A Translation-based Approach to the Verification of Modular Equivalence*

Emilia Oikarinen** and Tomi Janhunen

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

Abstract. In this paper, a method for verifying the equivalence of logic program modules under a Gaifman-Shapiro-style module architecture is proposed. The idea is to adapt a translation-based verification technique, which was originally devised for complete programs only, for program modules. In addition, optimization strategies are addressed in order to exploit the modular structure of programs in verification tasks. A number of experiments on verification strategies are also conducted using LPEQ which implements the verification method for the SMODELS system. Preliminary results indicate that at least in certain cases the overall time spent on verification tasks can be reduced through modularization.

1 Introduction

Answer set programming (ASP) [16, 18] provides the rule-based syntax of logic programs with a fully declarative nonmonotonic semantics based on stable models [7]. Stable models have a global nature in the sense that they are defined for entire logic programs rather than individual rules. For this reason, logic programs are typically treated as integral entities in ASP and relatively little attention has been paid to modular program development so far. Our motivation is to bring good software engineering practise to the realm of ASP and, in particular, to exploit modules and module architectures in order to ease the development of logic programs. The expected benefits from modular program development are manifold. First, it is conceptually much easier to develop a large logic program in smaller units using modules with a well-defined input/output interface. Thus a module system provides a way to govern complexity as the sizes of program instances grow—a clear trend in the applications of ASP. Second, a fully integrated module system enforces a good programming style for programmers, helps to delegate programming tasks among them, and enables the re-use of code, e.g., organized as libraries. Third, modularity plays also a role in the implementation of inference engines for ASP and exploiting modules in the search for answer sets is also becoming increasingly important as the demands of applications appear to be interminable.

* The research reported in this paper is partially funded by the Academy of Finland (project #211025 "*Advanced Constraint Programming Techniques for Large Structured Problems*").

** The financial support from Helsinki Graduate School in Computer Science and Engineering, Nokia Foundation, Finnish Foundation of Technology (TES), Emil Aaltonen Foundation, and Finnish Cultural Foundation is gratefully acknowledged.

Modularity has been studied quite extensively in the area of *conventional* logic programming (see [2] for a study) before ASP really emerged. Two mainstream disciplines, viz. *programming-in-the-large* where programs are composed with algebraic operators [6] and *programming-in-the-small* with abstraction mechanisms [17], are identified. In ASP, however, only few approaches describe a real module system with a clearly defined interface for module interaction. The approaches based on *generalized quantifiers* [4], *templates* [8], *import rules* [23], and *macros* [1] fall all in the programming-in-the-small category. On the other hand, those in the programming-in-the-large discipline are mostly based on Lifschitz and Turner's splitting set theorem [15] and its variants [3, 5].

Our interest in the modularity aspects of ASP emerged from the expressiveness analysis of normal logic programs and propositional theories [9] where the existence of *modular* and *faithful* translations is used as a criterion for comparisons. Further interest sparked from our previous research on verifying the equivalence of logic programs [10, 20, 11] and, in particular, from the study of equivalence relations better amenable to modularization. To this end, we have based our approach on Gaifman and Shapiro's module architecture [6] in which logic program modules interact through a well-defined *input/output interface*. In [21, 12, 19], we put forward analogous architectures in the context of ASP. The key observation is that the stable model semantics [7] becomes fully compatible with the module system if positively interdependent rules are enforced inside modules. The main result is a *module theorem* which interconnects module-level stability with program-level stability. This indicates that the global nature of stable models as discussed above is much illusory. Moreover, module-level equivalence gives rise to the notion of *modular equivalence* which is a proper *congruence relation* for program composition, i.e., it is preserved under substitutions of equivalent modules.

In the current paper, we are interested in the problem of verifying whether logic program modules are equivalent or not. In view of methods for solving this problem, the idea is to adopt our translation-based verification technique [11]. In this approach, the idea is to translate programs P and Q under consideration into another logic program $\mathrm{EQT}(P, Q)$ which has no stable models if every stable model of P is also a stable model of Q. Our main objective is to generalize this method for the verification of modular equivalence. Moreover, we aim at exploiting program modules in the presentation of the method itself. In analogy to [11], the syntax of programs is that of SMODELS programs.

We proceed according to the following plan for the rest of this paper. In Section 2, we give a more technical account of logic program modules in the context of the SMODELS system. In addition to the Gaifman-Shapiro-style module architecture, the stable model semantics of program modules is defined and shown to be compatible with the architecture. As the final outcome, the notion of modular equivalence is formalized and the computational complexity of the respective verification problem is briefly addressed. Next, we concentrate on the translation-based verification technique proposed for SMODELS programs [11] in Section 3. The method is revised to to the case of modular equivalence and represented in a modular fashion using program modules as natural building blocks. In addition, we address optimization strategies which try to exploit program modules in the verification task. Then, the integration of program modules in our verification tool, namely LPEQ, is described in Section 4. A number of experiments on verification strategies are also conducted and reported. Section 5 concludes the paper.

2 SMODELS Program Modules

In what follows, we provide a brief introduction to a Gaifman-Shapiro-style module architecture [6] that is presented for SMODELS programs in [19]. Analogous module systems have already been utilized within ASP, e.g., in the contexts of *normal* and *disjunctive logic programs* [21, 12]. In the rest of section, we address four aspects of SMODELS program modules, viz. their syntax, composition, semantics, and equivalence.

To keep the presentation of the module architecture compatible with an actual implementation, we cover the input language of the SMODELS system—excluding *optimization statements*. *Basic constraint rules* [22] are either *basic rules* of the form $a \leftarrow B, \sim C$, *weight rules* of the form $a \leftarrow w \leq \{B = W_B, \sim C = W_C\}$, or *choice rules* of the form $\{A\} \leftarrow B, \sim C$ where a is an atom and $A \neq \emptyset$, B, and C are sets of atoms, and $\sim$ denotes *negation as failure*. In addition, a weight rule involves a weight limit $w \in \mathbb{N}$ and the sets of weights $W_B, W_C \subseteq \mathbb{N}$ which associate the respective weights $w_b \in W_B$ and $w_c \in W_C$ with each atom $b \in B$ and $c \in C$. We may distinguish two parts for each basic constraint rule: a or A is the *head* of the rule and the rest is called the *body* which gives the conditions on which the head is activated. E.g., in case of a choice rule $\{A\} \leftarrow B, \sim C$, this means that any atom from A can be inferred if all atoms in B and none of the atoms in C can be inferred. In the sequel, we view a basic rule $a \leftarrow B, \sim C$ as a shorthand for a weight rule $a \leftarrow |B| + |C| \leq \{B = \mathbf{1}, \sim C = \mathbf{1}\}$[1] and omit similar definitions of *constraint rules* and *compute statements* that can be found in [11]. The exact model-theoretic semantics is deferred until Section 2.2.

Given a set R of basic constraint rules, we write $\mathrm{Hb}(R)$ for its signature, i.e., the set of atoms occurring in R, and $\mathrm{Head}(R)$ for the respective subset of $\mathrm{Hb}(R)$ having *head occurrences* in R. An individual SMODELS program module is structured as follows.

Definition 1. *An* SMODELS *program module $\mathbb{P}$ is a quadruple $\langle R, I, O, H\rangle$ where*

1. *R is a finite set of basic constraint rules;*
2. *I, O, and H are pairwise disjoint sets of input, output, and hidden atoms;*
3. *$\mathrm{Hb}(R) \subseteq \mathrm{Hb}(\mathbb{P})$ which is defined by $\mathrm{Hb}(\mathbb{P}) = I \cup O \cup H$; and*
4. *$\mathrm{Head}(R) \cap I = \emptyset$.*

The atoms in $\mathrm{Hb}_{\mathrm{v}}(\mathbb{P}) = I \cup O$ are considered to be *visible* and hence accessible to other modules conjoined with $\mathbb{P}$; either to produce input for $\mathbb{P}$ or to utilize the output of $\mathbb{P}$. We use notations $\mathrm{Hb}_{\mathrm{i}}(\mathbb{P})$ and $\mathrm{Hb}_{\mathrm{o}}(\mathbb{P})$ for referring to I and O, respectively. The *hidden* atoms in $\mathrm{Hb}_{\mathrm{h}}(\mathbb{P}) = H = \mathrm{Hb}(\mathbb{P}) \setminus \mathrm{Hb}_{\mathrm{v}}(\mathbb{P})$ are used to formalize some auxiliary concepts of $\mathbb{P}$ which may not be sensible for other modules but may save space substantially, cf. [11, Example 4.5]. The condition $\mathrm{Head}(R) \cap I = \emptyset$ ensures that a module may not interfere with its own input by defining input atoms of I in terms of its rules. Thus the rules of $\mathbb{P}$ may be conditioned by input atoms appearing in rule bodies only.

