Allowing Cyclic Dependencies in Modular Logic Programming

João Moura([✉]) and Carlos Viegas Damásio

CENTRIA / NOVA Laboratory for Computer Science and Informatics (NOVA-LINCS), Universidade NOVA de Lisboa, Lisbon, Portugal
joaomoura@yahoo.com, cd@fct.unl.pt

Abstract. Even though modularity has been studied extensively in conventional logic programming, there are few approaches on how to incorporate modularity into Answer Set Programming, a prominent rule-based declarative programming paradigm. A major approach is Oikarinnen and Janhunen's Gaifman-Shapiro-style architecture of program modules, which provides the composition of program modules. Their module theorem properly strengthens Lifschitz and Turner's splitting set theorem for normal logic programs. However, this approach is limited by module conditions that are imposed in order to ensure the compatibility of their module system with the stable model semantics, namely forcing output signatures of composing modules to be disjoint and disallowing positive cyclic dependencies between different modules. These conditions turn out to be too restrictive in practice and after recently discussing alternative ways of lifting the first restriction [17], we now show how one can allow positive cyclic dependencies between modules, thus widening the applicability of this framework and the scope of the module theorem.

1 Introduction

Over the last few years, answer set programming (ASP) [2,6,12,15,18] emerged as one of the most important methods for declarative knowledge representation and reasoning. Despite its declarative nature, developing ASP programs resembles conventional programming: one often writes a series of gradually improving programs for solving a particular problem, e.g., optimizing execution time and space. Until recently, ASP programs were considered as integral entities, which becomes problematic as programs become more complex, and their instances grow. Even though modularity is extensively studied in logic programming, there are only a few approaches on how to incorporate it into ASP [1,5,8,19] or other module-based constraint modeling frameworks [11,22]. The research on modular systems of logic program has followed two main-streams [3], one is programming in-the-large where compositional operators are defined in order to combine different modules [8,14,20]. These operators allow combining programs algebraically, which does not require an extension of the theory of logic programs. The other direction is programming-in-the-small [10,16], aiming at enhancing logic programming with scoping and abstraction mechanisms available in other

F. Pereira et al. (Eds.) EPIA 2015, LNAI 9273, pp. 363–375, 2015.
DOI: 10.1007/978-3-319-23485-4_37

programming paradigms. This approach requires the introduction of new logical connectives in an extended logical language. The two mainstreams are thus quite divergent.

The approach of [19] defines modules as structures specified by a program (knowledge rules) and by an interface defined by input and output atoms which for a single module are, naturally, disjoint. The authors also provide a module theorem capturing the compositionality of their module composition operator. However, two conditions are imposed: there cannot be positive cyclic dependencies between modules and there cannot be common output atoms in the modules being combined. Both introduce serious limitations, particularly in applications requiring integration of knowledge from different sources. The techniques used in [5] for handling positive cycles among modules are shown not to be adaptable for the setting of [19].

In this paper we discuss two alternative solutions to the cyclic dependencies problem, generalizing the module theorem by allowing positive loops between atoms in the interfaces of the modules being composed. A use case for this requirement can be found in the following example.

Example 1. Alice wants to buy a safe and inexpensive car; she preselected 3 cars, namely c_1, c_2 and c_3. Her friend Bob says that car c_2 is expensive and Charlie says that car c_3 is expensive. Meanwhile, she consulted two car magazines reviewing all three cars. The first considered c_1 safe and the second considered c_1 to be safe while saying that c_3 may be safe if it has an optional airbag. Furthermore, if a friend declares that a car is expensive, then she will consider it safe. Alice is very picky regarding safety, and so she seeks some kind of agreement between the reviews.

The described situation can be captured with five modules, one for Alice, other three for her friends, and another for each magazine. Alice should conclude that c_1 is safe since both magazines agree on this. Therefore, one would expect Alice to opt for car c_1 since it is not expensive, and it is safe. ■

In summary, the fundamental results of [19] require a syntactic operation to combine modules – basically corresponding to the union of programs –, and a compositional semantic operation joining the models of the modules. The module theorem states that the models of the combined modules can be obtained by applying the semantics of the natural join operation to the original models of the modules – which is compositional.

This paper proceeds in Section 2 with an overview of the modular logic programming paradigm, identifying some of its shortcomings. In Section 3 we discuss alternative methods for lifting the restriction that disallows positive cyclic dependencies. We finish with conclusions and a general discussion.

2 Modularity in Answer Set Programming

Modular aspects of ASP have been clarified in recent years, with authors describing how and when two program parts (modules) can be composed [5,11,19] under the stable model semantics. In this paper, we will make use of Oikarinen and Janhunen's logic program modules defined in analogy to [8] which we review after presenting the syntax of ASP.

