IFI TECHNICAL REPORTS

Institute of Computer Science,
Clausthal University of Technology

IfI-05-04

Clausthal-Zellerfeld 2005

Stable Model Semantics Algorithm: Approach Based on Relation of Blocking Between Sets of Defaults

Peter Novák

Institute of Computer Science, Clausthal University of Technology, Germany
peter.novak@in.tu-clausthal.de

Abstract

In the last years computational logic, and particularly non-monotonic reasoning, was introduced as a theory suitable for specification and implementation of multi-agent systems. In this context, fast algorithms for reasoning in computational logics are vitally important to support an industrial applications of such systems.
In this paper we investigate the properties of sets of default assumptions with respect to the stable model semantics of normal logic programs. At first, we introduce the notion of *generating set*, a set of default assumptions we have to accept in order to infer particular atom of the program.
Subsequently we introduce a construction of *canonical program* of a given normal logic program, which is syntactically much simpler, but preserves all the information of the original program w.r.t. stable model semantics. Based on the semantical relation between the original normal logic program and its canonical counterpart, we introduce an alternative definition of *strong semantical equivalence*, which we believe, is equivalent to standard definition.
Finally we show how a relation of *blocking* between generating sets can be exploited in order to convert the problem of search for stable models of normal logic program to the problem of graph coloring, what finally leads to a new algorithm of stable model search. The idea of our algorithm is based on the observation that accepting a generating set of default assumptions into the partial interpretation forces other generating sets to excluded from it. The brief descriptions of the prototype implementation, and first results of experiments with it, are also included in this paper.
Our approach is in many ways very simillar to some already published works (e.g. [Dimopoulos and Torres, 1996]).

1 Introduction

1.1 Motivation

In recent years, stable model semantics [M. Gelfond, 1988] and its extensions to systems evolving in time ([J. J. Alferes, 2000] and [J. A. Leite, 2001]) proved their powers in the field of knowledge representation and non-monotonic reasoning as an underlying theory for multi-agent systems [J. J. Alferes, 2001][J. A. Leite and Pereira, 2000][J. Dix, 2002]. To use theoretical results of these research directions in industrial applications, fast and efficient algorithms for stable model semantics are needed.

In the course of last decade various implementations of stable model semantics were developed, in order to prove the usefulness of this kind of non-monotonic reasoning not only in theory, but also in practice. Although in the meantime, the utilization of such tools is mainly in the academic scope, attempts to use stable semantics as an underlying engine for industrial prototypes were made (e.g. [de Lima Nogueira, 2003]). However, we believe that this paradigm has the ability to evolve into a powerful industrial tool in the future. Therefore fast and efficient implementations and optimizations of stable model semantics have to be developed. The aim of this paper is to introduce an original algorithm for computing stable model semantics of normal logic programs, based on properties of sets of default assumptions extracted from a given normal logic program.

The most recognized implementations of stable model semantics algorithms are currently *DLV*, by Nicola Leone et. al. [Eiter, 2000] and *smodels*, by Ilkka Niemelä et. al. [I. Niemelä, 1997]. Although designed for generation of extensions of default theories, *DeReS* system described in [P. Cholewinski, 1996] is also capable to compute stable models of normal logic programs. Comprehensive compilation of algorithms for stable model semantics, together with description of some implementation techniques, is provided by [Baral, 2003], or [Simons, 2000].

According to [Baral, 2003], the common feature of algorithms used by *DLV* and *smodels* is, that given a partial interpretation, they first try to *extend* it either by using a form of derivation based on algorithms computing well-founded semantics, or using some properties of stable models with respect to a given partial interpretation and the syntax of the program. If that fails, then they arbitrarily select a default literal, or use a heuristic to decide on a default literal to be included to the current partial interpretation and then iteratively extend the resulting partial interpretation. These attempts to extend the partial interpretation continue until a stable model is found, or a contradiction is obtained (what forces an employment of some backtrack strategy). As it is obvious, the most important points during the computation, are choices of default literals, when the extension process was unable to add new literals into the partial interpretation. The naive algorithm computing stable models, would have to enter such a branching point for each default literal in the program. Such an approach results into the exploration of the whole search space, which can be huge even for relatively small logic programs. We claim, that a preprocessing step, in which relations between sets of default literals are analyzed, can help to boost the computation of stable models of given logic programs.

Example 1

$$\begin{array}{lll} b \leftarrow not\, a. & g \leftarrow not\, h. & a \leftarrow g, not\, f. \\ c \leftarrow b. & h \leftarrow not\, g. & \\ e \leftarrow c, not\, d. & & \\ f \leftarrow e. & & \\ \bot \leftarrow e. & & \end{array}$$

For the purpose of this example, the program above is divided into three parts. The first two columns contain two disjoint programs, which are finally semantically interconnected by the clause in the third column. The program has one stable model $\{a, g\}$ (which can also be written as $\{a, g, not\, b, not\, c, not\, d, not\, e, not\, f, not\, h\}$).

There are few important observations which can potentially boost the computation of a stable model:

1. No stable model of program P can contain any of the following sets of default literals: $\{not\, a, not\, d\}$, $\{not\, a, not\, h, not\, f\}$ and $\{not\, g, not\, h\}$.

2. Because the atom d cannot be inferred from the given program, the only way to avoid accepting the set of default literals$\{not\, a, not\, d\}$ into a partial interpretation and thus cause a conflict in it, is to infer the atom a, by accepting default literals not h and not f in the early stage of the computation. This, in one step also avoids accepting of the set $\{not\, a, not\, h, not\, f\}$ and later also $\{not\, g, not\, h\}$, which also cause conflict in a partial interpretation.

3. Accepting a set $\{not\, h, not\, f\}$ leads to inference of atoms g and a. And because consistent accepting of any other default literals doesn't produce any new atom, which could cause a conflict, the set $\{a, g\}$ is a good candidate for a stable model (which it actually is).

The example 1 illustrates that there are dependencies between sets of default literals with respect to their inclusion into some stable model of a given program. These are based on the notion of *blocking*. Accepting the set of literals $\{not\, h, not\, f\}$ *blocks* accepting of the set $\{not\, a, not\, d\}$, which inevitably leads to a conflict, in any partial interpretation.