Example 1. Consider the classical pigeon hole principle in the case of n holes and $n + 1$ pigeons. This can be formalized using two SMODELS program modules $\mathbb{P}_n = \langle R_P, \emptyset, O_P, \emptyset\rangle$ and $\mathbb{H}_n = \langle R_H, O_P, \emptyset, \{e, f\}\rangle$ as follows. Let $p(i, j)$ be an atom denoting that the i^{th} pigeon is in the j^{th} hole where $0 < i \leq n + 1$ and $0 < j \leq n$. The

[1] Here the set of weights $\mathbf{1}$ associates the weight 1 with every atom in the set in question.

signature $\mathrm{Hb_o}(\mathbb{P}_n) = O_P = \mathrm{Hb_i}(\mathbb{H}_n) = \{p(i,j) \mid 0 < i \leq n+1,\ 0 < j \leq n\}$. The set of rules R_P contains a choice rule $\{p(i,1), \ldots, p(i,n)\}$ for each $0 < i \leq n+1$. Thus, roughly speaking, the purpose of $\mathbb{P}_n$ is to choose holes for $n+1$ pigeons. The role of $\mathbb{H}_n$ is to check that every pigeon is placed in at least one hole and no two pigeons share a hole. The first condition is captured with a weight rule $e \leftarrow n \leq \{\sim p(i,1) = 1, \ldots, \sim p(i,n) = 1\}$ for each $0 < i \leq n+1$. For the second, we introduce $e \leftarrow 2 \leq \{p(i,k) = 1,\ p(j,k) = 1\}$ for each $0 < i < j \leq n+1$ and $0 < k \leq n$. Finally, we express the need of *discarding* any assignment of pigeons that does not satisfy the two conditions above using a basic rule $f \leftarrow e, \sim f$. In particular, we note that e and f are auxiliary atoms that are hidden from other modules conjoined with $\mathbb{H}_n$. ■

2.1 Syntactic Conditions for Combining Modules

The stable model semantics of normal logic programs [7] does not lend itself directly for program composition. The problem is that in general, stable models associated with modules do not determine stable models assigned to their *composition*. As demonstrated in [21], the stable models assigned to $\mathbb{P}_1 = \langle \{a \leftarrow b\}, \{b\}, \{a\}, \emptyset \rangle$ and $\mathbb{P}_2 = \langle \{b \leftarrow a\}, \{a\}, \{b\}, \emptyset \rangle$ are $\emptyset$ and $\{a, b\}$ by symmetry, but $\{a, b\}$ is not a stable model of $\mathbb{P} = \langle \{a \leftarrow b, b \leftarrow a\}, \emptyset, \{a, b\}, \emptyset \rangle$ that represents the composition of $\mathbb{P}_1$ and $\mathbb{P}_2$ (see below). Gaifman and Shapiro [6] address positive normal programs under logical consequence. For their purposes, it is sufficient to assume that whenever two modules $\mathbb{P}_1$ and $\mathbb{P}_2$ are put together, their output signatures have to be disjoint and they have to *respect each other's hidden atoms*, i.e., $\mathrm{Hb_h}(\mathbb{P}_1) \cap \mathrm{Hb}(\mathbb{P}_2) = \emptyset$ and $\mathrm{Hb_h}(\mathbb{P}_2) \cap \mathrm{Hb}(\mathbb{P}_1) = \emptyset$.

Definition 2. *The* composition *of* $\mathbb{P}_1 = \langle R_1, I_1, O_1, H_1 \rangle$ *and* $\mathbb{P}_2 = \langle R_2, I_2, O_2, H_2 \rangle$ *is*

$$\mathbb{P}_1 \oplus \mathbb{P}_2 = \langle R_1 \cup R_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2 \rangle$$

if $\mathrm{Hb_o}(\mathbb{P}_1) \cap \mathrm{Hb_o}(\mathbb{P}_2) = \emptyset$ *and* $\mathbb{P}_1$ *and* $\mathbb{P}_2$ *respect each other's hidden atoms.*

Recalling the example above, the conditions given for $\oplus$ are not enough to guarantee compositionality in the case of stable models and further restrictions for program composition become necessary. Given $\mathbb{P} = \langle R, I, O, H \rangle$ and $a, b \in \mathrm{Hb}(\mathbb{P})$, we say that a *depends directly* on b, denoted by $b \leq_1 a$, iff R contains a weight rule $a \leftarrow w \leq \{B = W_B, \sim C = W_C\}$ with $b \in B$, or a choice rule $\{A\} \leftarrow B, \sim C$ with $a \in A$ and $b \in B$. The *positive dependency graph* of $\mathbb{P}$, denoted by $\mathrm{Dep}^+(\mathbb{P})$, is the graph $\langle \mathrm{Hb}(\mathbb{P}), \leq_1 \rangle$. The reflexive and transitive closure of $\leq_1$ gives rise to the dependency relation $\leq$ over $\mathrm{Hb}(\mathbb{P})$. A *strongly connected component* (SCC) S of $\mathrm{Dep}^+(\mathbb{P})$ is a maximal set $S \subseteq \mathrm{Hb}(\mathbb{P})$ such that $b \leq a$ holds for every $a, b \in S$. Given that $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined, we say that $\mathbb{P}_1$ and $\mathbb{P}_2$ are *mutually dependent* iff $\mathrm{Dep}^+(\mathbb{P}_1 \oplus \mathbb{P}_2)$ has a SCC S shared by $\mathbb{P}_1$ and $\mathbb{P}_2$ such that $S \cap \mathrm{Hb_o}(\mathbb{P}_1) \neq \emptyset$ and $S \cap \mathrm{Hb_o}(\mathbb{P}_2) \neq \emptyset$ [21, 19].

Definition 3. *The* join, $\mathbb{P}_1 \sqcup \mathbb{P}_2$, *of two* SMODELS *program modules* $\mathbb{P}_1$ *and* $\mathbb{P}_2$ *is* $\mathbb{P}_1 \oplus \mathbb{P}_2$, *providing* $\mathbb{P}_1 \oplus \mathbb{P}_2$ *is defined and* $\mathbb{P}_1$ *and* $\mathbb{P}_2$ *are mutually independent.*

Example 2. Recall modules $\mathbb{P}_n$ and $\mathbb{H}_n$ from Example 1. The composition $\mathbb{P}_n \oplus \mathbb{H}_n = \langle R_P \cup R_H, \emptyset, O_P, \{e, f\} \rangle$ is clearly defined because $\mathrm{Hb_o}(\mathbb{P}_n) \cap \mathrm{Hb_o}(\mathbb{H}_n) = \emptyset$ and $\mathbb{P}_n$ does not use the hidden atoms of $\mathbb{H}_n$. Moreover, we note that modules $\mathbb{P}_n$ and $\mathbb{H}_n$ are mutually independent since the strongly connected components in $\mathrm{Dep}^+(\mathbb{P}_n \oplus \mathbb{H}_n)$ are all singletons. Hence the join $\mathbb{P}_n \sqcup \mathbb{H}_n$ is also defined. ■

2.2 Stable Model Semantics for SMODELS Program Modules

The stable model semantics of normal logic programs [7] can be generalized for SMODELS programs in an analogous way using the notions of a *reduct* and the *least model* of a negation-free program. In order to cover modules as well, we must explicate the semantical role of input atoms. To this end, we will follow an approach[2] from [12] and take input atoms into account in the definition of the reduct adopted from [11]. It should be stressed that all negative literals and literals involving input atoms get evaluated in the reduction. Moreover, our definitions become equivalent with those proposed for normal programs [7] and SMODELS programs [11] if an empty input signature $I = \emptyset$ is additionally assumed. Using the same idea, a conventional SMODELS program, i.e., a set of SMODELS rules R, can be viewed as a module $\langle R, \emptyset, \mathrm{Hb}(R), \emptyset \rangle$.

