Testing the Equivalence of Logic Programs under Stable Model Semantics*

Tomi Janhunen and Emilia Oikarinen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
`Tomi.Janhunen@hut.fi, Emilia.Oikarinen@hut.fi`

Abstract. Solving a problem in the answer set programming approach means constructing a logic program so that the answer sets of the program correspond to the solutions to the problem. Typically, a programmer develops a series of improved formulations for a particular problem. Consequently, the programmer is confronted by another problem, namely ensuring that subsequent formulations are equivalent, i.e., give rise to the same answer sets. To ease answer set programming, we propose a methodology for testing the equivalence of logic programs. The basic idea is to translate the logic programs P and Q under consideration into a single logic program R whose answer sets (if such exist) yield counter-examples to the equivalence of P and Q. The translation function presented in the paper has been implemented as a translator program LPEQ that enables the equivalence testing of logic programs using the SMODELS system. Experiments performed with LPEQ and SMODELS suggest that establishing the equivalence of logic programs in this way is in certain cases much faster than explicit cross-checking of answer sets.

1 Introduction

Answer set programming (ASP) has recently been promoted as a self-standing logic programming paradigm [8, 9]. Indeed, the paradigm is receiving increasing attention and numerous applications have emerged since efficient implementations such as DLV [4] and SMODELS [12] became available in the late nineties. It is interesting to note that the current applications of ASP are already quite diverse ranging, e.g., from wire routing to a decision support system of the space shuttle (c.f. the topics of the 2001 AAAI Spring Symposium on ASP).

ASP is a very appealing approach because of its declarative nature, but it shares some problems with conventional approaches regarding the development of programs. Although logic programs lend themselves to concise encodings of the problems being solved, such encodings are by no means unique. Typically, a

* The research reported in this paper is affiliated with the project "Applications of Rule-Based Constraint Programming" (#53695) funded by the Academy of Finland.

programmer ends up with a series of improved programs for a particular problem, e.g., when optimizing the execution time and space. In this setting, there is a need to ensure that subsequent programs that differ in performance yield the same output. In case of ASP, this means establishing that given two logic programs P and Q give rise to the same answer sets, i.e., P and Q are semantically equivalent. This is the weaker notion of equivalence addressed by Lifschitz et al. [5]. In addition, they introduce a stronger notion of equivalence: P and Q are *strongly equivalent* if P and Q yield same answer sets in all possible contexts (i.e., as parts of any other programs). The strong equivalence can be characterized as equivalence in the logic of *here-and-there* (HT). As later shown by Pearce et al. [13], satisfiability in HT is reducible to propositional satisfiability (SAT), which implies that existing SAT solvers can be used for testing the strong equivalence of logic programs. Quite recently, Lin [6] reduces strong equivalence directly to propositional entailment without using HT as an intermediate logic. Unfortunately, these theoretical results lack an experimental evaluation to verify that the transformations are computationally feasible in practice.

The aim of this paper is to characterize the equivalence of logic programs directly in logic programming terms so that existing ASP implementations can be used for computations. For now we have developed a translation that captures the weak equivalence of logic programs supported by the SMODELS system as follows: the translation of P and Q has an answer set if and only if P has an answer set that is not an answer set of Q. This enables the search of counter-examples to the equivalence of P and Q using the translation and SMODELS.

The rest of the paper is organized as follows. Section 2 reminds the reader about the various kinds of rules supported by the current SMODELS system. It is then explained in Section 3 how the semantics of the rules is encompassed by stable models proposed by Gelfond and Lifschitz [2]. In Section 4, the translation for the equivalence testing of logic programs is presented. The correctness of the translation is also addressed. Results from experiments performed with an implementation (a translator called LPEQ) and SMODELS are reported in Section 5. The results indicate that in certain cases equivalence testing with LPEQ is one or two orders of magnitude faster than naive cross-checking of stable models. Finally, we end the paper with a brief conclusion in Section 6.

2 The SMODELS Language

Definition 1 lists the forms of rules supported by the current SMODELS system [14]. Besides *basic rules* (1) of conventional logic programs, there are also other rules available: *constraint rules* (2), *choice rules* (3), and *weight rules* (4).

Definition 1. *Rules are expressions of the forms*

$$h \leftarrow a_1, \ldots, a_n, \sim b_1, \ldots, \sim b_m \tag{1}$$

$$h \leftarrow k \; \{a_1, \ldots, a_n, \sim b_1, \ldots, \sim b_m\} \tag{2}$$

$$\{h_1, \ldots, h_l\} \leftarrow a_1, \ldots, a_n, \sim b_1, \ldots, \sim b_m \tag{3}$$

$$h \leftarrow w \leq \{a_1 = w_1, \ldots, a_n = w_n, \sim b_1 = w_{n+1}, \ldots, \sim b_m = w_{n+m}\} \tag{4}$$

where $n \geq 0$, $m \geq 0$, and $l > 0$, and where h, $a_1, \ldots, a_n$, $b_1, \ldots, b_m$, and $h_1, \ldots, h_l$ are atoms and k, $w_1, \ldots, w_{n+m}$, as well as w, are natural numbers.

