Modular Equivalence for Normal Logic Programs*

Emilia Oikarinen† and **Tomi Janhunen**
Department of Computer Science and Engineering
Helsinki University of Technology (TKK)
P.O. Box 5400, FI-02015 TKK, Finland
Emilia.Oikarinen@tkk.fi and Tomi.Janhunen@tkk.fi

Abstract

A Gaifman-Shapiro-style architecture of program modules is introduced in the case of normal logic programs under stable model semantics. The composition of program modules is suitably limited by module conditions which ensure the compatibility of the module system with stable models. The resulting module theorem properly strengthens Lifschitz and Turner's splitting set theorem (1994) for normal logic programs. Consequently, the respective notion of equivalence between modules, i.e. modular equivalence, proves to be a congruence relation. Moreover, it is analyzed (i) how the translation-based verification technique from (Janhunen & Oikarinen 2005) is accommodated to the case of modular equivalence and (ii) how the verification of weak/visible equivalence can be reorganized as a sequence of module-level tests and optimized on the basis of modular equivalence.

Introduction

Answer set programming (ASP) is a very promising constraint programming paradigm (Niemelä 1999; Marek & Truszczyński 1999; Gelfond & Leone 2002) in which problems are solved by capturing their solutions as *answer sets* or *stable models* of logic programs. The development and optimization of logic programs in ASP gives rise to a meta-level problem of verifying whether subsequent programs are equivalent. To solve this problem, a translation-based approach has been proposed and extended further (Janhunen & Oikarinen 2002; Turner 2003; Oikarinen & Janhunen 2004; Woltran 2004). The underlying idea is to combine two logic programs P and Q under consideration into two logic programs $\mathrm{EQT}(P, Q)$ and $\mathrm{EQT}(Q, P)$ which have no stable models iff P and Q are *weakly equivalent*, i.e. have the same stable models. This enables the use of the same ASP solver, such as SMODELS (Simons, Niemelä, & Soininen 2002), DLV (Leone *et al.* 2006) or GNT (Janhunen *et al.* 2006), for the equivalence verification problem as for the search of stable models in general. First experimental results (Janhunen & Oikarinen 2002; Oikarinen & Janhunen 2004) suggest that the translation-based method can be effective and sometimes much faster than performing a simple cross-check of stable models.

As a potential limitation, the translation-based method as described above treats programs as integral entities and therefore no computational advantage is sought by breaking programs into smaller parts, say *modules* of some kind. Such an optimization strategy is largely preempted by the fact that weak equivalence, denoted by $\equiv$, fails to be a *congruence relation* for $\cup$, i.e. weak equivalence is not preserved under substitutions in unions of programs. More formally put, $P \equiv Q$ does not imply $P \cup R \equiv Q \cup R$ in general. The same can be stated about *uniform equivalence* (Sagiv 1987) but not about *strong equivalence* (Lifschitz, Pearce, & Valverde 2001) which admits substitutions by definition.

From our point of view, strong equivalence seems inappropriate for *fully modularizing* the verification task of weak equivalence. This is simply because two programs P and Q may be weakly equivalent even if they build on respective modules $P_i \subseteq P$ and $Q_i \subseteq Q$ which are not strongly equivalent. For the same reason, program transformations that are known to preserve strong equivalence (Eiter *et al.* 2004) do not provide an inclusive basis for reasoning about weak equivalence. Nevertheless, there are cases where one can utilize the fact that strong equivalence implies weak equivalence. For instance, if P and Q are composed of strongly equivalent pairs of modules P_i and Q_i for all i, then P and Q can be directly inferred to be strongly and weakly equivalent. These observations about strong equivalence motivate the strive for a weaker congruence relation that is compatible with weak equivalence at program-level.

To address the lack of a suitable congruence relation in the context of ASP, we propose a new design in this article. The design superficially resembles that of Gaifman and Shapiro (1989) but stable model semantics (Gelfond & Lifschitz 1988) and special module conditions are incorporated. The feasibility of the design is crystallized in a *module theorem* which shows the module system fully compatible with stable models. In fact, the module theorem established here is a proper strengthening of the splitting set theorem established by Lifschitz and Turner (1994) in the case of normal logic programs. The main difference is that our result allows negative recursion between modules. Moreover, it enables the introduction of a notion of

*The research reported in this paper has been partially funded by the Academy of Finland (project #211025).

†The financial support from Helsinki Graduate School in Computer Science and Engineering, Nokia Foundation, and Finnish Cultural Foundation is gratefully acknowledged.

equivalence, i.e. *modular equivalence*, which turns out to be a proper congruence relation and reduces to weak equivalence for program modules which have a completely specified input and no hidden atoms. This kind of modules correspond to normal logic programs with completely visible Herbrand base. If normal programs P and Q are composed of modularly equivalent modules P_i and Q_i for all i, then P and Q are modularly equivalent or equivalently stated weakly equivalent. The notion of modular equivalence opens immediately new prospects as regards the translation-based verification method (Janhunen & Oikarinen 2002; Oikarinen & Janhunen 2004). First of all, the method can be tuned for the task of verifying modular equivalence by attaching a *context generator* to program modules in analogy to (Woltran 2004). Second, we demonstrate how the verification of weak equivalence can be reorganized as a sequence of tests, each of which concentrates on a pair of respective modules in the programs subject to the verification task.

The plan for the rest of this article is as follows. As a preparatory step, we briefly review the syntax and semantics of normal logic programs and define notions of equivalence addressed in the sequel. After that we specify program modules as well as establish the module theorem discussed above. Next, we define the notion of modular equivalence, prove the congruence property for it, and give a brief account of computational complexity involved in the respective verification problem. Connections between modular equivalence and the translation-based method for verifying visible equivalence (Janhunen & Oikarinen 2005) are also worked out. Finally, we briefly contrast our work with earlier approaches and present our conclusions.

Normal Logic Programs

We will consider *propositional normal logic programs* in this paper.

Definition 1 *A normal logic program (NLP) is a (finite) set of rules of the form $h \leftarrow B^+, {\sim}B^-$, where h is an atom, B^+ and B^- are sets of atoms, and ${\sim}B = \{{\sim}b \mid b \in B\}$ for any set of atoms B.*

The symbol "$\sim$" denotes *default negation* or *negation as failure* to prove (Clark 1978). Atoms a and their default negations ${\sim}a$ are called *default literals*. A rule consists of two parts: h is the *head* and the rest is the *body*. Let $\mathrm{Head}(P)$ denote the set of head atoms appearing in P, i.e.

$$\mathrm{Head}(P) = \{h \mid h \leftarrow B^+, {\sim}B^- \in P\}.$$

If the body of a rule is empty, the rule is called a *fact* and the symbol "$\leftarrow$" can be omitted. If $B^- = \emptyset$, the rule is *positive*. A program consisting only of positive rules is a *positive logic program*.

Usually the *Herbrand base* $\mathrm{Hb}(P)$ of a normal logic program P is defined to be the set of atoms appearing in the rules of P. We, however, use a revised definition: $\mathrm{Hb}(P)$ *is any fixed set of atoms containing all atoms appearing in the rules of P*. Under this definition the Herbrand base of P can be extended by atoms having no occurrences in P. This aspect is useful e.g. when P is obtained as a result of optimization and there is a need to keep track of the original Herbrand base. Moreover, $\mathrm{Hb}(P)$ is supposed to be finite whenever P is.

Given a normal logic program P, an *interpretation* M of P is a subset of $\mathrm{Hb}(P)$ defining which atoms of $\mathrm{Hb}(P)$ are true ($a \in M$) and which are false ($a \notin M$). An interpretation $M \subseteq \mathrm{Hb}(P)$ is a *(classical) model* of P, denoted by $M \models P$ iff $B^+ \subseteq M$ and $B^- \cap M = \emptyset$ imply $h \in M$ for each rule $h \leftarrow B^+, {\sim}B^- \in P$. For a positive program P, $M \subseteq \mathrm{Hb}(P)$ is the (unique) *least model* of P, denoted by $\mathrm{LM}(P)$, iff there is no $M' \models P$ such that $M' \subset M$. *Stable models* as proposed by Gelfond and Lifschitz (1988) generalize least models for normal logic programs.