An *interpretation* M is a subset of $\mathrm{Hb}(\mathbb{P})$ defining which atoms $a \in \mathrm{Hb}(\mathbb{P})$ are *true* ($a \in M$) and *false* ($a \notin M$). A weight rule $a \leftarrow w \leq \{B = W_B, {\sim}C = W_C\}$ is satisfied in M iff $a \in M$ whenever the sum of weights $\sum_{b \in B \cap M} w_b + \sum_{c \in C \setminus M} w_c$ is at least w. A choice rule $\{A\} \leftarrow B, {\sim}C$ is always satisfied in M.

Definition 4. *Given a module $\mathbb{P} = \langle R, I, O, H \rangle$, the* reduct *of R with respect to an interpretation $M \subseteq \mathrm{Hb}(\mathbb{P})$, denoted by R^M, contains*

1. *the basic rule $a \leftarrow (B \setminus I)$ iff there is a choice rule $\{A\} \leftarrow B, {\sim}C$ in R such that $a \in A \cap M$, $B \cap I \subseteq M$, and $M \cap C = \emptyset$; and*
2. *the reduced weight rule $a \leftarrow w' \leq \{B' = W_{B'}\}$ iff there is a weight rule $a \leftarrow w \leq \{B = W_B, {\sim}C = W_C\}$ in R, $W_{B'}$ is the restriction of W_B on $B' = B \setminus I$, and the limit $w' = \max(0, \sum_{b \in B \cap I \cap M} w_b + \sum_{c \in C \setminus M} w_c)$.*

An interpretation $M \subseteq \mathrm{Hb}(\mathbb{P})$ is a stable model of an SMODELS *program module $\mathbb{P} = \langle R, I, O, H \rangle$, denoted by $M \in \mathrm{SM}(\mathbb{P})$, iff $M \setminus I = \mathrm{LM}(R^M)$.*

The generalized reduct R^M is a positive program in the sense of [11] and thus it has a unique least model $\mathrm{LM}(R^M)$. Having the semantics of SMODELS program modules defined, we may characterize its properties under program composition using the notion of *compatibility*. Given modules $\mathbb{P}_1$ and $\mathbb{P}_2$, we say that interpretations $M_1 \subseteq \mathrm{Hb}(\mathbb{P}_1)$ and $M_2 \subseteq \mathrm{Hb}(\mathbb{P}_2)$ are *compatible* iff $M_1 \cap \mathrm{Hb}_{\mathrm{v}}(\mathbb{P}_2) = M_2 \cap \mathrm{Hb}_{\mathrm{v}}(\mathbb{P}_1)$. For sets of interpretations $A_1 \subseteq \mathbf{2}^{\mathrm{Hb}(\mathbb{P}_1)}$ and $A_2 \subseteq \mathbf{2}^{\mathrm{Hb}(\mathbb{P}_2)}$, the *natural join* of A_1 and A_2, denoted by $A_1 \bowtie A_2$, is $\{M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2, \text{ and } M_1 \text{ and } M_2 \text{ are compatible}\}$.

Theorem 1 (Module theorem [19]). *If $\mathbb{P}_1$ and $\mathbb{P}_2$ are* SMODELS *program modules such that $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined, then $\mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$.*

It is worth noting that classical propositional theories have an analogous property obtained by substituting $\cup$ for $\sqcup$ and replacing stable models by classical models. Moreover, Theorem 1 is not applicable to modules $\mathbb{P}_1$ and $\mathbb{P}_2$ addressed in the beginning of Section 2.1 as $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is not defined. However, consider $\mathbb{Q} = \langle \{a \leftarrow {\sim}b\}, \{b\}, \{a\}, \emptyset \rangle$ in the context of $\mathbb{P}_2 = \langle \{b \leftarrow a\}, \{a\}, \{b\}, \emptyset \rangle$. Since $\mathbb{Q} \sqcup \mathbb{P}_2$ is defined, $\mathrm{SM}(\mathbb{Q}) = \{\{a\}, \{b\}\}$, and $\mathrm{SM}(\mathbb{P}_2) = \{\emptyset, \{a, b\}\}$, we have $\mathrm{SM}(\mathbb{Q} \sqcup \mathbb{P}_2) = \emptyset$ by Theorem 1.

[2] There are also alternative ways to handle input atoms: One possibility is to combine a module with a set of facts (or a database) over its input signature [21, 19]. Yet another approach is to interpret input atoms as *fixed atoms* in the sense of parallel circumscription [13].

Example 3. In view of our pigeon hole example, the module $\mathbb{P}_n$ has plenty of stable models. In fact, any subset M of O_P is stable for $\mathbb{P}_n$. On the hand, the module $\mathbb{H}_n$ has no stable models which implies by Theorem 1 that the join $\mathbb{P}_n \sqcup \mathbb{H}_n$ has no stable models. This is in accordance with the pigeon hole principle captured by $\mathbb{P}_n \sqcup \mathbb{H}_n$. ■

2.3 Visible and Modular Equivalence

The notion of *visible equivalence* [9] was introduced in order to neglect hidden atoms when logic programs, or other theories of interest, are compared on the basis of their models. The compositionality property from Theorem 1 enables us to bring the same idea to the level of program modules—giving rise to *modular equivalence* of logic programs [21].[3] Visible and modular equivalence, denoted by the respective infix relation symbols $\equiv_{\mathrm{v}}$ and $\equiv_{\mathrm{m}}$, are formulated for SMODELS program modules as follows.

Definition 5. *For two* SMODELS *program modules* $\mathbb{P}$ *and* $\mathbb{Q}$*,*

- $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$ *iff* $\mathrm{Hb}_{\mathrm{v}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{v}}(\mathbb{Q})$ *and there is a bijection* $f : \mathrm{SM}(\mathbb{P}) \to \mathrm{SM}(\mathbb{Q})$ *such that for all* $M \in \mathrm{SM}(\mathbb{P})$*,* $M \cap \mathrm{Hb}_{\mathrm{v}}(\mathbb{P}) = f(M) \cap \mathrm{Hb}_{\mathrm{v}}(\mathbb{Q})$*; and*
- $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ *iff* $\mathrm{Hb}_{\mathrm{i}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{i}}(\mathbb{Q})$ *and* $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$*.*

It is worth noting that the condition $\mathrm{Hb}_{\mathrm{v}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{v}}(\mathbb{Q})$, as insisted by $\equiv_{\mathrm{v}}$, implies $\mathrm{Hb}_{\mathrm{o}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{o}}(\mathbb{Q})$ in the presence of $\mathrm{Hb}_{\mathrm{i}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{i}}(\mathbb{Q})$, as required by $\equiv_{\mathrm{m}}$. Moreover, the two relations coincide for *completely specified* SMODELS programs $\mathbb{P}$ which satisfy $\mathrm{Hb}_{\mathrm{i}}(\mathbb{P}) = \emptyset$ and hence require the presence of no further modules defining atoms. Modular equivalence lends itself for program substitutions in analogy to *strong equivalence* [14], i.e., the relation $\equiv_{\mathrm{m}}$ is a proper *congruence* for $\sqcup$.

Corollary 1 (Congruence). *Let* $\mathbb{P}, \mathbb{Q}$ *and* $\mathbb{R}$ *be* SMODELS *program modules such that* $\mathbb{P} \sqcup \mathbb{R}$ *and* $\mathbb{Q} \sqcup \mathbb{R}$ *are defined. If* $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$*, then* $\mathbb{P} \sqcup \mathbb{R} \equiv_{\mathrm{m}} \mathbb{Q} \sqcup \mathbb{R}$*.*

Example 4. The choice regarding the placement of pigeons in holes can be reformulated as an alternative module $\mathbb{Q}_n = \langle R_Q, \emptyset, O_P, H_Q \rangle$ where R_Q contains two basic rules $p(i,k) \leftarrow {\sim}q(i,k)$ and $q(i,k) \leftarrow {\sim}p(i,k)$ for each $0 < i \leq n+1$ and $0 < k \leq n$. The idea is to hide $H_Q = \{q(i,k) \mid 0 < i \leq n+1,\ 0 < k \leq n\}$. Now $\mathbb{P}_n \equiv_{\mathrm{m}} \mathbb{Q}_n$ which implies $\mathbb{P}_n \sqcup \mathbb{H}_n \equiv_{\mathrm{m}} \mathbb{Q}_n \sqcup \mathbb{H}_n$ by Corollary 1 as $\mathbb{Q}_n \sqcup \mathbb{H}_n$ is defined by the same arguments as for $\mathbb{P}_n \sqcup \mathbb{H}_n$. Thus $\mathrm{SM}(\mathbb{Q}_n \sqcup \mathbb{H}_n) = \emptyset$ on the basis of Example 3. ■

