TLP **8** (5 & 6): 717–761, 2008. © 2008 Cambridge University Press
doi:10.1017/S147106840800358X Printed in the United Kingdom

Achieving compositionality of the stable model semantics for SMODELS *programs*[1]

EMILIA OIKARINEN and TOMI JANHUNEN

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

submitted 29 February 2008; revised 18 September 2008; accepted 25 September 2008

Abstract

In this paper, a Gaifman–Shapiro-style module architecture is tailored to the case of SMODELS programs under the stable model semantics. The composition of SMODELS program modules is suitably limited by module conditions which ensure the compatibility of the module system with stable models. Hence the semantics of an entire SMODELS program depends directly on stable models assigned to its modules. This result is formalized as a *module theorem* which truly strengthens V. Lifschitz and H. Turner's splitting-set theorem (June 1994, Splitting a logic program. In *Logic Programming: Proceedings of the Eleventh International Conference on Logic Programming*, Santa Margherita Ligure, Italy, P. V. Hentenryck, Ed. MIT Press, 23–37) for the class of SMODELS programs. To streamline generalizations in the future, the module theorem is first proved for normal programs and then extended to cover SMODELS programs using a translation from the latter class of programs to the former class. Moreover, the respective notion of module-level equivalence, namely *modular equivalence*, is shown to be a proper congruence relation: it is preserved under substitutions of modules that are modularly equivalent. Principles for program decomposition are also addressed. The strongly connected components of the respective dependency graph can be exploited in order to extract a module structure when there is no explicit *a priori* knowledge about the modules of a program. The paper includes a practical demonstration of tools that have been developed for automated (de)composition of SMODELS programs.

KEYWORDS: answer set programming, module system, compositional semantics, stable model semantics, modular equivalence

1 Introduction

Answer set programming (ASP) (Marek and Truszczyński 1999; Niemelä 1999; Gelfond and Leone 2002; Baral 2003) is an approach to declarative rule-based constraint programming that has been successively used in many knowledge representation and

[1] This is an extended version of two conference papers, Oikarinen and Janhunen (2006) and Oikarinen (2007) presented at ECAI'06 and LPNMR'07, respectively.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

reasoning tasks (Nogueira *et al.* 2001; Soininen *et al.* 2001; Erdem *et al.* 2006; Brooks *et al.* 2007) In ASP, the problem in hand is solved declaratively

(1) by writing down a logic program the answer sets of which correspond to the solutions of the problem and
(2) by computing the answer sets of the program using a special purpose search engine that has been designed for this task.

A modeling philosophy of this kind suggests to treat programs as integral entities. The answer set semantics—originally defined for entire programs only (Gelfond and Lifschitz 1988, 1990)—also reflects this fact. Such indivisibility of programs is creating an increasing problem as program instances tend to grow along the demands of new application areas of ASP. It is to be expected that prospective application areas such as semantic web, bioinformatics, and logical cryptanalysis will provide us with huge program instances to create, to solve, and to maintain.

Modern programming languages provide means to exploit *modularity* in a number of ways to govern the complexity of programs and their development process. Indeed, the use of *program modules* or *objects* of some kind can be viewed as an embodiment of the classical *divide-and-conquer* principle in the art of programming. The benefits of modular program development are numerous. A software system is much easier to design as a set of interacting components rather than a monolithic system with unclear internal structure. A modular design lends itself better for implementation as programming tasks are then easier to delegate among the team of programmers. It also enables the re-use of code organized as module libraries, for instance. To achieve similar advantages in ASP, one of the central goals of our research is to foster modularity in the context of ASP.

Although, modularity has been studied extensively in the context of conventional logic programs (see Bugliesi *et al.* 1994 for a survey) relatively little attention has been paid to modularity in ASP. Many of the approaches proposed so far are based on very strict syntactic conditions on the module hierarchy, for instance, by enforcing stratification of some kind, or by prohibiting recursion altogether (Eiter *et al.* 1997a, 1997b; Tari *et al.* 2005) On the other hand, approaches based on *splitting sets* (Lifschitz and Turner 1994; Eiter *et al.* 1997b; Faber *et al.* 2005) are satisfactory from the point of view of *compositional semantics*: the answer sets of an entire program are obtained as specific combinations of the answer sets of its components. A limitation of splitting sets is that they divide logic programs in two parts, the *top* and the *bottom*, which is rather restrictive and conceptually very close to stratification. On the other hand, the compositionality of answer set semantics is neglected altogether in syntactic approaches (Ianni *et al.* 2004; Tari *et al.* 2005) and this aspect of models remains completely at the programmer's responsibility.

To address the deficiencies described, we accommodate a module architecture proposed by Gaifman and Shapiro (1989) to ASP and, in particular, in the context of the SMODELS system (Simons *et al.* 2002).[2] There are two main criteria for the design.

[2] Also other systems such as CLASP (Gebser *et al.* 2007) and CMODELS (Lierler 2005) that are compatible with the internal file format of the SMODELS system are implicitly covered.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

First of all, it is essential to establish the full compositionality of answer set semantics with respect to the module system. This is to ensure that various reasoning tasks—such as the verification of program equivalence (Janhunen and Oikarinen 2007)—can be modularized. Second, for the sake of flexibility of knowledge representation, any restrictions on the module hierarchy should be avoided as far as possible. We pursue these goals according to the following plan.

In Section 2, we take a closer look at modularity in the context of logic programming. In order to enable comparisons later on, we also describe related approaches in the area of ASP in more detail. The technical preliminaries of the paper begin with a recapitulation of *stable model semantics* (Gelfond and Lifschitz 1988) in Section 3. However, stable models, or answer sets, are reformulated for a class of programs that corresponds to the input language of the SMODELS solver (Simons *et al.* 2002). The definition of *splitting sets* is included to enable a detailed comparison with our results. Moreover, we introduce the concepts of *program completion* and *loop formulas* to be exploited in proofs later on, and review some notions of equivalence that have been proposed in the literature.

In Section 4, we present a *module architecture* for SMODELS programs, in which the interaction between modules takes place through a clearly defined *input/output interface*. The design superficially resembles that of Gaifman and Shapiro (1989), but in order to achieve the full compositionality of stable models, further conditions on program composition are incorporated. This is formalized as the main result of the paper, namely the *module theorem*, which goes beyond the splitting-set theorem (Lifschitz and Turner 1994) as *negative recursion* is tolerated by our definitions. The proof is first presented for normal programs and then extended for SMODELS programs using a translation-based scheme. The scheme is based on three distinguished properties of translations, *strong faithfulness*, *preservation of compositions*, and *modularity*, that are sufficient to lift the module theorem. In this way, we get prepared for even further syntactic extensions of the module theorem in the future. The respective notion of module-level equivalence, that is, *modular equivalence*, is proved to be a proper *congruence* for program composition. In other words, substitutions of modularly equivalent modules preserve modular equivalence. This way modular equivalence can be viewed as a reasonable compromise between *uniform equivalence* (Eiter and Fink 2003) which is not a congruence for program union, and *strong equivalence* (Lifschitz *et al.* 2001) which is a congruence for program union but allows only rather straightforward semantics-preserving transformations of (sets of) rules.

In Section 5, we address principles for the decomposition of SMODELS programs. It turns out that strongly connected components of dependency graphs can be exploited in order to extract a module structure when there is no explicit *a priori* knowledge about the modules of a program. In addition, we consider the possibility of relaxing our restrictions on program composition using the content of the module theorem as a criterion. The result is that the notion of modular equivalence remains unchanged but the computational cost of checking legal compositions of modules becomes essentially higher. In Section 6, we demonstrate how the module system can be exploited in practice in the context of the SMODELS system. We present tools

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

that have been developed for (de)composition of logic programs and conduct a practical experiment which illustrates the performance of the tools when processing very large benchmark instances, that is, SMODELS programs having up to millions of rules. The concluding remarks of this paper are presented in Section 7.

2 Modularity aspects of logic programming

Bugliesi *et al.* (1994) address several properties that are expected from a modular logic programming language. For instance, a modular language should

- allow *abstraction, parameterization*, and *information hiding*,
- *ease program development* and *maintenance* of large programs,
- allow *re-usability*,
- have a *nontrivial notion of program equivalence* to justify replacement of program components, and
- maintain the *declarativity* of logic programming.

Two mainstream programming disciplines are identified. In *programming-in-the-large* approaches programs are composed with algebraic operators (see, for instance, O'Keefe 1985; Mancarella and Pedreschi 1988; Gaifman and Shapiro 1989; Brogi *et al.* 1994). In *programming-in-the-small* approaches abstraction mechanisms are used (see, for instance, Miller 1989; Giordano and Martelli 1994).

The *programming-in-the-large* approaches have their roots in the framework proposed by O'Keefe (1985) where logic programs are seen as *elements of an algebra* and the operators for *composing programs* are seen as *operators in that algebra*. The fundamental idea is that a logic program should be understood as a part of a system of programs. Program composition is a powerful tool for structuring programs without any need to extend the underlying language of Horn clauses. Several algebraic operations such as *union*, *deletion*, *overriding union,* and *closure* have been considered. This approach *supports* naturally the *re-use* of the pieces of programs in different composite programs, and when combined with an adequate equivalence relation also the replacement of equivalent components. This approach is highly flexible, as new composition mechanisms can be obtained by introducing a corresponding operator in the algebra or combining existing ones. Encapsulation and information hiding can be obtained by introducing suitable *interfaces* for components.

The *programming-in-the-small* approaches originate from Miller (1989). In this approach the composition of modules is modeled in terms of logical connectives of a language that is defined as an *extension of Horn clause logic*. The approach by Giordano and Martelli (1994) employs the same structural properties, but suggests a more refined way of modeling visibility rules than the one by Miller (1989).

It is essential that a semantical characterization of a modular language is such that the meaning of composite programs can be defined in terms of the meaning of its components (Maher 1993). To be able to identify when it is safe to substitute a module with another without affecting the global behavior it is crucial to have a notion of *semantical equivalence*. More formally these desired properties can be

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

described under the terms of *compositionality* and *full abstraction* (Gaifman and Shapiro 1989; Meyer 1988).

Two programs are *observationally congruent* if and only if they exhibit the same observational behavior in every *context* they can be placed in. A semantics is compositional if semantical equality implies observational congruence, and fully abstract if semantical equivalence coincides with observational congruence. The compositionality and full abstraction properties for different notions of semantical equivalence (*subsumption equivalence*, *logical equivalence*, and *minimal Herbrand model equivalence*) and different operators in an algebra (union, closure, and overriding union) are considered by Bugliesi *et al.* (1994). It is worth noting that minimal Herbrand model equivalence coincides with the *weak equivalence* relation for positive logic programs. As to be defined in Section 3.3, two logic programs are weakly equivalent if and only if they have exactly the same answer sets. As the equivalence based on minimal Herbrand model semantics is not compositional with respect to program union (Bugliesi *et al.* 1994), we note that it is not a suitable composition operator for our purposes unless further constraints are introduced.

2.1 Modularity in answer set programming

There are a number of approaches within ASP involving modularity in some sense, but only a few of them really describe a flexible module architecture with a clearly defined interface for module interaction.

Eiter *et al.* (1997b) address modularity in ASP in the programming-in-the-small sense. They view program modules as *generalized quantifiers* as introduced in (Mostowski 1957). The definitions of quantifiers are allowed to nest, that is, program P can refer to another module Q by using it as a generalized quantifier. The main program is clearly distinguished from subprograms, and it is possible to nest calls to submodules if the so-called *call graph* is *hierarchical*, that is, *acyclic*. Nesting, however, increases the computational complexity depending on the depth of nesting.

Ianni *et al.* (2004) propose another programming-in-the-small approach to ASP based on *templates*. The semantics of programs containing template atoms is determined by an *explosion algorithm*, which basically replaces the template with a standard logic program. However, the explosion algorithm is not guaranteed to terminate if template definitions are used recursively.

Tari *et al.* (2005) extend the language of normal logic programs (NLPs) by introducing the concept of *import rules* for their ASP program modules. There are three types of import rules which are used to import a set of tuples $\overline{X}$ for a predicate q from another module. An *ASP module* is defined as a quadruple of a module name, a set of parameters, a collection of normal rules, and a collection of import rules. Semantics is only defined for modular programs with acyclic dependency graph, and answer sets of a module are defined with respect to the modular ASP program containing it. Also, it is required that import rules referring to the same module always have the same form.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Programming-in-the-large approaches in ASP are mostly based on Lifschitz and Turner's splitting-set theorem (Lifschitz and Turner 1994) or are variants of it. The class of logic programs considered by Lifschitz and Turner (1994) is that of *extended disjunctive logic programs*, that is, disjunctive logic programs with two kinds of negation. A *component structure* induced by a *splitting sequence*, that is, iterated splittings of a program, allows a bottom-up computation of answer sets. The restriction implied by this construction is that the dependency graph of the component chain needs to be acyclic.

Eiter *et al.* (1997a) consider *disjunctive logic programs as a query language for relational databases*. A query program π is instantiated with respect to an input database D confined by an input schema $\mathbf{R}$. The semantics of π determines, for example, the answer sets of $\pi[D]$ which are projected with respect to an output schema $\mathbf{S}$. 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*, that is, effectively NLPs. This involves the concept of an *independent set* S of a program P which is a specialization of a splitting set. Due to a close relationship with splitting sets, the flexibility of independent sets for parceling programs is limited in the same way.

The approach based on *lp-functions* (Gelfond and Gabaldon 1999; Baral 2003) is another programming-in-the-large approach. An lp-function has an interface based on input and output signatures. Several operations, for instance, *incremental extension*, *interpolation*, *input opening*, and *input extension* are introduced for composing and refining lp-functions. The composition of lp-functions, however, only allows incremental extension, and thus similarly to the splitting-set theorem there can be no recursion between lp-functions.

3 Preliminaries: SMODELS programs

To keep the presentation of our module architecture compatible with an actual implementation, we cover the input language of the SMODELS system—excluding *optimization statements*. In this section, we introduce the syntax and semantics for SMODELS programs, and, in addition, point out a number of useful properties of logic programs under stable model semantics. We end this section with a review of equivalence relations that have been proposed for logic programs.

3.1 Syntax and semantics

Basic constraint rules (Simons *et al.* 2002) are either *weight rules* of the form

$$a \leftarrow w \leqslant \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, \sim c_1 = w_{c_1}, \ldots, \sim c_m = w_{c_m}\}, \tag{1}$$

or *choice rules* of the form

$$\{a_1, \ldots, a_h\} \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \tag{2}$$

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

where a, a_i's, b_j's, and c_k's are atoms, $h > 0$, $n \geqslant 0$, $m \geqslant 0$, and $\sim$ denotes *negation as failure* or *default negation*. In addition, a weight rule (1) involves a weight limit $w \in \mathbb{N}$ and the respective weights $w_{b_j} \in \mathbb{N}$ and $w_{c_k} \in \mathbb{N}$ associated with each *positive literal* b_j and *negative literal* $\sim c_k$. We use a shorthand $\sim A = \{\sim a \mid a \in A\}$ for any set of atoms A. Each basic constraint rule r consists of two parts: a or $\{a_1, \ldots, a_h\}$ is the *head* of the rule, denoted by Head(r), whereas the rest is called its *body*. The set of atoms appearing in a body of a rule can be further divided into the set of *positive body atoms*, defined as $\text{Body}^+(r) = \{b_1, \ldots, b_n\}$, and the set of *negative body atoms*, defined as $\text{Body}^-(r) = \{c_1, \ldots, c_m\}$. We denote by $\text{Body}(r) = \text{Body}^+(r) \cup \text{Body}^-(r)$ the set of atoms appearing in the body of a rule r. Roughly speaking, the body gives the conditions on which the head of the rule must be satisfied. For example, in case of a choice rule (2), this means that any head atom a_i can be inferred to be true if $b_1, \ldots, b_n$ hold true by some other rules but none of the atoms $c_1, \ldots, c_m$. Weight rules of form (1) cover many other kinds of rules of interest as their special cases:

$$a \leftarrow l \leqslant \{b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m\}, \tag{3}$$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \tag{4}$$

$$\leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \tag{5}$$

$$\text{compute } \{b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m\}. \tag{6}$$

Cardinality rules of form (3) are essentially weight rules (1) where $w = l$ and all weights associated with literals equal to 1. A *normal rule* or alternatively a *basic rule* (4) is a special case of a cardinality rule (3) with $l = n + m$. The intuitive meaning of an integrity constraint (5) is that the conditions given in the body are never simultaneously satisfied. The same can be stated in terms of a basic rule $f \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m, \sim f$ where f is a new atom dedicated to integrity constraints. Finally, *compute statements* (6) of the SMODELS system effectively correspond to sets of integrity constraints $\leftarrow \sim b_1, \ldots, \leftarrow \sim b_n$ and $\leftarrow c_1, \ldots, \leftarrow c_m$.

Because the order of literals in (1) and (2) is considered irrelevant, we introduce shorthands $A = \{a_1, \ldots, a_h\}$, $B = \{b_1, \ldots, b_n\}$, and $C = \{c_1, \ldots, c_m\}$ for the sets of atoms involved in rules, and $W_B = \{w_{b_1}, \ldots, w_{b_n}\}$ and $W_C = \{w_{c_1}, \ldots, w_{c_m}\}$ for the respective sets of weights in (1). Using these notations (1) and (2) are abbreviated by $a \leftarrow w \leqslant \{B = W_B, \sim C = W_C\}$[3] and $\{A\} \leftarrow B, \sim C$.

In the SMODELS system, the internal representation of programs is based on rules of the forms (1)–(4) and (6) and one may conclude that basic constraint rules, as introduced earlier, provide a reasonable coverage of SMODELS programs. Thus we concentrate on rules of forms (1) and (2) and view others as syntactic sugar in the sequel.

Definition 3.1

An SMODELS program P is a finite set of basic constraint rules.

[3] Strictly speaking $B = W_B$ and $\sim C = W_C$ are to be understood as sets of pairs of the form (b, w_b) and $(\sim c, w_c)$, respectively. For convenience the exact matching between literals and weights is left implicit in the shorthand.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

An SMODELS program consisting only of basic rules is called a *normal logic program* (NLP), and a basic rule with an empty body is called a *fact*. Given an SMODELS program P, we write $\mathrm{At}(P)$ for its *signature*, that is, the set of atoms occurring in P, and $\mathrm{Body}(P)$ and $\mathrm{Head}(P)$ for the respective subsets of $\mathrm{At}(P)$ having *body occurrences* and *head occurrences* in the rules of P. Furthermore, $\mathrm{Choices}(P) \subseteq \mathrm{Head}(P)$ denotes the set of atoms having a head occurrence in a choice rule of P.

Given a program P, an *interpretation* M of P is a subset of $\mathrm{At}(P)$ defining which atoms $a \in \mathrm{At}(P)$ are *true* ($a \in M$) and which are *false* ($a \notin M$). A weight rule (1) is satisfied in M if and only if $a \in M$ 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. An interpretation $M \subseteq \mathrm{At}(P)$ is a (classical) model of P, denoted by $M \models P$, if and only if M satisfies all the rules in P.