A naive algorithm, employing the iterative two-step strategy of extension of actual partial interpretation and subsequent random choice of a default literal will need the depth of about five branching points, where it will choose the default literal for exhaustive traversal of a search space. An algorithm exploiting the relation of blocking between sets of default literals could be able to rule out a number of default literals from the possibility of inclusion into a stable model, because they are present only in sets which are blocked by some already accepted set. In the case of the example 1, such a literal was $not\, d$, which as itself is harmless, but together with $not\, a$ causes a conflict.

Another important observation regarding the computation of stable models of normal logic programs is, that a set of default literals unambiguously corresponds with a set of positive atoms. Given a set of default literals A of a program P accepted in some unknown partial interpretation, we can directly extend this partial interpretation by employing the idea used in a Gelfond-Lifschitz transformation (i.e. compute the minimal model of a reduct created using only default literals from A).

Using bits mentioned above, we can turn the idea of search for a stable model upside-down by not looking for a consistent set of atoms of a given program, but simply looking for a consistent set of default literals which are generating a searched stable model. To find a consistent set of such default literals, we need a relation between such sets, which will say us when the set of default literals induces consistent partial interpretation and when it does not.

1.2 Contributions and Organization of the Paper

In this paper we introduce a relation of *blocking* between sets of default literals and later we show how it can be exploited, in order to convert the problem of a search for stable models of a given normal logic program to the problem of *coloring* the graph constructed according to this relation. As an important byproduct, we introduce an alternative definition of *strong semantical equivalence* between two normal logic programs, which we believe is equivalent to the standard definition introduced in [Vladimir Lifschitz,]. This particular issue will be investigated in the future. Finally we describe our prototype implementation of a system computing stable models of grounded normal logic programs, which we developed as a proof of the concept for the purposes of this paper.

2 Preliminaries

Before introducing our results, let us recall some basic definitions and notions.

Let $\mathcal{A}$ be a finite set (alphabet) of propositional symbols including $\perp$ (*atoms*). Let a be an atom. A *literal* (usually denoted as L) is either an atom ($L = a$), or an atom preceded by a default negation symbol not ($L = not\, a$). From this point on, we will usually denote atoms by indexed lower case letters a, b, c and general literals by indexed upper case letter L. Literals of the form $L = not\, a$ will be called *default assumptions*, or simply *defaults*. Set of all default literals will be denoted as $\mathcal{D}$. Set of all literals is defined as $Lit = \mathcal{A} \cup \mathcal{D}$. Let A be a set of atoms, then $not(A) = \{not\, a | a \in A\}$.

Rule r is a formula of the form $L \leftarrow L_1, \ldots, L_n$, where $n \geq 0$, L is an atom and $L_1, \ldots, L_n$ are literals. Literal L is called the *head* of the rule (denoted as $head(r)$), while the set of literals $\{L_1, \ldots, L_n\}$ is called the *body* of the rule r (denoted as $body(r)$). $body(r)$ can be split in two parts $body(r) = body^+(r) \cup body^-(r)$, where $body^+(r) \subseteq \mathcal{A}$ and $body^-(r) \subseteq \mathcal{D}$.

A *normal logic program* P (*program*) is a finite set of rules over an alphabet $\mathcal{A}$. From this point on, when the context will be clear, we will use $\mathcal{A}$,Lit and $\mathcal{D}$ w.r.t. to an alphabet of the given program P. The set of all default assumptions of a program P is defined as $\mathcal{D}(P) = \bigcup_{r \in P} body^-(r)$.

We say that a set of literals $A = A^+ \cup not(A^-)$, where $A^+, A^- \subseteq \mathcal{A}$ is said to be *consistent* iff $A^+ \cap A^- = \emptyset$ and $\bot \notin A$. A *partial interpretation*[1] of a normal logic program P is a consistent set of literals over alphabet $\mathcal{A}$ w.r.t. P. A *total interpretation* (*interpretation*) $I = I^+ \cup not(I^-)$ is a partial interpretation, where $I^+ \cup I^- = \mathcal{A}$. We say that an interpretation I *satisfies* a literal L ($I \models L$) iff $L \in I$. The set of literals is satisfied ($I \models L_1, \ldots, L_k$), iff $I \models L_1, \ldots, I \models L_k$. Finally the rule $r = L \leftarrow L_1, \ldots, L_n$ is satisfied w.r.t. to an interpretation I iff, whenever $L_1, \ldots, L_n$ is satisfied by I, then L is also satisfied by I. We say that the an interpretation M is a *model* of a program P, iff $\forall r \in P : M \models r$.

Let the *least model* of a program be defined as usually (see [M. van Emden, 1976]). The original definition of *stable model* semantics was introduced in [M. Gelfond, 1988] as follows:

Definition 1 ***(Gelfond-Lifschitz operator)*** *Let P be a program and I be a partial interpretation. The GL-transformation of P w.r.t. I (called also* reduct *and denoted as $\frac{P}{I}$) is obtained from P by performing the following operations:*

- *remove from P all rules containing a default literal $not\, a$, such that $a \in I^+$;*
- *remove from all remaining rules all default literals.*

Since the resulting program $\frac{P}{I}$ is a definite program, it has a unique minimal model $least(\frac{P}{I})$ ([M. van Emden, 1976]). Hence a *stable model* is defined by the following definition.

Definition 2 ***(Stable model semantics)*** *A total interpretation M is a* stable model *of a program P, iff $M = least(\frac{P}{M})$.*

Let M be a partial interpretation of a normal logic program P.

Corollary 1 *M is a stable model of a program P if, and only if for each $a \in M^+$, there exists a minimal set of rules $P_a \subseteq P$, such that $a \in least(\frac{P_a}{M})$ and for each $a' \in M^-$, there's no $P_{a'} \subseteq P$, such that $a' \in least(\frac{P_{a'}}{M})$.*

PROOF. The corollary follows from the definition of a stable model of a normal logic program.

$\Longrightarrow$: Let's realize that, given a stable model M, not all the rules of program P participate on the inference of each $a \in M^+$, thus there has to be a minimal set of rules $P_a \subseteq P$, which is sufficient to consider (in a worst case let $P = P_a$) in order to infer a w.r.t. a stable model M. For the second part of the implication assume that there exists some $P_{a'} \subseteq P$, such that for some $a' \in M^-$ holds that $a' \in least(\frac{P_{a'}}{M})$. Then because $P_{a'} \subseteq P$, also $a' \in least(\frac{P}{M})$, what is a conflict with the assumption, that $a' \in M^-$.

[1]For the purposes of this paper, we will use a rather non-standard definition of interpretation, where default literals appear explicitly.

