A Module-Based Framework for Multi-language Constraint Modeling*

Matti Järvisalo, Emilia Oikarinen, Tomi Janhunen, and Ilkka Niemelä

Helsinki University of Technology TKK
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
{matti.jarvisalo,emilia.oikarinen,tomi.janhunen,
ilkka.niemela}@tkk.fi

Abstract. We develop a module-based framework for constraint modeling where it is possible to combine different constraint modeling languages and exploit their strengths in a flexible way. In the framework a constraint model consists of modules with clear input/output interfaces. When combining modules, apart from the interface, a module is a black box whose internals are invisible to the outside world. Inside a module a chosen constraint language (approaches such as CP, ASP, SAT, and MIP) can be used. This leads to a clear modular semantics where the overall semantics of the whole constraint model is obtained from the semantics of individual modules. The framework supports multi-language modeling without the need to develop a complicated joint semantics and enables the use of alternative semantical underpinnings such as default negation and classical negation in the same model. Furthermore, computational aspects of the framework are considered and, in particular, possibilities of benefiting from the known module structure in solving constraint models are studied.

1 Introduction

There are several constraint-based approaches to solving combinatorial search and optimization problems: constraint programming (CP), answer set programming (ASP), mixed integer programming (MIP), linear programming (LP), propositional satisfiability checking (SAT) and its extension to satisfiability modulo theories (SMT). Each has its particular strengths: for example, CP systems support global constraints, ASP recursive definitions and default negation, MIP constraints on real-valued variables, and SAT efficient solver technology. In larger applications it is often necessary to exploit the strengths of several languages and to reuse and combine available components. For example, in scheduling problems involving a large amount data and constraints, multi-language modeling can be very useful (as also exemplified in this paper in Sect. 5).

In this work we develop a module-based framework for modeling complex problems with constraints using a combination of different modeling languages. Rather than taking one language as a basis and extending it, we develop a framework for multi-language modeling where different languages are treated on equal terms. The starting

* This work is financially supported by Academy of Finland under the project *Methods for Constructing and Solving Large Constraint Models* (grant #122399).

E. Erdem, F. Lin, and T. Schaub (Eds.): LPNMR 2009, LNCS 5753, pp. 155–168, 2009.

point is to use modules with clear input/output (I/O) interfaces. When combining modules, apart from the interface, a module is a black box whose internals are invisible to the outside world. Inside a module a chosen constraint language (for example, CP, ASP, MIP) with its normal semantics can be used. In this way a clear modular semantics is obtained: the overall semantics of the whole constraint model (consisting of modules) is obtained by "composing" the semantics of individual modules.

We see substantial advantages of this approach for modeling. The clear module interfaces enable support for multi-language modeling without the need to develop a complicated joint semantics capturing arbitrary combinations of special constraints available in different languages. It is also possible to use alternative semantical underpinnings such as default negation and classical negation in the same model. The module-based approach brings the benefits of modular programming to developing constraints models and enables to create libraries to enhance module reuse. It also improves elaboration tolerance and facilitates maintaining and updating a constraint model. Moreover, extending the approach with further languages is conceptually straightforward.

Computational aspects of the framework are also promising. Module interfaces and separation of inputs and outputs can be exploited in decision methods, for example, with more top-down solution techniques where the overall output of the constraint model can be used to identify the relevant parts of the model. The module-based approach allows optimizing the computational efficiency of a model in a structured way: a module can be replaced by another (more optimized) version without altering the solutions of the model as long as the I/O relation of the module is not changed. Similarly, the framework supports modular testing, validation, and debugging of constraint models.

This module-based framework for multi-language modeling seems to be a novel approach. Several approaches to adding modularity to ASP languages [1,2,3,4] have been proposed. However, in these approaches modular multi-language modeling is not directly supported (although the combination of propositional ASP and SAT modules is studied in [5]). A large number of extended modeling languages have also been previously proposed. On one hand, ASP languages have been extended with constraints or other externally defined relations (see, e.g., [6,7,8,9,10,11]). On the other hand, Prolog systems have been extended with ASP features [12,13,14]. Extended modeling languages have been developed also for constraint programming, including ESRA [15], ESSENCE [16], and Zinc [17]. However, none of the approaches supports modular multi-language modeling where different languages are treated on equal terms. Instead, they can all be seen as extensions of a given basic language with features from other languages.

The rest of this paper is organized as follows. As preliminaries, we first give a generic definition of constraints and related notation (Sect. 2). Then constraint modules, the basic building blocks of module systems, are introduced (Sect. 3). The language of module systems, based on composing constraint modules, is discussed in Sect. 4. Then, in Sect. 5 we discuss how the framework can be instantiated in practice: A larger application is considered in order to illustrate the issues arising in using a multi-language modeling approach, and the required language interface for constructing a multi-language module system is sketched. Before conclusions (Sect. 7), computational aspects, and especially, possibilities of benefiting from the explicit modular constraint model description when solving such a model are highlighted (Sect. 6).

2 Constraints

In this section we give necessary definitions and notation related to the generic concept of constraints applied in this work. These serve as basic building blocks for constraint modules which are then combined to form complex constraint models.

Let $\mathcal{X}$ be a set of variables. For each variable $x \in \mathcal{X}$, we associate a set of *values* $D(x)$, called the *domain* of x. Given a set $X \subseteq \mathcal{X}$ of variables, an assignment over X is a function

$$\tau : X \to \bigcup_{x \in X} D(x),$$

which maps variables in X to values in their domains. A *constraint* $\mathcal{C}$ over a set of variables X is characterized by a set $\mathsf{Solutions}(\mathcal{C})$ of assignments over X, called the *satisfying assignments* of $\mathcal{C}$. We denote by $\mathsf{Vars}(\mathcal{C})$ the set X of variables.

It is important to notice that, since the satisfying assignments solely characterize the constraint, this generic way of describing constraints does not specify how a constraint should be implemented, i.e., the modeling language and semantics used for realizing the constraint declaratively remain unspecified.

Example 1. Let $\mathcal{C}$ be a constraint over a set of Boolean variables $\{a, b\}$, i.e., $D(a) = D(b) = \{\mathbf{t}, \mathbf{f}\}$, characterized by $\mathsf{Solutions}(\mathcal{C}) = \{\tau_1, \tau_2\}$, where $\tau_1 = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}\}$ and $\tau_2 = \{a \mapsto \mathbf{f}, b \mapsto \mathbf{t}\}$. Now, $\mathcal{C}$ can be implemented, for example, as a *normal logic program* $\{a \leftarrow {\sim}b.\ b \leftarrow {\sim}a\}$ or as a *disjunctive logic program* $\{a \vee b \leftarrow\}$ in ASP, or as a *conjunctive normal form (CNF) formula* $\{a \vee \neg b, \neg a \vee b\}$ in SAT.

