Solving Modular Model Expansion Tasks

Shahab Tasharrofi, Xiongnan (Newman) Wu, Eugenia Ternovska

Simon Fraser University
{sta44,xwa33,ter}@cs.sfu.ca

Abstract. The work we describe here is a part of a research program of developing foundations of declarative solving of search problems. We consider the model expansion task as the task representing the essence of search problems where we are given an instance of a problem and are searching for a solution satisfying certain properties. Such tasks are common in artificial intelligence, formal verification, computational biology. Recently, the model expansion framework was extended to deal with multiple modules. In the current paper, inspired by practical combined solvers, we introduce an algorithm to solve model expansion tasks for modular systems. We show that our algorithm closely corresponds to what is done in practice in different areas such as Satisfiability Modulo Theories (SMT), Integer Linear Programming (ILP), Answer Set Programming (ASP).

1 Introduction

The research described in this paper is a part of a research program of developing formal foundations for specification/modelling languages (declarative programming) for solving computationally hard problems. In [1], the authors formalize search problems as the logical task of *model expansion (MX)*, the task of expanding a given (mathematical) structure with new relations. They started a research program of finding common underlying principles of various approaches to specifying and solving search problems, finding appropriate mathematical abstractions, and investigating complexity-theoretic and expressiveness issues. It was emphasized that it is important to understand the expressiveness of a specification language in terms of the computational complexity of the problems it can represent. Complexity-theoretic aspects of model expansion for several logics in the context of related computational tasks of satisfiability and model checking were studied in [2]. Since built-in arithmetic is present in all realistic modelling languages, it was important to formalize built-in arithmetic in such languages. In [3], model expansion ideas were extended to provide mathematical foundation for dealing with arithmetic and aggregate functions (min, sum etc.). There, the instance and expansion structures are embedded into an infinite structure of arithmetic, and the property of capturing NP was proven for a logic which corresponds to practical languages. The proposed formalism applies to other infinite background structures besides arithmetic. The analysis of practical languages was given in [4]. It was proved that certain common problems involving numbers (e.g. integer factorization) are not expressible in the ASP and IDP system languages naturally, and in [5], the authors improved the result of [3] by defining a new logic which unconditionally captures NP over arithmetical structures.

The next step in the development of the MX-based framework is adding modularity concepts. It is convenient from the point of view of a user to be able to split a large problem into subproblems, and to use the most suitable formalism for each part, and thus a unifying semantics is needed. In a recent work [6], a subset of the authors extended the MX framework to be able to represent a modular system. The most interesting aspect of that proposal is that modules can be considered from both model-theoretic and operational view. Under the model-theoretic view, an MX module is a set (or class) of structures, and under the operational view it is an operator, mapping a subset of the vocabulary to another subset. An abstract algebra on MX modules is given, and it allows one to combine modules on abstract model-theoretic level, independently from what languages are used for describing them. Perhaps the most important operation in the algebra is the loop (or feedback) operation, since iteration underlies many solving methods. The authors show that the power of the loop operator is such that the combined modular system can capture all of the complexity class NP even when each module is deterministic and polytime. Moreover, in general, adding loops gives a jump in the polynomial time hierarchy, one step from the highest complexity of the components. It is also shown that each module can be viewed as an operator, and when each module is

(anti-) monotone, the number of the potential solutions can be significantly reduced by using ideas from the logic programming community.

To develop the framework further, we need a method for "solving" modular MX systems. By solving we mean finding structures which are in the modular system, where the system is viewed as a function of individual modules. *Our goal is to come up with a general algorithm which takes a modular system in input and generates its solutions.*

We take our inspiration in how "combined" solvers are constructed in the general field of declarative problem solving. The field consists of many areas such as Integer Linear Programming (ILP), Answer Set Programming (ASP), Satisfiability Modulo Theories (SMT), Satisfiability (SAT), and Constraint Programming (CP), and each of these areas has developed multitudes of solvers, including powerful "combined" solvers such as SMT solvers. Moreover, SMT-like techniques are needed in the ASP community [7]. Our main challenge is to come up with an appropriate mathematical abstraction of "combined" solving. Our contributions are as follows.