Due to the close relationship of $\equiv_{\mathrm{v}}$ and $\equiv_{\mathrm{m}}$, the respective verification problems have the same computational complexity. As analyzed in [11], the verification of $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$ involves a counting problem in general but a reduction of computational time complexity can be achieved for modules that have *enough visible atoms*, i.e., the EVA property. In order to formalize this property, we define the *hidden part* of a module $\mathbb{P} = \langle R, I, O, H \rangle$ as $\mathbb{P}_{\mathrm{h}} = \langle R_{\mathrm{h}}, I \cup O, H, \emptyset \rangle$ where R_{h} contains all rules of R involving atoms of H in their heads. For such a choice rule $\{A\} \leftarrow B, {\sim}C$, we take $\{A \cap H\} \leftarrow B, {\sim}C$ in R_{h}. Now $\mathbb{P}$ has the EVA property if and only if $\mathrm{SM}(\mathbb{P}_{\mathrm{h}})$ contains a unique stable model M for each interpretation $N \subseteq \mathrm{Hb}_{\mathrm{v}}(\mathbb{P}) = I \cup O$ such that $M \cap (I \cup O) = N$. As established in [19], the verification of $\equiv_{\mathrm{m}}$ forms a **coNP**-complete decision problem for SMODELS program modules with the EVA property. Such a level of computational complexity enables the use of SMODELS for verification.

[3] The reader is referred to [21, 12, 19] for other notions of equivalence as well as comparisons.

3 Verification of Modular Equivalence

Since modular equivalence can be reduced to visible equivalence [21], the translation-based technique from [11, Corollary 5.9] can be used to verify $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ given that SMODELS program modules $\mathbb{P}$ and $\mathbb{Q}$ have enough visible atoms. More specifically, using the *translation function* EQT in [11], the task is to show that SMODELS programs $\mathrm{EQT}(\mathbb{P} \sqcup \mathbb{G}_I, \mathbb{Q} \sqcup \mathbb{G}_I)$ and $\mathrm{EQT}(\mathbb{Q} \sqcup \mathbb{G}_I, \mathbb{P} \sqcup \mathbb{G}_I)$ have no stable models, where $\mathbb{G}_I = \langle \{\{I\} \leftarrow\}, \emptyset, I, \emptyset \rangle$ is a module generating all possible inputs for an input signature I.

However, there is still room for improvement since the common context $\mathbb{G}_I$ is handled separately for the modules involved. To adjust the translation-based method for the verification of modular equivalence we define a modified, modular version of EQT.

Definition 6. *Let $\mathbb{P} = \langle R_P, I, O, H_P \rangle$ and $\mathbb{Q} = \langle R_Q, I, O, H_Q \rangle$ be* SMODELS *program modules having enough visible atoms. The translation*

$$\mathrm{EQT}(\mathbb{P}, \mathbb{Q}) = \mathbb{P} \sqcup \mathrm{Hidden}^{\circ}(\mathbb{Q}) \sqcup \mathrm{Least}^{\bullet}(\mathbb{Q}) \sqcup \mathrm{UnStable}(\mathbb{Q})$$

combines $\mathbb{P}$ with modules $\mathrm{Hidden}^{\circ}(\mathbb{Q})$, $\mathrm{Least}^{\bullet}(\mathbb{Q})$, and $\mathrm{UnStable}(\mathbb{Q})$ presented in detail in Figure 1.

Now, the translation of SMODELS program modules $\mathbb{P}$ and $\mathbb{Q}$ is the module

$$\mathrm{EQT}(\mathbb{P}, \mathbb{Q}) = \langle R, I, O \cup O^{\bullet} \cup H_Q^{\bullet} \cup H_Q^{\circ}, H_P \cup \{d, f\} \rangle,$$

where R is the set of rules introduced by lines 1–8 in Figure 1 for $\mathbb{Q}$ together with the rules in R_P. The translation EQT introduces new atoms not appearing in $\mathrm{Hb}(\mathbb{P}) \cup \mathrm{Hb}(\mathbb{Q})$: d, f, a° for each $a \in \mathrm{Hb_h}(\mathbb{Q})$, and $a^{\bullet}$ for each $a \in \mathrm{Hb_o}(\mathbb{Q}) \cup \mathrm{Hb_h}(\mathbb{Q})$.

Intuitively, the modules in the translation work as follows. (i) The module $\mathbb{P}$ naturally captures a stable model $M \in \mathrm{SM}(\mathbb{P})$. (ii) The module $\mathrm{Hidden}^{\circ}(\mathbb{Q})$ includes rules that provide a representation for the hidden part of $\mathbb{Q}$ evaluated with respect to the visible part of M. This is achieved by taking the visible atoms from $\mathrm{Hb_v}(\mathbb{Q}) = \mathrm{Hb_v}(\mathbb{P}) = I \cup O$ to be in the input of $\mathrm{Hidden}^{\circ}(\mathbb{Q})$ and leaving their occurrences untouched in the rules. The hidden parts of rules are renamed systematically using atoms from $\mathrm{Hb_h}(\mathbb{Q})^{\circ}$. This is to capture the unique stable model N for $\mathbb{Q}$ such that $N \cap \mathrm{Hb_v}(\mathbb{Q}) = M \cap \mathrm{Hb_v}(\mathbb{Q})$ expressed in $\mathrm{Hb_v}(\mathbb{Q}) \cup \mathrm{Hb_h}(\mathbb{Q})^{\circ}$ rather than $\mathrm{Hb}(\mathbb{Q})$. Note that the existence and uniqueness of such N is guaranteed by the EVA property. (iii) The rules in $\mathrm{Least}^{\bullet}(\mathbb{Q})$ catch the least model $\mathrm{LM}((R_Q)^N)$. Thus $\mathrm{Least}^{\bullet}(\mathbb{Q})$ gets an interpretation for $\mathbb{Q}$ expressed in $\mathrm{Hb_v}(\mathbb{Q}) \cup \mathrm{Hb_h}(\mathbb{Q})^{\circ}$ as input. In the rules we rename systematically the positive occurrences of atoms in $\mathrm{Hb_o}(\mathbb{Q}) \cup \mathrm{Hb_h}(\mathbb{Q})$ in order to capture the least model expressed in $O^{\bullet} \cup H_Q^{\bullet}$ rather than $O \cup H_Q$. (iv) The purpose of $\mathrm{UnStable}(\mathbb{Q})$ is to disqualify N as a stable model of $\mathbb{Q}$, i.e., to show that $N \setminus I$ and $\mathrm{LM}((R_Q)^N)$, respectively expressed in $O \cup H_Q^{\circ}$ and $O^{\bullet} \cup H_Q^{\bullet}$, differ. Atom d is used to indicate that there is a difference between $N \setminus I$ and $\mathrm{LM}((R_Q)^N)$. It is then insisted by the last rule that d is true in each stable model of $\mathrm{UnStable}(\mathbb{Q})$. A more detailed discussion on the ideas behind the translation EQT can be found in [11].