Given an assignment τ and a set of variables X, the *projection* $\pi_X(\tau)$ of τ on X is the assignment that maps each variable $x \in X$ for which $\tau(x)$ is defined to $\tau(x)$. For instance, the projection $\pi_{\{a\}}(\tau_1)$ for τ_1 from Example 1 is the assignment $\pi_{\{a\}}(\tau_1) = \{a \mapsto \mathbf{t}\}$ over the set $\{a\}$.

Given a constraint $\mathcal{C}$, and an assignment τ over a set X of variables, the *restriction* $\mathcal{C}[\tau]$ of $\mathcal{C}$ to τ is characterized by

$$\mathsf{Solutions}(\mathcal{C}[\tau]) = \{\tau' \in \mathsf{Solutions}(\mathcal{C}) \mid \pi_{\mathsf{Vars}(\mathcal{C}) \cap X}(\tau') = \pi_{\mathsf{Vars}(\mathcal{C}) \cap X}(\tau)\}.$$

For instance, let $\tau_3 = \{b \mapsto \mathbf{f}\}$ be an assignment over $\{b\}$. Now, the restriction $\mathcal{C}[\tau_3]$ of $\mathcal{C}$ from Example 1 is a constraint characterized by $\{\tau_1\} \subseteq \mathsf{Solutions}(\mathcal{C})$, i.e., $\mathsf{Solutions}(\mathcal{C}[\tau_3]) = \{\tau_1\}$.

Given two constraints $\mathcal{C}$ and $\mathcal{C}'$, an assignment τ over $\mathsf{Vars}(\mathcal{C})$ is *compatible* with an assignment τ' over $\mathsf{Vars}(\mathcal{C}')$ if $\pi_{\mathsf{Vars}(\mathcal{C}) \cap \mathsf{Vars}(\mathcal{C}')}(\tau) = \pi_{\mathsf{Vars}(\mathcal{C}) \cap \mathsf{Vars}(\mathcal{C}')}(\tau')$. The union $\tau \cup \tau'$ of two compatible assignments, τ and τ' over X and X', respectively, is the assignment over $X \cup X'$ mapping each $x \in X$ to $\tau(x)$ and each $x \in X' \setminus X$ to $\tau'(x)$.

Example 2. Let $\mathcal{C}'$ be a constraint over a set of Boolean variables $\{b, c\}$ characterized by $\mathsf{Solutions}(\mathcal{C}') = \{\tau'\}$ such that $\tau' = \{b \mapsto \mathbf{f}, c \mapsto \mathbf{f}\}$. Consider $\mathcal{C}$ from Example 1. The assignment τ_1 is compatible with τ', because $\{a, b\} \cap \{b, c\} = \{b\}$ and $\tau_1(b) = \mathbf{f} = \tau'(b)$. On the other hand, τ_2 is not compatible with τ', because $\tau_2(b) = \mathbf{t} \neq \tau'(b)$. The union $\tau_1 \cup \tau' = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}, c \mapsto \mathbf{f}\}$ is an assignment over the set $\{a, b, c\}$.

3 Constraint Modules

The view to constructing complex constraint models proposed in this work is based on expressing such models as *module systems*. Module systems are built from *constraint modules* which are combined together in a controlled fashion. In this section we introduce the generic concept of a constraint module. Constraint modules are based on a chosen constraint, with the addition of an explicit I/O interface. Our definition for a constraint module is generic in the sense that it does not insist on a specific implementation of the constraint on the declarative level. The aim here is to allow implementing the constraint using different declarative languages, offering the implementer of a module the possibility to choose the constraint language and the semantics.

Definition 1. *A constraint module $\mathcal{M}$ is a triple $\langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$, where $\mathcal{C}$ is a constraint, and $\mathcal{I}$ and $\mathcal{O}$, with $\mathcal{I} \cap \mathcal{O} = \emptyset$, define the* I/O interface *of $\mathcal{M}$:*

- $\mathcal{I} \subseteq \mathsf{Vars}(\mathcal{C})$ *is the input specification of $\mathcal{M}$, and*
- $\mathcal{O} \subseteq \mathsf{Vars}(\mathcal{C})$ *is the output specification of $\mathcal{M}$.*

A module $\mathcal{M}$ is thus a constraint with a fixed I/O interface. In analogy to the characterization of a constraint, a module $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$ is characterized by a set $\mathsf{Solutions}(\mathcal{M})$ of assignments over $\mathcal{I} \cup \mathcal{O}$ called the *satisfying assignments of the module.* Given a constraint module $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$ and an assignment τ_I over $\mathcal{I}$, the set of consistent outputs of $\mathcal{M}$ w.r.t. τ_I is

$$\mathsf{SolutionOut}(\mathcal{M}, \tau_I) := \{\pi_{\mathcal{O}}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{C}[\tau_I])\}.$$

The satisfying assignments of a module are obtained by considering all possible input assignments.

Definition 2. *Given a constraint module $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$, the set* $\mathsf{Solutions}(\mathcal{M})$ *of satisfying assignments of $\mathcal{M}$ is the union of the sets $\{\tau_I \cup \tau_O \mid \tau_O \in \mathsf{SolutionOut}(\mathcal{M}, \tau_I)\}$ for all assignments τ_I over $\mathcal{I}$.*

Those variables in $\mathsf{Vars}(\mathcal{C})$ which are not in $\mathcal{I} \cup \mathcal{O}$ are *local to* $\mathcal{M}$; the assignments in $\mathsf{Solutions}(\mathcal{M})$ do not assign values to them. Notice that the possibility of local variables enables *encapsulation* and *information hiding*. A module offers through its I/O interface to the user a black-box implementation of a specific constraint. The idea behind this abstract way of defining a module is that, looking from the outside of a module when using the module as a part of a constraint model, the user is interested in the input-output relationship, i.e., the functionality of the module. This can be highlighted by making explicit the conditions under which two modules are considered equivalent.

