Representing Normal Programs with Clauses

Tomi Janhunen

Abstract. We present a new method for transforming normal logic programs into sets of clauses. This transformation is based on a novel characterization of stable models in terms of level numberings and it uses atomic normal programs, which are free of positive body atoms, as an intermediary representation. The corresponding translation function possesses a unique combination of properties: (i) a bijective relationship is established between stable models and classical models, (ii) the models coincide up to the set of atoms \(\text{At}(P)\) appearing in a program \(P\), and (iii) the length of the translation as well as the translation time are of order \(\|P\| \times \log_2 |\text{At}(P)|\) where \(\|P\|\) is the length of the program \(P\). Our preliminary experiments with an implementation of the transformation, namely translators called L2ATOMIC and L2SAT, and SAT solvers such as CHAFF and RELSAT suggest that our approach is competitive when the task is to compute not just one but all stable models for a normal program given as input.

1 INTRODUCTION

Normal logic programs under the stable model semantics \([9]\) are well-suited for a variety of knowledge representation tasks. Typically, a programmer solves a problem at hand (i) by formulating it as a logic program whose stable models correspond to the solutions of the problem and (ii) by computing stable models using a special-purpose search engine. The reader is referred e.g. to \([18, 20]\) for examples of using this kind of methodology, also known as answer set programming (ASP).

Similar problems are solvable by formulating them as classical satisfiability (SAT) problems and using SAT solvers. However, such formulations tend to be more difficult and less concise. E.g., formulating an AI planning problem is much easier as a normal logic program \([6]\) than as a set of clauses \([12]\). This indicates of a real difference in expressive power which can be established formally by showing that normal programs cannot be translated into sets of clauses in a faithful and modular way \([20, 10, 11]\). In spite of these intractability results, we develop a faithful and non-modular, but still fairly systematic, translation from normal programs into sets of clauses. Using a novel characterization of stable models based on level numberings, the time complexity remains sub-quadratic.

The rest of this article is structured as follows. In Section 2, we review the syntax and semantics of normal logic programs as well as sets of clauses. As a further preparatory step, we characterize stable models in terms of level numberings in Section 3. The translation mentioned above is presented in Section 4. We report on our preliminary experiments in Section 5. The discussion in Section 6 concludes the paper.

2 PRELIMINARIES

In this paper, we restrict ourselves to the purely propositional case and consider only programs that consist of propositional atoms. A normal (logic) program \(P\) is a set of rules which are expressions of the form

\[
a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m
\]

where \(a\) is an atom, and \(\{b_1, \ldots, b_n\}\) and \(\{c_1, \ldots, c_m\}\) form sets of atoms. Here \(\sim\) denotes default negation or Clark’s negation as failure to prove, which differs from classical negation denoted by \(\neg\). Intuitively speaking, a rule \(r\) of the form (1) is used as follows: if the positive body atoms in \(B^+(r) = \{b_1, \ldots, b_n\}\) are inferable by the rules of the program, but not the negative body atoms in \(B^-(r) = \{c_1, \ldots, c_m\}\), then the head atom \(H(r) = a\) can be inferred by applying \(r\).

The positive part \(r^+\) of a rule \(r\) is defined as \(H(r) = B^+(r)\). A (normal) program \(P\) is positive, if \(r = r^+\) holds for all rules \(r \in P\). In addition to positive programs, we distinguish normal programs that are obtained by restricting the number of positive body atoms, i.e. \(B^+(r)\), allowed in a rule \(r\). A rule \(r\) of a normal program is called atomic, unary or binary, if \(|B^+(r)| = 0, |B^+(r)| \leq 1,\) or \(|B^+(r)| \leq 2\), respectively. We extend these conditions to cover a normal program \(P\) in the obvious way: \(P\) is atomic, unary, or binary if every rule of \(P\) satisfies the respective condition. E.g., an atomic normal program \(P\) contains only rules of the form \(a \leftarrow \neg c_1, \ldots, \neg c_m\).

