Declarative Description of Module Dependencies in Logic Programming

Jozef Šiška

KAI FMFI UK, Mlynská dolina, 842 48 Bratislava*
`siska@ii.fmph.uniba.sk`

Abstract. Logic Programming and specifically Answer Set Programming are formalisms for knowledge representation and reasoning based on classical logic. They allow a clean and declarative characterization of many KR based problems.
Modularity is a key aspect in all programming languages. Every language allows authors to split a program into multiple modules, thus gaining advantages such as code simplification or reusability.
We consider modules as a way of reducing the complexity of logic programs by allowing authors to declaratively describe which modules should be used in which situations. Such information is represented directly in the language of logic programs. Two main abilities of such a modularization system are considered: the ability to express dependencies between modules and to define conditions when a particular module should be automatically included. We focus on the problem of selecting which modules to use in computations without making assumptions on internal structure or interaction of modules.

1 Introduction

Modularization is an interesting concept in logic programming. In the simplest way modules represented by different logic programs are just joined together to form one big logic program. Theoretical research has been done in different directions regarding modularization, whether to resolve conflicts between modules or to define certain interfaces for modules to improve communication between modules [GS89,BLM94,EGV97,BMPT94,JOTW07].

Our aim is to create a system that, given a set of modules that could be combined together into an ASP application, allows authors to declaratively specify which of the modules should be really used in the computation depending on the module dependencies or the data that is processed.

Two main abilities of such a modularization system are considered:

– express dependencies between modules,

* We acknowledge support from VEGA project of Slovak Ministry of Education and Slovak Academy of Sciences no. 1/0688/10 and Comenius University grant no. UK/486/2010.

– define conditions when a particular module should be automatically included.

The dependency requirement covers cases when a module requires another module to be present (possibly only when some conditions are met) to work correctly. This is similar to import or require statements in most languages. On the other hand the condition based inclusion of modules can be used to automatically load a module when certain data are present.

Two approaches are considered here: a declarative (stable model-like) approach and a simple two-level approach. Both approaches are based on the introduction of a special predicate, say *load_module*/1, whose presence in a stable model of a program states that a specific module should be loaded.

In the first approach we choose a subset of the modules, compute its stable models and select the ones which (through the added predicate) select exactly this subset of modules. A modification of this approach is also considered, where certain modules (initial modules) have to be always included in the program.

In the second approach we split the modules into two parts: an activator and a main part. First a program is formed from the activator parts of all modules and its stable models are used to select which modules to include in the second, main, program. Stable models of this second program are then the stable models of the whole modular program.

As mentioned earlier our focus is on the selection of the modules that should be used in the computation. We do not assume any specific way of combining modules (logic programs) and obtaining the stable models of the modular program. Throughout the examples given here we use simple union of the programs, but other approaches [OJ06] can be used too. We require only a way of determining the stable models of an arbitrary (sub-)set of modules. All presented approaches are based on this assumption.

2 Logic Programming

In this paper we consider *Normal Logic Programs*[GL88]. Let $\mathcal{A}$ be a set a disjoint set of constants, predicate, function and variable symbols, where both predicate and function symbols have an associated arity.

A *term* over $\mathcal{A}$ is inductively defined as: *(i)* a variable from $\mathcal{A}$, *(ii)* a constant from $\mathcal{A}$, *(iii)* an expression of the form $f(t_1, \dots, t_n)$, where f is a function symbol from $\mathcal{A}$ with arity n and t_i's are terms over $\mathcal{A}$. A term is called *ground* if it does not contain any variables. The set of all ground terms over $\mathcal{A}$ is called the *Herbrand universe* of $\mathcal{A}$.

An *atom* over $\mathcal{A}$ is an expression of the form $p(t_1, \dots, t_n)$, where p is an predicate symbol of $\mathcal{A}$ with arity n and t_i's are terms over $\mathcal{A}$. An atom is called *ground* if all terms t_i are ground. The set of all ground atoms over $\mathcal{A}$ is called *Herbrand base* of $\mathcal{A}$ ($HB(\mathcal{A})$).