Definition 2 *Given a normal logic program P and an interpretation $M \subseteq \mathrm{Hb}(P)$ the Gelfond-Lifschitz reduct*

$$P^M = \{h \leftarrow B^+ \mid h \leftarrow B^+, {\sim}B^- \in P \text{ and } M \cap B^- = \emptyset\},$$

and M is a stable model of P iff $M = \mathrm{LM}(P^M)$.

Stable models are not necessarily unique in general: a normal logic program may have several stable models or no stable models at all. The set of stable models of a NLP P is denoted by $\mathrm{SM}(P)$.

We define a *positive dependency relation* $\leq \subseteq \mathrm{Hb}(P) \times \mathrm{Hb}(P)$ as the reflexive and transitive closure of a relation $\leq_1$ defined as follows. Given $a, b \in \mathrm{Hb}(P)$, we say that b depends directly on a, denoted $a \leq_1 b$, iff there is a rule $b \leftarrow B^+, {\sim}B^- \in P$ such that $a \in B^+$. The *positive dependency graph* of P, $\mathrm{Dep}^+(P)$, is a graph with $\mathrm{Hb}(P)$ as the set of vertices and $\{\langle a, b\rangle \mid a, b \in \mathrm{Hb}(P) \text{ and } a \leq b\}$ as the set of edges. The *negative dependency graph* $\mathrm{Dep}^-(P)$ can be defined analogously. A *strongly connected component* of $\mathrm{Dep}^+(P)$ is a maximal subset $C \subseteq \mathrm{Hb}(P)$ such that for all $a, b \in C$, $\langle a, b\rangle$ is in $\mathrm{Dep}^+(P)$. Thus strongly connected components of $\mathrm{Dep}^+(P)$ partition $\mathrm{Hb}(P)$ into equivalence classes. The dependency relation $\leq$ can then be generalized for the strongly connected components: $C_i \leq C_j$, i.e. C_j depends on C_i, iff $c_i \leq c_j$ for any $c_i \in C_i$ and $c_j \in C_j$.

A *splitting set* for a NLP P is any set $U \subseteq \mathrm{Hb}(P)$ such that for every rule $h \leftarrow B^+, {\sim}B^- \in P$, if $h \in U$ then $B^+ \cup B^- \subseteq U$. The set of rules $h \leftarrow B^+, {\sim}B^- \in P$ such that $\{h\} \cup B^+ \cup B^- \subseteq U$ is the *bottom* of P relative to U, denoted by $\mathrm{b}_U(P)$. The set $\mathrm{t}_U(P) = P \setminus \mathrm{b}_U(P)$ is the *top* of P relative to U. The top can be partially evaluated with respect to an interpretation $X \subseteq U$ resulting a program $\mathrm{e}(\mathrm{t}_U(P), X)$ that contains a rule $h \leftarrow (B^+ \setminus U), {\sim}(B^- \setminus U)$ for each $h \leftarrow B^+, {\sim}B^- \in \mathrm{t}_U(P)$ such that $B^+ \cap U \subseteq X$ and $(B^- \cap U) \cap X = \emptyset$. Given a splitting set U for a NLP P, a *solution* to P with respect to U is a pair $\langle X, Y\rangle$ such that $X \subseteq U$, $Y \subseteq \mathrm{Hb}(P) \setminus U$, $X \in \mathrm{SM}(\mathrm{b}_U(P))$, and $Y \in \mathrm{SM}(\mathrm{e}(\mathrm{t}_U(P), X))$. The *splitting set theorem* relates solutions with stable models.

Theorem 1 *(Lifschitz & Turner 1994) Let U be a splitting set for a NLP P and $M \subseteq \mathrm{Hb}(P)$. Then $M \in \mathrm{SM}(P)$ iff the pair $\langle M \cap U, M \setminus U\rangle$ is a solution to P with respect to U.*

Notions of Equivalence

The notion of *strong equivalence* was introduced by Lifschitz, Pearce and Valverde (2001) whereas *uniform equiva-*

lence has its roots in the database community (Sagiv 1987); cf. (Eiter & Fink 2003) for the case of stable models.

Definition 3 *Normal logic programs P and Q are (weakly) equivalent, denoted $P \equiv Q$, iff $\mathrm{SM}(P) = \mathrm{SM}(Q)$; strongly equivalent, denoted $P \equiv_\mathrm{s} Q$, iff $P \cup R \equiv Q \cup R$ for any normal logic program R; and uniformly equivalent, denoted $P \equiv_\mathrm{u} Q$, iff $P \cup F \equiv Q \cup F$ for any set of facts F.*

Clearly, $P \equiv_\mathrm{s} Q$ implies $P \equiv_\mathrm{u} Q$, and $P \equiv_\mathrm{u} Q$ implies $P \equiv Q$, but not vice versa (in both cases). 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 normal programs, i.e. if $P \equiv_\mathrm{s} Q$, then $P \cup R \equiv_\mathrm{s} Q \cup R$ for all normal programs R. On the other hand, uniform equivalence is not a congruence, as shown in Example 1 below. Consequently, the same applies to weak equivalence and thus $\equiv$ and $\equiv_\mathrm{u}$ are best suited for the comparison of complete programs, and not for modules.

Example 1 *(Eiter* et al. *2004, Example 1) Consider normal logic programs $P = \{a.\}$ and $Q = \{a \leftarrow \sim b.\ a \leftarrow b.\}$. It holds $P \equiv_\mathrm{u} Q$, but $P \cup R \not\equiv Q \cup R$ for $R = \{b \leftarrow a.\}$. Thus $P \not\equiv_\mathrm{s} Q$ and $\equiv_\mathrm{u}$ is not a congruence relation for $\cup$.*

For $P \equiv Q$ to hold, the stable models in $\mathrm{SM}(P)$ and $\mathrm{SM}(Q)$ have to be identical subsets of $\mathrm{Hb}(P)$ and $\mathrm{Hb}(Q)$, respectively. The same can be stated about strong and uniform equivalence. This makes these notions of equivalence less useful if $\mathrm{Hb}(P)$ and $\mathrm{Hb}(Q)$ differ by some atoms which are not trivially false in all stable models. Such atoms might, however, be of use when formalizing some auxiliary concepts. Following the ideas from (Janhunen 2003) we partition $\mathrm{Hb}(P)$ into two parts $\mathrm{Hb}_\mathrm{v}(P)$ and $\mathrm{Hb}_\mathrm{h}(P)$ which determine the *visible* and the *hidden* parts of $\mathrm{Hb}(P)$, respectively. In *visible equivalence* the idea is that the hidden atoms in $\mathrm{Hb}_\mathrm{h}(P)$ and $\mathrm{Hb}_\mathrm{h}(Q)$ are local to P and Q and negligible as regards the equivalence of the two programs.

Definition 4 *(Janhunen 2003) Normal logic programs P and Q are visibly equivalent, denoted by $P \equiv_\mathrm{v} Q$, iff $\mathrm{Hb}_\mathrm{v}(P) = \mathrm{Hb}_\mathrm{v}(Q)$ and there is a bijection $f : \mathrm{SM}(P) \rightarrow \mathrm{SM}(Q)$ such that for all interpretations $M \in \mathrm{SM}(P)$, $M \cap \mathrm{Hb}_\mathrm{v}(P) = f(M) \cap \mathrm{Hb}_\mathrm{v}(Q)$.*

Note that the number of stable models is preserved under $\equiv_\mathrm{v}$. Such a strict correspondence of models is much dictated by the ASP methodology: the stable models of a program usually correspond to the solutions of the problem being solved and thus $\equiv_\mathrm{v}$ preserves the number of solutions, too. In the fully visible case, i.e. $\mathrm{Hb}_\mathrm{h}(P) = \mathrm{Hb}_\mathrm{h}(Q) = \emptyset$, the relation $\equiv_\mathrm{v}$ becomes very close to $\equiv$. The only difference is the additional requirement $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$ insisted by $\equiv_\mathrm{v}$. This is of little importance as Herbrand bases can always be extended to meet $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$. Since weak equivalence is not a congruence, visible equivalence cannot be a congruence either.