The symbol $\sim$ in Definition 1 denotes *default negation* or *negation as failure* to prove [1]. We define *default literals* in the standard way as atoms a or their negations $\sim a$. The exact model-theoretic semantics of rules is deferred to Section 3, but – informally speaking – the rules can be used for inference as follows. The head h of a basic rule can be inferred if the atoms $a_1, \ldots, a_n$ are inferable and the atoms $b_1, \ldots, b_m$ are *not* inferable. The head h of a constraint rule can be inferred if the number of inferable atoms among $a_1, \ldots, a_n$ plus the number of non-inferable atoms among $b_1, \ldots, b_m$ is at least k. Note that a constraint rule becomes equivalent to a basic rule if $k = n + m$. A choice rule is similar to a basic rule except that any subset (even the empty set) of the set of head atoms $\{h_1, \ldots, h_l\}$ can be inferred instead of a single atom h. A weight rule involves summing as follows: the weight w_i of a_i (w_i of $\sim b_i$) is one of the summands if and only if a_i is inferable (b_i is not inferable). The head h can be inferred if such a sum of weights is at least w. A weight rule reduces to a constraint rule when $w_1 = 1, \ldots, w_{n+m} = 1$, and $w = k$. Note that default literals may be assigned different weights in different weight rules, i.e., weights are local.

The four types of rules defined above are already well-suited for a variety of knowledge representation and reasoning tasks in a number of domains. Example 1 demonstrates the use of rules in a practical setting. The reader is referred to [9–11] for more examples and for more general rules that can be captured by the rules supported by the SMODELS system using suitable transformations.

Example 1. Consider describing coffee orders by the following rules[1]. An order is acceptable if and only if the atom $\mathsf{acceptable}$ can be inferred using the rules.

$\{\mathsf{coffee}, \mathsf{tea}, \mathsf{biscuit}, \mathsf{cake}, \mathsf{cognac}\}$. $\{\mathsf{cream}, \mathsf{sugar}\} \leftarrow \mathsf{coffee}$. $\mathsf{cognac} \leftarrow \mathsf{coffee}$.
$\{\mathsf{milk}, \mathsf{lemon}, \mathsf{sugar}\} \leftarrow \mathsf{tea}$. $\mathsf{coagulation} \leftarrow \mathsf{milk}, \mathsf{lemon}$.
$\mathsf{satisfied} \leftarrow 1\ \{\mathsf{biscuit}, \mathsf{cake}, \mathsf{cognac}\}$.
$\mathsf{bankrupt} \leftarrow 6 \leq \{\mathsf{coffee} = 1, \mathsf{tea} = 1, \mathsf{biscuit} = 1, \mathsf{cake} = 2, \mathsf{cognac} = 4\}$.
$\mathsf{acceptable} \leftarrow \mathsf{satisfied}, \sim\mathsf{bankrupt}, \sim\mathsf{coagulation}$.

A *logic program* P is a finite set of rules of the forms (1)–(4). We write $\mathrm{Hb}(P)$ for the Herbrand base of P, i.e., the set of atoms that appear in P. In addition to the four types of rules described above, logic programs supported by the SMODELS system may contain additional control information, namely *compute statements* and *maximize/minimize statements*. The exact syntax of this control information is not important for our purposes, and for this reason these aspects of SMODELS programs are addressed in later sections of this paper.

3 Stable Model Semantics

The semantics of *normal logic programs* (i.e., logic programs that consist solely of basic rules) is determined by *stable models* proposed by Gelfond and Lifschitz [2].

[1] Rules are separated with full stops and the symbol "$\leftarrow$" is dropped from a basic rule (1) or a choice rule (3) if the *body* of the rule is empty ($n = 0$ and $m = 0$).

The idea is to reduce a normal logic program P with respect to a model candidate M by pre-interpreting negative literals that appear in the rules. The resulting program P_M – also known as the Gelfond-Lifschitz *reduct* of P – is *positive*, i.e., free of default negation. A positive program P has a well-established semantics determined by the *least model* of P [7]. The least model semantics extends for normal logic programs in a straightforward way: a model $M \subseteq \mathrm{Hb}(P)$ is stable if and only if M is the least model of P_M. Stable models need not be unique: a normal logic program P may possess several stable models or no stable model at all. However, this is not considered as a problem in answer set programming, since the aim is to capture solutions to the problem at hand with the stable models of a program constructed for the problem.