Definition 3. *Two constraint modules, $\mathcal{M}_1 = \langle \mathcal{C}_1, \mathcal{I}_1, \mathcal{O}_1 \rangle$ and $\mathcal{M}_2 = \langle \mathcal{C}_2, \mathcal{I}_2, \mathcal{O}_2 \rangle$, are* equivalent, *denoted by $\mathcal{M}_1 \equiv \mathcal{M}_2$, if and only if $\mathcal{I}_1 = \mathcal{I}_2$, $\mathcal{O}_1 = \mathcal{O}_2$, and* $\mathsf{Solutions}(\mathcal{M}_1) = \mathsf{Solutions}(\mathcal{M}_2)$.

Example 3. Consider $\mathcal{M} = \langle \mathcal{C}, \{a\}, \{b\} \rangle$, where a and b are Boolean variables, and let $\mathsf{Solutions}(\mathcal{M}) = \{\tau_1, \tau_2\}$ where $\tau_1 = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}\}$ and $\tau_2 = \{a \mapsto \mathbf{f}, b \mapsto \mathbf{t}\}$. Since τ_1 and τ_2 are the same as in Example 1, $\mathcal{M}$ can be implemented using any of the implementations of the constraint described in Example 1.

Moreover, the set of variables used in implementing $\mathcal{C}$ is not limited to $\{a, b\}$. For instance, a *logic program module* [4] $\langle P, I, O\rangle = \langle\{c \leftarrow {\sim}a.\ b \leftarrow c\}, \{a\}, \{b\}\rangle$ is an implementation of $\mathcal{C}$ such that $\mathsf{Solutions}(\mathcal{C}) = \{\tau_3, \tau_4\}$ where $\tau_3 = \{a \mapsto \mathbf{t}, b \mapsto \mathbf{f}, c \mapsto \mathbf{f}\}$ and $\tau_4 = \{a \mapsto \mathbf{f}, b \mapsto \mathbf{t}, c \mapsto \mathbf{t}\}$.[1] Now, there are two possible assignments over $\{a\}$. If $\tau_I = \{a \mapsto \mathbf{t}\}$ we obtain $\mathsf{SolutionOut}(\mathcal{M}, \tau_I) = \{\pi_{\{b\}}(\tau_3)\}$ since $\mathsf{Solutions}(\mathcal{C}[\tau_I]) = \{\tau_3\}$ as $\tau_3(a) = \tau_I(a) = \mathbf{t}$. For the other possible input assignment $\tau'_I = \{a \mapsto \mathbf{f}\}$, we obtain $\mathsf{SolutionOut}(\mathcal{M}, \tau'_I) = \{\pi_{\{b\}}(\tau_4)\}$. Finally, notice that $\tau_I \cup \pi_{\{b\}}(\tau_3) = \tau_1$ and $\tau'_I \cup \pi_{\{b\}}(\tau_4) = \tau_2$. Thus, $\mathsf{Solutions}(\mathcal{M}) = \{\tau_1, \tau_2\}$.

4 Module Systems

In this section we discuss how larger module systems are built from individual constraint modules. The idea is that module systems are constructed by connecting smaller module systems through the I/O interfaces offered by such systems. In other words, in analogy to constraint modules, a module system has an I/O interface, and constraint modules are seen as primitive module systems. We will start by introducing a formal language for expressing such systems and then introduce the semantics for module systems which are *well-formed*.

Definition 4 (The language of module systems)

1. *A constraint module* $\mathcal{M} = \langle\mathcal{C}, \mathcal{I}, \mathcal{O}\rangle$ *is a module system with* $\mathsf{Input}(\mathcal{M}) = \mathcal{I}$ *and* $\mathsf{Output}(\mathcal{M}) = \mathcal{O}$.
2. *If* $\mathcal{M}$ *is a module system and* X *a set of variables, then* $\pi_X(\mathcal{M})$ *is a module system with* $\mathsf{Input}(\pi_X(\mathcal{M})) = \mathsf{Input}(\mathcal{M})$ *and* $\mathsf{Output}(\pi_X(\mathcal{M})) = \mathsf{Output}(\mathcal{M}) \cap X$.
3. *If* $\mathcal{M}$ *and* $\mathcal{M}'$ *are module systems, then* $(\mathcal{M} \triangleright \mathcal{M}')$ *is a module system with* $\mathsf{Input}(\mathcal{M}_1 \triangleright \mathcal{M}_2) = \mathsf{Input}(\mathcal{M}_1) \cup (\mathsf{Input}(\mathcal{M}_2) \setminus \mathsf{Output}(\mathcal{M}_1))$ *and* $\mathsf{Output}(\mathcal{M}_1 \triangleright \mathcal{M}_2) = \mathsf{Output}(\mathcal{M}_1) \cup \mathsf{Output}(\mathcal{M}_2)$.

Notice that Definition 4 is purely syntactical. Our next goal is to define the semantics for more complex module systems as we have already defined the sets of satisfying assignments for individual constraint modules. This is achieved by formalizing the semantics of the two operators: intuitively, projection π_X offers a way of filtering the output of a module system, whereas composition $\triangleright$ is used for merging two module systems into one. We start by defining the conditions under which two module systems are *composable* and *independent*.

Definition 5 (Composable and independent module systems). *Two module systems* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *are* composable *if* $\mathsf{Output}(\mathcal{M}_1) \cap \mathsf{Output}(\mathcal{M}_2) = \emptyset$. *Module system* $\mathcal{M}_1$ *is* independent *from module system* $\mathcal{M}_2$ *if* $\mathsf{Input}(\mathcal{M}_1) \cap \mathsf{Output}(\mathcal{M}_2) = \emptyset$.

Composability is used to ensure that if two module systems interfere with each others' output, they cannot be put together. Independence allows us to ensure that two modules are not in cyclic dependency. Notice that the independence of $\mathcal{M}_1$ from $\mathcal{M}_2$ does not imply that $\mathcal{M}_2$ is independent from $\mathcal{M}_1$.

[1] Notice that unlike other formalisms mentioned so far, the logic program modules in [4] already facilitate I/O interfaces, and their semantics differs from the standard stable model semantics since input variables have a classical interpretation.