Answer Set Programming Logic programs in the ASP paradigm are formed by finite sets of rules r having the following syntax:

$$L_1 \leftarrow L_2, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.\ \ (n \geq m \geq 0)(\mathbf{1})$$

where each L_i is a logical atom without the occurrence of function symbols – arguments are either variables or constants of the logical alphabet.

Considering a rule of the form **(1)**, let $Head_P(r) = L_1$ be the literal in the head, $Body_P^+(r) = \{L_2, \ldots, L_m\}$ be the set with all positive literals in the body, $Body_P^-(r) = \{L_{m+1}, \ldots, L_n\}$ be the set containing all negative literals in the body, and $Body_P(r) = \{L_2, \ldots, L_n\}$ be the set containing all literals in the body. If a program is positive we will omit the superscript in $Body_P^+(r)$. Also, if the context is clear we will omit the subscript mentioning the program and write simply $Head(r)$ and $Body(r)$ as well as the argument mentioning the rule. The semantics of stable models is defined via the reduct operation [9]. Given an interpretation M (a set of ground atoms), the reduct P^M of a program P with respect to M is program $P^M = \{Head(r) \leftarrow Body^+(r) \mid r \in P, Body^-(r) \cap M = \emptyset\}$. An interpretation M is a stable model (SM) of P iff $M = LM(P^M)$, where $LM(P^M)$ is the least model of program P^M.

The syntax of logic programs has been extended with other constructs, namely weighted and choice rules [18]. In particular, choice rules have the following form:

$$\{A_1, \ldots, A_n\} \leftarrow B_1, \ldots B_k, not\ C_1, \ldots, not\ C_m.(n \geq 1)(\mathbf{2})$$

As observed by [19], the heads of choice rules possessing multiple atoms can be freely split without affecting their semantics. When splitting such rules into n different rules

$$\{a_i\} \leftarrow B_1, \ldots B_k, not\ C_1, \ldots, not\ C_m \text{ where } 1 \leq i \leq n,$$

the only concern is the creation of n copies of the rule body

$$B_1, \ldots B_k, not\ C_1, \ldots, not\ C_m.$$

However, new atoms can be introduced to circumvent this. There is a translation of these choice rules to normal logic programs [7], which we assume is performed throughout this paper but that is omitted for readability. We deal only with ground programs and use variables as syntactic place-holders.

2.1 Modular Logic Programming (MLP)

Modules in the sense of [19] are essentially sets of rules with Input/Output interfaces:

Definition 1 (Program Module). *A logic program module $\mathcal{P}$ is a tuple $\langle R, I, O, H \rangle$ where:*

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

The set of atoms in $At_v(\mathcal{P}) = I \cup O$ are considered to be *visible* and hence accessible to other modules composed with $\mathcal{P}$ either to produce input for $\mathcal{P}$ or to make use of the output of $\mathcal{P}$. We use $At_i(\mathcal{P}) = I$ and $At_o(\mathcal{P}) = O$ to represent the input and output signatures of $\mathcal{P}$, respectively. The hidden atoms in $At_h(\mathcal{P}) = At(\mathcal{P}) \backslash At_v(\mathcal{P}) = H$ are used to formalize some auxiliary concepts of $\mathcal{P}$ which may not be sensible for other modules but may save space substantially. The condition $head(R) \notin I$ 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 2. The use case in Example 1 is encoded into the five modules shown here:

$$\begin{array}{ll}
\mathcal{P}_A =< & \{\, buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\
& \quad car(c_1).\quad car(c_2).\quad car(c_3).\}, \\
& \{\, safe(c_1), safe(c_2), safe(c_3), exp(c_1), exp(c_2), exp(c_3)\}, \\
& \{\, buy(c_1), buy(c_2), buy(c_3)\}, \\
& \{\, car(c_1), car(c_2), car(c_3)\} > \\
\mathcal{P}_B =< & \{\, exp(c_2).\}, \{\}, \{exp(c_2), exp(c_3)\}, \{\} > \\
\mathcal{P}_C =< & \{\, exp(c_3).\}, \{\}, \{exp(c_1), exp(c_2), exp(c_3)\}, \{\} > \\
\mathcal{P}_{mg_1} =< & \{ \leftarrow not\ safe(c_1).\quad airbag(C) \leftarrow safe(C).\}, \\
& \{\, safe(C)\}, \\
& \{\, airbag(C)\}, \\
& \{\, \} > \\
\mathcal{P}_{mg_2} =< & \{\, safe(X) \leftarrow car(X), airbag(X). \\
& \quad car(c_1).\ car(c_2).\ car(c_3).\ \leftarrow not\ airbag(c_1).\ \{\leftarrow not\ airbag(c_3)\}.\ \}, \\
& \{\, airbag(C)\}, \\
& \{\, safe(c_1), safe(c_2), safe(c_3)\}, \\
& \{\, airbag(c_1), airbag(c_2), airbag(c_3), car(c_1), car(c_2), car(c_3)\} > \quad \blacksquare
\end{array}$$