Note that the rules in modules $\mathrm{Hidden}^{\circ}(\mathbb{Q})$, $\mathrm{Least}^{\bullet}(\mathbb{Q})$, and $\mathrm{UnStable}(\mathbb{Q})$ are very similar to rules in [11, Definitions 5.2–5.4]. The modifications are as follows. In the

Module	:	$\mathrm{Hidden}^{\circ}(\mathbb{Q})$
Input	:	$I \cup O$
Output	:	H°
Hidden	:	$\emptyset$
1	:	$\{A^{\circ}_{\mathrm{h}}\} \leftarrow B^{\circ}_{\mathrm{h}}, B_{\mathrm{v}}, {\sim}C^{\circ}_{\mathrm{h}}, {\sim}C_{\mathrm{v}}$ for each $\{A\} \leftarrow B, {\sim}C \in R$ with $A_{\mathrm{h}} \neq \emptyset$
2	:	$a^{\circ} \leftarrow w \leq \{B^{\circ}_{\mathrm{h}} \cup B_{\mathrm{v}} = W_B, {\sim}(C^{\circ}_{\mathrm{h}} \cup C_{\mathrm{v}}) = W_C\}$ for each $a \leftarrow w \leq \{B = W_B, {\sim}C = W_C\} \in R$ with $a \in H$
Module	:	$\mathrm{Least}^{\bullet}(\mathbb{Q})$
Input	:	$I \cup O \cup H^{\circ}$
Output	:	$O^{\bullet} \cup H^{\bullet}$
Hidden	:	$\emptyset$
3	:	$a^{\bullet} \leftarrow a, B_{\mathrm{i}}, B^{\bullet}_{\mathrm{o}}, B^{\bullet}_{\mathrm{h}}, {\sim}C_{\mathrm{v}}, {\sim}C^{\circ}_{\mathrm{h}}$ for each $\{A\} \leftarrow B, {\sim}C \in R$ and $a \in A_{\mathrm{v}}$
4	:	$a^{\bullet} \leftarrow a^{\circ}, B_{\mathrm{i}}, B^{\bullet}_{\mathrm{o}}, B^{\bullet}_{\mathrm{h}}, {\sim}C_{\mathrm{v}}, {\sim}C^{\circ}_{\mathrm{h}}$ for each $\{A\} \leftarrow B, {\sim}C \in R$ and $a \in A_{\mathrm{h}}$
5	:	$a^{\bullet} \leftarrow w \leq \{B_{\mathrm{i}} \cup B^{\bullet}_{\mathrm{o}} \cup B^{\bullet}_{\mathrm{h}} = W_B, {\sim}(C_{\mathrm{v}} \cup C^{\circ}_{\mathrm{h}}) = W_C\}$ for each $a \leftarrow w \leq \{B = W_B, {\sim}C = W_C\} \in R$
Module	:	$\mathrm{UnStable}(\mathbb{Q})$
Input	:	$O \cup O^{\bullet} \cup H^{\bullet} \cup H^{\circ}$
Output	:	$\emptyset$
Hidden	:	$\{d, f\}$
6	:	$d \leftarrow a, {\sim}a^{\bullet}$ and $d \leftarrow a^{\bullet}, {\sim}a$ for each $a \in O$
7	:	$d \leftarrow a^{\circ}, {\sim}a^{\bullet}$ and $d \leftarrow a^{\bullet}, {\sim}a^{\circ}$ for each $a \in H$
8	:	$f \leftarrow {\sim}f, {\sim}d$

Fig. 1. Submodules for translation $\mathrm{EQT}(\mathbb{P}, \mathbb{Q}) = \mathbb{P} \sqcup \mathrm{Hidden}^{\circ}(\mathbb{Q}) \sqcup \mathrm{Least}^{\bullet}(\mathbb{Q}) \sqcup \mathrm{UnStable}(\mathbb{Q})$ for modules $\mathbb{P}$ and $\mathbb{Q} = \langle R, I, O, H \rangle$. We use shorthands $A_{\mathrm{o}} = A \cap O$, $A_{\mathrm{i}} = A \cap I$, $A_{\mathrm{v}} = A_{\mathrm{i}} \cup A_{\mathrm{o}}$, and $A_{\mathrm{h}} = A \cap H$, for any set of atoms $A \subseteq \mathrm{Hb}(\mathbb{Q})$. Each $a^{\bullet}$ and a° is a new atom not appearing in $\mathrm{Hb}(\mathbb{P}) \cup \mathrm{Hb}(\mathbb{Q})$, and $A^{\bullet} = \{a^{\bullet} \mid a \in A\}$ and $A^{\circ} = \{a^{\circ} \mid a \in A\}$ for any set of atoms A. Also d and f are new atoms not appearing in $\mathrm{Hb}(\mathbb{P}) \cup \mathrm{Hb}(\mathbb{Q})$.

rules of $\mathrm{Least}^{\bullet}(\mathbb{Q})$ and $\mathrm{UnStable}(\mathbb{Q})$ atoms in I are not renamed, i.e., atom a is used instead of $a^{\bullet}$ for each $a \in I$; and in $\mathrm{UnStable}(\mathbb{Q})$ the difference rules are introduced only for atoms in $O \cup H_Q$. Furthermore, *compute statements* are seen as a special case of weight rules. Note that for modules with a completely specified input, i.e., $\mathbb{P}$ and $\mathbb{Q}$ with $\mathrm{Hb}_{\mathrm{i}}(\mathbb{P}) = \mathrm{Hb}_{\mathrm{i}}(\mathbb{Q}) = \emptyset$, the translation EQT given here results in basically the same set of rules as the one presented in [11]. An example of the use of EQT follows.

Example 5. Consider SMODELS program modules $\mathbb{P}_3 = \langle \{p \leftarrow p, q\}, \{q\}, \{p\}, \emptyset \rangle$ and $\mathbb{P}_4 = \langle \{p \leftarrow {\sim}p, q\}, \{q\}, \{p\}, \emptyset \rangle$. From a stable model of translation $\mathrm{EQT}(\mathbb{P}_3, \mathbb{P}_4) = \mathbb{P}_3 \sqcup \mathrm{Hidden}^{\circ}(\mathbb{P}_4) \sqcup \mathrm{Least}^{\bullet}(\mathbb{P}_4) \sqcup \mathrm{UnStable}(\mathbb{P}_4)$ we get a counter-example for $\mathbb{P}_3 \equiv_{\mathrm{m}} \mathbb{P}_4$. Here, $\mathrm{Hidden}^{\circ}(\mathbb{P}_4)$ has no rules, $\mathrm{Least}^{\bullet}(\mathbb{P}_4) = \langle \{p^{\bullet} \leftarrow {\sim}p, q\}, \{p, q\}, \{p^{\bullet}\}, \emptyset \rangle$, $\mathrm{UnStable}(\mathbb{P}_4) = \langle \{d \leftarrow p, {\sim}p^{\bullet}.\ d \leftarrow p^{\bullet}, {\sim}p.\ f \leftarrow {\sim}f, {\sim}d\}, \{p, p^{\bullet}\}, \emptyset, \{d, f\} \rangle$, and it is easy to see that $M = \{p^{\bullet}, q, d\} \in \mathrm{SM}(\mathrm{EQT}(\mathbb{P}_3, \mathbb{P}_4))$. Now, $\{q\} = M \cap \mathrm{Hb}(\mathbb{P}_3) \in \mathrm{SM}(\mathbb{P}_3)$ and $\{q\} = M \cap \mathrm{Hb}(\mathbb{P}_4) \notin \mathrm{SM}(\mathbb{P}_4)$ since $\mathrm{LM}((\mathbb{P}_4)^{\{q\}}) = \{p\}$. ■

Given modules $\mathbb{P}$ and $\mathbb{Q}$, we say that $\mathbb{P}$ is *compatible* with $\mathbb{Q}$, if $\mathrm{Hb_i}(\mathbb{P}) = \mathrm{Hb_i}(\mathbb{Q})$ and $\mathrm{Hb_o}(\mathbb{P}) = \mathrm{Hb_o}(\mathbb{Q})$. Theorem 2 shows the correctness of the translation-based method for verification of modular equivalence.

Theorem 2. *Let $\mathbb{P}$ and $\mathbb{Q}$ be compatible* SMODELS *program modules having enough visible atoms. Then $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$ iff $\mathrm{SM}(\mathrm{EQT}(\mathbb{P}, \mathbb{Q})) = \mathrm{SM}(\mathrm{EQT}(\mathbb{Q}, \mathbb{P})) = \emptyset$.*

The proof of Theorem 2 is similar to the correctness proof of the translation EQT in [11, Theorem 5.8], and it will be presented in detail in an extended version of this paper.

When verifying modular equivalence of SMODELS program modules sharing a submodule, e.g., modules of the forms $\mathbb{P} \sqcup \mathbb{C}$ and $\mathbb{Q} \sqcup \mathbb{C}$, it is possible to streamline further the translations involved in the verification task.