The generalization of the *Gelfond–Lifschitz reduct* (Gelfond and Lifschitz 1988) for SMODELS programs is defined as follows:

Definition 3.2
For an SMODELS program P and an interpretation $M \subseteq \mathrm{At}(P)$, the reduct P^M contains

(1) a rule $a \leftarrow B$ if and only if there is a choice rule $\{A\} \leftarrow B, {\sim}C$ in P such that $a \in A \cap M$, and $M \cap C = \emptyset$;
(2) a rule $a \leftarrow w' \leqslant \{B = W_B\}$ if and only if there is a weight rule $a \leftarrow w \leqslant \{B = W_B, {\sim}C = W_C\}$ in P such that $w' = \max(0, w - \sum_{c \in C \setminus M} w_c)$.

We say that an SMODELS program P is *positive* if each rule in P is a weight rule restricted to the case $C = \emptyset$. Recalling that the basic rules are just a special case of weight rules, we note that the reduct P^M is always positive. An interpretation $M \subseteq \mathrm{At}(P)$ is the *least model* of a positive SMODELS program P, denoted by $\mathrm{LM}(P)$, if and only if $M \models P$ and there is no $M' \models P$ such that $M' \subset M$. Given the *least model semantics* for positive programs (Janhunen and Oikarinen 2007), the stable model semantics (Gelfond and Lifschitz 1988) straightforwardly generalizes for SMODELS programs (Simons *et al.* 2002).

Definition 3.3
An interpretation $M \subseteq \mathrm{At}(P)$ is a stable model of an SMODELS program P if and only if $M = \mathrm{LM}(P^M)$.

Given an SMODELS program P and $a, b \in \mathrm{At}(P)$, we say that *a depends directly* on b, denoted by $b \leqslant_1 a$, if and only if P contains a rule r such that $a \in \mathrm{Head}(r)$ and $b \in \mathrm{Body}^+(r)$. The *positive dependency graph* of P, denoted by $\mathrm{Dep}^+(P)$, is the graph $\langle \mathrm{At}(P), \leqslant_1 \rangle$. The reflexive and transitive closure of $\leqslant_1$ gives rise to the dependency relation $\leqslant$ over $\mathrm{At}(P)$. A *strongly connected component* (SCC) S of $\mathrm{Dep}^+(P)$ is a maximal set $S \subseteq \mathrm{At}(P)$ such that $b \leqslant a$ holds for every $a, b \in S$.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

3.2 Splitting sets and loop formulas

In this section, we only consider the class of *NLPs*. We formulate the *splitting-set theorem* (Lifschitz and Turner 1994) in the case of NLPs[4], and give an alternative definition of stable models based on the *classical models* of the *completion* of a program (Clark 1978) and its *loop formulas* (Lin and Zhao 2004). The splitting-set theorem can be used to simplify the computation of stable models by splitting a program into parts, and it is also a useful tool for structuring mathematical proofs for properties of logic programs.

Definition 3.4
A splitting set for an NLP P is any set $U \subseteq \mathrm{At}(P)$ such that for every rule r in P it holds that $\mathrm{At}(r) \subseteq U$ if $\mathrm{Head}(r) \in U$.

The set of rules $r \in P$ such that $\mathrm{At}(r) \subseteq U$ 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 which can be partially evaluated with respect to an interpretation $X \subseteq U$. The result is a program $\mathrm{e}(\mathrm{t}_U(P), X)$ defined as

$$\{\mathrm{Head}(r) \leftarrow (\mathrm{Body}^+(r) \setminus U), {\sim}(\mathrm{Body}^-(r) \setminus U) \mid r \in \mathrm{t}_U(P), \\ \mathrm{Body}^+(r) \cap U \subseteq X \text{ and } (\mathrm{Body}^-(r) \cap U) \cap X = \emptyset\}.$$

A *solution* to a program with respect to a splitting set is a pair consisting of a stable model X for the bottom and a stable model Y for the top partially evaluated with respect to X.

Definition 3.5
Given a splitting set U for an NLP P, a solution to P with respect to U is a pair $\langle X, Y \rangle$ such that

(i) $X \subseteq U$ is a stable model of $\mathrm{b}_U(P)$ and
(ii) $Y \subseteq \mathrm{At}(P) \setminus U$ is a stable model of $\mathrm{e}(\mathrm{t}_U(P), X)$.

Solutions and stable models relate as follows.

Theorem 3.6 (*The splitting-set theorem (Lifschitz and Turner 1994)*)
Let U be a splitting set for an NLP P and $M \subseteq \mathrm{At}(P)$ an interpretation. Then $M \in \mathrm{SM}(P)$ if and only if the pair $\langle M \cap U, M \setminus U \rangle$ is a solution to P with respect to U.

The splitting-set theorem can also be used in an iterative manner, if there is a *monotone sequence* of splitting sets $\{U_1, \ldots, U_i, \ldots\}$, that is, $U_i \subset U_j$ if $i < j$, for program P. This is called a *splitting sequence* and it induces a *component structure* for P. The splitting-set theorem generalizes to a *splitting sequence theorem* (Lifschitz and Turner 1994), and given a splitting sequence, the stable models of a program P can be computed iteratively bottom-up.

[4] Lifschitz and Turner (1994) consider a more general class of logic programs, *extended disjunctive logic programs*, that is, disjunctive logic programs with two kinds of negation.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Lin and Zhao present an alternative definition of stable models for NLPs based on the *classical models* of the *completion* of a program (Clark 1978) and its *loop formulas* (Lin and Zhao 2004). We will apply this definition later on in the proof of the module theorem (Theorem 4.14).

Definition 3.7 (*Program completion (Clark 1978; Fages 1994)*)
The completion of an NLP P is

$$\mathrm{Comp}(P) = \bigwedge_{a \in \mathrm{At}(P)} \left(a \leftrightarrow \bigvee_{\mathrm{Head}(r)=a} \left(\bigwedge_{b \in \mathrm{Body}^+(r)} b \wedge \bigwedge_{c \in \mathrm{Body}^-(r)} \neg c \right) \right). \tag{7}$$

Note that an empty body reduces to true and in that case the respective equivalence for an atom a is logically equivalent to $a \leftrightarrow \top$.

Definition 3.8
Given an NLP P, a set of atoms $L \subseteq \mathrm{At}(P)$ is a loop of P if for every $a, b \in L$ there is a path of nonzero length from a to b in $\mathrm{Dep}^+(P)$ such that all vertices in the path are in L.

Definition 3.9
Given an NLP P and a loop $L \subseteq \mathrm{At}(P)$ of P, the loop formula associated with L is

$$\mathrm{LF}(L, P) = \neg \left(\bigvee_{r \in \mathrm{EB}(L,P)} \left(\bigwedge_{b \in \mathrm{Body}^+(r)} b \wedge \bigwedge_{c \in \mathrm{Body}^-(r)} \neg c \right) \right) \rightarrow \bigwedge_{a \in L} \neg a$$

where $\mathrm{EB}(L, P) = \{ r \in P \mid \mathrm{Head}(r) \in L \text{ and } \mathrm{Body}^+(r) \cap L = \emptyset \}$ is the set of rules in P which have external bodies of L.

Now, stable models of a program and classical models of its completion that satisfy the loop formulas relate as follows.

Theorem 3.10 (*Lin and Zhao 2004*)
Given an NLP P and an interpretation $M \subseteq \mathrm{At}(P)$, $M \in \mathrm{SM}(P)$ if and only if $M \models \mathrm{Comp}(P) \cup \mathrm{LF}(P)$, where $\mathrm{LF}(P)$ is the set of all loop formulas associated with the loops of P.

3.3 *Equivalence relations for* SMODELS *programs*

There are several notions of equivalence that have been proposed for logic programs. We review a number of them in the context of SMODELS programs.

Lifschitz *et al.* (2001) address the notions of *weak/ordinary equivalence* and *strong equivalence*.

Definition 3.11
SMODELS programs P and Q are weakly equivalent, denoted by $P \equiv Q$, if and only if $\mathrm{SM}(P) = \mathrm{SM}(Q)$; and strongly equivalent, denoted by $P \equiv_s Q$, if and only if $P \cup R \equiv Q \cup R$ for any SMODELS program R.

The program R in this definition can be understood as an arbitrary context in which the two programs being compared could be placed. Therefore, strongly equivalent

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

logic programs are semantics-preserving substitutes of each other and relation $\equiv_s$ is a *congruence relation* for $\cup$ among SMODELS programs, that is, if $P \equiv_s Q$, then also $P \cup R \equiv_s Q \cup R$ for all SMODELS programs R. Using $R = \emptyset$ as context, one sees that $P \equiv_s Q$ implies $P \equiv Q$. The converse does not hold in general.

A way to weaken strong equivalence is to restrict possible contexts to sets of facts. The notion of *uniform equivalence* has its roots in the database community (Sagiv 1987); see (Eiter and Fink 2003) for the case of the stable model semantics).

Definition 3.12
SMODELS programs P and Q are uniformly equivalent, denoted by $P \equiv_u Q$, if and only if $P \cup F \equiv Q \cup F$ for any set of facts F.

Example 3.13 shows that uniform equivalence is not a congruence for union.

Example 3.13 (*Eiter et al. 2004*)
Consider programs $P = \{a.\}$ and $Q = \{a \leftarrow \sim b.\ a \leftarrow b.\}$. It holds that $P \equiv_u Q$, but $P \cup R \not\equiv Q \cup R$ for the context $R = \{b \leftarrow a.\}$. This implies $P \not\equiv_s Q$ and $P \cup R \not\equiv_u Q \cup R$. □

There are also *relativized variants of strong and uniform equivalence* (Woltran 2004) which allow the context to be constrained using a set of atoms A.

For weak equivalence of programs P and Q to hold, $\mathrm{SM}(P)$ and $\mathrm{SM}(Q)$ have to be identical subsets of $\mathbf{2}^{\mathrm{At}(P)}$ and $\mathbf{2}^{\mathrm{At}(Q)}$, respectively. The same effect can be seen with $P \equiv_s Q$ and $P \equiv_u Q$. This makes these relations less useful if $\mathrm{At}(P)$ and $\mathrm{At}(Q)$ differ by some (local) atoms not trivially false in all stable models. The *visible equivalence relation* (Janhunen 2006) takes the interfaces of programs into account. The atoms in $\mathrm{At}(P)$ are partitioned into two parts, $\mathrm{At}_v(P)$ and $\mathrm{At}_h(P)$, which determine the *visible* and the *hidden* parts of $\mathrm{At}(P)$, respectively. Visible atoms form an interface for interaction between programs, and hidden atoms are local to each program and thus negligible when visible equivalence of programs is concerned.

Definition 3.14
SMODELS programs P and Q are visibly equivalent, denoted by $P \equiv_v Q$, if and only if $\mathrm{At}_v(P) = \mathrm{At}_v(Q)$ and there is a bijection $f : \mathrm{SM}(P) \to \mathrm{SM}(Q)$ such that for all $M \in \mathrm{SM}(P)$ it holds that $M \cap \mathrm{At}_v(P) = f(M) \cap \mathrm{At}_v(Q)$.

Note that the number of stable models is also preserved under $\equiv_v$. 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 the exact preservation of models is highly significant. In the fully visible case, that is, for $\mathrm{At}_h(P) = \mathrm{At}_h(Q) = \emptyset$, the relation $\equiv_v$ becomes very close to $\equiv$. The only difference is the requirement that $\mathrm{At}(P) = \mathrm{At}(Q)$ insisted on $\equiv_v$. This is of little importance as $\mathrm{At}(P)$ can always be extended by adding (tautological) rules of the form $a \leftarrow a$ to P without affecting the stable models of the program. Since weak equivalence is not a congruence for $\cup$, visible equivalence cannot be a congruence for program union either.

The verification of weak, strong, or uniform equivalence is a **coNP**-complete decision problem for SMODELS programs (Marek and Truszczyński 1991; Pearce *et al.*

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

2001; Eiter and Fink 2003). The computational complexity of deciding $\equiv_v$ is analyzed by Janhunen and Oikarinen (2007). If the use of hidden atoms is not limited in any way, the problem of verifying visible equivalence becomes at least as hard as the counting problem #**SAT** which is #**P**-complete (Valiant 1979). It is possible, however, to govern the computational complexity by limiting the use of hidden atoms by the property of having *enough visible atoms* (Janhunen and Oikarinen 2007). Intuitively, if P has enough visible atoms, the EVA property for short, then each interpretation of $\mathrm{At}_v(P)$ *uniquely* determines an interpretation of $\mathrm{At}_h(P)$. Consequently, the stable models of P can be distinguished on the basis of their visible parts. Although verifying the EVA property can be hard in general (Janhunen and Oikarinen 2007, Proposition 4.14), there are syntactic subclasses of SMODELS programs with the EVA property. The use of visible atoms remains unlimited and thus the full expressiveness of SMODELS programs remains at programmer's disposal. Also note that the EVA property can always be achieved by declaring sufficiently many atoms visible. For SMODELS programs with the EVA property, the verification of visible equivalence is a **coNP**-complete decision problem (Janhunen and Oikarinen 2007).

Eiter *et al.* (2005) introduce a very general framework based on *equivalence frames* to capture various kinds of equivalence relations. All the equivalence relations defined in this section can be defined using the framework. Visible equivalence, however, is exceptional in the sense that it does not fit into equivalence frames based on *projected answer sets*. As a consequence, the number of answer sets may not be preserved which is somewhat unsatisfactory because of the general nature of ASP as discussed in the previous section. Under the EVA assumption, however, the *projective variant of visible equivalence* defined by

$$\{M \cap \mathrm{At}_v(P) \mid M \in \mathrm{SM}(P)\} = \{N \cap \mathrm{At}_v(Q) \mid N \in \mathrm{SM}(Q)\}$$

coincides with visible equivalence.

Recently, Woltran (2007) presented another general framework characterizing $\langle \mathscr{H}, \mathscr{B} \rangle$*-equivalence*. $\langle \mathscr{H}, \mathscr{B} \rangle$-equivalence is defined similarly to strong equivalence, but the set of possible contexts is restricted by limiting the head and body occurrences of atoms in a context program R by $\mathscr{H}$ and $\mathscr{B}$, respectively. Thus, programs P and Q are $\langle \mathscr{H}, \mathscr{B} \rangle$-equivalent if and only if $P \cup R \equiv Q \cup R$ for all R such that $\mathrm{Head}(R) \subseteq \mathscr{H}$ and $\mathrm{Body}(R) \subseteq \mathscr{B}$. Several notions of equivalence such as weak equivalence together with (relativized) strong and (relativized) uniform equivalence can be seen as special cases of $\langle \mathscr{H}, \mathscr{B} \rangle$-equivalence by varying the sets $\mathscr{H}$ and $\mathscr{B}$.

4 SMODELS program modules

We start this section by introducing the syntax and the stable model semantics for an individual SMODELS program module and then formalize the conditions for module composition. One of the main results is the *module theorem* showing that module composition is suitably restricted so that compositionality of stable model semantics for SMODELS programs is achieved. We also introduce an equivalence relation for modules and propose a general translation-based scheme for introducing syntactical extensions for the module theorem. The scheme is then utilized in the proof of the

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

module theorem. We end this section with a brief comparison between our module architecture and other similar proposals.

4.1 Syntax and semantics of an SMODELS *program module*

We define a *logic program module* similar to that of Gaifman and Shapiro (1989) but consider the case of SMODELS programs instead of positive NLPs covered by Gaifman and Shapiro (1989). An analogous module system in the context of *disjunctive logic programs* is presented in (Janhunen *et al.* 2007).

Definition 4.1
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) $\mathrm{At}(R) \subseteq \mathrm{At}(\mathbb{P})$ which is defined by $\mathrm{At}(\mathbb{P}) = I \cup O \cup H$; and
(4) $\mathrm{Head}(R) \cap I = \emptyset$.

The atoms in $\mathrm{At_v}(\mathbb{P}) = I \cup O$ 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 $\mathrm{At_i}(\mathbb{P})$ and $\mathrm{At_o}(\mathbb{P})$ for referring to the *input signature* I and the *output signature* O, respectively. The *hidden* atoms in $\mathrm{At_h}(\mathbb{P}) = H = \mathrm{At}(\mathbb{P}) \setminus \mathrm{At_v}(\mathbb{P})$ are used to formalize some auxiliary concepts of $\mathbb{P}$ which may not be sensible for other modules but may save space substantially. The use of hidden atoms may yield exponential savings in space (see Janhunen and Oikarinen 2007, Example 4.5, for instance). 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, input atoms are only allowed to appear as conditions in rule bodies.

Example 4.2
Consider the Hamiltonian cycle problem for directed graphs, that is, whether there is a cycle in the graph such that each node is visited exactly once returning to the starting node.

Let n denote the number of nodes in the graph and let $\mathsf{arc}(x, y)$ denote that there is a directed edge from node x to node y in the graph.

Module $\mathbb{H}^n = \langle R, I, O, \{c, d\}\rangle$ selects the edges to be taken into a cycle by insisting that each node must have exactly one incoming and exactly one outgoing edge. The input signature of $\mathbb{H}^n$ is a graph represented as a set of edges: $I = \{\mathsf{arc}(x, y) \mid 1 \leqslant x, y \leqslant n\}$. The output signature of $\mathbb{H}^n$ represents which edges get selected into a candidate for a Hamiltonian cycle: $O = \{\mathsf{hc}(x, y) \mid 1 \leqslant x, y \leqslant n\}$. The set R contains rules

$$\{\mathsf{hc}(x, y)\} \leftarrow \mathsf{arc}(x, y), \tag{8}$$

$$c \leftarrow 2 \leqslant \{\mathsf{hc}(x, 1), \ldots, \mathsf{hc}(x, n)\}, \tag{9}$$

$$c \leftarrow \sim\mathsf{hc}(x, 1), \ldots, \sim\mathsf{hc}(x, n), \tag{10}$$

$$c \leftarrow 2 \leqslant \{\mathsf{hc}(1, x), \ldots, \mathsf{hc}(n, x)\}, \text{ and} \tag{11}$$

$$c \leftarrow \sim\mathsf{hc}(1, x), \ldots, \sim\mathsf{hc}(n, x), \tag{12}$$

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

for each $1 \leqslant x,\ y \leqslant n$, and a rule $d \leftarrow \sim d, c$ which enforces c to be false in every stable model. The rules in (8) encode the selection of edges taken in the cycle. The rules in (9) and (10) are used to guarantee that each node has exactly one outgoing edge, and the rules in (11) and (12) give the respective condition concerning incoming edges.

We also need to check that each node is reachable from the first node along the edges in the cycle. For this, we introduce module $\mathbb{R}^n = \langle R', I', O', \{e\}\rangle$. The input signature of $\mathbb{R}^n$ is $I' = O$, and the output signature is $O' = \{\mathsf{reached}(x) \mid 1 \leqslant x \leqslant n\}$, where $\mathsf{reached}(x)$ tells that node x is reachable from the first node. The set R' contains rules

$$\begin{aligned} \mathsf{reached}(y) &\leftarrow \mathsf{hc}(1, y), \\ \mathsf{reached}(y) &\leftarrow \mathsf{reached}(x), \mathsf{hc}(x, y), \ \textit{and} \\ e &\leftarrow \sim e, \sim\mathsf{reached}(y), \end{aligned}$$

for each $2 \leqslant x \leqslant n$ and $1 \leqslant y \leqslant n$. □

To generalize the stable model semantics to cover modules as well, we must explicate the semantical role of input atoms. To this end, we will follow an approach[5] from (Janhunen *et al.* 2007) and take input atoms into account in the definition of the reduct adopted from (Janhunen and Oikarinen 2007). 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 (Gelfond and Lifschitz 1988) and SMODELS programs (Janhunen and Oikarinen 2007) if an empty input signature $I = \emptyset$ is additionally assumed. Using the same idea, a conventional SMODELS program, that is, a set of basic constraint rules R, can be viewed as a module $\langle R, \emptyset, \mathrm{At}(R), \emptyset\rangle$ without any input atoms and all atoms visible.

