TPLP **20** (5): 767–782, 2020. © The Author(s), 2020. Published by Cambridge University Press
doi:10.1017/S1471068420000265

Modular Answer Set Programming as a Formal Specification Language

PEDRO CABALAR
University of Corunna, Spain
(*e-mail:* cabalar@udc.es)

JORGE FANDINNO
University of Potsdam, Germany
(*e-mail:* fandinno@uni-potsdam.de)

YULIYA LIERLER
University of Nebraska Omaha, USA
(*e-mail:* ylierler@unomaha.edu)

submitted 7 August 2020; revised 12 August 2020; accepted 12 August 2020

Abstract

In this paper, we study the problem of formal verification for Answer Set Programming (ASP), namely, obtaining a *formal proof* showing that the answer sets of a given (non-ground) logic program P correctly correspond to the solutions to the problem encoded by P, regardless of the problem instance. To this aim, we use a formal specification language based on ASP modules, so that each module can be proved to capture some informal aspect of the problem in an isolated way. This specification language relies on a novel definition of (possibly nested, first order) *program modules* that may incorporate local hidden atoms at different levels. Then, *verifying* the logic program P amounts to prove some kind of equivalence between P and its modular specification.

KEYWORDS: Answer Set Programming, Formal Specification, Formal Verification, Modular Logic Programs.

1 Introduction

Achieving trustworthy AI systems requires, among other qualities, the assessment that those systems produce correct judgments[1] or, in other words, the ability to verify that produced results adhere to specifications on expected solutions. These specifications may have the form of expressions in some formal language or may amount to statements in natural language (consider English used in mathematical texts). Under this trust-oriented perspective, AI systems built upon some Knowledge Representation (KR) paradigm start from an advantageous position, since their behavior is captured by some declarative

[1] *Ethics Guidelines For Trustworthy AI*, High-level Expert Group on Artificial Intelligence set up by the European Commission.
https://ec.europa.eu/digital-single-market/en/news/ethics-guidelines-trustworthy-ai.

machine-interpretable formal language. Moreover, depending on its degree of declarativity, a KR formalism can also be seen as a specification language by itself.

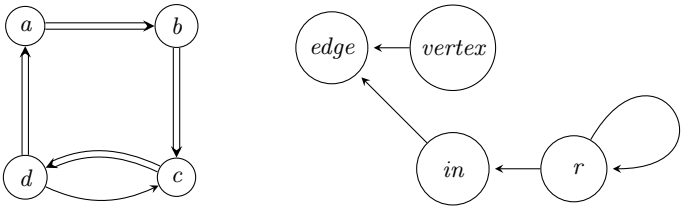
Answer Set Programming (ASP; Niemelä 1999; Marek and Truszczyński 1999) is a well-established KR paradigm for solving knowledge-intensive search/optimization problems. Based on logic programming under the *answer set semantics* (Gelfond and Lifschitz 1988), the ASP methodology relies on devising a logic program so that its answer sets are in one-to-one correspondence to the solutions of the target problem. This approach is fully declarative, since the logic program only describes a problem and conditions on its solutions, but abstracts out the way to obtain them, delegated to systems called answer set solvers. Thus, it would seem natural to consider an ASP program to serve a role of a formal specification on its expected solutions. However, the non-monotonicity of the ASP semantics makes it difficult to directly associate an independent meaning to an arbitrary program fragment (as we customary do, for instance, with theories in classical logic). This complicates assessing that a given logic program reflects, in fact, its intended *informal description*. And yet, ASP practitioners do build logic programs in groups of rules and identify each group with some part of the informal specification (Erdoğan and Lifschitz 2004; Lifschitz 2017). Moreover, modifications on a program frequently take place within a group of rules rather than in the program as a whole. The safety of these local modifications normally relies on such properties in ASP as *splitting* (Lifschitz and Turner 1994; Ferraris et al. 2011).

With these observations at hand, we propose a *verification methodology for logic programs*, where the argument of correctness of a program is decomposed into respective statements of its parts, relying to this aim on a modular view of ASP in spirit of approaches by Oikarinen and Janhunen (2009), and by Harrison and Lierler (2016). In this methodology, a *formal specification* Π is formed by a set of modules called *modular program*, so that each module (a set of program rules) is *ideally small enough to be naturally related to its informal description* in isolation. This relation can be established using quasi-formal English statements (Denecker et al. 2012; Lifschitz 2017) or relying on classical logic (Lifschitz et al. 2018). The same specification Π may serve to describe different (non-modular) ASP programs encoding the same problem. Each such program P respects given priorities involving efficiency, readability, flexibility, etc. As usual in *Formal Methods* (Monin 2003), *verification* then consists in obtaining a *formal proof* of the correspondence between the verified object (in our case, the non-modular encoding P) and its specification (the set Π of modules matching the informal aspects of that problem). It is important to note that the formal specification language used is subject to two important requirements: (i) dealing with *non-ground* programs; and (ii) capturing stable models of these programs using an expression that can be formally manipulated. For (i), we could use *Quantified Equilibrium Logic* (Pearce and Valverde 2008) but for (ii) the equivalent formulation by Ferraris (2011) is more suitable, as it captures stable models of a program as a second-order logic formula,[2] the SM operator.

Once a modular specification is guaranteed (related to its informal description/proved to be correct with respect to its informal description), we expect that arguing the correctness of the replacement of some of its module by a different encoding is reduced to

[2] The need for second-order logic is not surprising: the stable models of a logic program allow us to capture a transitive closure relation.

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press



(a) Graph G_1 (b) Dependency graph for Π^m_{hc}

Fig. 1: A pair of graphs used in the examples.

arguing some kind of equivalence between modules without affecting the rest of the correctness proof. The difficulty here appears when *auxiliary predicates* are involved. These predicates are quite common to improve the solver performance in a given encoding P but, more importantly, they are sometimes indispensable in the specification Π to express some property that the ASP language cannot directly capture otherwise (Gonçalves et al. 2016). In both cases, their presence must be taken into account when proving correctness which, in its turn, normally depends on their *local use in some part of the program.*

In this paper, we extend the modular language by Harrison and Lierler (2016) to allow for a *hierarchical tree* of modules and submodules, each of them possibly declaring its own set of *public* and *hidden* predicates at different levels. We use this extension as a language for formal specifications. For illustration, we consider a logic program encoding the well known Hamiltonian Cycle (HC) problem. We start by providing a formal specification of the HC problem using the hierarchy of modules and relate it to the informal description of the problem. We then formally prove the correspondence between the HC logic program and its hierarchical specification. This constitutes the argument of correctness for the considered logic program. We also provide an example of module replacement that is verified through an equivalence proof that disregards the existing auxiliary predicates.
Paper outline: Section 2 provides a running example of the HC problem encoding and presents our methodology. In Section 3, we revisit the SM operator and extend it with hidden predicates. Section 4 presents our modular logic programs, while Sections 5 and 6 explain their use for formal verification, illustrating the proposed methodology on the running example.

