Enhancing Disjunctive Logic Programming Systems by SAT Checkers

Christoph Koch [a] Nicola Leone [b] Gerald Pfeifer [a]

[a] *Institut für Informationssysteme, Technische Universität Wien, A-1040 Wien, Austria, {koch,pfeifer}@dbai.tuwien.ac.at*

[b] *Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy, leone@unical.it*

Abstract

Disjunctive logic programming (DLP) with stable model semantics is a powerful nonmonotonic formalism for knowledge representation and reasoning. Reasoning with DLP is harder than with normal ($\vee$-free) logic programs, because *stable model checking* – deciding whether a given model is a stable model of a propositional DLP program – is co-NP-complete, while it is polynomial for normal logic programs.

This paper proposes a new transformation $\Gamma_M(\mathcal{P})$, which reduces stable model checking to UNSAT – i.e., to deciding whether a given CNF formula is unsatisfiable. The stability of a model M of a program $\mathcal{P}$ thus can be verified by calling a Satisfiability Checker on the CNF formula $\Gamma_M(\mathcal{P})$. The transformation is parsimonious (i.e., no new symbol is added), and efficiently computable, as it runs in logarithmic space (and therefore in polynomial time). Moreover, the size of the generated CNF formula never exceeds the size of the input (and is usually much smaller). We complement this transformation with modular evaluation results, which allow for efficient handling of large real-world reasoning problems.

The proposed approach to stable model checking has been implemented in `DLV`– a state-of-the-art implementation of DLP. A number of experiments and benchmarks have been run using SATZ as Satisfiability checker. The results of the experiments are very positive and confirm the usefulness of our techniques.

Keywords: Disjunctive Logic Programming, Nonmonotonic Reasoning, Head-Cycle-Free Programs, Answer Set Programs, Stable Model Checking.

⋆ A preliminary version of this paper has been presented at IJCAI'99 [1]. This work

1 Introduction

Disjunctive logic programming (DLP) with the stable model semantics is a powerful nonmonotonic formalism for knowledge representation and common sense reasoning [2–5]. DLP has a very high expressive power [6] – it allows to express all problems in the complexity class Σ_2^P (i.e., NP^{NP}). It is well known that many important nonmonotonic reasoning and AI problems are Σ_2^P-complete [7–12], and that nonmonotonic reasoning systems using the stable model semantics are currently among the most efficient declarative systems that can deal with such problems. Moreover, many complex problems can be represented in a simple and easy-to-understand fashion [13,14] using DLP with the stable model semantics.

Roughly, a DLP program is a set of disjunctive rules, i.e., clauses of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \text{not } b_{k+1} \wedge \cdots \wedge \text{not } b_m$$

with a possibly empty body (i.e., $m \geq 0$). The intuitive reading of such a rule is "If all $b_1, \ldots, b_k$ are true and all $b_{k+1}, \ldots, b_m$ are false, then at least one atom in $a_1, \ldots, a_n$ must be true." Atoms $a_1, \ldots, a_n, b_1, \ldots, b_m$ may contain variables but no function terms. A clause with an empty head (i.e., $n = 0$) and a nonempty body is called an integrity constraint and is read as "At least one atom in $b_1, \ldots, b_k$ must be false or at least one atom in $b_{k+1}, \ldots, b_m$ must be true." (i.e., the body of the constraint must be false). The intended models of a DLP program (i.e., the semantics of the program) are subset-minimal models which are "grounded" in a precise sense. They are called *stable models* or *answer sets* [2,5].

The DLP language allows for a fully declarative programming style, which is called *answer set programming (ASP)*. The idea of answer set programming is to represent a given computational problem by a DLP program whose stable models (answer sets) correspond to solutions, and then use a DLP system to find such a solution [15].

Example 1.1 Consider 3-Colorability, a well-known NP-complete problem from graph theory, which closely relates to the problem of coloring a map with a minimal number of colors such that no two neighboring countries are assigned the same color.

Given a graph, the problem is to decide whether there exists an assignment of one of three colors (say, red, green, or blue) to each node such that adjacent nodes always have different colors.

---

was supported by the European Commission under project INFOMIX, project no. IST-2001-33570, and under project ICONS, project no. IST-2001-32429.

Suppose that the graph is represented by a set of facts F using a unary predicate $node(X)$ and a binary predicate $arc(X, Y)$. Then, the following DLP program (in combination with F) computes all 3-Colorings (as stable models) of that graph.

$$r_1 : \quad color(X, red) \vee color(X, green) \vee color(X, blue) \leftarrow node(X)$$
$$r_2 : \quad \leftarrow color(X_1, C) \wedge color(X_2, C) \wedge arc(X_1, X_2)$$

Rule r_1 expresses that each node must either be colored red, green, or blue; [1] due to minimality of the stable models, a node cannot be assigned more than one color. The subsequent integrity constraint checks that no pair of adjacent nodes (connected by an arc) is assigned the same color.

Thus, there is a one-to-one correspondence between the solutions of the 3-Coloring problem and the stable models of $F \cup \{r_1, r_2\}$. The graph is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some stable model. □

Answer set programming has recently found a number of promising applications: Several tasks in information integration require complex reasoning capabilities, which are explored in the INFOMIX project (funded by the European Commission, project IST-2002-33570). Another EC-funded project, ICONS (IST-2001-32429), employs a DLP system as intelligent query engine for knowledge management. The Polish company Rodan Systems S.A. uses a DLP system in a tool for the detection of price manipulations and unauthorized uses of confidential information, which is used by the Polish securities and exchange commission. ASP solvers are used also for decision support in the Space Shuttle [16], for product and software configuration tasks [17,18], for model checking applications [19], and more.

The high expressive power - a key reason for the success of disjunctive logic programming - is paid for by high computational complexity. Indeed, as for the other main nonmonotonic formalisms like Default Logic or Circumscription, reasoning with DLP (under stable model semantics) is very hard. The high complexity of DLP reasoning stems from two sources: On the one hand the exponential number of possible models (model candidates), and on the other hand from the hardness of *stable model checking* – deciding whether a given model is a stable model of a propositional DLP program – which is co-NP-complete. The hardness of this problem has discouraged the implementation of DLP engines.

Indeed, at the time being only few systems – namely `DLV` [13] and GnT/Smodels

[1] Variable names start with an upper case letter and constants start with a lower case letter.

[20] – are available which fully support (function-free) DLP with the stable model semantics.

In this paper, we study the stable model checking problem to provide efficient methods for its implementation. We come up with a new transformation which reduces stable model checking to Unsatisfiability (UNSAT) – that is, to deciding whether a given CNF formula is unsatisfiable. This is the complement of Satisfiability (SAT), a problem for which very efficient systems have been developed in AI during the last decade.

Besides providing an elegant characterization of stable models which sheds new light on their intrinsic nature, the proposed transformation has a strong practical impact. Indeed, by using this transformation, the huge amount of work done in AI on the design and implementation of efficient algorithms for checking Satisfiability can be profitably used for the implementation of DLP engines supporting stable model semantics. In a sense this transformation thus opens "new frontiers" in the implementation of Disjunctive Logic Programming.

In addition we derive new modularity properties of stable models which permit the use of modular evaluation techniques for stable model checking. Those prove extremely useful in the light of co-NP-completeness results for that task.

We have implemented the proposed technique in the DLP system `DLV`, and performed a number of experiments and benchmarks.

In sum, the main contributions of this paper are the following:

- We define a new transformation from stable model checking for general DLP with negation to UNSAT of propositional CNF formulas. We prove the correctness of the transformation. The transformation is parsimonious (i.e., it does not add any new symbol) and efficiently computable, since it runs in LOGSPACE (and therefore in polynomial time). Moreover, the size of the generated CNF formula never exceeds the size of the input (and is usually much smaller because many rules are simplified or removed).
- We present some new results based on the application of modular evaluation techniques to our approach, which allow for further efficiency improvements by splitting the process of stable model checking. Instead of checking the stability of the model at once on the entire program, the model is split into components that are independently checked for stability on the respective subprograms.
- We realize our approach in the DLP system `DLV` – a state-of-the-art implementation of disjunctive logic programming – by using the efficient Davis-Putnam procedure SATZ [21] as the Satisfiability checker to solve UNSAT.

- We compare our approach with the GnT system and with the original stable-model checking method of DLV. We highlight the main differences of the methods, and we carry out an experimental activity over a number of $\Sigma_2^{\mathcal{P}}$-complete problems. The results of the experiments witness the efficiency of our approach to stable model checking - DLV with our new strategy outperforms the competing systems.

It is worth noting that, since stable model checking of DLP programs generalizes minimal model checking for Horn CNF formulas, our results can be employed also for reasoning with minimal models or circumscription over these formulas.

The DLV system, which implements the results described in this paper, can be downloaded from http://www.dlvsystem.com/. From the same Web page, one can also retrieve the benchmark problems that we used in our experiments.

The paper is organized as follows. Section 2 first provides definitions of syntax and semantics of DLP with the stable model semantics. After that we give further examples of using DLP for a couple of knowledge representation problems. In Section 3, we outline previously known results on stable model checking. Section 4 presents our main result, the transformation from stable model checking to UNSAT. Section 5 applies modularity properties of DLP programs in the context of our transformation. Section 6 briefly describes our implementation, compares our approach to other stable model checking methods, and presents benchmark problems used and results obtained in our experiments. Finally, Section 7 addresses a number of further related works and draws our conclusions.

2 Disjunctive Logic Programming with Stable Model Semantics

In this section, we provide an overview of (function-free) disjunctive logic programming with stable model semantics [2,22,23,14].

2.1 Syntax

A variable or constant is a *term*. An *atom* is of the form $a(t_1, ..., t_n)$, where a is a *predicate* of arity $n \geq 0$ and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* not p, where p is an atom.

A *(disjunctive) rule* r is a clause of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \text{not } b_{k+1} \wedge \cdots \wedge \text{not } b_m \qquad n \geq 1,\ m \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and r needs to be *safe*, i.e. each variable occurring in r must appear in one of the positive body literals $b_1, \cdots, b_k$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1 \wedge \ldots \wedge b_k \wedge \text{not } b_{k+1} \wedge \ldots \wedge \text{not } b_m$ is the *body* of r. We denote by $H(r)$ the set $\{a_1, \ldots, a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, \ldots, b_k, \text{not } b_{k+1}, \ldots, \text{not } b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$, i.e. $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. *Constraints* are special rules with an empty head ($n = 0$), written as

$$\leftarrow b_1 \wedge \cdots \wedge b_k \wedge \text{not } b_{k+1} \wedge \cdots \wedge \text{not } b_m \qquad m \geq 1$$

which we define as syntactic sugaring equivalent to a rule $a \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \text{not } b_{k+1} \wedge \cdots \wedge \text{not } b_m \wedge \text{not } a$ for some new nullary (i.e., propositional) atom a. A *(disjunctive) program* (also called DLP program) is a set of rules (and constraints). A not-free (resp., $\vee$-free) program is called *positive* (resp., *normal*). An atom, a literal, a rule, a constraint, or a program, resp., is *ground* if no variables appear in it. A finite ground program is also called a *propositional* program.

2.2 Stable Model Semantics

Now let $\mathcal{P}$ be a program. The *Herbrand universe* $U_{\mathcal{P}}$ (in the function-free case) of $\mathcal{P}$ is the set of constants that appear in the program [2]. The *Herbrand base* $B_{\mathcal{P}}$ of $\mathcal{P}$ is the set of all possible ground atoms that can be constructed from the predicates appearing in the rules of $\mathcal{P}$ and the terms occurring in $U_{\mathcal{P}}$. Given a rule r occurring in $\mathcal{P}$, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a mapping from the variables occurring in r to the terms in $U_{\mathcal{P}}$. We denote by *ground*$(\mathcal{P})$ the set of all the ground instances of the rules occurring in $\mathcal{P}$.

A (total) *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_{\mathcal{P}}$. A ground positive literal A is *true* (resp., *false*) with respect to I if $A \in I$ (resp., $A \notin I$). A ground negative literal not A is *true* w.r.t. I if A is false w.r.t. I; otherwise not A is false w.r.t. I.

Let r be a rule in *ground*$(\mathcal{P})$. The head of r is *true* with respect to I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

[2] If no constants appear in the program, one is added arbitrarily.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in ground(\mathcal{P})$ is true (and the body of each ground constraint is false) w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$.

The first proposal for assigning a semantics to a disjunctive logic program appears in [24], which presents a model-theoretic semantics for positive programs. According to [24], the semantics of a program $\mathcal{P}$ is described by the set $\mathrm{MM}(\mathcal{P})$ of the minimal models for $\mathcal{P}$. Observe that every positive program $\mathcal{P}$ admits at least one minimal model, that is, for every positive program $\mathcal{P}$, $\mathrm{MM}(\mathcal{P}) \neq \emptyset$ holds.

Example 2.1 [3] For the positive program $P_1 = \{a \vee b \leftarrow\}$, the interpretations $\{a\}$ and $\{b\}$ are its minimal models ($\mathrm{MM}(\mathcal{P}) = \{\ \{a\},\ \{b\}\ \}$).

For the program $P_2 = \{a \vee b \leftarrow;\ b \leftarrow a;\ a \leftarrow b\}$, $\{a, b\}$ is the only minimal model. □

As far as general programs (that is, programs where negation may appear in the bodies) are concerned, a number of semantics have been proposed [25,2,24,26,3,27–29] (see [30,31] for comprehensive surveys). A generally acknowledged semantics for DLP programs is the extension of the stable model semantics [32,33] to take into account disjunction [2,3]. Given a program $\mathcal{P}$ and an interpretation I, the *Gelfond-Lifschitz (GL) transformation* of $\mathcal{P}$ w.r.t. I, denoted $\mathcal{P}^I$, is the set of positive rules defined as follows:

$$\begin{aligned}\mathcal{P}^I = \{\ & a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_k \mid \\ & a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \text{not } b_{k+1} \wedge \cdots \wedge \text{not } b_m \\ & \text{is in } ground(\mathcal{P}) \text{ and } b_i \notin I, \text{for all } k+1 \leq i \leq m\}\end{aligned}$$

Clearly, if $\mathcal{P}$ is positive, then $\mathcal{P}^I$ coincides with $ground(\mathcal{P})$. It turns out that for positive programs, minimal and stable models coincide.