Definition 4.3
Given a module $\mathbb{P} = \langle R, I, O, H\rangle$, the *reduct* of R with respect to an interpretation $M \subseteq \mathrm{At}(\mathbb{P})$ and input signature I, denoted by $R^{M,I}$, contains

(1) a rule $a \leftarrow (B \setminus I)$ if and only if 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) a rule $a \leftarrow w' \leqslant \{B \setminus I = W_{B\setminus I}\}$ if and only if there is a weight rule $a \leftarrow w \leqslant \{B = W_B, \sim C = W_C\}$ in R, and

$$w' = \max\left(0, w - \sum_{b \in B \cap I \cap M} w_b - \sum_{c \in C \setminus M} w_c\right).$$

As all occurrences of atoms in the input signature and all negative occurrences of atoms are evaluated, the generalized reduct $R^{M,I}$ is a positive program in the

[5] There are 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 (Oikarinen and Janhunen 2006; Oikarinen 2007). Yet another approach is to interpret input atoms as *fixed atoms* in the sense of parallel circumscription (Lifschitz 1985).

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

sense of Janhunen and Oikarinen (2007) and thus it has a unique least model $\mathrm{LM}(R^{M,I}) \subseteq \mathrm{At}(R) \setminus I$.

Definition 4.4
An interpretation $M \subseteq \mathrm{At}(\mathbb{P})$ is a stable model of an SMODELS program module $\mathbb{P} = \langle R, I, O, H \rangle$, denoted by $M \in \mathrm{SM}(\mathbb{P})$, if and only if $M \setminus I = \mathrm{LM}(R^{M,I})$.

If one is interested in computing stable models of a module with respect to certain input interpretation, it is easier to use an alternative definition of stable semantics for modules (Oikarinen and Janhunen 2006), where an actual input is seen as a set of facts (or a database) to be combined with the module.

Definition 4.5
Given an SMODELS program module $\mathbb{P} = \langle R, I, O, H \rangle$ and a set of atoms $A \subseteq I$, the instantiation of $\mathbb{P}$ with an actual input A is

$$\mathbb{P}(A) = \langle R \cup \{a. \mid a \in A\}, \emptyset, I \cup O, H \rangle.$$

The module $\mathbb{P}(A)$ is essentially an SMODELS program with $I \cup O$ as the set of visible atoms. Thus the stable model semantics of SMODELS programs in Definition 3.3 directly generalizes for an instantiated module.

Definition 4.6
An interpretation $M \subseteq \mathrm{At}(\mathbb{P})$ is a stable model of an SMODELS program module $\mathbb{P} = \langle R, I, O, H \rangle$ if and only if $M = \mathrm{LM}(R^M \cup \{a. \mid a \in M \cap I\})$.

It is worth emphasizing that Definitions 4.4 and 4.6 result in exactly the same semantics for SMODELS program modules.

Example 4.7
Recall module $\mathbb{H}^n$ from Example 4.2. We consider the stable models of $\mathbb{H}^n$ for $n = 2$ to see that the rules in $\mathbb{H}^n$ do not alone guarantee that each node is reachable along the edges taken in the cycle candidate. Consider $M = \{\mathsf{arc}(1,1), \mathsf{arc}(2,2), \mathsf{hc}(1,1), \mathsf{hc}(2,2)\}$. The reduct $R^{M,I}$ contains facts $\mathsf{hc}(1,1)$ and $\mathsf{hc}(2,2)$; and rules $c \leftarrow 2 \leqslant \{\mathsf{hc}(1,1), \mathsf{hc}(1,2)\}$, $c \leftarrow 2 \leqslant \{\mathsf{hc}(2,1), \mathsf{hc}(2,2)\}$, $c \leftarrow 2 \leqslant \{\mathsf{hc}(1,1), \mathsf{hc}(2,1)\}$, and $c \leftarrow 2 \leqslant \{\mathsf{hc}(1,2), \mathsf{hc}(2,2)\}$; and finally the rule $d \leftarrow c$. Now $M \in \mathrm{SM}(\mathbb{H}^n)$ since $M \setminus I = \mathrm{LM}(R^{M,I})$. However, M does not correspond to a graph with a Hamiltonian cycle, as node 2 is not reachable from node 1. □

4.2 Composing programs from modules

The stable model semantics (Gelfond and Lifschitz 1988) 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*.

Gaifman and Shapiro (1989) cover positive normal programs under logical consequences. 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*, that is, $\mathrm{At}_\mathrm{h}(\mathbb{P}_1) \cap \mathrm{At}(\mathbb{P}_2) = \emptyset$ and $\mathrm{At}_\mathrm{h}(\mathbb{P}_2) \cap \mathrm{At}(\mathbb{P}_1) = \emptyset$.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Definition 4.8
Given SMODELS program modules $\mathbb{P}_1 = \langle R_1, I_1, O_1, H_1 \rangle$ and $\mathbb{P}_2 = \langle R_2, I_2, O_2, H_2 \rangle$, their composition 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 $\mathrm{At_o}(\mathbb{P}_1) \cap \mathrm{At_o}(\mathbb{P}_2) = \emptyset$ and $\mathbb{P}_1$ and $\mathbb{P}_2$ respect each other's hidden atoms.

The following example shows that the conditions given for $\oplus$ are not enough to guarantee compositionality in the case of stable models and further restrictions on program composition become necessary.

Example 4.9
Consider NLP modules $\mathbb{P}_1 = \langle \{a \leftarrow b.\}, \{b\}, \{a\}, \emptyset \rangle$ and $\mathbb{P}_2 = \langle \{b \leftarrow a.\}, \{a\}, \{b\}, \emptyset \rangle$ both of which have stable models $\emptyset$ and $\{a, b\}$ by symmetry. The *composition* of $\mathbb{P}_1$ and $\mathbb{P}_2$ is $\mathbb{P}_1 \oplus \mathbb{P}_2 = \langle \{a \leftarrow b.\ b \leftarrow a.\}, \emptyset, \{a, b\}, \emptyset \rangle$ and $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \{\emptyset\}$, that is, $\{a, b\}$ is not a stable model of $\mathbb{P}_1 \oplus \mathbb{P}_2$. □

We define the positive dependency graph of an SMODELS program module $\mathbb{P} = \langle R, I, O, H \rangle$ as $\mathrm{Dep}^+(\mathbb{P}) = \mathrm{Dep}^+(R)$. Given that $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined, we say that $\mathbb{P}_1$ and $\mathbb{P}_2$ are *mutually dependent* if and only if $\mathrm{Dep}^+(\mathbb{P}_1 \oplus \mathbb{P}_2)$ has an SCC S such that $S \cap \mathrm{At_o}(\mathbb{P}_1) \neq \emptyset$ and $S \cap \mathrm{At_o}(\mathbb{P}_2) \neq \emptyset$, that is, S is *shared by* $\mathbb{P}_1$ and $\mathbb{P}_2$.

Definition 4.10
The *join* $\mathbb{P}_1 \sqcup \mathbb{P}_2$ of two SMODELS program modules $\mathbb{P}_1$ and $\mathbb{P}_2$ is $\mathbb{P}_1 \oplus \mathbb{P}_2$, provided $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined and $\mathbb{P}_1$ and $\mathbb{P}_2$ are not mutually dependent.

Example 4.11
Consider modules $\mathbb{H}^n$ and $\mathbb{R}^n$ from Example 4.2. Since $\mathbb{H}^n$ and $\mathbb{R}^n$ respect each other's hidden atoms and are not mutually dependent, their join $\mathbb{H}^n \sqcup \mathbb{R}^n = \langle R \cup R', I, O \cup O', \{c, d, e\} \rangle$ is defined. □

The conditions in Definition 4.10 impose no restrictions on positive dependencies *inside* modules or on *negative* dependencies in general. It is straightforward to show that $\sqcup$ has the following properties:

(i) *Identity*: $\mathbb{P} \sqcup \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \sqcup \mathbb{P} = \mathbb{P}$ for all modules $\mathbb{P}$.
(ii) *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.
(iii) *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 all pairwise joins are defined.

The equality "=" used above denotes syntactical equality. Also note that $\mathbb{P} \sqcup \mathbb{P}$ is usually undefined, which is a difference with respect to $\cup$ for which it holds that $P \cup P = P$ for all programs P. Furthermore, considering the join $\mathbb{P}_1 \sqcup \mathbb{P}_2$, since each atom is defined in exactly one module, the sets of rules in $\mathbb{P}_1$ and $\mathbb{P}_2$ are distinct, that is, $R_1 \cap R_2 = \emptyset$, and also, $\mathrm{At}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{At}(\mathbb{P}_1) \cup \mathrm{At}(\mathbb{P}_2)$, $\mathrm{At_v}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{At_v}(\mathbb{P}_1) \cup \mathrm{At_v}(\mathbb{P}_2)$, and $\mathrm{At_h}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{At_h}(\mathbb{P}_1) \cup \mathrm{At_h}(\mathbb{P}_2)$.

Having the semantics of an individual SMODELS program module now defined, we may characterize the properties of the semantics under program composition using the notion of *compatibility*.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Definition 4.12
Given SMODELS program modules $\mathbb{P}_1$ and $\mathbb{P}_2$ such that $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined, we say that interpretations $M_1 \subseteq \mathrm{At}(\mathbb{P}_1)$ and $M_2 \subseteq \mathrm{At}(\mathbb{P}_2)$ are compatible if and only if $M_1 \cap \mathrm{At_v}(\mathbb{P}_2) = M_2 \cap \mathrm{At_v}(\mathbb{P}_1)$.

We use *natural join* $\bowtie$ to combine compatible interpretations.

Definition 4.13
Given SMODELS program modules $\mathbb{P}_1$ and $\mathbb{P}_2$ and sets of interpretations $A_1 \subseteq \mathbf{2}^{\mathrm{At}(\mathbb{P}_1)}$ and $A_2 \subseteq \mathbf{2}^{\mathrm{At}(\mathbb{P}_2)}$, the natural join of A_1 and A_2, denoted by $A_1 \bowtie A_2$, is

$$\{M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2 \text{ such that } M_1 \text{ and } M_2 \text{ are compatible}\}.$$

The stable model semantics is compositional for $\sqcup$, that is, if a program (module) consists of several submodules, its stable models are locally stable for the respective submodules, and, on the other hand, local stability implies global stability for compatible stable models of the submodules.

Theorem 4.14 (*Module theorem (Oikarinen 2007)*)
If $\mathbb{P}_1$ and $\mathbb{P}_2$ are SMODELS program modules such that $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined, then

$$\mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2).$$

Instead of proving Theorem 4.14 directly from scratch we will propose *a general translation-based scheme* for introducing syntactical extensions for the module theorem. For this we need to define a concept of *modular equivalence* first and thus the proof of Theorem 4.14 is deferred until Section 4.4.

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 in Theorem 4.14, that is, for any SMODELS programs P_1 and P_2, $\mathrm{CM}(P_1 \cup P_2) = \mathrm{CM}(P_1) \bowtie \mathrm{CM}(P_2)$, where $\mathrm{CM}(P) = \{M \subseteq \mathrm{At}(P) \mid M \models P\}$.

Example 4.15
Recall modules $\mathbb{H}^n$ and $\mathbb{R}^n$ given in Example 4.2. In Example 4.7 we showed that $M = \{\mathsf{arc}(1,1), \mathsf{arc}(2,2), \mathsf{hc}(1,1), \mathsf{hc}(2,2)\}$ is a stable model of $\mathbb{H}^2$. Now module $\mathbb{R}^2$ has six stable models, but none of them is compatible with M. Thus, by Theorem 4.14 there is no stable model N for $\mathbb{H}^2 \sqcup \mathbb{R}^2$ such that $N \cap \mathrm{At}(\mathbb{H}^2) = M$.

The join $\mathbb{H}^n \sqcup \mathbb{R}^n$ can be used to find any graph of n nodes which has a Hamiltonian cycle. For instance $\mathbb{H}^2 \sqcup \mathbb{R}^2$ has following four stable models:

$$\{\mathsf{arc}(1,2), \mathsf{arc}(2,1), \mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{reached}(1), \mathsf{reached}(2)\},$$
$$\{\mathsf{arc}(1,1), \mathsf{arc}(1,2), \mathsf{arc}(2,1), \mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{reached}(1), \mathsf{reached}(2)\},$$
$$\{\mathsf{arc}(1,2), \mathsf{arc}(2,1), \mathsf{arc}(2,2), \mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{reached}(1), \mathsf{reached}(2)\}, \textit{ and}$$
$$\{\mathsf{arc}(1,1), \mathsf{arc}(1,2), \mathsf{arc}(2,1), \mathsf{arc}(2,2), \mathsf{hc}(1,2), \mathsf{hc}(2,1), \mathsf{reached}(1), \mathsf{reached}(2)\}.$$

These models represent the four possible graphs of two nodes having a Hamiltonian cycle. □

Theorem 4.14 straightforwardly generalizes for modules consisting of several submodules. Consider a collection of SMODELS program modules $\mathbb{P}_1, \ldots, \mathbb{P}_n$ such

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

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 interpretations $\{M_1, \ldots, M_n\}$ for modules $\mathbb{P}_1, \ldots, \mathbb{P}_n$, respectively, is *compatible*, if and only if M_i and M_j are pairwise compatible for all $1 \leqslant i, j \leqslant n$. The natural join generalizes for a collection of modules as

$$A_1 \bowtie \cdots \bowtie A_n = \{M_1 \cup \cdots \cup M_n \mid M_i \in A_i \text{ and } \{M_1, \ldots, M_n\} \text{ is compatible}\},$$

where $A_1 \subseteq \mathbf{2}^{\mathrm{At}(\mathbb{P}_1)}, \ldots, A_n \subseteq \mathbf{2}^{\mathrm{At}(\mathbb{P}_n)}$.

Corollary 4.16
For a collection of SMODELS program modules $\mathbb{P}_1, \ldots, \mathbb{P}_n$ such that the join $\mathbb{P}_1 \sqcup \cdots \sqcup \mathbb{P}_n$ is defined, it holds that

$$\mathrm{SM}(\mathbb{P}_1 \sqcup \cdots \sqcup \mathbb{P}_n) = \mathrm{SM}(\mathbb{P}_1) \bowtie \cdots \bowtie \mathrm{SM}(\mathbb{P}_n).$$

Although, Corollary 4.16 enables the computation of stable models on a module-by-module basis, it leaves us the task of excluding mutually incompatible combinations of stable models. It should be noted that applying the module theorem in a naive way by first computing stable models for each submodule and then finding the compatible pairs afterwards, might not be preferable.

Example 4.17
Consider SMODELS program modules

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

and their join $\mathbb{P} = \mathbb{P}_1 \sqcup \mathbb{P}_2 \sqcup \mathbb{P}_3 = \langle\{a \leftarrow {\sim}b.\ b \leftarrow {\sim}c.\ c \leftarrow {\sim}a.\}, \emptyset, \{a, b, c\}, \emptyset\rangle$. We have $\mathrm{SM}(\mathbb{P}_1) = \{\{a\}, \{b\}\}$, $\mathrm{SM}(\mathbb{P}_2) = \{\{b\}, \{c\}\}$, and $\mathrm{SM}(\mathbb{P}_3) = \{\{a\}, \{c\}\}$. To apply Corollary 4.16 for finding $\mathrm{SM}(\mathbb{P})$, a naive approach is to compute all stable models of all the modules and try 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 $\{a\} \cap \mathrm{At_v}(\mathbb{P}_2) = \emptyset = \{c\} \cap \mathrm{At_v}(\mathbb{P}_1)$. However, $\{a\} \in \mathrm{SM}(\mathbb{P}_3)$ is not compatible with $\{c\} \in \mathrm{SM}(\mathbb{P}_2)$, since $\{c\} \cap \mathrm{At_v}(\mathbb{P}_3) = \{c\} \neq \emptyset = \{a\} \cap \mathrm{At_v}(\mathbb{P}_2)$. On the other hand, $\{c\} \in \mathrm{SM}(\mathbb{P}_3)$ is not compatible with $\{a\} \in \mathrm{SM}(\mathbb{P}_1)$, since $\{a\} \cap \mathrm{At_v}(\mathbb{P}_3) = \{a\} \neq \emptyset = \{c\} \cap \mathrm{At_v}(\mathbb{P}_1)$.
- 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, there are no $M_1 \in \mathrm{SM}(\mathbb{P}_1)$, $M_2 \in \mathrm{SM}(\mathbb{P}_2)$, and $M_3 \in \mathrm{SM}(\mathbb{P}_3)$ such that $\{M_1, M_2, M_3\}$ is compatible, which is natural as $\mathrm{SM}(\mathbb{P}) = \emptyset$. □

It is not necessary to test all combinations of stable models to see whether we have a compatible triple. Instead, we use the alternative definition of stable models (Definition 4.6) based on instantiating the module with respect to an input interpretation, and apply the module theorem similarly to the splitting-set theorem.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

One should notice that the set of rules in $\mathbb{P}$ presented in Example 4.17 has no nontrivial splitting sets, and thus the splitting-set theorem is not applicable (in a nontrivial way) in this case.

Example 4.18
Consider SMODELS program modules $\mathbb{P}_1$, $\mathbb{P}_2$, and $\mathbb{P}_3$ from Example 4.17. Now, $\mathbb{P}_1$ has two stable models $M_1 = \{a\}$ and $M_2 = \{b\}$.

- The set $M_1 \cap \mathrm{At_i}(\mathbb{P}_3) = \{a\} = A_1$ can be seen as an input interpretation for $\mathbb{P}_3$. Module $\mathbb{P}_3$ instantiated with A_1 has one stable model: $\mathrm{SM}(\mathbb{P}_3(A_1)) = \{\{a\}\}$. Furthermore, we can use $A_2 = \{a\} \cap \mathrm{At_i}(\mathbb{P}_2) = \emptyset$ to instantiate $\mathbb{P}_2$: $\mathrm{SM}(\mathbb{P}_2(A_2)) = \{\{b\}\}$. However, $\{b\}$ is not compatible with M_1, and thus there is no way to find a compatible collection of stable models for the modules starting from M_1.
- We instantiate $\mathbb{P}_3$ with $M_2 \cap \mathrm{At_i}(\mathbb{P}_3) = \emptyset = A_3$ and get $\mathrm{SM}(\mathbb{P}_3(A_3)) = \{\{c\}\}$. Continuing with $\{c\} \cap \mathrm{At_i}(\mathbb{P}_2) = \{c\} = A_4$, we get $\mathrm{SM}(\mathbb{P}_2(A_4)) = \{\{c\}\}$. Again, we notice that $\{c\}$ is not compatible with M_2, and thus it is not possible to find a compatible triple of stable models starting from M_2 either.

Thus we can conclude $\mathrm{SM}(\mathbb{P}) = \emptyset$. □

4.3 Equivalence relations for modules

The notion of *visible equivalence* (Janhunen 2006) 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 4.14 enables us to bring the same idea to the level of program modules—giving rise to *modular equivalence* of logic programs. Visible and modular equivalence are formulated for SMODELS program modules as follows.