Simons [14] shows how the principles behind stable models can be used for defining the semantics of other rules presented in Section 2. However, the reduced program is not explicitly present in the semantical definitions given by Simons. This is why we resort to an alternative definition (see Definition 3 below). The forthcoming definition can be understood as a special case of that given by Niemelä and Simons [10] for more general classes of rules. In contrast to their definitions that involve deductive closures of rules, our intention is to formulate the stable model semantics purely in model-theoretic terms.

Given a logic program P, an *interpretation* is a subset of $\mathrm{Hb}(P)$ defining which atoms a are considered to be *true* ($a \in I$) and which *false* ($a \notin I$). Given an interpretation $I \subseteq \mathrm{Hb}(P)$, an atom $a \in \mathrm{Hb}(P)$ is *true/satisfied* in I (denoted by $I \models a$) $\iff$ $a \in I$. Negative literals $\sim a$ are given a classical interpretation at this point: $I \models \sim a \iff I \not\models a$ ($\iff a \notin I$). The satisfaction relation $I \models r$ is defined for the rules r under consideration as follows.

Definition 2. *Given an interpretation $I \subseteq \mathrm{Hb}(P)$ for a logic program P,*

- *a basic rule (1) is satisfied in I*
  $\iff I \models a_1, \ldots, I \models a_n$ *and* $I \models \sim b_1, \ldots, I \models \sim b_m$ *imply* $I \models h$,
- *a constraint rule (2) is satisfied in I*
  $\iff k \leq |\{a_i \mid I \models a_i\} \cup \{\sim b_j \mid I \models \sim b_j\}|$ *implies* $I \models h$,
- *a choice rule (3) is always satisfied in I, and*
- *a weight rule (4) is satisfied in I*
  $\iff I \models h$ *is implied by* $w \leq \sum_{I \models a_i} w_i + \sum_{I \models \sim b_j} w_{n+j}$.

An interpretation I is a *model* of a logic program P (denoted by $I \models P$) if and only if $I \models r$ for every $r \in P$. However, stable models are not arbitrary models of logic programs. As hinted in the beginning of this section, the stable model semantics involves a reduction of logic programs which is based on a pre-interpretation of negative literals. The following definition gives the Gelfond-Lifschitz reduction for the four rule types. Note that in addition to valuating negative literals in the bodies of rules, the head atoms $h \in \{h_1, \ldots, h_l\}$ of choice rules (3) are subject to a special treatment: an essential prerequisite for including $h \leftarrow a_1, \ldots, a_n$ in the reduct P_M is that $M \models h$, i.e., $h \in M$. This is the way in which the choice regarding h takes place technically.

Definition 3. *For a logic program P and $M \subseteq \mathrm{Hb}(P)$, the reduct P_M contains*

- *a basic rule $h \leftarrow a_1,\ldots,a_n \iff$ there is a basic rule (1) in P such that $M \models {\sim}b_1, \ldots, M \models {\sim}b_m$,* ***or*** *there is a choice rule (3) in P such that $h \in \{h_1, \ldots, h_l\}$, $M \models h$, and $M \models {\sim}b_1, \ldots, M \models {\sim}b_m$.*
- *a constraint rule $h \leftarrow k' \ \{a_1,\ldots,a_n\} \iff$ there is a constraint rule (2) in P and $k' = \max(0, k - |\{{\sim}b_i \mid M \models {\sim}b_i\}|)$,*
- *a weight rule $h \leftarrow w' \leq \{a_1 = w_1,\ldots,a_n = w_n\} \iff$ there is a weight rule (4) in P and $w' = \max(0, w - \sum_{M \models {\sim}b_j} w_{n+j})$.*

It is clear by Definition 3 that the reduct P_M is free of default negation and it contains only basic rules, constraint rules and weight rules. Thus we call a logic program P *positive* if each rule $r \in P$ is of the forms (1), (2) and (4) restricted to the case $m = 0$. The least model semantics can be generalized for this particular class of programs as follows. A model $M \models P$ is *minimal* (with respect to subset inclusion) if and only if there is no $M' \subset M$ such that $M' \models P$. Within the class of positive programs under consideration, every program P is guaranteed to have a unique minimal model $\mathrm{LM}(P)$ which is the *least model* of P. Moreover, the intersection $I = \bigcap\{M \subseteq \mathrm{Hb}(P) \mid M \models P\}$ is also a model of P and $I = \mathrm{LM}(P)$.

Now the semantics of a logic program P with default negation can be defined very concisely: M is a stable model of P if and only if $M = \mathrm{LM}(P_M)$. Note that a positive program P has a unique stable model $\mathrm{LM}(P)$, as $P_M = P$ holds always. It is also worth noting that $M = \mathrm{LM}(P_M)$ implies $M \models P$ in the sense of Definition 2. However, the converse does not hold in general, i.e., $M \models P$ need not imply $M = \mathrm{LM}(P_M)$. E.g., interpretations $M_1 = \{a\}$ and $M_2 = \{a, b\}$ are models of $P = \{a \leftarrow 1 \ \{{\sim}a, {\sim}b\}\}$, but only the former model is a stable one.