An *objective literal* is an atom A. A *default literal* is an objective literal preceded with the default negation i.e. **not** A. We denote $\mathcal{L}$ the set of both objective and default literals. We call L and **not** L *conflicting* literals. For a set of literals $M \subseteq \mathcal{L}$ we denote M^+ the set of all objective literals and M^- the set of all default literals from M. A set of literals M is called *consistent* if it does not contain two conflicting literals, otherwise it is *inconsistent*.

A *normal logic program* P is a countable set of *rules* of the form $A \leftarrow L_1, L_2, \ldots, L_n$, where L is an objective literal and each of L_i are literals. When all literals are objective, P is called a *definite logic program*.

For a rule r of the form $L \leftarrow L_1, L_2, \ldots, L_n$ we call L the *head* of r and denote by $\text{head}(r)$. Similarly by $\text{body}(r)$ we denote the set $\{L_1, L_2, \ldots, L_n\}$ and call it the *body* of r. If $\text{body}(r) = \emptyset$ then r is called a *fact*.

An *interpretation* is any consistent set of ground literals I. I is called a *total* interpretation if for each $A \in HB(\mathcal{A})$ either $A \in I$ or **not** $A \in I$. A literal L is *satisfied* in an interpretation I ($I \models L$) if $L \in I$. A set of literals $S \subset \mathcal{L}$ is satisfied in I ($I \models S$) if each literal $L \in S$ is satisfied in I. A rule r is satisfied in I ($I \models r$) if $I \models \text{body}(r)$ implies $I \models \text{head}(r)$. A total interpretation M is called a model of logic program P if for each rule $r \in P$ $M \models r$. We also say that M *models* P and write $M \models P$.

Let P be a normal logic program and I an interpretation. The Gelfond-Lifschitz reduct of P w.r.t. I is the program P^I obtained from P by:

1. removing from P all rules containing a default literal **not** A such that $A \in I$.
2. removing all default literals from remaining rules.

Since the resulting program P^I is a definite program, it has a unique least model. We denote it by $\text{least}\left(P^I\right)$. A model M of a normal logic program P is a *stable model* of P iff $M = \text{least}\left(P^I\right)$.

3 Modularization

We start with the common definitions used in both approaches, which allow us to define what is the semantics of a set of modules. We will not restrict ourselves to a fixed definition of a module or of the semantics of a modular program. Instead we will just assume a mapping that produces a set of stable models for a given set of modules. Our goal will then be to find an expressive, yet practical way to define which modules should be used to form the final program in a particular computation.

Definition 1. *(Modular program) Let $Names$ be the set of all possible module names. Let $\mathcal{L}^{Mod}$ be the language of logic programs $\mathcal{L}$ extended with the special predicate load_module and constants from $Names$.*

A modular program *is a set of named LP* modules

$$\mathcal{P} = \{P_{name} \mid name \in N\}$$

such that $N \subseteq Names$.

We consider a modular program to be just a collection of named modules. A module can take any form, as long as we have the means to determine stable models of arbitrary sets of modules:

Definition 2. *Given a set of LP modules $\mathcal{P}$, we denote the set of its stable models by*

$$\mathrm{Sem}\,(\sqcup\mathcal{P})$$

Because we want to use the logic programs themselves to encode which modules should be "active" we will usually extract this information from stable models or interpretations. For this we use the special predicate *load_module*/1.

If the predicate *load_module*(*modname*) holds in an interpretation we are interested in, we take this to mean, that the module named *modname* should be selected when we form the desired modular program.. We therefore introduce the notion of a *subprogram selected by an interpretation.* Note that in our definition, a subprogram is also a modular program and thus we can compute its stable models.