Definition 4.19
For two SMODELS program modules $\mathbb{P}$ and $\mathbb{Q}$,

- $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$ if and only if $\mathrm{At_v}(\mathbb{P}) = \mathrm{At_v}(\mathbb{Q})$ and there is a bijection $f : \mathrm{SM}(\mathbb{P}) \to \mathrm{SM}(\mathbb{Q})$ such that for all $M \in \mathrm{SM}(\mathbb{P})$,

$$M \cap \mathrm{At_v}(\mathbb{P}) = f(M) \cap \mathrm{At_v}(\mathbb{Q}); \text{ and}$$

- $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ if and only if $\mathrm{At_i}(\mathbb{P}) = \mathrm{At_i}(\mathbb{Q})$ and $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$.

We note that the condition $\mathrm{At_v}(\mathbb{P}) = \mathrm{At_v}(\mathbb{Q})$ insisted on the definition of $\equiv_{\mathrm{v}}$, implies $\mathrm{At_o}(\mathbb{P}) = \mathrm{At_o}(\mathbb{Q})$ in the presence of $\mathrm{At_i}(\mathbb{P}) = \mathrm{At_i}(\mathbb{Q})$ as required by the relation $\equiv_{\mathrm{m}}$. Moreover, these relations coincide for completely specified SMODELS programs, that is module $\mathbb{P}$ with $\mathrm{At_i}(\mathbb{P}) = \emptyset$.

Modular equivalence lends itself for program substitutions in analogy to *strong equivalence* (Lifschitz *et al.* 2001), that is, the relation $\equiv_{\mathrm{m}}$ is a proper *congruence* for the join operator $\sqcup$.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Theorem 4.20 (*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 $\mathbb{P} \sqcup \mathbb{R} \equiv_{\mathrm{m}} \mathbb{Q} \sqcup \mathbb{R}$.

The proof of Theorem 4.20 is given in the Appendix. The following examples illustrate the use of modular equivalence in practice.

Example 4.21
Recall programs $P = \{a.\}$ and $Q = \{a \leftarrow \sim b.\ a \leftarrow b.\}$ from Example 3.13. We can define modules based on them: $\mathbb{P} = \langle P, \{b\}, \{a\}, \emptyset \rangle$ and $\mathbb{Q} = \langle Q, \{b\}, \{a\}, \emptyset \rangle$. Now it is impossible to define a module $\mathbb{R}$ based on $R = \{b \leftarrow a.\}$ in a way that $\mathbb{Q} \sqcup \mathbb{R}$ would be defined. Moreover, it holds that $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$. □

Example 4.22
Module $\mathbb{HR}^n = \langle R'', I'', O'', \{f\} \rangle$ is based on an alternative encoding for Hamiltonian cycle problem given by Simons *et al.* (2002). In contrast to the encoding described in Example 4.2, this encoding does not allow us to separate the selection of the edges to the cycle and the checking of reached vertices into separate modules as their definitions are mutually dependent. The input signature of $\mathbb{HR}^n$ is the same as for $\mathbb{H}^n$, that is, $I'' = I = \{\mathsf{arc}(x, y) \mid 1 \leqslant x, y \leqslant n\}$. The output signature of $\mathbb{HR}^n$ is the output signature of $\mathbb{H}^n \sqcup \mathbb{R}^n$, that is,

$$O'' = O \cup O' = \{\mathsf{hc}(x, y) \mid 1 \leqslant x, y \leqslant n\} \cup \{\mathsf{reached}(x) \mid 1 \leqslant x \leqslant n\}.$$

The set R'' contains rules

$$\begin{aligned}
\{\mathsf{hc}(1, x)\} &\leftarrow \mathsf{arc}(1, x), \\
\{\mathsf{hc}(x, y)\} &\leftarrow \mathsf{reached}(x), \mathsf{arc}(x, y), \\
\mathsf{reached}(y) &\leftarrow \mathsf{hc}(x, y), \\
f &\leftarrow \sim f, \sim \mathsf{reached}(x), \\
f &\leftarrow \sim f, \mathsf{hc}(x, y), \mathsf{hc}(x, z), \text{ and} \qquad (13) \\
f &\leftarrow \sim f, \mathsf{hc}(x, y), \mathsf{hc}(z, y), \qquad (14)
\end{aligned}$$

for each $1 \leqslant x, y, z \leqslant n$ such that $y \neq z$ in (13) and $x \neq z$ in (14).

Now, one may notice that $\mathbb{HR}^n$ and $\mathbb{H}^n \sqcup \mathbb{R}^n$ have the same input/output interface, and $\mathrm{SM}(\mathbb{HR}^n) = \mathrm{SM}(\mathbb{H}^n \sqcup \mathbb{R}^n)$ which implies $\mathbb{HR}^n \equiv_{\mathrm{m}} \mathbb{H}^n \sqcup \mathbb{R}^n$. □

As regards the relationship between modular equivalence and previously proposed notions of equivalence, we note the following: First, if one considers the *fully visible case*, that is, the restriction $\mathrm{At_h}(\mathbb{P}) = \mathrm{At_h}(\mathbb{Q}) = \emptyset$, modular equivalence can be seen as a special case of A-uniform equivalence for $A = I$. Recall, however, the restriction that input atoms may not appear in the heads of the rules as imposed by module structure. With a further restriction $\mathrm{At_i}(\mathbb{P}) = \mathrm{At_i}(\mathbb{Q}) = \emptyset$, modular equivalence basically coincides with weak equivalence because $\mathrm{At}(\mathbb{P}) = \mathrm{At}(\mathbb{Q})$ can always be satisfied by extending the interface of the module. Setting $\mathrm{At_i}(\mathbb{P}) = \mathrm{At}(\mathbb{P})$ would in principle give us uniform equivalence, but the additional condition $\mathrm{Head}(R) \cap I = \emptyset$ leaves room for the empty module only.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

In the general case with hidden atoms, the problem of verifying $\equiv_{\mathrm{m}}$ for SMODELS program modules can be reduced to verifying $\equiv_{\mathrm{v}}$ for SMODELS programs. This is achieved by introducing a special module $\mathbb{G}_I$ containing a single choice rule, which acts as a context generator in analogy to (Woltran 2004). We say that two modules $\mathbb{P}$ and $\mathbb{Q}$ are *compatible* if they have the same input/output interface, that is, $\mathrm{At_i}(\mathbb{P}) = \mathrm{At_i}(\mathbb{Q})$ and $\mathrm{At_o}(\mathbb{P}) = \mathrm{At_o}(\mathbb{Q})$.

Lemma 4.23
Consider compatible SMODELS program modules $\mathbb{P}$ and $\mathbb{Q}$. Now $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ if and only if $\mathbb{P} \sqcup \mathbb{G}_I \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{G}_I$ where $I = \mathrm{At_i}(\mathbb{P}) = \mathrm{At_i}(\mathbb{Q})$ and $\mathbb{G}_I = \langle \{\{I\} \leftarrow\}, \emptyset, I, \emptyset \rangle$ generates all possible input interpretations for $\mathbb{P}$ and $\mathbb{Q}$.

Proof
Notice that $\mathbb{P} \sqcup \mathbb{G}_I$ and $\mathbb{Q} \sqcup \mathbb{G}_I$ are SMODELS program modules with empty input signatures, and thus they can also be viewed as SMODELS programs. ($\Longrightarrow$) Assume $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$. Since $\mathbb{P} \sqcup \mathbb{G}_I$ and $\mathbb{Q} \sqcup \mathbb{G}_I$ are defined, $\mathbb{P} \sqcup \mathbb{G}_I \equiv_{\mathrm{m}} \mathbb{Q} \sqcup \mathbb{G}_I$ by Theorem 4.20. This implies $\mathbb{P} \sqcup \mathbb{G}_I \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{G}_I$. ($\Longleftarrow$) Assume $\mathbb{P} \sqcup \mathbb{G}_I \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{G}_I$, that is, $\mathrm{At_v}(\mathbb{P}) = \mathrm{At_v}(\mathbb{Q})$ and there is a bijection $f : \mathrm{SM}(\mathbb{P} \sqcup \mathbb{G}_I) \rightarrow \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{G}_I)$ such that for each $M \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{G}_I)$, $M \cap \mathrm{At_v}(\mathbb{P}) = f(M) \cap \mathrm{At_v}(\mathbb{Q})$. By Theorem 4.14, $\mathrm{SM}(\mathbb{P} \sqcup \mathbb{G}_I) = \mathrm{SM}(\mathbb{P}) \bowtie \mathrm{SM}(\mathbb{G}_I)$ and $\mathrm{SM}(\mathbb{Q} \sqcup \mathbb{G}_I) = \mathrm{SM}(\mathbb{Q}) \bowtie \mathrm{SM}(\mathbb{G}_I)$. Now, $\mathrm{SM}(\mathbb{G}_I) = \mathbf{2}^I$, and thus $\mathrm{SM}(\mathbb{P} \sqcup \mathbb{G}_I) = \mathrm{SM}(\mathbb{P})$ and $\mathrm{SM}(\mathbb{Q} \sqcup \mathbb{G}_I) = \mathrm{SM}(\mathbb{Q})$. This implies $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$, and furthermore $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ since $\mathbb{P}$ and $\mathbb{Q}$ are compatible SMODELS program modules. □

Due to the close relationship of $\equiv_{\mathrm{v}}$ and $\equiv_{\mathrm{m}}$, the respective verification problems have the same computational complexity. As already observed in Janhunen and Oikarinen (2007), the verification of $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$ involves a *counting problem* in general and, in particular, if $\mathrm{At_v}(\mathbb{P}) = \mathrm{At_v}(\mathbb{Q}) = \emptyset$. In this special setting, $\mathbb{P} \equiv_{\mathrm{v}} \mathbb{Q}$ holds if and only if $|\mathrm{SM}(\mathbb{P})| = |\mathrm{SM}(\mathbb{Q})|$, that is, the numbers of stable models for $\mathbb{P}$ and $\mathbb{Q}$ coincide. A reduction of computational time complexity can be achieved for modules that have *enough visible atoms*, that is, the EVA property. Basically, we say that module $\mathbb{P} = \langle R, I, O, H \rangle$ has enough visible atoms, if and only if R has enough visible atoms with respect to $\mathrm{At_v}(P) = I \cup O$. However, the property of having enough visible atoms can be elegantly stated using modules. 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 a choice rule $\{A\} \leftarrow B, {\sim}C \in R$, we take the projection $\{A \cap H\} \leftarrow B, {\sim}C$ in R_{h}.

Definition 4.24 (*The EVA property (Janhunen and Oikarinen 2007)*)
An SMODELS program module $\mathbb{P} = \langle R, I, O, H \rangle$ has enough visible atoms if and only if the hidden part $\mathbb{P}_{\mathrm{h}} = \langle R_{\mathrm{h}}, I \cup O, H, \emptyset \rangle$ has a unique stable model M for each interpretation $N \subseteq \mathrm{At_v}(\mathbb{P}) = I \cup O$ such that $M \cap (I \cup O) = N$.

Verifying the EVA property is **coNP**-hard and in $\mathbf{\Pi_2^P}$ for SMODELS programs (Janhunen and Oikarinen 2007, Proposition 4.14). The complexity result hold also for SMODELS program modules. It is always possible to enforce the EVA property by

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

uncovering sufficiently many hidden atoms: a module $\mathbb{P}$ for which $\mathrm{At_h}(\mathbb{P}) = \emptyset$ has clearly enough visible atoms because $\mathbb{P}_\mathrm{h}$ has no rules. It is also important to realize that choice rules involving hidden atoms in their heads most likely break up the EVA property—unless additional constraints are introduced to exclude multiple models created by choices.

On the basis of these observations we can conclude that verifying the modular equivalence of modules with the EVA property is a **coNP**-complete decision problem. Motivated by the complexity result and by previous proposals for translating various equivalence verification problems into the problem of computing stable models (see Janhunen and Oikarinen 2002; Turner 2003; Woltran 2004, for instance), we recently introduced a translation-based method for verifying modular equivalence (Oikarinen and Janhunen 2008). In the following theorem, $\mathrm{EQT}(\cdot,\cdot)$ is the linear translation function which maps two SMODELS program modules into one SMODELS program module presented in Oikarinen and Janhunen (2008, Definition 10).

Theorem 4.25 (*Oikarinen and Janhunen 2008, Theorem 4*)
Let $\mathbb{P}$ and $\mathbb{Q}$ be compatible SMODELS program modules with the EVA property, and $\mathbb{C}$ any SMODELS program module such that $\mathbb{P} \sqcup \mathbb{C}$ and $\mathbb{Q} \sqcup \mathbb{C}$ are defined. Then $\mathbb{P} \sqcup \mathbb{C} \equiv_\mathrm{m} \mathbb{Q} \sqcup \mathbb{C}$ if and only if $\mathrm{SM}(\mathrm{EQT}(\mathbb{P},\mathbb{Q}) \sqcup \mathbb{C}) = \mathrm{SM}(\mathrm{EQT}(\mathbb{Q},\mathbb{P}) \sqcup \mathbb{C}) = \emptyset$.

4.4 Proving the module theorem using a general translation-based extension scheme

Let us now proceed to the proof of the module theorem. We describe the overall strategy in this section whereas detailed proofs for the theorems are provided in the Appendix. Instead of proving Theorem 4.14 from scratch, we first show that the theorem holds for NLP modules, and then present a general scheme that enables us to derive extensions of the module theorem syntactically in terms of translations.

We start by stating the module theorem for NLP modules.

Theorem 4.26 (*Oikarinen and Janhunen 2006*)
If $\mathbb{P}_1$ and $\mathbb{P}_2$ are NLP modules such that $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined, then

$$\mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2).$$

Proof for Theorem 4.26 is given in the Appendix.

Next definition states the conditions required by a translation function in order to achieve syntactical extensions to the module theorem. Intuitively, the conditions serve the following purposes: first, the translation has to be *strongly faithful*, that is, it preserves the roles of all atoms in the original module; second, it is $\sqcup$-*preserving*, that is, possible compositions of modules are not limited by the translation; and third, the translation is *modular*.

For convenience, we define an operator $\mathrm{reveal}(\mathbb{P}, A) = \langle R, I, O \cup A, H \setminus A \rangle$ for any program module $\mathbb{P} = \langle R, I, O, H \rangle$ and for any set of atoms $A \subseteq H$. The revealing operator is used to make a set of hidden atoms of a module visible to other modules.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Definition 4.27
Let $\mathscr{C}_1$ and $\mathscr{C}_2$ be two classes of logic program modules such that $\mathscr{C}_2 \subseteq \mathscr{C}_1$. A translation function $\mathrm{Tr} : \mathscr{C}_1 \to \mathscr{C}_2$ is *strongly faithful, modular, and* $\sqcup$*-preserving*, if the following hold for any of the program modules $\mathbb{P}, \mathbb{Q} \in \mathscr{C}_1$:

(1) $\mathrm{reveal}(\mathbb{P}, \mathrm{At_h}(\mathbb{P})) \equiv_\mathrm{m} \mathrm{reveal}(\mathrm{Tr}(\mathbb{P}), \mathrm{At_h}(\mathbb{P}))$;
(2) if $\mathbb{P} \sqcup \mathbb{Q}$ is defined, then $\mathrm{Tr}(\mathbb{P}) \sqcup \mathrm{Tr}(\mathbb{Q})$ is defined; and
(3) $\mathrm{Tr}(\mathbb{P}) \sqcup \mathrm{Tr}(\mathbb{Q}) = \mathrm{Tr}(\mathbb{P} \sqcup \mathbb{Q})$.

Notice that the condition for strong faithfulness requires $\mathrm{At_i}(\mathrm{Tr}(\mathbb{P})) = \mathrm{At_i}(\mathbb{P})$, $\mathrm{At_o}(\mathrm{Tr}(\mathbb{P})) \cup \mathrm{At_h}(\mathbb{P}) = \mathrm{At_o}(\mathbb{P}) \cup \mathrm{At_h}(\mathbb{P})$, and $\mathrm{At_h}(\mathbb{P}) \subseteq \mathrm{At_h}(\mathrm{Tr}(\mathbb{P}))$ to hold. Moreover, strong faithfulness implies *faithfulness*, that is, $\mathbb{P} \equiv_\mathrm{m} \mathrm{Tr}(\mathbb{P})$.

Theorem 4.28
Let $\mathscr{C}_1$ and $\mathscr{C}_2$ be two classes of logic program modules such that $\mathscr{C}_2 \subseteq \mathscr{C}_1$ and there is a translation function $\mathrm{Tr} : \mathscr{C}_1 \to \mathscr{C}_2$ that is strongly faithful, $\sqcup$-preserving, and modular as given in Definition 4.27. If the module theorem holds for the modules in $\mathscr{C}_2$ then it holds for the modules in $\mathscr{C}_1$.

The proof of Theorem 4.28 is provided in the Appendix.

As regards the translation from SMODELS program modules to NLP modules, it suffices, for example, to take a natural translation similar to that by Simons *et al.* (2002). Note that the translation presented in Definition 4.29 is in the worst case exponential with respect to the number of rules in the original module. For a more compact translation see (Ferraris and Lifschitz 2005), for example.

Definition 4.29
Given an SMODELS program module $\mathbb{P} = \langle R, I, O, H \rangle$, its translation into an NLP module is $\mathrm{Tr_{NLP}}(\mathbb{P}) = \langle R', I, O, H \cup H' \rangle$, where R' contains the following rules:

- for each choice rule $\{A\} \leftarrow B, {\sim}C \in R$ the set of rules

$$\{a \leftarrow B, {\sim}C, {\sim}\overline{a}.\ \ \overline{a} \leftarrow {\sim}a \mid a \in A\};$$

- for each weight rule $a \leftarrow w \leqslant \{B = W_B, {\sim}C = W_C\} \in R$ the set of rules

$$\left\{ a \leftarrow B', {\sim}C' \mid B' \subseteq B, C' \subseteq C \text{ and } w \leqslant \sum_{b \in B'} w_b + \sum_{c \in C'} w_c \right\},$$

where each $\overline{a}$ is a new atom not appearing in $\mathrm{At}(\mathbb{P})$ and $H' = \{\overline{a} \mid a \in \mathrm{Choices}(R)\}$.

Theorem 4.30
The translation $\mathrm{Tr_{NLP}}$ from SMODELS program modules to NLP modules given in Definition 4.29 is strongly faithful, $\sqcup$-preserving, and modular.

The proof of Theorem 4.30 is given in the Appendix.

The module theorem now directly follows from Theorems 4.26, 4.28, and 4.30.

Proof of Theorem 4.14
By Theorem 4.26 we know that the module theorem holds for NLP modules. Theorem 4.28 shows that Definition 4.27 gives the conditions under which Theorem 4.26 can be directly generalized for a larger class of logic program modules. By

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Theorem 4.30 we know that the translation $\mathrm{Tr}_{\mathrm{NLP}}$ from SMODELS program modules to NLP modules introduced in Definition 4.29 satisfies the conditions given in Definition 4.27, and therefore SMODELS program modules are covered by the module theorem. □

4.5 Comparison with earlier approaches

Our module system resembles the module system proposed by Gaifman and Shapiro (1989). However, to make our system compatible with the stable model semantics we need to introduce a further restriction of mutual dependence, that is, we need to deny positive recursion between modules. Also other propositions involve similar conditions for module composition. For example, Brogi *et al.* (1994) employ visibility conditions that correspond to respecting hidden atoms. However, their approach covers only positive programs under the least model semantics. Maher (1993) forbids all recursion between modules and considers Przymusinski's *perfect models* (Przymusinski 1988) rather than stable models. Etalle and Gabbrielli (1996) restrict the composition of *constraint logic program* (Jaffar and Maher 1994) modules with a condition that is close to ours: $\mathrm{At}(P) \cap \mathrm{At}(Q) \subseteq \mathrm{At_v}(P) \cap \mathrm{At_v}(Q)$ but no distinction between input and output is made, for example, $\mathrm{At_o}(P) \cap \mathrm{At_o}(Q) \neq \emptyset$ is allowed according to their definitions.