Theorem 3. *Let $\mathbb{P}$ and $\mathbb{Q}$ be compatible* SMODELS *program modules with the EVA property, and $\mathbb{C}$ an* SMODELS *program module such that $\mathbb{P} \sqcup \mathbb{C}$ and $\mathbb{Q} \sqcup \mathbb{C}$ are defined. Then $\mathbb{P} \sqcup \mathbb{C} \equiv_\mathrm{m} \mathbb{Q} \sqcup \mathbb{C}$ iff $\mathrm{SM}(\mathrm{EQT}(\mathbb{P}, \mathbb{Q}) \sqcup \mathbb{C}) = \mathrm{SM}(\mathrm{EQT}(\mathbb{Q}, \mathbb{P}) \sqcup \mathbb{C}) = \emptyset$.*

Notice that the context $\mathbb{C}$ can be an arbitrary SMODELS program module, i.e., it is not necessary for $\mathbb{C}$ to have the EVA property, as long as $\mathbb{P} \sqcup \mathbb{C}$ and $\mathbb{Q} \sqcup \mathbb{C}$ are defined. To prove Theorem 3 notice that due to the structure of the translation $\mathrm{EQT}(\mathbb{P}, \mathbb{Q}) \sqcup \mathbb{C}$ is defined whenever $\mathbb{P} \sqcup \mathbb{C}$ is defined, and apply Theorems 1 and 2.

If a module $\mathbb{Q}$ is obtained from a module $\mathbb{P}$ through local modifications, it is likely that their components are pairwise compatible, and there is a partitioning for $\mathbb{P}$ and $\mathbb{Q}$ such that $\mathbb{P} = \mathbb{P}_1 \sqcup \cdots \sqcup \mathbb{P}_n$ and $\mathbb{Q} = \mathbb{Q}_1 \sqcup \cdots \sqcup \mathbb{Q}_n$ where $\mathbb{P}_i$ is compatible with $\mathbb{Q}_i$ for all i. Notice that $\mathbb{P}_i = \mathbb{Q}_i$ might even hold for a number of i's. Also, several possible compatible module structures can be obtained for $\mathbb{P}$ and $\mathbb{Q}$, for example, by regrouping or taking compositions of submodules. Verifying $\mathbb{P}_i \equiv_\mathrm{m} \mathbb{Q}_i$ for every i is not of interest as such, since $\mathbb{P}_i \not\equiv_\mathrm{m} \mathbb{Q}_i$ does not necessarily imply $\mathbb{P} \not\equiv_\mathrm{m} \mathbb{Q}$. Of course, if $\mathbb{P}_i \equiv_\mathrm{m} \mathbb{Q}_i$ holds and their equivalence can be verified efficiently, then Corollary 1 implies that $\mathbb{P}_i$ and $\mathbb{Q}_i$ are modularly equivalent in every possible context. However, if this is not the case, it is still possible to organize the verification of $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$ as a sequence of n module-level tests as follows:

$$(\bigsqcup_{j=1}^{i-1} \mathbb{Q}_j) \sqcup \mathbb{P}_i \sqcup (\bigsqcup_{j=i+i}^{n} \mathbb{P}_j) \equiv_\mathrm{m} (\bigsqcup_{j=1}^{i-1} \mathbb{Q}_j) \sqcup \mathbb{Q}_i \sqcup (\bigsqcup_{j=i+1}^{n} \mathbb{P}_j) \tag{1}$$

where $1 \leq i \leq n$. The resulting chain of equivalences conveys $\mathbb{P} \equiv_\mathrm{m} \mathbb{Q}$. In each test (1) modules differ in $\mathbb{P}_i$ and $\mathbb{Q}_i$ for which the other modules form a common context

$$\mathbb{C}_i = (\bigsqcup_{j=1}^{i-1} \mathbb{Q}_j) \sqcup (\bigsqcup_{j=i+i}^{n} \mathbb{P}_j).$$

Theorem 3 gives us the means to modularize the task of equivalence verification together with (1). We expect computational advantage from the strategy described above, especially when the context $\mathbb{C}_i$ is clearly larger than the modules $\mathbb{P}_i$ and $\mathbb{Q}_i$, and focusing EQT solely to $\mathbb{P}_i$ and $\mathbb{Q}_i$ reduces the length of programs involved in verification steps. Note that if there are n submodules, there are $n!$ possible different orders in which to verify the chain of equivalences. It seems likely that the order has an effect on the efficiency of the approach, e.g., if the test for (1) fails for only one value of i.

4 Experiments

The translation function EQT for verifying modular equivalence presented in Section 3 has been incorporated into a translator called LPEQ (v. 1.18)[4]. The translation for verifying modular equivalence is obtained using option flag `-m`. Furthermore, to make the translator compatible with current solvers, an input generator can be augmented to the translation using flag `-i`. Together with a tool called LPCAT (v. 1.6) used to compose two modules together, the new version of LPEQ allows us to examine the feasibility of verification of modular equivalence, compared to the approach where programs are seen as integral entities. It is worth noticing that even though LPEQ and LPCAT give us the means to evaluate the modular approach to equivalence verification, current versions of LPARSE and SMODELS are not yet fully compatible with our module architecture, which brings us certain difficulties when simulating modular answer set programming. For examples of use, we refer to the LPEQ homepage.

In our experiments we use SMODELS (v. 2.32) for the computation of stable models with flag `-nolookahead` (for our benchmarks this option gives the best running times) for programs instantiated with LPARSE (v. 1.0.17). The total running time for the equivalence verification in one direction is the time needed by LPEQ to produce the translation EQT plus the running time needed by SMODELS for trying to compute *one* stable model for the translation. However, the translation time is negligible. We consider *modularly equivalent* SMODELS program modules. Therefore one always has to count running times in both directions. Since the running times of SMODELS depend on the order of rules in programs and literals in rules, we shuffle them randomly. All the tests reported were run under the Linux 2.6.8 operating system on a 1.7GHz AMD Athlon XP 2000+ CPU with 1 GB of main memory. As regards timing, we report the sum of user and system times as measured by `/usr/bin/time` command in UNIX.

First, we consider two modular encodings solving the n-queens problem, i.e., how to place n queens on an $n \times n$ chess board so that they do not threaten each other. The programs Q_1^n and Q_2^n are composed of two modules each, i.e., $Q_1^n = \mathbb{G}_{\mathrm{x}}^n \sqcup \mathbb{C}_1^n$, $Q_2^n = \mathbb{G}_{\mathrm{x}}^n \sqcup \mathbb{C}_2^n$. Basically $\mathbb{G}_{\mathrm{x}}^n$ generates a placement of n queens row-by-row. The generator module $\mathbb{G}_{\mathrm{x}}^n$ has an empty input signature, and its output signature consists of ground instances of the predicate `q(X,Y)` denoting the placement of a queen in square `(X,Y)`. Modules $\mathbb{C}_1^n$ and $\mathbb{C}_2^n$ are used to check that the placement induced by the generator module is legal and $\mathbb{C}_2^n$ is an optimized version of $\mathbb{C}_1^n$ obtained by reducing symmetric rules. The checking modules expect a placement of queens, expressed using ground instances of `q(X,Y)`, as their input and have an empty output signature. We compare the times needed to verify (a) $\mathbb{C}_1^n \equiv_{\mathrm{m}} \mathbb{C}_2^n$, (b) $\mathbb{C}_1^n \sqcup \mathbb{G}_{\mathrm{x}}^n \equiv_{\mathrm{m}} \mathbb{C}_2^n \sqcup \mathbb{G}_{\mathrm{x}}^n$, and (c) $Q_1^n \equiv_{\mathrm{v}} Q_2^n$. Notice that in (b) and (c) exactly the same equivalence is verified, the difference being that in (c) the knowledge about modular structure is not taken into account and the translation-based method is applied to complete programs. In (b) we use the approach described in Theorem 3 with $\mathbb{G}_{\mathrm{x}}^n$ as the common context for $\mathbb{C}_1^n$ and $\mathbb{C}_2^n$. In (a) we verify the equivalence of $\mathbb{C}_1^n$ and $\mathbb{C}_2^n$ in every possible context. This is a stronger result than what is obtained in the cases (b) and (c). By Corollary 1 it is clear that (a) implies (b) and (c), but not vice versa.

[4] Available at `http://www.tcs.hut.fi/Software/lpeq/`.