2 Motivating example and methodology

We consider a well-known domain in the ASP literature: the search of Hamiltonian cycles in a graph. A Hamiltonian cycle is a cyclic path from a directed graph that visits each of the graph's vertex exactly once. For instance, Figure 1a depicts graph G_1, whose unique Hamiltonian cycle is marked in double lines, whereas Listing 1 presents a possible encoding of the problem in the language of ASP solver CLINGO (Gebser et al. 2007). The `#show` directive is used to tell CLINGO which predicates (we call these *public*) should appear in the obtained answer sets: in this case, only predicate `in/2` that captures the edges in the solution. Now, if we add the facts in Listing 2 corresponding to graph G_1 and instruct CLINGO to find all the answer sets we obtain the output in Listing 3, which is the only Hamiltonian cycle in the graph. ASP practitioners usually explain Listing 1 in groups of rules. Rule 3 is used to generate all possible subsets of edges, while Rules 7-8

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

Listing 1: Encoding of a Hamiltonian cycle problem using CLINGO.

```
1 vertex(X):- edge(X,Y).
  vertex(X):- edge(Y,X).
3 { in(X,Y) }:- edge(X,Y).
  r(X,Y):- in(X,Y).
5 r(X,Y):- r(X,Z), r(Z,Y).
  :- not r(X,Y), vertex(X), vertex(Y).
7 :- in(X,Y), in(X,Z), Y != Z.
  :- in(X,Y), in(Z,Y), X != Y.
9 #show in/2.
```

Listing 2: Facts describing graph G_1.

```
edge(a,b).  edge(b,c).  edge(c,d).  edge(d,a).  edge(d,c).
```

Listing 3: Output by CLINGO for program composed of lines in Listing 1 and 2.

```
Answer: 1
in(a,b) in(b,c) in(c,d) in(d,a).
```

Listing 4: Alternative code to lines 4-6 in Listing 1 (Marek and Truszczyński 1999).

```
ra(Y) :- in(a,Y).
ra(Y) :- in(X,Y), ra(X).
:- not ra(X), vertex(X).
```

guarantee that connections among them are linear. Rules 4-5 are meant to define the auxiliary predicate r (for "reachable") as the transitive closure of predicate *in*. To assign this meaning to r, we *implicitly assume* that this predicate does not occur in other rule heads in the rest of the program. Predicate r is then used in Rule 6 to enforce that any pair of vertices are connected. Rules 4-6 together guarantee that facts for *in* form a strongly connected graph covering all vertices in the given graph.

Methodology. The methodology we propose for verifying the correctness of some logic program under answer set semantics consists of the following steps:

Step I.	Decompose the informal description of the problem into independent (natural language) statements S_i, identifying their possible hierarchical organization.
Step II.	Fix the public predicates used to represent the problem and its solutions.
Step III.	Formalize the specification of the statements as a non-ground modular program Π, possibly introducing (modularly local) auxiliary predicates.
Step IV.	Construct an *argument* (a "metaproof" in natural language) for the correspondence between Π and the informal description of the problem.
Step V.	Verify that the given logic P program adheres to the formal specification Π.

Note that the first four steps are exclusively related to the formal specification Π of the problem, while the particular program P to be verified is only considered in Step V, where formal verification proofs are produced.

Now, back to Hamiltonian cycle problem, a possible and reasonable result of Step I is the hierarchy of statements:

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

1. A Hamiltonian cycle G' of graph G must be a subgraph of G that contains all vertices of G, that is:
   (a) G' has the same vertices as G, and
   (b) all edges of G' also belong to G.
2. A Hamiltonian cycle G' of graph G is a cycle that visits all vertices of G exactly once, that is:
   (a) no vertex has more than one outgoing/incoming edge on G', and
   (b) G' is strongly connected.

The choice for public predicates (Step II) is, of course, arbitrary, but must be decided to compare different encodings (as also happens, for instance, when we fix a benchmark). Here, we choose predicates *edge*/2 and *in*/2 to encode the edges of the input graph G and the Hamiltonian cycle G', respectively. To prove the correctness of the encoding in Listing 1, we resume the rest of our methodological steps later on, in Sections 5 and 6. For instance, Step III is shown in Section 5, where we define a formal specification Π_1 that happens to comprise the same rules as Listing 1 but for the one in line 8. The main difference is that rules in Π_1 are grouped in modules corresponding to the above hierarchy. A set of propositions in Section 5 are used to establish the correspondence between Π_1 (Step IV) and the informal statements. The already mentioned strong relation between Listing 1 and Π_1 is not something we can always expect. As happens with *refactoring* in software engineering, encodings usually suffer a sequence of modifications to improve some of their attributes (normally, a better efficiency) without changing their functionality. Each new version implies a better performance of the answer set solver, but its correspondence with the original problem description becomes more and more obscure (Buddenhagen and Lierler 2015). For instance, it might be noted that program in Listing 1 produces an excessively large ground program when utilized on graphs of non trivial size. It turns out that it is enough to require that all vertices are reachable from some fixed node in the graph (for instance a). Thus, rules 4-6 of Listing 1 are usually replaced by rules in Listing 4. The answer sets with respect to predicate *in* are identical, if the graph contains a vertex named a. In this sense, *verifying* an ASP program can mean establishing some kind of *equivalence* result between its formal specification in the form of a modular program and the final program obtained from the refactoring process (Lierler 2019). This is tackled in Section 6 and constitutes Step V.

3 Operator SM with hidden predicates

Answer set semantics has been extended to arbitrary first-order (FO) theories with the introduction of *Quantified Equilibrium Logic* (Pearce and Valverde 2008) and its equivalent formulation using the second-order (SO) operator SM (Ferraris et al. 2011). These approaches allow us to treat program rules as logical sentences with a meaning that bypasses grounding. For instance, rules in Listing 1 respectively correspond to:

$$\forall xy(edge(x,y) \rightarrow vertex(x)) \tag{1}$$

$$\forall xy(edge(y,x) \rightarrow vertex(x)) \tag{2}$$

$$\forall xy(\neg\neg in(x,y) \wedge edge(x,y) \rightarrow in(x,y)) \tag{3}$$

$$\forall xy(in(x,y) \rightarrow r(x,y)) \tag{4}$$

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

$$\forall xyz(r(x,z) \wedge r(z,y) \rightarrow r(x,y)) \tag{5}$$
$$\forall xy(\neg r(x,y) \wedge vertex(x) \wedge vertex(y) \rightarrow \bot) \tag{6}$$
$$\forall xyz(in(x,y) \wedge in(x,z) \wedge \neg(y=z) \rightarrow \bot) \tag{7}$$
$$\forall xyz(in(x,z) \wedge in(y,z) \wedge \neg(x=y) \rightarrow \bot) \tag{8}$$

As we can see, the correspondence is straightforward except, perhaps, for the *choice* rule in line 3 of Listing 1 that is represented as formula (3) (see Ferraris 2005 for more details). We name the conjunction of sentences (1)-(8) as our encoding P_1.