The *relativized variants of strong and uniform equivalence* introduced by Woltran (2004) allow the context to be constrained using a set of atoms A.

Definition 5 *Normal logic programs P and Q are strongly equivalent relative to A, denoted by $P \equiv_\mathrm{s}^A Q$, iff $P \cup R \equiv Q \cup R$ for all normal logic programs R over the set of atoms A; uniformly equivalent relative to A, denoted by $P \equiv_\mathrm{u}^A Q$, iff $P \cup F \equiv Q \cup F$ for all sets of facts $F \subseteq A$.*

Setting $A = \emptyset$ in the above reduces both relativized notions to weak equivalence, and thus neither is a congruence.

Eiter *et al.* (2005) introduce a very general framework based on *equivalence frames* to capture various kinds of equivalence relations. Most of the notions of equivalence defined above can be defined using the framework. Visible equivalence is exceptional in the sense that it does not fit into equivalence frames based on *projected answer sets*. A projective variant of Definition 4 would simply equate $\{M \cap \mathrm{Hb}_\mathrm{v}(P) \mid M \in \mathrm{SM}(P)\}$ to $\{N \cap \mathrm{Hb}_\mathrm{v}(Q) \mid N \in \mathrm{SM}(Q)\}$. As a consequence, the number of answer sets may not be preserved which we find somewhat unsatisfactory because of the general nature of ASP as discussed after Definition 4. Consider, for instance programs $P = \{a \leftarrow \sim b.\ b \leftarrow \sim a.\ \}$ and $Q_n = P \cup \{c_i \leftarrow \sim d_i.\ d_i \leftarrow \sim c_i.\ \mid 0 < i \leq n\}$ with $\mathrm{Hb}_\mathrm{v}(P) = \mathrm{Hb}_\mathrm{v}(Q_n) = \{a, b\}$. Whenever $n > 0$ these programs are not visibly equivalent but they would be equivalent under the projective definition. With sufficiently large values of n it is no longer feasible to count the number of different stable models (i.e. solutions) if Q_n is used.

Modular Logic Programs

We define a *logic program module* similarly to Gaifman and Shapiro (1989), but consider the case of normal logic programs instead of positive (disjunctive) logic programs.

Definition 6 *A triple $\mathbb{P} = (P, I, O)$ is a (propositional logic program) module, if*

1. *P is a finite set of rules of the form $h \leftarrow B^+, \sim B^-$;*
2. *I and O are sets of propositional atoms such that $I \cap O = \emptyset$; and*
3. *$\mathrm{Head}(P) \cap I = \emptyset$.*

The Herbrand base of module $\mathbb{P}$, $\mathrm{Hb}(\mathbb{P})$, is the set of atoms appearing in P combined with $I \cup O$. Intuitively the set I defines the *input* of a module and the set O is the *output*. The input and output atoms are considered visible, i.e. the visible Herbrand base of module $\mathbb{P}$ is $\mathrm{Hb}_\mathrm{v}(\mathbb{P}) = I \cup O$. Notice that I and O can also contain atoms not appearing in P, similarly to the possibility of having additional atoms in the Herbrand bases of normal logic programs. All other atoms are hidden, i.e. $\mathrm{Hb}_\mathrm{h}(\mathbb{P}) = \mathrm{Hb}(\mathbb{P}) \setminus \mathrm{Hb}_\mathrm{v}(\mathbb{P})$.

As regards the composition of modules, we follow (Gaifman & Shapiro 1989) and take the union of the disjoint sets of rules involved in them. The conditions given by Gaifman and Shapiro are not yet sufficient for our purposes, and we impose a further restriction denying positive recursion between modules.

Definition 7 *Consider modules $\mathbb{P}_1 = (P_1, I_1, O_1)$ and $\mathbb{P}_2 = (P_2, I_2, O_2)$ and let $C_1, \ldots, C_n$ be the strongly connected components of $\mathrm{Dep}^+(P_1 \cup P_2)$. There is a positive recursion between $\mathbb{P}_1$ and $\mathbb{P}_2$, if $C_i \cap O_1 \neq \emptyset$ and $C_i \cap O_2 \neq \emptyset$ for some component C_i.*

The idea is that all inter-module dependencies go through the input/output interface of the modules, i.e. the output of one module can serve as the input for another and hidden atoms are local to each module. Now, if there is a strongly connected component C_i in $\mathrm{Dep}^+(P_1 \cup P_2)$ containing atoms from both O_1 and O_2, we know that, if programs P_1 and P_2 are combined, some output atom a of $\mathbb{P}_1$ depends positively on some output atom b of $\mathbb{P}_2$ which again depends positively on a. This yields a positive recursion.

Definition 8 *Let $\mathbb{P}_1 = (P_1, I_1, O_1)$ and $\mathbb{P}_2 = (P_2, I_2, O_2)$ be modules such that*

1. $O_1 \cap O_2 = \emptyset$;
2. $\mathrm{Hb_h}(\mathbb{P}_1) \cap \mathrm{Hb}(\mathbb{P}_2) = \mathrm{Hb_h}(\mathbb{P}_2) \cap \mathrm{Hb}(\mathbb{P}_1) = \emptyset$; *and*
3. *there is no positive recursion between $\mathbb{P}_1$ and $\mathbb{P}_2$.*

Then the join of $\mathbb{P}_1$ and $\mathbb{P}_2$, denoted by $\mathbb{P}_1 \sqcup \mathbb{P}_2$, is defined, and $\mathbb{P}_1 \sqcup \mathbb{P}_2 = (P_1 \cup P_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2)$.

Remark. Condition 1 in Definition 8 is actually redundant as it is implied by condition 3. Also, condition 2 can be circumvented in practice using a suitable scheme, e.g. based on module names, to rename the hidden atoms uniquely for each module.

Some observations follow. Since each atom is defined in one module, the sets of rules in $\mathbb{P}_1$ and $\mathbb{P}_2$ are disjoint, i.e. $P_1 \cap P_2 = \emptyset$. Also,

$$\begin{aligned} \mathrm{Hb}(\mathbb{P}_1 \sqcup \mathbb{P}_2) &= \mathrm{Hb}(\mathbb{P}_1) \cup \mathrm{Hb}(\mathbb{P}_2), \\ \mathrm{Hb_v}(\mathbb{P}_1 \sqcup \mathbb{P}_2) &= \mathrm{Hb_v}(\mathbb{P}_1) \cup \mathrm{Hb_v}(\mathbb{P}_2), \text{ and} \\ \mathrm{Hb_h}(\mathbb{P}_1 \sqcup \mathbb{P}_2) &= \mathrm{Hb_h}(\mathbb{P}_1) \cup \mathrm{Hb_h}(\mathbb{P}_2). \end{aligned}$$

Note that the module conditions above impose no restrictions on *negative* dependencies or on positive dependencies *inside* modules. The input of $\mathbb{P}_1 \sqcup \mathbb{P}_2$ might be smaller than the union of inputs of individual modules. This is illustrated by the following example.

Example 2 *Consider modules $\mathbb{P} = (\{a \leftarrow \sim b.\}, \{b\}, \{a\})$ and $\mathbb{Q} = (\{b \leftarrow \sim a.\}, \{a\}, \{b\})$. The join of $\mathbb{P}$ and $\mathbb{Q}$ is defined, and $\mathbb{P} \sqcup \mathbb{Q} = (\{a \leftarrow \sim b.\ b \leftarrow \sim a.\}, \emptyset, \{a, b\})$.*

The following hold for the intersections of Herbrand bases under the conditions 1 and 2 in Definition 8:

$$\begin{aligned} &\mathrm{Hb_v}(\mathbb{P}_1) \cap \mathrm{Hb_v}(\mathbb{P}_2) \\ &\qquad = \mathrm{Hb}(\mathbb{P}_1) \cap \mathrm{Hb}(\mathbb{P}_2) \\ &\qquad = (I_1 \cap I_2) \cup (I_1 \cap O_2) \cup (I_2 \cap O_1), \text{ and} \\ &\mathrm{Hb_h}(\mathbb{P}_1) \cap \mathrm{Hb_h}(\mathbb{P}_2) = \emptyset. \end{aligned}$$

Join operation $\sqcup$ has the following properties:

- Identity: $\mathbb{P} \sqcup (\emptyset, \emptyset, \emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup \mathbb{P} = \mathbb{P}$ for all $\mathbb{P}$.
- Commutativity: $\mathbb{P}_1 \sqcup \mathbb{P}_2 = \mathbb{P}_2 \sqcup \mathbb{P}_1$ for all modules $\mathbb{P}_1$ and $\mathbb{P}_2$ such that $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined.
- Associativity: $(\mathbb{P}_1 \sqcup \mathbb{P}_2) \sqcup \mathbb{P}_3 = \mathbb{P}_1 \sqcup (\mathbb{P}_2 \sqcup \mathbb{P}_3)$ for all modules $\mathbb{P}_1, \mathbb{P}_2$ and $\mathbb{P}_3$ such that the joins are defined.