$\Longleftarrow$: Let $M = M^+ \cup not(M^-)$ be a set of literals such, that for each $a \in \mathcal{D}(P)$ holds, that if there exists $P_a \subseteq P$, such that $a \in least(\frac{P_a}{M})$, than $a \in M^+$. Otherwise $a \in M^-$. M is a total interpretation of P.

Now for each $a \in M^+$, exists $P_a \subseteq P$, such that $a \in least(\frac{P_a}{M})$, therefore also $a \in least(\frac{P}{M})$. And because for no $a' \in M^-$, there's no $P_{a'} \subseteq P$, such that $a' \in least(\frac{P_{a'}}{M})$, we also have that $a' \notin least(\frac{P}{M})$. Therefore $M = least(\frac{P}{M})$, thus M must be a stable model of P. □

3 Generating Sets

The corollary 2 from the Sect. 2 shows two important observations. The first is, that not all the rules of the normal logic program P contribute to the inference of a particular atom, thus for each positive literal in the final stable model, there is a subprogram of the program P, which models this atom. The second is hidden in the construction of the reduct of the given logic program, with respect to some partial interpretation. Particularly the observation is, that given a subprogram $P_a \subseteq P$ modeling an atom a w.r.t. to some partial interpretation, the partial interpretation must contain the set of all default literals from P_a, in order to introduce the atom a in a positive form using this subprogram. This leads us to the idea of a *generating set* of default literals for a given atom a.

Definition 3 ***(Generating set)*** *Let P be a normal logic program and let a be an atom. We say that S_a is a* direct generating set *of atom a, iff there exists $P_a \subseteq P$, such that $S_a = \mathcal{D}(P_a)$, $a \in least(\frac{P_a}{\mathcal{D}(P_a)})$ and $\forall P'_a \subset P_a : a \notin least(\frac{P'_a}{\mathcal{D}(P'_a)})$.*

We say, that the set $S_{a'}$ is an indirect generating set *of atom a, iff there exists $a' \neq a$, such that $S_{a'}$ is a direct generating set of a' and there exists a direct generating set S_a of atom a, such that $S_a \subset S_{a'}$ (i.e. $a \in least(\frac{P_{a'}}{S_{a'}})$).*

The set of all generating sets of an atom a is defined as follows:

$$GSets^P(a) = \{S | S \textit{ is a direct or indirect generating set of } a \textit{ w.r.t. } P\}$$

Simply said, a *direct* generating set of an atom a is a minimal set of atoms, which must be accepted in negative form , in order to infer this atom from the program P. *Indirect* generating set of an atom a is a direct generating set of some different atom, which, as a byproduct, forces inference of atom a. Direct and indirect generating sets will generally be referred to as *generating sets.*

As it is obvious, there can be more generating sets for one particular atom a. This is because there may be several different rules which trigger inference of a (i.e. $head(r) = a$).

Note also the special meaning of the sets from $GSets^P(\bot)$. Such sets, can be called *conflict triggers*, because accepting such a set in a negative form in some partial interpretation, causes inconsistency (or conflict) $\bot$.

In the following we show how generating sets of a given program P can be used to construct another normal logic program P°, which, indeed syntactically different, has the same properties as P w.r.t. stable semantics as the original program.

Definition 4 ***(Canonical logic program)*** *Let P be a normal logic program over an alphabet $\mathcal{A}$. We construct* canonical logic program *P° of P by the following construction:*

$$P^\circ = \{r | \exists a \in \mathcal{A} : \exists S \subseteq \mathcal{A} : head(r) = a \wedge body(r) = not(S) \Leftrightarrow S \in GSets^P(a)\}$$

Corollary 2 *Let P be a normal logic program and P° be its canonical logic program. Finally let a be an atom of the program P. Then $GSets^P(a) = GSets^{P^\circ}(a)$.*

Each generating set of an atom a is a direct generating set of some (possibly different) atom a'. Each direct generating set $S_a \in GSets^{P^\circ}(a)$ is defined by exactly one rule $r \in P^\circ$, so $P^\circ_a = \{r\}$, is minimal w.r.t. set inclusion such, that $a \in least(\frac{P^\circ_a}{S_a})$. Finally, from the construction of P° we have, that there exists $P_a \subseteq P$, such that $a \in least(\frac{P_a}{S_a})$. This holds in both directions of equivalence, therefore S_a is a generating set of both P and P°.

Theorem 1 *Let P be a normal logic program and P° be its canonical logic program. The following holds*

$$S \text{ is a stable model of } P \Longleftrightarrow S \text{ is a stable model of } P^\circ$$

PROOF. At first realize that in order to produce the reduct $\frac{P}{M}$ of some program P, the GL-transformation can only use defaults from $\mathcal{D}(P)$ as a reason for removing some rule $r \in P$ from the reduct. And if the program P is minimal set of rules, such that atom a is still present in the minimal model of its reduct, no rule from P could be removed from it, in order to obtain the reduct $\frac{P}{M}$, what means that $\mathcal{D}(P) \subseteq M^-$. Therefore both $\frac{P}{M} = \frac{P}{\mathcal{D}(P)}$ and $least(\frac{P}{M}) = least(\frac{P}{\mathcal{D}(P)})$ hold for such program P.

Now the proof of the theorem itself follows in subsequently equivalent steps, which prove the equivalence in both directions:

M is a stable model of the program P.

1. According to corollary 2, for each $a \in M^+$, there exists a minimal set of rules $P_a \subseteq P$, such that $a \in least(\frac{P_a}{M})$ **and** for each $a' \in M^-$ holds that $\forall P_{a'} \subseteq P$: $a' \notin least(\frac{P_{a'}}{M})$.
2. From the remark at the beginning of this proof and the definition 3 we have, that P_a is precisely the set of rules defining some direct generating set S_a of an atom a (i.e. $\mathcal{D}(P_a) = S_a \in GSets^P(a)$). Therefore, for each $a \in M^+$, there is a generating set $S_a \in GSets^P(a)$ induced by the set P_a (i.e. $S_a = \mathcal{D}(P_a)$), such that $S_a \subseteq M^-$ **and** because for each $a' \in M^-$there's no $P_{a'} \subseteq P$, such that $a' \in least(\frac{P_{a'}}{M})$, there also cannot be such $S_{a'} = \mathcal{D}(P_{a'}) \in GSets^P(a)$ for which $a' \in least(\frac{P_{a'}}{S_{a'}})$ for any $P_{a'}$.