We now recall the SM operator (Ferraris et al. 2011), assuming some familiarity with second order (SO) logic. We adopt some notation: a letter in boldface inside a formula denotes a tuple of elements also treated as a set, if all elements are different. Quantifiers with empty tuples (or empty sets of variables) can be removed: $\exists\emptyset F = \forall\emptyset F := F$. Expression $pred(F)$ stands for the set of free predicate names (different from equality) in a SO formula F. If p and q are predicate names of the same arity m then $p \leq q$ is an abbreviation of $\forall\mathbf{x}(p(\mathbf{x}) \rightarrow q(\mathbf{x}))$, where $\mathbf{x}$ is an m-tuple of FO variables. Let $\mathbf{p}$ and $\mathbf{q}$ be tuples $p_1,\ldots,p_n$ and $q_1,\ldots,q_n$ of predicate symbols or variables. Then $\mathbf{p} \leq \mathbf{q} := (p_1 \leq q_1) \wedge \cdots \wedge (p_n \leq q_n)$, and $\mathbf{p} < \mathbf{q}$ is an abbreviation of $(\mathbf{p} \leq \mathbf{q}) \wedge \neg(\mathbf{q} \leq \mathbf{p})$. Given a FO formula F, its *stable model operator with intensional predicates* $\mathbf{p}$ (not including equality) is the SO formula

$$\mathrm{SM}\mathbf{p}[F] \quad := \quad F \wedge \neg\exists\mathbf{U}\big(\ (\mathbf{U} < \mathbf{p}) \wedge F^*\ \big), \tag{9}$$

with $\mathbf{U} = U_1,\ldots,U_n$ distinct predicate variables not occurring in $\mathbf{p}$ and:

$$F^* := \begin{cases} F & \text{if } F \text{ is an atomic formula without members of } \mathbf{p} \\ U_i(\mathbf{t}) & \text{if } F \text{ is } p_i(\mathbf{t}) \text{ for } p_i \in \mathbf{p} \text{ and } \mathbf{t} \text{ a tuple of terms} \\ G^* \otimes H^* & \text{if } F = (G \otimes H) \text{ and } \otimes \in \{\wedge,\vee\} \\ (G^* \rightarrow H^*) \wedge (G \rightarrow H) & \text{if } F = (G \rightarrow H) \\ Qx\ (G^*) & \text{if } F = Qx\ G \ \text{ and } Q \in \{\forall,\exists\} \end{cases}$$

Predicate symbols occurring in F but not in $\mathbf{p}$ are called *extensional* and are interpreted classically. In fact, if $\mathbf{p}$ is empty, it is easy to see that $\mathrm{SM}_{\mathbf{p}}[F]$ coincides with F. We say that an interpretation $\mathcal{I}$ over a signature $\mathcal{P}$ is an *answer set* of a FO formula F (representing a logic program) when it is a Herbrand model of $\mathrm{SM}_{\mathbf{p}}[F]$ and $\mathbf{p} = pred(F)$. When F is a logic program, answer sets defined in this way correspond to the traditional definition based on grounding (Ferraris et al. 2011). It is common to identify Herbrand interpretations with sets of atoms corresponding to its predicates and their extensions. For a Herbrand interpretation $\mathcal{I}$ over set $\mathcal{P}$ of predicates and set $\mathcal{S} \subseteq \mathcal{P}$, we write $\mathcal{I}_{|\mathcal{S}}$ to denote the restriction of $\mathcal{I}$ to $\mathcal{S}$. As usual, the *extent* of a predicate p in interpretation $\mathcal{I}$, written $p^{\mathcal{I}}$, collects every tuple of Herbrand terms $\mathbf{t}$ for which $p(\mathbf{t})$ holds in $\mathcal{I}$.

As a small example, let F_g be the conjunction of facts in Listing 2, F_1 denote a conjunction of F_g and (1) and let $\mathbf{U}$ be $\langle E, V\rangle$. Then, F_1^* is formed by the conjunction of $\forall xy((E(x,y) \rightarrow V(x)) \wedge (edge(x,y) \rightarrow vertex(x)))$ and all atoms $E(x,y)$, one per each fact $edge(x,y)$ in F_g. The answer sets of F_1 are captured by the Herbrand models of $\mathrm{SM}_{\langle edge,vertex\rangle}[F_1]$. This formula has a unique model that minimizes the extension of *edge* to the exact set of facts in F_g (and not more) and the extension of *vertex* to be precisely all nodes used as left arguments in those edges. If we take instead the formula $F_2 := P_1 \wedge F_g$ and $\mathbf{q} = pred(F_2)$ then $\mathrm{SM}_{\mathbf{q}}[F_2]$ has a unique Herbrand model that has the same atoms for predicate *in* as those in Listing 3 but, obviously, has also more atoms

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

for the remaining predicates in $pred(F_2)$. A simple way of removing (or forgetting) those extra predicates in SO is adding their existential quantification. Given formula F we define the *answer sets of* F *for* $\mathbf{p}$ *hiding predicates* $\mathbf{h}$ as the Herbrand models of:

$$\exists \mathbf{H}\ \mathrm{SM}_{\mathbf{p}}[F^{\mathbf{h}}_{\mathbf{H}}] \tag{10}$$

where $\mathbf{H}$ is a tuple of predicate variables of the same length as $\mathbf{h}$ and $F^{\mathbf{h}}_{\mathbf{H}}$ is the result of replacing all occurrences of predicate symbols from $\mathbf{h}$ by the corresponding predicate variables from $\mathbf{H}$. We abuse the notation and write $\exists \mathbf{h}\ \mathrm{SM}_{\mathbf{p}}[F]$ instead of (10) when it does not lead to confusion. We call predicates in tuples $\mathbf{p}$ and $\mathbf{h}$ *intensional* and *hidden*, respectively. For instance, $\exists edge\ vertex\ r\ \mathrm{SM}_{\mathbf{q}}[F_2]$ stands for the formula $\exists E\, V\, R\ \mathrm{SM}_{\mathbf{q}}[F'_2]$ where F'_2 is obtained from F_2 by replacing predicate symbols *edge*, *vertex*, r by variables E, V, R; and it has a unique Herbrand model that coincides with the one in Listing 3. Thus, this model forms the unique answer set of F_2 for $\mathbf{q}$ hiding predicates *edge*, *vertex* and r. This corresponds to the behavior produced by the `#show` directive in Line 9 of the Hamiltonian cycle encoding. The following proposition states that the existential quantifiers for hidden predicates filter out their information in the models.