Definition 2.2 [3,2] Let I be an interpretation for a program $\mathcal{P}$. I is a *(disjunctive) stable model* for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model of the positive program $\mathcal{P}^I$). □

Example 2.3 Let $P = \{a \vee b \leftarrow c;\quad b \leftarrow \text{not } a \wedge \text{not } c;\quad a \vee c \leftarrow \text{not } b\}$ and $I = \{b\}$. Then, $P^I = \{a \vee b \leftarrow c;\quad b \leftarrow\}$. It is easy to verify that I is

[3] For simplicity, we often use propositional examples, in which the programs coincide with their ground instantiations, throughout most of this paper. However, all results and algorithms apply equally to the general case of (function-free) disjunctive programs with variables.

a minimal model for P^I. Thus, I is a stable model for P. □

2.3 Knowledge Representation in DLP

Next we give two examples of how to use DLP for solving complicated reasoning problems, notably the Hamiltonian Path problem and a case of Network Diagnosis.

Example 2.4 Hamiltonian Path is a classical NP-complete problem from the area of graph theory:

> Given an undirected graph $G = (V, E)$, where V is the set of vertices of G and E is the set of arcs, and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?

Suppose that the graph G is specified by using two predicates $node(X)$ and $arc(X, Y)$ [4], and the starting node is specified by the unary predicate *start* which contains only a single tuple. Then, the following program $\mathcal{P}_{hp}$ solves the Hamiltonian Path problem.

$$
\begin{aligned}
inPath(X,Y) \vee outPath(X,Y) &\leftarrow reached(X) \wedge arc(X,Y); \\
&\leftarrow node(X) \wedge \text{not } reached(X); \\
reached(X) &\leftarrow start(X); \\
reached(X) &\leftarrow inPath(Y,X); \\
&\leftarrow inPath(X,Y) \wedge \\
&\qquad inPath(X,Y_1) \wedge Y \neq Y_1; \\
&\leftarrow inPath(X,Y) \wedge \\
&\qquad inPath(X_1,Y) \wedge X \neq X_1
\end{aligned}
$$

The first rule guesses a subset $S \subseteq E$ of all given arcs to be in the path, while the rest of the program checks whether that subset S constitutes a Hamiltonian Path.

The first constraint enforces that all nodes V in the graph are reached from the starting node in the subgraph induced by S and also ensures that this subgraph is connected. The two rules after this constraint define reachability from the starting node with respect to the set of arcs S.

The final two constraints check whether the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian Path must satisfy:

[4] Predicate *arc* is symmetric, since undirected arcs are bidirectional.

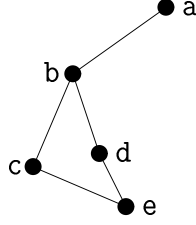


Fig. 1. A computer network

There must not be two arcs starting at the same node, nor may there be two arcs ending in the same node.

If the input graph and the starting node are specified by a set F of facts (with predicates *node, arc*, and *start*), then there is a one-to-one correspondence between the solutions of the Hamiltonian Path problem and the stable models of $F \cup \mathcal{P}_{hp}$. The graph has an Hamiltonian Path if and only if $F \cup \mathcal{P}_{hp}$ has some stable model. □

Example 2.5 (Abduction) Consider the computer network depicted in Figure 1. We make the observation that, sitting at machine a, which is online, we cannot reach machine e. Which machines are offline?

This can be easily modeled as the program $\mathcal{P}_{net} = \langle Hyp_{net}, Obs_{net}, LP_{net} \rangle$, where the theory LP_{net} is

$$\begin{aligned} LP_{net} = \{ & reaches(X,X) \leftarrow node(X) \wedge \text{not } offline(X); \\ & reaches(X,Z) \leftarrow reaches(X,Y) \wedge connected(Y,Z) \wedge \text{not } offline(Z); \\ & connected(a,b);\ connected(b,c);\ connected(b,d);\ connected(c,e); \\ & connected(d,e);\ node(a);\ node(b);\ node(c);\ node(d);\ node(e) \} \end{aligned}$$

and the set of hypotheses (that each node *may* be offline) is encoded as

$$Hyp_{net} = \{\ offline(x) \vee not_offline(x) \mid x \text{ is a network node } \},$$

Observations are encoded as constraints

$$Obs_{net} = \{\ \leftarrow offline(a);\ \leftarrow offline(b);\ \leftarrow reaches(a,e)\ \}.$$

where positive observations x would be encoded as constraints $\leftarrow$ not x. The five stable models of $\mathcal{P}_{net}$ contain the explanations

$E_1 = \{offline(c),\ offline(d)\}$,

$E_2 = \{offline(e)\}$,

$E_3 = \{offline(c),\ offline(e)\}$,

$E_4 = \{offline(d),\ offline(e)\}$,

$E_5 = \{\mathit{offline}(c),\ \mathit{offline}(d),\ \mathit{offline}(e)\}$

as subsets, respectively.

The program shown in this example can be refined to *subset minimal diagnosis* (only resulting in explanations E_1 and E_2) using a slightly more involved encoding and *minimal cardinality* diagnosis (with the single "most probable" explanation E_2) using DLP with *weak constraints* [34]. □

3 Previous Results on Stable Model Checking

Next we review some known results on stable model checking, the problem of determining whether a model M of a disjunctive logic program $\mathcal{P}$ is stable. We refer to [23] for a more detailed discussion of these issues.

3.1 Stable Models and Unfounded Sets

In this section, we present a characterization of the stable models of disjunctive logic programs in terms of unfounded sets. This characterization will be used to prove the correctness of our reduction from stable model checking to UNSAT in the next section. The characterization is obtained by slight modifications of the results presented in [23]. In particular, by providing the notion of unfounded sets directly for total (2-valued) interpretations, we obtain a simpler characterization than in [23], where unfounded sets were defined w.r.t. partial (3-valued) interpretations.

Definition 3.1 (Definition 3.1 in [23]) Let I be a total interpretation for a program $\mathcal{P}$. A set $X \subseteq B_{\mathcal{P}}$ of ground atoms is an *unfounded set* for $\mathcal{P}$ w.r.t. I if, for each rule $r \in \mathit{ground}(\mathcal{P})$ such that $X \cap H(r) \neq \emptyset$, at least one of the following conditions holds:

C_1. $(B^+(r) \not\subseteq I) \vee (B^-(r) \cap I \neq \emptyset)$, that is, the body of r is false w.r.t. I.
C_2. $B^+(r) \cap X \neq \emptyset$, that is, some positive body literal belongs to X.
C_3. $(H(r) - X) \cap I \neq \emptyset$, that is, an atom in the head of r, distinct from the elements in X, is true w.r.t. I. □

Example 3.2 Let $\mathcal{P} = \{a \vee b \leftarrow\}$ and $I = \{a, b\}$. Due to Condition 3, both $\{a\}$ and $\{b\}$ are unfounded sets of I w.r.t. $\mathcal{P}$. □

Definition 3.3 An interpretation I for a program $\mathcal{P}$ is unfounded-free iff no non-empty subset of I is an unfounded set for $\mathcal{P}$ w.r.t. I. □

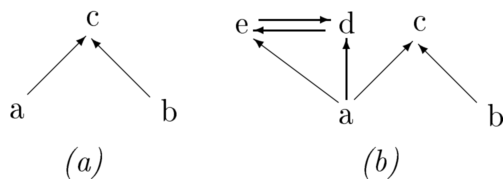


Fig. 2. *Graphs (a) $DG_{\mathcal{P}_1}$ and (b) $DG_{\mathcal{P}_2}$*

The unfounded-free condition singles out precisely the stable models.

Proposition 3.4 *(Theorem 4.6 in [23]) Let M be a model for a program $\mathcal{P}$. M is a stable model for $\mathcal{P}$ iff M is unfounded-free.*

Example 3.5 Let $\mathcal{P} = \{a \vee b \leftarrow\}$. $M_1 = \{a\}$ is a stable model of $\mathcal{P}$, since there is no non-empty subset of M_1 which is an unfounded set. As shown in Example 3.2, $M_2 = \{a, b\}$ is not unfounded-free, and therefore it is not a stable model. □

3.2 A Tractable Class: HCF Programs

In this section, we discuss the special case of DLP programs having the so-called head-cycle-free property. Informally, this property ensures that there is no recursion through disjunction, allowing for an efficient stable-model checking method.

With every program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (\mathcal{N}, E)$, called the *dependency graph* of $\mathcal{P}$ which has the following properties: (i) Each predicate of $\mathcal{P}$ is a node in $\mathcal{N}$. (ii) There is a directed arc in E from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that a and b are the predicates of a positive literal appearing in $B(r)$ and $H(r)$, respectively.

The graph $DG_{\mathcal{P}}$ singles out the dependencies of the head predicates of a rule on the positive predicates in the body of that rule. [5]

Example 3.6 Consider the program $\mathcal{P}_1$ consisting of the following rules:

$$a \vee b \leftarrow \qquad\qquad c \leftarrow a \qquad\qquad c \leftarrow b$$

The dependency graph $DG_{\mathcal{P}_1}$ of $\mathcal{P}_1$ is depicted in Figure 2a. (Note that, since the sample programs are propositional, the nodes of the sample graphs in Figure 2 are atoms, as atoms coincide with predicates in this case.)

[5] Note that negative literals do not cause an arc in $DG_{\mathcal{P}}$.

Consider now program $\mathcal{P}_2$, obtained by adding to $\mathcal{P}_1$ the rules

$$d \vee e \leftarrow a \qquad\qquad d \leftarrow e \qquad\qquad e \leftarrow d \wedge \text{not } b$$

The dependency graph $DG_{\mathcal{P}_2}$ is shown in Figure 2b. □

The dependency graphs allow us to single out head-cycle-free (HCF) programs [35,36]: A program $\mathcal{P}$ is *HCF* iff there is no clause r in $\mathcal{P}$ such that two predicates occurring in the head of r are in the same cycle of $DG_{\mathcal{P}}$.

Example 3.7 The dependency graphs given in Figure 2 reveal that program $\mathcal{P}_1$ of Example 3.6 is HCF and that program $\mathcal{P}_2$ is not HCF, as rule $d \vee e \leftarrow a$ contains two predicates in its head that belong to the same cycle of $DG_{\mathcal{P}_2}$. □

Definition 3.8 Let $\mathcal{P}$ be a program and I an interpretation. Then we define an operator $\mathcal{R}_{\mathcal{P},I}$ as follows:

$$\begin{aligned} \mathcal{R}_{\mathcal{P},I} : 2^{B_{\mathcal{P}}} &\rightarrow 2^{B_{\mathcal{P}}} \\ X &\mapsto \{a \in X \mid \forall r \in ground(\mathcal{P}) \text{ with } a \in H(r), \\ &\qquad B(r) \cap (\neg.I \cup X) \neq \emptyset \text{ or } (H(r) - \{a\}) \cap I \neq \emptyset\} \end{aligned}$$

where $\neg.I$ denotes the set of (ground) literals $\{\text{not } l \mid l \in I\}$. □

It is easy to see that the above operator $\mathcal{R}_{\mathcal{P},I}$ is monotonic. Moreover, given a set $X \subseteq B_{\mathcal{P}}$, it is obvious that the sequence $R_0 = X$, $R_n = \mathcal{R}_{\mathcal{P},I}(R_{n-1})$ decreases monotonically and converges finitely to a limit that we denote by $\mathcal{R}^{\omega}_{\mathcal{P},I}(X)$. As shown in [23], given a program $\mathcal{P}$ and a model M (in fact, the result holds for interpretations in general) of $\mathcal{P}$, all unfounded sets (of $\mathcal{P}$ w.r.t. M) contained in M are subsets of $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$.

Proposition 3.9 *Let $\mathcal{P}$ be a program*[6] *and M be a model for $\mathcal{P}$. Then, $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \emptyset$ implies that M is unfounded-free w.r.t. $\mathcal{P}$.*

In the case that a program $\mathcal{P}$ is head-cycle-free, a model M of $\mathcal{P}$ is unfounded-free if and *only if* $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \emptyset$.

Proposition 3.10 *(Theorem 6.9 in [23]) Let $\mathcal{P}$ be an HCF program and M a total interpretation for it. Then M is unfounded-free iff $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \emptyset$.*

It has been shown that for head-cycle-free programs Model Checking can be performed in polynomial time.

Corollary 3.11 *(Corollary 6.10 in [23]) Let $\mathcal{P}$ be a propositional HCF program and M be a model for $\mathcal{P}$. Recognizing whether M is a stable model is*

[6] This program does *not* need to be HCF.

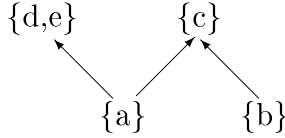


Fig. 3. *Graph* $\widehat{DG}_{\mathcal{P}_2}$

polynomial.

Example 3.12 Given the HCF program $\mathcal{P}$ containing the first five rules of program $\mathcal{P}_2$ of Example 3.6, i.e.

$$a \vee b \leftarrow \qquad c \leftarrow a \qquad c \leftarrow b$$
$$d \vee e \leftarrow a \qquad d \leftarrow e$$

and the model $M = \{a, c, d\}$. We have $\mathcal{R}_{\mathcal{P},M}(M) = \{c, d\}$ and $\mathcal{R}^2_{\mathcal{P},M}(M) = \mathcal{R}_{\mathcal{P},M}(\{c, d\}) = \emptyset = \mathcal{R}^{\omega}_{\mathcal{P},M}(M)$. Thus, M is a stable model of $\mathcal{P}$ as stated by Proposition 3.10. □

3.3 Modularity Properties

In this section, we summarize the main modular evaluation result of [23], which is also related to the work in [37,6].

Given a program $\mathcal{P}$, we denote by $\widehat{DG}_{\mathcal{P}}$ the graph of the strongly connected components of $DG_{\mathcal{P}}$ (i.e., the graph obtained by collapsing the strongly connected components of $DG_{\mathcal{P}}$). The subprogram $subp(Q, \mathcal{P})$ corresponding to a component Q of that graph is defined as the set of rules $r \in \mathcal{P}$ with $H(r) \cap Q \neq \emptyset$. By $\frac{I}{Q}$ we denote the set of all atoms of a component Q that are true w.r.t. an interpretation I, i.e., $I \cap Q$.