Let us then turn our attention to the semantics of normal programs. We write \(\text{At}(P)\) for the set of atoms that appear in a program \(P\). An interpretation \(I \subseteq \text{At}(P)\) of \(P\) determines which atoms \(a \in \text{At}(P)\) are true (\(a \in I\)) and which atoms are false (\(a \not\in \text{At}(P)\) – \(I\)). A rule \(r\) is satisfied in \(I\), denoted by \(I \models r\), if \(I \models H(r)\) is implied by \(I \models B(r)\) where \(B(r) = \{b_1, \ldots, b_n\} \cup \{\neg c_1, \ldots, \neg c_m\}\) and \(\sim\) is interpreted classically, i.e. \(I \models \sim c_i\) if \(I \not\models c_i\). Then an interpretation \(I\) is a (classical) model of \(P\), denoted \(I \models P\), if \(I \models r\) for each \(r \in P\).

But the semantics of normal programs is not solely based on classical models. A model \(M \models P\) is a minimal model of \(P\) iff there is no model \(M' \models P\) such that \(M' \subset M\). In particular, every positive normal program \(P\) has a unique minimal model which equals to the intersection of all models of \(P\) \([16]\).

We let \(\text{LM}(P)\) stand for this particular model, i.e. the least model of \(P\). The least model semantics is inherently monotonic: if \(P \subseteq P'\) holds for two positive programs \(P\) and \(P'\), then \(\text{LM}(P) \subseteq \text{LM}(P')\). Gelfond and Lifschitz \([9]\) extend the least model semantics for arbitrary normal programs. Given a
that if \( M \subseteq \text{At}(P) \), the idea is to reduce \( P \) to a positive program \( P^{M} = \{ r^{+} \mid r \in P \text{ and } M \cap B^{-}(r) = \emptyset \} \) having the least model \( \text{LM}(P^{M}) \). Equating this model with the model candidate \( M \) implies the definition of a stable model \([9]; M = \text{LM}(P^{M}) \) which implies \( M \models P \), but not vice versa.

In general, a normal logic program need not have a unique stable model nor stable models at all. The stable model semantics of normal programs was preceded by an alternative semantics, namely the one based on supported models \([1]\). A classical model \( M \) of a normal program \( P \) is a supported model of \( P \) if for every atom \( a \in M \) there is a rule \( r \in P \) such that \( H(r) = a \) and \( M \models B(r) \). Inspired by this idea, we define for any program \( P \) and \( I \subseteq \text{At}(P) \), the set of supporting rules \( \text{SR}(P, I) = \{ r \in P \mid I \models B(r) \} \subseteq P \). As shown in \([17]\), any stable model \( M \subseteq \text{At}(P) \) of a normal logic program \( P \) is also a supported model of \( P \), but not vice versa in general.

We define classical literals in the standard way using classical negation as the connective. Syntactically, a clause \( C = \{ a_{1}, \ldots, a_{n}, \neg b_{1}, \ldots, \neg b_{m} \} \) is a finite set of classical literals representing a disjunction of its constituents. A set of clauses \( S \) represents a conjunction of the clauses contained in it. We define the set of atoms \( \text{At}(S) \) and interpretations \( I \subseteq \text{At}(S) \) in analogy to normal programs. A clause \( C \) of the form above is satisfied in an interpretation \( I \) if \( I \models a \), for some \( i \in \{ 1, \ldots, n \} \) or \( I \not\models b \), for some \( i \in \{ 1, \ldots, m \} \).

An interpretation \( I \subseteq \text{At}(S) \) is a classical model of \( S \), denoted by \( I \models S \), iff each clause \( C \in S \) is satisfied in \( I \). Finally, a set of clauses \( S \) gives rise to a set of classical models \( \text{CM}(S) = \{ M \subseteq \text{At}(S) \mid M = \text{At}(C) \} \). This differs essentially from a normal program \( P \) for which the set of stable models \( \text{SM}(P) = \{ M \subseteq \text{At}(P) \mid M = \text{LM}(P^{M}) \} \) is of interest.

3 CHARACTERIZING STABILITY

In this section, we characterize stable models in terms of supported models and level numberings to be defined next.