Proposition 1. *Let F be a formula, $\mathbf{h}$ a tuple of predicate symbols from F and let $\mathcal{S} = pred(F) \setminus \mathbf{h}$. Then, $\mathcal{I}$ is a Herbrand model of* (10) *iff there is some answer set $\mathcal{J}$ of F such that $\mathcal{I} = \mathcal{J}_{|\mathcal{S}}$.*

4 Nested modular programs

As explained in the introduction, our specification language departs from the work on modular logic programs by Harrison and Lierler (2016), where each module is a FO formula whose semantics is captured by the SM operator. This choice is motivated by two factors. First, it allows treating programs or modules with variables as FO formulas that can be manipulated without resorting to the program grounding at all. This is crucial is we wish to argue about the meaning of each module regardless of the particular instance of the problem to be solved. Second, the SM operator captures the semantics of the program also as a formula that, although including SO quantifiers, can be formally treated using a manifold of technical results from the literature (including theorems on splitting or constraints, for instance). To adapt this modular approach to our purposes in this paper, we extend it in two ways: (i) handling auxiliary predicates inside modules and (ii) introducing nested modules.

A *def-module* M is a pair $(\mathbf{p} : F)$, where $\mathbf{p}$ is a tuple of intensional predicate symbols and F is a FO formula. Its semantics is captured by the formula $\Phi(M) \stackrel{\text{def}}{=} \mathrm{SM}_{\mathbf{p}}[F]$. When $\mathbf{p}$ is empty, we write F in place of def-module $(\mathbf{p} : F)$ — indeed, $\Phi(M)$ amounts to F. On the contrary, when $\mathbf{p} = pred(F)$ all predicates in F are intensional, so def-module $(pred(F) : F)$ represents the usual situation in a logic program F. If we leave some predicates as extensional, then their extent is not "minimized." For instance, rule (1) alone, i.e., the def-module $(edge, vertex : (1))$, has a unique answer set $\emptyset$ whereas in a def-module $(vertex : (1))$ no assumption is made on extensional predicate *edge*, while *vertex* collects precisely all left arguments of *edge*.

A *modular program* is a pair $\Pi = \langle \mathcal{S}, \mathcal{M} \rangle$, where $\mathcal{S} \subseteq \mathcal{P}$ is the set of *public* predicate symbols and $\mathcal{M}$ is the set of *modules* $\{M_1, \ldots, M_n\}$, where M_i $(1 \leq i \leq n)$ is either a

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

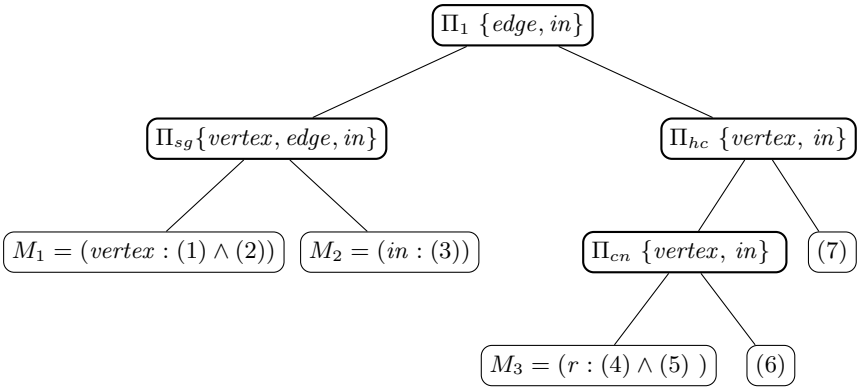


Fig. 2: Hierarchical structure of modular program Π_1.

def-module or another modular program (called *subprogram* of Π), so that all predicate symbols occurring in Π are in $\mathcal{P}$. A modular program can be depicted as a hierarchical tree, whose leafs are def-modules. The interpretation of a modular program $\Pi = \langle \mathcal{S}, \mathcal{M} \rangle$ is captured by the (recursively defined) formula:

$$\Phi(\Pi) \quad \stackrel{\text{def}}{=} \quad \exists \mathbf{h} \bigwedge \{\Phi(M_i) \mid M_i \in \mathcal{M}\}$$

where $\mathbf{h}$ contains all free predicate symbols in $\Phi(M_i)$ that are not in $\mathcal{S}$. Notice that, if M_i is a def-module $(\mathbf{p} : F)$, then $\Phi(M_i) = \mathrm{SM}_{\mathbf{p}}[F]$ as we defined before, while if M_i is a subprogram, we apply the formula above recursively. We say that an interpretation $\mathcal{I}$ is a *model* of a modular program Π, when it satisfies the formula $\Phi(\Pi)$. As in the previous section, we say that interpretation $\mathcal{I}$ over public predicate symbols $\mathcal{S}$ is an *answer set* of a modular program Π when it is a Herbrand model of Π. Programs Π and Π' are said to be *equivalent* if they have the same answer sets. Under logic programming syntax (like Listing 1), public predicates are declared via `#show` clauses. We sometimes allow the def-module $M = (\mathbf{p} : F)$ as an abbreviation of the modular program $\langle pred(F), \{M\} \rangle$.

We define $defmods(P)$ as the set of all def-modules in Π at any level in the hierarchy. Program $\Pi = \langle \mathcal{S}, \mathcal{M} \rangle$ is said to be *flat* when it does not contain subprograms, i.e., $\mathcal{M}$ coincides with $defmods(P)$. We call flat modular program $\langle \mathcal{S}, \mathcal{M} \rangle$ *HL-modular*, when its set $\mathcal{S}$ of public predicates contains all predicate symbols occurring in $\mathcal{M}$. HL-modular programs capture the definition of modular programs from (Harrison and Lierler 2016).

To illustrate these definitions, we provide a modular program Π_1 that will act later on as a specification for encoding P_1= (1)-(8) from Section 3. Program Π_1 comprises the same rules (1)-(7) (as we will see, (8) is actually redundant) but organizes them in the tree of Figure 2. The tree contains 5 def-modules (the leaf nodes) and has three subprograms, Π_{sb}, Π_{hc} and Π_{cn} at different levels. Each modular program node (drawn as a thick line box) also shows inside its set of public predicates. Note, for instance, how predicate r is local to subprogram $\Pi_{cn} = \langle \{vertex, in\}, \{\big(r : (4) \wedge (5)\big), (6)\} \rangle$ that corresponds to rules 4-6 in Listing 1 and intuitively states that relation in forms a strongly connected graph covering all vertices.