Example 2. Recall the set of rules P given in Example 1. According to SMODELS there are 33 acceptable orders, i.e., stable models of P containing $\mathsf{acceptable}$. One of them is $M_7 = \{\mathsf{acceptable}, \mathsf{satisfied}, \mathsf{lemon}, \mathsf{tea}, \mathsf{biscuit}\}$. The reader may compute P_{M_7} using Definition 3 and verify that $M_7 = \mathrm{LM}(P_{M_7})$ is the case.

4 Translation for Equivalence Testing

Lifschitz et al. [5] address two major notions of equivalence for logic programs. The first one arises naturally from the stable model semantics. Logic programs P and Q are *equivalent* (denoted by $P \equiv Q$) if and only if P and Q have exactly the same stable models. The second notion is definable in terms of the first: P and Q are *strongly equivalent* (denoted by $P \equiv_{\mathrm{s}} Q$) if and only if $P \cup R \equiv Q \cup R$ for all logic programs R. Here R can be understood as an arbitrary context in which P and Q could be placed. This is how strongly equivalent logic programs are semantics preserving substitutes of each other and the relation $\equiv_{\mathrm{s}}$ can be understood as a congruence relation among logic programs. It is easy to see that $P \equiv_{\mathrm{s}} Q$ implies $P \equiv Q$ but not vice versa: $\equiv_{\mathrm{s}}$ relates fewer programs than $\equiv$.

In this paper, we concentrate on developing an implementation for the relation $\equiv$ and leave $\equiv_{\mathrm{s}}$ as future work. Roughly speaking, our idea is to translate given logic programs P and Q into a single logic program $\mathrm{EQT}(P, Q)$ which has a stable model if and only if P has a stable model M which is *not* a stable model of Q. This is how our aim is to use the translation for finding a counter-example for the relationship $P \equiv Q$. However, in order to support the compute statements of SMODELS programs mentioned in Section 2, we have to cover a slightly more general setting. Using a compute statement, a programmer can specify additional constraints on the stable models of P to be computed. This is achieved by declaring a set of default literals L which is construed as follows. An interpretation I is *compatible* with L if and only if (i) $a \in I$ for each $a \in L$ and (ii) $a \notin I$ for each $\sim a \in L$. Given a logic program P and a set of default literals L based on $\mathrm{Hb}(P)$, a *constrained (logic) program* $P[L]$ is understood to have a stable model M if and only if M is a stable model of P and M is compatible with L. Note that compute statements can be avoided (if necessary) by adding extra rules to the program. Given a new atom $f \notin \mathrm{Hb}(P)$, the stable models of $P[L]$ are captured by $P' = P \cup \{f \leftarrow a, \sim f \mid \sim a \in L\} \cup \{f \leftarrow \sim a, \sim f \mid a \in L\}$. Nevertheless, we have to take compute statements into account in order to cover arbitrary SMODELS programs. For instance, a typical *headless* rule $\leftarrow a, \sim b$, which is supported by the front-end LPARSE, is passed to SMODELS as a rule $f \leftarrow a, \sim b$ accompanied by a compute statement which requires compatibility with $\{\sim f\}$.

We are now ready to formulate a translation $\mathrm{EQT}(P, Q, L)$ involving two constrained programs $P[K]$ and $Q[L]$ satisfying $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$.[2] Note that K plays no role in $\mathrm{EQT}(P, Q, L)$, but the stable models of $\mathrm{EQT}(P, Q, L)[K]$ will be of our interest later on. We introduce a new atom $a^\star$ for each $a \in \mathrm{Hb}(Q)$. Such atoms are additional (renamed) copies of atoms that are needed to compute $\mathrm{LM}(Q_M)$ given a stable model M of $P[K]$. We write $A^\star = \{a^\star \mid a \in A\}$ for any set of atoms A and $P^\star$ denotes a *positive* program P whose atom occurrences a have been systematically replaced by the respective new atoms $a^\star$.