In Example 2, module $\mathcal{P}_A$ encodes the rule used by Alice to decide if a car should be bought. The safe and expensive atoms are its inputs, and the buy atoms its outputs; it uses hidden atoms $car/1$ to represent the domain of variables. Modules $\mathcal{P}_B$, $\mathcal{P}_C$ and $\mathcal{P}_{mg_1}$ captures the factual information in Example 1 and depends on input literal $safe$ to determine if its output states that a car has an $airbag$ or not. They have no input and no hidden atoms, but *Bob* has only analyzed the price of cars c_2 and c_3. The ASP program module for the second magazine is more interesting[1], and expresses the rule used to determine if a car

[1] car belongs to both hidden signatures of $\mathcal{P}_A$ and $\mathcal{P}_{mg_2}$ which is not allowed when composing these modules, but for clarity we omit a renaming of the $car/1$ predicate.

is safe, namely that a car is safe if it has an airbag; it is known that car c_1 has an airbag, c_2 does not, and the choice rule states that car c_3 may or may not have an airbag.

Next, the SM semantics is generalized to cover modules by introducing a generalization of the Gelfond-Lifschitz's fixpoint definition. In addition to weakly negated literals (i.e., *not*), also literals involving input atoms are used in the stability condition. In [19], the SMs of a module are defined as follows:

Definition 2 (Stable Models of Modules). *An interpretation $M \subseteq At(\mathcal{P})$ is a SM of an ASP program module $\mathcal{P} = \langle R, I, O, H\rangle$, iff $M = LM\left(R^M \cup \{a.|a \in M \cap I\}\right)$. The SMs of $\mathcal{P}$ are denoted by $AS(\mathcal{P})$.*

Intuitively, the SMs of a module are obtained from the SMs of the rules part, for each possible combination of the input atoms.

Example 3. Program modules $\mathcal{P}_B$, $\mathcal{P}_C$, and $\mathcal{P}_{mg_1}$ have each a single answer set:

$AS(\mathcal{P}_B) = \{\{exp(c_2)\}\}$, $AS(\mathcal{P}_C) = \{\{exp(c_3)\}\}$, and $AS(\mathcal{P}_{mg_1}) = \{\{safe(c_1), airbag(c_1)\}\}$.

Module $\mathcal{P}_{mg_2}$ has two SMs, namely:

$\{safe(c_1), car(c_1), car(c_2), car(c_3), airbag(c_1)\}$, and

$\{safe(c_1), safe(c_3), car(c_1), car(c_2), car(c_3), airbag(c_1), airbag(c_3)\}$.

Alice's ASP program module has $2^6 = 64$ models corresponding each to an input combination of safe and expensive atoms. Some of these models are:

$$\begin{array}{l} \{\ buy(c_1), car(c_1), car(c_2), car(c_3), safe(c_1)\ \} \\ \{\ buy(c_1), buy(c_3), car(c_1), car(c_2), car(c_3), safe(c_1), safe(c_3)\ \} \\ \{\ buy(c_1), car(c_1), car(c_2), car(c_3), exp(c_3), safe(c_1), safe(c_3)\ \} \blacksquare \end{array}$$

2.2 Composing Programs from Models

The composition of models is obtained from the union of program rules and by constructing the composed output set as the union of modules' output sets, thus removing from the input all the specified output atoms. [19] define their first composition operator as follows: Given two modules $\mathcal{P}_1 = \langle R_1, I_1, O_1, H_1\rangle$ and $\mathcal{P}_2 = \langle R_2, I_2, O_2, H_2\rangle$, their composition $\mathcal{P}_1 \oplus \mathcal{P}_2$ is defined when their output signatures are disjoint, that is, $O_1 \cap O_2 = \emptyset$, and they respect each others hidden atoms, i.e., $H_1 \cap At(\mathcal{P}_2) = \emptyset$ and $H_2 \cap At(\mathcal{P}_1) = \emptyset$. Then their composition is

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

However, the conditions given for $\oplus$ are not enough to guarantee compositionality in the case of answer sets and as such they define a restricted form:

Definition 3 (Module Union Operator $\sqcup$). *Given modules $\mathcal{P}_1, \mathcal{P}_2$, their union is $\mathcal{P}_1 \sqcup \mathcal{P}_2 = \mathcal{P}_1 \oplus \mathcal{P}_2$ whenever* **(i)** *$\mathcal{P}_1 \oplus \mathcal{P}_2$ is defined and* **(ii)** *$\mathcal{P}_1$ and $\mathcal{P}_2$ are mutually independent meaning that there are no positive cyclic dependencies among rules in different modules, defined as loops through input and output signatures.*

Natural join ($\bowtie$) on visible atoms is used in [19] to combine the stable models of modules as follows:

Definition 4 (Join). *Given modules $\mathcal{P}_1$ and $\mathcal{P}_2$ and sets of interpretations $A_1 \subseteq 2^{At(\mathcal{P}_1)}$ and $A_2 \subseteq 2^{At(\mathcal{P}_2)}$, the natural join of A_1 and A_2 is:*

$$A_1 \bowtie A_2 = \{\ M_1 \cup M_2 \mid M_1 \in A_1, M_2 \in A_2 \text{ and } M_1 \cap At_v(\mathcal{P}_2) = M_2 \cap At_v(\mathcal{P}_1)\}$$

This leads to their main result, stating that:

Theorem 1 (Module Theorem). *If $\mathcal{P}_1, \mathcal{P}_2$ are modules such that $\mathcal{P}_1 \sqcup \mathcal{P}_2$ is defined, then:*

$$AS(\mathcal{P}_1 \sqcup \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)$$

Still according to [19], their module theorem also straightforwardly generalizes for a collection of modules because the module union operator $\sqcup$ is commutative, associative, and has the identity element $< \emptyset, \emptyset, \emptyset, \emptyset >$.

Example 4. Consider the composition $\mathcal{Q} = (\mathcal{P}_A \sqcup \mathcal{P}_{mg_1}) \sqcup \mathcal{P}_B$. First, we have

$$\mathcal{P}_A \sqcup \mathcal{P}_{mg_1} = \left\langle \begin{array}{l} \{buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\ \ \ car(c_1).\ car(c_2).\ car(c_3).\ safe(c_1).\}, \\ \{exp(c_1), exp(c_2), exp(c_3)\}, \\ \{buy(c_1), buy(c_2), buy(c_3), safe(c_1),\ safe(c_2),\ safe(c_3)\}, \\ \{car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

It is immediate to see that the module theorem holds in this case. The visible atoms of $\mathcal{P}_A$ are $safe/1$, $exp/1$ and $buy/1$, and the visible atoms for $\mathcal{P}_{mg_1}$ are $\{safe(c_1),\ safe(c_2)\}$. The only model for $\mathcal{P}_{mg_1} = \{safe(c_1)\}$ when naturally joined with the models of $\mathcal{P}_A$, results in eight possible models where $safe(c_1)$, $not\ safe(c_2)$, and $not\ safe(c_3)$ hold, and $exp/1$ vary. The final ASP program module $\mathcal{Q}$ is

$$\left\langle \begin{array}{l} \{buy(X) \leftarrow car(X), safe(X), not\ exp(X). \\ car(c_1).\ car(c_2).\ car(c_3).\ exp(c_2).\ safe(c_1).\}, \\ \{exp(c_1)\}, \\ \{buy(c_1), buy(c_2), buy(c_3), exp(c_2), safe(c_1), safe(c_2), safe(c_3)\}, \\ \{car(c_1), car(c_2), car(c_3)\} \end{array} \right\rangle$$

The SMs of $\mathcal{Q}$ are thus:
$\{safe(c_1), exp(c_1), exp(c_2), car(c_1), car(c_2), car(c_3)\}$ and
$\{buy(c_1), safe(c_1), exp(c_2), car(c_1), car(c_2), car(c_3)\}$

2.3 Shortcomings

The conditions imposed in these definitions bring about some shortcomings such as the fact that the output signatures of two modules must be disjoint which disallows many practical applications e.g., we are not able to combine the results of program module $\mathcal{Q}$ with any of $\mathcal{P}_C$ or $\mathcal{P}_{mg_2}$, and thus it is impossible to obtain the combination of the five modules. Also because of this, the module union operator $\sqcup$ is not reflexive. By trivially waiving this condition, we immediately get problems with conflicting modules. The compatibility criterion for the operator $\bowtie$ also rules out the compositionality of mutually dependent modules, but allows positive loops inside modules or negative loops in general. We illustrate this in Example 5, which has been solved recently in [17] and the issue with positive loops between modules in Example 6 .