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

5 A formal specification language

Hamiltonian cycles constitute a good example of a typical use of ASP for solving a search problem. Following Brewka et al. (2011), a *search problem* X can be seen as a set of instances, being each *instance* I assigned a finite set $\Theta_X(I)$ of solutions. Under our verification method, we propose constructing a modular program Π_X that adheres to the specifications of X so that when extended with modular program Π_I representing an instance I of X, the answer sets of this join are in one to one correspondence with members in $\Theta_X(I)$. Then, we can use Π_X to argue, module by module, that each of its components actually corresponds to some part of the informal description of the problem. To illustrate these ideas we prove next that, indeed, program Π_1 presented in Figure 2 and described in the last section is a *formal specification* for the search of Hamiltonian cycles. We start by presenting the informal readings of all def-modules occurring in the program — i.e., members of $defmods(\Pi_1)$. The def-modules M_1, M_2 and M_3 intuitively formulate the following *Statements*:

S_1: "In any model of M_1, the extent of *vertex* collects all objects in the extent of *edge*."
S_2: "In any model of M_2, the extent of *in* is a subset of the extent of *edge*."
S_3: "In any model of M_3, the extent of r is the transitive closure of the extent of *in*."

These statements can be seen as *achievements* of each def-module in the sense of (Lifschitz 2017), that are agnostic to the context where def-modules appear. This closely aligns with good software engineering practices, where the emphasis is made on modularity/independence of code development. An intuitive meaning of formulas (6) and (7) is self explanatory given the underlying conventions of FO logic:

$S_{(6)}$: "In any model of (6), the extent of r contains each possible pair of vertices."
$S_{(7)}$: "In any model of (7), the extent of *in* does not contain two different pairs with the same left component."

Statements S_1 and S_2 translate into a joint *Statement* about module Π_{sg}:

S_{sg}: "In any model $\mathcal{I}$ of Π_{sg}, $\langle vertex^{\mathcal{I}}, in^{\mathcal{I}}\rangle$ is a subgraph of $\langle vertex^{\mathcal{I}}, edge^{\mathcal{I}}\rangle$."

Similarly, we identify the following two combined statements:

S_{cn}: "In any model of Π_{cn}, the extent of *in* forms a strongly connected graph covering all vertices."
S_{hc}: "In any model of Π_{hc}, the extent of *in* is a cycle visiting all vertices exactly once" or equivalently "it induces a graph that is a Hamiltonian cycle."

Note that each subcomponent of Π_1 is small enough so that the verification of its corresponding statement is a manageable and self-isolated task. Note also that statements S_{sg} and S_{hc} are the result of fixing the public vocabulary for the statements 1 and 2 identified from the informal description in Section 2. Statements 1a and 1b in the informal description correspond to statements S_1 and S_2, respectively. Statements 2a and 2b correspond to statements $S_{(7)}$ and S_{cn}.

At this point, we have completed Step III for our example, with the modular program Π_1 and the informal statements to compare with. Now, Step IV consists on building claims about the correctness of the modules versus the statements. Proofs for the correctness of statements S_1, S_2, S_3, $S_{(6)}$ and $S_{(7)}$ are obtained using properties of the SM

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

operator and can be found in (Cabalar et al. 2020). The formalization of Statement $S_{(7)}$ follows from the general result below, if we just replace predicate names p and q by *in* and r, respectively. Its proof can also be found in the extended version online (Cabalar et al. 2020).

Proposition 2. *Let formula* F_{tr}^{qp} *be*

$$\forall xy\big(p(x,y) \to q(x,y)\big) \land \forall xyz\big(q(x,z) \land q(z,y) \to q(x,y)\big) \tag{11}$$

For any arbitrary predicates p *and* q*, any model* $\mathcal{I}$ *of def-module* $(q : F_{tr}^{qp})$ *is such that the the extent of* q *is the transitive closure of the relation constructed from the extent of* p*.*

The next two results prove the correctness of S_{cn} and S_{hc}, respectively, for Π_{cn} and Π_{hc}.

Proposition 3. *Let* F^v *be formula* $\forall xy(\neg q(x,y) \land v(x) \land v(y) \to \bot)$*;* F_{tr}^{qp} *be formula* (11)*;* Π_{cn}^{vpq} *be a modular program* $\langle \{v,p\}, \{\, (q : F_{tr}^{qp}),\ F^v \,\}\rangle$*,* $\mathcal{I}$ *be an interpretation, and* $\langle v^{\mathcal{I}}, p^{\mathcal{I}}\rangle$ *be a graph. Then,* $\mathcal{I}$ *is a model of* Π_{cn}^{vpq} *iff for every pair* $a, b \in v^{\mathcal{I}}$ *of distinct vertices, there is a directed path from* a *to* b *in* $\langle v^{\mathcal{I}}, p^{\mathcal{I}}\rangle$*.*

Proof sketch. We just show the left to right direction. The complete proof can be found in the extended version online (Cabalar et al. 2020). Note first that F^v is classically equivalent to $\forall xy(v(x) \land v(y) \to q(x,y))$, so any pair of vertices satisfies $(a,b) \in q^{\mathcal{I}}$. From Proposition 2, relation $q^{\mathcal{I}}$ is the transitive closure of $p^{\mathcal{I}}$. Hence, path from a to b exists in graph $\langle v^{\mathcal{I}}, p^{\mathcal{I}}\rangle$. □

Proposition 4. *Let* Π_{cn}^{vpq} *be as in Proposition 3,* F^p *be formula* $\forall xyz(p(x,y) \land p(x,z) \land \neg(y = z) \to \bot)$*,* Π_{hc}^{vpq} *be modular program* $\langle \{v,p\}, \{\, \Pi_{cn}^{vpq},\ F^p \,\}\rangle$ *and* $\mathcal{I}$ *be an interpretation of* Π_{hc}^{vpq}*. Then,* $\mathcal{I}$ *is a model of graph* Π_{hc}^{vpq} *iff* $p^{\mathcal{I}}$ *are the edges of a Hamiltonian cycle of* $\langle v^{\mathcal{I}}, p^{\mathcal{I}}\rangle$ *that is, the elements of* $p^{\mathcal{I}}$ *can be arranged as a directed cycle* $(v_1,v_2),(v_2,v_3),\ldots,(v_n,v_1)$ *so that* $v_1,\ldots,v_n$ *are pairwise distinct and* $v^{\mathcal{I}} = \{v_1,\ldots,v_n\}$*.*

Proof sketch. Again, we show only the left to right direction. See the the extended version online version for the complete proof (Cabalar et al. 2020). If $\mathcal{I}$ is a model of Π_{hc}^{vpq}, the hypothesis in the enunciate implies that for any pair $v_1, v_m \in v^{\mathcal{I}}$ of vertices, there is a directed path $(v_1,v_2),(v_2,v_3),(v_3,v_4),\ldots,(v_{m-1},v_m)$ in $\langle v^{\mathcal{I}}, p^{\mathcal{I}}\rangle$. Hence, there exists:

$$(v_1,v_2),(v_2,v_3),(v_3,v_4),\ldots,(v_{m-1},v_m),(v_{m+1},v_{m+1}),\ldots,(v_n,v_1) \tag{12}$$

in graph $\langle v^{\mathcal{I}}, p^{\mathcal{I}}\rangle$ such that every vertex in $v^{\mathcal{I}}$ appears in it. Since $\mathcal{I}$ is also a model of F^p, Statement $S_{(7)}$ (modulo names of predicate symbols) is applicable. This implies $v_i \neq v_j$ for all $i \neq j$ and, thus, that all edges in (12) are distinct. Therefore, (12) is a directed cycle. Since this cycle covers all vertices of $\langle v^{\mathcal{I}}, p^{\mathcal{I}}\rangle$, it is also a Hamiltonian cycle. □