Note that equality sign "=" used here denotes syntactical equivalence, whereas semantical equivalence will be defined in the next section.

The stable semantics of a module is defined with respect to a given input, i.e. a subset of the input atoms of the module. Input is seen as a set of facts (or a database) to be added to the module.

Definition 9 *Given a module $\mathbb{P} = (P, I, O)$ and a set of atoms $A \subseteq I$ the instantiation of $\mathbb{P}$ with the input A is $\mathbb{P}(A) = \mathbb{P} \sqcup \mathbb{F}_A$, where $\mathbb{F}_A = (\{a. \mid a \in A\}, \emptyset, I)$.*

Note that $\mathbb{P}(A) = (P \cup \{a. \mid a \in A\}, \emptyset, I \cup O)$ is essentially a normal logic program with $I \cup O$ as the visible Herbrand base. We can thus generalize the stable model semantics for modules. In the sequel we identify $\mathbb{P}(A)$ with the respective set of rules $P \cup F_A$, where $F_A = \{a. \mid a \in A\}$. In the following $M \cap I$ acts as a particular input with respect to which the module is instantiated.

Definition 10 *An interpretation $M \subseteq \mathrm{Hb}(\mathbb{P})$ is a stable model of a module $\mathbb{P} = (P, I, O)$, denoted by $M \in \mathrm{SM}(\mathbb{P})$, iff $M = \mathrm{LM}(P^M \cup F_{M \cap I})$.*

We define a concept of *compatibility* to describe when a stable model M_1 of module $\mathbb{P}_1$ can be combined with a stable model M_2 of another module $\mathbb{P}_2$. This is exactly when M_1 and M_2 share the common (visible) part.

Definition 11 *Let $\mathbb{P}_1$ and $\mathbb{P}_2$ be modules, and $M_1 \in \mathrm{SM}(\mathbb{P}_1)$ and $M_2 \in \mathrm{SM}(\mathbb{P}_2)$ their stable models which are compatible, iff $M_1 \cap \mathrm{Hb_v}(\mathbb{P}_2) = M_2 \cap \mathrm{Hb_v}(\mathbb{P}_1)$.*

If a program (module) consists of several modules, its stable models are locally stable for the respective submodules; and on the other hand, local stability implies global stability as long as the stable models of the submodules are compatible.

Theorem 2 *(Module theorem). Let $\mathbb{P}_1$ and $\mathbb{P}_2$ be modules such that $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined. Now, $M \in \mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$ iff $M_1 = M \cap \mathrm{Hb}(\mathbb{P}_1) \in \mathrm{SM}(\mathbb{P}_1)$, $M_2 = M \cap \mathrm{Hb}(\mathbb{P}_2) \in \mathrm{SM}(\mathbb{P}_2)$, and M_1 and M_2 are compatible.*

Proof sketch. "$\Rightarrow$" M_1 and M_2 are clearly compatible and it is straightforward to show that conditions 1 and 2 in Definition 8 imply $M_1 \in \mathrm{SM}(\mathbb{P}_1)$ and $M_2 \in \mathrm{SM}(\mathbb{P}_2)$.
"$\Leftarrow$" Consider $\mathbb{P}_1 = (P_1, I_1, O_1)$, $\mathbb{P}_2 = (P_2, I_2, O_2)$ and their join $\mathbb{P} = \mathbb{P}_1 \sqcup \mathbb{P}_2 = (P, I, O)$. Let $M_1 \in \mathrm{SM}(\mathbb{P}_1)$, and $M_2 \in \mathrm{SM}(\mathbb{P}_2)$ be compatible and define $M = M_1 \cup M_2$. There is a strict total ordering $<$ for the strongly connected components C_i of $\mathrm{Dep}^+(P)$ such that if $C_i < C_j$, then $C_i \leq C_j$ and $C_j \not\leq C_i$; or $C_i \not\leq C_j$ and $C_j \not\leq C_i$. Let $C_1 < \cdots < C_n$ be such an ordering. Show that exactly one of the following holds for each C_i: (i) $C_i \subseteq I$, (ii) $C_i \subseteq O_1 \cup \mathrm{Hb_h}(\mathbb{P}_1)$, or (iii) $C_i \subseteq O_2 \cup \mathrm{Hb_h}(\mathbb{P}_2)$. Finally, show by induction that

$$M \cap (\bigcup_{i=1}^{k} C_i) = \mathrm{LM}(P^M \cup F_{M \cap I}) \cap (\bigcup_{i=1}^{k} C_i)$$

holds for $0 \leq k \leq n$ by applying the splitting set theorem (Lifschitz & Turner 1994). □

Example 3 shows that condition 3 in Definition 8 is necessary to guarantee that local stability implies global stability.

Example 3 *Consider $\mathbb{P}_1 = (\{a \leftarrow b.\}, \{b\}, \{a\})$ and $\mathbb{P}_2 = (\{b \leftarrow a.\}, \{a\}, \{b\})$ with $\mathrm{SM}(\mathbb{P}_1) = \mathrm{SM}(\mathbb{P}_2) = \{\emptyset, \{a, b\}\}$. The join of $\mathbb{P}_1$ and $\mathbb{P}_2$ is not defined because of positive recursion (conditions 1 and 2 in Definition 8 are satisfied, however). For a NLP $P = \{a \leftarrow b.\ b \leftarrow a.\}$, we get $\mathrm{SM}(P) = \{\emptyset\}$. Thus, the positive dependency between a and b excludes $\{a, b\}$ from $\mathrm{SM}(P)$.*

Theorem 2 is strictly stronger than the splitting set theorem (Lifschitz & Turner 1994) for normal logic programs. If U is a splitting set for a NLP P, then

$$P = \mathbb{B} \sqcup \mathbb{T} = (\mathrm{b}_U(P), \emptyset, U) \sqcup (\mathrm{t}_U(P), U, \mathrm{Hb}(P) \setminus U),$$

and it follows from Theorems 1 and 2 that $M_1 \in \mathrm{SM}(\mathbb{B})$ and $M_2 \in \mathrm{SM}(\mathbb{T})$ iff $\langle M_1, M_2 \setminus U \rangle$ is a solution for P with respect to U. On the other hand the splitting set theorem cannot be applied to e.g. $\mathbb{P} \sqcup \mathbb{Q}$ from Example 2, since neither $\{a\}$ nor $\{b\}$ is a splitting set. Our theorem also strengthens a module theorem given in (Janhunen 2003, Theorem 6.22) to cover normal programs that involve positive body literals, too. Moreover, Theorem 2 can easily be generalized for modules consisting of several submodules. Consider a collection of modules $\mathbb{P}_1, \ldots, \mathbb{P}_n$ such that the join $\mathbb{P}_1 \sqcup \cdots \sqcup \mathbb{P}_n$ is defined (recall that $\sqcup$ is associative). We say that a collection of stable models $\{M_1, \ldots, M_n\}$ for modules $\mathbb{P}_1, \ldots, \mathbb{P}_n$, respectively, is *compatible*, iff M_i and M_j are pairwise compatible for all $1 \leq i, j \leq n$.