Example 5 (Common Outputs). Given $\mathcal{P}_B$ and $\mathcal{P}_C$, which respectively have: $AS(\mathcal{P}_B) = \{\{exp(c_2)\}\}$ and $AS(\mathcal{P}_C)=\{\{exp(c_3)\}\}$,
the single SM of their union $AS(\mathcal{P}_B \sqcup \mathcal{P}_C)$ is: $\{exp(c_2), exp(c_3)\}$. However, the join of their SMs is $AS(\mathcal{P}_B) \bowtie AS(\mathcal{P}_C) = \emptyset$, invalidating the module theorem.■

Example 6 (Cyclic Dependencies). Take the following two program modules (a simplification of the magazine modules in Example 2):

$$\mathcal{P}_1 = \langle\{airbag \leftarrow safe.\}, \{safe\}, \{airbag\}, \emptyset\rangle$$
$$\mathcal{P}_2 = \langle\{safe \leftarrow airbag.\}, \{airbag\}, \{safe\}, \emptyset\rangle$$

Their SMs are: $AS(\mathcal{P}_1) = AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$ while the single SM of the union $AS(\mathcal{P}_1 \sqcup \mathcal{P}_2)$ is the empty model $\{\}$. Therefore $AS(\mathcal{P}_1 \sqcup \mathcal{P}_2) \neq AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$, also invalidating the module theorem.
■

3 Positive Cyclic Dependencies Between Modules

To attain a generalized form of compositionality we need to be able to deal with both restrictions identified previously and in particular cyclic dependencies between modules. In the literature, [5] presents a solution based on a model minimality property. It forces one to check for minimality on every comparable models of all program modules being composed. It is not applicable to our setting though, which can be seen in Example 7 where logical constant $\bot$ represents value $false$. Example 7 shows that [5] is not compositional in the sense of Oikarinen and Janhunen.

Example 7. Given modules $\mathcal{P}_1 = \langle\{a \leftarrow b.\ \bot \leftarrow not\ b.\}, \{b\}, \{a\}, \{\}\rangle$ with one SM $\{a, b\}$, and $\mathcal{P}_2 = \langle\{b \leftarrow a.\}, \{a\}, \{b\}, \{\}\rangle$ with SMs $\{\}$ and $\{a, b\}$, their composition has no inputs and no intended SMs while their minimal join contains $\{a, b\}$. ■

Another possible solution requires the introduction of extra information in the models to allow detecting mutual positive dependencies. This route has been identified before [21] and is left for future work.

3.1 Model Minimization

We present a model join operation that requires one to look at every model of both modules being composed in order to check for minimality on models comparable on account of their inputs. However, this operation is able to distinguish between atoms that are self supported through positive loops and atoms with proper support, allowing one to lift the condition in Definition 3 disallowing positive dependencies between modules.

Definition 5 (Minimal Join). *Given modules $\mathcal{P}_1$ and $\mathcal{P}_2$, let their composition be $\mathcal{P}_C = \mathcal{P}_1 \oplus \mathcal{P}_2$. Define $AS(\mathcal{P}_1) \bowtie^{min} AS(\mathcal{P}_2) = \{M \mid M \in AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)$ such that $\nexists_{M' \in AS(\mathcal{P}_1) \bowtie AS(\mathcal{P}_2)} : M' \subset M$ and $M \cap At_i(\mathcal{P}_C) = M' \cap At_i(\mathcal{P}_C)\}$*

Example 8 (Minimal Join). A car is safe if it has an airbag and it has an airbag if it is safe and the airbag is an available option. This is captured by two modules, namely: $\mathcal{P}_1 = \langle\{airbag \leftarrow safe, available\_option.\}, \{safe, available\_option\}, \{airbag\}, \emptyset\rangle$ and $\mathcal{P}_2 = \langle\{safe \leftarrow airbag.\}, \{airbag\}, \{safe\}, \emptyset\rangle$ which respectively have $AS(\mathcal{P}_1) = \{\{\}, \{safe\}, \{available\_option\}, \{airbag, safe, available\_option\}\}$ and $AS(\mathcal{P}_2) = \{\{\}, \{airbag, safe\}\}$. The composition has as its input signature $\{available\_option\}$ and therefore its answer set {airbag,safe,available_option} is not minimal regarding the input signature of the composition because $\{available\_option\}$ is also a SM (and the only intended model among these two). Thus $AS(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie^{min} AS(\mathcal{P}_2) = \{\{\}, \{available\_option\}\}$. ■

This join operator allows us to lift the prohibition of composing mutually dependent modules under certain situations. Integrity constraints containing only input atoms in their body are still a problem with this approach as these would exclude models that would otherwise be minimal in the presence of unsupported loops.