Definition 3. *Let $\mathcal{P}$ be a modular program and I an interpretation. We define the sets of module names selected(skipped) by I as*

$$Sel(I) = \{name \mid name \in Names \wedge I \models load\_module(name)\}$$
$$Skip(I) = \{name \mid name \in Names \wedge I \not\models load\_module(name)\}$$

Further we define the sub-program of $\mathcal{P}$ selected by I as

$$\mathcal{P}^I = \{P_{name} \mid name \in Sel(I)\}$$

The definition leaves the opportunity to use more advanced approaches to modularization, such as preferences or input/output specifications for modules. We will however consider only very simple method of modularization throughout our examples and intuitions. We will assume *modules* to be logic programs and $\mathrm{Sem}(\sqcup\{P_i\})$ to be defined as the standard stable model semantics of the union of the modules:

$$\mathrm{Sem}(\sqcup\{P_i\}) = \mathrm{Sem}(\bigcup\{P_i\})$$

4 Declarative Semantics

In the first approach we provide a declarative definition of the stable models of a modularized program. This definition is very similar to the usual definition of stable models: we *assume* a certain subset of the modules and if, by using these modules, we can derive exactly this selection of modules then we have found a stable model.

Given an interpretation, we select the relevant modules denoted by a special predicate according to definition 3. We then determine the sub-program selected by this interpretation, compute its stable models and check for our interpretation.

Definition 4. *An interpretation I is a stable model of a modular logic program $\mathcal{P}$ iff*

$$I \in \mathrm{Sem}(\sqcup \mathcal{P}^I).$$

Corollary 1. *Let $\mathcal{P}$ be a modularized logic program. Then the empty model $M = \emptyset$ is a stable model of $\mathcal{P}$.*

One drawback of this approach is that modularized logic programs always have a trivial stable model $M = \emptyset$. A possible approach to avoid this is to require certain initial modules to be always included in $Mod(I, \mathcal{P})$. We do this by defining the semantics of a modular logic program with an initial set of modules and force these modules to be present in each model.

Definition 5. *An interpretation I is a stable model of a modular logic program $\mathcal{P}$ with initial modules $\mathcal{I} \subseteq \mathcal{P}$ iff*

$$I \in Sem\left(\sqcup \mathcal{P}^{I \cup \{load\_module(name) | P_{name} \in \mathcal{I}\}}\right) \wedge \mathcal{I} \subseteq Sel(I).$$

The initial modules are always taken into account and they can *require* other modules to be present in the computation when conditions are met. Modules can also *support* themselves, thus enabling them to be included in the computation, this however creates dual models where the modules may or may not be included if no external requirements are present.

Example 1. Consider the following modular program

$$\mathcal{P} = \{P_{data}, P_{base}, P_{home}, P_{school}\}$$

$$P_{data} = \left\{ \begin{array}{c} at(home).\ task(homework).\ load\_module(data). \\ location(home).\ location(school). \end{array} \right\}$$

$$P_{base} = \left\{ \begin{array}{c} load\_module(x) \leftarrow at(x), location(x). \\ load\_module(base). \end{array} \right\}$$

$$P_{home} = \left\{ \begin{array}{c} do(X) \leftarrow task(X) \end{array} \right\}$$

$$P_{school} = \left\{ \begin{array}{c} load\_module(school) \leftarrow task(X), school\_related(X). \\ done(X) \leftarrow do(X), school\_related(X). \\ school\_related(homework). \end{array} \right\}$$

The first module contains data (e.g. a game or application state). The second module servers as an initial module that decides which modules should be used in which situations. The *home* module describes what to do in a certain situation. The last module tries to additionally force itself to be considered when it is needed.