Definition 6 (Module composition). *Given two composable module systems $\mathcal{M}_1$ and $\mathcal{M}_2$, their composition $\mathcal{M}_1 \triangleright \mathcal{M}_2$ is defined if and only if $\mathcal{M}_1$ is independent from $\mathcal{M}_2$. The set of satisfying assignments of $\mathcal{M}_1 \triangleright \mathcal{M}_2$,* $\mathsf{Solutions}(\mathcal{M}_1 \triangleright \mathcal{M}_2)$*, is*

$$\{(\tau_1 \cup \tau_2) \mid \tau_1 \in \mathsf{Solutions}(\mathcal{M}_1), \tau_2 \in \mathsf{Solutions}(\mathcal{M}_2), \text{ and } \tau_2 \text{ is compatible with } \tau_1\}.$$

Example 4. Let $\mathcal{M} = \langle \mathcal{C}, \{a\}, \{n\} \rangle$ and $\mathcal{M}' = \langle \mathcal{C}', \{n\}, \{m\} \rangle$ be constraint modules where a is a Boolean variable, $D(n) = D(m) = \{1, 2, 3\}$, $\mathsf{Solutions}(\mathcal{M}) = \{\tau_1\}$, and $\mathsf{Solutions}(\mathcal{M}') = \{\tau_2, \tau_3\}$ where $\tau_1 = \{a \mapsto \mathbf{f}, n \mapsto 3\}$, $\tau_2 = \{n \mapsto 1, m \mapsto 1\}$, and $\tau_3 = \{n \mapsto 3, m \mapsto 2\}$. Since $\mathcal{M}$ is independent from $\mathcal{M}'$, their composition $\mathcal{M} \triangleright \mathcal{M}'$ is defined. Notice that $n \in \mathsf{Output}(\mathcal{M}) \cap \mathsf{Input}(\mathcal{M}')$ provides the connection between $\mathcal{M}$ and $\mathcal{M}'$, i.e., $n \in \mathsf{Output}(\mathcal{M})$ is the input for $\mathcal{M}'$ because $n \in \mathsf{Input}(\mathcal{M}')$. Furthermore, $\mathsf{Input}(\mathcal{M} \triangleright \mathcal{M}') = \{a\}$, $\mathsf{Output}(\mathcal{M} \triangleright \mathcal{M}') = \{n, m\}$, and $\mathsf{Solutions}(\mathcal{M} \triangleright \mathcal{M}') = \{\tau_1 \cup \tau_3\}$, because τ_1 is not compatible with τ_2 and τ_1 is compatible with τ_3.

As a special case, the *empty module* $\mathcal{E}$ is a constraint module such that $\mathsf{Input}(\mathcal{E}) = \mathsf{Output}(\mathcal{E}) = \emptyset$ and $\mathsf{Solutions}(\mathcal{E}) = \{\tau_e\}$, where τ_e is the *empty assignment*. Given any module system $\mathcal{M}$, both $\mathcal{E} \triangleright \mathcal{M}$ and $\mathcal{M} \triangleright \mathcal{E}$ are defined, and $\mathcal{E} \triangleright \mathcal{M} \equiv \mathcal{M} \triangleright \mathcal{E} \equiv \mathcal{M}$.

Definition 7 (Projecting output of a module system). *Given a module system $\mathcal{M}$ and set of variables $\mathcal{O}$, the module system $\pi_{\mathcal{O}}(\mathcal{M})$ is defined if and only if $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$. The set of satisfying assignments of $\pi_{\mathcal{O}}(\mathcal{M})$,* $\mathsf{Solutions}(\pi_{\mathcal{O}}(\mathcal{M}))$*, is*

$$\{\pi_{\mathcal{O} \cup \mathsf{Input}(\mathcal{M})}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{M})\}.$$

Example 5. Consider the module system $\mathcal{M} \triangleright \mathcal{M}'$ from Example 4 and assume that we are not interested in the values assigned to n. Thus, we consider the projection $\mathcal{M}_\pi = \pi_{\{m\}}(\mathcal{M} \triangleright \mathcal{M}')$. Now $\mathsf{Input}(\mathcal{M}_\pi) = \{a\}$, $\mathsf{Output}(\mathcal{M}_\pi) = \{m\}$, and $\mathsf{Solutions}(\mathcal{M}_\pi) = \{\tau_\pi\}$ where $\tau_\pi = \pi_{\{a,m\}}(\tau_1 \cup \tau_3) = \{a \mapsto \mathbf{f}, m \mapsto 2\}$.

We are interested in so called *well-formed* module systems that respect the conditions for applying $\triangleright$ (independence) and π_X (projection is focused on output).

Definition 8 (Well-formed module system). *A module system is* well-formed *if each composition and projection operation is defined in the sense of Definitions 6 and 7.*

Determining whether an arbitrary module system is well-formed consists of a syntactic check on the compositionality and compatibility of the I/O interfaces ($\triangleright$) and subset relation (π). From now on we use the term *module system* to refer to a well-formed module system. The graph formed by taking into account the input-output dependencies of parts of a module system is *directed* and *acyclic*, and is referred to as the *module dependency graph*. More precisely, the module dependency graph of a given module system $\mathcal{M}$ has the set of constraint modules appearing in $\mathcal{M}$ as the set of vertices. There is a edge from a constraint module $\mathcal{M}_1$ to module $\mathcal{M}_2$ if and only if at least one output variable of $\mathcal{M}_1$ is an input variable of $\mathcal{M}_2$. Notice that the acyclicity comes from that fact that recursive definitions can be stated only inside individual modules.

By definition, the semantics of a well-formed module system is *compositional*: compatible solutions for individual parts form a solution for the whole system and a solution for the module system gives solutions for the individual parts.

Remark 1. Operators for $\triangleright$ and π_X provide flexible ways for building complex module systems. Additional operators useful in practice can be defined as combinations of these basic operators. For instance, by combining composition with projection we obtain $\mathcal{M}_1 \blacktriangleright \mathcal{M}_2$ defined as $\pi_{\mathsf{Output}(\mathcal{M}_2)}(\mathcal{M}_1 \triangleright \mathcal{M}_2)$. One could also be interested in a non-deterministic choice of solutions for $\mathcal{M}_1$ and $\mathcal{M}_2$ (denoted $\mathcal{M}_1 \cup \mathcal{M}_2$) or common solutions for $\mathcal{M}_1$ and $\mathcal{M}_2$ (denoted $\mathcal{M}_1 \cap \mathcal{M}_2$). In order to define $\mathcal{M}_1 \cup \mathcal{M}_2$ and $\mathcal{M}_1 \cap \mathcal{M}_2$, we cannot assume that $\mathcal{M}_1$ and $\mathcal{M}_2$ are composable. However, even these operators can be expressed in terms of composition and projection using an additional renaming scheme for variables.

5 Module Systems in Practice

We now outline how the framework for module systems developed in the previous section can be instantiated in practice. First, we consider a demanding application to illustrate the issues arising in using a multi-language modeling approach. Then we sketch the required language interface for constructing a multi-language module system.