Basically, the unfounded-free property (and, consequently, stable-model checking) may be verified independently for each of the component subprograms of a program $\mathcal{P}$, and a model is unfounded-free iff it is unfounded-free w.r.t. each of the individual subprograms.

Proposition 3.13 *Let $\mathcal{P}$ be a program, and I an interpretation for $\mathcal{P}$. I is not unfounded-free iff there exists a node Q of $\widehat{DG}_{\mathcal{P}}$ such that $\frac{I}{Q}$ contains a non-empty unfounded set for $subp(Q, \mathcal{P})$ w.r.t. I.*

Example 3.14 Consider program $\mathcal{P}_2$ of Example 3.6 once again:

$$a \vee b \leftarrow \qquad c \leftarrow a \qquad c \leftarrow b$$
$$d \vee e \leftarrow a \qquad d \leftarrow e \qquad e \leftarrow d \wedge \text{not } b$$

```
Function unfounded-free(P: Program; M: SetOfAtoms): Boolean;
var X, Y, Q: SetOfAtoms;
begin
    compute DG^_P;
    for each node Q of DG^_P
        X := R^ω_{subp(Q,P),M}(M/Q)
        if X ≠ ∅ then
            if subp(Q,P) is HCF then return False;
            else   (* Computation of non-HCF components *)
                for each Y ⊆ X with Y ≠ ∅ do
                    if Y is an unfounded set for subp(Q,P)
                      w.r.t. M then return False;
                end for;
            end if;
        end if;
    end for;
    return True;
end;
```

Fig. 4. *The old model checking algorithm of* `DLV`

The component dependency graph $\widehat{DG}_{\mathcal{P}_2}$ is shown in Figure 3.

Interpretation $M = \{a, b, c, d, e\}$ is clearly a model of $\mathcal{P}_2$. For the node $Q = \{b\}$ of $\widehat{DG}_{\mathcal{P}_2}$ we have $\frac{M}{Q} = \{b\}$, and $subp(Q, \mathcal{P}_2) = \{a \vee b\}$. Since $subp(Q, \mathcal{P}_2)$ has unfounded sets $\{a\}$ and $\{b\}$ w.r.t. M, the model M is not stable. $\Box$

This property allows us to check the unfounded-freeness (and, therefore, the stability) of a model M in a modular way as described in the following section.

3.4 A Stable Model Checking Algorithm

The combination of the results of the previous subsections yields the modular model checking algorithm shown in Figure 4. This algorithm has been proposed in [23] and implemented in the `DLV` system (this method is referred to as `Old Checker` in Section 6).

Roughly, we first compute the component dependency graph $\widehat{DG}_{\mathcal{P}}$ of the program. Then, we visit the nodes of $\widehat{DG}_{\mathcal{P}}$ (that is, the components of $\mathcal{P}$) in sequence. For each node Q of $\widehat{DG}_{\mathcal{P}}$, we compute the fixpoint of the $\mathcal{R}$ operator for the nodes $\frac{M}{Q}$ (i.e., those nodes of the component that are true w.r.t. the model M) and the subprogram of Q. If this fixpoint is empty, the component is certain to be unfounded-free. Otherwise, we check whether this fixpoint (which is all we need to check inside the component Q) contains a

non-empty unfounded set for $subp(Q, \mathcal{P})$ with respect to M or not. If this is not the case for any of the nodes of the component dependency graph $\widehat{DG}_{\mathcal{P}}$ (i.e., no non-empty unfounded set has been found), then M is stable (since it is unfounded-free); otherwise, M is not stable.

As shown in [23], the algorithm of Figure 4 is correct:

Proposition 3.15 *Let $\mathcal{P}$ be a DLP program and M be a model for $\mathcal{P}$. Then, M is a stable model of $\mathcal{P}$ iff unfounded-free($\mathcal{P}$, M) - i.e., the algorithm of Figure 4 - returns true.*

This method has two main advantages:

(1) Since the subprograms are evaluated one-at-a-time, the evaluation method is selected according to the characteristics of the subprogram. This way, head-cycle free subprograms are always evaluated efficiently, and the inefficient part of the computation is limited only to the non-HCF subprograms.
(2) In order to check the stability (i.e., the unfounded-freeness) of a component Q, only the rules of the subprogram $subp(Q, \mathcal{P})$ are to be considered. All remaining rules of $\mathcal{P}$ are irrelevant for this purpose. Consequently, the stability check is applied on several small subprograms rather than on a big one, with evident advantages from the perspective of complexity [7].

4 From Stable Model Checking to UNSAT

In this section we present a reduction from stable model checking to UNSAT, the complement of Satisfiability (SAT). SAT is among the best researched problems in AI and several efficient algorithms and systems have been developed for solving SAT (and thus UNSAT as well).

Recall that a CNF formula over a set A of atomic propositions is a conjunction of the form $\phi = c_1 \wedge \cdots \wedge c_n$, where $c_1, \cdots, c_n$ are clauses over A. Without loss of generality, in this paper a clause $c = a_1 \vee \cdots \vee a_m \vee \neg b_1 \vee \cdots \vee \neg b_r$ will be written as $a_1 \vee \cdots \vee a_m \leftarrow b_1 \wedge \cdots \wedge b_r$; a CNF formula ϕ thus is a conjunction of such implications. (The usual form of writing CNFs can be immediately obtained by transforming each clause c_i from above to a disjunction of literals.)

A formula ϕ over A is *satisfiable* if there exists a truth assignment to the

[7] Recall that the stability check is co-NP-hard, and requires an amount of time of the order of $2^{|\mathcal{P}|}$ in the worst case. If $\mathcal{P}$ consists of two non-HCF subprograms A and B, then $|\mathcal{P}| = |A| + |B|$, and the modular evaluation technique requires an amount of time of the order of $2^{|A|} + 2^{|B|}$, which may be sensibly smaller than $2^{|A|+|B|}$.

Input: A ground DLP program $\mathcal{P}$ and a model M for $\mathcal{P}$.
Output: A propositional CNF formula $\Gamma_M(\mathcal{P})$ over M.
var $\mathcal{P}'$: DLP Program; S: Set of Clauses;
begin
1. Delete from $\mathcal{P}$ each rule whose body is false w.r.t. M;
2. Remove all negative literals from the (bodies of the) remaining rules;
3. Remove all false atoms (w.r.t. M) from the heads of the resulting rules;
4. $S := \emptyset$;
5. Let $\mathcal{P}'$ be the program resulting from steps 1–3;
6. **for** each rule $a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_m$ in $\mathcal{P}'$ **do**
7. $\quad S := S \cup \{\ b_1 \vee \cdots \vee b_m \leftarrow a_1 \wedge \cdots \wedge a_n\ \}$;
8. **end for**;
9. $\Gamma_M(\mathcal{P}) \ := \ \bigwedge_{c \in S} c \ \ \wedge \ \ (\ \bigvee_{x \in M} x\ )$;
10. **output** $\Gamma_M(\mathcal{P})$

end.

Fig. 5. *The transformation* $\Gamma_M(\mathcal{P})$

propositions of A which makes ϕ true; otherwise, ϕ is *unsatisfiable* (or *inconsistent*).

UNSAT is the following decision problem:

Given a CNF formula ϕ, is it true that ϕ is unsatisfiable?

Our reduction from stable model checking to UNSAT is implemented by the algorithm shown in Figure 5. In order to clarify the steps performed in the transformation, we will use the following running example.

Example 4.1 Let $\mathcal{P}$ be the program

$$a \vee b \vee c \leftarrow \qquad a \leftarrow b \qquad a \leftarrow c \qquad b \leftarrow a \wedge \text{not } c$$

Consider the model $M_1 = \{a, b\}$ of $\mathcal{P}$. In the first step of the algorithm shown in Figure 5, the rule $a \leftarrow c$ is deleted. In the second step, not c is removed from the body of the last rule of $\mathcal{P}$, while the third step removes c from the head of the first rule. Thus, after Step 3, the program becomes $\{a \vee b \leftarrow;$ $a \leftarrow b;\quad b \leftarrow a\}$. Steps 4 to 8 switch the bodies and the heads of the rules, yielding the set of clauses $S = \{\leftarrow a \wedge b; \quad b \leftarrow a; \quad a \leftarrow b\}$. Finally, Step 9 constructs the conjunction of the clauses in S plus the clause $a \vee b \leftarrow$. Therefore, the output of the algorithm is

$$\Gamma_{M_1}(\mathcal{P}) = (\leftarrow a \wedge b) \bigwedge (b \leftarrow a) \bigwedge (a \leftarrow b) \bigwedge (a \vee b \leftarrow).$$

Now consider the model $M_2 = \{a, c\}$. Here, the first three steps simplify $\mathcal{P}$ to $\{a \vee c \leftarrow; \quad a \leftarrow c.\}$. Steps 4 to 8 swap the heads and bodies of the rules

resulting in $\{\leftarrow a \wedge c; \quad c \leftarrow a\}$, and Step 9 adds $a \vee c \leftarrow$. So the outcome for M_2 is $\Gamma_{M_2}(\mathcal{P}) = (\leftarrow a \wedge c) \bigwedge (c \leftarrow a) \bigwedge (a \vee c \leftarrow)$. □

Theorem 4.2 *Given a model M for a ground DLP program $\mathcal{P}$, let $\Gamma_M(\mathcal{P})$ be the CNF formula computed by the algorithm of Figure 5 on input $\mathcal{P}$ and M. Then, M is a stable model for $\mathcal{P}$ if and only if $\Gamma_M(\mathcal{P})$ is unsatisfiable.*

In the remainder of this section we demonstrate Theorem 4.2 (i.e., we show the correctness of our $\Gamma_M(\mathcal{P})$ reduction). We proceed in an incremental way, dividing the $\Gamma_M(\mathcal{P})$ transformation into three steps, and showing the correctness of each of these. As mentioned before, we will use Example 4.1 as a running example to illustrate each of these steps.

Definition 4.3 Let $\mathcal{P}$ be a DLP program and M be a model for $\mathcal{P}$. Define the *simplified version* $\alpha_M(\mathcal{P})$ of $\mathcal{P}$ w.r.t. M as:

$$
\begin{aligned}
\alpha_M(\mathcal{P}) = \{\ & a_1 \vee \ldots \vee a_m \leftarrow b_1 \wedge \ldots \wedge b_n \mid r \in \mathit{ground}(\mathcal{P}) \text{ and} \\
& \{a_1, \ldots, a_m\} = H(r) \cap M \text{ and} \\
& \{b_1, \ldots, b_n\} = B^+(r) \text{ and} \\
& B(r) \text{ is true w.r.t. } M\ \}
\end{aligned}
$$

□

It is easy to see that $\alpha_M(\mathcal{P})$ coincides with the program $\mathcal{P}'$ obtained by steps 1–3 of Figure 5. Observe that every rule in $\alpha_M(\mathcal{P})$ has a non-empty head. Indeed, if, for some interpretation M and program $\mathcal{P}$, $\alpha_M(\mathcal{P})$ would contain a rule r with an empty head, then M would not be a model for $\mathcal{P}$, as the rule of $\mathcal{P}$ corresponding to r would have a true body and a false head. Moreover, the simplified program $\alpha_M(\mathcal{P})$ is positive (not-free) and it only contains atoms that are true w.r.t. M.

Next, we observe that $\alpha_M(\mathcal{P})$ is equivalent to $\mathcal{P}$ as far as the stability of M is concerned.

Lemma 4.4 *Let $\mathcal{P}$ be a DLP program and M be a model for $\mathcal{P}$. Then, M is a stable model for $\mathcal{P}$ if and only if it is a stable model for $\alpha_M(\mathcal{P})$.*

Proof. C_1, C_2, and C_3 refer to the three unfoundedness conditions from Definition 3.1. Therefore, we rewrite Definition 3.1 to define unfounded sets $X \subseteq M$ as those sets satisfying $\bigwedge_{r \in \mathcal{P}}((H(r) \cap X = \emptyset) \vee C_1 \vee C_2 \vee C_3)$.

Now we partition $\mathcal{P}$ into two sets, $\mathcal{P}'$ and $\mathcal{P} - \mathcal{P}'$, where $\mathcal{P}' = \{r \in \mathcal{P} | B(r)$ is false w.r.t. $M\}$. We claim that for all $X \subseteq M$, $\bigwedge_{r \in \mathcal{P}'}((H(r) \cap X = \emptyset) \vee C_1 \vee C_2 \vee C_3) \wedge \bigwedge_{r \in (\mathcal{P} - \mathcal{P}')}((H(r) \cap X = \emptyset) \vee C_1 \vee C_2 \vee C_3)$ equals $\bigwedge_{r \in \alpha_M(\mathcal{P})}((H(r) \cap X = \emptyset) \vee C_1 \vee C_2 \vee C_3)$.

Clearly, for every rule r in $\mathcal{P}'$, $C_1 = (B(r)$ is false w.r.t. $M)$ is true. Therefore, the conjunction over these rules which is shown above is true and can be eliminated. The corresponding rules do not exist in $\alpha_M(\mathcal{P})$.

For the remaining rules (i.e., those from $\mathcal{P} - \mathcal{P}'$), there exists a one-to-one relationship to the rules in $\alpha_M(\mathcal{P})$ that where derived from them. Here, for each pair $(r_1 \in (\mathcal{P} - \mathcal{P}'),\ r_2 \in \alpha_M(\mathcal{P}))$ of corresponding rules, each pair of conditions in the disjunctions associated to the rules has the same values. It is easy to see that $(H(r_1) \cap X = \emptyset) = (H(r_2) \cap X = \emptyset)$ since $H(r_1) \cap M = H(r_2)$ and $X \subseteq M$. We also know that both for $\mathcal{P} - \mathcal{P}'$ and for $\alpha_M(\mathcal{P})$, C_1 is always false. The value of C_2 is equal for all pairs (r_1, r_2) because $B^+(r_1) = B^+(r_2)$, and finally, regarding C_3, $H(r_1) \cap M = H(r_2) \cap M$.