3. From the construction of the canonical program P° we have, that P° contains rules with bodies corresponding to all generating sets of P. Thus for each $a \in M^+$, there exists $S_a \in GSets^P(a)$, such that $S_a \subseteq M^-$. Thus there also exists a set of rules $P^\circ_a = \{r^\circ\} \subseteq P^\circ$, such that $head(r^\circ) = a$ and $body(r^\circ) = not(S_a)$. Finally because $S_a = \mathcal{D}(P_a) = \mathcal{D}(P^\circ_a) \subseteq M^-$, for each $a \in M^+$ also $a \in least(\frac{P^\circ_a}{S_a})$. **And** from what was mentioned above and the corollary 2, we have that for each $a' \in M^-$there cannot be $P^\circ_{a'} \in P^\circ$ and $S_{a'} = \mathcal{D}(P_{a'}) = \mathcal{D}(P^\circ_{a'})$, such that $a' \in least(\frac{P^\circ_{a'}}{S_{a'}})$.

4. Now if for some $a \in M^+$ holds that $a \in least(\frac{P^\circ_a}{S_a})$ and $S_a \subseteq M^-$, then also $a \in least(\frac{P^\circ_a}{M})$ **and** if for no $a' \in M^-$, there exists $P^\circ_{a'} \subseteq P^\circ$, such that $a' \in least(\frac{P^\circ_{a'}}{S_{a'}})$, where $S_{a'} = \mathcal{D}(P^\circ_{a'}) \subseteq M^-$, then also $a' \notin least(\frac{P^\circ_{a'}}{M})$.

5. Finally from the corollary 2, we conclude that, M is a stable model of P°.

$\square$

Theorem 1, is the most important result of this section, because it shows us, that for the purpose of computing stable models of a given program P, we can use syntactically much simpler canonical program P°. Note, that the canonical program P° does not contain chains of rules like $\{b \leftarrow not\, c. a \leftarrow b.\}$. Thus in order to introduce the positive literal a into a partial interpretation during the computation of stable models of program P°, only one rule is needed ($a \leftarrow not\, c.$). By contrast, such inference would need two steps during the computation of stable models of P. At first we would have to infer the atom b and then, finally body of the rule $a \leftarrow b$ will be satisfied, what would lead directly to inference of a.

The remark above leads to a corollary:

Corollary 3 *M is a stable model of canonical program P° of some normal logic program P if, and only if for each $a \in M^+$, there exists a rule $r \in P^\circ$, such that $head(r) = a \wedge M \models body(r)$ and for no $a' \in M^-$, there's a rule $r' \in P^\circ$, in which $head(r') = a'$ and $M \models body(r')$.*

Example 2 *Let program P be:*

$e \leftarrow not\, c.$ $\quad a \leftarrow not\, c.$
$d \leftarrow not\, b.$ $\quad b \leftarrow not\, d.$
$c \leftarrow not\, a, d.$ $\quad f \leftarrow .$

Generating sets for each atom occurring in the head of some clause of program P are as follows:

$GSets^P(a) = \{\{c\}\}$
$GSets^P(b) = \{\{d\}\}$
$GSets^P(c) = \{\{a, b\}\}$
$GSets^P(d) = \{\{b\}, \{a, b\}\}$
$GSets^P(e) = \{c\}$
$GSets^P(f) = \{\emptyset\}$

Generally, the semantical equivalence between two normal logic programs w.r.t. stable model semantics, is a relation based on the equality of the set of stable models of these programs. In recent years, when more attention was dedicated to investigation of modifications of knowledge bases represented by logic programs, problems with the definition of semantical equivalence arose. It was observed (e.g. in [J. J. Alferes, 2000], or [Leite, 2003]), that although two logic programs share the same set of stable models, their behavior with respect to modification of the knowledge base can be completely different, even sometimes leading to counter-intuitive results. We believe, that the stronger notion of semantical equivalence is needed and we propose one based on the equivalence of generating sets. The notion of *strong semantic equivalence* was first introduced in [Vladimir Lifschitz,]. In this paper we introduce an alternative definition of this notion without a comparison of them. We believe that the two definitions are related and probably even equivalent. These issues will be investigated in the future.

Definition 5 ***(Strong semantic equivalence)*** *Let P_1 and P_2 be normal logic programs over an alphabet $\mathcal{A}$. We say that P_1* is strongly semantically equivalent to *P_2 (w.r.t. stable semantics) iff for each $a \in \mathcal{A}$ holds $GSets^{P_1}(a) = GSets^{P_2}(a)$.*

Corollary 4 *Normal logic programs P_1 and P_2 are strongly semantically equivalent iff $P_1^\circ = P_2^\circ$.*

4 Blocking Graph

In the motivation example 1 in Sect. 1, we already touched the idea, that there exists a relation between generating sets which could help to boost the computation of stable models of a given normal logic program P.

Let us consider a naive algorithm for computing stable model, which is used as a blueprint in the core of systems like *DLV* and *smodels*. In each branching point, such algorithms are considering only one default literal at a time, although there may be a dependency between different literals.

The example 1 shows us, that it is worthless to consider a choice of literal $not\, d$ alone at the beginning of the computation, because this literal itself is useless. Only together with the literal $not\, a$, the computation can move further. This is because there's only one generating set ($\{a, d\} \in GSets^P(e)$) in which the atom d occurs in the program from the example 1. Therefore we can either consider accepting the whole set of default assumptions $\{not\, a, not\, d\}$ into a given partial interpretation, or we have to find a reason for not including this set into that partial interpretation.

Including the set $\{not\, a, not\, d\}$ into a partial interpretation, directly leads to an observation, that the set $\{not\, h, not\, f\}$ cannot be subset of any stable model which includes the resulting partial interpretation. This is because $\{a, d\} \in GSets^P(f)$.

Contrary, if a stable model does not contain the set $\{not\, a, not\, d\}$, it must contain the set $\{not\, h, not\, f\}$, because the only way of introducing the positive literal a into any stable model, is to accept the set of default assumptions $\{not\, h, not\, f\}$.