**Definition 1** Let \( M \) be a supported model of a normal program \( P \). A function \# : \( M \cup \text{SR}(P, M) \to \mathbb{N} \) is a level numbering w.r.t. \( M \) iff for all \( a \in M \),

\[
\# a = \min \{ \# r \mid r \in \text{SR}(P, M) \text{ and } a = H(r) \}
\]

and for all \( r \in \text{SR}(P, M) \),

\[
\# r = \max \{ \# b \mid b \in B^{+}(r) \} + 1
\]

where we interpret \( \max \emptyset = 0 \) to cover rules \( r \) with \( B^{+}(r) = \emptyset \).

It is important to realize that a level numbering need not exist for every supported model, as demonstrated below.

**Example 2** Consider a logic program \( P \) consisting of two rules \( r_{1} = a \leftarrow b \) and \( r_{2} = b \leftarrow a \). There are two supported models of \( P \): \( M_{1} = \emptyset \) and \( M_{2} = \{ a, b \} \). The first model has a trivial level numbering with an empty domain, since \( M_{1} \cup \text{SR}(P, M_{1}) = \emptyset \). For the second, the domain \( M_{2} \cup \text{SR}(P, M_{2}) = M_{2} \) and \( P \). The requirements in Definition 1 lead to four equations: \( \# a = \# r_{1}, \# r_{1} = \# b + 1, \# b = \# r_{2}, \) and \( \# r_{2} = \# a + 1 \). These imply \( \# a = \# a + 2 \), which has no solutions. Hence there is no level numbering w.r.t. \( M_{2} \). □

**Proposition 3** Let \( M \) be a supported model of \( P \). If there is a level numbering \# w.r.t. \( M \), then \# is unique.

The key observation is that the existence of a level numbering is inherently connected to stability. To determine level numberings in practice, we resort to the van Emden-Kowalski operator \( T_{P} \) which is defined by \( T_{P}(A) = \{ H(r) \mid r \in P \text{ and } B^{+}(r) \subseteq A \} \) for a positive program \( P \) and any set of atoms \( A \subseteq \text{At}(P) \). The iteration sequence of \( T_{P} \) is then defined inductively as follows: \( T_{P} \uparrow 0 = \emptyset, T_{P} \uparrow i = T_{P}(T_{P} \uparrow i - 1) \) for \( i > 0 \), and \( T_{P} \uparrow \omega = \bigcup_{i < \omega} T_{P} \uparrow i \). Then we have \( \text{LM}(P) = T_{P} \uparrow \omega \). This defines extends for rules \( r \in \text{SR}(P, \text{LM}(P)) \) in analogy to \((3): \)

\[
\text{lev}(r) = \max \{ \text{lev}(b) \mid b \in B^{+}(r) \} + 1
\]

Assigning level numbers in this way is compatible with Definition 1 which implies a characterization of stable models based on the existence of level numberings.

**Theorem 4** Let \( P \) be a normal program.

1. If \( M \) is a stable model of \( P \), then \( M \) is a supported model of \( P \) and there is a unique level numbering \# : \( M \cup \text{SR}(P, M) \to \mathbb{N} \) w.r.t. \( M \) as defined above.

   (a) For atoms \( a \in M \), let \( \# a = \text{lev}(a) \).

   (b) For rules \( r \in \text{SR}(P, M) \), let \( \# r = \text{lev}(r^{+}) \).

2. If \( M \) is a supported model of \( P \) and there is a level numbering \# : \( M \cup \text{SR}(P, M) \to \mathbb{N} \) w.r.t. \( M \), then \# is unique and \( M \) is a stable model of \( P \).

4 NEW CLAUSAL REPRESENTATION

In this section, we develop a new way to translate a normal logic program into a set of clauses so that a tight correspondence of models is obtained. More precisely, we aim at faithfulness in the sense proposed in \([10, 11]\): the stable models of a program \( P \) and the (stable) models of its translation \( T_{P} \) are in a bijective relationship and coincide up to \( \text{At}(P) \). This is to properly preserve the semantics of the program, including the number of models \(^2\). As a further requirement, we try to keep the length of the translation \( \| T_{P}(P) \| \) as low as possible; preferably sub-quadratic, i.e. of order \( \| P \| \times \log_{2} \| \text{At}(P) \| \).