But this was all we had to show to demonstrate that $X \subseteq M$ is an unfounded set of $\mathcal{P}$ if and only if it is an unfounded set of $\alpha_M(\mathcal{P})$. Lemma 4.4 follows directly from Definition 3.3 and Proposition 3.4. $\Box$

Example 4.5 Consider $\mathcal{P}$ and the two models M_1 and M_2 from Example 4.1. M_1 is a stable model for $\mathcal{P}$, while M_2 is not. Indeed, M_1 is a stable model for $\alpha_{M_1}(\mathcal{P}) = \{a \vee b \leftarrow; \quad a \leftarrow b; \quad b \leftarrow a\}$ and M_2 is not a stable model for $\alpha_{M_2}(\mathcal{P}) = \{a \vee c \leftarrow; \quad a \leftarrow c\}$. $\Box$

Next, we show that by simply swapping the heads and bodies of the rules of the simplified program $\alpha_M(\mathcal{P})$, we get a set of clauses whose models correspond to the unfounded sets of $\mathcal{P}$ w.r.t. M.

Definition 4.6 Let $\mathcal{P}$ be a DLP program and M be a model for $\mathcal{P}$. Define $\beta_M(\mathcal{P})$ as the following set of clauses over M:

$$\begin{aligned}\beta_M(\mathcal{P}) = \{\ & b_1 \vee \ldots \vee b_m \leftarrow a_1 \wedge \ldots \wedge a_n \mid \\ & a_1 \vee \ldots \vee a_n \leftarrow b_1 \wedge \ldots \wedge b_m \in \alpha_M(\mathcal{P})\ \}\end{aligned}$$

$\Box$

Observe that $\beta_M(\mathcal{P})$ coincides with the set of clauses S constructed after steps 1–8 of Figure 5.

Lemma 4.7 *Let $\mathcal{P}$ be a ground DLP program, M a model for $\mathcal{P}$, and $X \subseteq M$. Then X is a model for $\beta_M(\mathcal{P})$ iff it is an unfounded set for $\mathcal{P}$ w.r.t. M.*

Proof. We know that $X \subseteq M$ is an unfounded set of M w.r.t. $\alpha_M(\mathcal{P})$ if and only if for each rule in $\alpha_M(\mathcal{P})$ either $H(r) \cap X = \emptyset$ or at least one of the three conditions C_1–C_3 from Definition 3.1 is true. Condition C_1 is always false because all rules in $\alpha_M(\mathcal{P})$ have true bodies. Therefore, X is an unfounded set of M iff $\bigwedge_{r \in \alpha_M(\mathcal{P})}((H(r) \cap X = \emptyset) \vee (B^+(r) \cap X \neq \emptyset) \vee ((H(r) - X) \cap M \neq \emptyset))$. For all rules in $\alpha_M(\mathcal{P})$, the bodies are positive and all atoms in the heads are true

w.r.t. M. Furthermore, $H(r) \cap X = \emptyset$ is subsumed by $H(r) - X \neq \emptyset$, since for all rules in $\alpha_M(\mathcal{P})$, $H(r) \neq \emptyset$. Because of that, we can simplify our requirements for X to be an unfounded set to $\bigwedge_{r \in \alpha_M(\mathcal{P})}((B(r) \cap X \neq \emptyset) \vee (H(r) - X \neq \emptyset))$, which equals $\bigwedge_{r \in \alpha_M(\mathcal{P})}((\bigvee_{b \in B(r)} b \in X) \vee (\bigvee_{h \in H(r)} h \notin X))$. Therefore, finding the unfounded sets of M w.r.t. $\alpha_M(\mathcal{P})$ is equal to computing the models of $\bigwedge_{r \in \alpha_M(\mathcal{P})}((\bigvee_{b \in B(r)} b) \vee (\bigvee_{h \in H(r)} \neg h))$. □

Example 4.8 $\beta_{M_1}(\mathcal{P})$ is $\{\leftarrow a \wedge b; \quad b \leftarrow a; \quad a \leftarrow b\}$. The only subset of M_1 which is a model of $\beta_{M_1}(\mathcal{P})$ is $\emptyset$ and M_1 is thus unfounded-free. Indeed, $\emptyset$ is the only unfounded set for M_1 w.r.t. $\mathcal{P}$.

$\beta_{M_2}(\mathcal{P})$ is equal to $\{\leftarrow a \wedge c; \quad c \leftarrow a\}$. M_2 has two subsets, $\emptyset$ and $\{c\}$, which are models of $\beta_{M_2}(\mathcal{P})$. Indeed these are precisely the unfounded sets for M_2 w.r.t. $\mathcal{P}$. □

We are now in a position to demonstrate our main theorem.

Proof of Theorem 4.2. In the following, we show that $\Gamma_M(\mathcal{P})$ is unsatisfiable iff M is unfounded-free. The statement will then directly follow from Proposition 3.4.

It is easy to see that the output $\Gamma_M(\mathcal{P})$ of the algorithm of Figure 5 coincides with the conjunction of all clauses in $\beta_M(\mathcal{P})$ and the clause $\vee_{x \in M} x$. From Lemma 4.7, the models of $\beta_M(\mathcal{P})$ are precisely the unfounded sets of $\mathcal{P}$ w.r.t. M. Therefore, the models of $\Gamma_M(\mathcal{P})$ are exactly the non-empty unfounded sets of $\mathcal{P}$ w.r.t. M, since every model of $\Gamma_M(\mathcal{P})$ must satisfy also the clause $\vee_{x \in M} x$, which states that at least one element of M has to be true in any model of $\Gamma_M(\mathcal{P})$. Thus, M contains no *non-empty* unfounded set for $\mathcal{P}$ (i.e., it is unfounded-free) iff $\Gamma_M(\mathcal{P})$ has no model (i.e., it is unsatisfiable). □

Example 4.9 $M_1 = \{a, b\}$ is a stable model for $\mathcal{P}$. Indeed,

$$\Gamma_{M_1}(\mathcal{P}) = (\leftarrow a \wedge b) \bigwedge (b \leftarrow a) \bigwedge (a \leftarrow b) \bigwedge (a \vee b \leftarrow)$$

is unsatisfiable. $M_2 = \{a, c\}$, on the other hand, is not stable for $\mathcal{P}$.

$$\Gamma_{M_2}(\mathcal{P}) = (\leftarrow a \wedge c) \bigwedge (c \leftarrow a) \bigwedge (a \vee c \leftarrow)$$

is satisfied by the model $\{c\}$. □

The next theorem shows that $\Gamma_M(\mathcal{P})$ is also an efficient transformation.

Theorem 4.10 *Given a model M for a ground DLP program $\mathcal{P}$, let $\Gamma_M(\mathcal{P})$ be the CNF formula computed by the algorithm of Figure 5 on input $\mathcal{P}$ and M. Then, the following holds.*

(1) $|\Gamma_M(\mathcal{P})| \leq |\mathcal{P}| + |M|$.

(2) $\Gamma_M(\mathcal{P})$ *is a parsimonious transformation.*
(3) $\Gamma_M(\mathcal{P})$ *is LOGSPACE computable from* $\mathcal{P}$ *and* M*.*

Proof. $\Gamma_M(\mathcal{P})$ is the conjunction of the clauses in $\beta_M(\mathcal{P})$ plus the disjunction of the propositions in M. The size of $\beta_M(\mathcal{P})$ is equal to the size of $\alpha_M(\mathcal{P})$, which is smaller than or equal to the size of $\mathcal{P}$. Thus, $|\Gamma_M(\mathcal{P})| \leq |\mathcal{P}| + |M|$.

$\Gamma_M(\mathcal{P})$ is clearly parsimonious, as it is a formula over the propositions of M only.

Finally, it is easy to see that $\Gamma_M(\mathcal{P})$ can be computed by a LOGSPACE Turing Machine. Indeed, $\Gamma_M(\mathcal{P})$ can be generated by dealing with one rule of $\mathcal{P}$ at a time, without storing any intermediate data apart from a fixed number of indices. □

The $\Gamma_M(\mathcal{P})$ transformation, reducing stable-model checking to UNSAT, suggests a straightforward way to implement a stable-model checker, namely

External Function *SAT*(Φ: CNF): Boolean;

Function *unfounded-free($\mathcal{P}$: Program; M: SetOfAtoms): Boolean;*
begin
  if $SAT(\Gamma_M(\mathcal{P}))$ **then return** *False*;
  else return *True*;
end;

Thus, we compute the unfounded-free property using an existing SAT solver by checking whether for a program $\mathcal{P}$ and a model M, the transformation $\Gamma_M(\mathcal{P})$ is unsatisfiable.

Theorem 4.11 *Given a program* $\mathcal{P}$ *and a model* M *for* $\mathcal{P}$*, the above-stated function* unfounded-free *returns true iff* M *is a stable model of* $\mathcal{P}$*.*

Proof. Immediate from Theorem 4.2. □

5 Enhancing the SAT-based Approach to Stable Model Checking by Modularity

As we have seen, the task of checking the stability condition of DLP can be transformed to UNSAT, a problem that is fairly well known and for which sophisticated algorithms exist, although, like stable model checking per se, it is still co-NP-complete.

In this section, we exploit two important properties of the problem of checking

the unfounded-freeness property of DLP programs: On the one hand, we know that for the important class of head-cycle-free (HCF) programs this computation can be done in polynomial time. On the other hand, we know that a form of modular evaluation is possible. To that end, we combine the $\mathcal{R}_{\mathcal{P},M}$ operator and modularity results described in Section 3 with our transformation. Apart from a new practical stable model checking algorithm, which we present in Section 5.2 and which is an improvement over the algorithm of Figure 4 in Section 3, we provide various minor equivalence results for combinations of our basic building blocks for simplification, namely the Γ transformation, the $\mathcal{R}_{\mathcal{P},M}$ operator, and modularity. Given the additional degree of freedom introduced with the Γ transformation of Section 4, this discussion is clearly needed.

First, however, we introduce a slightly generalized version of our transformation presented in the previous section.

5.1 Parameterizing $\alpha_M(\mathcal{P})$ and $\Gamma_M(\mathcal{P})$

The transformation presented in this section has a separate parameter allowing to make use of knowledge regarding which ground atoms may occur in unfounded sets and which atoms may not. We will later make use of this in the context of modular evaluation and for the efficient evaluation of head-cycle-free programs.

Definition 5.1 Let $\mathcal{P}$ be a program, M be a model for $\mathcal{P}$, and let X be a set such that $X \subseteq M$. We define the simplified version of $\mathcal{P}$ using M and X as:

$$\begin{aligned} \alpha_{M,X}(\mathcal{P}) = \{\ & a_1 \vee \ldots \vee a_m \leftarrow b_1 \wedge \ldots \wedge b_n \mid r \in \mathit{ground}(\mathcal{P}) \text{ and} \\ & \{a_1, \ldots, a_m\} = H(r) \cap M \text{ and} \\ & \{b_1, \ldots, b_n\} = B^+(r) \cap X \text{ and} \\ & B(r) \text{ is true w.r.t. } M \text{ and } H(r) \cap M \subseteq X\ \} \end{aligned}$$

□

Analogously to $\alpha_{M,X}(\mathcal{P})$, we can extend the full transformation $\Gamma_M(\mathcal{P})$ of Figure 5 by an additional parameter to filter out atoms that are known not to be in any unfounded sets.

Definition 5.2 Let $\mathcal{P}$ be a program, M be a model for $\mathcal{P}$, and let X be a set such that $X \subseteq M$. The transformation $\Gamma_{M,X}(\mathcal{P})$ is defined as

$$\begin{aligned} \Gamma_{M,X}(\mathcal{P}) = \{\textstyle\bigvee_{x \in X} x\} \cup \{\ & b_1 \vee \ldots \vee b_m \leftarrow a_1 \wedge \ldots \wedge a_n \mid \\ & a_1 \vee \ldots \vee a_n \leftarrow b_1 \wedge \ldots \wedge b_m \in \alpha_{M,X}(\mathcal{P})\ \}. \end{aligned}$$

□

Of course, both $\alpha_M(\mathcal{P})$ and $\Gamma_M(\mathcal{P})$ are special cases of $\alpha_{M,X}(\mathcal{P})$ and $\Gamma_{M,X}(\mathcal{P})$, respectively, where $X = M$. We generalize Lemma 4.4 and Theorem 4.2 to the transformations $\alpha_{M,X}(\mathcal{P})$ and $\Gamma_{M,X}(\mathcal{P})$.

Lemma 5.3 *Given a program $\mathcal{P}$, a model M for $\mathcal{P}$, and a set $X \subseteq M$ s.t. it is known that for each unfounded set U of M w.r.t. $\mathcal{P}$, $U \subseteq X$* [8]*.*

(1) M is unfounded-free for $\mathcal{P}$ iff it is unfounded-free for $\alpha_{M,X}(\mathcal{P})$.
(2) $\Gamma_M(\mathcal{P})$ is satisfiable if and only if $\Gamma'_{M,X}(\mathcal{P})$ is satisfiable.

Proof.
(1) Remember the known equivalence between the unfounded sets of $\alpha_M(\mathcal{P})$ and the models of $\beta_M(\mathcal{P})$. Since no atom in $M - X$ is in an unfounded set of $\alpha_M(\mathcal{P})$, no atom in $M-X$ may be in a model of $\beta_M(\mathcal{P})$. Thus, we may remove those atoms from the heads of clauses in $\beta_M(\mathcal{P})$ (and thus the bodies of rules in $\alpha_M(\mathcal{P})$) and may remove all clauses from $\beta_M(\mathcal{P})$ whose bodies contain atoms in $M - X$ (these bodies must be "false"). Translated back to the perspective of $\alpha_M(\mathcal{P})$, this results in $\alpha_{M,X}(\mathcal{P})$.

(2) follows trivially from (1) and Theorem 4.2. □

Example 5.4 By Lemma 5.3, we can combine the transformation $\alpha_{M,X}(\mathcal{P})$ with $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$, which is of course guaranteed to subsume all unfounded sets of M w.r.t. $\mathcal{P}$.

