compared MidL to Smodels [21] and idsat(zChaff) [23, 29]. We have thus a representative for each of the three approaches mentioned in Section 1. Note that the input to Smodels is handcoded and hence behaves possibly better than what one would obtain by an automatic mapping from PC(ID) to ASP. We used two classes of problems: N-queens and Hamiltonian cycles[5]. All the experiments were run on 863 MHz P-III with 254 MB of RAM. In Table 1 timings to find the first model (averaged over 5 runs) are given in seconds.

Since MidL doesn't enjoy the fine-tuned optimizations of a state of the art SAT or ASP solver, we don't expect comparable effiencies. Most notably, MidL's data structures are not yet minimized, which can be expected to lead to serious losses in cache usage. Also its heuristics are very crude.

Indeed, the results show poor scaling. However, observe that MidL outperforms the other solvers on some of the smaller problems. Remarkably, zChaff cannot cope with several Hamiltonian cycle problems; as they contain a lot of induction, idsat increased the size of these problems 20-fold. This confirms the usefulness of our native approach with respect to a translation to SAT. For an important class of problems, MidL is currently the best PC(ID) model generator.

In Table 2 we compare timings[6], the total number of atoms found to be in an unfounded set (U) and the total number of endangered atoms considered (E) for some variants of MidL: in MidL- and MidL SCC-, the optional Step 1 of

[5] We denote the Hamiltonian cycle problems by "H-*#vertices-#nodes*". Encodings are taken from `http://asparagus.cs.uni-potsdam.de/`; randomly generated graphs were used.

[6] To save space, we've selected but a few example problems; other problems exhibit the same behaviour.

	MidL	idsat+zCh	Smod.
9-queens	0.08	0.56+0.02	**0.05**
11-queens	1.49	0.79+0.21	**0.18**
13-queens	10.77	1.18+0.13	**0.43**
15-queens	296.91	**1.67+0.16**	1.85
H-20-200	0.32	61.19+7.62	**0.10**
H-25-200	**0.06**	96.5+82.2	0.10
H-30-200	1.97	98.5+137	**0.13**
H-35-200	3.65	#+#	**0.18**
H-20-400	**0.12**	72.87+#	0.35
H-25-400	**0.20**	128.0+#	0.55
H-30-400	**0.18**	209.2+#	0.56
H-35-400	#	#+#	**0.87**

Table 1. Timings (*sec*) of MidL, idsat using zChaff, and Smodels. # = >10min.

		MidL	MidL-	MidL SCC	MidL SCC-
H-20-200	U	1427	1461	1022	1433
	E	4570	20327	2535	11363
	time	**0.32**	0.34	3.52	1.57
H-30-200	U	1933	2013	474	1976
	E	10567	116186	2708	62460
	time	1.97	2.87	**1.20**	3.62
H-20-400	U	248	283	112	179
	E	1112	6573	785	4435
	time	**0.12**	0.17	0.25	0.24
H-30-400	U	209	231	85	180
	E	2002	31618	2031	22975
	time	**0.18**	1.70	0.32	1.77

Table 2. Comparison of unfounded set sizes (U) vs. number of "endangered atoms" (E) and timings (*sec*) for different variants of MidL

Algorithm 3 is disabled, in MidL SCC and MidL SCC- the FindSet procedure returns the SCC of the dynamic positive dependency graph, as described in Section 4.4. The first important observation is that the cycle-safety check almost consistently yields faster results. A second observation is that MidL SCC is doing more intelligent work: it uses less endangered atoms, and it also finds much less unfounded sets, i.e., finds falsity of atoms through Direct Propagation more often. Still, MidL outperforms MidL SCC, suggesting that a more careful implementation that removes some overhead of the SCC computation might be beneficial.

6 Conclusions, related and future work

This work is one of the first attempts to build a SAT(PC(ID)) system. We have chosen for a direct implementation, in contrast to a mapping to ASP, or to propositional logic, as was done in [23]. The latter approach is similar to ASP systems such as ASSAT [14] and Cmodels [13].

Despite semantical differences between PC(ID) and ASP, the algorithms presented here share a lot of structure with those of Smodels and DLV. The main novelties of our approach come through the use of justifications:

– this enables us to do the beneficial cycle-safety check;
– it integrates nicely with a watched literal technique for rules;
– the justification graph can be seen as a straightforward extension of the implication graph, which is used for Clause Learning [18]. In the near future, we intend to include this important SAT technique in MidL.