Approaches to modularity within ASP typically do not allow any recursion (negative or positive) between modules (Lifschitz and Turner 1994; Eiter *et al.* 1997a; Gelfond and Gabaldon 1999; Tari *et al.* 2005). Theorem 4.14, the module theorem, is strictly stronger than the splitting-set theorem (Lifschitz and Turner 1994) for NLPs, and the general case allows us to generalize the splitting-set theorem for SMODELS programs. First consider the case of NLPs. A *splitting* of a program can be used as a basis for a module structure. If U is a splitting set for an NLP P, then we can define

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

It follows directly from Theorems 3.6 and 4.14 that $M_1 \in \mathrm{SM}(\mathbb{B})$ and $M_2 \in \mathrm{SM}(\mathbb{T})$ are compatible if and only if $\langle M_1, M_2 \setminus U \rangle$ is a solution for P with respect to U.

Example 4.31
Consider an NLP $P = \{a \leftarrow \sim b.\ b \leftarrow \sim a.\ c \leftarrow a.\}$. The set $U = \{a, b\}$ is a splitting set for P, and therefore the splitting-set theorem (Theorem 3.6) can be applied: $\mathrm{b}_U(P) = \{a \leftarrow \sim b.\ b \leftarrow \sim a.\}$ and $\mathrm{t}_U(P) = \{c \leftarrow a.\}$. Now $M_1 = \{a\}$ and $M_2 = \{b\}$ are the stable models of $\mathrm{b}_U(P)$, and we can evaluate the top with respect to M_1 and M_2, resulting in solutions $\langle M_1, \{c\} \rangle$ and $\langle M_2, \emptyset \rangle$, respectively. On the other hand, P can be seen as join of modules $\mathbb{P}_1 = \langle \mathrm{b}_U(P), \emptyset, U, \emptyset \rangle$ and $\mathbb{P}_2 = \langle \mathrm{t}_U(P), U, \{c\}, \emptyset \rangle$. Now, we have $\mathrm{SM}(\mathbb{P}_1) = \{M_1, M_2\}$ and $\mathrm{SM}(\mathbb{P}_2) = \{\emptyset, \{b\}, \{a, c\}, \{a, b, c\}\}$. Out of eight possible combinations only M_1 and $\{a, c\}$ are compatible, and M_2 and $\{b\}$ are compatible.

However, it is possible to apply Theorem 4.14 in a manner similar to the splitting-set theorem, that is, we only need to compute the stable models of $\mathbb{P}_2$ compatible with the stable models of $\mathbb{P}_1$. Notice that when the splitting-set theorem is applicable,

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

the stable models of $\mathbb{P}_1$ fully define the possible input interpretations for $\mathbb{P}_2$. This leaves us with stable models $\{a, c\}$ and $\{b\}$ for the composition. □

On the other hand, consider the module $\mathbb{P}_1 = \langle \mathrm{b}_U(P), \emptyset, U, \emptyset \rangle$ in Example 4.31. There are no nontrivial splitting sets for the bottom program $\mathrm{b}_U(P) = \{a \leftarrow \sim b.\ b \leftarrow \sim a.\}$. However, $\mathbb{P}_1$ can be viewed as the join of two NLP modules $\mathbb{Q}_1 = \langle \{a \leftarrow \sim b.\}, \{b\}, \{a\}, \emptyset \rangle$, and $\mathbb{Q}_2 = \langle \{b \leftarrow \sim a.\}, \{a\}, \{b\}, \emptyset \rangle$ to which the module theorem is applicable.

In the general case of SMODELS program modules we can use the module theorem to generalize the splitting-set theorem for SMODELS programs. Then the bottom module acts as an input generator for the top module, and one can simply find the stable models for the top module instantiated with the stable models of the bottom module. The latter strategy used in Example 4.31 works even if there is negative recursion between the modules, as already shown in Example 4.18.

The module theorem strengthens an earlier version given in (Janhunen 2006) to cover programs that involve positive body literals, too. The independent sets proposed by Faber *et al.* (2005) push negative recursion inside modules which is unnecessary in view of our results. Their version of the module theorem is also weaker than that of Theorem 4.14.

The approach to modularity based on lp-functions (Gelfond and Gabaldon 1999; Baral 2003) has features similar to our approach. The components presented by lp-functions have an input/output interface and a *domain* reflecting the possible input interpretations. The functional specification requires an lp-function to have a *consistent answer set* for any interpretation in its domain. This is something that is not required in our module system. Lp-functions are flexible in the sense that there are several operators for refining them. However, the composition operator for lp-functions allows only incremental compositions, which again basically reflects the splitting-set theorem.

5 More on program (de)composition

So far we have established a module architecture for the class SMODELS programs, in which modules interact through an input/output interface and the stable model semantics is fully compatible with the architecture. In this section, we investigate further the ways to understand the internal structure of logic programs by seeing them as compositions of logic program modules. First, we use the conditions for module composition to introduce a method for decomposing an SMODELS program into modules. A more detailed knowledge of the internal structure of a program (or a module) might reveal ways to improve search for stable models. Another application can be found in modularization of the translation-based equivalence verification method given in Oikarinen and Janhunen (2008). Second, we consider possibilities of relaxing the conditions for module composition, that is, whether it is possible to allow positive recursion between modules in certain cases.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

5.1 Finding a program decomposition

Recall that any SMODELS program P can be viewed as a module $\langle P, \emptyset, \mathrm{At}(P), \emptyset\rangle$, and thus we consider here a more general case of finding a module decomposition for an arbitrary SMODELS program module $\mathbb{P} = \langle R, I, O, H\rangle$. The first step is to exploit the strongly connected components $D_1, \ldots, D_n$ of $\mathrm{Dep}^+(\mathbb{P})$ and define submodules $\mathbb{P}_i$ by grouping the rules so that for each D_i all the rules $r \in R$ such that $\mathrm{Head}(r) \subseteq D_i$ are put into one submodule.

Now, the question is whether $\mathbb{P}_i$'s defined this way would form a valid decomposition of $\mathbb{P}$ into submodules. First notice that input atoms form a special case because $\mathrm{Head}(R) \cap I = \emptyset$. Each $a \in I$ ends up in its own strongly connected component and there are no rules to include into a submodule corresponding to strongly connected component $\{a\}$. Thus, it is actually unnecessary to include a submodule based on such a component. Obviously, each weight rule in R goes into exactly one of the submodules. One should notice that for a choice rule $r \in R$ it can happen that $\mathrm{Head}(r) \cap D_i \neq \emptyset$ and $\mathrm{Head}(r) \cap D_j \neq \emptyset$ for $i \neq j$. This is not a problem, since it is always possible to *split a choice rule* by projecting the head, that is, by replacing a choice rule of the form $\{A\} \leftarrow B, \sim C$ with choice rules $\{A \cap D_i\} \leftarrow B, \sim C$ for each SCC D_i such that $A \cap D_i \neq \emptyset$.[6]

On the basis of the earlier discussion, we define the *set of rules defining a set of atoms* for an SMODELS program module.

Definition 5.1
Given an SMODELS program module $\mathbb{P} = \langle R, I, O, H\rangle$ and a set of atoms $D \subseteq \mathrm{At}(\mathbb{P})\backslash I$, the set of rules defining D, denoted by $R[D]$, contains the following rules:

- a choice rule $\{A \cap D\} \leftarrow B, \sim C$ if and only if there is a choice rule $\{A\} \leftarrow B, \sim C$ in R such that $A \cap D \neq \emptyset$; and
- a weight rule $a \leftarrow w \leqslant \{B = W_B, \sim C = W_C\}$ if and only if there is a weight rule $a \leftarrow w \leqslant \{B = W_B, \sim C = W_C\}$ in R such that $a \in D$.

We continue by defining a submodule of $\mathbb{P} = \langle R, I, O, H\rangle$ induced by a set of atoms $D \subseteq \mathrm{At}(\mathbb{P}) \setminus I$. We use Definition 5.1 for the set of rules, and choose $D \cap O$ to be the output signature and the rest of the visible atoms appearing in $R[D]$ to be the input signature.

Definition 5.2
Given an SMODELS program module $\mathbb{P} = \langle R, I, O, H\rangle$ and a set of atoms $D \subseteq \mathrm{At}(\mathbb{P})\backslash I$, a submodule induced by D is

$$\mathbb{P}[D] = (R[D], (\mathrm{At}(R[D]) \setminus D) \cap (I \cup O), D \cap O, D \cap H).$$

Let $D_1, \ldots, D_m$ be the strongly connected components of $\mathrm{Dep}^+(\mathbb{P})$ such that $D_i \cap I = \emptyset$. Now we can define $\mathbb{P}_i = \mathbb{P}[D_i]$ for each $1 \leqslant i \leqslant m$. Since the strongly connected components of $\mathrm{Dep}^+(\mathbb{P})$ are used as a basis, it is guaranteed that there is no positive recursion between any of the submodules $\mathbb{P}_i$. Also, it is clear that the output

[6] Note that in the case of disjunctive logic programs, splitting a rule into two modules is more involved (see Janhunen *et al.* 2007) for a discussion on a general shifting principle.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

signatures of the submodules are pairwise disjoint. Unfortunately, this construction does not yet guarantee that hidden atoms stay local, and therefore the composition $\mathbb{P}_1 \oplus \cdots \oplus \mathbb{P}_m$ might not be defined because certain $\mathbb{P}_i$'s might not respect each others hidden atoms.

A solution is to combine D_i's in a way that modules will be closed with respect to dependencies caused by the hidden atoms, that is, if a hidden atom h belongs to a component D_i then also all the atoms in the heads of rules in which h or $\sim h$ appears, have to belong to D_i. This can be achieved by finding the strongly connected components, denoted by $E_1, \ldots, E_k$, for $\mathrm{Dep}^{\mathrm{h}}(\mathbb{P}, \{D_1, \ldots, D_m\})$, where $\mathrm{Dep}^{\mathrm{h}}(\mathbb{P}, \{D_1, \ldots, D_m\})$ has $\{D_1, \ldots, D_m\}$ as the set of vertices, and

$$\{\langle D_i, D_j\rangle, \langle D_j, D_i\rangle \mid a \in D_i, b \in D_j, r \in R, \\ b \in \mathrm{Head}(r) \text{ and } a \in \mathrm{Body}(r) \cap \mathrm{At_h}(\mathbb{P})\}$$

as the set of edges. Now, we take the sets

$$F_i = \bigcup_{D \in E_i} D$$

for $1 \leqslant i \leqslant k$ and use them to induce a module structure for $\mathbb{P}$ by defining $\mathbb{P}_i = \mathbb{P}[F_i]$ for $1 \leqslant i \leqslant k$.

As there may be atoms in $\mathrm{At}(\mathbb{P})$ not appearing in the rules of $\mathbb{P}$, that is, $\mathrm{At}(\mathbb{P}) = \mathrm{At}(R)$ does not necessarily hold for $\mathbb{P} = \langle R, I, O, H\rangle$, it is possible that

$$\mathrm{At}(\mathbb{P}) \setminus (\mathrm{At}(\mathbb{P}_1) \cup \cdots \cup \mathrm{At}(\mathbb{P}_k)) \neq \emptyset.$$

To keep track of such atoms in $I \setminus \mathrm{At}(R)$ we need an additional module defined as

$$\mathbb{P}_0 = \langle \emptyset, I \setminus \mathrm{At}(R), \emptyset, \emptyset\rangle.$$

There is no need for a similar treatment for atoms in $(O \cup H) \setminus \mathrm{At}(R)$ as each atom in $O \cup H$ belongs to some $\mathrm{At}(\mathbb{P}_i)$ by definition.

Theorem 5.3 shows that we have a valid decomposition of $\mathbb{P}$ into submodules.

Theorem 5.3

Consider an SMODELS program module $\mathbb{P}$, and let $D_1, \ldots, D_m$ be the SCCs of $\mathrm{Dep}^{+}(\mathbb{P})$ such that $D_i \cap I = \emptyset$, and $E_1, \ldots, E_k$ the strongly connected components of $\mathrm{Dep}^{\mathrm{h}}(\mathbb{P}, \{D_1, \ldots, D_m\})$. Define $\mathbb{P}_0 = \langle \emptyset, I \setminus \mathrm{At}(R), \emptyset, \emptyset\rangle$, and $\mathbb{P}_i = \mathbb{P}[F_i]$ for $F_i = \bigcup_{D \in E_i} D$ and $1 \leqslant i \leqslant k$. Then the join of the submodules $\mathbb{P}_i$ for $0 \leqslant i \leqslant k$ is defined and $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{P}_0 \sqcup \cdots \sqcup \mathbb{P}_k$.

Proof

Based on the construction of F_i's and the discussion in this section it is clear that $\mathbb{P}' = \mathbb{P}_0 \sqcup \cdots \sqcup \mathbb{P}_k$ is defined. It is easy to verify that the sets of input, output, and hidden atoms of modules $\mathbb{P}'$ and $\mathbb{P}$ are exactly the same. The only difference between the sets of rules in $\mathbb{P}$ and $\mathbb{P}'$ is that some choice rules in $\mathbb{P}$ may have been split into several rules in $\mathbb{P}'$. This is a syntactical change not affecting the stable models of the modules, that is, $\mathrm{SM}(\mathbb{P}) = \mathrm{SM}(\mathbb{P}')$. Notice also that $\mathrm{Dep}^{+}(\mathbb{P}) = \mathrm{Dep}^{+}(\mathbb{P}')$. Thus it holds that $\mathbb{P}' \equiv_{\mathrm{m}} \mathbb{P}$. □

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

5.2 Semantical conditions for module composition

Even though Example 4.9 shows that conditions for $\oplus$ are not enough to guarantee that the module theorem holds, there are cases where $\mathbb{P} \sqcup \mathbb{Q}$ is not defined and still it holds that $\mathrm{SM}(\mathbb{P} \oplus \mathbb{Q}) = \mathrm{SM}(\mathbb{P}) \bowtie \mathrm{SM}(\mathbb{Q})$.

Example 5.4
Consider modules $\mathbb{P} = \langle \{a \leftarrow b.\ a \leftarrow {\sim}c.\}, \{b\}, \{a, c\}, \emptyset \rangle$ and $\mathbb{Q} = \langle \{b \leftarrow a.\}, \{a\}, \{b\}, \emptyset \rangle$. Now, the composition

$$\mathbb{P} \oplus \mathbb{Q} = \langle \{a \leftarrow b.\ a \leftarrow {\sim}c.\ b \leftarrow a.\}, \emptyset, \{a, b, c\}, \emptyset \rangle$$

is defined as the output sets are disjoint and there are no hidden atoms. Since $\mathrm{SM}(\mathbb{P}) = \{\{a\}, \{a, b\}\}$ and $\mathrm{SM}(\mathbb{Q}) = \{\emptyset, \{a, b\}\}$, we get $\mathrm{SM}(\mathbb{P} \oplus \mathbb{Q}) = \{\{a, b\}\} = \mathrm{SM}(\mathbb{P}) \bowtie \mathrm{SM}(\mathbb{Q})$. □

Example 5.4 suggests that the denial of positive recursion between modules can be relaxed in certain cases. We define a semantical characterization for module composition that maintains the compositionality of the stable model semantics.

Definition 5.5
The *semantical join* $\mathbb{P}_1 \sqcup \mathbb{P}_2$ of two SMODELS program modules $\mathbb{P}_1$ and $\mathbb{P}_2$ is $\mathbb{P}_1 \oplus \mathbb{P}_2$, provided $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined and $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$.

The module theorem holds by definition for SMODELS program modules composed with $\sqcup$. We can now present an alternative formulation for modular equivalence taking features from strong equivalence (Lifschitz *et al.* 2001).

Definition 5.6
SMODELS program modules $\mathbb{P}$ and $\mathbb{Q}$ are *semantically modularly equivalent*, denoted by $\mathbb{P} \equiv_{\mathrm{sem}} \mathbb{Q}$, if and only if $\mathrm{At_i}(\mathbb{P}) = \mathrm{At_i}(\mathbb{Q})$ and $\mathbb{P} \sqcup \mathbb{R} \equiv_{\mathrm{v}} \mathbb{Q} \sqcup \mathbb{R}$ for all $\mathbb{R}$ such that $\mathbb{P} \sqcup \mathbb{R}$ and $\mathbb{Q} \sqcup \mathbb{R}$ are defined.

It is straightforward to see that $\equiv_{\mathrm{sem}}$ is a congruence for $\sqcup$ and reduces to $\equiv_{\mathrm{v}}$ for modules with completely specified input, that is, module $\mathbb{P}$ such that $\mathrm{At_i}(\mathbb{P}) = \emptyset$.

Theorem 5.7
$\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ if and only if $\mathbb{P} \equiv_{\mathrm{sem}} \mathbb{Q}$ for any SMODELS program modules $\mathbb{P}$ and $\mathbb{Q}$.

Proof
Assume $\mathbb{P} \equiv_{\mathrm{sem}} \mathbb{Q}$. Now, $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$ is implied by Definition 5.6 with empty context module $\mathbb{R} = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$. Assume then $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$, that is, there is a bijection $f : \mathrm{SM}(\mathbb{P}) \rightarrow \mathrm{SM}(\mathbb{Q})$ such that for each $M \in \mathrm{SM}(\mathbb{P})$, $M \cap \mathrm{At_v}(\mathbb{P}) = f(M) \cap \mathrm{At_v}(\mathbb{Q})$. Consider arbitrary $\mathbb{R}$ such that $\mathbb{P} \sqcup \mathbb{R}$ and $\mathbb{Q} \sqcup \mathbb{R}$ are defined. Then $\mathrm{SM}(\mathbb{P} \sqcup \mathbb{R}) = \mathrm{SM}(\mathbb{P}) \bowtie \mathrm{SM}(\mathbb{R})$ and $\mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R}) = \mathrm{SM}(\mathbb{Q}) \bowtie \mathrm{SM}(\mathbb{R})$.

We now define $g : \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R}) \rightarrow \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R})$ such that for any $M \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$,

$$g(M) = f(M_P) \cup M_R,$$

where $M = M_P \cup M_R$ such that $M_P \in \mathrm{SM}(\mathbb{P})$ and $M_R \in \mathrm{SM}(\mathbb{R})$ are compatible. Now, g is a bijection and $M \cap (\mathrm{At_v}(\mathbb{P}) \cup \mathrm{At_v}(\mathbb{R})) = g(M) \cap (\mathrm{At_v}(\mathbb{Q}) \cup \mathrm{At_v}(\mathbb{R}))$ for each $M \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$. Since $\mathbb{R}$ was arbitrary, $\mathbb{P} \equiv_{\mathrm{sem}} \mathbb{Q}$ follows. □

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Theorem 5.7 implies that $\equiv_m$ is a congruence for $\sqcup$, too. Thus, it is possible to replace $\mathbb{P}$ with modularly equivalent $\mathbb{Q}$ in the contexts allowed by $\sqcup$.