Let $\mathcal{P} = \{a \vee b;\ a \leftarrow b;\ b \leftarrow a \wedge c;\ c \leftarrow\}$ and $M = \{a, b, c\}$. We have $\alpha_M(\mathcal{P}) = \mathcal{P}$ and $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \{a, b\}$. Here,

$$\alpha_{M,\mathcal{R}^{\omega}_{\mathcal{P},M}(M)}(\mathcal{P}) = \{a \vee b;\ a \leftarrow b;\ b \leftarrow a\}$$

and

$$\Gamma_{M,\mathcal{R}^{\omega}_{\mathcal{P},M}(M)}(\mathcal{P}) = \{\leftarrow a \wedge b;\ b \leftarrow a;\ a \leftarrow b\} \cup \{a \vee b\},$$

which is unsatisfiable, as is

$$\Gamma_M(\mathcal{P}) = \{\leftarrow a, b;\ b \leftarrow a;\ c \vee a \leftarrow b; \leftarrow c\} \cup \{a \vee b \vee c\}.$$

□

5.2 A Dynamically-Modular Stable-Model Checker

The modular evaluation result of Proposition 3.13 allows to reduce the task of computing the unfounded-freeness property for programs to computing it over

[8] Trivially, this property holds for $X = M$.

```
Function unfounded-free(P: Program; M: SetOfAtoms): Boolean;
var P': Program;
        X, Y, Q: SetOfAtoms;
begin
        X := R^ω_{P,M}(M);
        if X = ∅ then return True;
        if P is HCF then return False;
        P' := α_{M,X}(P);
        compute \widehat{DG}_{P'};
        for each node Q of \widehat{DG}_{P'}
                if subp(Q, P') is HCF then
                        Y := R^ω_{subp(Q,P'),M}(M);
                        if Y ≠ ∅ then return False;
                else (* Computation for non-HCF components *)
                        compute Γ_{M,Q}(subp(Q, P'));
                        if SAT(Γ_{M,Q}(subp(Q, P'))) then return False;
                end if;
        end for;
        return True;
end;
```

Fig. 6. *The new algorithm for checking the Unfounded-Free Property*

the (often much smaller) strongly connected components of their dependency graph.

By combining Proposition 3.13 with Lemma 5.3, we obtain that a model M is unfounded-free w.r.t. a program P iff for all components in the dependency graph $\widehat{DG}_{\mathcal{P}}$, $\Gamma'_{M,\frac{M}{Q}}(subp(Q, \mathcal{P}))$ is unsatisfiable. It is clear that the modularity result applies also to the simplified versions of programs. In particular, it applies to $\alpha_M(\mathcal{P})$ as well to $\alpha_{M,\mathcal{R}^{\omega}_{\mathcal{P},M}(M)}(\mathcal{P})$, instead of just $\mathcal{P}$. The components of such simplified programs may be fewer and much smaller than the components of the dependency graph of the non-simplified program. A program as a whole is unfounded-free iff each of the transformed subprograms is unsatisfiable.

These ideas now need to be combined. Figure 6 shows an algorithm for model checking which incorporates our results. Initially, we start by computing the fixpoint $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$. The reason for this is that $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$ can be computed in linear time, which eliminates the need to save time by splitting the program. Also, some rules may be contained in several component subprograms, and by keeping the program together, we even save time. Furthermore, by computing the dependency graph on the simplified version of $\mathcal{P}$, there is a chance that the program is split into smaller components. In the simplification, some rules may have been removed that contributed to the arcs in the dependency graph.

Next, we handle the case that $\mathcal{P}$ is HCF, in which we can immediately deter-

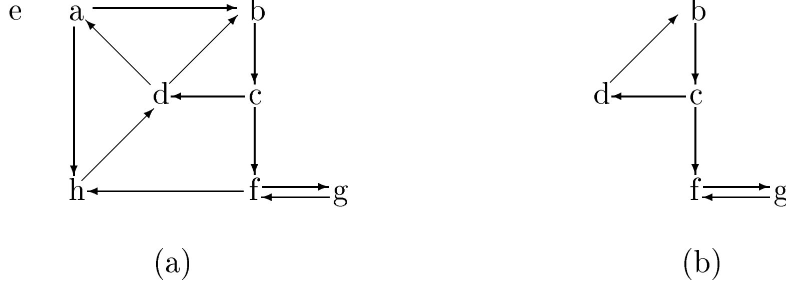


Fig. 7. *Graphs (a) $DG_{\mathcal{P}}$ and (b) $DG_{\alpha_{M,\mathcal{R}^{\omega}_{\mathcal{P},M}(M)}(\mathcal{P})}$ for the program $\mathcal{P}$ of Example 5.6. Note that the component graph of (a) consists of the nodes $\{a, b, c, d, f, g, h\}, \{e\}$, while in the component graph of (b), the small component has been removed and the large one has been pruned and split up into $\{b, c, d\}, \{f, g\}$.*

mine the unfounded-freeness property by checking the size of $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$. Then we compute the dependency graph of the simplified program $\alpha_{M,\mathcal{R}^{\omega}_{\mathcal{P},M}(M)}(\mathcal{P})$ and check the unfounded-freeness of each of its component subprograms independently, similarly to the old algorithm of Figure 4. (That is, a program is unfounded-free if and only if all of its subprograms are unfounded-free.) In the HCF-case, we recompute the fixpoint of the $\mathcal{R}$ operator for the component and check its size against 0. In the non-HCF case, we refrain from reapplying the $\mathcal{R}$ operator because it proved more efficient to directly run the SAT solver in practice.

Our algorithm computes the unfounded-free property for a ground program and a model:

Theorem 5.5 *Let $\mathcal{P}$ be a ground DLP program and M a model for $\mathcal{P}$. Then, M is stable w.r.t. $\mathcal{P}$ iff the function* unfounded-free*($\mathcal{P}$, M) of Figure 6 returns* true*.*

Proof. The correctness of Theorem 5.5 follows immediately from the theoretical results of Sections 3 and 4, and Lemma 5.3. □

Example 5.6 Let $\mathcal{P}$ be the program

$$
\begin{array}{lll}
a \vee e \leftarrow & b \leftarrow a \wedge d & c \leftarrow b \\
b \vee c \leftarrow & d \leftarrow c \wedge h & a \leftarrow d \\
f \leftarrow g & g \leftarrow f & f \leftarrow c \\
h \leftarrow f & h \leftarrow a &
\end{array}
$$

and $M = \{a, b, c, d, f, g, h\}$ a model for $\mathcal{P}$. The dependency graph $DG_{\mathcal{P}}$ is shown in Figure 7 (a). Since all the atoms in M are in the same non-HCF strongly connected component, the model checking algorithm of Figure 4 cannot make use of the modularity results.

The situation is different for the algorithm of Figure 6, which operates as follows. In the first step, we compute the fixpoint $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \{b, c, d, f, g\}$. Then we simplify $\mathcal{P}$ to obtain $\mathcal{P}' = \alpha_{M,\mathcal{R}^{\omega}_{\mathcal{P},M}(M)}(\mathcal{P}) =$

$$b \leftarrow d \qquad c \leftarrow b \qquad b \vee c \leftarrow \qquad d \leftarrow c$$
$$f \leftarrow g \qquad g \leftarrow f \qquad f \leftarrow c$$

with the dependency graph shown in Figure 7 (b). $DG_{\mathcal{P}'}$ has two strongly connected components, $Q_1 = \{b, c, d\}$ and $Q_2 = \{f, g\}$. Q_1 is not HCF, but unfounded-free as $\Gamma'_{M,Q_1}(subp(Q_1, \mathcal{P}')) =$

$$d \leftarrow b; \qquad b \leftarrow c; \qquad \leftarrow b \wedge c; \qquad c \leftarrow d; \qquad b \vee c \vee d \leftarrow$$

is unsatisfiable. Q_2 is head-cycle-free and unfounded-free, with

$$\mathcal{R}^{\omega}_{subp(Q_2,\mathcal{P}'),Q_2}(Q_2) = \emptyset.$$

Thus, M is a stable model of $\mathcal{P}$. □

We conclude this section with one more remark on our stable model checking algorithm.

We made use of Proposition 3.10, which provides an efficient method for checking the unfounded-freeness of head-cycle-free programs by checking whether $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$ is the empty set, in the algorithm of Figure 4, where we also combined it with modularity results. At the first glance it may seem that if the component dependency graph of $\alpha_{M,\mathcal{R}^{\omega}_{\mathcal{P},M}(M)}(\mathcal{P})$ contains a head-cycle-free component, M is not a stable model. Unfortunately, the following counterexample shows that this is not true in general.

Example 5.7 Let $\mathcal{P}$ be the program

$$a \leftarrow b \qquad b \leftarrow a \qquad a \leftarrow c \qquad c \vee d \leftarrow \qquad c \leftarrow d \qquad d \leftarrow c$$

and $M = \{a, b, c, d\}$ a model for $\mathcal{P}$. We have $\alpha_M(\mathcal{P}) = \mathcal{P}$ and $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = M$. One of the strongly connected components, $Q = \{a, b\}$, is head-cycle-free with $subp(Q, \mathcal{P}) = \{a \leftarrow b;\ b \leftarrow a;\ a \leftarrow c\}$. Reapplying the $\mathcal{R}$ operator on Q results in $\emptyset$, though, which is correct, because M is the unique stable model of $\mathcal{P}$. □

Thus, computing subcomponents of a program may lead to rules "breaking apart" which may permit further simplifications using the $\mathcal{R}_{\mathcal{P},M}$ operator. This shows why for HCF components, we have to re-apply the $\mathcal{R}$ operator, as we do in the algorithm of Figure 6.

6 Implementation, Comparisons and Benchmarks

In order to test the usefulness of our proposal, we have implemented our method in the `DLV` system. `DLV` [38,13] is a knowledge representation system based on disjunctive logic programming which has been developed at Technische Universität Wien. Recent comparisons [13,39,40] have shown that `DLV` is nowadays a state-of-the-art implementation of disjunctive logic programming.

The computational engine of `DLV` implements the theoretical results achieved in [23]. Roughly, the system consists of two main modules: the Model Generator and the Model Checker (MC). The former produces stable-model candidates, whose stability is then checked by the latter.

We have replaced the original Model Checker of `DLV` by a new module implementing the results of the previous sections, performed various benchmarks, and compared the execution times.

In addition to `DLV`, we have evaluated `GnT` [20] (an extension of Smodels [41,42]), which is, to the best of our knowledge, the only publicly available system apart from `DLV` which supports full (function-free) disjunctive logic programming under the stable model semantics.

In the remainder of this section, we compare the (disjunctive) stable-model checking methods considered and their differences, and report on the experiments we have carried out.

Comparative Overview of Disjunctive Stable-Model Checking Methods

In this section, we briefly recall the disjunctive stable-model checking methods we consider, and we discuss their main differences.

We have tested the following systems and methods for stable model checking (the labels below will be used in the benchmark figures).

- (`Old Checker`) The `DLV` system with its original Model Checker using the algorithm of Figure 4 which employs the modularity results from [23].

  Its strong points are the efficient evaluation of head-cycle-free (HCF) programs [35,36] and the use of modular evaluation techniques. Indeed, HCF programs are evaluated in polynomial time and, if the program is not HCF, the inefficient part of the computation is limited only to those subprograms which are not HCF (while the polynomial time algorithm is applied to the HCF subprograms). Polynomial space and single exponential time bounds are always guaranteed.

- ($\mathtt{DLV}_{new}$) This is the algorithm depicted in Figure 6 and described in Section 5.2. Here, we again summarize its main ideas.

  Given a program $\mathcal{P}$ and a model M to be checked for stability, an implementation of the transformation of Figure 5 generates the CNF formula $\Gamma_M(\mathcal{P})$, which is then submitted to a satisfiability checker. If the SAT checker returns true ($\Gamma_M(\mathcal{P})$ is satisfiable), then M is not a stable model of $\mathcal{P}$; otherwise ($\Gamma_M(\mathcal{P})$ is unsatisfiable), M is a stable model of $\mathcal{P}$. For checking satisfiability of $\Gamma_M(\mathcal{P})$, we have used SATZ [21] - an efficient implementation of the Davis-Putnam procedure ([43]) - customized for our setting.

  Furthermore, this stable model checking method is enhanced by modular evaluation techniques derived from the combination of Lemma 4.4 with the modularity results of [23]. Roughly, given $\mathcal{P}$ and M, $\mathcal{P}$ is first simplified (steps 1–3 of Figure 5) resulting in the program $\alpha_M(\mathcal{P})$. The subprograms of $\alpha_M(\mathcal{P})$ are then evaluated one after the other. [9] A polynomial time method is applied to HCF subprograms (as in `Old Checker`), while the transformation to SAT is applied to non-HCF subprograms.
- (`GnT`) The `GnT` system [20] is a disjunctive extension of the system Smodels [41,42]. (It is included in the Smodels distribution [44] under the name of `example4`.) Once a stable-model candidate M for a (disjunctive) program $\mathcal{P}$ has been generated, a disjunction-free logic program $\mathcal{P}(M)$ is generated from $\mathcal{P}$ and M. The stable models of $\mathcal{P}(M)$ are the models of $\mathcal{P}^M$ which are strictly contained in M. [10] Therefore, M is a stable model of $\mathcal{P}$ if and only if $\mathcal{P}(M)$ has no stable model. The logic program $\mathcal{P}(M)$ is evaluated by a self call to (another instance of) Smodels; if no stable models are generated then M is stable. In our benchmarks, we have used Smodels 2.26, which was the current version at that time. Recently Smodels 2.27 was released which fixes an unrelated bug and packaging issues only; we verified that this does not affect performance and thus did not re-run all benchmarks.

Comparing our approach (i.e., $\mathtt{DLV}_{new}$) to the stable model checking method of `GnT`, we observe the following main differences:

(1) The method implemented in `GnT` can be seen as the dual method of our approach. Indeed, through $\mathcal{P}(M)$ `GnT` tries to generate directly a model M' of $\mathcal{P}^M$ which is strictly contained in M (disproving the minimality of M). In contrast, through the CNF formula $\Gamma_M(\mathcal{P})$ we try to build a non-empty unfounded set X contained in M, which witnesses the non-minimality of M without building explicitly the model contained in M (the existence of such an unfounded set X implies that $M - X$ is a model of $\mathcal{P}^M$).

[9] Note that the subprograms of $\alpha_M(\mathcal{P})$ are smaller in general than the subprograms of $\mathcal{P}$, since the simplification process may break components.
[10] Recall that $\mathcal{P}^M$ denotes the Gelfond-Lifschitz transformation of $\mathcal{P}$ w.r.t. M (see Section 2).