As it is obvious, there's a direct correspondence between the sets of default literals discussed above and generating sets. What was informally discussed above are consequences of the following relation between sets of default literals:

Definition 6 ***(Blocking)*** *Let P be a normal logic program over an alphabet $\mathcal{A}$. Let also $a, a' \in \mathcal{A}$ be atoms.*

We say that the generating set $S_1 \in GSets^P(a)$ blocks *the generating set $S_2 \in GSets^P(a')$, iff $a \in S_2$.*

Simply, when the set $not(S_1)$ is included in some stable model of the program P, the set $not(S_2)$ cannot be a subset of this same stable model, because inconsistency would arise from the presence of both a and $not\, a$ in it. The relation of blocking creates a network of dependencies between generating sets which can be exploited in the process of computation of stable models of a program P. This network is embodied in the construction of the *blocking graph.*

Definition 7 ***(Blocking graph)*** *Let P be a normal logic program over an alphabet $\mathcal{A}$. The oriented graph $\mathcal{G}^P = (\mathcal{V}, \mathcal{E})$ is called the* blocking graph *of the program P iff*

- *set of nodes of $\mathcal{G}^P$ is $\mathcal{V} = \{S | \exists a \in \mathcal{A} : S \in GSets^P(a)\}$;*
- *set of edges of $\mathcal{G}^P$ is $\mathcal{E} = \{(S_1, S_2) | S_1 \text{ blocks } S_2\}$.*

In other words, there's an edge $e = (S_1, S_2)$ between two generating sets S_1 and S_2 in the graph $\mathcal{G}^P$, whenever the acceptation of a set of default assumptions S_1 in a partial interpretation leads to inference of an atom a, which is presented in the set S_2. Thus sets S_1 and S_2 cannot be accepted at the same time in any stable model of the program P. We also say, that accepting the set S_1 *blocks* accepting the set S_2 in any stable model of the program P.

Now we are prepared to define the algorithm for computing stable models of the program P using the graph $\mathcal{G}^P$.

Theorem 2 ***(Coloring the blocking graph)*** *Let P be a normal logic program and $\mathcal{G}^P = (\mathcal{V}, \mathcal{E})$ be its blocking graph. Let the color be the coloring of the graph $\mathcal{G}^P$ by two colors, with the following properties:*

Then $M = M^+ \cup (\mathcal{D} \setminus not(M^+))$ is a stable model of P if, and only if exists a coloring $color$ of the blocking graph $\mathcal{G}^P$, such that

1. *if $V \in \mathcal{V}$ and $color(V) = blue$, then $\forall W \in \mathcal{V} : ((V, W) \in \mathcal{E} \vee (W, V) \in \mathcal{E}) \Rightarrow color(W) = red$*

2. *if $V \in \mathcal{V}$ and $color(V) = red$, then $\exists W \in \mathcal{V} : (W, V) \in \mathcal{E} \wedge color(W) = blue$*

3. *for each $V \in \mathcal{V}$, if $V \in GSets^P(\bot)$, then $color(V) = red$*

and $M^+ = \{a | \exists S \in GSets^P(a) \wedge color(S) = blue\}$.

At first, note that if the union of all blue nodes stands for a partial interpretation, then *blue* nodes are those which are included in this partial interpretation and *red* nodes are those, which are not subsets of the particular partial interpretation. Hence the condition 1 says, that if a generating set S is a part of a partial interpretation I (in the form of $not(S)$), then none of the generating sets which are either *blocking* the set S, or are *blocked* by this set, is also a part of the partial interpretation I. The condition 2 similarly says, that there always must be a reason for exclusion of some generating set S from the partial interpretation (i.e. there must be some generating set S_1, already included in the partial interpretation, which is blocking the set S). Finally the condition 3 ensures, that the conflict $\bot$ cannot be a part of a partial interpretation (i.e. body of no rule r with $head(r) = \bot$ is satisfied).

Now the proof of Thm. 2 follows:

PROOF. From Thm. 1, we have that M being a stable model of P is equivalent to being a stable model of P°, so we can operate here with P° instead of P. Also note, that for each rule r of P°, there's a node $V = S_{head(r)}$ in the graph $\mathcal{G}^P = (\mathcal{V}, \mathcal{E})$, such that $S_{head(r)} \models body(r) \in GSets^P(head(r))$. We will also denote $M^- = \mathcal{D} \setminus not(M^+)$.

- $\Longleftarrow$: Let M be a set induced by some coloring satisfying conditions 1, 2 and 3 of Thm. 2.
  M is obviously a consistent set, because M cannot contain $\bot$ (due to condition 3 all generating sets of $\bot$ are red) and $M = M^+ \cup (\mathcal{D} \setminus not(M^+))$, thus M is a partial interpretation of P. Each $blue$ node, forces body of some rule r of P°, to be satisfied, hence from the definition of M we have that $\forall a \in M^+ : \exists r \in P^\circ : head(r) = a \wedge M \models body(r)$.
  To complete this part of the proof using the corollary 3, we have to show, that the body of no other rule $r' \in P^\circ$, such that $head(r') \in M^-$, is satisfied w.r.t. M. Let the body of a rule $r' \in P^\circ$, such that $a' = head(r') \in M^-$ be satisfied in M. Because $a' \notin M^+$, all nodes $S_{a'} \in \mathcal{V}$, such that $S_{a'} \in GSets^{P^\circ}(a')$ must be colored to red color. According to the condition 2, for each such node $S_{a'}$, there has to be a $blue$ node $S_b \subseteq M^-$, blocking $S_{a'}$, what means that, there is an atom $b \in S_{a'}$ for which $S_b \in GSets^{P^\circ}(b)$. The inclusion $S_b \subseteq M^-$ and the fact that S_b is a generating set of b, forces $b \in M^+$, but this, together with $b \in S_{a'} \subseteq M^-$ causes a contradiction with the consistency of M (a blue node contains an atom, which is satisfied in M^+). Therefore the body of no rule r', such that $head(r') \in M^-$ can be satisfied in M.
  Hence from the corollary 3, M is a stable model of P°.

- $\Longrightarrow$: Let M be a stable model of the program P°. Let $color$ be coloring of the blocking graph $\mathcal{G}^{P^\circ} = (\mathcal{V}, \mathcal{E})$, such that each node $V \in \mathcal{V}$, such that $V \subseteq M^-$ is colored to $blue$ color. Let all other nodes, to be colored to red color. We will show, that all nodes of the graph $\mathcal{G}^P$ satisfy the coloring conditions 1, 2 and 3.