The syntactical restriction denying positive recursion between modules is easy to check, since SCCs can be found in a linear time with respect to the size of the dependency graph (Tarjan 1972). To the contrary, checking whether $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$ is a computationally harder problem.

Theorem 5.8
Given SMODELS program modules $\mathbb{P}_1$ and $\mathbb{P}_2$ such that $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined, deciding whether it holds that $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$ is a **coNP**-complete decision problem.

Proof
Let $\mathbb{P}_1$ and $\mathbb{P}_2$ be SMODELS program modules such that $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined. We can show $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) \neq \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$ by choosing $M \subseteq \mathrm{At}(\mathbb{P}_1 \oplus \mathbb{P}_2)$ and checking that

- $M \in \mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2)$ and $M \cap \mathrm{At}(\mathbb{P}_1) \notin \mathrm{SM}(\mathbb{P}_1)$; or
- $M \in \mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2)$ and $M \cap \mathrm{At}(\mathbb{P}_2) \notin \mathrm{SM}(\mathbb{P}_2)$; or
- $M \notin \mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2)$, $M \cap \mathrm{At}(\mathbb{P}_1) \in \mathrm{SM}(\mathbb{P}_1)$, and $M \cap \mathrm{At}(\mathbb{P}_2) \in \mathrm{SM}(\mathbb{P}_2)$.

Once we have chosen M, these tests can be performed in polynomial time, which shows that the problem is in **coNP**. To establish **coNP**-hardness we present a reduction from $\overline{\mathbf{3SAT}}$. Consider a finite set $S = \{C_1, \ldots, C_n\}$ of three-literal clauses C_i of the form $l_1 \vee l_2 \vee l_3$ where each l_i is either an atom a or its classical negation $\neg a$. Each clause C_i is translated into rules $r_{i,j}$ of the form $c_i \leftarrow f_j$, where $1 \leqslant j \leqslant 3$, and $f_j = a$ if $l_j = a$ and $f_j = {\sim}a$ if $l_j = \neg a$. The intuitive reading of c_i is that clause C_i is satisfied. We define modules $\mathbb{P}_1 = \langle \{e \leftarrow d.\}, \{d\}, \{e\}, \emptyset \rangle$ and

$$\mathbb{P}_2 = \langle \{r_{i,j} \mid 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant 3\} \cup \{d \leftarrow e, c_1, \ldots, c_n.\}, \\ \mathrm{At}(S) \cup \{e\}, \{d\}, \{c_1, \ldots, c_n\} \rangle.$$

Now $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined, and $\mathrm{SM}(\mathbb{P}_1) = \{\emptyset, \{d, e\}\}$. There is $M \in \mathrm{SM}(\mathbb{P}_2)$ that is compatible with $\{d, e\}$ if and only if $S \in \mathbf{3SAT}$. Since $d \notin N$ and $e \notin N$ for all $N \in \mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2)$, it follows that $S \in \overline{\mathbf{3SAT}}$ if and only if $\mathrm{SM}(\mathbb{P}_1 \oplus \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$. □

Theorem 5.8 shows that there is a tradeoff for allowing positive recursion between modules, as more effort is needed to check that composition of such modules does not compromise the compositionality of the stable model semantics.

6 Tools and practical demonstration

The goal of this section is to demonstrate how the module system introduced in Section 4 can be exploited in practise in the context of the SMODELS system and other compatible systems. In this respect, we present tools that have been developed for the (de)composition of logic programs that are represented in the *internal file*

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

format[7] of the SMODELS engine. The binaries for both tools are available under the ASPTOOLS collection.[8] Moreover, we conduct and report a practical experiment which illustrates the performance of the tools when processing substantially large benchmark instances, that is, SMODELS programs having up to millions of rules (see the ASPTOOLS web page for examples).

The first tool, namely MODLIST, is targeted at program decomposition based on the strongly connected components of an SMODELS program given as input. In view of the objectives of Section 5.1, there are three optional outcomes of the decomposition, that is, strongly connected components that take into account

(1) positive dependencies only,
(2) positive dependencies and hidden atoms, and
(3) both positive and negative dependencies as well as hidden atoms.

The number of modules created by MODLIST decreases in this order. However, our benchmarks cover program instances that get split in tens of thousands of modules. To tackle the problem of storing such numbers of modules in separate files we decided to use file compression and packaging tools and, in particular, the ZIP utility available in standard Linux installations. We found ZIP superior to TAR as it allows random access to files in an archive, or a *zip file*. This feature becomes valuable when the modules are accessed from the archive for further processing.

The tool for program composition has been named as LPCAT which refers to the concatenation of files containing logic programs. A new version of the tool was implemented for experiments reported later for better performance as well as usability. The old version (version 1.8) is only able to combine two modules at a time which gives a quadratic nature for a process of combining n modules together: modules are added one-by-one to the composition. The new version, however, is able to read in modules from several files and, even more conveniently, a *stream of modules* from an individual file. The ZIP facility provides an option for creating such a stream that can then be forwarded for LPCAT for composition. This is the strategy for composing programs in experiments that are described next.

To test the performance of our tools, we picked a set of benchmark instances having from tens of thousands up to millions of rules—expressed in the SMODELS format. For each instance, the first task is to decompose the instance into modules using MODLIST and to create a zip file containing the modules. The type of modules to be created is varied according to the three schemes summarized earlier. The second task is to recreate the benchmark instance from a stream of modules extracted from the respective zip file. As suggested earlier, the actual composition is carried out using LPCAT and we also check that the number of rules matches with the original instance. Due to high number of rules, checking the equivalence of the original and composed programs (Janhunen and Oikarinen 2007) is unfeasible in many cases. If all atoms are visible, this can be accomplished syntactically on the basis of sorted textual representations of the programs involved. To ensure that MODLIST and LPCAT

[7] The reader is referred to (Janhunen 2007) for a detailed description and analysis of the format.
[8] `http://www.tcs.hut.fi/Software/asptools/`

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Table 1. *Summary of benchmark results for module (de)composition*

Benchmark	**na**	**nr**	**bt**	**nm**	**dt** (s)	**ct** (s)
ephp-13	35,518	90,784	+	35,518	2,110	362
			+h	35,518	2,110	362
			±h	35,362	2,090	361
mutex3	276,086	2,406,357	+	101,819	22,900	9,570
			+h	101,819	23,300	9,640
			±h	101,609	24,000	9,580
phi3	7,379	14,274	+	6,217	74.3	3.32
			+h	6,217	74.3	3.35
			±h	5,686	63.2	2.92
seq4-ss4	6,873	1,197,182	+	3,425	121	60.0
			+h	1,403	89.4	31.9
			±h	107	20.2	7.58

Legends for abbreviations: **na**: Number of atoms; **nr**: Number of rules; **nm**: Number of modules; **bt**: Benchmark type; **dt**: Decomposition time; **ct**: Composition time

produce correct (de)compositions of programs, such a check was performed for all compositions created for the first three benchmarks which involve no hidden atoms. As regards computer hardware, we run MODLIST and LPCAT on a PC with a 1.8GHz Intel Core 2 Duo CPU and 2GBs of main memory—operating under the Linux 2.6.18 system. In experimental results collected in Table 1, we report the sum of user and system times that are measured with the `/usr/bin/time` command. There are three benchmark types (**bt** for short) as enumerated in the beginning of this section. We refer to them using the respective abbreviations +, +h, and ±h.

The first benchmark instance in Table 1, viz. *ephp-13*, is a formalization (Järvisalo and Oikarinen 2007) of the classical *pigeon hole principle* for 13 pigeons—extended by redundant rules in analogy to Tseitin's *extended resolution* proof system. This program can be deemed medium sized within our benchmarks. There are no hidden atoms, no positive recursion, and little negative recursion in this program instance as indicated by the number of atoms (35,518) and the respective numbers of modules (see the column **nm**). Thus, we have an example of a very fine-grained decomposition where the *definition*[9] of each atom ends up as its own module in the outcome. The given timings indicate that MODLIST and LPCAT are able to handle 15 and 100 modules/s, respectively. The share of file I/O and (de)compression is substantial in program decomposition. For instance, the actual splitting of the *ephp-13* benchmark (+h) using MODLIST takes only 0.59 s—the rest of approximately 2,110 s is spent to create the zip file. To the contrary, inflating the stream of modules from the zip file is very efficient as it takes only 0.45 s in case of *ephp-13*. After that the restoration of the original program instance takes roughly 361 s. The creation and compression of a joint symbol table for the modules accounts for the most of the time spent on this operation. It should also be stressed that it is impractical to store modules

[9] The set of rules that mention the atom in question in their head.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Table 2. *Distribution of the sizes of modules (see Table 1 for legends)*

Benchmark	ephp-13	mutex3	phi3	seq-ss4
nr		**nm**		
1	14,474	67,749	2,811	2,969
2	7,014	2,757	1,434	
3–4	12,680	41	1,962	
5–8	149	30,798	2	
9–16	618	255	6	
17–32	582			11
33–64			1	
65–128				134
129–512				296
513–1,024	1			9
over 1,024		2	1	2

in separate files for this program. For instance, a shell command that refers to all modules fails due to excessive number of arguments at the respective command line.

The next two programs in Table 1, *mutex3* and *phi3*, are related to the *distributed implementability problem* of asynchronous automata, and particular formalizations of classical *mutual exclusion* and *dining philosophers* problems (Heljanko and Ştefănescu 2004, 2005). These programs involve no hidden atoms and both positive and negative interdependencies of atoms occur. The extremely high numbers of rules (2,406,357) and modules (101,819) are clearly reflected in running times perceived for *mutex3*. However, the respective rates of 4 and 10 modules/s do not differ too much from those obtained for *ephp-13* given the fact that the number of rules is about 25 times higher. The data observed for *phi3* is analogous to those obtained for *ephp-13* and *mutex3* but the respective modules-per-second rates are much higher: approximately 90 and 2,000. This may partly boil down to the fact that *phi3* is the smallest program under consideration and it has also the smallest number of rules per module ratio.

Our last benchmark program, *seq4-ss4*, is taken from benchmark sets of Brain *et al.* (2006) where the optimization of machine code using ASP techniques is of interest. The program in question formalizes the optimization of a particular sequence of four SPARC-v7 instructions. This program instance has the greatest modules as regards the number of rules—the average number of rules per module varies from about 350–11,200 depending on the module type. It also has hidden atoms which make a difference between modules based on plain SCCs and their combinations induced by the dependencies caused by the use of hidden atoms. The respective modules-per-second rates 28, 19, and 5 are all better than the four obtained for *mutex3*.

To provide the reader with a better idea of sizes of individual modules, we have collected some numbers about their distribution in Table 2. Each program involves a substantial number of modules with just one rule each of which defines a single atom of interest. On the other hand, the largest SCCs for *ephp-13*, *mutex3*, *phi3*, and *seq4-ss4* involve 949; 2,091,912; 2,579; and 1,071,689 rules, respectively. For *mutex3*,

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

the biggest module consists of a definition of an equivalence relation over states in the verification domain—creating a huge set of positively interdependent atoms. For *ephp-13*, the greatest module is a collection of nogoods which can be shown to have no stable models in roughly 99,500 s using SMODELS (version 2.32). However, the remaining rules of *ephp-13* make this fact much faster to prove: only 61 s elapse.

A few concluding remarks follow. Increasing the number of modules in a program tends to decrease the number of modules that can be decomposed per time unit. This observation suggests that the creation of the zip file has a quadratic flavor although modules themselves can be figured out in linear time (using a variant of Tarjan's algorithm). Perhaps this can be improved in the future by better integrating the creation of the zip file into MODLIST. For now, it creates a shell script for this purpose. Handling the biggest program instances is also subject to the effects of memory allocation which may further slow down computations. On the other hand, the cost of increasing the number of rules in modules seems to be relatively small. Moreover, it is clear on the basis of data given in Table 1 that the composition of programs is faster than decomposition. This would not be the case if the old versions of LPCAT were used for composition. Last, we want to emphasize that MODLIST and LPCAT have been implemented as supplementary tools that are not directly related to the computation of stable models. Nevertheless, we intend to exploit these tools in order to modularize different tasks in ASP such as verifying ordinary/modular equivalence and program optimization. The existence of such tools enables modular program development and the creation of *module libraries* for SMODELS programs, and thus puts forward the use of module architectures in the realm of ASP.

7 Conclusions

In this paper, we introduce a simple and intuitive notion of a logic program module that interacts with other modules through a well-defined input/output interface. The design has its roots in a module architecture proposed for conventional logic programs (Gaifman and Shapiro 1989), but as regards our contribution, we tailor the architecture in order to better meet the criteria of ASP. Perhaps the most important objective in this respect is to achieve the compositionality of stable model semantics, that is, the semantics of an entire program depends directly on the semantics assigned to its modules. To this end, the main result of this paper is formalized as the *module theorem* (Theorem 4.14) which links program-level stability with module-level stability. The theorem holds under the assumption that positively interdependent atoms are always placed in the same module. The *join* operation $\sqcup$ defined for program modules effectively formalizes this constraint—which we find acceptable when it comes to good programming style in ASP.

The module theorem is also a proper generalization of the splitting-set theorem (Lifschitz and Turner 1994) recast for SMODELS programs. The main difference is that splitting sets do not enable any kind of recursion between modules. Even though the module theorem is proved to demonstrate the feasibility of the respective module architecture, it is also applied as a tool to simplify mathematical proofs in this paper and recently also in (Oikarinen and Janhunen 2008, in press). It also

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

lends itself to extensions for further classes of logic programs which can be brought into effect in terms of *strongly faithful*, $\sqcup$*-preserving*, and *modular* translations for the removal of new syntax (Theorem 4.28). Moreover, the module theorem paves the way for the modularization of various reasoning tasks, such as search for answer sets, query evaluation, and verification, in ASP.

The second main theme of the paper is the notion of modular equivalence which is proved to be a proper congruence relation for program composition using $\sqcup$ (Theorem 4.20). Thus modular equivalence is preserved under substitutions of modularly equivalent program modules. Since uniform equivalence is not a congruence for ordinary $\cup$ but strong equivalence is, by definition; modular equivalence can be viewed as a reasonable compromise between these two extremes. In addition to the congruence property, we present a number of results about modular equivalence.

(1) We show that deciding modular equivalence forms a **coNP**-complete decision problem for SMODELS program modules with the EVA property, that is, those having enough visible atoms so that their stable models can be distinguished from each other on the basis of visible atoms only. In this way, it is possible to use the SMODELS solver for the actual verification task.
(2) We consider the possibility of redefining the join operation $\sqcup$ using a semantical condition that corresponds to the content of the module theorem. The notion of modular equivalence is not affected, but the cost of verifying whether a particular join of modules is defined becomes a **coNP**-complete decision problem. This is in contrast with the linear time check for positive recursion (Tarjan's algorithm for strongly connected components) but it may favorably extend the coverage of modular equivalence in certain applications.
(3) Finally, we also analyze the problem of decomposing an SMODELS program into modules when there is no *a priori* knowledge about the structure of the program. The strongly connected components of the program provide the starting point in this respect, but the usage of hidden atoms may enforce a higher degree of amalgamation when the modules of a program are extracted.

The theoretical results presented in this paper have emerged in close connection with the development of tools for ASP. The practical demonstration in Section 6 illustrates the basic facilities that are required to deal with *object-level* modules within the SMODELS system.[10] The linker, namely LPCAT, enables the composition of ground programs in the SMODELS format. Using this tool, for instance, it is possible to add a query to a program afterwards without grounding the program again. On the other hand, individual modules of a program can be accessed from the zip file created by the module extractor MODLIST. This is highly practical since we intend to pursue techniques for module-level optimization in the future.

[10] Likewise, source-level modules could be incorporated to the front-end of the system (LPARSE).

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Acknowledgements

This work has been partially supported by the Academy of Finland through Projects #211025 and #122399. The first author gratefully acknowledges the financial support from Helsinki Graduate School in Computer Science and Engineering, Emil Aaltonen Foundation, the Finnish Foundation for Technology Promotion TES, the Nokia Foundation, and the Finnish Cultural Foundation.

Appendix Proofs

Proof of Theorem 4.20

Let $\mathbb{P}$ and $\mathbb{Q}$ be modules such that $\mathbb{P} \equiv_{\mathrm{m}} \mathbb{Q}$. Let $\mathbb{R}$ be an arbitrary module such that $\mathbb{P} \sqcup \mathbb{R}$ and $\mathbb{Q} \sqcup \mathbb{R}$ are defined. From $\mathrm{At_v}(\mathbb{P}) = \mathrm{At_v}(\mathbb{Q})$ and $\mathrm{At_i}(\mathbb{P}) = \mathrm{At_i}(\mathbb{Q})$ it follows that $\mathrm{At_v}(\mathbb{P} \sqcup \mathbb{R}) = \mathrm{At_v}(\mathbb{Q} \sqcup \mathbb{R})$ and $\mathrm{At_i}(\mathbb{P} \sqcup \mathbb{R}) = \mathrm{At_i}(\mathbb{Q} \sqcup \mathbb{R})$.

Consider any model $M \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$. By Theorem 4.14, $\mathrm{SM}(\mathbb{P} \sqcup \mathbb{R}) = \mathrm{SM}(\mathbb{P}) \bowtie \mathrm{SM}(\mathbb{R})$, that is, $M_P = M \cap \mathrm{At}(\mathbb{P}) \in \mathrm{SM}(\mathbb{P})$ and $M_R = M \cap \mathrm{At}(\mathbb{R}) \in \mathrm{SM}(\mathbb{R})$. 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

$$M_P \cap \mathrm{At_v}(\mathbb{P}) = f(M_P) \cap \mathrm{At_v}(\mathbb{Q}). \tag{A1}$$

Denote $M_Q = f(M_P)$. Clearly, M_P and M_R are compatible. Since (A1) holds, also M_Q and M_R are compatible. Applying Theorem 4.14 we get $M_Q \cup M_R \in \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R}) = \mathrm{SM}(\mathbb{Q}) \bowtie \mathrm{SM}(\mathbb{R})$.

Now, define a function $g : \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R}) \to \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R})$ as

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

Clearly, g maps the set of visible atoms in M to itself, that is,

$$M \cap (\mathrm{At_v}(\mathbb{P} \sqcup \mathbb{R})) = g(M) \cap (\mathrm{At_v}(\mathbb{Q} \sqcup \mathbb{R})).$$

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 $f(M \cap \mathrm{At}(\mathbb{P})) \neq f(N \cap \mathrm{At}(\mathbb{P}))$ or $M \cap \mathrm{At}(\mathbb{R}) \neq N \cap \mathrm{At}(\mathbb{R})$.
- g is a surjection: for any $M \in \mathrm{SM}(\mathbb{Q} \sqcup \mathbb{R})$, $N = f^{-1}(M \cap \mathrm{At}(\mathbb{Q})) \cup (M \cap \mathrm{At}(\mathbb{R})) \in \mathrm{SM}(\mathbb{P} \sqcup \mathbb{R})$ 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 $g^{-1}(N) = f^{-1}(N \cap \mathrm{At}(\mathbb{Q})) \cup (N \cap \mathrm{At}(\mathbb{R}))$. Thus $\mathbb{P} \sqcup \mathbb{R} \equiv_{\mathrm{m}} \mathbb{Q} \sqcup \mathbb{R}$. □

Proof of Theorem 4.26

We present an alternative proof to the one given in (Oikarinen and Janhunen 2006). We use the characterization of stable models based on the programs completion and loop formulas presented in Theorem 3.10.