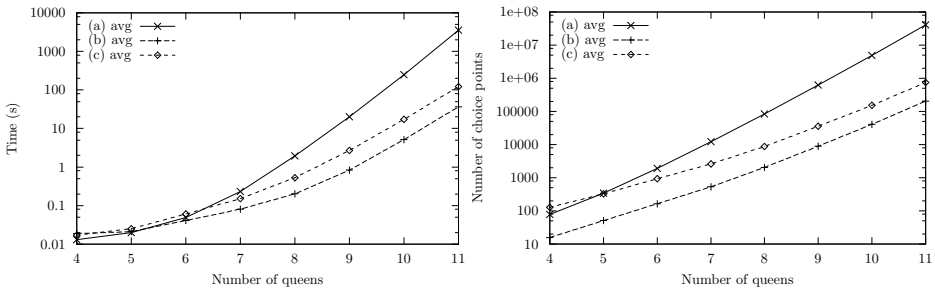

Fig. 2. The averages for running times (left) and numbers of choice points (right) for verifying (a) $\mathbb{C}_1^n \equiv_{\mathrm{m}} \mathbb{C}_2^n$, (b) $\mathbb{C}_1^n \sqcup \mathbb{G}_{\mathrm{x}}^n \equiv_{\mathrm{m}} \mathbb{C}_2^n \sqcup \mathbb{G}_{\mathrm{x}}^n$, and (c) $Q_1^n \equiv_{\mathrm{v}} Q_2^n$ in the n-queens experiment.

We vary the number of queens n from 4 to 11 and repeat the verification task 10 times for each number of queens generating each time new randomly shuffled versions of the modules involved. The average running times and the average numbers of choice points, i.e., the number of choices made by SMODELS during the search, for each approach are presented in Figure 2. We see that taking into account the common context improves the efficiency of the translation-based method, as regards both time and the number of choice points. Checking modular equivalence without a specific context can be time consuming if the number of possible inputs is high, as is the case with the queens encodings (there are n^2 squares in the chess board and as a queen can either be placed or not in each of them, there are 2^{n^2} possible inputs for modules $\mathbb{C}_1^n$ and $\mathbb{C}_2^n$). However, it should be kept in mind that once $\mathbb{C}_1^n \equiv_{\mathrm{m}} \mathbb{C}_2^n$ has been verified, one knows that $\mathbb{C}_1^n$ and $\mathbb{C}_2^n$ are modularly equivalent in any possible context.

We use the problem of finding a Hamiltonian cycle for directed graphs of n vertices as our second benchmark problem. Again we consider modularly equivalent encodings. The first encoding consists of three modules $\mathbb{G}_1^n \sqcup \mathbb{H}_1^n \sqcup \mathbb{R}^n$, and is similar to the Hamiltonian cycle encoding used by Simons et al. [22]. We, however, consider directed graphs instead of undirected ones. Module $\mathbb{G}_1^n$ is used to generate all possible directed graphs of n vertices represented as a set of edges. Given a set of edges for n vertices as an input, module $\mathbb{H}_1^n$ selects the edges to be taken into a cycle by insisting that each vertex is incident to exactly two edges in the cycle. Finally, given a cycle candidate as an input, module $\mathbb{R}^n$ checks that each vertex is reachable from the starting vertex along the edges in the cycle. We also use an optimized variant of module $\mathbb{H}_1^n$. Module $\mathbb{H}_2^n$ takes into account that the input graph is directed and each vertex must then have exactly one incoming and exactly one outgoing edge in the cycle. The second encoding $\mathbb{G}_1^n \sqcup \mathbb{HR}^n$ is based on the alternative encoding presented in [22]. In this encoding we cannot separate the selection of the edges to be taken into the cycle and the checking of reached vertices into two modules as their definitions are mutually dependent. Thus module $\mathbb{HR}^n$ solves the Hamiltonian cycle problem given a graph of n vertices as input.

In addition to the module $\mathbb{G}_1^n$ generating all directed graphs, we consider also other graph generator modules. Each module $\mathbb{G}_i^n$ for $i = 1, \ldots, 5$ generates a family of directed graphs with n vertices with the following properties: all (directed) graphs ($i = 1$), irreflexive graphs ($i = 2$), symmetric and irreflexive graphs ($i = 3$), asymmetric graphs

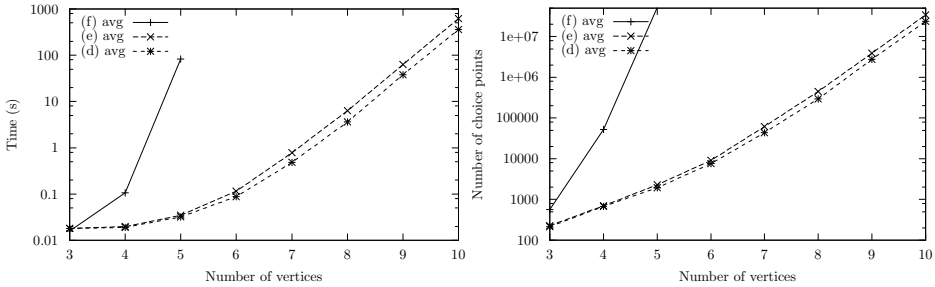

Fig. 3. The average running times (left) and the average numbers of choice points (right) for verifying (d) $\mathbb{H}_1^n \sqcup (\mathbb{R}^n \sqcup \mathbb{G}_1^n) \equiv_{\mathrm{m}} \mathbb{H}_2^n \sqcup (\mathbb{R}^n \sqcup \mathbb{G}_1^n)$, (e) $(\mathbb{H}_1^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} (\mathbb{H}_2^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n$, and (f) $\mathbb{H}_1^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} \mathbb{H}_2^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n$ in the Hamiltonian cycle experiment.

($i = 4$), and graphs with Euclidean edge relation ($i = 5$). We vary the number of vertices n from 3 to 10 and repeat the verification task 10 times for each number of vertices generating each time new randomly shuffled versions of the modules involved.

We start by comparing the times needed to verify

(d) $\mathbb{H}_1^n \sqcup (\mathbb{R}^n \sqcup \mathbb{G}_1^n) \equiv_{\mathrm{m}} \mathbb{H}_2^n \sqcup (\mathbb{R}^n \sqcup \mathbb{G}_1^n)$,
(e) $(\mathbb{H}_1^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} (\mathbb{H}_2^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n$, and
(f) $\mathbb{H}_1^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} \mathbb{H}_2^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n$.

In this experiment we want to study the impact of varying the size of the common context on the time complexity of verifying $\mathbb{H}_1^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} \mathbb{H}_2^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n$. Thus the equivalence verified in each item is the same and the difference is that the common context is varied from $\mathbb{R}^n \sqcup \mathbb{G}_1^n$ in (d) to empty (module) in (f). The average running times and the average numbers of choice points for the approaches are presented in Figure 3. We see that the approach in (f) becomes infeasible for graphs with only six vertices (a timeout of 6000 seconds was used). The difference in running times for equivalences in (d) and (e) is smaller, but best results are achieved when the maximal common context is used in (d). The average number of choices made by SMODELS behave similarly to the average running times. For comparison, the time needed to find all stable models for $\mathbb{H}_2^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n$ is approximately 1.5 seconds for $n = 4$ and 2000 seconds for $n = 5$. This shows the effectiveness of the modular translation-based method as a naive approach of cross-checking stable models [10] is likely to be infeasible even for $n = 5$, because the number of stable models becomes very high.

Next we compare the times needed to verify

(g) equivalence in (d) and $(\mathbb{H}_2^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} \mathbb{HR}^n \sqcup \mathbb{G}_1^n$, and
(h) $(\mathbb{H}_1^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} \mathbb{HR}^n \sqcup \mathbb{G}_1^n$.