1. We formalize common principles of "combined" solving in different communities in the context of modular model expansion. Just as in [6], we use a combination of a model-theoretic, algebraic and operational view of modular systems.
2. We design an abstract algorithm that given a modular system, computes the models of that modular system iteratively, and we formulate conditions on languages of individual modules to participate in the iterative solving. We use the formalization above of these common principles to show the effectiveness of our algorithm.
3. We introduce abstractions for many ideas in practical systems such as the concept of a *valid acceptance procedure* that abstractly represents unit propagation in SAT, well-founded model computation in ASP, arc-consistency checkers in CP, etc.
4. As a proof of concept, we show that, in the context of the model expansion task, our algorithm generalizes the work of different solvers from different communities in a unifying and abstract way. In particular, we investigate the branch-and-cut technique in ILP and methods used in SMT, DPLL(Agg) and combinations of ASP and CP [8–11]. We aim to show that, although no implementation is presented, the algorithm should work fine as it mimics the current technology.
5. We develop an improvement of our algorithm by using approximation methods proposed in [6].

2 Background

2.1 Model Expansion

In [1], the authors formalize combinatorial search problems as the task of *model expansion (MX)*, the logical task of expanding a given (mathematical) structure with new relations. Formally, the user axiomatizes the problem in some logic $\mathcal{L}$. This axiomatization relates an instance of the problem (a *finite structure*, i.e., a universe together with some relations and functions), and its solutions (certain *expansions* of that structure with new relations or functions). Logic $\mathcal{L}$ corresponds to a specification/modelling language. It could be an extension of first-order logic such as FO(ID), or an ASP language, or a modelling language from the CP community such as ESSENCE [12].

Recall that a vocabulary is a set of non-logical (predicate and function) symbols. An interpretation for a vocabulary is provided by a *structure*, which consists of a set, called the domain or universe and denoted by $dom(.)$, together with a collection of relations and (total) functions over the universe. A structure can be viewed as an *assignment* to the elements of the vocabulary. An expansion of a structure $\mathcal{A}$ is a structure $\mathcal{B}$ with the same universe, and which has all the relations and functions of $\mathcal{A}$, plus some additional relations or functions. The task of model expansion for an arbitrary logic $\mathcal{L}$ (abbreviated $\mathcal{L}$-MX), is:

Model Expansion for logic $\mathcal{L}$
Given: *1.* An $\mathcal{L}$-formula ϕ with vocabulary $\sigma \cup \varepsilon$
2. A structure $\mathcal{A}$ for σ
Find: an expansion of $\mathcal{A}$, to $\sigma \cup \varepsilon$, that satisfies ϕ.

Thus, we expand the structure $\mathcal{A}$ with relations and functions to interpret ε, obtaining a model B of ϕ. We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary[1].

Example 1. The following formula ϕ of first order logic constitutes an MX specification for Graph 3-colouring:

$$\begin{gathered}\forall x\,[(R(x) \vee B(x) \vee G(x)) \\ \wedge\neg((R(x) \wedge B(x)) \vee (R(x) \wedge G(x)) \vee (B(x) \wedge G(x)))] \\ \wedge\, \forall x \forall y\,[E(x,y) \supset (\neg(R(x) \wedge R(y)) \\ \wedge\neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y)))].\end{gathered}$$

An instance is a structure for vocabulary $\sigma = \{E\}$, i.e., a graph $\mathcal{A} = \mathcal{G} = (V; E)$. The task is to find an interpretation for the symbols of the expansion vocabulary $\varepsilon = \{R, B, G\}$ such that the expansion of $\mathcal{A}$ with these is a model of ϕ:

$$\underbrace{\overbrace{(V; E^{\mathcal{A}}}^{\mathcal{A}},\ R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})}_{\mathcal{B}} \models \phi.$$

The interpretations of ε, for structures $\mathcal{B}$ that satisfy ϕ, are exactly the proper 3-colourings of $\mathcal{G}$.