The translation of a normal program \( P \) takes place in two subsequent steps. First, we remove positive body atoms from all rules of \( P \). The result is an atomic normal program \( T_{\text{AT}}(P) \) which is then easy to convert into a set of clauses using Clark’s completion. However, the first step is much more complicated. Our idea is to apply the characterization of stable models developed in Section 3 so that each stable model \( M \) of a normal program \( P \) is eventually captured as a supported model \( M \) of \( P \) possessing a level numbering w.r.t. \( M \). Let us consider another example on a level numbering in order to better understand the range taken by level numbers.

**Example 5** Let \( P = \{ r_{1} = a \leftarrow b; r_{2} = a \leftarrow b; r_{3} = b \leftarrow a \} \) be a (positive) normal program. The unique stable model \( M = \text{LM}(P) = \{ a, b \} \) is supported by the set of rules \( \text{SR}(P, M) = P \). The unique level numbering \# w.r.t. \( M \) is determined by \( \# r_{1} = 1, \# a = 1, \# r_{3} = 2, \# b = 2, \) and \( \# r_{2} = 3 \). □

\(^{2}\) Note that an interpretation \( M \subseteq \text{At}(P) \) is a supported model of \( P \) if for all \( a \in M \), let \( \# a = \text{lev}(a) \). □

\(^{3}\) This is essential, as models correspond to solutions in ASP.
4.1 Representing Level Numbers

It is natural to use a binary encoding when representing individual level numbers determined by a level numbering \( \# \) in terms of propositional atoms. Unfortunately, every atom in \( \mathcal{A}(P) \) may be assigned a different level number in the worst case, as demonstrated in Example 5. Thus the level numbers of atoms may vary from 1 to \( \mathcal{A}(P) \) and the highest possible level number of a rule \( r \in P \) is \( \mathcal{A}(P) + 1 \), as for \( r_2 \) in our example. Although level numbers are positive by definition, we leave room for 0 which is to act as the least binary value. Therefore, the maximum number of bits in level numbers is

\[
\log_2(\mathcal{A}(P) + 2).
\]

Given the number of bits \( b \), a natural number \( 0 \leq n < 2^b \), and \( 0 < i \leq b \), we write \( n[i] \) for the \( i \)'th bit in the binary representation of \( n \) in the decreasing order of significance. Our idea is to encode the level number \( \#a \) of an atom \( a \in \mathcal{A}(P) \) using a vector \( a_1, \ldots, a_j \) of new atoms. Such a vector can be understood as a representation of a binary counter of \( j \) bits, which is to hold \( \#a \) as its binary value. Since atoms may take only two values under the stable model semantics, we aim at the following relationship: \( \#a[i] = 1 \) (resp. \( \#a[i] = 0 \)) iff \( a \) evaluates to true (resp. false) under stable model semantics.

In order to capture level numberings with binary counters, we need certain primitive operations to be used as subprograms of the forthcoming translation \( \mathcal{A}(P) \). The first set of subprograms, as listed in upper half of Table 1, concentrates on setting counters to particular values. Each subprogram is to be activated only when an additional controlling atom \( c \) cannot be inferred by other rules. The first subprogram \( \text{SEL}_j(a, c) \) selects a value \( 0 \leq n < 2^j \) for the \( i \)'th counter associated with an atom \( a \). Note that the new atoms \( \overline{a}_1, \ldots, \overline{a}_j \) act as complements of \( a_1, \ldots, a_j \) and we need them to keep subprograms and the overall translation \( \mathcal{A}(P) \) atomic. The second subprogram \( \text{NXT}_j(a, b, c) \) binds the values of the binary counters associated with atoms \( a \) and \( b \), respectively, so that the latter is the former increased by one (modulo \( 2^j \)). The last subprogram \( \text{FIX}_j(a, n, c) \) assigns a fixed value \( 0 \leq n < 2^j \) to the counter associated with \( a \).