  1. Because M is a stable model of P°, it does not contain $\perp$. Thus for all sets $S_\perp$, such that $S_\perp \in GSets^{P^\circ}(\perp)$ holds, that $S_\perp \nsubseteq M^-$. From the construction of the coloring $color$ we have that al such $S_\perp$ must therefore be red, by what the condition 3 is satisfied.
  2. To satisfy the condition 1 let $S_{blue} \in \mathcal{V}$ be a $blue$ node of the graph $\mathcal{G}^P$. Let $V^{in} = \{U \in \mathcal{V} | (U, S_{blue}) \in \mathcal{E}\}$ be a set of nodes with edges leading to S_{blue}. From the definition of blocking 6 and the definition of blocking graph 7 we have that for each $U \in V^{in}$, there exists $a \in S_{blue}$, such that $U \in GSets^{P^\circ}(a)$. Because $a \in S_{blue} \subseteq M^-$, for all such sets U holds, that $U \nsubseteq M^-$ (otherwise $a \in M^+$) , thus $color(U)$ must be red.
     Similarly let $V^{out} = \{U \in \mathcal{V} | (S_{blue}, U) \in \mathcal{E}\}$ be set of nodes to which there are edges leading from S_{blue}. Again from definitions 6 and 7 each $U \in V^{out}$ must contain some atom $a \in M^+$, such that $S_{blue} \in GSets^{P^\circ}(a)$. Therefore no such U is a subset of M^-, so it must be colored to red color.
  3. Let $S_{red} \in \mathcal{V}$ be a red node of the graph $\mathcal{G}^P$. From the construction of the coloring $color$ we have that $S_{red} \nsubseteq M^-$, thus there must be an atom $a \in S_{red}$, such that $a \in M^+$. But that means that there exists a generating set $S_a \in GSets^{P^\circ}(a)$, such that $S_a \subseteq M^-$. Therefore the node $S_a \in \mathcal{V}$ must be colored to $blue$ color by what the condition 2 is satisfied.

$\square$

The theorem 2, straightforwardly leads to an algorithm for computation of the stable models of the program P. The graph contains only generating sets, which are minimal sets which lead to inference of new positive information in a given point of a computation. Algorithm can also benefit from the fact, that accepting a set of default assumptions into a partial interpretation automatically rules out many other generating sets from the possible considering, whether they can be part of defaults of any stable model, or not.

We conclude this section with two examples illustrating the relation between the colorings of the blocking graph and the set of stable models of some normal logic program.

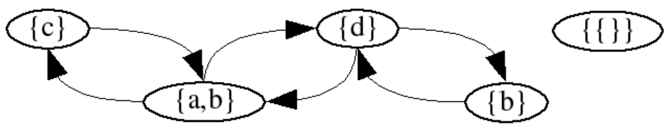
Example 3 *Let P be the program from the example 2 from the Sect. 3.*

The blocking graph of this program is depicted in Fig. 1.

The program P has three stable models characterized by corresponding colorings of the blocking graph[2]*:*

[2]We give only set of $blue$ nodes. All non-$blue$ nodes are indeed red.

Figure 1: Blocking graph for the example 3.



coloring	***stable model***
$blue = \{\{c\}, \{d\}, \emptyset\}$	$M_1 = \{a, b, e, f\}$
$blue = \{\{a, b\}, \{b\}, \emptyset\}$	$M_2 = \{d, c, f\}$
$blue = \{\{c\}, \{b\}, \emptyset\}$	$M_3 = \{a, d, e, f\}$

There's one valuable observation following from the example 3. Note that the node $\{\emptyset\}$ was always colored to blue. This was because there was no generating set blocking the set $\emptyset$, thus it could always be accepted in any stable model. This observation can be generalized in order to further optimize the search for valid colorings of blocking graph.

Claim 1 *Let $\mathcal{G}^P$ be a blocking graph of some normal logic program P. Nodes without any incoming edges (i.e. the set of nodes $\mathcal{V}_{blue} = \{V \in \mathcal{V} | \nexists W \in \mathcal{V} : (W, V) \in \mathcal{E}\}$) must be colored to $blue$ color in all valid colorings satisfying conditions 1 and 2 of the theorem 2.*

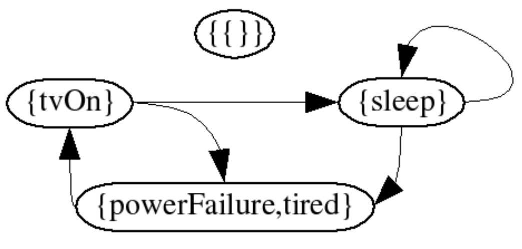
Example 4 *The following example was inspired by examples used in [J. J. Alferes, 2000].*
Let a program P be:

$sleep \leftarrow nightTime, tired.$	$tvOn \leftarrow nightTime, not\, powerFailure, not\, tired.$
$sleep \leftarrow not\, tvOn.$	$tired \leftarrow nightTime, not\, tvOn.$
$watchTv \leftarrow tvOn.$	$tired \leftarrow not\, sleep.$
$nightTime \leftarrow .$	

Generating sets of this program are as follows:
$GSets^P(sleep) = \{\{tvOn\}, \{sleep\}\}$
$GSets^P(watchTv) = \{\{powerFailure, tired\}\}$
$GSets^P(tvOn) = \{\{powerFailure, tired\}\}$
$GSets^P(tired) = \{\{tvOn\}, \{sleep\}\}$
$GSets^P(nightTime) = \{\emptyset\}$

There's just one valid coloring of the blocking graph of the program P (depicted in Fig. 2), therefore the program P has just one stable model $M = \{nightTime, tired, sleep\}$ generated by the generating set $\{tvOn\}$. In this example we can see that the node $\{sleep\}$ is blocking itself, thus it is forced to be of red color and because there's no other incoming edge to it than the one from the node $\{tvOn\}$, this node is forced to be

Figure 2: Blocking graph for the example 4.



blue, thus forcing the node $\{powerFailure, tired\}$ *to be also* red*. Finally the node* $\{\emptyset\}$ *is forced to be blue because it has no incoming edges.*

Also from this example, there follows a general observation embodied in the following claim:

Claim 2 *Let* $\mathcal{G}^P$ *be a blocking graph of some normal logic program* P*. All nodes with an edge leading to themselves (i.e.* $\mathcal{V}_{red} = \{V \in \mathcal{V} | \exists (V, V) \in \mathcal{E}\}$*) are colored to* red *color in all valid colorings according to conditions of the theorem 2.*

5 Prototype

Based on the result stated and proved in the theorem 2, we developed a prototype of an algorithm for computation of all stable models of normal logic programs. The aim of this section is to briefly describe it and sketch future directions in the development of it. In this paper, we want to show and proof new base of possible algorithm for stable model semantics, what we already did in the previous sections. The algorithm proposed in this section served us only as a proof of concept and prototype, therefore we do not want to comprehensively compare the proposed algorithm with today's most recognized systems like *smodels*, or *DLV*.