Definition 4. *Let P be a logic program P and $Q[L]$ a constrained program such that $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$. Let c, d, and e be new atoms not appearing in P nor Q. The translation $\mathrm{EQT}(P, Q, L)$ contains the following rules:*

1. *all rules of P without modifications,*
2. *a rule $h^\star \leftarrow a_1^\star, \ldots, a_n^\star, \sim b_1, \ldots, \sim b_m$ for each basic rule (1) in Q,*
3. *a rule $h^\star \leftarrow k\ \{a_1^\star, \ldots, a_n^\star, \sim b_1, \ldots, \sim b_m\}$ for each constraint rule (2) in Q,*
4. *a rule $h_i^\star \leftarrow a_1^\star, \ldots, a_n^\star, h_i, \sim b_1, \ldots, \sim b_m$ for each choice rule (3) in Q and head atom $h_i \in \{h_1, \ldots, h_l\}$,*
5. *a rule $h^\star \leftarrow w \leq \{a_1^\star = w_1, \ldots, a_n^\star = w_n, \sim b_1 = w_{n+1}, \ldots, \sim b_m = w_{n+m}\}$ for each weight rule (4) in Q,*
6. *a rule $c \leftarrow \sim a$ for each $a \in L$ and a rule $c \leftarrow a$ for each $\sim a \in L$,*
7. *rules $d \leftarrow a, \sim a^\star$ and $d \leftarrow a^\star, \sim a$ for each $a \in \mathrm{Hb}(P) = \mathrm{Hb}(Q)$,*
8. *rules $e \leftarrow c$ and $e \leftarrow d$.*

Note that $\mathrm{Hb}(\mathrm{EQT}(P, Q, L))$ equals to $\mathrm{Hb}(P) \cup \mathrm{Hb}(Q)^\star \cup \{c, d, e\}$. Moreover, the translation is close to being linear. The fourth item in Definition 4 makes

[2] Note that this restriction is not significant, as it is possible to extend any logic program P with useless rules $a \leftarrow a$ without affecting the stable models of P

an exception in this respect, but linearity can be achieved by introducing a new atom b_r for each choice rule r. Then the rules in the fourth item can be replaced by $h_i^\star \leftarrow h_i, b_r$ ($h_i \in \{h_1, \ldots, h_l\}$) and $b_r \leftarrow a_1^\star, \ldots, a_n^\star, {\sim}b_1, \ldots, {\sim}b_m$. However, we use the current definition to simplify the correctness proof.

The rules in the translation $\mathrm{EQT}(P, Q, L)$ serve the following purposes. The program P is included as such, since the goal is to compute a stable model M for $P[K]$. The remaining rules provide means to check that M is not a stable model of $Q[L]$. Rules from the second item to the fifth capture $\mathrm{LM}(Q_M)$ but expressed in $\mathrm{Hb}(Q)^\star$ rather than $\mathrm{Hb}(Q)$. The rules in the sixth item check if M is not compatible with L. The rules in the seventh item check if M and $\mathrm{LM}(Q_M)$ differ. The last two rules summarize the reasons for M **not** being a stable model of $Q[L]$. The correctness of the translation $\mathrm{EQT}(P, Q, L)$ is addressed next.

Lemma 1. *Let P be a logic program and $Q[L]$ a constrained program such that $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$ and I an interpretation for the translation $\mathrm{EQT}(P, Q, L)$. Define $J_1 = I \cap \mathrm{Hb}(P)$ and $J_2 = \{a \in \mathrm{Hb}(Q) \mid a^\star \in I\}$ so that ${J_2}^\star = I \cap \mathrm{Hb}(Q)^\star$. The reduct $\mathrm{EQT}(P, Q, L)_I$ consists of the following rules:*

- *the rules of P_{J_1} as such,*
- *a rule $h^\star \leftarrow a_1^\star, \ldots, a_n^\star$*
  $\iff$ there is a basic rule (1) in Q such that $J_1 \models {\sim}b_1, \ldots, J_1 \models {\sim}b_m$,
- *for each constraint rule (2) in Q, a rule $h^\star \leftarrow k' \ \{a_1^\star, \ldots, a_n^\star\}$ with $k' = \max(0, k - |\{{\sim}b_j \mid J_1 \models {\sim}b_j\}|)$,*
- *a rule $h_i^\star \leftarrow a_1^\star, \ldots, a_n^\star, h_i$*
  $\iff$ there is a choice rule (3) in Q such that $J_1 \models {\sim}b_1, \ldots, J_1 \models {\sim}b_m$,
- *for each weight rule (4) in Q, a rule $h^\star \leftarrow w' \leq \{a_1^\star = w_1, \ldots, a_n^\star = w_n\}$ with $w' = \max(0, w - \sum_{J_1 \models {\sim}b_j} w_{n+j})$,*
- *the rule $c \leftarrow \iff$ there is $a \in L$ such that $a \notin J_1$,*
- *a rule $c \leftarrow a$ for each ${\sim}a \in L$,*
- *a rule $d \leftarrow a \iff$ there is $a \in \mathrm{Hb}(Q)$ such that $a \notin J_2$,*
- *a rule $d \leftarrow a^\star \iff$ there is $a \in \mathrm{Hb}(P)$ such that $a \notin J_1$, and*
- *the rules $e \leftarrow c$ and $e \leftarrow d$.*

Lemma 1 can be easily verified by inspecting the definition of $\mathrm{EQT}(P, Q, L)$ (Definition 4), the definitions of J_1 and J_2 and the generalization of Gelfond-Lifschitz reduct for the various rule types (Definition 3). Proposition 1 (see below) lists the essential properties of $\mathrm{LM}(\mathrm{EQT}(P, Q, L)_I)$, which are sufficient to establish the main result of this paper given as Theorem 1 (proofs are omitted due to space restrictions). As a corollary of Theorem 1, we obtain a new method for testing whether the equivalence $P[K] \equiv Q[L]$ holds or not.