Given a specification, we can talk about a set of $\sigma \cup \varepsilon$-structures which satisfy the specification. Alternatively, we can simply talk about a set of $\sigma \cup \varepsilon$-structures as an MX-task, without mentioning a particular specification the structures satisfy. This abstract view makes our study of modularity language-independent.

2.2 Modular Systems

This section reviews the concept of a modular system defined in [6] based on the initial development in [13]. As in [6], *each modular system abstractly represents an MX task*, i.e., a set (or class) of structures over some instance and expansion vocabulary. A modular system is formally described as a set of primitive modules (individual MX tasks) combined using the operations of:

1. Projection($\pi_\tau(M)$) which restricts the vocabulary of a module,
2. Composition($M_1 \rhd M_2$) which connects outputs of M_1 to inputs of M_2,
3. Union($M_1 \cup M_2$),
4. Feedback($M[R = S]$) which connects output S of M to its inputs R and,
5. Intersection($M_1 \cap M_2$).

Formal definitions of these operations are not essential for understanding this paper, thus, we refer the reader to [6] for details. The algebraic operations are illustrated in Examples 2 and 3. In this paper, we only consider modular systems which do not use the union operator.

Our goal in this paper is to solve the MX task for a given modular system, i.e., given a modular system M (described in algebraic terms using the operations above) and structure $\mathcal{A}$, find a structure $\mathcal{B}$ in M expanding $\mathcal{A}$. We find our inspiration in existing solver architectures by viewing them at a high level of abstraction.

Example 2 (Timetabling [13]). Here, we use the example of timetabling from [13] and modify its representation using our additional feedback operator. Figure 1 shows the new modular representation of the timetabling problem where the event data and the resource data are the inputs and a list of events with their associated sessions and resources (locations) is the output. This timetabling is done so that the allocations of resource and sessions to the events do not conflict. Unlike [13] where the "allDifferent" module is completely independent of the "testAllocation" module, here, through our feedback operator, these modules are inter-dependent. This inter-dependency provides a better model of the whole system by making the model closer to the reality. Also, here, unlike [13] module "allDifferent" can be a deterministic module. In fact, as proved in [6], the non-determinacy of all NP problems can be modeled through the feedback operator. As will be shown later in this paper, the existence of such loops can also help us to speed up the solving process of some problems.

[1] By ":=" we mean "is by definition" or "denotes".

In this paper, we propose an algorithm such that: given a modular system as in Figure 1 and given the inputs to this modular system, the algorithm finds a solution to (or a model of) the given modular system, i.e., an interpretation to the symbol "occurs" on this example that is not in conflict with the constraints on the timetable.

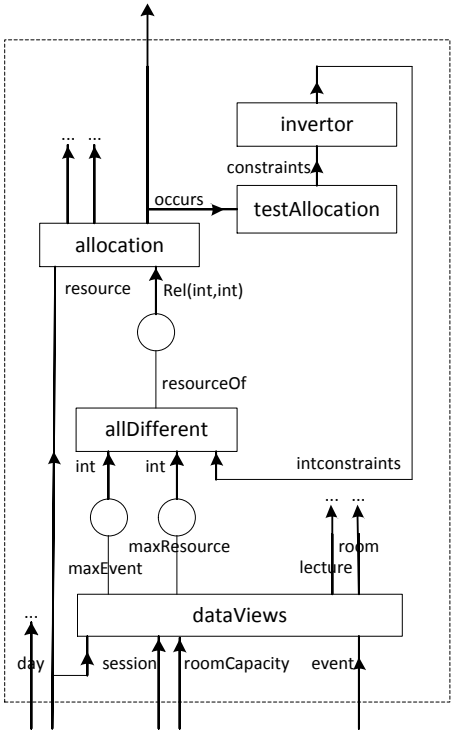

Fig. 1. Modular System Representing a Timetabling Problem

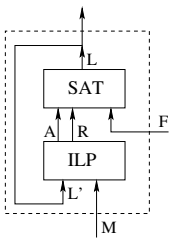

Fig. 2. Modular System Representing an SMT Solver for the Theory of Integer Linear Arithmetic