There are five stable models of $\mathcal{P}$:

$$M_1 = \{\}$$

$$M_2 = \{load\_module(base).\}$$

$$M_3 = \left\{ \begin{array}{c} load\_module(data). \\ at(home).\ task(homework). \\ location(home).\ location(school). \end{array} \right\}$$

$$M_4 = M_2 \cup M_3 \cup \left\{ \begin{array}{c} load\_module(home). \\ do(homework). \end{array} \right\}$$

$$M_5 = M_4 \cup \left\{ \begin{array}{c} load\_module(school). \\ school\_related(homework).done(homework). \end{array} \right\}$$

There are two stale models of $\mathcal{P}$ with initial modules *data* and *home*: M_4 and M_5.

If we assume modules to be logic programs and the semantics of a modular program to be the stable model semantics of the set-join of the modules, we can transform a modular logic program into an equivalent normal logic program.

Definition 6. *Let $\mathcal{P} = \{P_n \mid n \in Names\}$ be a modular logic program, where each P_i is a normal logic program that does not contain the predicate $skipped(X)$. Let $Tr(P_n)$ be a logic program that consists of all the rules from P_n with* **not** *$skipped(n)$ appended to the body. Let $Choose(\mathcal{P})$ be a program containing the rules*

$$skipped(n) \leftarrow \textbf{not}\ load\_module(n).$$

And finally let the program $Tr(\mathcal{P})$ be defined as

$$Tr(\mathcal{P}) = Choose(\mathcal{P}) \cup \bigcup_{n \in Names} Tr(P_n)$$

Theorem 1. *Let $\mathcal{P} = \{P_n \mid n \in Names\}$ be a modular logic program, where each P_i is a normal logic program. Then M is a stable model of $\mathcal{P}$ (i.e. it is a stable model of the normal logic program $\bigcup(\mathcal{P}^M)$) iff M' is stable model of $Tr(\mathcal{P})$, where $M' = M \cup \{skipped(n) \mid n \in Skip(M)\}$.*

Proof. Let I be an interpretation such that $I \not\models load\_module(n) \rightarrow I \models skipped(n)$ for each $n \in Names$. The GL reducts of the joined sub-program $(\bigcup(\mathcal{P}^I))^I$ and the transformed modules (without $Choose(\mathcal{P})$) $\bigcup Tr(P_n))^I$ are equivalent because:

– Rules from modules not present in the subprogram $\mathcal{P}^I$, i.e. the ones for which $I \not\models load\_modules(n)$, are excluded from GL reducts of the transformed modules because they contain **not** $skipped(n)$ in their bodies and $I \models skipped(n)$.

– For other rules the literal **not** $\mathit{skipped}(n)$ is removed from the transformed rules reverting them to their original form.

The GL reduct of $\mathit{Choose}(\mathcal{P})^{I'} = \mathit{Choose}(\mathcal{P})^{I}$ is $\{\mathit{skipped}(n).\mid n \in \mathit{Skip}(I)\}$. Furthermore let I and I' be interpretations such that $I' = I \cup \{\mathit{skipped}(n) \mid n \in \mathit{Skip}(I)\}$. Then $(\bigcup(\mathcal{P}^{I}))^{I} = (\bigcup(\mathcal{P}^{I'}))^{I'}$ are identical because none of the P_n contain $\mathit{skipped}(n)$, and $\mathit{Skip}(I) = \mathit{Skip}(I')$as well as $\mathit{Sel}(I) = \mathit{Sel}(I')$. Therefore $I' = \mathrm{least}(\bigcup(Tr(P_n)^{I'}) \cup \mathit{Choose}(\mathcal{P})^{I'})$ iff $I = \mathrm{least}(\bigcup(Tr(P_n)^{I'}))$.