Proposition 1. *Assume the same setting and definitions as in Lemma 1.*

(i) *The intersection $\mathrm{LM}(\mathrm{EQT}(P, Q, L)_I) \cap \mathrm{Hb}(P) = \mathrm{LM}(P_{J_1})$.*
(ii) *If $J_1 = \mathrm{LM}(P_{J_1})$, then $\mathrm{LM}(\mathrm{EQT}(P, Q, L)_I) \cap \mathrm{Hb}(Q)^\star = \mathrm{LM}(Q_{J_1})^\star$.*
(iii) *If $J_1 = \mathrm{LM}(P_{J_1})$ and $J_2 = \mathrm{LM}(Q_{J_1})$, then $A = \mathrm{LM}(\mathrm{EQT}(P, Q, L)_I) \cap \{c, d, e\}$ satisfies (1) $c \in A \iff J_1$ is not compatible with L, (2) $d \in A \iff J_1 \neq J_2$, and (3) $e \in A \iff c \in A$ or $d \in A$.*

Theorem 1. *For any constrained programs $P[K]$ and $Q[L]$ satisfying* $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$*, the constrained translation* $\mathrm{EQT}(P, Q, L)[K \cup \{e\}]$ *has a stable model* $\iff$ *$P[K]$ has a stable model M which is not a stable model of $Q[L]$.*

Corollary 1. *For any constrained logic programs $P[K]$ and $Q[L]$ with* $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$*, $P[K] \equiv Q[L]$* $\iff$ *the constrained translations* $\mathrm{EQT}(P, Q, L)[K \cup \{e\}]$ *and* $\mathrm{EQT}(Q, P, K)[L \cup \{e\}]$ *have no stable models.*

5 Experiments

The translation function EQT presented in Section 4 has been implemented in the C programming language under the Linux operating system. The translator – which we have named as LPEQ – takes two logic programs as its input and produces the translation as its output. The program assumes the internal format of SMODELS which enables using the front-end LPARSE of SMODELS in conjunction with LPEQ. Moreover, the translation is printed in a textual form upon request. The current implementation of LPEQ is available in the World Wide Web (`http://www.tcs.hut.fi/Software/lpeq/`) for testing purposes. Some examples related with our experiments are also provided. It is yet important to note that LPEQ checks that the *visible* Herbrand bases of the programs being compared are exactly the same (a visible atom has a name in the symbol table of the program). However, the front-end LPARSE may produce some *invisible* atoms that are not taken into account in this comparison. To support programs produced by LPARSE, such atoms keep their roles in the respective programs.

To assess the feasibility of LPEQ in practice we ran various tests with different test cases. The running times of the LPEQ approach were compared with those of a fictitious approach: in the NAIVE approach, one (i) generates a stable model M of P, (ii) tests whether M is a stable model of Q, (iii) stops if not, and otherwise continues from step (i) until all stable models of P get enumerated. It is obvious that a similar testing has to be carried out in the other direction to establish $P \equiv Q$ (in analogy to Corollary 1). There is still room for optimization in both approaches. If one finds a counter-example in one direction, then $P \not\equiv Q$ is known to hold and there is no need to do testing in the other direction except if one wishes to perform a thorough analysis. However, we decided not to follow this optimization principle, because running times turned out to scale differently depending on the direction. Thus we counted running times in both directions.

In both approaches, the SMODELS system (version 2.26) was responsible for the computation of stable models. In the LPEQ approach, the total running time (in one direction) is the running time needed by SMODELS for trying to compute *one* stable model of the translation produced by LPEQ. The actual translation time is not taken into account, as it is negligible. In the NAIVE approach, we tried to exclude possible overhead due to implementing the NAIVE approach as a shell script. Thus the total running time (in one direction) consists solely of the running time of SMODELS for finding the necessary (but not necessarily all) stable models of P plus the individual running times of SMODELS for testing that