Corollary 1 *Let $\mathbb{P}_1, \ldots, \mathbb{P}_n$ be a collection of modules such that $\mathbb{P}_1 \sqcup \cdots \sqcup \mathbb{P}_n$ is defined. Now $M \in \mathrm{SM}(\mathbb{P}_1 \sqcup \cdots \sqcup \mathbb{P}_n)$ iff $M_i = M \cap \mathrm{Hb}(\mathbb{P}_i) \in \mathrm{SM}(\mathbb{P}_i)$ for all $1 \leq i \leq n$, and the set of stable models $\{M_1, \ldots, M_n\}$ is compatible.*

Corollary 1 enables the computation of stable models on a module-by-module basis, but it leaves us the task of excluding mutually incompatible combinations of stable models.

Example 4 *Consider modules*

$$\begin{aligned} \mathbb{P}_1 &= (\{a \leftarrow {\sim}b.\}, \{b\}, \{a\}), \\ \mathbb{P}_2 &= (\{b \leftarrow {\sim}c.\}, \{c\}, \{b\}), \text{ and} \\ \mathbb{P}_3 &= (\{c \leftarrow {\sim}a.\}, \{a\}, \{c\}). \end{aligned}$$

The join $\mathbb{P} = \mathbb{P}_1 \sqcup \mathbb{P}_2 \sqcup \mathbb{P}_3$ is defined,

$$\mathbb{P} = (\{a \leftarrow {\sim}b.\ b \leftarrow {\sim}c.\ c \leftarrow {\sim}a.\}, \emptyset, \{a, b, c\}).$$

Now $\mathrm{SM}(\mathbb{P}_1) = \{\{a\}, \{b\}\}$, $\mathrm{SM}(\mathbb{P}_2) = \{\{b\}, \{c\}\}$ and $\mathrm{SM}(\mathbb{P}_3) = \{\{a\}, \{c\}\}$. To apply Corollary 1 for finding $\mathrm{SM}(\mathbb{P})$, one has to find a compatible triple of stable models M_1, M_2, and M_3 for $\mathbb{P}_1$, $\mathbb{P}_2$, and $\mathbb{P}_3$, respectively.

- *Now $\{a\} \in \mathrm{SM}(\mathbb{P}_1)$ and $\{c\} \in \mathrm{SM}(\mathbb{P}_2)$ are compatible, since $\{a\} \cap \mathrm{Hb}_\mathrm{v}(\mathbb{P}_2) = \emptyset = \{c\} \cap \mathrm{Hb}_\mathrm{v}(\mathbb{P}_1)$. However, $\{a\} \in \mathrm{SM}(\mathbb{P}_3)$ is not compatible with $\{c\} \in \mathrm{SM}(\mathbb{P}_2)$, since $\{c\} \cap \mathrm{Hb}_\mathrm{v}(\mathbb{P}_3) = \{c\} \neq \emptyset = \{a\} \cap \mathrm{Hb}_\mathrm{v}(\mathbb{P}_2)$. On the other hand, $\{c\} \in \mathrm{SM}(\mathbb{P}_3)$ is not compatible with $\{a\} \in \mathrm{SM}(\mathbb{P}_1)$, since $\{a\} \cap \mathrm{Hb}_\mathrm{v}(\mathbb{P}_3) = \{a\} \neq \emptyset = \{c\} \cap \mathrm{Hb}_\mathrm{v}(\mathbb{P}_1)$.*
- *Also $\{b\} \in \mathrm{SM}(\mathbb{P}_1)$ and $\{b\} \in \mathrm{SM}(\mathbb{P}_2)$ are compatible, but $\{b\} \in \mathrm{SM}(\mathbb{P}_1)$ is incompatible with $\{a\} \in \mathrm{SM}(\mathbb{P}_3)$. Nor is $\{b\} \in \mathrm{SM}(\mathbb{P}_2)$ compatible with $\{c\} \in \mathrm{SM}(\mathbb{P}_3)$.*

Thus it is impossible to select $M_1 \in \mathrm{SM}(\mathbb{P}_1)$, $M_2 \in \mathrm{SM}(\mathbb{P}_2)$ and $M_3 \in \mathrm{SM}(\mathbb{P}_3)$ such that $\{M_1, M_2, M_3\}$ is compatible, which is understandable as $\mathrm{SM}(\mathbb{P}) = \emptyset$.

Modular Equivalence

The definition of *modular equivalence* combines features from relativized uniform equivalence (Woltran 2004) and visible equivalence (Janhunen 2003).

Definition 12 *Logic program modules $\mathbb{P} = (P, I_P, O_P)$ and $\mathbb{Q} = (Q, I_Q, O_Q)$ are modularly equivalent, denoted by $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$, iff*

1. *$I_P = I_Q = I$ and $O_P = O_Q = O$, and*
2. *$\mathbb{P}(A) \equiv_\mathrm{v} \mathbb{Q}(A)$ for all $A \subseteq I$.*

Modular equivalence is very close to visible equivalence defined for modules. As a matter a fact, if Definition 4 is generalized for program modules, the second condition in Definition 12 can be revised to $\mathbb{P} \equiv_\mathrm{v} \mathbb{Q}$. However, $\mathbb{P} \equiv_\mathrm{v} \mathbb{Q}$ is not enough to cover the first condition in Definition 12, as visible equivalence only enforces $\mathrm{Hb}_\mathrm{v}(\mathbb{P}) = \mathrm{Hb}_\mathrm{v}(\mathbb{Q})$. If $I = \emptyset$, modular equivalence coincides with visible equivalence. If $O = \emptyset$, then $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$ means that $\mathbb{P}$ and $\mathbb{Q}$ have the same number of stable models on each input.

Furthermore, if one considers the *fully visible case*, i.e. $\mathrm{Hb}_\mathrm{h}(\mathbb{P}) = \mathrm{Hb}_\mathrm{h}(\mathbb{Q}) = \emptyset$, modular equivalence can be seen as a special case of A-uniform equivalence for $A = I$. Recall, however, the restrictions $\mathrm{Head}(P) \cap I = \mathrm{Head}(Q) \cap I = \emptyset$ imposed by module structure. With a further restriction $I = \emptyset$, modular equivalence coincides with weak equivalence because $\mathrm{Hb}(\mathbb{P}) = \mathrm{Hb}(\mathbb{Q})$ can always be satisfied by extending Herbrand bases. Basically, setting $I = \mathrm{Hb}(\mathbb{P})$ would give us uniform equivalence, but the additional condition $\mathrm{Head}(P) \cap I = \emptyset$ leaves room for the empty module only.

Since $\equiv_\mathrm{v}$ is not a congruence relation for $\cup$, neither is modular equivalence. The situation changes, however, if one considers the join operation $\sqcup$ which suitably restricts possible contexts. Consider for instance the programs P and Q given in Example 1. We can define modules based on them: $\mathbb{P} = (P, \{b\}, \{a\})$ and $\mathbb{Q} = (Q, \{b\}, \{a\})$. Now $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$ and it is not possible to define a module $\mathbb{R}$ based on $R = \{b \leftarrow a.\}$ such that $\mathbb{Q} \sqcup \mathbb{R}$ is defined.