Let M be a stable model of $\mathcal{P}$ and $M' = M \cup \{\mathit{skipped}(n) \mid n \in \mathit{Skip}(M)\}$. M' is therefore a stable model of $Tr(\mathcal{P}) = \mathit{Choose}(\mathcal{P}) \cup \bigcup_{n \in \mathit{Names}} Tr(P_n)$ because

$$\begin{aligned}
&\mathit{least}\left(Tr(\mathcal{P})^{M'}\right) = \\
&= \mathrm{least}\left(\bigcup Tr(P_n))^{M'} \cup \{\mathit{skipped}(n).\mid n \in \mathit{Skip}(M')\}\right) = \\
&= \mathrm{least}(\bigcup Tr(P_n))^{M'}) \cup \mathrm{least}(\{\mathit{skipped}(n).\mid n \in \mathit{Skip}(M')\}) = \\
&= M \cup \{\mathit{skipped}(n) \mid n \in \mathit{Skip}(M')\} = M'.
\end{aligned}$$

Conversely let M' be a stable model of $Tr(\mathcal{P})$. Then there is M such that $M' = M \cup \{\mathit{skipped}(n) \mid n \in \mathit{Skip}(M)\}$. Then $\mathrm{least}((\bigcup(\mathcal{P}^{M}))^{M}) = \mathrm{least}((\bigcup(\mathcal{P}^{M'}))^{M'}) = \mathrm{least}(Tr(P_n)^{M'})$ which is M.

5 Two Step Semantics

The declarative approach is not very suited for practical applications. We therefore present a second approach where a module consists of two programs: a main program and an activator program as shown in fig. 1. First the activator programs from all modules are put together and executed. The stable models are used to decide which modules should be loaded. In the second step the main programs of selected modules are added together and the query is executed as shown in fig. 2.

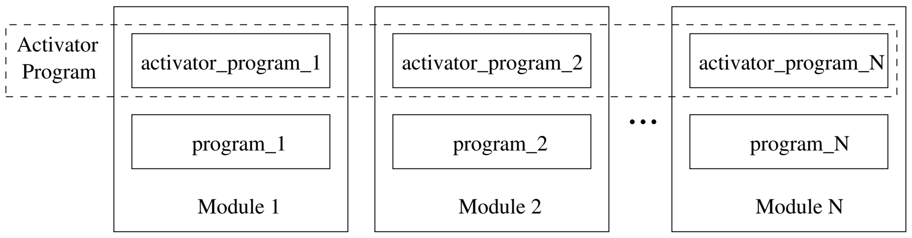

Fig. 1. Modules

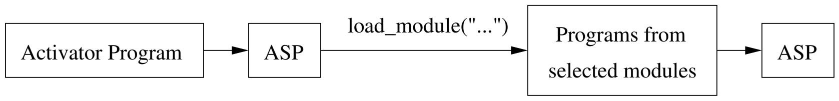

Fig. 2. Modules Processing

We start with the definition of a *split module* that consists of two LP modules: an *activator* module and a *main* module. A *split modular program* is then a set of named split modules. For a split modular program we also define the sets of all activator and main programs, which are in fact modular programs of definition 1

Definition 7. *(Split module) A* split *module is an ordered pair* $M_{name} = (A_{name}, P_{name})$*, where* $name \in Names$ *is an unique identifier of the module and* A_{name} *and* P_{name} *are modules.* A_{name} *and* P_{name} *are called* activator *and* main *program respectively.*

Definition 8. *(Split modular program) Let* $N \subseteq Names$*. A* split modular program *is a set of split modules*

$$D = \{(A_{name}, P_{name}) \mid name \in N\}.$$

We denote $Act(D)$ *(*$Main(D)$*) the modular program consisting of all activator (main) modules and call it the activator (main) program respectively:*

$$Act(D) = \{A_{name} \mid (A_{name}, P_{name}) \in D\}$$
$$Main(D) = \{P_{name} \mid (A_{name}, P_{name}) \in D\}$$

.

To define the semantics of split modular programs we start by computing the stable models of the activator program $Act(D)$. For a model $I \in Sem(\sqcup Act(D))$ we select only relevant sub-programs of the main program: $Main(D)^I$. In the second step we compute the stable models of this sub-program and pronounce them to be the stable models of the split modular program.