Example 3 (SMT Solvers). Consider Figure 2: The inner boxes (with solid borders) show simpler MX modules and the outer box shows our module of interest. The vocabulary consists of all symbols A, R, L, L', M and F where A, R and L' are internal to the module, and others form its interface. Also, there is a feedback from L to L'.

Overall, this modular system describes a simple SMT solver for the theory of Integer Linear Arithmetic (T_{ILA}). Our two MX modules are SAT and ILP. They work on different parts of a specification. The ILP module takes a set L' of literals and a mapping M from atoms to linear arithmetic formulas. It returns two sets R and A. Semantically, R represents a set of subsets of L' so that $T_{ILA} \cup M|_r$[2] is *unsatisfiable* for all subsets $r \in R$. Set A represents a set of propagated literals together with their justifications, i.e., a set of pairs (l, Q) where l is an unassigned literal (i.e., neither $l \in L'$ nor $\neg l \in L'$) and Q is a set of assigned literals asserting $l \in L'$, i.e., $Q \subseteq L'$ and $T_{ILA} \cup M|_Q \models M|_l$ (the ILA formula $M|_l$ is a logical consequence of ILA formulas $M|_Q$). The SAT module takes R and A and a propositional formula F and returns set L of literals such that: (1) L makes F true, (2) L is not a superset of any $r \in R$ and, (3) L respects all propagations (l, Q) in A, i.e., if $Q \subseteq L$ then $l \in L$. Using these modules and our operators, module SMT is defined as below to represent our simple SMT solver:

$$SMT := \pi_{\{F,M,L\}}((ILP \rhd SAT)[L = L']). \tag{1}$$

The combined module SMT is correct because, semantically, L satisfies F and all models in it should have $R = \emptyset$, i.e., $T_{ILA} \cup M|_L$ is satisfiable. This is because ILP contains structures for which if $r \in R$, then $r \subseteq L' = L$. Also, for structures in SAT, if $r \in R$ then $r \not\subseteq L$. Thus, to satisfy both these conditions, R has to be empty. Also, one can easily see that all sets L which satisfy F and make $T_{ILA} \cup M|_L$ satisfiable are solutions to this modular system (set $A = R = \emptyset$ and $L' = L$).

So, there is a one-to-one correspondence between models of the modular system above and SMT's solutions to the propositional part. To find a solution, one can compute a model of this modular system. Note that, looking at modules as operators, all models of module SMT are its fixpoints.

A description of a modular system (1) looks like a formula in some logic. One can define a satisfaction relation for that logic, however it is not needed here. Still, since each modular system is a set of structures, we call the structures in a modular system *models* of that system. We are looking for models of a modular system M which expand a given instance structure $\mathcal{A}$. We call them *solutions of M for $\mathcal{A}$*.

3 Computing Models of Modular Systems

In this section, we introduce an algorithm which takes a modular system M and a structure $\mathcal{A}$ and finds an expansion $\mathcal{B}$ of $\mathcal{A}$ in M. Our algorithm uses a tool external to the modular system (a solver). It uses modules of a modular system to "assist" the solver in finding a model (if one exists). Starting from an empty expansion of $\mathcal{A}$ (i.e., a partial structure which contains no information about the expansion predicates), the solver gradually extends the current structure (through an interaction with the modules of the given modular system) until it either finds a model that satisfies the modular system or concludes that none exists. To model this procedure, a definition of a partial structure is needed.