Modular programs Π_{cn} and Π_{hc} coincide with modular programs Π_{cn}^{vpq} and Π_{hc}^{vpq}, when predicate symbols v, p and q are replaced by *vertex*, *in* and r, respectively. Statements S_{cn} and S_{hc} follow immediately. Note also that modular programs Π_{cn} and Π_{hc} also preserve their meaning inside a larger modular program mentioning predicate symbol r in other parts. Indeed, symbol r is hidden or *local* (existentially quantified) so that its use elsewhere in a larger modular program has a different meaning.

To complete the modularization of the Hamiltonian cycle problem, we develop the encoding for a graph instance. Given a set E of graph edges, M_E denotes a def-module

$$\Big(\, edge : \bigwedge\{\, edge(a,b) \mid (a,b) \in E\}\,\Big) \tag{13}$$

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

The intuitive and formal meaning of def-module M_E is captured by the statement:

S_E: "In any model of M_E, the extent of *edge* is E."

Now, the Hamiltonian cycle problem on a given graph instance with edges E is encoded by $\Pi_1(E) := \langle\, \{in\},\, \{\Pi_1,\, M_E\}\,\rangle$. To prove that $\Pi_1(E)$ obtains the correct solutions, we can now just simply rely on the already proved fulfillment of the statements for Π_1.

Proposition 5. *Let $G = \langle V, E\rangle$ be a graph with non-empty sets of vertices V and edges E, where every vertex occurs in some edge, and $\mathcal{I}$ be an interpretation over signature $\{in\}$. Then, $\mathcal{I}$ is an answer set of $\Pi_1(E)$ iff $in^{\mathcal{I}}$ is a Hamiltonian cycle of G.*

Proof sketch. We only showcase the left to right direction (see Appendix B for the rest). Take interpretation $\mathcal{I}$ to be an answer set of $\Pi_1(E)$. Then, there exists a Herbrand interpretation $\mathcal{J}$ over signature $in, vertex, edge$ so that $\mathcal{J}$ coincides with $\mathcal{I}$ on the extent of in and $\mathcal{J}$ is also a model of all submodules of $\Pi_1(E)$. Thus, $\mathcal{J}$ adheres to Statements S_{sg}, S_{hc} and S_E about these submodules. This implies that the extent $in^{\mathcal{I}}$ forms a Hamiltonian cycle of $\langle vertex^{\mathcal{J}}, in^{\mathcal{J}}\rangle$ (Statement S_{hc}) and that $\langle vertex^{\mathcal{J}}, in^{\mathcal{J}}\rangle$ is a subgraph of $\langle vertex^{\mathcal{J}}, edge^{\mathcal{J}}\rangle$ (Statement S_{sg}). These two facts imply that $in^{\mathcal{I}}$ forms a Hamiltonian cycle of the graph $\langle vertex^{\mathcal{J}}, edge^{\mathcal{J}}\rangle$. Moreover, $vertex^{\mathcal{J}}$ and $edge^{\mathcal{J}}$ respectively coincide with sets V and E (Statements S_2 and S_E). Therefore, $in^{\mathcal{J}}$ forms a Hamiltonian cycle of the graph $\langle V, E\rangle = G$. Finally, recall that $in^{\mathcal{J}} = in^{\mathcal{I}}$, so the result holds. □

This result confirms that Π_1 is indeed a correct *formal specification* of the Hamiltonian cycle problem, for any arbitrary graph instance E. Propositions 2-5 are, in fact, an example of the application of Step IV of our methodology to the Hamiltonian cycle problem. It is worth to mention that, in most cases, the decomposition in modules and the properties of the SM operator allow us to replace the SM operator by a FO formula (using Clark's completion) or by the circumscription operator. This replacement greatly simplify the effort of the proofs.

6 Verification based on modular programs

The results in the previous section state that the answer sets of our modular specification Π_1 correspond to the Hamiltonian cycles of a graph. However, in general, there is no guarantee that the non-modular version of Π (i.e., the regular ASP program P formed by all rules in Π) has the same answer sets. Next, we introduce some general conditions under which the answer sets of a modular program Π and its non-modular version P coincide. These results are useful for Step V of the proposed methodology.

In the rest of the section, we assume that Π has the form $\langle \mathcal{S}, \mathcal{M}\rangle$. We also identify a regular program P with its direct modular version $\langle pred(P), \{(pred(P) : P)\}\rangle$. The *flattening* of Π is defined as $flat(\Pi) \stackrel{\text{def}}{=} \langle \mathcal{S}, defmods(\Pi)\rangle$. For example, $flat(\Pi_1(E)) = \langle\{in\}, \{M_1, M_2, M_3, (6), (7), (13)\}\rangle$. We say that Π is in *α-normal form* (α-NF) if all occurrences of a predicate name in $\Phi(\Pi)$ are free or they are all bound to a unique occurrence of existential quantifier. This happens, for instance, in $\Pi_1(E)$. Still, any formula $\Phi(\Pi)$ can always be equivalently reduced to α-NF by applying so-called α-transformations (i.e., choosing new names for quantified variables). In our context, this means changing hidden (auxiliary) predicate names in P until *different* symbols are used for distinct auxiliary predicates.

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

The next theorem states that, when modular program Π is in α-NF, we can ignore its recursive structure and instead consider its flat version.

Proposition 6. *For any modular program* Π *in* α*-NF, an interpretation* $\mathcal{I}$ *is a model of* Π *iff* $\mathcal{I}$ *is a model of* $\mathit{flat}(\Pi)$.