There is also a need to compare values. The fourth subprogram \( \text{LT}_j(a, b, c) \) checks if the value of the binary counter associated with an atom \( a \) is strictly lower than the value of the binary counter associated with another atom \( b \). To keep the program linear in \( j \), we need a vector of new atoms \( \overline{a}(b), \overline{b}(a) \) \( j \) plus their complements which we associate with \( a \) and \( b \). The atoms \( \overline{a}(b) \) and \( \overline{b}(a) \), which refer to the most significant bits, capture the result of the comparison. The fifth subprogram \( \text{EQ}_j(a, b, c) \) checks if the counters associated with \( a \) and \( b \) hold the same value. Only two new atoms \( \overline{a}(b) \) and \( \overline{b}(a) \) are needed for the result.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Primitive & Definition with atomic rules \\
\hline
\text{SEL}_j(a, c) & \{a_1 \ldots a_i \overline{c} \overline{a}_i \overline{a}_i \overline{c} \quad 0 < i \leq j\} \\
\text{NXT}_j(a, b, c) & \{b_1 \ldots b_i \overline{c} b_i \overline{b}_i \overline{b}_i \overline{c} \quad 0 < i \leq j\} \\
\text{FIX}_j(a, n, c) & \{a_1 \ldots a_i n[i] \overline{c} \overline{a}_i \overline{a}_i \overline{c} \quad 0 < i \leq j\} \\
\hline
\end{tabular}
\caption{Encoding primitive operations for binary counters}
\end{table}

4.2 Removing Positive Body Atoms

Any non-binary rule \( r \) with \( |B^+(r)| > 2 \) can be transformed into a set of binary rules in a faithful way using new atoms \([10]\). Thus we assume without loss of generality that programs under consideration are free of such rules. Moreover, we partition each program \( P \) into its strongly connected components (SCCs) \( C_1, \ldots, C_n \) using positive dependencies (cf. [14]). In the sequel, we describe the translation \( \mathcal{A}(C_i) \) for a SCC

\[
\mathcal{C}_i.
\]

Note that the atoms in the set \( \mathcal{H}(C_i) = \{ H(r) \mid r \in C_i \} \) are mutually reachable in the positive dependency graph \( DG^+(P) \) having \( H(P) \) and \( \{ H(r), b \mid r \in P \) and \( b \in B^+(P) \} \) as the sets of vertices and edges, respectively. The translation \( \mathcal{A}(C_i) \) is then obtained as \( \bigcup_{i=1}^{n} \mathcal{A}(C_i) \). In the sequel, we describe the contribution of an atom \( a \in \mathcal{H}(C_i) \) to \( \mathcal{A}(C_i) \).

If \( a \) appears positively in \( P \), we introduce a new atom \( a_i \) equal to the complement of \( a \), and its definition \( \overline{a_i} \) is \( \overline{a_i} \) to \( \overline{a_i} \) in \( \mathcal{C}_i \). If \( \mathcal{H}(C_i) = 1 \), i.e. \( \mathcal{H}(C_i) = \{a\} \), it is sufficient to include

\[
a \overline{a_1} \overline{a_2} \ldots \overline{a_j}, \overline{a_1}, \overline{a_2} \overline{a_3} \ldots \overline{a_m} (5)
\]

as the translation of a rule \( r \in C_i \) with \( H(r) = a \). If \( a \in B^+(r) \), then (5) can be omitted. On the other hand, if \( \mathcal{H}(C_i) > 1 \), we have to introduce two binary counters for \( a \), one for holding \( \#a \) and the other for holding \( \#a + 1 \) mod \( 2^{|C_i|} \) where \( |C_i| = \log_2(|\mathcal{H}(C_i)| + 2) \) is defined according to (4).

These counters are represented using vectors of new atoms \( \overline{a_i}, \ldots, \overline{a_i} \overline{a_j} \) and \( \overline{a_i} \overline{a_j} \), respectively, including their complements. To set the values of these counters appropriately, we include subprograms \( \text{SEL}_{\text{VEC}}(\overline{a_i}, \overline{a_j}) \) and \( \text{NXT}_{\text{VEC}}(\overline{a_i}, \overline{a_j}) \) in \( \mathcal{A}(C_i) \). Note that these subprograms are activated only if \( \overline{a} \) cannot be inferred, i.e. \( a \) is inferable.