5.1 Modular Representation for the Timetabling Domain

For illustrating multi-language modeling, we describe components involved in a modular constraint model for *university timetabling*, variants of which have previously been formalized in SAT, CP, and ASP [18,19]. Designing a feasible weekly schedule for events related to courses in a university curriculum is a challenging task. The problem is not just about allocation time and space resources; the interdependencies of courses and the respective events give rise to a rich body of constraints. For modeling, one needs to express the mutual exclusion of events as regards, e.g., placing any two events in the same lecture hall at the same time. A straightforward representation of such a constraint with clauses or rules may require quadratic space. In contrast, a concise encoding can be obtained with global constraints such as *all-different* or *cumulative* constraints typically supported by constraint programming systems. On the other hand, there are features which are cumbersome to describe in CP. For example, exceptions like the temporary unavailability of a particular lecture hall in a timetable are easy to represent with non-monotonic rules such as those used in ASP. Moreover, rules provide a flexible way of defining new relations on the basis of existing ones.

The structure of a modular constraint model for the university timetabling domain is given in Fig. 1. The two ASP modules at the bottom define relations specific to a particular problem instance. The first module, *eventData*, defines which events are involved in the problem. The second, *resourcesData*, formalizes the time and space resources available for scheduling. An individual *resource* is conceptualized as a pair $\langle r, s \rangle$ where r is a room and s is a session. The ASP module on top of these two modules, *dataViews*, defines a number of subsidiary relations, such as $\textsc{room}(r)$ (available rooms) and $\textsc{lecturer}(l)$ (involved lecturers), on the basis of the relations provided by modules *eventData* and *resourcesData*. The relations $\textsc{maxEvent}(n)$ and $\textsc{maxResource}(m)$ hold (only) for the numbers of events n and resources m, respectively. After suitable type conversions (represented by the circles in Fig. 1), these two

size parameters serve as input for the CP module *allDifferent* whose purpose is to assign different resources (represented by integers in the range $1 \ldots m$) to all events (represented by an array of integers indexed by $1 \ldots n$). Through such a conversion, a constraint library implementation of *allDifferent* which works only on integer-valued variables can be directly used. The resulting array of assignments of integers, RESOURCEOF, is then converted to a relation for events e and resources r and the ASP module *allocation* is used to restore the representation of resources as integers back to pairs of rooms and sessions. The outcome relation $\text{OCCURS}(e, r, s)$ denotes the fact that an event e takes place in room r during session s. The topmost module *testAllocation* ensures that the given allocation of resources to events, i.e., the relation $\text{OCCURS}(e, r, s)$ meets further criteria of interest. For instance, one could insist on the property that sessions related with a particular lecture hall are always reserved in a contiguous manner, i.e., no gaps are allowed between reservations in the respective schedule.

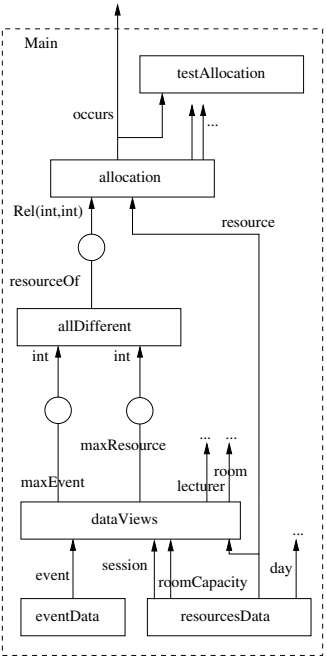

Fig. 1. Example of a Module System

5.2 Language Interface for Combining Constraint Modules

Referring to the theory developed in Sect. 3 and 4, we distinguish two types of module declarations. An individual constraint module is written in a particular constraint language accompanied by an appropriate I/O interface specification. The language of each constraint module is declared using an identifier "SAT", "ASP", "CP", etc. A module system is effectively a definition of the interconnections between submodules encapsulated by it. Since module systems are not confined to a particular constraint language the identifier "SYSTEM" is used. In addition, simple *type converters* are declared when needed, as outlined above.

In practice, a module system is not described as an expression (recall Definition 4) using explicitly composition and projection operators. Instead, it is very useful to give primitive constraint module descriptions as schemata which can be reused by instantiating them with appropriate input and output variables. To support this we follow an approach which handles module instantiation and composition simultaneously. Modules are instantiated using a declaration `[outputlist]=modulename(inputlist);` where `modulename` is the name of the module being instantiated, and `inputlist` and `outputlist` are the lists of input and output variables, respectively. This allows for writing a module composition $\mathcal{M}_1 \triangleright \mathcal{M}_2$ as suitable module instantiations: $[x_1, x_2, \ldots] = \mathcal{M}_1(\ldots)$; $[\ldots] = \mathcal{M}_2(x_1, x_2, \ldots)$; where appropriate output variables of $\mathcal{M}_1$ are used as input variables of $\mathcal{M}_2$. A module system is described as a sequence $\mathcal{M}_1; \mathcal{M}_2; \ldots; \mathcal{M}_n$; of such instantiation declarations which is acyclic, i.e.,

```
#module ASP dataViews
  (Rel(int, string, string, int, string, int) event,
   Rel(int, string, int) resource,
   Rel(int) session,
   Rel(string,int) roomCapacity)
  [Rel(int) maxEvent,
   Rel(int) maxResource,
   Rel(string) room,
   Rel(string) lecturer]

% Determine problem dimensions
eventId(I) :- event(I,CC,T,D,L,C).
maxEvent(I) :- eventId(I), not eventId(I+1).
resourceId(I) :- resource(I,R,S).
maxResource(I) :- resourceId(I), not resourceId(I+1).

% Rooms and personnel
room(R) :- resource(I,R,S).
lecturer(L) :- event(I,CC,l,D,L,C), L!=noname.

...

#endmodule
```

```
#module SYSTEM main()

% Data (problem instance)
[event] = eventData();
[day,session,resource,roomCapacity] =
  resourcesData();

% Different views of data
[maxEvent, maxResource, room, lecturer] =
  dataViews(event,resource,session,roomCapacity);

% Allocating resources
[resourceOf] =
  allDifferent(indexOfTrueElement(maxEvent),
               indexOfTrueElement(maxResource));

% Recover rooms and sessions from resources
[occurs] = allocation(resource,
                      arrayToRel(resourceOf));

% Checking the feasibility allocation
[] = testAllocation(occurs);

#solve[occurs]

#endmodule
```

Fig. 2. Examples of a constraint module and a module system as illustrated in Fig. 1

output variables of $\mathcal{M}_i$ cannot be used as input variables for any $\mathcal{M}_j$, $j \leq i$. This guarantees that the set of declarations can be seen as a well-formed composition $\mathcal{M}'_1 \triangleright (\mathcal{M}'_2 \triangleright (\ldots \triangleright \mathcal{M}'_n) \ldots)$ where $\mathcal{M}'_i$s are the corresponding instantiated constraint modules. The projection operator is handled implicitly in the instantiation of modules. For the top level of a module system we provide an explicit projection operator as the `#solve`$[\cdot]$ directive for defining the actual output variables of the whole module system.