Another strongly related system is `dcs` [11]. This system can be viewed as a model generator for a fragment of FO(ID). The system takes as input a function free FO theory and an inductive definition consisting of Horn rules, and computes Herbrand models of this theory.

In future work, we plan to re-implement MidL, investigating a variety of techniques from SAT, logic programming and ASP. Interesting recent optimizations to SAT solvers are described in [20, 24, 18, 29]. Potentially relevant techniques for computing the well-founded semantics are described in [27, 1, 15, 25].

Finally, we mention some issues that will be investigated in the near future:

- The current solver uses rules only in a bottom up propagation. In some situations it is definitely worthwhile to also exploit proagation from head to body.
- As shown in [20, 24], the quality of the search algorithm strongly depends on the heuristics. Now that our search algorithm is more or less fixed, we should start to evaluate different heuristics for the system.
- A major task in the project is to build an efficient grounder which reduces a FO(ID) theory to a propositional theory by grounding it with respect to the Herbrand universe.

References

1. K.A. Berman, J.S. Schlipf, and J.V. Franco. Computing the well-founded semantics faster. In V. Marek and A. Nerode, editors, *LPNMR*, volume 928 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 1995.
2. R.J. Brachman and H.J. Levesque. Competence in knowledge representation. In *Proc. of the National Conference on Artificial Intelligence*, pages 189–192, 1982.
3. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
4. T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System description: DLV. In T. Eiter, W. Faber, and M. Truszczyński, editors, *LPNMR*, volume 2173 of *LNCS*, pages 424–428. Springer, 2001.
5. M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming.* PhD thesis, Department of Computer Science, K.U.Leuven, 1993.
6. M. Denecker. The well-founded semantics is the principle of inductive definition. In J. Dix, L. Fariñas del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence*, volume 1489 of *LNAI*, pages 1–16. Springer-Verlag, 1998.
7. M. Denecker. Extending classical logic with inductive definitions. In J. Lloyd et al., editor, *CL*, volume 1861 of *LNAI*, pages 703–717. Springer, 2000.
8. M. Denecker, M. Bruynooghe, and V. Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2(4):623–654, 2001.
9. M. Denecker and D. De Schreye. Justification semantics: a unifying framework for the semantics of logic programs. In *LPNMR*, pages 365–379. MIT Press, 1993.
10. M. Denecker and E. Ternovska. Inductive situation calculus. In D. Dubois, C.A. Welty, and M. Williams, editors, *KR*, pages 545–553. AAAI Press, 2004.
11. D. East and M. Truszczyński. `dcs`: An implementation of datalog with constraints. *CoRR*, cs.AI/0003061, 2000.
12. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–387, 1991.

13. Y. Lierler and M. Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In V. Lifschitz and I. Niemelä, editors, *LPNMR*, volume 2923 of *LNCS*, pages 346–350. Springer, 2004.
14. F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.
15. Z. Lonc and M. Truszczyński. On the problem of computing the well-founded semantics. In J.W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, and P.J. Stuckey, editors, *Computational Logic*, volume 1861 of *LNCS*, pages 673–687. Springer, 2000.
16. M. Mariën, D. Gilis, and M. Denecker. On the relation between ID-logic and answer set programming. In J.J. Alferes and J.A. Leite, editors, *JELIA*, volume 3229 of *LNCS*, pages 108–120. Springer, 2004.
17. M. Mariën, R. Mitra, M. Denecker, and M. Bruynooghe. Satisfiability checking for PC(ID). Technical Report CW426, K.U. Leuven, 2005.
18. J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
19. D.G. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In AAAI Press/MIT Press, editor, *Twentieth National Conf. on Artificial Intelligence (AAAI-05)*, 2005.
20. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.
21. I. Niemelä, P. Simons, and T. Syrjänen. Smodels: a system for answer set programming. In *NMR*, 2000.
22. E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Inf. Process. Lett.*, 49(1):9–14, 1994.
23. N. Pelov and E. Ternovska. Reducing inductive definitions to propositional satisfiability. In *ICLP*, LNCS, 2005.
24. L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
25. K.F. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In R.T. Snodgrass and M. Winslett, editors, *SIGMOD Conference*, pages 442–453. ACM Press, 1994.
26. R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
27. A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.
28. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
29. L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285. ACM, 2001.