Theorem 3 *Let $\mathbb{P}, \mathbb{Q}$ and $\mathbb{R}$ be logic program modules. If $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$ and both $\mathbb{P} \sqcup \mathbb{R}$ and $\mathbb{Q} \sqcup \mathbb{R}$ are defined, then $\mathbb{P} \sqcup \mathbb{R} \equiv_\mathrm{m} \mathbb{Q} \sqcup \mathbb{R}$.*

Proof. Let $\mathbb{P} = (P, I, O)$ and $\mathbb{Q} = (Q, I, O)$ be modules such that $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$. Let $\mathbb{R} = (R, I_R, O_R)$ be an arbitrary module such that $\mathbb{P} \sqcup \mathbb{R}$ and $\mathbb{Q} \sqcup \mathbb{R}$ are defined. Consider an arbitrary $M \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$. By Theorem 2, $M_P = M \cap \mathrm{Hb}(\mathbb{P}) \in \mathrm{SM}(\mathbb{P})$ and $M_R = M \cap \mathrm{Hb}(\mathbb{R}) \in \mathrm{SM}(\mathbb{R})$. Since $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$, there is a bijection $f : \mathrm{SM}(\mathbb{P}) \to \mathrm{SM}(\mathbb{Q})$ such that $M_P \in \mathrm{SM}(\mathbb{P}) \iff f(M_P) \in \mathrm{SM}(\mathbb{Q})$, and

$$M_P \cap (O \cup I) = f(M_P) \cap (O \cup I). \tag{1}$$

Let $M_Q = f(M_P)$. Clearly, M_P and M_R are compatible. Since (1) holds, also M_Q and M_R are compatible. Applying Theorem 2 we get $M_Q \cup M_R \in \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R})$. Define function $g : \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R}) \to \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R})$ as

$$g(M) = f(M \cap \mathrm{Hb}(\mathbb{P})) \cup (M \cap \mathrm{Hb}(\mathbb{R})).$$

Clearly, g restricted to the visible part is an identity function, i.e. $M \cap (I \cup I_R \cup O \cup O_R) = g(M) \cap (I \cup I_R \cup O \cup O_R)$. Function g is a bijection, since

- g is an injection: $M \neq N$ implies $g(M) \neq g(N)$ for all $M, N \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$, since $f(M \cap \mathrm{Hb}(\mathbb{P})) \neq f(N \cap \mathrm{Hb}(\mathbb{P}))$ or $M \cap \mathrm{Hb}(\mathbb{R}) \neq N \cap \mathrm{Hb}(\mathbb{R})$.

- g is a surjection: for any $M \in \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R})$, $N = f^{-1}(M \cap \mathrm{Hb}(\mathbb{Q})) \cup (M \cap \mathrm{Hb}(\mathbb{R})) \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$ and $g(N) = M$, since f is a surjection.

The inverse function $g^{-1} : \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R}) \to \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$ can be defined as $g^{-1}(N) = f^{-1}(N \cap \mathrm{Hb}(\mathbb{Q})) \cup (N \cap \mathrm{Hb}(\mathbb{R}))$. Thus $\mathbb{P} \sqcup \mathbb{R} \equiv_{\mathrm{m}} \mathbb{Q} \sqcup \mathbb{R}$. □

It is instructive to consider a potentially stronger variant of modular equivalence defined in analogy to strong equivalence (Lifschitz *et al.* 2001): $\mathbb{P} \equiv^{\mathrm{s}}_{\mathrm{m}} \mathbb{Q}$ iff $\mathbb{P} \sqcup \mathbb{R} \equiv_{\mathrm{m}} \mathbb{Q} \sqcup \mathbb{R}$ holds for all $\mathbb{R}$ such that $\mathbb{P} \sqcup \mathbb{R}$ and $\mathbb{Q} \sqcup \mathbb{R}$ are defined. However, Theorem 3 implies that $\equiv^{\mathrm{s}}_{\mathrm{m}}$ adds nothing to $\equiv_{\mathrm{m}}$ since $\mathbb{P} \equiv^{\mathrm{s}}_{\mathrm{m}} \mathbb{Q}$ iff $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$.

Complexity Remarks

Let us then make some observations about the computational complexity of verifying modular equivalence of normal logic programs. In general, deciding $\equiv_{\mathrm{m}}$ is **coNP**-hard, since deciding the weak equivalence $P \equiv Q$ reduces to deciding $(P, \emptyset, \mathrm{Hb}(P)) \equiv_{\mathrm{m}} (Q, \emptyset, \mathrm{Hb}(Q))$. In the fully visible case $\mathrm{Hb}_{\mathrm{h}}(P) = \mathrm{Hb}_{\mathrm{h}}(Q) = \emptyset$, deciding $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ can be reduced to deciding relativized uniform equivalence $P \equiv^{I}_{\mathrm{u}} Q$ (Woltran 2004) and thus deciding $\equiv_{\mathrm{m}}$ is **coNP**-complete in this restricted case. In the other extreme, $\mathrm{Hb}_{\mathrm{v}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{v}}(\mathbb{Q}) = \emptyset$ and $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ iff $\mathbb{P}$ and $\mathbb{Q}$ have the same number of stable models. This suggests a much higher computational complexity of verifying $\equiv_{\mathrm{m}}$ in general because classical models can be captured with stable models (Niemelä 1999) and counting stable models cannot be easier than #SAT which is #**P**-complete (Valiant 1979).

A way to govern the computational complexity of verifying $\equiv_{\mathrm{m}}$ is to limit the use of hidden atoms as done in the case of $\equiv_{\mathrm{v}}$ by Janhunen and Oikarinen (2005). Therefrom we adopt the property of having *enough visible atoms* (the EVA property for short) defined as follows. For a normal program P and an interpretation $M_{\mathrm{v}} \subseteq \mathrm{Hb}_{\mathrm{v}}(P)$ for the visible part of P, the *hidden part* $P_{\mathrm{h}}/M_{\mathrm{v}}$ of P relative M_{v} contains for each rule $h \leftarrow B^+, {\sim}B^- \in P$ such that $h \in \mathrm{Hb}_{\mathrm{h}}(P)$ and $M_{\mathrm{v}} \models B^+_{\mathrm{v}} \cup {\sim}B^-_{\mathrm{v}}$, the respective hidden part $h \leftarrow B^+_{\mathrm{h}}, {\sim}B^-_{\mathrm{h}}$. The construction of the hidden part $P_{\mathrm{h}}/M_{\mathrm{v}}$ is closely related to the simplification operation $\mathrm{simp}(P, T, F)$ proposed by Cholewinski and Truszczyński (1999), but restricted in the sense that T and F are subsets of $\mathrm{Hb}_{\mathrm{v}}(P)$ rather than $\mathrm{Hb}(P)$. More precisely put, we have $P_{\mathrm{h}}/M_{\mathrm{v}} = \mathrm{simp}(P, M_{\mathrm{v}}, \mathrm{Hb}_{\mathrm{v}}(P) - M_{\mathrm{v}})$ for any program P.

Definition 13 *A normal logic program P has enough visible atoms iff $P_{\mathrm{h}}/M_{\mathrm{v}}$ has a unique stable model for every interpretation $M_{\mathrm{v}} \subseteq \mathrm{Hb}_{\mathrm{v}}(P)$.*

The intuition behind Definition 13 is that the interpretation of $\mathrm{Hb}_{\mathrm{h}}(P)$ is uniquely determined for each interpretation of $\mathrm{Hb}_{\mathrm{v}}(P)$ if P has the EVA property. Consequently, the stable models of P can be distinguished on the basis of their visible parts. By the EVA assumption (Janhunen & Oikarinen 2005), the verification of $\equiv_{\mathrm{v}}$ becomes a **coNP**-complete problem for SMODELS programs[1] involving hidden atoms. This complexity result enables us to generalize the translation-based method from (Janhunen & Oikarinen 2002) for deciding $\equiv_{\mathrm{v}}$. Although verifying the EVA property can be hard in general, there are syntactic subclasses of normal programs (e.g. those for which $P_{\mathrm{h}}/M_{\mathrm{v}}$ is always stratified) with the EVA property. It should be stressed that the use of visible atoms remains unlimited and thus the full expressiveness of normal rules remains at our disposal.

So far we have discussed the role of the EVA assumption in the verification of $\equiv_{\mathrm{v}}$. It is equally important in conjunction with $\equiv_{\mathrm{m}}$. This becomes evident once we work out the interconnections of the two relations in the next section.

Application Strategies

The objective of this section is to describe ways in which modular equivalence can be exploited in the verification of visible/weak equivalence. One concrete step in this respect is to reduce the problem of verifying $\equiv_{\mathrm{m}}$ to that of $\equiv_{\mathrm{v}}$ by introducing a special module $\mathbb{G}_I$ that acts as a context generator. A similar technique is used by Woltran (2004) in the case of relativized uniform equivalence.