Theorem 2 (Minimal Module Theorem). *If $\mathcal{P}_1, \mathcal{P}_2$ are modules such that $\mathcal{P}_1 \oplus \mathcal{P}_2$ is defined (allowing cyclic dependencies between modules), and that only normal rules are used in modules, then:*

$$AS(\mathcal{P}_1 \oplus \mathcal{P}_2) = AS(\mathcal{P}_1) \bowtie^{min} AS(\mathcal{P}_2)$$

3.2 Annotated Models for Composing Mutualy Dependent Modules

Because the former operator is not general and it forces us to compare one model with every other model for minimality, thus it is not local, we present next an alternative that requires adding annotations to models. We start by looking at positive cyclic dependencies (loops) that are formed by composition. It is known from the literature (e.g. [21]) that in order to do without looking at the rules of the program modules being composed, which in the setting of MLP we assume not having access to, we need to have extra information incorporated into the models.

Definition 6 (Dependency Transformation). *Let $\mathcal{P}$ be an MLP. Its dependency transformation is defined as the set of rules $(R_{\mathcal{P}})^A$ obtained from $R_{\mathcal{P}}$ by replacing each clause $L_1 \leftarrow L_2, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n.(n \neq m)$ in $R_{\mathcal{P}}$ with the following clause, where $n \neq m$ and $D = D_2 \cup \ldots \cup D_m$ is a set of dependency sets:*

$$(1) \quad L_1 : D \leftarrow L_2 : D_2, \ldots, L_m : D_m, not\ L_{m+1} : D_{m+1}, \ldots, not\ L_n : D_n.$$

Definition 7 (Annotated Model). *Given a module $\mathcal{P} = \langle R^A, I, O, H \rangle$, its set of annotated models is constructed as before: An interpretation $M \subseteq At^A(\mathcal{P})$ is an annotated answer set of an ASP program module $\mathcal{P} = \langle R, I, O, H \rangle$, if and only if:*

$$M = LM\left((R^A)^M \cup \{a : \{\{a\}\}. \mid a : D \in M \cap I\}\right),$$

where $(R^A)^M_I$ is the version of the Gelfond-Lifschitz reduct allowing weighted and choice rules, of the dependency transformation of R, and LM is the operator returning the least model of the positive program argument. The set of annotated stable models of $\mathcal{P}$ is denoted by $AS^A(\mathcal{P})$.

Semantic of Annotated Programs. An annotated interpretation maps every atom into a set of subsets of input atoms, tracking the dependencies of the atom in combinations of input atoms. The semantics of annotated programs is obtained by iterating an immediate consequences monotonic operator applied to a definite program, defined as follows:

$$T_P(I)(L_1) = \bigcup \left\{T_P(I)(L_2) \cup \ldots \cup T_P(I)(L_m) \mid L_1 \leftarrow L_2, \ldots L_m \in R^A\right\}$$

starting from the interpretation mapping every atom into the empty set. In order to consider input atoms in modules we set $I(a) = \{\{a\}\}$ for every $a \in M \cap I$, and $\{\}$ otherwise.

Collapsed Annotated Models. Previous Definition 7 generates equivalent models for each alternative rule where atoms from the model belong to the head of the rule. We need to merge them into a collapsed annotated model where the alternatives are listed as sets of annotations, in order to retain a one to one correspondence between these and the SMs of the original program. As we are only interested in this collapsed form, we will henceforth take collapsed annotated models as annotated models.

Definition 8 (Collapsed Annotated Model). *Let M' and M'' be two annotated models such that for every atom $a \in M'$, it is also the case that $a \in M''$ and vice-versa. A collapsed annotated model M of M' and M'' is constructed as follows:*

$$M = \{a : \{D', D''\} \mid a : D' \in M' and\ a : D'' \in M''\}$$

Given a module $\mathcal{P}$, a program $P(M)$ can conversely be constructed from one of the module's annotated models M simply by adding rules of the form $a \leftarrow D_1, \ldots, D_m$. for each annotated atom $a_{\{D_1,\ldots,D_m\}} \in M$. Such constructed program $P(M)$ will be equivalent (but not strongly equivalent [13]) to taking the original program and adding facts that belong to the annotated model M, intersected with the input signature of $\mathcal{P}$, correspondingly.

Example 9 (Annotated Model). Let $\mathcal{P} = \langle\{a \leftarrow b, c.\ \ b \leftarrow d, not\ e, not\ f.\}, \{d, f\}, \{a, b\}, \emptyset\rangle$ be a module. $\mathcal{P}$ has one annotated model as per Definition 7: $\{b_{\{\{d\}\}}, d\}$. ■

In the previous example, the first rule $a \leftarrow b, c.$ can never be activated because c is not an input atom ($c \notin I$) and it is not satisfied by the rules of the module ($R_{\mathcal{P}} \not\models c$). Thus, the only potential positive loop is identified by $\{b_{\{d\},d}\}$. If we compose $\mathcal{P}$ with a module containing e.g., rule $d \leftarrow b$. and thus with an annotated model $\{d_{\{b\}}, b\}$, then it is easy to identify this as being a loop and if any atoms in the loop are satisfied by the module composition then there will be a stable model reflecting that. Also notice that since e is not a visible atom, it does not interfere with other modules, as long as it is respected, and thus it does not need to be in the annotation.