Thus, since $\Pi_1(E)$ is in α-NF, it is simply equivalent to $\mathit{flat}(\Pi_1(E))$. Next, we show how to relate a flat program with a non-modular program that contains exactly the same rules. We use this result to verify that program P_1 (corresponding to Listing 1) satisfies the Hamiltonian cycle specification. To formalize this relation we focus on a syntax closer to one of logic programs. Consider FO formulas that are conjunctions of rules of the form:

$$\widetilde{\forall}(a_{k+1} \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \ \wedge \neg\neg a_{m+1} \wedge \cdots \wedge \neg\neg a_n \rightarrow a_1 \vee \cdots \vee a_k), \quad (14)$$

where all a_i are atoms and $\widetilde{\forall}$ stands for the universal closure of their variables. As usual, the consequent and the antecedent of (14) are called *head* and *body*, respectively. The conjunction $a_{k+1} \wedge \cdots \wedge a_l$ constitutes the *positive (part of the) body*. A modular program Π is called *simple* if, for every def-module $(\mathbf{p} : F) \in \mathit{defmods}(\Pi)$, formula F is a conjunction of rules, and all head predicate symbols in F are intensional (occur in $\mathbf{p}$). This is, in fact, the case of all def-modules we considered so far. Let $\mathit{int}(\Pi)$ collect all intensional predicates in Π, that is $\mathit{int}(\Pi) \stackrel{\text{def}}{=} \bigcup\{\mathbf{p} \mid (\mathbf{p} : F) \in \mathit{defmods}(P)\}$. Then, we form the directed *dependency graph* $DG[\Pi]$ by taking $\mathit{int}(\Pi)$ as nodes and adding an edge (p, q) each time there is a rule occurring in Π with p in the head and q is in the positive body. For instance, the dependency graph of program $\Pi_1(E)$ is given in Figure 1b. This graph has four strongly connected components, each one consisting of a single node.

A modular program Π is *coherent* if it is simple, in α-NF, and satisfies two more conditions: (i) every pair of distinct def-modules $(\mathbf{p} : F)$ and $(\mathbf{p}' : F')$ in $\mathit{defmods}(\Pi)$ is such that $\mathbf{p} \cap \mathbf{p}' = \emptyset$, and (ii) for every strongly connected component `SCC` in $DG[\Pi]$, there is a def-module $(\mathbf{p} : F) \in \mathit{defmods}(\Pi)$ such that $\mathbf{p}$ contains all vertices in `SCC`. For example, $\Pi_1(E)$ is a coherent program. Now, let us collect the conjunction of all def-module formulas in Π as $\mathcal{F}(\Pi) \stackrel{\text{def}}{=} \bigwedge\{F \mid (\mathbf{p} : F) \in \mathit{defmods}(\Pi)\}$. We can observe, for instance, that $P_1 \wedge \mathcal{F}(M_E) = \mathcal{F}(\Pi_1(E)) \wedge (8) = \mathcal{F}(\mathit{flat}(\Pi_1(E))) \wedge (8)$, that is, the modular encoding of the Hamiltonian cycle problem and the non-modular one share the same rules but for (8). We now obtain a similar result to Theorem 3 in (Harrison and Lierler 2016) but extended to our framework. Together with Proposition 1, it connects modular programs with logic programs as used in practice. The proof of this result together with the proof of Theorem 2 below can be found in the extended version online (Cabalar et al. 2020).

Theorem 1. *Let* $\Pi = \langle \mathcal{S}, \mathcal{M} \rangle$ *be a coherent modular program,* $\mathbf{p}$ *be* $\mathit{int}(\Pi)$*, and* $\mathbf{h}$ *be* $\mathit{pred}(\Phi(\Pi)) \setminus \mathcal{S}$*. Then, (i) any interpretation* $\mathcal{I}$ *is a model of* Π *iff* $\mathcal{I}$ *is a model of formula* $\exists \mathbf{h}\ \mathrm{SM}_{\mathbf{p}}[\mathcal{F}(\Pi)]$*; (ii) any interpretation* $\mathcal{I}$ *is an answer set of* Π *iff there is some answer set* $\mathcal{J}$ *of* $\mathcal{F}(\Pi)$ *such that* $\mathcal{I} = \mathcal{J}_{|\mathcal{S}}$.

As a result, we can now prove that program in Listing 1 satisfies the formal specification Π_1 which, as we saw, captures the Hamiltonian cycle problem.

Proposition 7. *The answer sets of modular program* $\Pi_1(E)$ *coincide with the answer sets of logic program* $P_1 \wedge \mathcal{F}(M_E)$ *for intensional predicate symbol 'in' hiding all other predicate symbols of the program.*

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

Proof. Since modular program $\Pi_1(E)$ is coherent, by Theorem 1, it is equivalent to the formula to $\varphi := \exists \mathbf{h}\ \mathrm{SM}_{\mathbf{p}}[\mathcal{F}(\Pi_1) \wedge \mathcal{F}(M_E)]$ where $\mathbf{h} = \langle vertex, edge, r\rangle$. Now, φ is in its turn equivalent to $\exists \mathbf{h} \mathrm{SM}_{\mathbf{p}}[\mathcal{F}(\Pi_1) \wedge (8) \wedge \mathcal{F}(M_E)]$ since $\mathcal{F}(\Pi_1)$ entails formula (8) and def-module (8) has no intensional predicate symbols. Besides, formulas $\mathcal{F}(\Pi_1) \wedge (8)$ and P_1 are identical. From Proposition 1, it follows that Herbrand models of φ are the answer sets of $P_1 \wedge \mathcal{F}(M_E)$ restricted to predicate *in*. □

This proposition constitutes verification Step V that links an ASP encoding P_1 of the Hamiltonian cycle problem to its formal specification as a modular program Π_1. At a first sight, the effort may seem worthless, given that Π_1 and P_1 almost share the same rules. But this is a wrong impression, since P_1 is actually an ideal case, i.e. the one closest to Π_1, while the latter can still be used as a specification for other encodings. To show how, let us take another encoding P_1' that results from replacing (4)-(6) in P_1 by rules in Listing 4 respectively corresponding to:

$$\forall y\big(in(a, y) \rightarrow ra(y)\big) \tag{15}$$

$$\forall xy\big(in(x, y) \wedge ra(x) \rightarrow ra(y)\big) \tag{16}$$

$$\forall y\big(\neg ra(y) \wedge vertex(y) \rightarrow \bot\big) \tag{17}$$

Verifying program P_1' amounts to proving its adherence to Π_1 and, for that purpose, requires a proper modularization Π_1' of P_1'. In this case, that modularization is obvious since the change is *local* to the module checking Hamiltonian cycles, Π_{hc}. We define the modular programs $\Pi'_{cn} := \langle\{vertex, in\}, \{(ra : (15) \wedge (16)), (17)\}\rangle$, $\Pi'_{hc} := \langle\{vertex, in\}, \{\Pi'_{cn}, (7), (8)\}\rangle$ and Π_1', as the result of replacing Π_{hc} by Π'_{hc} in Π_1. Even though they use different auxiliary predicates, programs Π_{hc} and Π'_{hc} have the same intuitive meaning (Statement S_{hc}) as long as there exists some vertex a in the graph. One would, therefore, expect that the correctness of Π'_{hc} could be proved by checking some kind of equivalence with respect to Π_{hc}. We formalize next this idea.