A simplified example of a constraint module and a module system is given in Fig. 2. Each module description begins with a header line. The keyword "`#module`" is followed by (i) the language identifier, e.g., `SAT`, `ASP`, `CP`, or `SYSTEM`, (ii) the name of the module, and (iii) the specification of input and output variables enclosed in parentheses "`(...)`" and brackets "`[...]`", respectively. The types of variables are declared using elementary types (`int`, `string`, ...) and type constructors such as `Rel`.[2] Local variables (if any) and their types are declared with lines that begin with the keyword `#type`. A module description ends with a line designated by a keyword `#endmodule`. The module instantiation declarations need to be well-typed, i.e., the given input and output variables must conform to the module interfaces. The top-level module is distinguished by the reserved name `main` and the `#solve` directive for defining the output variables of the whole module system can be used only there.

6 Computational Aspects and Benefits of the Modular Approach

In this section we consider computational aspects related to module systems. First we analyze how certain computational properties of individual constraint modules are related to those of more complex module systems. Then we show how the structure of a

[2] Description of a complete typing mechanism is beyond the scope of this paper. For now, we aim at type specifications which allow for static type checking.

module system can be exploited when one is interested in finding a satisfying assignment for a subset of the output variables of the module system.

We describe computational properties of a constraint module under the terms *checkable*, *solvable*, and *finite output for fixed input*, defined as follows.

Definition 9. *A constraint module* $\mathcal{M} = \langle \mathcal{C}, \mathcal{I}, \mathcal{O} \rangle$

- *is checkable if and only if given any assignment* τ *over the variables in* $\mathcal{I} \cup \mathcal{O}$*, it can be decided whether* $\tau \in \mathsf{Solutions}(\mathcal{M})$*;*
- *is solvable if and only if there is a computable function that, given any assignment* τ *over the variables in* $\mathcal{I}$*, returns an assignment in* $\mathsf{SolutionOut}(\mathcal{M}, \tau)$ *if one exists, and reports unsatisfiability otherwise; and*
- *has* FOFI *(finite output for fixed input) if and only if (i) the set* $\mathsf{SolutionOut}(\mathcal{M}, \tau)$ *is finite for any assignment* τ *over the variables in* $\mathcal{I}$*, and (ii) there is a computable function that, given any assignment* τ *over the variables in* $\mathcal{I}$*, outputs* $\mathsf{SolutionOut}(\mathcal{M}, \tau)$.

In general, a constraint module which has FOFI is both checkable and solvable. However, a solvable (checkable, respectively) module is not necessarily checkable (solvable, respectively).

The knowledge about a specific property for $\mathcal{M}$ and $\mathcal{M}'$ is not necessarily enough to guarantee that the property holds for a module system obtained using $\mathcal{M}$ and $\mathcal{M}'$. Clearly, if $\mathcal{M}$ and $\mathcal{M}'$ are checkable, then $\mathcal{M} \triangleright \mathcal{M}'$ is checkable, too. Solvability of $\mathcal{M}$ and $\mathcal{M}'$ does not, however, imply that $\mathcal{M} \triangleright \mathcal{M}'$ is solvable. For instance, let $\mathcal{M} = \langle \mathcal{C}, \emptyset, \{a\} \rangle$ and $\mathcal{M}' = \langle \mathcal{C}', \{a\}, \{b\} \rangle$ be solvable constraint modules such that $\mathsf{Solutions}(\mathcal{M}) = \{\{a \mapsto 1\}, \{a \mapsto 2\}\}$, $\mathsf{Solutions}(\mathcal{M}') = \{\{a \mapsto 2, b \mapsto 2\}\}$, and $\mathcal{M} \triangleright \mathcal{M}'$ is defined. Assume that the computable function for $\mathcal{M}$ always returns $\tau = \{a \mapsto 1\}$. Now, $\mathsf{SolutionOut}(\mathcal{M}', \tau) = \emptyset$, and which leads us to think that $\mathsf{Solutions}(\mathcal{M} \triangleright \mathcal{M}') = \emptyset$. But this is in contradiction with $\mathsf{Solutions}(\mathcal{M} \triangleright \mathcal{M}') = \{\{a \mapsto 2, b \mapsto 2\}\}$. If we in addition assume that $\mathcal{M}$ and $\mathcal{M}'$ have the FOFI property, then $\mathcal{M} \triangleright \mathcal{M}'$ is solvable and, moreover, has the FOFI property.

For projection, the situation is slightly different. If $\mathcal{M}$ is a checkable constraint module, then $\pi_{\mathcal{O}}(\mathcal{M})$ is not necessarily checkable for $\mathcal{O} \subset \mathsf{Output}(\mathcal{M})$. Given τ over $\mathsf{Input}(\mathcal{M}) \cup \mathcal{O} \subset \mathsf{Input}(\mathcal{M}) \cup \mathsf{Output}(\mathcal{M})$, we cannot decide whether $\tau \in \mathsf{Solutions}(\pi_{\mathcal{O}}(\mathcal{M}))$ as we do not know the assignment for variables in $\mathsf{Output}(\mathcal{M}) \setminus \mathcal{O}$. If, in addition, $\mathcal{M}$ is solvable, then using the projection τ' of τ to $\mathsf{Input}(\mathcal{M})$ we can compute $\tau'' \in \mathsf{SolutionOut}(\mathcal{M}, \tau')$ and the projection of τ'' to $\mathcal{O}$. Thus $\pi_{\mathcal{O}}(\mathcal{M})$ is solvable.