Theorem 4 *Let $\mathbb{P}$ and $\mathbb{Q}$ be program modules such that $\mathrm{Hb}_{\mathrm{v}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{v}}(\mathbb{Q}) = O \cup I$. Then $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ iff $\mathbb{P} \sqcup \mathbb{G}_I \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{G}_I$ where $\mathbb{G}_I = (\{a \leftarrow {\sim}\overline{a}.\ \overline{a} \leftarrow {\sim}a \mid a \in I\}, \emptyset, I)$ is a module generating all possible inputs for $\mathbb{P}$ and $\mathbb{Q}$.*

Proof sketch. Note that $\mathbb{G}_I$ has $2^{|I|}$ stable models of the form $A \cup \{\overline{a} \mid a \in I \setminus A\}$ for each $A \subseteq I$. Thus $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{P} \sqcup \mathbb{G}_I$ and $\mathbb{Q} \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{G}_I$ follow by Definitions 2 and 4 and Theorem 2. It follows that $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ iff $\mathbb{P}(A) \equiv_{\mathrm{v}} \mathbb{Q}(A)$ for all $A \subseteq I$ iff $\mathbb{P} \sqcup \mathbb{G}_I \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{G}_I$. □

As a consequence of Theorem 4, the translation-based technique from (Janhunen & Oikarinen 2005, Theorem 5.4) can be used to verify $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ given that $\mathbb{P}$ and $\mathbb{Q}$ have enough visible atoms ($\mathbb{G}_I$ has the EVA property trivially). More specifically, the task is to show that $\mathrm{EQT}(\mathbb{P} \sqcup \mathbb{G}_I, \mathbb{Q} \sqcup \mathbb{G}_I)$ and $\mathrm{EQT}(\mathbb{Q} \sqcup \mathbb{G}_I, \mathbb{P} \sqcup \mathbb{G}_I)$ have no stable models.

The introduction of modular equivalence was much motivated by the need of modularizing the verification of weak equivalence[2]. We believe that such a modularization could be very effective in a setting where Q is an optimized version of P. Typically Q is obtained by making some local modifications to P. In the following, we propose a further strategy to utilize modular equivalence in the task of verifying the visible/weak equivalence of P and Q.

An essential prerequisite is to identify a module structure for P and Q. Basically, there are two ways to achieve this: either the programmer specifies modules explicitly or strongly connected components of $\mathrm{Dep}^+(P)$ and $\mathrm{Dep}^+(Q)$ are computed to detect them automatically. Assuming the relationship of P and Q as described above, it is likely that these components are pairwise compatible and we can partition P and Q so that $P = \mathbb{P}_1 \sqcup \cdots \sqcup \mathbb{P}_n$ and $Q = \mathbb{Q}_1 \sqcup \cdots \sqcup \mathbb{Q}_n$ where the respective modules $\mathbb{P}_i$ and $\mathbb{Q}_i$ have the same input and output. Note that $\mathbb{P}_i$ and $\mathbb{Q}_i$ can be the same for a number of i's under the locality assumption.

In this setting, the verification of $\mathbb{P}_i \equiv_{\mathrm{m}} \mathbb{Q}_i$ for each pair of modules $\mathbb{P}_i$ and $\mathbb{Q}_i$ is not of interest as $\mathbb{P}_i \not\equiv_{\mathrm{m}} \mathbb{Q}_i$ does

[1] This class of programs includes normal logic programs.

[2] Recall that $\equiv_{\mathrm{v}}$ coincides with $\equiv$ for programs P and Q having equal and fully visible Herbrand bases.

not necessarily imply $P \not\equiv_{\mathrm{v}} Q$. However, the verification of $P \equiv_{\mathrm{v}} Q$ can still be organized as a sequence of n tests at the level of modules, i.e. it is sufficient to show

$$\mathbb{Q}_1 \sqcup \cdots \sqcup \mathbb{Q}_{i-1} \sqcup \mathbb{P}_i \sqcup \cdots \sqcup \mathbb{P}_n \equiv_{\mathrm{m}} \mathbb{Q}_1 \sqcup \cdots \sqcup \mathbb{Q}_i \sqcup \mathbb{P}_{i+1} \sqcup \cdots \sqcup \mathbb{P}_n \quad (2)$$

for each $1 \leq i \leq n$ and the resulting chain of equalities conveys $P \equiv_{\mathrm{v}} Q$ under the assumption that P and Q have a completely specified input. If not, then $\equiv_{\mathrm{m}}$ can be addressed using a similar chaining technique based on (2).

Example 5 *Consider normal logic programs P and Q both consisting of two submodules, i.e. $P = \mathbb{P}_1 \sqcup \mathbb{P}_2$ and $Q = \mathbb{Q}_1 \sqcup \mathbb{Q}_2$ where $\mathbb{P}_1, \mathbb{P}_2, \mathbb{Q}_1$, and $\mathbb{Q}_2$ are defined by*

$$\begin{aligned} \mathbb{P}_1 &= (\{c \leftarrow \sim a.\}, \{a, b\}, \{c\}), \\ \mathbb{P}_2 &= (\{a \leftarrow b.\}, \emptyset, \{a, b\}), \\ \mathbb{Q}_1 &= (\{c \leftarrow \sim b.\}, \{a, b\}, \{c\}), \text{ and} \\ \mathbb{Q}_2 &= (\{b \leftarrow a.\}, \emptyset, \{a, b\}). \end{aligned}$$

Now, $\mathbb{P}_1 \not\equiv_{\mathrm{m}} \mathbb{Q}_1$, but $\mathbb{P}_1$ and $\mathbb{Q}_1$ are visibly equivalent in all contexts produced by both $\mathbb{P}_2$ and $\mathbb{Q}_2$ (in this case actually $\mathbb{P}_2 \equiv_{\mathrm{m}} \mathbb{Q}_2$ holds, but that is not necessary). Thus

$$\mathbb{P}_1 \sqcup \mathbb{P}_2 \equiv_{\mathrm{m}} \mathbb{Q}_1 \sqcup \mathbb{P}_2 \equiv_{\mathrm{m}} \mathbb{Q}_1 \sqcup \mathbb{Q}_2,$$

which verifies $P \equiv_{\mathrm{v}} Q$ as well as $P \equiv Q$.

It should be stressed that the programs involved in each test (2) differ in $\mathbb{P}_i$ and $\mathbb{Q}_i$ for which the other modules form a common context, say $\mathbb{C}_i$. A way to optimize the verification of $\mathbb{P}_i \sqcup \mathbb{C}_i \equiv_{\mathrm{m}} \mathbb{Q}_i \sqcup \mathbb{C}_i$ is to view $\mathbb{C}_i$ as a module generating input for $\mathbb{P}_i$ and $\mathbb{Q}_i$ and to adjust the translation-based method from (Janhunen & Oikarinen 2005) for such generators. More specifically, we seek computational advantage from using $\mathrm{EQT}(\mathbb{P}_i, \mathbb{Q}_i) \sqcup \mathbb{C}_i$ rather than $\mathrm{EQT}(\mathbb{P}_i \sqcup \mathbb{C}_i, \mathbb{Q}_i \sqcup \mathbb{C}_i)$ and especially when the context $\mathbb{C}_i$ is clearly larger than the modules $\mathbb{P}_i$ and $\mathbb{Q}_i$. By symmetry, the same strategy is applicable to $\mathbb{Q}_i$ and $\mathbb{P}_i$.

Related Work

The notion of modular equivalence is already contrasted with other equivalence relations in previous sections.

Bugliesi, Lamma and Mello (1994) present an extensive survey of modularity in conventional logic programming. Two mainstream programming disciplines can be identified: *programming-in-the-large* where programs are composed with algebraic operators (O'Keefe 1985) and *programming-in-the-small* with abstraction mechanisms (Miller 1986). Our approach can be classified in the former discipline due to resemblance to that of Gaifman and Shapiro (1989). But stable model semantics and the denial of positive recursion between modules can be pointed out as obvious differences in view of their approach.