3.1 Partial Structures

Recall that a structure is a domain together with an interpretation of a vocabulary. A partial structure, however, may contain unknown values. For example, for a structure $\mathcal{B}$ and a unary relation R, we may know that $\langle 0 \rangle \in R^{\mathcal{B}}$ and $\langle 1 \rangle \notin R^{\mathcal{B}}$, but we may not know whether $\langle 2 \rangle \in R^{\mathcal{B}}$ or $\langle 2 \rangle \notin R^{\mathcal{B}}$. Partial structures deal with gradual accumulation of knowledge.

Definition 1 (Partial Structure). *We say $\mathcal{B}$ is a τ_p-partial structure over vocabulary τ if:*

1. *$\tau_p \subseteq \tau$,*
2. *$\mathcal{B}$ gives a total interpretation to symbols in $\tau \backslash \tau_p$ and,*
3. *for each n-ary symbol R in τ_p, $\mathcal{B}$ interprets R using two sets R^+ and R^- such that $R^+ \cap R^- = \emptyset$, and $R^+ \cup R^- \subsetneq (dom(\mathcal{B}))^n$.*

[2] For a set τ of literals, $M|_\tau$ denotes a set of linear arithmetical formulas containing: (1) M's image of positive atoms in τ, and (2) the negation of M's image of negative atoms in τ.

We say that τ_p is the partial vocabulary of $\mathcal{B}$. If $\tau_p = \emptyset$, then we say $\mathcal{B}$ is total. For two partial structures $\mathcal{B}$ and $\mathcal{B}'$ over the same vocabulary and domain, we say that $\mathcal{B}'$ extends *$\mathcal{B}$ if all unknowns in $\mathcal{B}'$ are also unknowns in $\mathcal{B}$, i.e., $\mathcal{B}'$ has at least as much information as $\mathcal{B}$.*

If a partial structure $\mathcal{B}$ has enough information to satisfy or falsify a formula ϕ, then we say $\mathcal{B} \models \phi$, or $\mathcal{B} \models \neg\phi$, respectively. Note that for partial structures, $\mathcal{B} \models \neg\phi$ and $\mathcal{B} \not\models \phi$ may be different. We call a ε-partial structure $\mathcal{B}$ over $\sigma \cup \varepsilon$ the *empty expansion* of σ-structure $\mathcal{A}$, if $\mathcal{B}$ agrees with $\mathcal{A}$ over σ but $R^+ = R^- = \emptyset$ for all $R \in \varepsilon$.

In the following, by structure we always mean a total structure, unless otherwise specified. We may talk about "bad" partial structures which, informally, are the ones that cannot be extended to a structure in M. Also, when we talk about a τ_p-partial structure, in the MX context, τ_p is always a subset of ε.

Total structures are partial structures with no unknown values. Thus, in the algorithmic sense, total structures need no further guessing and should only be checked against the modular system. A good algorithm rejects "bad" partial structures sooner, i.e., the sooner a "bad" partial structure is detected, the faster the algorithm is.

Up to now, we defined partial and total structures and talked about modules rejecting "bad" partial structures. However, modules are sets of structures (in contrast with sets of partial structures). Thus, acceptance of a partial structure has to be defined properly. Towards this goal, we first formalize the informal concept of "good" partial structures. The actual acceptance procedure for partial structures is defined later in the section.