Cyclic Compatibility. We define next the compatibility of mutually dependent models. We assume that the outputs are disjoint as per the original definitions. The compatibility is defined as a two step criterion. The first is similar to the original compatibility criterion, only adapted to dealing with annotated models by disregarding the annotations. This first step makes annotations of negative dependencies unnecessary. The second step takes models that are compatible according to the first step and, after reconstructing two possible programs from the compatible annotated models, implies computing the minimal model of the union of these reconstructed programs and see if the union of the compatible models is a model of the union of the reconstructed programs.

Definition 9 (Basic Model Compatibility). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two modules. Let $AS^A(\mathcal{P}_1)$, respectively $AS^A(\mathcal{P}_2)$ be their annotated models. Let now $M_1 \in AS^A(\mathcal{P}_1)$ and $M_2 \in AS^A(\mathcal{P}_2)$ be two models of the modules, they will be compatible if:*

$$M_1 \cap At_v(\mathcal{P}_2) = M_2 \cap At_v(\mathcal{P}_1)$$

Now, for the second step of the cyclic compatibility criterion one takes models that passed the basic compatibility criterion and reconstruct their respective possible programs as defined previously. Then one computes the minimal model of the union of these reconstructed programs and see if the union of the originating models is a model of the union of their reconstructed programs.

Definition 10 (Annotation Compatibility). *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two modules. Let $AS^A(\mathcal{P}_1)$, respectively $AS^A(\mathcal{P}_2)$ be their annotated models. Let now $M_1 \in AS^A(\mathcal{P}_1)$ and $M_2 \in AS^A(\mathcal{P}_2)$ be two compatible models according to Definition 9. They will be compatible annotated models if $AS^A(\mathcal{P}_1 \cup \mathcal{P}_2) = M_1 \cup M_2$.*

3.3 Attaining Cyclic Compositionality

After setting the way by which one can deal with positive loops by using annotations in models, the join operator needs to be redefined. The original composition operators are applicable to annotated modules after applying Definition 6. This way, their atoms positive dependencies are added to their respective models.

Definition 11 (Modified Join). *Given two compatible annotated (in the sense of Definition 10) modules $\mathcal{P}_1, \mathcal{P}_2$, their composition is $\mathcal{P}_1 \otimes \mathcal{P}_2 = \mathcal{P}_1 \oplus \mathcal{P}_2$ provided that (i) $\mathcal{P}_1 \oplus \mathcal{P}_2$ is defined. This way, given modules $\mathcal{P}_1$ and $\mathcal{P}_2$ and sets of annotated interpretations $A_1^A \subseteq 2^{A_t(\mathcal{P}_1)}$ and $A_2^A \subseteq 2^{A_t(\mathcal{P}_2)}$, the natural join of A_1^A and A_2^A, denoted by $A_1^A \bowtie_A A_2^A$, is defined as follows for intersecting output atoms:*

$$\{M_1 \cup M_2 |\ M_1 \in A_1, M_2 \in A_2, s.t.\ M_1 \textit{ and } M_2 \textit{ are compatible.}\}$$

Theorem 3 (Cyclic Module Theorem). *If $\mathcal{P}_1, \mathcal{P}_2$ are modules with annotated models such that $\mathcal{P}_1 \sqcup \mathcal{P}_2$ is defined, then:*

$$AS^A(\mathcal{P}_1 \sqcup \mathcal{P}_2) = AS^A(\mathcal{P}_1) \bowtie_A AS^A(\mathcal{P}_2)$$

3.4 Shortcomings Revisited

By adding the facts contained in stable models of one composing module to the other composing module, through a program transformation, one is able to counter the fact that the inputs of the composed module are removed if they are met by the outputs of either composing modules [17]. As for positive loops, going back to Example 6, the new composition operator also produces desired results:

Example 10 (Cyclic Dependencies Revisited). Take again the two program modules in Example 6:

$$\mathcal{P}_1 = \langle \{airbag \leftarrow safe.\}, \{safe\}, \{airbag\}, \emptyset \rangle$$
$$\mathcal{P}_2 = \langle \{safe \leftarrow airbag.\}, \{airbag\}, \{safe\}, \emptyset \rangle$$

which respectively have annotated models $AS^A(\mathcal{P}_1)= \{\{\}, \{airbag_{\{safe\}}, safe\}\}$ and $AS^A(\mathcal{P}_2)=\{\{\},\{airbag, safe_{\{airbag\}}\}\}$ while $AS^A(\mathcal{P}_1 \otimes \mathcal{P}_2) = \{\{\}, \{airbag_{\{safe\}}, safe_{\{airbag\}}\}\}$. Because of this, $AS^A(\mathcal{P}_1 \otimes \mathcal{P}_2) = AS^A(\mathcal{P}_1) \bowtie_A AS^A(\mathcal{P}_2)$. Now, take $\mathcal{P}_3 = \langle \{airbag.\}, \{\}, \{airbag\}, \emptyset \rangle$ and compose it with $\mathcal{P}_1 \otimes \mathcal{P}_2$. We get $AS^A(\mathcal{P}_1 \otimes \mathcal{P}_2 \otimes \mathcal{P}_3)= \{\{airbag, safe\}\}$. ■

4 Conclusions and Future Work

We lift the restriction that disallows composing modules with cyclic dependencies in the framework of Modular Logic Programming [19]. We present a model join operation that requires one to look at every model of two modules being composed in order to check for minimality of models that are comparable on account of their inputs. This operation is able to distinguish between atoms that are self supported through positive loops and atoms with proper support, allowing one to lift the condition disallowing positive dependencies between modules. However, this approach is not local as it requires comparing every models and, as it is not general because it does not allow combining modules with integrity constraints, it is of limited applicability.

Because of this lack of generality of the former approach, we present an alternative solution requiring the introduction of extra information in the models for one to be able to detect dependencies. We use models annotated with the way they depend on the atoms in their module's input signature. We then define their semantics in terms of a fixed point operator. After setting the way by which one deals with positive loops by using annotations in models, the join operator needs to be redefined. The original composition operators are applicable to annotated modules after applying Definition 7. This way, their positive dependencies are added to their respective models. This approach turns out to be local, in the sense that we need only look at two models being joined and unlike the first alternative we presented, it works well with integrity constraints.

As future work we can straightforwardly extend these results to probabilistic reasoning with ASP by applying the new module theorem to [4], as well as to DLP functions and general stable models. An implementation of the framework is also foreseen in order to assess the overhead when compared with the original benchmarks in [19].

Acknowledgments. The work of João Moura was supported by grant SFRH/BD/69006/2010 from Fundação para a Ciência e Tecnologia (FCT) from the Portuguese Ministério do Ensino e da Ciência.

References

1. Babb, J., Lee, J.: Module theorem for the general theory of stable models. TPLP **12**(4–5), 719–735 (2012)
2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (2003)
3. Bugliesi, M., Lamma, E., Mello, P.: Modularity in logic programming. J. Log. Program. **19**(20), 443–502 (1994)
4. Viegas Damásio, C., Moura, J.: Modularity of P-log programs. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 13–25. Springer, Heidelberg (2011)
5. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular nonmonotonic logic programming revisited. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 145–159. Springer, Heidelberg (2009)
6. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Computing preferred and weakly preferred answer sets bymeta-interpretation in answer set programming. In: Proceedings AAAI 2001 Spring Symposium on Answer Set Programming, pp. 45–52. AAAI Press (2001)

7. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. TPLP **5**(1–2), 45–74 (2005)
8. Gaifman, H., Shapiro, E.: Fully abstract compositional semantics for logic programs. In: Symposium on Principles of Programming Languages, POPL, pp. 134–142. ACM, New York (1989)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the 5th International Conference on Logic Program. MIT Press (1988)
10. Giordano, L., Martelli, A.: Structuring logic programs: a modal approach. The Journal of Logic Programming **21**(2), 59–94 (1994)
11. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 155–168. Springer, Heidelberg (2009)
12. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(1–2), 39–54 (2002)
13. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic **2**, 2001 (2000)
14. Mancarella, P., Pedreschi, D.: An algebra of logic programs. In: ICLP/SLP, pp. 1006–1023 (1988)
15. Marek, V.W., Truszczynski, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: A 25-Year Perspective (1999)
16. Miller, D.: A theory of modules for logic programming. In. In Symp. Logic Programming, pp. 106–114 (1986)
17. Moura, J., Damásio, C.V.: Generalising modular logic programs. In: 15th International Workshop on Non-Monotonic Reasoning (NMR 2014) (2014)
18. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**, 72–79 (1998)
19. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory Pract. Log. Program. **8**(5–6), 717–761 (2008)
20. O'Keefe, R.A.: Towards an algebra for constructing logic programs. In: SLP, pp. 152–160 (1985)
21. Slota, M., Leite, J.: Robust equivalence models for semantic updates of answer-set programs. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) Proc. of KR 2012. AAAI Press (2012)
22. Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 259–274. Springer, Heidelberg (2011)