n	stable models	t_{ave} (s) LPEQ	t_{ave} (s) NAIVE	t_{ave} NAIVE/ t_{ave} LPEQ	choices LPEQ	choices NAIVE	$\lvert\text{EQT}(Q_n, Q'_n)\rvert+$ $\lvert\text{EQT}(Q_n, Q'_n)\rvert$	$\lvert Q_n\rvert+$ $\lvert Q'_n\rvert$
1	1	0.000	0.024	-	0	0	49	13
2	0	0.015	0.008	0.533	0	0	194	68
3	0	0.017	0.013	0.780	0	0	521	205
4	2	0.025	0.057	2.280	0	2	1110	464
5	10	0.124	0.285	2.298	4	18	2041	885
6	4	0.392	0.276	0.705	14	16	3394	1508
7	40	1.439	2.404	1.671	44	84	5249	2373
8	92	5.415	8.117	1.499	158	245	7686	3520
9	352	24.455	41.501	1.697	598	962	10785	4989
10	724	118.848	132.319	1.113	2525	3036	14626	6820
11	2680	647.307	672.410	1.039	12144	13792	19289	9053
12	14200	3808.970	4364.230	1.146	59759	69329	24854	11728

Table 1. Results for two equivalent logic programs (n-queens).

the stable models found are also stable models of Q. These tests were realized in practice by adding M as a compute statement to Q. All the tests were run under the Linux 2.2.13/4 operating system on 450MHz Pentium III computers.

The first experiment was based on the program Q_n formulated by Niemelä [9, p. 260] to solve the n-queens problem. The second program Q'_n is a variant of Q_n where the choice between placing or not placing a queen in a particular cell is equivalently formulated using choice rules and constraint rules rather than basic rules. We tested the equivalence of these two programs using both the LPEQ and the NAIVE approach. The number n of queens was varied from 1 to 12 and the equivalence test was repeated 10 times for each number of queens to eliminate any inaccuracies caused by the system clock. The results of this experiment are shown in Table 1. In case of two equivalent well-structured logic programs the performance of the LPEQ approach is somewhat better than that of NAIVE approach. Also the number of *choice points* (i.e., the number of choices made by SMODELS while searching for stable models) is slightly smaller in the LPEQ approach than in the NAIVE approach. This is a further indication that verifying the equivalence of logic programs is easier using the LPEQ approach.

We also performed some tests with random logic programs. We generated logic programs that solve an instance of a random 3-sat problem with a constant clauses to variables ratio $c/v = 4$. Such instances are typically satisfiable, but so close to the *phase transition point* (approximately 4.3) that finding models is already demanding for satisfiability solvers. To simulate a sloppy programmer making mistakes, we dropped one random rule from each program. Then we tested the equivalence of the modified program and the original program to see if making a mistake affects the stable models of the program or not. As a consequence, the tested pairs of programs included both equivalent and nonequivalent cases. In our first experiment with random 3-sat programs, we varied the number of variables v from 10 to 50 with steps of 5. For each number of variables we

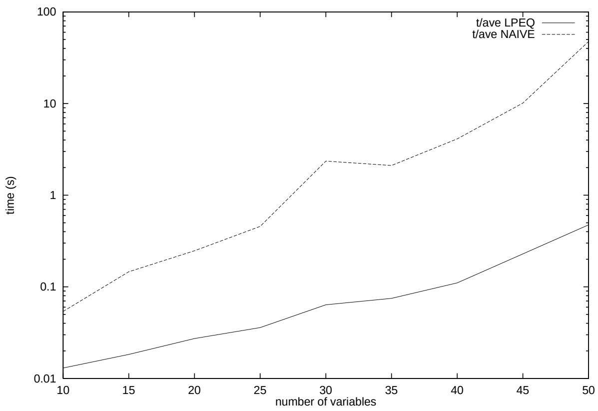

Fig. 1. The equivalence testing with random 3-sat instances with the ratio $c/v = 4$.

repeated the test 100 times – generating each time a new random instance. The average running times of both approaches are plotted in Figure 1. The LPEQ approach turned out to be significantly faster than the NAIVE one. Moreover, the difference in running times increases as the sizes of the programs grow.

In our second experiment with random 3-sat instances we generated the programs as in the previous experiment, but we kept the number of variables constant, $v = 40$, and varied the ratio c/v from 3.75 to 4.75 with steps of 0.125. For each value of the ratio c/v, we repeated the test 100 times generating each time a new random instance. The motivation behind this experiment was to see how the LPEQ approach performs compared to the NAIVE one as the programs change from almost always satisfiable (many stable models) to almost always unsatisfiable (no stable models). With lower values of c/v the LPEQ approach was significantly better than the NAIVE one, as in the previous experiment. As the ratio increased the performance of the NAIVE approach gradually improved, but still the LPEQ approach performed slightly better.