(2) In GnT, the stable model check is performed by a call to a logic programming system (Smodels); while we employ a SAT checker (over $\Gamma_M(\mathcal{P})$) to check the stability.
(3) GnT always uses the same model checking strategy whatever is the input program. Instead, we make some syntactic checks, and adopt specialized algorithms for some syntactically recognizable classes of programs. In particular, our model checker works in polynomial time if the input program is head-cycle-free.
(4) To our knowledge, GnT checks the stability of a model candidate M at once, and it does not employ any modular evaluation techniques; while an important feature of our approach is the use of the dynamic modular evaluation techniques (see Section 5), which limit the inefficient part of the computation only to the components which are still not head-cycle-free after that a simplification has been applied on the program.

The $\texttt{DLV}_{new}$ method enhances `Old Checker` in the following respects.

- In $\texttt{DLV}_{new}$, the hard (non-HCF) subprograms are evaluated by calling a SAT checker on a suitable CNF formula ($\Gamma_M(\mathcal{P})$); while an enumeration of the possible unfounded sets is performed in `Old Checker`.
- $\texttt{DLV}_{new}$ implements more advanced modularity techniques, which allow for a finer splitting the stability check in subtasks (in general, $\texttt{DLV}_{new}$ deals with smaller subprograms, thanks to the dynamic modularity technique).

6.1 Benchmark Problems and Data

In order to generate co-NP-hard model checking instances which can be used to evaluate the differences between various model checking techniques, we needed to perform benchmarks of Σ_2^P-hard problems. Finding a suitable set of hard instances was not easy, since only a few experimental works have been done so far on Σ_2^P-complete problems, and systematic studies to single out cross-over points similar to those done for Satisfiability are still missing.

We have considered two problems for benchmarks:

- Quantified Boolean Formulas (2QBF), and
- Strategic Companies (STRATCOMP).

For 2QBF we could exploit previous works studying hard instances; while for STRATCOMP there was no such a study, the instances previously used for benchmarks appear very easy to solve (as pointed out also in [20]), and we had to single out harder instances experimentally (this is to be considered a further side-contribution of this paper).

2QBF

Our first benchmark problem residing on the second level of the polynomial hierarchy is 2QBF, which is well known to be Σ_2^P-complete [45]. The problem here is to decide whether a quantified Boolean formula (QBF) $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a 3DNF formula over $X \cup Y$, is valid.

The transformation from 2QBF to disjunctive logic programming is a slightly altered form of a reduction used in [46]. The propositional disjunctive logic program $\mathcal{P}_\phi$ produced by the transformation requires $2 * (|X| + |Y|) + 1$ propositional predicates (with one dedicated predicate w), and consists of the following rules.

(1) Rules of the form $v \vee \bar{v}$ for each variable $v \in X \cup Y$.
(2) Rules of the form $y \leftarrow w$; $\bar{y} \leftarrow w$ for each variable $y \in Y$.
(3) Rules of the form $w \leftarrow v_1 \wedge \ldots \wedge v_m \wedge \bar{v}_{m+1} \wedge \ldots \wedge \bar{v}_n$ for each conjunction $v_1 \wedge \ldots \wedge v_m \wedge \neg v_{m+1} \wedge \ldots \wedge \neg v_n$ in ϕ.
(4) The rule $\leftarrow$ not w.

The 2QBF formula Φ is valid iff $\mathcal{P}_\Phi$ has a stable model [46].

Example 6.1 The formula $\Phi = \exists x \forall y[(\neg x \wedge y) \vee (\neg y \wedge x)]$ translates into

$$\mathcal{P}_\Phi = \{x \vee \bar{x};\ y \vee \bar{y};\ y \leftarrow w;\ \bar{y} \leftarrow w;\ w \leftarrow \bar{x} \wedge y;\ w \leftarrow \bar{y} \wedge x;\ \leftarrow \text{not } w\}$$

$\mathcal{P}_\Phi$ does not have a stable model, thus the QBF ϕ is not valid. (To check this manually, it is simpler to verify that $\neg\Phi = \forall x \exists y : x \leftrightarrow y$ is valid.) □

In disjunctive logic programming, unlike SAT-based programming, programs should be kept separated from data. One designs a program encoding the problem at hand, which is then fixed and allows you to solve all problem instances which are provided by set of facts. In our benchmarks, we adhere to the above principle, and clearly separate the encoding from the problem instance. To this end, we create the following disjunctive logic program $\mathcal{P}_{qbf}$:

$$\begin{aligned}
T(X) \vee F(X) &\leftarrow \mathit{Exists}(X);\\
T(X) \vee F(X) &\leftarrow \mathit{Forall}(X);\\
T(X) &\leftarrow w \wedge \mathit{Forall}(X);\\
F(X) &\leftarrow w \wedge \mathit{Forall}(X);\\
w &\leftarrow \mathit{Conjunct}(X, Y, Z, Na, Nb, Nc) \wedge\\
&\quad T(X) \wedge T(Y) \wedge T(Z) \wedge F(Na) \wedge F(Nb) \wedge F(Nc);\\
T(\mathit{true}) \leftarrow;& \qquad F(\mathit{false}) \leftarrow; \qquad \leftarrow \text{not } w
\end{aligned}$$

A 2QBF instance $\Phi = \exists X \forall Y \phi$ is encoded by the following set F_Φ of facts:

- *Exists*(v), for each existential variable $v \in X$
- *Forall*(v), for each universal variable $v \in Y$
- *Conjunct*$(x_1, x_2, x_3, y_1, y_2, y_3)$, for each conjunct $l_1 \wedge l_2 \wedge l_3$ in ϕ, where (i) if l_i is a positive atom l_i then $x_i = v_i$, otherwise $x_i =$"true", and (ii) if l_i is a negated atom not v_i, then $y_i = v_i$, otherwise $x_i =$"false".

The 2QBF instance Φ is valid if and only if $\mathcal{P}_{qbf} \cup F_\Phi$ has a stable model.

We generated two different kinds of data sets following two works presented in the literature. Each data set was randomly generated. In both cases the number of $\forall$-variables is equal to the number of $\exists$-variables (that is, $|X| = |Y|$) and each conjunct contains at least two universal variables. [11] In the first case, the number of clauses equals the overall number of variables (that is, $|X|+|Y|$); in the second case, suggested by Gent and Walsh [47], the number of clauses is $\sqrt{((|X| + |Y|)/2)}$. In the following, we will refer to instances generated according to the first schema simply as QBF, those generated according to the second schema as QBF_{GW}.

Strategic Companies (STRATCOMP)

This problem has been introduced by [48] in the context of Default Logic. It is a Σ_2^P-complete problem from the business domain.

The Strategic Companies Problem is defined as follows: A holding owns companies, each of which produces some goods. Moreover, several companies may have joint control over another company. Now, some of these companies should be sold, under two constraints: All goods can be still produced, and no company is sold which would still be controlled by the holding after the transaction. A company is *strategic* if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints. Using our formalism, these sets can be expressed by the following natural program:

$$
\begin{aligned}
&\mathit{strategic}(C_1) \vee \mathit{strategic}(C_2) \vee \mathit{strategic}(C_3) \vee \mathit{strategic}(C_4) \\
&\qquad\qquad \leftarrow \mathit{produced_by}(P, C_1, C_2, C_3, C_4) \\
&\mathit{strategic}(C) \leftarrow \mathit{controlled_by}(C, C_1, C_2, C_3, C_4) \wedge \mathit{strategic}(C_1) \\
&\qquad\qquad \wedge\ \mathit{strategic}(C_2) \wedge\ \mathit{strategic}(C_3) \wedge\ \mathit{strategic}(C_4)
\end{aligned}
$$

Here the atom *strategic*(C) means that C is a strategic company, the atom *produced_by*(P, C_1, C_2, C_3, C_4) that product P is produced by companies C_1,

[11] In conjunction with the second variable ratio, this constitutes the so-called Model A whose hardness has been experimentally evaluated in [47].

C_2, C_3 and C_4, and *controlled_by*(C, C_1, C_2, C_3, C_4) that a company C is jointly controlled by companies C_1, C_2, C_3 and C_4. We have released the constraints imposed in [48], where each product is produced by at most two companies and each company is jointly controlled by at most three other companies, to at most four producers per product and four controllers per company (in principle these numbers can be increased arbitrarily). We experimentally determined that releasing these constraints allows us to generate harder instances on average.

The problem now is to determine whether a given company c is strategic or not (i.e., if the fact *strategic*(c) is true in at least one stable model of the program above).

Note that this problem cannot be expressed by a fixed normal ($\vee$-free) logic program uniformly on all collections of facts over the predicates *produced_by* and *controlled_by* unless NP $= \Sigma_2^P$, which is highly unlikely. Thus, Strategic Companies is an example of a relevant problem where the full expressive power of disjunctive logic programming is really needed.

We have generated tests with instances for n companies ($5 \leq n \leq 170$), $3n$ products, 10 uniform randomly chosen *controlled_by* relations per company, and uniform randomly chosen *produced_by* relations. To make the problem harder, we are only considering strategic sets containing two fixed companies (1 and 2, without loss of generality) using the constraints:

$$\leftarrow \text{not } \mathit{strategic}(1)$$
$$\leftarrow \text{not } \mathit{strategic}(2)$$

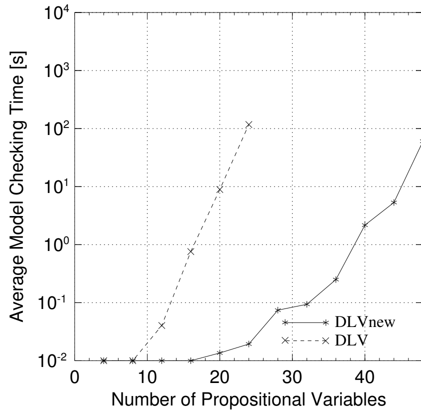
Results and Discussion

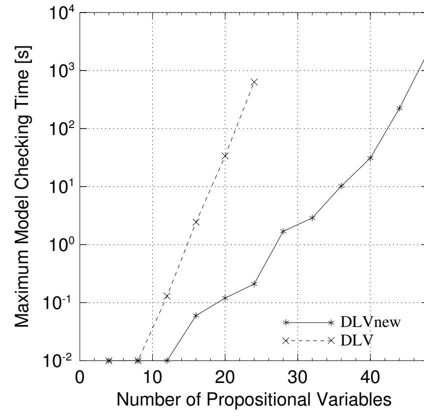
Our experiments were run on an Athlon/1200 with 512MB of main memory under FreeBSD 4.4, using the GCC 2.95.3 C++ compiler.

For each problem size we have generated 50 random instances as indicated in the respective descriptions, and for each such instance we allowed a maximum running time of 7200 seconds (two hours). In the graphs displaying the benchmark results, the line of a system stops whenever some problem instance was not solved in the maximum allowed time.

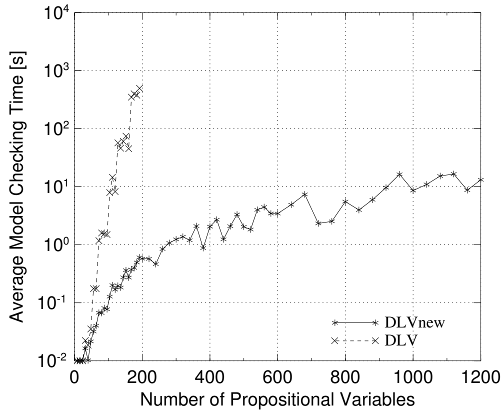
In this framework, we ran two series of benchmarks: In the first series, we compared our new model checking strategy (`DLV`$_{new}$) against the old model checking strategy of `DLV` (`Old Checker`). In the second series, we compared `DLV`$_{new}$ against the `GnT` system.

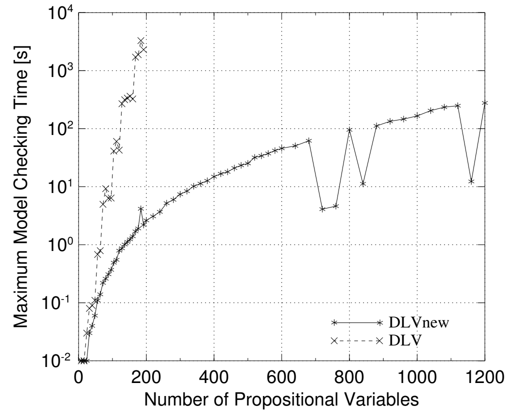
QBF



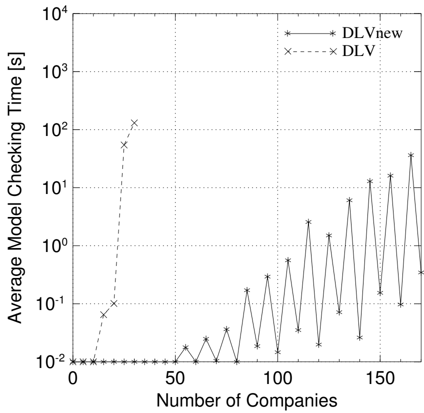


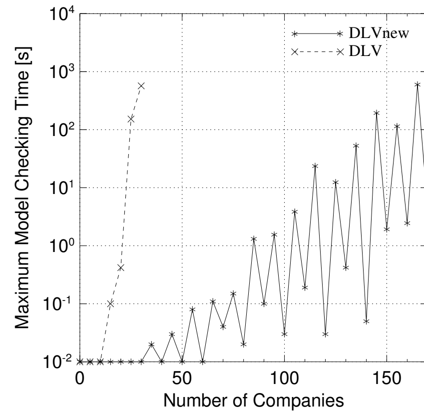
QBF_{GW}





STRATCOMP





Average model checking time Maximum model checking time

Fig. 8. $\texttt{DLV}_{new}$ vs `Old Checker`: Model checking times for first stable model

6.1.1 $\texttt{DLV}_{new}$ *vs* `Old Checker`

We used `Old Checker` and $\texttt{DLV}_{new}$ to compute the first stable model of the program (which determines the solution of the decision problem – see above)

and summed up the total time spent for model checking. [12] Thus, we obtained a precise comparison of the efficiency of the two model checking strategies.

The results of the experiments comparing $\texttt{DLV}_{new}$ vs `Old Checker` are displayed in Figure 8. The graphs on the left sides display the average (over the 50 instances of the same size) time spent for model checking; while the graphs on the right sides display the maximum time spent for model checking. In particular, the graphs on the top, mid and bottom of the figure refer to the QBF, QBF_{GW}, and STRATCOMP instances. (Average and Maximum) Execution times (expressed in CPU seconds) are reported on the vertical axis, while the horizontal axis displays the problem-instance size (number of propositional variables for QBF problems; number of companies for STRATCOMP). Note that, in all figures of this section, the vertical axis is in a logarithmic scale, and we have cut resp. rounded all values below 0.01s.