Definition 9. *Let* D *be a split modular program and* I *an interpretation. We define the* active program *of* D *w.r.t.* I *as*

$$D^I = Main(D)^I.$$

Definition 10. *Let* I *and* M *be interpretations and* D *a split modular program. We say that* (I, M) *is a split stable model of* D *iff*

i) I *is a stable model of* $Act(D)$ *(*$I \in Sem(\sqcup Act(D))$*)*
ii) M *is a stable model of* D^I *(*$M \in Sem(\sqcup D^I$*)*

We say that M *is a stable model of* D *if there is an interpretation* I *such that* (I, M) *is a split stable model of* D*.*

An important difference to the declarative approach is that the activator modules of *all* split modules are *always* considered. This means that a module can force itself to be included in the active program, either unconditionally or when certain facts are present. The main intuition is that activator programs should be simpler so that the decision which main modules to include can be made quickly.

Example 2. Consider a split modular program consisting of the following split modules:

$$\begin{array}{rl} P_{data} = & \{\quad at(home).\ homework\_done. \quad\} \\ A_{data} = & P_{data} \cup \{\quad load\_module(data). \quad\} \end{array}$$

$$\begin{array}{rl} P_{home} = & \{\quad \{go\_cycling \leftarrow homework\_done.\} \quad\} \\ A_{home} = & \left\{ \begin{array}{c} load\_module(home) \leftarrow at(home). \\ load\_module(school) \leftarrow \textbf{not}\ homework\_done. \end{array} \right\} \end{array}$$

$$\begin{array}{rl} P_{school} = & \{\quad \ldots \quad\} \\ A_{school} = & \{\quad load\_module(school) \leftarrow at(school). \quad\} \end{array}$$

The *data* module is always active, and thus P_{data} will always be considered. We include the data in both activator and main module so then the other activator programs can use it to make decisions. The activator program of the *home* module creates a conditional dependency on another module and also automatically loads itself when certain facts are present.

Note that it is often desired for the A to be also included in P. We could incorporate this directly to the semantics modifying the $Main(D)$ to use both modules ($Main(D) = \{P_{name}, A_{name} \mid (A_{name}, P_{name}) \in D\}$), especially when A is supposed to be simple. On the other hand, this is something that can be easily handled by an implementation, giving the user an option to specify the behaviour either globally or separately for each module.

Also our definition does not give the main program any information about which modules are really loaded. This can of course be easily circumvented by including a dedicated fact in each main module, e.g. *loaded*(*name*).

As in the case of declarative semantics we can easily create a transformation into normal logic programs if we consider logic modules to be normal logic programs and $\mathrm{Sem}(\sqcup\{P_i\}) = \mathrm{Sem}(\bigcup\{P_i\})$. To do this we take a new predicate not used in any program, say *act* and for every rule in an activator program of the form

$$L_0 \leftarrow L_1, \ldots, L_n, \textbf{not}\ L_{n+1}, \ldots, \textbf{not}\ L_m.$$

we create a rule

$$act(L_0) \leftarrow act(L_1), \ldots, act(L_n), \textbf{not}\ act(L_{n+1}), \ldots, \textbf{not}\ act(L_M).$$

Then we create a new rule for each rule of the main programs by adding $act(load\_module(name))$ to the body of the rule, where *name* is the name of the respective module the rule comes from.

6 Comparison

We allow modules to be of any form and the semantics of $\mathrm{Sem}(\sqcup\{P_n\})$ to be completely arbitrary. Therefore we cannot describe any non-trivial relationship between stable models of modular and split modular programs. We have however already shown transformations to normal logic programs in the case when modules are normal logic programs and $\mathrm{Sem}(\sqcup\{P_n\})$ is the stable model semantics of normal logic program $\bigcup P_n$.

We can still construct a transformation from split modular programs into modular programs if we slightly restrict the behavior of $\mathrm{Sem}(\sqcup\{P_n\})$. The requirement we need is that two sets of modules with disjoint literals do not *interact* with each other and that the joint models can be constructed from the respective singular models of the sets.