Proposition 1. *Let* $\mathcal{M}$ *and* $\mathcal{M}'$ *be constraint modules s.t.* $\mathcal{M} \triangleright \mathcal{M}'$ *is defined, and* $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$*. If* $\mathcal{M}$ *and* $\mathcal{M}'$ *are checkable,* $\mathcal{M} \triangleright \mathcal{M}'$ *is checkable. If* $\mathcal{M}$ *is solvable,* $\pi_{\mathcal{O}}(\mathcal{M})$ *is solvable. If* $\mathcal{M}$ *and* $\mathcal{M}'$ *have FOFI,* $\mathcal{M} \triangleright \mathcal{M}'$ *and* $\pi_{\mathcal{O}}(\mathcal{M})$ *have FOFI.*

Based on the concepts of *total module systems* and *don't care variables*, the *cone-of-influence* of a system is intuitively the part of the system that may influence the values of output variables of interest. We will define the *cone-of-influence reduction* for module systems which can be used in disregarding parts of a module system in the case we are only interested in the values assigned to a subset of the output of the system.

Definition 10. *A constraint module $\mathcal{M}$ is* total *if* $\mathsf{SolutionOut}(\mathcal{M}, \tau) \neq \emptyset$ *for all assignments τ over* $\mathsf{Input}(\mathcal{M})$.

If $\mathcal{M}_1$ and $\mathcal{M}_2$ are total module systems such that $\mathcal{M}_1 \triangleright \mathcal{M}_2$ is defined, then $\mathcal{M}_1 \triangleright \mathcal{M}_2$ is total. Furthermore $\pi_{\mathcal{O}}(\mathcal{M})$ is total for any total $\mathcal{M}$ and $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$.

Seen as a black-box entity, testing totality from the outside is hard even on the level of constraint modules. However, if the declarative implementation of the module is known, there are easy-to-test syntactic conditions guaranteeing totality. For example, in Boolean circuit satisfiability, we know that if no gate of a circuit is constrained to a specific truth value, any module implemented by such a Boolean circuit is total. In practice, when implementing reusable modules for inclusion in a module library, the totality of a module could be explicitly declared in the module interface specification.

Definition 11. *Given a constraint module $\mathcal{M}$, $x \in \mathsf{Input}(\mathcal{M})$, and $y \in \mathsf{Output}(\mathcal{M})$, we say that x is a* $\mathcal{M}$-don't care *w.r.t. y, if for any assignment τ over* $\mathsf{Input}(\mathcal{M}) \setminus \{x\}$,

$$\{\pi_{\{y\}}(\tau') \mid \tau' \in \mathsf{SolutionOut}(\mathcal{M}, \tau \cup \tau_1)\} = \{\pi_{\{y\}}(\tau') \mid \tau' \in \mathsf{SolutionOut}(\mathcal{M}, \tau \cup \tau_2)\}$$

for all pairs of assignments τ_1, τ_2 for x.

As in the case of totality, in general checking whether a given input variable is a don't care is hard when constraint modules are seen as black-box entities. But again, if the declarative implementation of the module is known, there are easy-to-test syntactic conditions which guarantee that a variable is a don't care. For example, if a CNF formula can be split into two disjoint components, i.e., sets of clauses which do not share variables. A similar check can be done, e.g., for ASP programs and CSP instances.

In addition to totality and don't cares, we use the concept of *relevant I/O variables*. Let $\mathsf{CM}(\mathcal{M})$ denote the set of constraint modules appearing in a module system $\mathcal{M}$. For instance, if $\mathcal{M} = \pi_{\mathcal{O}}(\mathcal{M}_1 \triangleright \mathcal{M}_2)$ then $\mathsf{CM}(\mathcal{M}) = \{\mathcal{M}_1, \mathcal{M}_2\}$.

Definition 12. *Given a module system $\mathcal{M}$ and $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$, the set of relevant I/O variables in $\mathcal{M}$ w.r.t. $\mathcal{O}$, denoted by* $\mathsf{Rel}(\mathcal{M}, \mathcal{O})$*, is the smallest set $S \supseteq \mathcal{O}$ of variables that fulfills the following conditions:*

- $\mathsf{Input}(\mathcal{M}') \subseteq S$ *for each non-total* $\mathcal{M}' \in \mathsf{CM}(\mathcal{M})$.
- *If $y \in S$, then for each total* $\mathcal{M}' \in \mathsf{CM}(\mathcal{M})$ *such that* $y \in \mathsf{Output}(\mathcal{M}')$, $\{x \in \mathsf{Input}(\mathcal{M}') \mid x \text{ is not } \mathcal{M}'\text{-don't care w.r.t. } y\} \subseteq S$.

The cone-of-influence reduction allows the parts not belonging to the cone-of-influence to be neglected when solving the constraint model.

Definition 13. *Given a module system $\mathcal{M}$ and a set X of variables, the* module system reduction $\mathcal{M}|_X$ *is defined as follows.*

- *If $\mathcal{M}$ is a constraint module, then*
$$\mathcal{M}|_X = \begin{cases} \mathcal{E} \text{ (the empty module)} & \text{, if } \mathsf{Output}(\mathcal{M}) \cap X = \emptyset \text{ and } \mathcal{M} \text{ is total} \\ \mathcal{M} & \text{, otherwise.} \end{cases}$$
- *If $\mathcal{M}$ is of the form $\mathcal{M}_1 \triangleright \mathcal{M}_2$, then $\mathcal{M}|_X = (\mathcal{M}_1|_X \triangleright \mathcal{M}_2|_X)$.*
- *If $\mathcal{M}$ is of the form $\pi_O(\mathcal{M}')$, then $\mathcal{M}|_X = \pi_{\mathsf{Output}(\mathcal{M}'|_X) \cap O}(\mathcal{M}'|_X)$.*

Given a module system $\mathcal{M}$ and a set of variables $\mathcal{O}$, the cone-of-influence reduction *of $\mathcal{M}$ w.r.t. $\mathcal{O}$ is the module system $\mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})}$.*

For finding a satisfying assignment for $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$ of a module system $\mathcal{M}$, one needs to consider only the *subsystem* $\mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})}$ of $\mathcal{M}$.

Proposition 2. *Given a module system $\mathcal{M}$ and a set of variables $\mathcal{O} \subseteq \mathsf{Output}(\mathcal{M})$, then* $\{\pi_{\mathcal{O}}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{M})\} = \{\pi_{\mathcal{O}}(\tau) \mid \tau \in \mathsf{Solutions}(\mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})})\}$.