The graphs of Figure 8 show very clearly the strong impact of the new strategy on the efficiency of the model checker of `DLV`. $\texttt{DLV}_{new}$ is significantly faster than `Old Checker` in each of the three experiments. Both the average model checking time, and the maximum model checking time of `Old Checker` are always higher than the respective times of $\texttt{DLV}_{new}$. The lines of `Old Checker` stop much earlier than those of $\texttt{DLV}_{new}$, evidencing that some instance of small size is not solvable by `DLV` using the original model checking strategy; while, `DLV` with the new strategy performs much better, and is able to solve all instances of significantly larger sizes.

6.1.2 *$\texttt{DLV}_{new}$ vs GnT*

In the second series of experiments, we compare $\texttt{DLV}_{new}$ and `GnT`. Due to the completely different model generation strategies of these systems which may lead to a very different number of stable model candidates (and thus stable model checks), we solve the decision problem whether any stable model exists (thus looking for one stable model), as before; but we consider the total execution times.In other words, we check whether the version of `DLV`, implementing the model checking techniques proposed in this paper, is competitive with the `GnT` system on Σ_2^P-complete problems.

The results are displayed by the six graphs of Figure 9 in the same way as in Figure 8. In general, $\texttt{DLV}_{new}$ outperforms `GnT` in all benchmark problems. However, the results are very different in the three experiments. On STRATCOMP, the performance of the two systems are basically the same up to the instance size of 115 companies. But instance-size 115 is the last size where `GnT` can solve all of the 50 instances; while $\texttt{DLV}_{new}$ goes beyond solving many further

[12] Note that the computation of one stable model requires $m \geq 1$ calls to the Model Checker ($m - 1$ is the number of calls on models which are not stable).

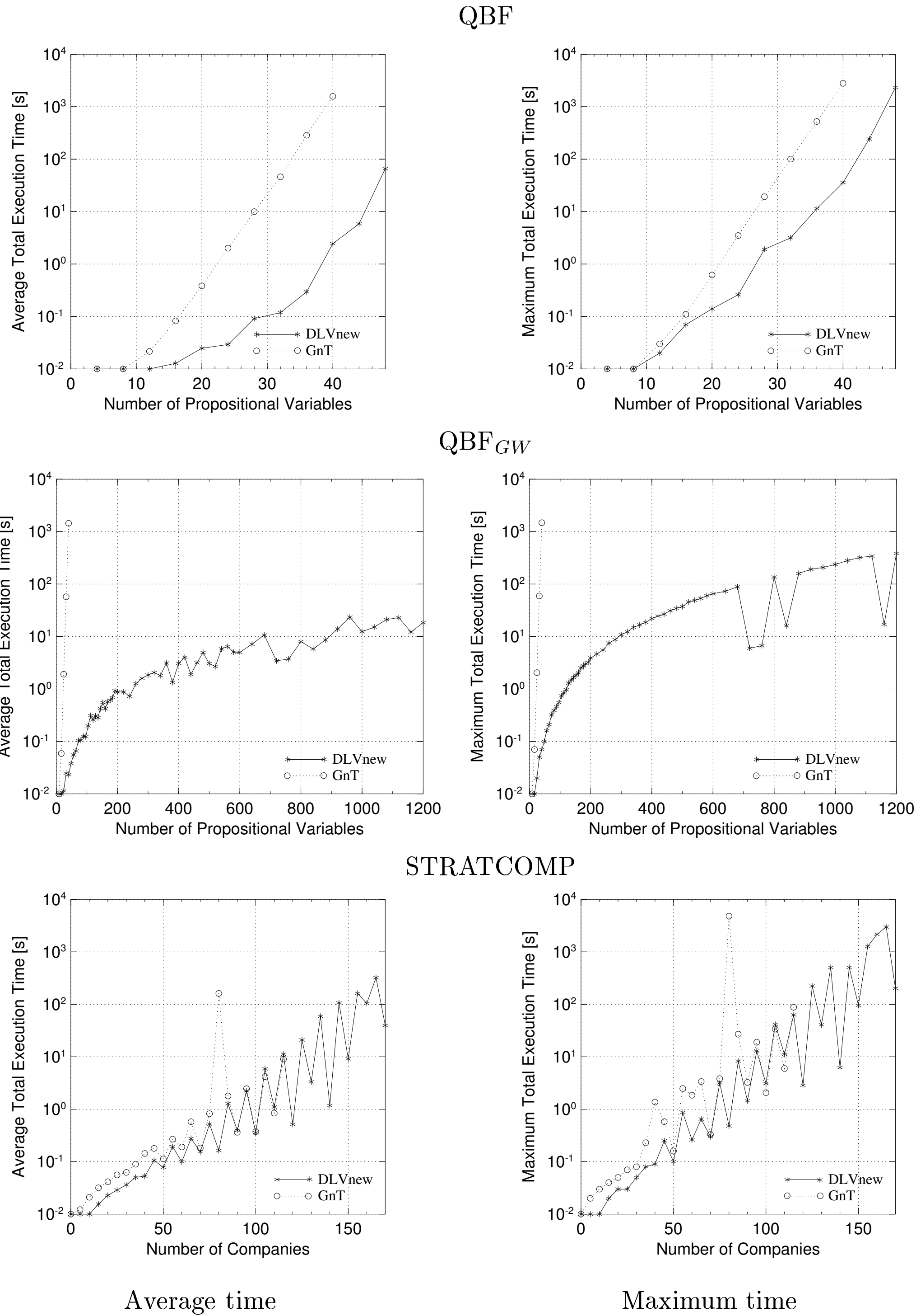


Fig. 9. $\mathtt{DLV}_{new}$ vs `GnT`: Total Running times for first stable model

instances. `GnT` performs relatively well also on the first kind of QBF instances. Instead, the difference between `GnT` and $\mathtt{DLV}_{new}$ becomes very impressive on QBF$_{GW}$, where `GnT` stops at size 25; while $\mathtt{DLV}_{new}$ solves all instances of size 1200 employing less than 17 seconds on average.

7 Related Work and Conclusion

7.1 Further Related Work

There is not much work in the literature on efficient methods for stable model checking of disjunctive logic programs. The work more closely related to our method is probably the one by Ben-Eliyahu-Zohary and Palopoli [36], since it focuses on efficient methods for (minimal) model checking. [13]

Based on the seminal work by Ben-Eliyahu and Dechter [35] where the notion of head-cycle-freeness (HCF) was introduced, Ben-Eliyahu-Zohary and Palopoli [36] describe an efficient algorithm to compute minimal models of head-cycle-free theories and logic programs and use this algorithm to compute one (arbitrary) stable model of a stratified head-cycle-free DLP program in linear time. Our $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$ operator extends this to non-stratified input and we also use $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$ to (possibly) simplify problems that are not HCF before applying more expensive techniques.

The dynamic modular evaluation techniques employed by our algorithm to check the stability condition (see Section 5.2) extends [35] and [36] in that it allows us to apply an efficient model checking procedure also to programs which are not head-cycle-free initially, but become such once they are simplified w.r.t. the model to be checked for stability.

There are several other works on computational aspects of DLP, which do not focus on stable model checking, though, and thus we only briefly mention them for completeness:

Fernández and Minker [49] employ a fixpoint characterization to evaluate stratified programs, using so called model-trees which encode finite families of interpretations.

Another algorithm for computing stable models which uses a bottom-up strategy is presented by Brass and Dix in [25]. Their algorithm first computes the "residual program" – a program where no positive literals appear in the rules' bodies – which is equivalent to the original program under stable model semantics. Stable models are then computed on (a simple extension of) Clark's completion of the residual program.

Also Dix and Müller have implemented various semantics of disjunctive logic

[13] Recall that stable models coincide with minimal models on positive disjunctive logic programs, and, also on general disjunctive programs, minimal model checking is the the hard task of stable model checking.

programs based on abstract properties [50], but their procedure applies only to stratified programs.

Stuber's bottom-up approach [51], finally, works similar to `DLV` and `GnT` in that it employs a procedure analogous to Davis-Putnam [43], using case analysis and simplification. Like `DLV` and `GnT`, and unlike the approaches mentioned above, Stuber's procedure only requires polynomial space and avoids the generation of duplicate (stable) models. Instead of performing a model check for every model found, this approach performs (co-NP-hard) checks already as part of the backtracking model computation. Stuber leaves the concrete implementation of these checks as an open issue, but in general his algorithm may require exponential time for checking even if the program is HCF while our procedure for checking stability is always polynomial on such programs.

Polynomial space complexity is a crucial requirement both for logic programming based as well as deductive database systems, cf. [52], and of the approaches above that are able to deal with hard input, only `DLV`, `GnT`, and Stuber's meet this property.

Several approaches to the implementation of answer set programming systems like ASSAT [53], CCALC [54], cmodels [55], DCS [56], DeReS [57], DisLog [58], DisLoP [59], NoMoRe [60], QUIP [61], Smodels [41], and XSB [62], including the two systems described and evaluated in the previous section, namely `DLV/Old Checker` [13,14,63] and `GnT` [20], are evidently in connection to our paper as well.

7.2 Summary

As evidenced before by practical examples, disjunctive logic programming (DLP) with the stable model semantics is a powerful knowledge representation and nonmonotonic reasoning formalism. Reasoning with DLP is harder than with disjunction-free logic programs because *stable model checking* (that is, deciding whether a given model is a stable model of a propositional DLP program) is co-NP-complete.

The model checking component is an essential part of nonmonotonic reasoning systems following the stable model semantics which can deal with Σ_2^P-complete problems. In this paper, we have proposed a new, efficient transformation $\Gamma_M(\mathcal{P})$, which reduces stable model checking to UNSAT. The rationale of this is that UNSAT is the prototypical and best-researched co-NP-complete problem. By this step, the best special-purpose algorithms and systems for UNSAT can be used to solve the stable model checking problem. Thus, our work allows for a very substantial improvement of model checking performance of DLP systems. This in turn has significant repercussions on the efficiency

frontier of AI systems for Σ_2^P-complete problems overall.

The proposed approach to stable model checking has been implemented in DLV– a state-of-the-art implementation of DLP which is publicly available for a large number of platforms, and a number of experiments and benchmarks have been run using SATZ – one of the best SAT solvers currently available – as an engine for stable model checking. The results of the experiments are very positive and confirm the usefulness of our techniques.

As future work, we plan to further improve the integration of model checking with the other reasoning modules of DLV (in particular, model generation) and to add model checking heuristics; for instance, we want to exploit unfounded sets (evidenced by models violating the UNSAT check) to guide the model generation process. Many other possible heuristics are awaiting experimental evaluation; indeed, we are not aware of any existing work on heuristics for Σ_2^P problems.

Our experiments are a first foray into benchmarking Σ_2^P-hard problems in the context of DLP. We see a strong need for further studies on cross-over points for problems such as STRATCOMP and QBF, the prototypical problem of this complexity class. Although Σ_2^P is a practically important complexity class that characterizes a large number of nonmonotonic reasoning problems, nothing is known on this front to date. In the light of this, the experimental data provided with this paper are a valuable contribution of their own.

Acknowledgments

We thank Wolfgang Faber for insightful discussions on both theoretical issues and the design and implementation of the DLV system. We are indebted to the remaining current and former members of the DLV team, in particular Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Georg Gottlob, Giovambattista Ianni, Giuseppe Ielpa, Cristinel Mateis, Simona Perri, Axel Polleres, and Francesco Scarcello.

References

[1] C. Koch, N. Leone, Stable Model Checking Made Easy, in: T. Dean (Ed.), Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999, Morgan Kaufmann Publishers, Stockholm, Sweden, 1999, pp. 70–75.

[2] M. Gelfond, V. Lifschitz, Classical Negation in Logic Programs and Disjunctive Databases, New Generation Computing 9 (1991) 365–385.

[3] T. C. Przymusinski, Stable Semantics for Disjunctive Programs, New Generation Computing 9 (1991) 401–424.

[4] C. Baral, M. Gelfond, Logic Programming and Knowledge Representation, Journal of Logic Programming 19/20 (1994) 73–148.

[5] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2002.

[6] T. Eiter, G. Gottlob, H. Mannila, Disjunctive Datalog, ACM Transactions on Database Systems 22 (3) (1997) 364–418.

[7] G. Gottlob, Complexity Results for Nonmonotonic Logics, Journal of Logic and Computation 2 (3) (1992) 397–425.

[8] T. Eiter, G. Gottlob, The Complexity of Logic-Based Abduction, Journal of the ACM 42 (1) (1995) 3–42.

[9] T. Eiter, G. Gottlob, On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals, Artificial Intelligence 57 (2–3) (1992) 227–270.

[10] H. Turner, Polynomial-Length Planning Spans the Polynomial Hierarchy, in: S. Flesca, S. Greco, G. Ianni, N. Leone (Eds.), Proceedings of the 8th European Conference on Artificial Intelligence (JELIA), no. 2424 in Lecture Notes in Computer Science, 2002, pp. 111–124.

[11] C. Baral, V. Kreinovich, R. Trejo, Computational Complexity of Planning and Approximate Planning in the Presence of Incompleteness, Artificial Intelligence 122 (1-2) (2000) 241–267.

[12] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and Expressive Power of Logic Programming, ACM Computing Surveys 33 (3) (2001) 374–425.

[13] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello, The KR System `dlv`: Progress Report, Comparisons and Benchmarks, in: A. G. Cohn, L. Schubert, S. C. Shapiro (Eds.), Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Morgan Kaufmann Publishers, 1998, pp. 406–417.

[14] T. Eiter, W. Faber, N. Leone, G. Pfeifer, Declarative Problem-Solving Using the DLV System, in: J. Minker (Ed.), Logic-Based Artificial Intelligence, Kluwer Academic Publishers, 2000, pp. 79–103.

[15] V. Lifschitz, Answer Set Planning, in: D. D. Schreye (Ed.), Proceedings of the 16th International Conference on Logic Programming (ICLP'99), The MIT Press, Las Cruces, New Mexico, USA, 1999, pp. 23–37.