Definition 11. *Let $\{P_n \mid n \in Names\}$ be a set of LP modules. Then $Lit(\{P_n\})$ is the set of all literals used in the modules P_n.*

Definition 12. *Given a set of LP modules $\mathcal{P} = \{P_n \mid n \in Names\}$, we denote the set of its stable models by*

$$\mathrm{DSem}(\sqcup\mathcal{P})$$

if the following condition holds: Let $\mathcal{P}$ and $\mathcal{T}$ be two sets of LP modules such that $Lit(\mathcal{P}) \cap Lit(\mathcal{T}) = \emptyset$. Then

$$M_1 \cup M_2 \in \mathrm{DSem}(\sqcup(\mathcal{P} \cup \mathcal{T})) \iff M_1 \in \mathrm{DSem}(\sqcup(\mathcal{P}) \wedge M_2 \in \mathrm{DSem}(\sqcup(\mathcal{T}))$$

We just need to ensure that activator and main programs of a split module do not share literals, that they have unique names and that the main programs do not include the special predicate *load_module*.

Definition 13. *Let $\mathcal{D} = \{(n, A_n, P_n)\}$ be a split modular program such that $Lit(Act(\mathcal{D})) \cap Lit(Main(\mathcal{D})) = \emptyset$ and $Lit(Main(\mathcal{D})) \cap \{load\_module(n) \mid n \in Names\} = \emptyset$. We also assume that there is no module named act_n for any module named n. We define $Rename(Act(\mathcal{D}))$ as*

$$Rename(Act(\mathcal{D})) = \{A_{act\_n} \mid A_{act\_n} = A_n \in Act(\mathcal{D})$$

Then the modular program $Tr(\mathcal{D})$ can be defined as

$$Tr(\mathcal{D}) = Rename(Act(\mathcal{D})) \cup Main(\mathcal{D}).$$

We can now ask for the stable models $Tr(\mathcal{D})$. Because of our assumption on the semantics we can split these modules into two parts: the one computed by the activator programs that actually selects which main modules should be loaded and the one computed by the selected main programs. We just need to ensure that all activator programs will be considered. We will do this by computing the stable models of the modular logic program $Tr(\mathcal{D})$ with the set of initial modules $Rename(Act(\mathcal{D}))$.

Theorem 2. *Let $\mathcal{D}$ be a split modular program as specified in definition 13, Let I and M be interpretations such that $I \cap M = \emptyset$, $I \subseteq Lit(Act(\mathcal{D}))$, $M \subseteq Lit(Main(\mathcal{D}))$. Then (I, M) is a split stable model of $\mathcal{D}$ iff $I \cup M$ is a stable model of $Tr(\mathcal{D})$ with the initial set of modules $\{act\_n \mid A_n \in Act(\mathcal{D})\}$.*

Proof. Let (I, M) be a split stable model of $\mathcal{D}$. This means that $I \in \mathrm{Sem}(Act(\mathcal{D})) = \mathrm{Sem}(Rename(Act(\mathcal{D}))$ and $M \in \mathrm{Sem}(Main(\mathcal{D})^I$. Because $Lit(\mathcal{D}) \cap Lit(\mathcal{D}) = \emptyset$ we have $Sel(I) = Sel(I \cup M)$ and therefore

$$Rename(Act(\mathcal{D})) \cup Main(\mathcal{D})^I = (Rename(Act(\mathcal{D})) \cup Main(\mathcal{D}))^J$$

$$J = I \cup M \cup \{load\_module(n) \mid n \in Rename(Act(\mathcal{D}))\}.$$

Because of the assumption on our semantics $I \cup M$ is a stable model of $Tr(\mathcal{D})$.