First, we need to generalize the concepts of completion and loop formulas for NLP modules. Given a NLP module $\mathbb{P} = \langle R, I, O, H \rangle$, we define

$$\mathrm{Comp}(\mathbb{P}) = \bigwedge_{a \in O \cup H} \left(a \leftrightarrow \bigvee_{\mathrm{Head}(r) = a} \left(\bigwedge_{b \in \mathrm{Body}^+(r)} b \wedge \bigwedge_{c \in \mathrm{Body}^-(r)} \neg c \right) \right), \tag{A2}$$

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

that is, we take the completion in the normal way for the set of rules R with the exception that we take into account that input atoms do not have any defining rules. As regards loop formulas, we define $\mathrm{LF}(\mathbb{P}) = \mathrm{LF}(R)$, since no atom in the input signature can appear in any of the loops.

Consider an arbitrary NLP module $\mathbb{P} = \langle R, I, O, H\rangle$. Define a set of rules, that is, a conventional NLP, $R' = R \cup G_I$, where $G_I = \{a \leftarrow {\sim}\bar{a}.\ \bar{a} \leftarrow {\sim}a \mid a \in I\}$ and all atoms $\bar{a}$ are new atoms not appearing in $\mathrm{At}(\mathbb{P})$. Now, $M \in \mathrm{SM}(\mathbb{P})$ if and only if $N = M \cup \{\bar{a} \mid a \in I \setminus M\} \in \mathrm{SM}(R')$. On the other hand, by Theorem 3.10, $N \in \mathrm{SM}(R')$ if and only if $N \models \mathrm{Comp}(R') \cup \mathrm{LF}(R')$. Considering the completion, since $\mathrm{Head}(R) \cap I = \emptyset$, and atoms $\bar{a}$ are new, it holds that $\mathrm{Comp}(R') = \mathrm{Comp}(\mathbb{P}) \cup \mathrm{Comp}(G_I)$. As regards loop formulas, we notice that $\mathrm{Dep}^+(R')$ is $\mathrm{Dep}^+(R)$ together with vertices for atoms $\{\bar{a} \mid a \in I\}$, which have no edges in $\mathrm{Dep}^+(R')$. Therefore, $\mathrm{LF}(R') = \mathrm{LF}(R) = \mathrm{LF}(\mathbb{P})$.

Thus $N \models \mathrm{Comp}(R') \cup \mathrm{LF}(R')$ if and only if $N \models \mathrm{Comp}(\mathbb{P}) \cup \mathrm{Comp}(G_I) \cup \mathrm{LF}(\mathbb{P})$. Furthermore, based on the relationship between M and N, it holds that $N \models \mathrm{Comp}(\mathbb{P}) \cup \mathrm{Comp}(G_I) \cup \mathrm{LF}(\mathbb{P})$ if and only if $M \models \mathrm{Comp}(\mathbb{P}) \cup \mathrm{LF}(\mathbb{P})$. Thus, Theorem 3.10 generalizes directly for NLP modules: given an NLP module $\mathbb{P}$ and an interpretation $M \subseteq \mathrm{At}(\mathbb{P})$, it holds that $M \in \mathrm{SM}(\mathbb{P})$ if and only if $M \models \mathrm{Comp}(\mathbb{P}) \cup \mathrm{LF}(\mathbb{P})$.

Now, since the join operation does not allow positive recursion between two modules, and $(\mathrm{At_o}(\mathbb{P}_1) \cup \mathrm{At_h}(\mathbb{P}_1)) \cap (\mathrm{At_o}(\mathbb{P}_2) \cup \mathrm{At_h}(\mathbb{P}_2)) = \emptyset$, it holds that

$$\mathrm{Comp}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{Comp}(\mathbb{P}_1) \cup \mathrm{Comp}(\mathbb{P}_2), \text{ and} \tag{A3}$$

$$\mathrm{LF}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{LF}(\mathbb{P}_1) \cup \mathrm{LF}(\mathbb{P}_2). \tag{A4}$$

Furthermore, the satisfaction relation is compositional for $\cup$, that is, $M \models P \cup Q$ if and only if $M \cap \mathrm{At}(P) \models P$ and $M \cap \mathrm{At}(Q) \models Q$ for any propositional theories P and Q. Thus,

$$\begin{aligned}
M \in \mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2) &\iff M \models \mathrm{Comp}(\mathbb{P}_1 \sqcup \mathbb{P}_2) \cup \mathrm{LF}(\mathbb{P}_1 \sqcup \mathbb{P}_2) \\
&\iff M \models \mathrm{Comp}(\mathbb{P}_1) \cup \mathrm{Comp}(\mathbb{P}_2) \cup \mathrm{LF}(\mathbb{P}_1) \cup \mathrm{LF}(\mathbb{P}_2) \\
&\iff M \cap \mathrm{At}(\mathbb{P}_1) \models \mathrm{Comp}(\mathbb{P}_1) \cup \mathrm{LF}(\mathbb{P}_1) \text{ and} \\
&\qquad\ M \cap \mathrm{At}(\mathbb{P}_2) \models \mathrm{Comp}(\mathbb{P}_2) \cup \mathrm{LF}(\mathbb{P}_2) \\
&\iff M \cap \mathrm{At}(\mathbb{P}_1) \in \mathrm{SM}(\mathbb{P}_1) \text{ and } M \cap \mathrm{At}(\mathbb{P}_2) \in \mathrm{SM}(\mathbb{P}_2).
\end{aligned}$$

It follows that $\mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$. □

Proof of Theorem 4.28

Let $\mathscr{C}_1$ and $\mathscr{C}_2$ be two classes of logic program modules such that $\mathscr{C}_2 \subseteq \mathscr{C}_1$, and the module theorem holds for modules in $\mathscr{C}_2$. Consider a translation function $\mathrm{Tr} : \mathscr{C}_1 \to \mathscr{C}_2$ such that conditions one to three from Definition 4.27 are satisfied. Let $\mathbb{P}_1, \mathbb{P}_2 \in \mathscr{C}_1$ be modules such that $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined. Then condition 2 implies that $\mathrm{Tr}(\mathbb{P}_1), \mathrm{Tr}(\mathbb{P}_2) \in \mathscr{C}_2$ are modules such that $\mathrm{Tr}(\mathbb{P}_1) \sqcup \mathrm{Tr}(\mathbb{P}_2)$ is defined. Since the module theorem holds for modules in $\mathscr{C}_2$, $\mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1) \sqcup \mathrm{Tr}(\mathbb{P}_2)) = \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1)) \bowtie \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_2))$. Moreover, condition 3 implies that $\mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1 \sqcup \mathbb{P}_2)) = \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1) \sqcup \mathrm{Tr}(\mathbb{P}_2))$.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Condition 1 implies there is a bijection

$$g : \quad \mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2) \to \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1 \sqcup \mathbb{P}_2))$$

such that for any $M \in \mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$ we have $M = g(M) \cap \mathrm{At}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$. Notice that strong faithfulness requires that the projections of M and $g(M)$ have to be identical over whole $\mathrm{At}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$ not just over $\mathrm{At_v}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$. Similarly there are bijections

$$\begin{aligned} g_1 : &\quad \mathrm{SM}(\mathbb{P}_1) \to \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1)) \text{ and} \\ g_2 : &\quad \mathrm{SM}(\mathbb{P}_2) \to \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_2)) \end{aligned}$$

such that for any $M_i \in \mathrm{SM}(\mathbb{P}_i)$ $(i = 1, 2)$ it holds that $M_i = g_i(M_i) \cap \mathrm{At}(\mathbb{P}_i)$. Consider arbitrary $M \subseteq \mathrm{At}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$, and its projections $M_1 = M \cap \mathrm{At}(\mathbb{P}_1)$ and $M_2 = M \cap \mathrm{At}(\mathbb{P}_2)$. Now, M_1 and M_2 are compatible, and $M = M_1 \cup M_2$.

Assume that $M \in \mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$. Since the module theorem holds for $\mathscr{C}_2$, we have

$$g(M) \in \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1 \sqcup \mathbb{P}_2)) = \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1) \sqcup \mathrm{Tr}(\mathbb{P}_2)) = \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1)) \bowtie \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_2)),$$

that is,

$$\begin{aligned} N_1 &= g(M) \cap \mathrm{At}(\mathrm{Tr}(\mathbb{P}_1)) \in \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1)), \\ N_2 &= g(M) \cap \mathrm{At}(\mathrm{Tr}(\mathbb{P}_2)) \in \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_2)), \text{ and} \end{aligned}$$

N_1 and N_2 are compatible projections of $g(M)$. Moreover, $M = g(M) \cap \mathrm{At}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$ and $M_i = N_i \cap \mathrm{At}(\mathbb{P}_i)$ for $i = 1, 2$. Using the inverse functions of g_1 and g_2 we get $M_1 = g_1^{-1}(N_1) \in \mathrm{SM}(\mathbb{P}_1)$ and $M_2 = g_2^{-1}(N_2) \in \mathrm{SM}(\mathbb{P}_2)$.

For the other direction, assume that $M_1 \in \mathrm{SM}(\mathbb{P}_1)$ and $M_2 \in \mathrm{SM}(\mathbb{P}_2)$. Then $N_1 = g_1(M_1) \in \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1))$ and $N_2 = g_2(M_2) \in \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_2))$. Since

$$\begin{aligned} N_1 \cap \mathrm{At_v}(\mathrm{Tr}(\mathbb{P}_2)) &= M_1 \cap \mathrm{At_v}(\mathbb{P}_2), \\ N_2 \cap \mathrm{At_v}(\mathrm{Tr}(\mathbb{P}_1)) &= M_2 \cap \mathrm{At_v}(\mathbb{P}_1), \text{ and} \end{aligned}$$

M_1 and M_2 are compatible, also N_1 and N_2 are compatible. By applying the module theorem for $\mathscr{C}_2$, we get $N = N_1 \cup N_2 \in \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1) \sqcup \mathrm{Tr}(\mathbb{P}_2)) = \mathrm{SM}(\mathrm{Tr}(\mathbb{P}_1 \sqcup \mathbb{P}_2))$. Furthermore, $N \cap \mathrm{At}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = M_1 \cup M_2 = M$, and using the inverse of g we get $M = g^{-1}(N) \in \mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$.

Thus, we have shown $\mathrm{SM}(\mathbb{P}_1 \sqcup \mathbb{P}_2) = \mathrm{SM}(\mathbb{P}_1) \bowtie \mathrm{SM}(\mathbb{P}_2)$. □

Proof of Theorem 4.30

Consider SMODELS program modules $\mathbb{P}_1$ and $\mathbb{P}_2$. It is straightforward to see that $\mathrm{Tr_{NLP}}$ is $\sqcup$-preserving, that is, if $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined, then also $\mathrm{Tr_{NLP}}(\mathbb{P}_1) \sqcup \mathrm{Tr_{NLP}}(\mathbb{P}_2)$ is defined. The key observation is that for every edge in the dependency graph $\mathrm{Dep}^+(\mathrm{Tr_{NLP}}(\mathbb{P}_1) \sqcup \mathrm{Tr_{NLP}}(\mathbb{P}_2))$ there is also an edge in $\mathrm{Dep}^+(\mathbb{P}_1 \sqcup \mathbb{P}_2)$[11]. Since, $\mathbb{P}_1$ and $\mathbb{P}_2$ are mutually independent, also $\mathrm{Tr_{NLP}}(\mathbb{P}_1)$ and $\mathrm{Tr_{NLP}}(\mathbb{P}_2)$ are mutually independent. Furthermore, if $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined, then $\mathrm{Tr_{NLP}}(\mathbb{P}_1)$ and $\mathrm{Tr_{NLP}}(\mathbb{P}_2)$ respect each other's hidden atoms. This is because new atoms are introduced only

[11] It might be the case that $\mathrm{Dep}^+(\mathbb{P}_1 \sqcup \mathbb{P}_2)$ contains some edges that are not in $\mathrm{Dep}^+(\mathrm{Tr_{NLP}}(\mathbb{P}_1) \sqcup \mathrm{Tr_{NLP}}(\mathbb{P}_2))$. This happens when there is a weight rule the body of which can never be satisfied.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

for $\mathrm{Choices}(\mathbb{P}_1) \subseteq \mathrm{At_o}(\mathbb{P}_1) \cup \mathrm{At_h}(\mathbb{P}_1)$ and $\mathrm{Choices}(\mathbb{P}_2) \subseteq \mathrm{At_o}(\mathbb{P}_2) \cup \mathrm{At_h}(\mathbb{P}_2)$, and $(\mathrm{At_o}(\mathbb{P}_1) \cup \mathrm{At_h}(\mathbb{P}_1)) \cap (\mathrm{At_o}(\mathbb{P}_2) \cup \mathrm{At_h}(\mathbb{P}_2)) = \emptyset$. Since

$$\mathrm{Tr_{NLP}}(\mathbb{P}_1) \sqcup \mathrm{Tr_{NLP}}(\mathbb{P}_2) = \mathrm{Tr_{NLP}}(\mathbb{P}_1 \sqcup \mathbb{P}_2)$$

holds, $\mathrm{Tr_{NLP}}$ is also modular. Thus, $\mathrm{Tr_{NLP}}$ satisfies conditions 2 and 3 in Definition 4.27.

We are left to show that

$$\mathrm{reveal}(\mathbb{P}, H) \equiv_{\mathrm{m}} \mathrm{reveal}(\mathrm{Tr_{NLP}}(\mathbb{P}), H),$$

for any SMODELS program module $\mathbb{P} = \langle P, I, O, H \rangle$ and its translation $\mathrm{Tr_{NLP}}(\mathbb{P}) = \langle R, I, O, H \cup H' \rangle$. Note that $\mathrm{SM}(\mathbb{P}) = \mathrm{SM}(\mathrm{reveal}(\mathbb{P}, H))$ and $\mathrm{SM}(\mathrm{Tr_{NLP}}(\mathbb{P})) = \mathrm{SM}(\mathrm{reveal}(\mathrm{Tr_{NLP}}(\mathbb{P}), H))$, and the additional restriction imposed by revealing is that the bijection between these sets of stable models needs to be such that their projections over $\mathrm{At}(\mathbb{P})$, not just over $\mathrm{At_v}(\mathbb{P})$, coincide.

We define a function $f : \mathrm{SM}(\mathbb{P}) \to 2^{\mathrm{At}(\mathrm{Tr_{NLP}}(\mathbb{P}))}$ such that

$$f(M) = M \cup \{\overline{a} \mid a \in \mathrm{Choices}(P) \setminus M\}.$$

Clearly, $M = f(M) \cap \mathrm{At}(\mathbb{P})$. We need to show that

(i) given any $M \in \mathrm{SM}(\mathbb{P})$, $f(M) \in \mathrm{SM}(\mathrm{Tr_{NLP}}(\mathbb{P}))$; and
(ii) $f : \mathrm{SM}(\mathbb{P}) \to \mathrm{SM}(\mathrm{Tr_{NLP}}(\mathbb{P}))$ is a bijection.

Note that for any atom $a \in \mathrm{Choices}(P)$, it holds that $\overline{a} \in f(M)$ if and only if $a \notin M$ if and only if $a \notin f(M)$.

(i) *We show that* $N = f(M) \in \mathrm{SM}(\mathrm{Tr_{NLP}}(\mathbb{P}))$ *for any* $M \in \mathrm{SM}(\mathbb{P})$*:*
Assume first $N \not\models R^{N,I}$, that is, there is a rule r in $R^{N,I}$ that is not satisfied by N. If $\mathrm{Head}(r) \in \mathrm{At}(\mathbb{P})$, then r is of the form $a \leftarrow B' \setminus I$ and there are two possibilities:

(1) There is a rule $a \leftarrow B', {\sim}C, {\sim}\overline{a} \in R$ corresponding to a choice rule $\{A\} \leftarrow B', {\sim}C \in P$ such that $a \in A$, $\overline{a} \notin N$, $B' \cap I \subseteq N$, and $C \cap N = \emptyset$. Since $\overline{a} \notin N$ implies $a \in N$, r is satisfied in N, a contradiction.
(2) There is a rule $a \leftarrow B', {\sim}C' \in R$ corresponding to a weight rule $r' = a \leftarrow w \leqslant \{B = W_B, {\sim}C = W_C\} \in P$ such that $B' \subseteq B$, $C' \subseteq C$, $w \leqslant \sum_{b \in B'} w_b + \sum_{c \in C'} w_c$, $B' \cap I \subseteq N$, and $C' \cap N = \emptyset$. Since $N \not\models r$ we must have $a \notin N$ (which implies $a \notin M$) and $B' \setminus I \subseteq N$. Thus $B' \subseteq N$ which implies $B' \subseteq M$. Moreover, $C' \cap N = \emptyset$ implies $C' \cap M = \emptyset$. But then $w \leqslant \sum_{b \in B' \cap M} w_b + \sum_{c \in C' \setminus M} w_c$. Since $B' \subseteq B$ and $C' \subseteq C$, we have $w \leqslant \sum_{b \in B \cap M} w_b + \sum_{c \in C \setminus M} w_c$ and $M \not\models r'$, a contradiction.

Otherwise each $r \in R^{N,I}$ is of the form $\overline{a}$, in which case there is a rule $\overline{a} \leftarrow {\sim}a \in R$ and $a \notin N$. Since $a \notin N$ implies $\overline{a} \in N$, then $N \models r$, a contradiction. Thus $N \models R^{N,I}$, and furthermore $N \setminus I \models R^{N,I}$.

Assume now $N \setminus I \neq \mathrm{LM}(R^{N,I})$, that is, there is $N' \subset N \setminus I$ such that $N' \models R^{N,I}$. We define $M' = N' \cap \mathrm{At}(\mathbb{P})$ and show $M' \models P^{M,I}$, which contradicts the assumption $M \setminus I = \mathrm{LM}(P^{M,I})$, since $M' \subset M \setminus I$. Assume that there is a rule $r \in P^{M,I}$ such that $M' \not\models r$. There are two possibilities:

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

(1) r is of the form $a \leftarrow B \setminus I$. Then there is a choice rule $\{A\} \leftarrow B, {\sim}C \in P$, such that $B \cap I \subseteq M$, $C \cap M = \emptyset$, and $a \in M \cap A$. Now, $B \cap I \subseteq M$ implies $B \cap I \subseteq N$, $C \cap M = \emptyset$ implies $C \cap N = \emptyset$, and $a \in M$ implies $a \in N$ and $\overline{a} \notin N$. Together with $\{A\} \leftarrow B, {\sim}C \in P$ these imply $r \in R^{N,I}$. Since $M' \not\models r$, we have $a \notin M'$ and $B \setminus I \subseteq M'$. But, since $N' \cap \mathrm{At}(\mathbb{P}) = M'$, this implies $N' \not\models r$, a contradiction to $N' \models R^{N,I}$.