[16] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, M. Barry, An A-Prolog Decision Support System for the Space Shuttle, in: G. Gupta (Ed.), Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), no. 1551 in Lecture Notes in Computer Science, Springer, 1999, pp. 169–183.

[17] T. Soininen, I. Niemelä, Developing a Declarative Rule Language for Applications in Product Configuration, in: G. Gupta (Ed.), Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), no. 1551 in Lecture Notes in Computer Science, Springer, 1999, pp. 305–319.

[18] T. Syrjänen, A Rule-Based Formal Model for Software Configuration, Tech. Rep. A55, Digital Systems Laboratory, Department of Computer Science, Helsinki University of Technology, Espoo, Finland (1999).

[19] K. Heljanko, I. Niemelä, Bounded LTL Model Checking with Stable Models, in: T. Eiter, W. Faber, M. Truszczyński (Eds.), Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings, no. 2173 in Lecture Notes in AI (LNAI), Springer Verlag, 2001, pp. 200–212.

[20] T. Janhunen, I. Niemela, P. Simons, J.-H. You, Partiality and Disjunctions in Stable Model Semantics, in: A. G. Cohn, F. Giunchiglia, B. Selman (Eds.), Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA, Morgan Kaufmann Publishers, Inc., 2000, pp. 411–419.

[21] C. Li, Anbulagan, Heuristics Based on Unit Propagation for Satisfiability Problems, in: Proceedings of the Fourteen International Joint Conference on Artificial Intelligence (IJCAI) 1997, Nagoya, Japan, 1997, pp. 366–371.

[22] J. Lobo, J. Minker, A. Rajasekar, Foundations of Disjunctive Logic Programming, The MIT Press, Cambridge, Massachusetts, 1992.

[23] N. Leone, P. Rullo, F. Scarcello, Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation, Information and Computation 135 (2) (1997) 69–112.

[24] J. Minker, On Indefinite Data Bases and the Closed World Assumption, in: D. Loveland (Ed.), Proceedings 6th Conference on Automated Deduction (CADE '82), no. 138 in Lecture Notes in Computer Science, Springer, New York, 1982, pp. 292–308.

[25] S. Brass, J. Dix, Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation, in: L. Sterling (Ed.), Proceedings of the 12th Int. Conf. on Logic Programming, MIT Press, Tokyo, 1995, pp. 199–213.

[26] T. Przymusinski, Stationary Semantics for Disjunctive Logic Programs and Deductive Databases, in: Proceedings of North American Conference on Logic Programming, 1990, pp. 40–62.

[27] T. Przymusinski, Static Semantics for Normal and Disjunctive Logic Programs, Annals of Mathematics and Artificial Intelligence 14 (1995) 323–357.

[28] K. Ross, The Well-Founded Semantics for Disjunctive Logic Programs, in: W. Kim, J.-M. Nicolas, S. Nishio (Eds.), Deductive and Object-Oriented Databases, Elsevier Science Publishers B. V., 1990, pp. 385–402.

[29] C. Sakama, Possible Model Semantics for Disjunctive Databases, in: Proceedings First Intl. Conf. on Deductive and Object-Oriented Databases (DOOD-89), North-Holland, Kyoto, Japan, 1989, pp. 369–383.

[30] K. Apt, N. Bol, Logic Programming and Negation: A Survey, Journal of Logic Programming 19/20 (1994) 9–71.

[31] J. Dix, Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview, in: Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn'92), DeGruyter, 1995, pp. 241–329.

[32] M. Gelfond, V. Lifschitz, The Stable Model Semantics for Logic Programming, in: Logic Programming: Proceedings Fifth Intl Conference and Symposium, MIT Press, Cambridge, Mass., 1988, pp. 1070–1080.

[33] N. Bidoit, C. Froidevaux, Negation by Default and Unstratifiable Logic Programs, Theoretical Computer Science 78 (1991) 85–112.

[34] F. Buccafurri, N. Leone, P. Rullo, Enhancing Disjunctive Datalog by Constraints, IEEE Transactions on Knowledge and Data Engineering 12 (5) (2000) 845–860.

[35] R. Ben-Eliyahu, R. Dechter, Propositional Semantics for Disjunctive Logic Programs, Annals of Mathematics and Artificial Intelligence 12 (1994) 53–87.

[36] R. Ben-Eliyahu-Zohary, L. Palopoli, Reasoning with Minimal Models: Efficient Algorithms and Applications, Artificial Intelligence 96 (1997) 421–449.

[37] V. Lifschitz, H. Turner, Splitting a Logic Program, in: P. Van Hentenryck (Ed.), Proceedings of the 11th International Conference on Logic Programming (ICLP'94), MIT Press, Santa Margherita Ligure, Italy, 1994, pp. 23–37.

[38] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello, A Deductive System for Nonmonotonic Reasoning, in: Jürgen Dix and Ulrich Furbach and Anil Nerode (Ed.), Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), no. 1265 in Lecture Notes in AI (LNAI), Springer, Berlin, 1997, pp. 363–374.

[39] E. Erdem, Applications of Logic Programming to Planning: Computational Experiments, unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html` (1999).

[40] P. Nicolas, New Generation Systems for Non-Monotonic Reasoning, Presentation at LPNMR'01 (Sep. 2001).

[41] I. Niemelä, P. Simons, T. Syrjänen, Smodels: A System for Answer Set Programming, in: C. Baral, M. Truszczyński (Eds.), Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000), Breckenridge, Colorado, USA, 2000.

[42] I. Niemelä, P. Simons, Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs, in: J. Dix, U. Furbach, A. Nerode (Eds.), Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97), Vol. 1265 of Lecture Notes in AI (LNAI), Springer Verlag, Dagstuhl, Germany, 1997, pp. 420–429.

[43] M. Davis, H. Putnam, A Computing Procedure for Quantification Theory, Journal of the ACM 7 (1960) 201–215.

[44] P. Simons, Smodels Homepage, `<URL:http://www.tcs.hut.fi/Software/smodels/>` (1999).

[45] C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.

[46] T. Eiter, G. Gottlob, On the Computational Cost of Disjunctive Logic Programming: Propositional Case, Annals of Mathematics and Artificial Intelligence 15 (3/4) (1995) 289–323.

[47] I. Gent, T. Walsh, The QSAT Phase Transition, in: Proceedings of the 16th AAAI, 1999.

[48] M. Cadoli, T. Eiter, G. Gottlob, Default Logic as a Query Language, IEEE Transactions on Knowledge and Data Engineering 9 (3) (1997) 448–463.

[49] J. Fernández, J. Minker, Bottom-Up Computation of Perfect Models for Disjunctive Theories, Journal of Logic Programming 25 (1) (1995) 33–51.

[50] J. Dix, M. Müller, Implementing Semantics of Disjunctive Logic Programs Using Fringes and Abstract Properties, in: L.-M. Pereira, A. Nerode (Eds.), Proceedings of the Second International Workshop on Logic Programming and Nonmonotonic Reasoning (LPNMR'93), MIT Press, Lisbon, Portugal, 1993, pp. 43–59.

[51] J. Stuber, Computing Stable Models by Program Transformation, in: P. V. Hentenryck (Ed.), Proc. 11th Int. Conf. on Logic Programming, MIT Press, Santa Margherita Ligure, Italy, 1994, pp. 58–73.

[52] I. Niemelä, P. Simons, Efficient Implementation of the Well-founded and Stable Model Semantics, in: M. J. Maher (Ed.), Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming (ICLP'96), MIT Press, Bonn, Germany, 1996, pp. 289–303.

[53] F. Lin, Y. Zhao, ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002), AAAI Press / MIT Press, Edmonton, Alberta, Canada, 2002.

[54] N. McCain, H. Turner, Satisfiability Planning with Causal Theories, in: A. G. Cohn, L. Schubert, S. C. Shapiro (Eds.), Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Morgan Kaufmann Publishers, 1998, pp. 212–223.

[55] Y. Babovich, Cmodels homepage, `http://www.cs.utexas.edu/users/tag/cmodels.html` (2002).

[56] D. East, M. Truszczyński, dcs: An Implementation of DATALOG with Constraints, in: C. Baral, M. Truszczyński (Eds.), Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000), Breckenridge, Colorado, USA, 2000.

[57] P. Cholewiński, V. W. Marek, M. Truszczyński, Default Reasoning System DeReS, in: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR '96), Morgan Kaufmann Publishers, Cambridge, Massachusetts, USA, 1996, pp. 518–528.

[58] D. Seipel, H. Thöne, DisLog – A System for Reasoning in Disjunctive Deductive Databases., in: A. Olivé (Ed.), Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94), Universitat Politecnica de Catalunya (UPC), 1994, pp. 325–343.

[59] C. Aravindan, J. Dix, I. Niemelä, DisLoP: A Research Project on Disjunctive Logic Programming, AI Communications – The European Journal on Artificial Intelligence 10 (3/4) (1997) 151–165.

[60] C. Anger, K. Konczak, T. Linke, `NoMoRe`: A System for Non-Monotonic Reasoning, in: T. Eiter, W. Faber, M. Truszczyński (Eds.), Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings, no. 2173 in Lecture Notes in AI (LNAI), Springer Verlag, 2001, pp. 406–410.

[61] U. Egly, T. Eiter, H. Tompits, S. Woltran, Solving Advanced Reasoning Tasks using Quantified Boolean Formulas, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA, AAAI Press / MIT Press, 2000, pp. 417–422.

[62] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, J. Freire, XSB: A System for Efficiently Computing Well-Founded Semantics, in: J. Dix, U. Furbach, A. Nerode (Eds.), Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97), no. 1265 in Lecture Notes in AI (LNAI), Springer Verlag, Dagstuhl, Germany, 1997, pp. 2–17.

[63] N. Leone, C. Pizzuti, P. Rullo, Efficient Evaluation of a Class of Ordered Logic Programs, Data and Knowledge Engineering 23 (2) (1997) 185–213.

Appendix

The $\mathcal{R}_{\mathcal{P},M}$ operator of Definition 3.8 efficiently (more specifically, in linear time [36]) evaluates the stability condition for HCF programs and usually greatly reduces the set of possible unfounded sets that need to be checked for non-HCF programs. In this appendix, we discuss some interesting properties of this operator. We first show that the definition of $\mathcal{R}_{\mathcal{P},M}$ for simplified programs $\alpha_M(\mathcal{P})$ is simpler and more intuitive than the original one.

Definition 7.1 Let $\alpha_M(\mathcal{P})$ be the simplified version of a program $\mathcal{P}$, as described in Definition 4.3. We define the $\mathcal{R}_{\alpha_M(\mathcal{P}),M}$ operator as follows:

$$\begin{aligned} \mathcal{R}_{\alpha_M(\mathcal{P}),M} \colon 2^{B_\mathcal{P}} &\to 2^{B_\mathcal{P}} \\ X &\mapsto \{a \in X \mid \nexists r \in \alpha_M(\mathcal{P}) \\ &\qquad \text{with } (H(r) = \{a\}) \wedge (B(r) \cap X = \emptyset)\ \} \end{aligned} \qquad \Box$$

Note that we use a set-based notation for $H(r)$, and trivial disjunctions of the type $a \vee \ldots \vee a$ in heads are of course only represented by a single occurrence of a in $H(r)$. Further below, we will assume without loss of generality that such trivial disjunctions do not occur in our programs.

Proposition 7.2 *Let $\mathcal{P}$ be a program and M a model of $\mathcal{P}$. Then,*

$$\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \mathcal{R}^{\omega}_{\alpha_M(\mathcal{P}),M}(M).$$

Example 7.3 $\mathcal{P} = \{a \vee b;\ c \leftarrow b\}$, $M = \{b, c\}$. $\alpha_M(\mathcal{P}) = \{b;\ c \leftarrow b\}$. In the first iteration of $\mathcal{R}_{\alpha_M(\mathcal{P}),M}$, b is removed. In the second, c is deleted and $\mathcal{R}^{\omega}_{\alpha_M(\mathcal{P}),M}(M) = \mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \emptyset$. Thus, M is a stable model of $\mathcal{P}$. $\Box$

Proof. Suppose $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) \neq \mathcal{R}^{\omega}_{\alpha_M(\mathcal{P}),M}(M)$. Then, the expressions $(\forall r \in \mathit{ground}(\mathcal{P})$ with $a \in H(r), (B(r)$ is false w.r.t. $M) \vee (B(r) \cap X \neq \emptyset) \vee ((H(r) - \{a\}) \cap M \neq \emptyset))$ and $(\nexists r \in \alpha_M(\mathcal{P})$ with $(B(r) \cap X = \emptyset) \wedge (H(r) = \{a\}))$ must not be equivalent. The first expression can be rewritten as $\nexists r \in \mathit{ground}(\mathcal{P})$ with $a \in H(r) \wedge (B(r)$ is true w.r.t. $M) \wedge (B(r) \cap X = \emptyset) \wedge (H(r) \cap M = \{a\})$. We know that in $\alpha_M(\mathcal{P})$ every rule body is true w.r.t. M and $H(r) \subseteq M$. Hence, the two expressions above are equivalent and $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) = \mathcal{R}^{\omega}_{\alpha_M(\mathcal{P}),M}(M)$. $\Box$

It is easy to see that the $\mathcal{R}_{\alpha_M(\mathcal{P}),M}$ operator of Definition 7.1 is the converse of the classical direct consequence operator $\mathcal{T}_\mathcal{P}$. Starting from "facts" in $\alpha_M(\mathcal{P})$, $\mathcal{T}_{\alpha_M(\mathcal{P})}$ computes all atoms in $M - \mathcal{R}^{\omega}_{\mathcal{P},M}(M)$ (which certainly cannot be in any unfounded set), i.e., $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) \cup \mathcal{T}^{\omega}_{\alpha_M(\mathcal{P})}(\emptyset) = M$ and $\mathcal{R}^{\omega}_{\mathcal{P},M}(M) \cap \mathcal{T}^{\omega}_{\alpha_M(\mathcal{P})}(\emptyset) = \emptyset$. Finally, note that if we abbreviate $\mathcal{R}^{\omega}_{\mathcal{P},M}(M)$ as X, we have $\mathcal{R}^{\omega}_{\alpha_{M,X}(\mathcal{P})}(X) = X$. Therefore, we cannot obtain an even better (that is, smaller) fixpoint by iteratively applying $\alpha_{M,X}(\mathcal{P})$ and the $\mathcal{R}$ operator.