The constraints associated with a rule \( r \) having \( H(r) = a \) will be conditioned with a negative body atom \( b \) which whenever not inferable captures the fact that
$r \in \text{SR}(P, M)$. If $r$ is atomic, we use the subprogram $\text{FIX}_{\text{CTR}}(\text{ctr}(a), 1, \text{bt}(r))$ to ensure $#a = 1$ whenever $r \in \text{SR}(P, M)$; and a rule $\text{min}(a) \leftarrow \neg \text{bt}(r)$ to infer the minimality of the value held by $\text{ctr}(a)$, as insisted by (2).

On the other hand, if $r$ is unary with $H(r) = a$ and $B^+(r) = \{b\}$, two cases arise. If $b \notin H(C)$, then $r$ is transcribed like unary rules. Otherwise, we have $b \in H(C)$ and the value of (3) is held by $\text{nt}(b)$. This is needed to express the contribution of $r$ in (2) w.r.t. $H(r) = a$. For this purpose, we have to compare the values held by the counters $\text{nt}(b)$ and $\text{ctr}(a)$ using subprograms $\text{LT}_{\text{CTR}}(\text{ctr}(\text{nt}(b)), \text{ctr}(a), \text{bt}(r))$ and $\text{EQ}_{\text{CTR}}(\text{ctr}(\text{nt}(b)), \text{ctr}(a), \text{bt}(r))$. Then the requirement that $#a \geq #b + 1$, as insisted by (2), can be expressed in terms of a constraint $x \leftarrow \neg x, \neg \text{bt}(r), \neg r(\text{nt}(b), \text{ctr}(a))$ where $x$ is a new atom.

The other half of (2) is taken care by a rule of the form $\text{min}(a) \leftarrow \neg \text{bt}(r), \neg \text{eq}(\text{nt}(b), \text{ctr}(a))$.

Example 6 Let us reconsider the program $P$ from Example 2. Both rules and atoms are involved in the only SCC of $P$, say $C$, so that $\mathcal{NC} = 2$. The translation $\text{TR}_{\text{AT}}(P) = \text{TR}_{\text{AT}}(C)$ contains the following rules associated with $a$: $a \leftarrow \neg a$; $x \leftarrow \neg x, \neg x, \neg \text{min}(a)$; $\text{bt}(r_1) \leftarrow \neg \text{bt}(r_2)$; $\text{bt}(r_2) \leftarrow \neg \text{bt}(r_1)$; $\text{min}(b) \leftarrow \neg \text{bt}(r_1)$; $\neg \text{eq}(\text{nt}(a), \text{ctr}(b))$ in addition to four subprograms for choosing the values of $\text{ctr}(a)$ and $\text{nt}(a)$ as well as comparing the latter with $\text{ctr}(b)$. The rules for $b$ are symmetric, just exchange the roles of $a$ and $b$ and $r_2$ and $r_1$. The only stable model $M = \emptyset$ of $P$ is then captured as the only stable model $N = \{x, b, \text{bt}(r_1), \text{bt}(r_2)\}$ of $\text{TR}_{\text{AT}}(P)$.

4.3 From Atomic Rules to Clauses

Atomic normal programs provide a promising intermediary representation that is straightforward to translate into a set of propositional clauses. Such programs are positive order consistent in the sense proposed by Fages [8]. As a consequence, stable and supported models coincide for this class of programs, and Clark’s program completion is sufficient to capture stable models in a faithful way. However, new atoms must be introduced in order to keep the translation linear. Since this is a quite standard procedure, we skip the details.

5 EXPERIMENTS

We have implemented the translation described in Section 4. The implementation consists of two translators called $\text{LP2ATOMIC}$ and $\text{LP2SAT}$, which correspond to the two phases of the translation. The task of $\text{LP2ATOMIC}$ is to translate away positive body atoms from a normal program given as input in the internal file format of the $\text{SMODELS}$ system [21]; typically produced by the front-end $\text{LPARSE}$. The latter translator, $\text{LP2SAT}$, takes the output of $\text{LP2ATOMIC}$ as its input and produces Clark’s completion for the program. The output is in the $\text{DIMACS}$ format which is understood by most SAT solvers.