A variety of conditions on modules have also been introduced. For instance, in contrast to our work, Maher (1993) forbids all recursion between modules and considers Przymusinski's *perfect models* rather than stable models. Brogi et al. (1994) employ operators for program composition and visibility conditions that correspond to the second item in Definition 8. However, their approach covers only positive programs and the least model semantics. Etalle and Gabbrielli (1996) restrict the composition of *constraint logic program* (CLP) modules with a condition that is close to ours: $\mathrm{Hb}(P) \cap \mathrm{Hb}(Q) \subseteq \mathrm{Hb}_{\mathrm{v}}(P) \cap \mathrm{Hb}_{\mathrm{v}}(Q)$ but no distinction between input and output is made; e.g. $O_P \cap O_Q \neq \emptyset$ is allowed according to their definitions. They also strive for congruence relations but in the case of CLPs.

Eiter, Gottlob, and Mannila (1997) consider the class of *disjunctive Datalog* used as query programs π over relational databases. As regards syntax, such programs are disjunctive programs which cover normal programs (involving variables though) as a special case. The rough idea is that π is instantiated with respect to an input database D for the given input schema $\mathbf{R}$. The resulting models of $\pi[D]$, which depend on the semantics chosen for π, are projected with respect to an output schema $\mathbf{S}$. To link this approach to ours, it is possible to view π as a program module $\mathbb{P}$ with input I and output O based on $\mathbf{R}$ and $\mathbf{S}$, respectively. Then $\pi[D]$ is obtained as $\mathbb{P}(D)$. In contrast to our work, their module architecture is based on both *positive and negative dependencies* and no recursion between modules is tolerated. These constraints enable a straightforward generalization of the splitting set theorem for the architecture.

Faber *et al.* (2005) apply the *magic set method* in the evaluation of Datalog programs with negation, i.e. effectively normal programs. This involves the concept of an *independent set* S of a program P which is a specialization of a splitting set (recall Theorem 1). Roughly speaking, the idea is that the semantics of an independent set S is not affected by the rest of P and thus S gives rise to a *module* $T = \{h \leftarrow B^+, \sim B^- \in P \mid h \in S\}$ of P so that $T \subseteq P$ and $\mathrm{Head}(T) = S$. Due to close relationship to splitting sets, independent sets are not that flexible as regards parceling normal programs. For instance, the splittings demonstrated in Examples 2 and 4 are impossible with independent sets. In certain cases, the distinction of *dangerous rules* in the definition of independent sets pushes negative recursion inside modules which is unnecessary in view of our results. Finally, the module theorem of Faber *et al.* (2005) is weaker than Theorem 2.

Eiter, Gottlob and Veith (1997) address modularity within ASP and view program modules as *generalized quantifiers* the definitions of which are allowed to nest, i.e. P can refer to another module Q by using it as a generalized quantifier. This is an abstraction mechanism typical to programming-in-the-small approaches.

Conclusion

In this article, we a propose a module architecture for logic programs in answer set programming. The compatibility of the module system and stable models is achieved by allowing positive recursion to occur inside modules only. The current design gives rise to a number of interesting results. First, the splitting set theorem by Lifschitz and Turner (1994) is generalized to the case where negative recursion is allowed between modules. Second, the resulting notion of *modular equivalence* is a proper congruence relation for the join operation between modules. Third, the verification

of modular equivalence can be accomplished with existing methods so that specialized solvers need not be developed. Last but not least, we have a preliminary understanding how the task of verifying weak equivalence can be modularized using modular equivalence.

Yet the potential gain from the modular verification strategy has to be evaluated by conducting experiments. A further theoretical question is how the existing model theory based on *SE-models* and *UE-models* (Eiter & Fink 2003) is tailored to the case of modular equivalence. There is also a need to expand the module architecture and module theorem proposed here to cover other classes of logic programs such as e.g. weight constraint programs, disjunctive programs, and nested programs.

References

Brogi, A.; Mancarella, P.; Pedreschi, D.; and Turini, F. 1994. Modular logic programming. *ACM Transactions on Programming Languages and Systems* 16(4):1361–1398.

Bugliesi, M.; Lamma, E.; and Mello, P. 1994. Modularity in logic programming. *Journal of Logic Programming* 19/20:443–502.

Cholewinski, P., and Truszczyński, M. 1999. Extremal problems in logic programming and stable model computation. *Journal of Logic Programming* 38(2):219–242.

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*. New York: Plenum Press. 293–322.

Eiter, T., and Fink, M. 2003. Uniform equivalence of logic programs under the stable model semantics. In *Proc. of the 19th International Conference on Logic Programming*, volume 2916 of *LNCS*, 224–238. Mumbay, India: Springer.

Eiter, T.; Fink, M.; Tompits, H.; and Woltran, S. 2004. Simplifying logic programs under uniform and strong equivalence. In *Proc. of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNAI*, 87–99. Fort Lauderdale, USA: Springer.

Eiter, T.; Gottlob, G.; and Mannila, H. 1997. Disjunctive datalog. *ACM Transactions on Database Systems* 22(3):364–418.

Eiter, T.; Gottlob, G.; and Veith, H. 1997. Modular logic programming and generalized quantifiers. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNCS*, 290–309. Dagstuhl, Germany: Springer.

Eiter, T.; Tompits, H.; and Woltran, S. 2005. On solution correspondences in answer-set programming. In *Proc. of 19th International Joint Conference on Artificial Intelligence*, 97–102. Edinburgh, UK: Professional Book Center.

Etalle, S., and Gabbrielli, M. 1996. Transformations of CLP modules. *Theoretical Computer Science* 166(1–2):101–146.

Faber, W.; Greco, G.; and Leone, N. 2005. Magic sets and their application to data integration. In *Proc. of 10th International Conference on Database Theory, ICDT'05*, volume 3363 of *LNCS*, 306–320. Edinburgh, UK: Springer.

Gaifman, H., and Shapiro, E. 1989. Fully abstract compositional semantics for logic programs. In *Proc. of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 134–142. Austin, Texas, USA: ACM Press.

Gelfond, M., and Leone, N. 2002. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence* 138:3–38.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the 5th International Conference on Logic Programming*, 1070–1080. Seattle, Washington: MIT Press.

Janhunen, T., and Oikarinen, E. 2002. Testing the equivalence of logic programs under stable model semantics. In *Proc. of the 8th European Conference on Logics in Artificial Intelligence*, volume 2424 of *LNAI*, 493–504. Cosenza, Italy: Springer.

Janhunen, T., and Oikarinen, E. 2005. Automated verification of weak equivalence within the SMODELS system. Submitted to Theory and Practice of Logic Programming.

Janhunen, T.; Niemelä, I.; Seipel, D.; Simons, P.; and You, J.-H. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic* 7(1):1–37.

Janhunen, T. 2003. Translatability and intranslatability results for certain classes of logic programs. Series A: Research report 82, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*. Accepted for publication.

Lifschitz, V., and Turner, H. 1994. Splitting a logic program. In *Proc. of the 11th International Conference on Logic Programming*, 23–37. Santa Margherita Ligure, Italy: MIT Press.

Lifschitz, V.; Pearce, D.; and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic* 2(4):526–541.

Maher, M. J. 1993. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science* 110(2):377–403.

Marek, W., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer-Verlag. 375–398.

Miller, D. 1986. A theory of modules for logic programming. In *Proc. of the 1986 Symposium on Logic Programming*, 106–114. Salt Lake City, USA: IEEE Computer Society Press.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Math. and Artificial Intelligence* 25(3-4):241–273.

Oikarinen, E., and Janhunen, T. 2004. Verifying the equivalence of logic programs in the disjunctive case. In *Proc. of the 7th International Conference on Logic Programming*

and Nonmonotonic Reasoning, volume 2923 of *LNAI*, 180–193. Fort Lauderdale, USA: Springer.

O'Keefe, R. A. 1985. Towards an algebra for constructing logic programs. In *Proc. of the 1985 Symposium on Logic Programming*, 152–160.

Sagiv, Y. 1987. Optimizing datalog programs. In *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 349–362. San Diego, USA: ACM Press.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1–2):181–234.

Turner, H. 2003. Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming* 3(4-5):609–622.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8(3):410–421.

Woltran, S. 2004. Characterizations for relativized notions of equivalence in answer set programming. In *Proc. of the 9th European Conference on Logics in Artificial Intelligence*, volume 3229 of *LNAI*, 161–173. Lisbon, Portugal: Springer.