Our prototype algorithm consists of two main steps:

1. Preprocessing step in which all generating sets of atoms of a given normal logic program P are computed and blocking graph is constructed.

2. Search for all colorings of the blocking graph of a normal logic program P satisfying conditions of the theorem 2.

We do not show here all the details of the preprocessing step. In our prototype, we used an algorithm which, given an atom a and a rule $r \in P$ recursively computed all generating sets for all atoms from $body^+(r)$, unified them with atoms from $body^-(r)$

and thus created some generating set for an atom a. If a rule r was a fact (i.e. $body(r) = \emptyset$), the generating set was $\emptyset$. This process was performed as a backtrack on rules of a given program P. Surprisingly, the algorithm was not as slow as we expected. The details are given later in description of tests.

In this section we dedicate more attention to the second step of the algorithm which, given a blocking graph $\mathcal{G}^P$ of a normal logic program P, computes all the colorings satisfying conditions 1, 2 and 3 of the theorem 2. However, we want to stress, that this algorithm was just a prototype, and any other algorithm for coloring the blocking graph could be employed. No optimizations were employed.

The main algorithm, implemented as a procedure *findColoring*, performs a backtrack search for all valid colorings of a blocking graph $\mathcal{G}^P$.

```
procedure findColoring()
{Performs a main backtrack search for all colorings satisfying conditions of theorem 2.}
INVARIANT CONDITION: {
- for each node $V \in \mathcal{V}$, such that $color(V) = blue$ holds, that all nodes with incoming, or out-coming edges to/from $V$ are either $red$, or $uncolored$.
- for each node $V \in \mathcal{V}$, such that $color(V) = red$ holds, that there is at least one node with an edge incoming to $V$, which is either $blue$, or $uncolored$.}
if $\forall W \in \mathcal{V} : color(W) \neq uncolored$ then
  $StableModel = \{a | \exists GSet_{blue} \in GSets^P(a) \wedge color(GSet_{blue}) = blue\}$
  print $StableModel$
  return
end if
choose node $V \in \mathcal{V}$ such, that $color(V) = uncolored$;
{Try to color $V$ to $blue$ color:}
$color(V) = blue$
if checkBlueNodeAndColorNeighbors$(V, ColoredNodes)$ then
  findColoring()
end if
{Now tidy up all nodes colored during the attempt to color $V$ to blue.}
for all $W \in ColoredNodes$ do
  $color(W) = uncolored$
end for
{Try to color $V$ to $red$ color:}
$color(V) = red$
if $checkRedNodeColoring(V)$ then
  findColoring()
end if
$color(V) = uncolored$
end procedure.
```

At the beginning of the procedure *findColoring*, the invariant condition is listed.

Without a formal proof we claim, that if this invariant condition is satisfied always when the algorithm reaches its position, any coloring, which includes all the nodes of the blocking graph, is valid according to conditions in theorem 2.

The algorithm itself is simple. In any valid coloring, each node must be colored either to $blue$, or red color. Therefore the algorithm arbitrarily chooses uncolored node from $\mathcal{G}^P$, tries at first to color it to $blue$ and subsequently to red color. If the invariant condition is still satisfied after this coloring step, the algorithm recursively calls itself until all nodes of the blocking graph are colored, or it is not possible to color some node either to $blue$, or red, what causes a backtrack.

When we are coloring a chosen node $V \in \mathcal{V}$ to red color, we only have to check, whether for all red nodes of the blocking graph $\mathcal{G}^P$ it still has at least one incoming edge from some either $blue$, or $uncolored$ node. Because relevant nodes are only those, to which there is an edge from the currently colored node, we only have to check these. If this condition is not satisfied in some red node $V_{red} \in \mathcal{V}$, the condition 2 from the theorem 2 won't be satisfied in any coloring. The code chunk implementing the try to color a given node to red color is the function *checkRedNodeColoring*.

```
function checkRedNodeColoring($V$)
{Returns true if it is possible to color the node $V$ without a violation of the algorithm invariant condition.}
$InNeighbors = \{W \in \mathcal{V} : (W, V) \in \mathcal{E}\}$
if for all $W \in InNeighbors$ holds, that $color(W) = red$ then
   return $false$
end if
return $true$
end function.
```

Checking whether a chosen node can be colored to $blue$ color and the invariant condition still satisfied, is a bit more complicated issue because coloring a node $V \in \mathcal{V}$ to $blue$ color , enforces coloring all nodes with an edge connecting them with V to red color. For all these enforced red nodes, the function *checkRedNodeColoring* has to be performed. Because these red nodes can be connected with each other, the safe strategy is to first color all of them to red and then check whether their coloring is still correct w.r.t. invariant condition of the algorithm. During the backtrack from either a successful, or unsuccessful coloring to $blue$, the algorithm has to uncolor all the red colored nodes enforced to red during the attempt to color some node to $blue$. Algorithm for coloring a node to $blue$ and enforcing its neighbors to red is implemented as a function *checkBlueNodeAndColorNeighbors*.

```
function checkBlueNodeAndColorNeighbors($V, ColoredNodes$)
{Returns true if it is possible to color the node $V$ to $blue$ without a violation of the algorithm invariant condition. In $ColoredNodes$ returns all nodes which were forced to be $red$.}
$Neighbors = \{W \in \mathcal{V} | (V, W) \in \mathcal{E} \vee (W, V) \in \mathcal{E}\}$
```

```
$ColoredNodes = \{W \in Neighbors | color(W) = uncolored\}$
**for all** $W \in ColoredNodes$ **do**
  $color(W) = red$
**end for**
$goodColoringFlag = true$
**for all** $W \in Neighbors$ **do**
  $goodColoringFlag = checkRedNodeColoring(W)$ && $goodColoringFlag$
**end for**
return $goodColoringFlag$
**end function.**
```

As we already mentioned, our prototype algorithm works on already grounded normal logic programs. For grounding, we used the *lparse* system developed by Syrjänen [Syrjänen, 1998], which was also employed by authors of *smodels* system. The prototype of the proposed algorithm was implemented in *XSB Prolog 2.6* system in Linux operating system. Results of experiments with simple test cases are very promising. The grounded program for the block world problem had 132 atoms and 629 rules. The *smodels* system computed all the 6 solutions in about 0.6 sec., while our prototype algorithm needed 95 seconds in average to print all stable models. The preprocessing step took in average 5.5 seconds. To complete the picture, the blocking graph consisted of 252 nodes and 2120 edges.