As a test program, we use a normal logic program given in Figure 1. The program is given in the input syntax of $\text{LPARSE}$ and it formalizes the problem of finding any subgraph of $D_n$, i.e. the complete directed graph with $n$ vertices and $n^2$ edges, in which all vertices are still reachable from each other. In our benchmark, the task is to compute all stable models of the program instantiated by $\text{LPARSE}$ when $n$ varies from 1 to 5. As a result, the number of SCCs and positive loops increases.

Figure 1. Normal logic program used in the benchmark

\text{vertex}(1..n).\notag
\text{in}(V1,V2) :- \neg \text{out}(V1,V2), \text{vertex}(V1;V2), V1!=V2.\notag
\text{out}(V1,V2) :- \neg \text{in}(V1,V2), \text{vertex}(V1;V2), V1!=V2.\notag
\text{reach}(V,V) :- \text{vertex}(V).\notag
\text{reach}(V1,V3) :- \text{in}(V1,V2), \text{reach}(V2,V3), V1!=V2, V1!=V3.\notag
\neg \text{not reach}(V1,V2), \text{vertex}(V1;V2).\notag

We run five benchmark instances generated by $\text{LPARSE}$ on six different systems also listed in Table 2: $\text{SMODELS}$, $\text{CMODELS}$ [13], and four combinations of $\text{LP2ATOMIC}$ and $\text{LP2SAT}$ with other solvers. The first combines plain $\text{LP2ATOMIC}$ with $\text{SMODELS}$ just to get an idea how much overhead results from the removal of positive body atoms. The second combination uses both translators and a state-of-the-art SAT solver $\text{CHAFF}$ [19] for the actual computation of classical models corresponding to stable models. The third is the same as the second except another solver, namely $\text{RELSAT}$ [3], is used as the back-end. The last system incorporates a strengthened well-founded reduction to this setting: we use $\text{SMODELS}$ to simplify the intermediate program representations before and after invoking $\text{LP2ATOMIC}$. This has a favorable effect on the number of clauses generated, as notable from Table 2.

Our benchmark is easy for $\text{SMODELS}$ which reaches a performance of 47 kMPS (models per second) on a 1.67 GHz CPU. However, the main objective here is to compare $\text{CMODELS}$ with our approach, as it is also based on the idea of using SAT solvers to compute stable models instead of a special purpose engine like $\text{SMODELS}$. We did not include $\text{ASSAT}$ [15], as it can compute only one stable model for a program given as input. When $n = 4$ the performance of the systems based on $\text{LP2ATOMIC}$ and $\text{LP2SAT}$ is between 1.0–2.8 kMPS, which clearly exceeds that of $\text{CMODELS}$, i.e. only 5.5 Mps. When $n = 5$, $\text{CMODELS}$ exceeds the time limit of 24 hours and $\text{CHAFF}$
The research reported in this paper is partially funded by the Academy of Finland (project #53695) and the European commission (contract IST-FET-2001-37004).

ACKNOWLEDGMENTS

The research reported in this paper is partially funded by the Academy of Finland (project #53695) and the European commission (contract IST-FET-2001-37004).

REFERENCES


Table 2. Timings in seconds when computing all stable models

<table>
<thead>
<tr>
<th>Vertices</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Models</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>1606</td>
<td>565080</td>
</tr>
<tr>
<td>SCCs with</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Rules (LP2ORE)</td>
<td>3</td>
<td>14</td>
<td>39</td>
<td>84</td>
<td>155</td>
</tr>
<tr>
<td>Rules (LP2ATOMIC)</td>
<td>3</td>
<td>18</td>
<td>240</td>
<td>664</td>
<td>1920</td>
</tr>
<tr>
<td>Clauses (LP2SAT)</td>
<td>4</td>
<td>36</td>
<td>818</td>
<td>2386</td>
<td>7642</td>
</tr>
<tr>
<td>Clauses (WF+LP2SAT)</td>
<td>2</td>
<td>10</td>
<td>553</td>
<td>1677</td>
<td>5971</td>
</tr>
</tbody>
</table>

runs out of memory (1 GB) as the back-end of the fourth system. The systems perform differently when we compute only one stable model for the program and \( n = 8 \). The respective timings are 0.012, 0.043, >10\(^2\), 0.80, 2.6 and 2.8 seconds for the systems in Table 2; and 0.020 seconds for ASSAT [15].