Finally, we combined structured logic programs with randomness. We used two graph problems and the respective formulations by Niemelä [9, p. 262], namely n-coloring of a graph with n colors, and finding a Hamiltonian circuit for a graph. Using the Stanford GraphBase library, we generated random planar graphs with v vertices where v ranged from 10 to 17. Using the front-end LPARSE, the logic programs for 4-coloring and Hamiltonian circuit were instantiated for the generated graphs. As in the preceding experiments with random 3-sat programs, the second program for equivalence testing was obtained by dropping one random rule from the one instantiated by LPARSE. The tests were repeated 100 times for each value of v using a new random planar graph each time. The average running times for both experiments are presented in Figure 2. In both experiments the LPEQ approach was significantly better than the NAIVE approach, though running times differed more in case of 4-coloring random planar graphs.

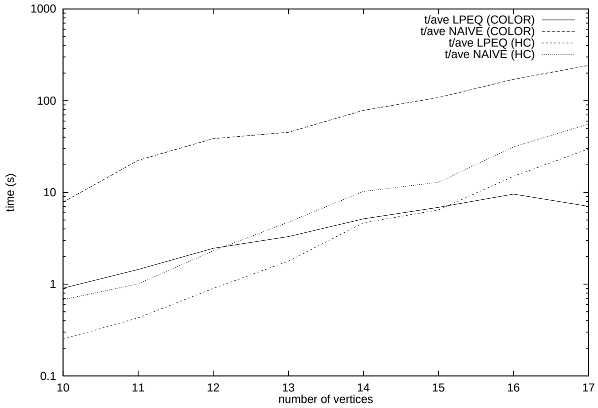

Fig. 2. Results for 4-coloring random planar graphs and finding Hamiltonian circuits.

6 Conclusion

In this paper, we proposed a translation-based approach for testing the equivalence of logic programs under the stable model semantics. The current translation and the implementation LPEQ cover the types of rules supported by the SMODELS system. Since the internal format of SMODELS is used, more general forms of rules implemented in the front-end LPARSE are also covered. The experiments carried out in Section 5 suggest that the translation-based approach can be useful in practice. In many cases, the number of choice points and time needed for computations is less than in a naive cross-checking approach. To the best of our understanding, this is because the translation provides an explicit specification of a counter-example for $P \equiv Q$ that guides the search performed by SMODELS. This is not possible in the NAIVE approach where the stable models of P and Q are of interest. However, if the programs being compared are likely to have few stable models or no stable models at all, we expect that the NAIVE approach becomes superior to ours (c.f. Table 1). Recall that P is included in the translation $\mathrm{EQT}(P, Q, L)[K]$ which has no stable models if $P[K]$ has no stable models.

There are further issues to be addressed. (i) The current translation and implementation do not support minimize/maximize statements (and global weights of default literals) that are available in SMODELS. (ii) Other notions of equivalence – such as the stronger notion of equivalence proposed by Lifschitz et al. [5] – should be covered by devising suitable translations. (iii) At the moment, we rely on the fact that $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$ holds for the programs P and Q under comparison (except that any invisible atoms introduced by LPARSE are ignored). One possibility to cover programs not satisfying $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$ is to extend P and Q to programs P' and Q' (respectively) by adding useless rules not affecting stable models so that $\mathrm{Hb}(P') = \mathrm{Hb}(Q')$. (iv) The case of disjunctive logic programs is also interesting, as efficient implementations are available: DLV [4] and GNT [3]. The latter uses SMODELS for actual computations in analogy to the translation-based approach followed by this paper. We expect that a translation similar to the one presented in this paper can be developed for disjunctive programs as well. (v) Even more general classes of logic programs – such as *nested* programs [5] – are worth addressing once implementations become available.

References

1. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
2. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.
3. T. Janhunen, I. Niemelä, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference*, pages 411–419, Breckenridge, Colorado, April 2000. Morgan Kaufmann.
4. N. Leone et al. DLV – a disjunctive datalog system. `http://www.dbai.tuwien.ac.at/proj/dlv/`.
5. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
6. F. Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 8th International Conference*, pages 170–176, Tolouse, France, April 2002. Morgan Kaufmann.
7. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987.
8. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.
9. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
10. I. Niemelä and P. Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, chapter 21, pages 491–521. Kluwer Academic Publishers, 2000.
11. I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on LP & NMR*, pages 317–331, El Paso, Texas, USA, December 1999. Springer-Verlag. LNAI 1730.
12. I. Niemelä, P. Simons, and T. Syrjänen. Smodels: a system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (cs.AI/0003073)*, Breckenridge, Colorado, USA, April 2000. cs.AI/0003033.
13. D. Pearce, H. Tompits, and S. Woltran. Encodings for equilibirium logic and logic programs with nested expressions. In P. Brazdil and A. Jorge, editors, *Proceedings of the 10th Portuguese Conference on Artificial Intelligence*, pages 306–320, Porto, Portugal, December 2001. Springer Verlag. LNAI 2258.
14. P. Simons. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on LP & NMR*, pages 305–316, El Paso, Texas, USA, December 1999. Springer-Verlag. LNAI 1730.