(2) r is of the form $a \leftarrow w' \leqslant \{B \setminus I = W_{B \setminus I}\}$. Then there is a weight rule $a \leftarrow w \leqslant \{B = W_B, {\sim}C = W_C\} \in P$ such that

$$w' = \max(0, w - \sum_{b \in B \cap I \cap M} w_b - \sum_{c \in C \setminus M} w_c).$$

Since $M' \not\models r$, we have $a \notin M'$ and $w' \leqslant \sum_{b \in (B \setminus I) \cap M'} w_b$. Define $B' = (B \cap I \cap M) \cup ((B \setminus I) \cap M')$ and $C' = C \setminus M$, and recall that $N' \cap \mathrm{At}(\mathbb{P}) = M'$. Now $w \leqslant \sum_{b \in B'} w_b + \sum_{c \in C'} w_c$, $C' \cap N = \emptyset$, $B' \cap I \subseteq N$, $B' \setminus I \subseteq N'$ and $a \notin N'$, which implies that there is a rule $r' = a \leftarrow B' \setminus I \in R^{N,I}$ such that $N' \not\models r'$ which is in contradiction with $N' \models R^{N,I}$.

Thus assuming that there is $N' \subset N \setminus I$ such that $N' \models R^{N,I}$ leads to a contradiction, and it holds that $N \setminus I = \mathrm{LM}(R^{N,I})$, that is, $N \in \mathrm{SM}(\mathrm{Tr}_{\mathrm{NLP}}(\mathbb{P}))$.

(ii) *We show that* $f : \mathrm{SM}(\mathbb{P}) \to \mathrm{SM}(\mathrm{Tr}_{\mathrm{NLP}}(\mathbb{P}))$ *is a bijection:*
Clearly f is an injection: $M \neq M'$ implies $f(M) \neq f(M')$. To show that f is a surjection, we consider an arbitrary $N \in \mathrm{SM}(\mathrm{Tr}_{\mathrm{NLP}}(\mathbb{P}))$ and show $N \cap \mathrm{At}(\mathbb{P}) = M \in \mathrm{SM}(\mathbb{P})$ and $f(M) = f(N \cap \mathrm{At}(\mathbb{P})) = N$.

— Assume first $M \not\models P^{M,I}$, that is, there is a rule $r \in P^{M,I}$ that is not satisfied. Notice that all the rules in $P^{M,I}$ corresponding to a choice rule in P are always satisfied in M. Thus, we need to consider only rules that correspond to a weight rule in P. Now, $r = a \leftarrow w' \leqslant \{B \setminus I = W_{B \setminus I}\} \in P^{M,I}$, if there is a weight rule $a \leftarrow w \leqslant \{B = W_B, {\sim}C = W_C\} \in P$ such that $w' = \max(0, w - \sum_{b \in B \cap M \cap I} w_b - \sum_{c \in C \setminus M} w_c)$. Since $M \not\models r$, then $a \notin M$ and $w' \leqslant \sum_{b \in (B \setminus I) \cap M} w_b$. Define $B' = B \cap M$ and $C' = C \setminus M$. Since $N \cap \mathrm{At}(\mathbb{P}) = M$ and $w \leqslant \sum_{b \in B'} w_b + \sum_{c \in C'} w_c$, there is a normal rule $a \leftarrow B', {\sim}C' \in R$. Furthermore, $C' \cap N = \emptyset$ and $B' \cap I \subseteq N$ imply $r' = a \leftarrow B' \setminus I \in R^{N,I}$. Since $a \notin M$, also $a \notin N$. Furthermore, $B' \setminus I \subseteq M \subseteq N$. These imply $N \not\models r'$ which leads to a contradiction. Thus $M \models r$, and moreover $M \models P^{M,I}$.

— Next, assume that there is $M' \subset M \setminus I$ such that $M' \models P^{M,I}$ and define $N' = M' \cup (N \setminus \mathrm{At}(\mathbb{P})) \subset N \setminus I$. Since $N' \setminus \mathrm{At}(\mathbb{P}) = N \setminus \mathrm{At}(\mathbb{P})$ by definition, each rule of the form $\overline{a} \in R^{N,I}$ is satisfied in N'. Other rules in $R^{N,I}$ are of the form $r = a \leftarrow B' \setminus I$ where $a \in \mathrm{At}(\mathbb{P})$. There are now two possibilities:

(1) There is a choice rule $\{A\} \leftarrow B', {\sim}C \in P$, such that $a \in A$, $B' \cap I \subseteq N$, $C \cap N = \emptyset$, and $\overline{a} \notin N$. Now, $M \cap \mathrm{At}(\mathbb{P}) = N \cap \mathrm{At}(\mathbb{P})$ implies $r \in P^{M,I}$, and furthermore, $M' \models P^{M,I}$ implies $M' \models r$. Recalling $M' \cap \mathrm{At}(\mathbb{P}) = N' \cap \mathrm{At}(\mathbb{P})$, we get $N' \models r$.

(2) There is a rule $a \leftarrow B', {\sim}C' \in R$ corresponding to a weight rule $a \leftarrow w \leqslant \{B = W_B, {\sim}C = W_C\} \in P$ such that $B' \subseteq B$, $C' \subseteq C$, $w \leqslant \sum_{b \in B'} w_b + \sum_{c \in C'} w_c$, $B' \cap I \subseteq N$ and $C' \cap N = \emptyset$. If $B' \setminus I \not\subseteq N'$, then $N' \models r$. Assume

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

that $B' \setminus I \subseteq N'$. It follows from $B' \subseteq B$ and $C' \subseteq C$ that

$$w \leqslant \sum_{b \in (B \setminus I) \cap N'} w_b + \sum_{b \in B \cap I \cap N} w_b + \sum_{c \in C \setminus N} w_c.$$

Since $M \cap \mathrm{At}(\mathbb{P}) = N \cap \mathrm{At}(\mathbb{P})$, there is $r' = a \leftarrow w' \leqslant \{B \setminus I = W_{B \setminus I}\} \in P^{M,I}$ such that $w' = \max(0, w - \sum_{b \in B \cap I \cap M} w_b + \sum_{c \in C \setminus M} w_c)$. Furthermore, $w' \leqslant \sum_{b \in (B \setminus I) \cap M'} w_b = \sum_{b \in (B \setminus I) \cap N'} w_b$. Since $M' \models r'$, we have $a \in M'$, and also $a \in N'$. Thus $N' \models r$.

Thus, using the assumption $M \setminus I \neq \mathrm{LM}(P^{M,I})$ we can show that there is $N' \subset N \setminus I$ such that $N' \models R^{N,I}$, which leads to a contradiction with $N \setminus I = \mathrm{LM}(R^{N,I})$. Therefore, $M \setminus I = \mathrm{LM}(P^{M,I})$.

— Finally, we show that $f(M) = f(N \cap \mathrm{At}(\mathbb{P})) = N$. Let $N' = f(M)$, that is,

$$N' = M \cup \{\overline{a} \mid a \in \mathrm{Choices}(P) \setminus M\}.$$

Notice that $N \cap \mathrm{At}(\mathbb{P}) = N' \cap \mathrm{At}(\mathbb{P}) = M$. Assume $N \nsubseteq N'$, that is, there is $\overline{a} \in N$ such that $\overline{a} \notin N'$. Since $\overline{a} \notin N'$, we have $a \in N'$ and furthermore $a \in N$. The only rule r in R such that $\overline{a} = \mathrm{Head}(r)$ is $\overline{a} \leftarrow {\sim}a$. However, if $a \in N$, there is no rule in $R^{N,I}$ in which $\overline{a}$ appears in the head. Because $N \setminus I$ is the least model of $R^{N,I}$, we have $\overline{a} \notin N$, a contradiction. Assume then $N' \nsubseteq N$, that is, there is $\overline{a} \in N'$ such that $\overline{a} \notin N$. Since $\overline{a} \in N'$, we have $a \notin N'$ and furthermore $a \notin N$. If $a \notin N$, then $\overline{a} \in R^{N,I}$. Since $N \setminus I \models R^{N,I}$, we must have $\overline{a} \in N$, a contradiction. Therefore it holds that $N = N'$.

Thus we have shown that $\mathrm{Tr}_{\mathrm{NLP}}$ is a strongly faithful, $\sqcup$-preserving, and modular translation function. □

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK.

Brain, M., Crick, T., Vos, M. D. and Fitch, J. August 2006. TOAST: Applying answer set programming to superoptimisation. In *Logic Programming: 22nd International Conference, ICLP 2006*, Seattle, WA, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer, Heidelberg, Germany, 270–284.

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

Brooks, D. R., Erdem, E., Erdogan, S. T., Minett, J. W. and Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning 39*(4), 471–511.

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

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

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Eiter, T. and Fink, M. December 2003. Uniform equivalence of logic programs under the stable model semantics. In *Logic Programming: Nineteenth International Conference, ICLP 2003*, Mumbai, India, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916. Springer, Heidelberg, Germany, 224–238.

Eiter, T., Fink, M., Tompits, H. and Woltran, S. January 2004. Simplifying logic programs under uniform and strong equivalence. In *Logic Programming and Nonmonotonic Reasoning: Seventh International Conference, LPNMR 2004*, Fort Lauderdale, FL, V. Lifschitz and I. Niemelä, Eds. Lecture Notes in Artificial Intelligence, vol. 2923. Springer, Heidelberg, Germany, 87–99.

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

Eiter, T., Gottlob, G. and Veith, H. July 1997b. Modular logic programming and generalized quantifiers. In *Logic Programming and Nonmonotonic Reasoning: Fourth International Conference, LPNMR'97*, Dagstuhl Castle, Germany, J. Dix, U. Furbach and A. Nerode, Eds. Lecture Notes in Artificial Intelligence, vol. 1265. Springer, Heidelberg, Germany, 290–309.

Eiter, T., Tompits, H. and Woltran, S. July–August 2005. On solution correspondences in answer-set programming. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, Edinburgh, Scotland, UK, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 97–102.

Erdem, E., Lifschitz, V. and Ringe, D. 2006. Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming 6*(5), 539–558.

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. January 2005. Magic sets and their application to data integration. In *Database Theory—ICDT 2005: Tenth International Conference*, Edinburgh, UK, T. Eiter and L. Libkin, Eds. Lecture Notes in Computer Science, vol. 3363. Springer, Heidelberg, Germany, 306–320.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science 1*, 51–60.

Ferraris, P. and Lifschitz, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5*(1–2), 45–74.

Gaifman, H. and Shapiro, E. Y. 1989. Fully abstract compositional semantics for logic programs. In *Conference Record of the Sixteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, Austin, TX, 134–142.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. May 2007. clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning: Ninth International Conference, LPNMR 2007*, Tempe, AZ, USA, C. Baral, G. Brewka and J. S. Schlipf, Eds. Lecture Notes in Artificial Intelligence, vol. 4483. Springer, Heidelberg, Germany, 260–265.

Gelfond, M. and Gabaldon, A. 1999. Building a knowledge base: An example. *Annals of Mathematics and Artificial Intelligence 25*(3–4), 165–199.

Gelfond, M. and Leone, N. 2002. Logic programming and knowledge representation—the A-Prolog perspective. *Artificial Intelligence 138*(1–2), 3–38.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Gelfond, M. and Lifschitz, V. August 1988. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, Seattle, WA, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Cambridge, MA, 1070–1080.

Gelfond, M. and Lifschitz, V. June 1990. Logic programs with classical negation. In *Logic programming: Proceedings of the Seventh International Conference*, Jerusalem, Israel, D. H. D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, MA, 579–597.

Giordano, L. and Martelli, A. 1994. Structuring logic programs: A modal approach. *Journal of Logic Programming 21*(2), 59–94.

Heljanko, K. and Ştefănescu, A. 2004. Complexity results for checking distributed implementability. Technical Report 05/2004, Institute of Formal Methods in Computer Science, University of Stuttgart, Stuttgart, Germany.

Heljanko, K. and Ştefănescu, A. June 2005. Complexity results for checking distributed implementability. In *Fifth International Conference on Application of Concurrency to System Design, ACSD 2005*, St. Malo, France, J. Desel and Y. Watanabe, Eds. IEEE Computer Society, 78–87.

Ianni, G., Ielpa, G., Pietramala, A., Santoro, M. C. and Calimeri, F. June 2004. Enhancing answer set programming with templates. In *Tenth International Workshop on Non-Monotonic Reasoning (NMR 2004)*, Whistler, Canada, J. P. Delgrande and T. Schaub, Eds. 233–239.

Jaffar, J. and Maher, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Janhunen, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics 16*(1–2), 35–86.

Janhunen, T. 2007. Intermediate languages of ASP systems and tools. In *Proceedings of the First International Workshop on Software Engineering for Answer Set Programming*, M. De Vos and T. Schaub, Eds. Number CSBU-2007-05 in Deparment of Computer Science, University of Bath, Technical Report Series. Tempe, AZ, Heidelberg, Germany, 12–25.

Janhunen, T. and Oikarinen, E. September 2002. Testing the equivalence of logic programs under stable model semantics. In *Logics in Artificial Intelligence, Eighth European Conference, JELIA 2002*, Cosenza, Italy, S. Flesca, S. Greco, N. Leone and G. Ianni, Eds. Lecture Notes in Artificial Intelligence, vol. 2424. Springer, Heidelberg, Germany, 493–504.

Janhunen, T. and Oikarinen, E. 2007. Automated verification of weak equivalence within the Smodels system. *Theory and Practice of Logic Programming* 7(6), Heidelberg, Germany, 697–744.

Janhunen, T., Oikarinen, E., Tompits, H. and Woltran, S. May 2007. Modularity aspects of disjunctive stable models. In *Logic Programming and Nonmonotonic Reasoning: Ninth International Conference, LPNMR 2007*, Tempe, AZ, USA, C. Baral, G. Brewka and J. Schlipf, Eds. Lecture Notes in Artificial Intelligence, vol. 4483. Springer, 175–187.

Järvisalo, M. and Oikarinen, E. September 2007. Extended ASP tableaux and rule redundancy in normal logic programs. In *Logic Programming: 23rd International*

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Conference, ICLP 2007, Porto, Portugal, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, Heidelberg, Germany, 134–148. To appear in Theory and Practice of Logic Programming, arXiv:cs.AI/0809.3204.

Lierler, Y. September 2005. cmodels—SAT-based disjunctive answer set solver. In *Logic Programming and Nonmonotonic Reasoning, Eighth International Conference, LPNMR 2005*, Diamante, Italy, C. Baral, G. Greco, N. Leone and G. Terracina, Eds. Lecture Notes in Artificial Intelligence, vol. 3662. Springer, Heidelberg, Germany, 447–451.

Lifschitz, V. August 1985. Computing circumscription. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, A. K. Joshi, Ed. Morgan Kaufmann, San Francisco, CA, 121–127.

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

Lifschitz, V. and Turner, H. June 1994. Splitting a logic program. In *Logic Programming: Proceedings of the Eleventh International Conference on Logic Programming*, Santa Margherita Ligure, Italy, P. V. Hentenryck, Ed. MIT Press, Cambridge, MA, 23–37.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157*(1–2), 115–137.

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

Mancarella, P. and Pedreschi, D. August 1988. An algebra of logic programs. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, Seattle, WA, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Cambridge, MA, 1006–1023.

Marek, V. W. and Truszczyński, M. 1991. Autoepistemic logic. *Journal of the ACM 38*(3), 588–619.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczyński and D. S. Warren, Eds. Springer, Heidelberg, Germany, 375–398.

Meyer, A. R. July 1988. Semantical paradigms: Notes for an invited lecture, with two appendices by Stavros S. Cosmadakis. In *Third Annual Symposium on Logic in Computer Science*, Edinburgh, Scotland, UK, Y. Gurevich, Ed. IEEE Computer Society, 236–253.

Miller, D. 1989. A logical analysis of modules in logic programming. *Journal of Logic Programming 6*(1–2), 79–108.

Mostowski, A. 1957. On a generalization of quantifiers. *Fundamenta Mathematicae 44*, 12–36.

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

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. and Barry, M. March 2001. An A-Prolog decision support system for the space shuttle. In *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001*,

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Las Vegas, NV, I. V. Ramakrishnan, Ed. Lecture Notes in Computer Science, vol. 1990. Springer, Heidelberg, Germany, 169–183.

Oikarinen, E. May 2007. Modularity in smodels programs. In *Logic Programming and Nonmonotonic Reasoning: Ninth International Conference, LPNMR 2007*, Tempe, AZ, C. Baral, G. Brewka and J. Schlipf, Eds. Lecture Notes in Artificial Intelligence, vol. 4483. Springer, Heidelberg, Germany, 321–326.

Oikarinen, E. and Janhunen, T. August–Sepetember 2006. Modular equivalence for normal logic programs. In *ECAI 2006, Seventeenth European Conference on Artificial Intelligence*, Riva del Garda, Italy, G. Brewka, S. Coradeschi, A. Perini and P. Traverso, Eds. IOS Press, Amsterdam, the Netherlands, 412–416.

Oikarinen, E. and Janhunen, T. September 2008. Implementing prioritized circumscription by computing disjunctive stable models. In *Artificial Intelligence: Methodology, Systems, and Applications, Thirteenth International Conference, AIMSA 2008*, Varna, Bulgaria, D. Dochev, M. Pistore and P. Traverso, Eds. Lecture Notes in Artificial Intelligence, vol. 5223. Springer, Heidelberg, Germany, 167–180.

Oikarinen, E. and Janhunen, T. August 2008. A translation-based approach to the verification of modular equivalence. *Journal of Logic and Computation*, Advance Access published, doi:10.1093/logcom/exn039.

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

Pearce, D., Tompits, H. and Woltran, S. December 2001. Encodings for equilibrium logic and logic programs with nested expressions. In *Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, Tenth Portuguese Conference on Artificial Intelligence, EPIA 2001*, Porto, Portugal, P. Brazdil and A. Jorge, Eds. Lecture Notes in Artificial Intelligence, vol. 2258. Springer, Heidelberg, Germany, 306–320.

Przymusinski, T. C. August 1988. Perfect model semantics. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, Seattle, WA, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Cambridge, MA, 1081–1096.

Sagiv, Y. March 1987. Optimizing datalog programs. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Diego, CA. ACM Press, New York, 349–362.

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

Soininen, T., Niemelä, I., Tiihonen, J. and Sulonen, R. 2001. Representing configuration knowledge with weight constraint rules. In *Proceedings of the First International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge (ASP 2001)*, A. Provetti and T. C. Son, Eds. AAAI Press, Stanford, CA.

Tari, L., Baral, C. and Anwar, S. September 2005. A language for modular answer set programming: Application to ACC tournament scheduling. In *Answer Set Programming, Advances in Theory and Implementation, Proceedings of the Third International ASP'05 Workshop*, Bath, UK, M. D. Vos and A. Provetti, Eds. CEUR Workshop Proceedings, vol. 142. CEUR-WS.org.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*(2), 146–160.

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. September 2004. Characterizations for relativized notions of equivalence in answer set programming. In *Logics in Artificial Intelligence: Ninth European Conference, JELIA 2004*, Lisbon, Portugal, J. J. Alferes and J. A. Leite, Eds. Lecture Notes in Artificial Intelligence, vol. 3229. Springer, Heidelberg, Germany, 161–173.

Woltran, S. May 2007. A common view on strong, uniform, and other notions of equivalence in answer-set programming. In *Proceedings of the LPNMR'07 Workshop on Correspondence and Equivalence for Nonmonotonic Theories (CENT2007)*, Tempe, AZ, D. Pearce, A. Polleres, A. Valverde, and S. Woltran, Eds. CEUR Workshop Proceedings, vol. 265. CEUR-WS.org, 13–24.

https://doi.org/10.1017/S147106840800358X Published online by Cambridge University Press