Example 6. Consider the module system $\mathcal{M} = (\mathcal{M}_1 \triangleright \mathcal{M}_2) \triangleright (\mathcal{M}_3 \triangleright \mathcal{M}_4)$ illustrated in Fig. 3. Thus, $\mathsf{Input}(\mathcal{M}) = \{a, b, c\}$ and $\mathsf{Output}(\mathcal{M}) = \{d, e, f, g\}$. The constraint module $\mathcal{M}_2$ represented with gray in Fig. 3 is not total, while the other constraint modules in $\mathsf{CM}(\mathcal{M})$, i.e., $\mathcal{M}_1$, $\mathcal{M}_3$, and $\mathcal{M}_4$, are total. Assume that, in addition, it is known that e and f are $\mathcal{M}_4$-don't cares w.r.t. g. Assume that we are only interested in finding a satisfying assignment for $\mathcal{O} = \{g\}$. By Proposition 2 we can exploit the cone-of-influence reduction. The set of relevant I/O variables $\mathsf{Rel}(\mathcal{M}, \mathcal{O}) = X = \{a, b, c, d, g\}$ because $\mathcal{O} \subseteq \mathsf{Rel}(\mathcal{M}, \mathcal{O})$, $\mathsf{Input}(\mathcal{M}_2) \subseteq \mathsf{Rel}(\mathcal{M}, \mathcal{O})$, d is not $\mathcal{M}_4$-don't care w.r.t. $g \in \mathsf{Output}(\mathcal{M}_4)$, and a and b are not $\mathcal{M}_1$-don't cares w.r.t. $d \in \mathsf{Output}(\mathcal{M}_1)$. Using the set of relevant I/O variables, the cone-of-influence reduction of $\mathcal{M}$ w.r.t. $\mathcal{O}$ is

$$\begin{aligned}\mathcal{M}|_{\mathsf{Rel}(\mathcal{M},\mathcal{O})} &= (\mathcal{M}_1 \triangleright \mathcal{M}_2)|_X \triangleright (\mathcal{M}_3 \triangleright \mathcal{M}_4)|_X \\ &= (\mathcal{M}_1|_X \triangleright \mathcal{M}_2|_X) \triangleright (\mathcal{M}_3|_X \triangleright \mathcal{M}_4|_X) \\ &= (\mathcal{M}_1 \triangleright \mathcal{M}_2) \triangleright (\mathcal{E} \triangleright \mathcal{M}_4) \\ &= (\mathcal{M}_1 \triangleright \mathcal{M}_2) \triangleright \mathcal{M}_4.\end{aligned}$$

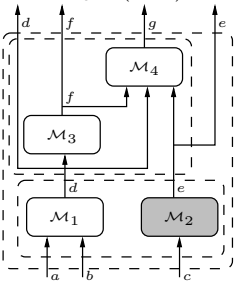

Fig. 3. A module system

7 Conclusions

We develop a generic framework for module-based constraint modeling using multiple modeling languages within the same model. In the framework, constraint models are constructed as module systems which are composed of constraint modules each having an explicit input/output interface specification. This approach has many interesting properties. First of all, individual constraint modules can be implemented using a constraint language most suitable for modeling the constraint in question. The approach paves the way for reusable constraint module libraries and also allows for multiple modelers to implement parts of a constraint model in parallel. Our framework supports modular multi-language modeling by treating different constraint languages on equal terms whereas previous approaches can be seen as extensions of a given basic language with features from other languages. The modular construction of constraint models as module systems yields in itself a structured view to the model which can be exploited when solving the model. We describe a system-level cone-of-influence reduction, which allows parts of the module system to be disregarded when solving a constraint model, without the need to consider properties specific to the constraint languages employed in implementing the individual constraint modules.

For further work, we see a number of possible approaches to solving constraint models expressed using the multi-language framework. In a hybrid system individual constraint modules (or parts of the module system modeled using the same constraint language) are solved using language-specific solvers which have to interact in order to

compute solutions to the whole constraint model. In a translation-based approach all parts of the model are mapped into a single constraint language for which highly efficient off-the-shelf solvers are available. For example, there is interesting recent work on bit-blasting more general CP models into SAT [20]. Another interesting paradigm is the extension of SAT to Satisfiability Module Theories (SMT), into which e.g. stable model computation can be very compactly encoded [21]. Additionally, the modular structure of module systems poses interesting research topics such as harnessing the I/O interfaces in developing novel decision heuristics and devising techniques to instantiate and ground module schemata lazily.

References

1. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 290–309. Springer, Heidelberg (1997)
2. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)
3. Balduccini, M.: Modules and signature declarations for A-Prolog: Progress report. In: SEA, pp. 41–55 (2007)
4. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory and Practice of Logic Programming 8(5-6), 717–761 (2008)
5. Janhunen, T.: Modular equivalence in general. In: ECAI, pp. 75–79. IOS Press, Amsterdam (2008)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: IJCAI, pp. 90–96 (2005)
7. Elkabani, I., Pontelli, E., Son, T.: Smodels^{a} - a system for computing answer sets of logic programs with aggregates. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 427–431. Springer, Heidelberg (2005)
8. Gebser, M., et al.: Clingcon (2009), `http://www.cs.uni-potsdam.de/clingcon/`
9. Tari, L., Baral, C., Anwar, S.: A language for modular answer set programming: Application to ACC tournament scheduling. In: ASP, pp. 277–292 (2005)
10. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
11. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Ann. Math. Artif. Intell. 53(1-4), 251–287 (2008)
12. Castro, L., Swift, T., Warren, D.: Xasp (2009), `http://xsb.sourceforge.net/`
13. El-Khatib, O., Pontelli, E., Son, T.: Integrating an answer set solver into Prolog: ASP-PROLOG. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 399–404. Springer, Heidelberg (2005)
14. Pontelli, E., Son, T., Baral, C.: A logic programming based framework for intelligent web services composition. In: Managing Web Services Quality: Measuring Outcomes and Effectiveness. IDEA Group Publishing (2008)
15. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Bruynooghe, M. (ed.) LOPSTR 2004. LNCS, vol. 3018, pp. 214–232. Springer, Heidelberg (2004)
16. Frisch, A., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: ESSENCE: A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)

17. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., de la Banda, M.G., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (2008)
18. Goltz, H.J., Matzke, D.: University timetabling using constraint logic programming. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 320–334. Springer, Heidelberg (1999)
19. Perri, S., Scarcello, F., Catalano, G., Leone, N.: Enhancing DLV instantiator by backjumping techniques. Ann. Math. Artif. Intell. 51(2-4), 195–228 (2007)
20. Huang, J.: Universal Booleanization of constraint models. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 144–158. Springer, Heidelberg (2008)
21. Niemelä, I.: Stable models and difference logic. Ann. Math. Artif. Intell. 53(1-4), 313–329 (2008)