Given modular programs Π, Π_1 and Π_2, we write $\Pi[\Pi_1/\Pi_2]$ to denote the result of replacing all occurrences of module Π_1 in Π by Π_2. We also define $\Phi(\Pi - \Pi_1) \stackrel{\text{def}}{=} \bigwedge\{\Phi(M) \mid M \in \mathcal{M}, \Pi_1 \notin \mathit{defmods}(M)\}$. For any finite theory Γ, two modular programs Π_1 and Π_2 are said to be *strongly equivalent with respect to context* Γ when any modular program Π with $\Phi(\Pi - \Pi_1) \models \Gamma$ satisfies that Π and $\Pi[\Pi_1/\Pi_2]$ have the same answer sets.

Theorem 2. *Two modular programs* Π *and* Π' *are strongly equivalent under context* Γ *iff* $\Gamma \models \Phi(\Pi) \leftrightarrow \Phi(\Pi')$ *holds for all Herbrand interpretations.*

In our example, although Π_{hc} and Π'_{hc} are not equivalent in general, we can prove:

Proposition 8. *Modules* Π_{hc} *and* Π'_{hc} *are strongly equivalent w.r.t.* $\Gamma = \{vertex(a)\}$.

Proof sketch. Recall that $\Pi_{hc} \models (8)$. The rest of the proof follows two steps. First, given:

$$\exists r\ \big(\ \Phi(M_3) \wedge \forall y(vertex(a) \wedge vertex(y) \rightarrow r(a, y))\ \big) \tag{18}$$

we get $vertex(a) \models \Phi(\Pi'_{cn}) \leftrightarrow (18)$ and, furthermore, $\models \Phi(\Pi_c) \rightarrow (18)$ follows by instantiation of $\forall x$ with $x = a$. Second, we can prove $(7) \models (18) \rightarrow \Phi(\Pi_c)$. □

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press

7 Conclusions and future work

We presented a modular ASP framework that allows nested modules possibly containing hidden local predicates. The semantics of these programs and their modules is specified via the second-order SM operator, and so, it does not resort to grounding. We illustrated how, under some reasonable conditions, a modular reorganization of a logic program can be used for verifying that it adheres to its (in)formal specification. This method has two important advantages. First, it applies a divide-and-conquer strategy, decomposing the correctness proof for the target program into almost self-evident pieces. Second, it can be used to guarantee correctness of a module replacement, even if interchanged modules are non-ground and use different local predicates. In this way, correctness proofs are also reusable. The need for second-order logic is inherent to the expressiveness of first-order stable models but has the disadvantage of lacking a proof theory in the general case. Yet, there are well-known specific cases in which the second-order quantifiers can be removed. This is often the case of the SM operator so that we can use formal results from the literature (splitting, head-cycle free transformations, relation to Clark's Completion or Circumscription to reduce the these second-order formulas to first-order ones (Ferraris et al. 2011). We also intend to exploit the correspondence between SM and Equilibrium Logic to study general inter-theory relations (Pearce and Valverde 2004).

Our definition of contextual strong equivalence using hidden predicates is a variation of *strong equivalence* (Lifschitz et al. 2001; Lifschitz et al. 2007). We leave it to future work the relation to other program equivalence and correspondence notions (Eiter et al. 2005; Oetsch and Tompits 2008; Oikarinen and Janhunen 2009; Aguado et al. 2019; Geibinger and Tompits 2019). Another topic for future work is the extension of automated reasoning tools for ASP verification (Lifschitz et al. 2018) to incorporate modularity.

Acknowledgments. We are thankful to Vladimir Lifschitz and the anonymous reviewers for their comments that help us to improve the paper. This work was partially supported by MINECO, Spain, grant TIN2017-84453-P and NSF, USA grant 1707371. The second author is funded by the Alexander von Humboldt Foundation.

References

Aguado, F., Cabalar, P., Fandinno, J., Pearce, D., Pérez, G., and Vidal, C. 2019. Forgetting auxiliary atoms in forks. *Artificial Intelligence 275*, 575–601.

Brewka, G., Niemelä, I., and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the ACM 54(12)*, 92–103.

Buddenhagen, M. and Lierler, Y. 2015. Performance tuning in answer set programming. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*.

Cabalar, P., Fandinno, J., and Lierler, Y. 2020. Modular answer set programming as a formal specification language.

Denecker, M., Lierler, Y., Truszczyński, M., and Vennekens, J. 2012. A Tarskian informal semantics for answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP)*. 277–289.

Eiter, T., Tompits, H., and Woltran, S. 2005. On solution correspondences in answer set programming. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, L. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 97–102.

Erdoğan, S. T. and Lifschitz, V. 2004. Definitions in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Springer-Verlag, 114–126.

Ferraris, P. 2005. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 119–131.

Ferraris, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic 12,* 4, 25.

Ferraris, P., Lee, J., and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence 175,* 1, 236–263.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*. MIT Press, 386–392.

Geibinger, T. and Tompits, H. 2019. Characterising relativised strong equivalence with projection for non-ground answer-set programs. In *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*. Lecture Notes in Computer Science, vol. 11468. Springer, 542–558.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080.

Gonçalves, R., Knorr, M., and Leite, J. 2016. You can't always forget what you want: On the limits of forgetting in answer set programming. In *Proceedings of 22nd European Conference on Artificial Intelligence (ECAI'16)*. Frontiers in Artificial Intelligence and Applications, vol. 285. IOS Press, 957–965.

Harrison, A. and Lierler, Y. 2016. First-order modular logic programs and their conservative extensions. *Theory and Practice of Logic programming, 32nd Int'l. Conference on Logic Programming (ICLP) Special Issue*.

Lierler, Y. 2019. Strong equivalence and program's structure in arguing essential equivalence between first-order logic programs. In *Proceedings of the 21st International Symposium on Practical Aspects of Declarative Languages (PADL)*.

Lifschitz, V. 2017. Achievements in answer set programming. *Theory and Practice of Logic Programming 17,* 5-6, 961–973.

Lifschitz, V., Lühne, P., and Schaub, T. 2018. anthem: Transforming gringo programs into first-order theories (preliminary report). *CoRR abs/1810.00453*.

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

Lifschitz, V., Pearce, D., and Valverde, A. 2007. A characterization of strong equivalence for logic programs with variables. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 188–200.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*. MIT Press, 23–37.

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

Monin, J. 2003. *Understanding formal methods*. Springer.

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

Oetsch, J. and Tompits, H. 2008. Program correspondence under the answer-set semantics: The non-ground case. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 591–605.

Oikarinen, E. and Janhunen, T. 2009. A translation-based approach to the verification of modular equivalence. *J. Log. Comput. 19,* 4, 591–613.

Pearce, D. and Valverde, A. 2004. Synonymous theories in answer set programming and equilibrium logic. In *Proceedings of the 16th European Conference on Artificial Intelligence*. ECAI'04. IOS Press, Amsterdam, The Netherlands, The Netherlands, 388–392.

Pearce, D. and Valverde, A. 2008. Quantified equilibrium logic and foundations for answer set programs. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 546–560.

https://doi.org/10.1017/S1471068420000265 Published online by Cambridge University Press