Conversely let M' be a stable model of $Tr(\mathcal{D})$. According to the assumption on our semantics there are interpretations $I \subseteq Lit(Act(\mathcal{D}))$ and $M \subseteq Lit(Main(\mathcal{D}))$ such that

$$\begin{aligned} I &\in \mathrm{Sem}(Rename(Act(\mathcal{D})) = \mathrm{Sem}(Act(\mathcal{D})) \\ M &\in \mathrm{Sem}(Main(\mathcal{D})^{M' \cup \{load\_module(n) \mid n \in Rename(Act(\mathcal{D}))\}} = \\ &= \mathrm{Sem}(Main(\mathcal{D})^{M'}) = \mathrm{Sem}(Main(\mathcal{D})^I). \end{aligned}$$

Thus (I, M) is a split stable model of $\mathcal{D}$.

The requirement for activator and main modules to have completely disjoint literals is problematic when evaluating queries. In such situations we usually want activator and main programs to get the same input data. This can be of course circumvented by *renaming* not only the programs themselves but also the data so it uses new literal names.

It is however possible to relax the assumption by allowing the programs to share certain literals when these literals have the nature of *facts*, i.e. they will then be present in all stable models. Formally we would change the disjointness requirement in definition 12 to $Lit(\mathcal{P}) \cup Lit(\mathcal{T}) = F$ and require then that $F \subseteq M_1$ and $F \subseteq M_2$.

We also used the semantics of modular logic programs with an initial set of modules. The plain semantics of definition 4 can be used too, we just need to

- Consider only models that *select* all the activator programs
- Make sure activator programs will be selected in some model. This can be done by either adding a module or modify each activator program[1].

7 Conclusion

We presented a simple modularization framework, that allows on-demand loading of ASP modules. The language of logic programs is extended with a special

[1] Because we do not make any assumptions on the modules, this may or may not be achieved by just adding $load\_modules(act\_n)$ as facts.

predicate that allows a direct description of dependencies between modules and conditions specifying when the modules should be loaded directly in ASP programs. Two approaches were presented: a declarative approach and a two-step approach more suitable for practical applications. Neither of the approaches assumes any specific way of combining the modules or of obtaining the stable models of the modular program. They can be thus used in conjunction with other approaches to modularization or with other classes of logic programs than normal logic programs.

References

[BLM94] Michele Bugliesi, Evelina Lamma, and Paola Mello, *Modularity in logic programming*, Journal of Logic Programming. **19/20** (1994), 443–502.

[BMPT94] Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini, *Modular logic programming*, ACM Trans. Program. Lang. Syst. **16** (1994), no. 4, 1361–1398.

[EGV97] Thomas Eiter, Georg Gottlob, and Helmut Veith, *Modular logic programming and generalized quantifiers*, LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (London, UK), Springer-Verlag, 1997, pp. 290–309.

[GL88] Michael Gelfond and Vladimir Lifschitz, *The stable model semantics for logic programming*, Proceedings of the Fifth International Conference on Logic Programming, The MIT Press, 1988, pp. 1070–1080.

[GS89] H. Gaifman and E. Shapiro, *Fully abstract compositional semantics for logic programs*, POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM, 1989, pp. 134–142.

[JOTW07] Tomi Janhunen, Emilia Oikarinen, Hans Tompits, and Stefan Woltran, *Modularity aspects of disjunctive stable models*, Logic Programming and Nonmonotonic Reasoning, 9th International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2007, Tempe, AZ, USA, May 2007, Proceedings (Tempe, Arizona, USA) (Chitta Baral, Gerhard Brewka, and John Schlipf, eds.), Lecture Notes in Artificial Intelligence, vol. 4483, Springer-Verlag, May 2007, pp. 175–187.

[OJ06] Emilia Oikarinen and Tomi Janhunen, *Modular equivalence for normal logic programs*, Proceedings of the 17th European Conference on Artificial Intelligence (Riva del Garda, Italy) (Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, eds.), IOS Press, August 2006, pp. 412–416.