6 Related work and discussion results

At the end of this paper we would like discuss results of our work in the broader context of works we later found to be simillar, or even almost identical to our approach. On this place we would like to add, that the work described above was done independently without any information on simillar results and only later we recognized that simillar approaches were proposed and described much earlier.

The most relevant and related work to our approach was done by Dimopoulos and Torres published in mid-nineties. In [Dimopoulos and Torres, 1996] authors show the stable model semantics can be expressed in terms on notions emerging from graph theory, namely that *stable models*, *partial stable models*, and the well founded semantics correspond respectivelly to kernels, semikernels and the initial acyclic part of an associated graph. Particularly for stable models, they show that given *negative logic program* (logic program containing only default literals in bodies of rules) set of stable models corresponds to kernels of the *rule dependency graph* of this program. In fact, notion of *blocking graph* corresponds to the notion of *dependency graph* of a given negative logic program[3] *conditions* on coloring of a blocking graph in our work (see Theorem 2) obviously correspond with conditions for kernel of a dependency graph (where *blue* nodes form the kernel of dependency graph).

[3]Note that if a logic program P° is *canonical logic program* then it is also a *negative logic program*.

Having said this we can say that from this point of view, our work describes same results as [Dimopoulos and Torres, 1996], although our approach was more on *engineering* side (we described concrete algorithms, while the work by Dimopoulos and Torres is more on theoretical side). Our approach was also driven more by implementation of ASP solver, than by previous theoretical study of a problem.

Other relevant study was done by Torsten Schaub et. al. (e.g. [Konczak et al., 2003]) on exploring possibilities of using graph coloring of dependency graphs of logic programs for computing stable models. However this approach is simillar to our work only in a using a proprietary graph coloring as an algorithm for computation of stable models. For example, authors of this approach propose coloring a kind of dependency graph with two types of edges while algorithms proposed in this work are working above simple dependency (blocking) graph constructed from logic program transformed into a canonical program.

7 Conclusion

In this paper we introduced a simple theory, on which the simple algorithm for computation of stable model semantics of normal logic programs was built. There are two main results of this paper. The first is an introduction of the notion of *generating sets* together with a notion of *canonical logic program*. We believe that such canonical logic programs could be used as a form into which normal logic programs could be compiled and stored, because their simple form allows easy manipulations, while preserving all semantical properties of the original normal logic program.

The second major result of this paper is a theorem 2, according to which, the problem of search for stable models of normal logic programs can be converted to a problem of search for all valid colorings of a graph constructed using already mentioned generating sets. To prove this concept, simple algorithm prototype was implemented.

References

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

[de Lima Nogueira, 2003] de Lima Nogueira, M. (2003). *Building Knowledge Systems in A-Prolog*. PhD thesis, Computer Science Department, The University of Texas, El Paso.

[Dimopoulos and Torres, 1996] Dimopoulos, Y. and Torres, A. (1996). Graph theoretical structures in logic programs and default theories. *Theor. Comput. Sci.*, 170(1-2):209–244.

[Eiter, 2000] Eiter, Faber, G. e. a. (2000). The dlv system. In Minker, J., editor, *Preprints of Workshop on Logic-Based AI.*

[I. Niemelä, 1997] I. Niemelä, P. S. (1997). Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. F. and Nerode, A., editors, *Proc. 4th international conference on Logic programming and non-monotonic reasoning*, pages 420–429. Springer.

[J. A. Leite and Pereira, 2000] J. A. Leite, J. J. A. and Pereira, L. M. (2000). Multi-dimensional dynamic logic programming. In F. Sadri, K. S., editor, *Proceedings of the CL-2000 Workshop on Computational Logic in Multi-Agent Systems*, pages 17–26, London, England. CLIMA'00.

[J. A. Leite, 2001] J. A. Leite, J. J. Alferes, L. M. P. (2001). Multi-dimensional logic programming. Technical report, Dept. de Informatica, Faculdade de Ciencias e Tecnologia, Universidade Nova de Lisboa.

[J. Dix, 2002] J. Dix, J. A. Leite, K. S., editor (2002). *Proceedings of the 3rd International Workshop on Computational Logic in Multi-Agent Systems*, number 93 in Datalogiske Skrifter (Writings on Computer Science), Roskilde University, Denmark. CLIMA'02.

[J. J. Alferes, 2001] J. J. Alferes, P. Dell'Acqua, E. J. A. L. L. M. P. F. R. (2001). A logic based approach to multi-agent systems. *The Association for Logic Programming Newsletter*. Invited paper.

[J. J. Alferes, 2000] J. J. Alferes, J. A. Leite, L. M. P. H. P. T. C. P. (2000). Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*.

[Konczak et al., 2003] Konczak, K., Linke, T., and Schaub, T. (2003). Graphs and colorings for answer set programming: Abridged report. In Vos, M. D. and Provetti, A., editors, *Proceedings of the Second International Workshop on Answer Set Programming (ASP'03)*, volume 78, pages 137–150. CEUR Workshop Proceedings. `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-78/`.

[Leite, 2003] Leite, J. A. (2003). *Evolving Knowledge Bases*, volume 81 of *Frontiers of Artificial Intelligence and Applications*. IOS Press.

[M. Gelfond, 1988] M. Gelfond, V. L. (1988). The stable model semantics for logic programming. pages 1070–1080. 5th International Conference on Logic Programming, MIT Press.

[M. van Emden, 1976] M. van Emden, R. K. (1976). The semantics of predicate logic as a programming language. *Journal of ACM*, 4(23):733–742.

[P. Cholewinski, 1996] P. Cholewinski, V. W. Marek, M. T. (1996). Default reasoning system DeReS. In *Proceedings of KR-96*. Morgan Kaufmann.

[Simons, 2000] Simons, P. (2000). *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Helsinki, Finland.

[Syrjänen, 1998] Syrjänen, T. (1998). Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Digital Systems Laboratory, Helsinki University of Technology.

[Vladimir Lifschitz,] Vladimir Lifschitz, David Pearce, A. V. Strongly equivalent logic programs.

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/~wjamroga/techreports/

The IfI Review Board