Notice that the equivalence verified in (g) is the same as verified in (h). The motivation is to see whether it is more efficient to verify the equivalence of $\mathbb{H}_1^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n$ and $\mathbb{HR}^n \sqcup \mathbb{G}_1^n$ directly or having $\mathbb{H}_2^n \sqcup \mathbb{R}^n \sqcup \mathbb{G}_1^n$ as an intermediate step. Furthermore, we want to see the effect of hiding predicate `reached(X)` in the encodings on the efficiency. Recall that this increases the size of the translation. Thus, we verify the

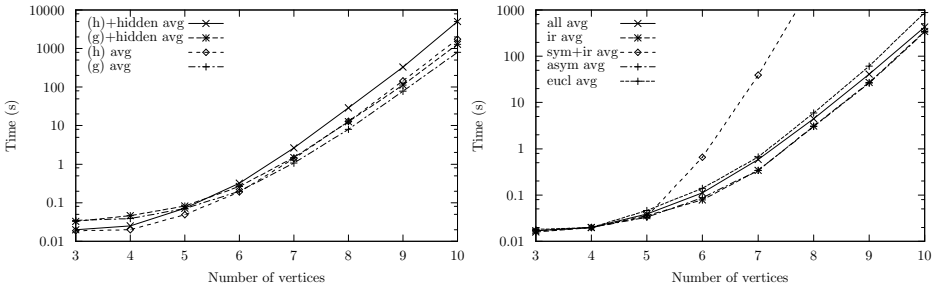

Fig. 4. The average running times for verifying (g) equivalence in (d) plus $(\mathbb{H}_2^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} \mathbb{HR}^n \sqcup \mathbb{G}_1^n$, and (h) $(\mathbb{H}_1^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_1^n \equiv_{\mathrm{m}} \mathbb{HR}^n \sqcup \mathbb{G}_1^n$ with predicate `reached(X)` hidden/visible (left); and the average running times for verifying $(\mathbb{H}_2^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_i^n \equiv_{\mathrm{m}} \mathbb{HR}^n \sqcup \mathbb{G}_i^n$ for different graph families $\mathbb{G}_i^n$, $i = 1, \ldots, 5$ (right).

equivalences (g) and (h) also with predicate `reached(X)` hidden and compare the time needed when predicate `reached(X)` is visible.

The average running times for this experiment are presented in Figure 4 (left). Hiding predicate `reached(X)` in approach (g) increases the average running time by approximately one third. The effect of hiding is more significant in approach (h) in which the average running time is doubled when predicate `reached(X)` is hidden. In practice it seems to be a good idea to hide as few predicates as necessary. The difference in the average numbers of choice points is marginal. The average running times of approach (g) are less than those of approach (h) regardless the visibility of predicate `reached(X)`, and thus it seems to be a good idea to do the optimization step first.

Finally, we want to see how the choice of the graph family, i.e., the choice of the graph generator module $\mathbb{G}_i^n$ for $i = 1, 2, \ldots, 5$ affects the time needed to verify

$$(\mathbb{H}_2^n \sqcup \mathbb{R}^n) \sqcup \mathbb{G}_i^n \equiv_{\mathrm{m}} \mathbb{HR}^n \sqcup \mathbb{G}_i^n.$$

As $\mathbb{G}_1^n$ generates all directed graphs, verifying the equivalence for $i = 1$ actually verifies $\mathbb{H}_2^n \sqcup \mathbb{R}^n \equiv_{\mathrm{m}} \mathbb{HR}^n$ and modular equivalence for the other generator modules follows from Corollary 1. The motivation, however, is to see whether it is faster to verify a weaker result, i.e., to verify the equivalence for certain subclasses of directed graphs.

The average running times for this experiment are presented in Figure 4 (right). Using symmetric and irreflexive graphs as context turned out to be especially time consuming. We used a timeout of 6000 seconds in this experiment and were not able to verify the equivalence for the values $n = 9$ and $n = 10$ with this limit. For the sake of clarity, we also drop the average time for $n = 8$ (4048 seconds) from Figure 4 (right). The average numbers of choice points behave similarly to the average running times. The somewhat surprising implication of this experiment is that restricting the number of possible inputs by applying a graph generator having less stable models than $\mathbb{G}_1^n$ does not necessarily make the equivalence verification task more efficient as opposed to our n-queens experiment. The reason for this might be that it can be more difficult to find stable models for a more specific generator module.

5 Conclusions

In this paper, we continue our work with modular program development within answer set programming [21, 12, 19]. In particular, we are interested in the problem of verifying the *modular equivalence* of SMODELS program modules. We show how an existing translation-based approach to verifying visible equivalence [11] can be adjusted to the task of verifying modular equivalence. The current translation $\mathrm{EQT}(\mathbb{P}, \mathbb{Q})$ and its implementation LPEQ cover the types of rules supported by the SMODELS search engine which provide the basic knowledge representation primitives. We also show that in a case where modules in question share a submodule, e.g., $\mathbb{P} = \mathbb{P}' \sqcup \mathbb{C}$ and $\mathbb{Q} = \mathbb{Q}' \sqcup \mathbb{C}$, the method can be further streamlined and it is not necessary to translate the common context $\mathbb{C}$. If the context $\mathbb{C}$ is large, then the size of the translation involved in the equivalence verification task is potentially reduced notably.

We evaluate experimentally the efficiency of the translation-based method in the verification of modular equivalence. The results indicate that it depends on the specific problem domain and encodings used whether checking modular equivalence without a specific context is more time consuming than with a context module. However, if two modules share a submodule, taking the shared context into account has the potential of speeding up the translation-based method significantly. Furthermore, in practice it seems to be a good idea to hide as few atoms as necessary when using the translation-based method. Based on the experimental evaluation, modularization of the verification of modular equivalence seems to be a good idea in many cases, especially if the common context shared by the modules is large and the number of submodules stays reasonable.

Even though LPEQ and LPCAT give us the means to evaluate the modular approach to equivalence verification, current ASP solvers do not directly utilize the module architecture described in this paper. This is an issue to taken into account in the design of future solvers. Another direction for our research is to extend the translation-based verification method for disjunctive programs [20] to the case of modular equivalence [12].

References

1. C. Baral, J. Dzifcak, and H. Takahashi. Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming. In *Proc. of the 22nd International Conference on Logic Programming* (*ICLP 2006*), volume 4079 of *LNCS*, pages 376–390. Springer, 2006.
2. M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.
3. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.
4. T. Eiter, G. Gottlob, and H. Veith. Modular logic programming and generalized quantifiers. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNCS*, pages 290–309. Springer, 1997.
5. W. Faber, G. Greco, and N. Leone. Magic sets and their application to data integration. In *Proc. of the 10th International Conference on Database Theory*, volume 3363 of *LNCS*, pages 306–320, Edinburgh, UK, January 2005. Springer.
6. H. Gaifman and E.Y. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 134–142, Austin, Texas, USA, January 1989. ACM Press.

7. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
8. G. Ianni, G. Ielpa, A. Pietramala, M.C. Santoro, and F. Calimeri. Enhancing answer set programming with templates. In *10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings*, pages 233–239, 2004.
9. T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.
10. T. Janhunen and E. Oikarinen. 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*, pages 493–504, Cosenza, Italy, September 2002. Springer.
11. T. Janhunen and E. Oikarinen. Automated verification of weak equivalence within the SMODELS system. *Theory and Practice of Logic Programming*, to appear.
12. T. Janhunen, E. Oikarinen, H. Tompits, and S. Woltran. Modularity aspects of disjunctive stable models. In *Proc. 9th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2007*, volume 4483 of *LNAI*, pages 175–187, Tempe, Arizona, USA, May 2007. Springer.
13. V. Lifschitz. Computing Circumscription. In *Proc. of the 9th International Joint Conference on Artificial Intelligence* (*IJCAI'85*), pages 121–127, Los Angeles, California, USA, August 1985. Morgan Kaufmann.
14. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
15. V. Lifschitz and H. Turner. Splitting a logic program. In *Proc. of the 11th International Conference on Logic Programming*, pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.
16. V.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, 1999.
17. D. Miller. A theory of modules for logic programming. In *Proc. of 1986 Symposium on Logic Programming*, pages 106–114, Salt Lake City, USA, September 1986. IEEE Computer Society Press.
18. I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
19. E. Oikarinen. Modularity in smodels programs. In *Proc. 9th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2007*, volume 4483 of *LNAI*, pages 321–326, Tempe, Arizona, USA, May 2007. Springer-Verlag.
20. E. Oikarinen and T. Janhunen. 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*, pages 180–193, Fort Lauderdale, USA, January 2004. Springer.
21. E. Oikarinen and T. Janhunen. Modular equivalence for normal logic programs. In *Proc. of the 17th European Conference on Artificial Intelligence*, pages 412–416, Riva del Garda, Italy, August 2006. IOS Press.
22. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
23. L. Tari, C. Baral, and S. Anwar. A language for modular answer set programming: Application to ACC tournament scheduling. In *Proc. of the 3rd International Workshop on Answer Set Programming*, volume 142 of *CEUR Workshop Proceedings*, Bath, UK, September 2005.