Definition 2 (Good Partial Structures). *For a set of structures S and partial structure $\mathcal{B}$, we say $\mathcal{B}$ is a* good partial structure wrt *S if there is $\mathcal{B}' \in S$ which extends $\mathcal{B}$.*

3.2 Requirements on the Modules

Untill now, we have the concept of partial structures that the solver can work on, but, clearly, as the solver does not have any information about the internals of the modules, it needs to be assisted by the modules. Therefore, the next question could be: "what assistance does the solver need from modules so that its correctness is always guaranteed?" Intuitively, modules should be able to tell whether the solver is on the "right" direction or not, i.e., whether the current partial structure is bad, and if so, tell the solver to stop developing this direction further. We accomplish this goal by letting a module accept or reject a partial structure produced by the solver and, in the case of rejection, provide a "reason" to prevent the solver from producing the same model later on. Furthermore, a module may "know" some extra information that solver does not. Due to this fact, modules may give the solver some hints to accelerate the computation in the current direction. Our algorithm models such hints using "advices" to the solver.

Note that reasons and advices we are now talking about are different from predicate symbols R and A in Example 3. While, conceptually, R and A also represent reasons and advices there, to our algorithm, they are just predicate symbols for which an interpretation has to be found. On the other hand, reasons and advices used by our algorithm are not specific to a modular system. They are entities known to our algorithm which contain information to guide the solver in its search for a model.

Also note that in order to pass a reason or an advice to a solver, there should be a common language that the solver and the modules understand (although it may be different from all internal languages of the modules). We expect this language to have its own model theory and to support basic syntax such as conditionals or negations. We expect the model theory of this language to 1) be monotone: adding a sentence can not decrease the set of consequences and 2) have resolution theorem which is the converse of the deduction theorem, i.e., if $\Gamma \models A \supset B$ then $\Gamma \cup \{A\} \models B$. The presence of the resolution theorem guarantees that, once an advice of form $Pre \supset Post$ is added to the solver, and when the solver has deduced Pre under some assumptions, it can also deduce $Post$ under the same assumptions. From now on, we assume that our advices and reasons are expressed in such a language.

We talked about modules assisting the solver, but a module is a set of structures and has no computational power. Instead, we associate each module with an "oracle" to accept/reject a partial structure and give "reasons" and "advices" accordingly. Note that it is unreasonable to require a strong acceptance condition from oracles because, for example, assuming access to oracles which accept a partial structure iff it

is a good partial structure, one can always find a total model by polynomially many queries to such oracles. While theoretically possible, in practice, access to such oracles is usually not provided. Thus, we have to (carefully) relax our assumptions for a weaker procedure (what we call a Valid Acceptance Procedure).

Definition 3 (Advice). *Let Pre and $Post$ be formulas in the common language of advices and reasons, Formula $\phi := Pre \supset Post$ is an* advice *wrt a partial structure $\mathcal{B}$ and a set of structures M if:*

1. *$\mathcal{B} \models Pre$,*
2. *$\mathcal{B} \not\models Post$ and,*
3. *for every total structure $\mathcal{B}'$ in M, we have $\mathcal{B}' \models \phi$.*

The role of an advice is to prune the search and to accelerate extending a partial structure $\mathcal{B}$ by giving a formula that is not yet satisfied by $\mathcal{B}$, but is always satisfied by any total extensions of $\mathcal{B}$ in M. Pre corresponds to the part that is satisfied by $\mathcal{B}$ and $Post$ corresponds to the unknown part that is not yet satisfied by $\mathcal{B}$.

Definition 4 (Valid Acceptance Procedure). *Let S be a set of τ-structures. We say that P is a valid acceptance procedure for S if for all τ_p-partial structures $\mathcal{B}$, we have:*

- *If $\mathcal{B}$ is total, then if $\mathcal{B} \in S$, then P accepts $\mathcal{B}$, and if $\mathcal{B} \notin S$, then P rejects $\mathcal{B}$.*
- *If $\mathcal{B}$ is not total but $\mathcal{B}$ is good wrt S, then P accepts $\mathcal{B}$.*
- *If $\mathcal{B}$ is neither total nor good wrt $\mathcal{B}$, then P is free to either accept or reject $\mathcal{B}$.*

The procedure above is called valid as it never rejects any good partial structures. However, it is a weak acceptance procedure because it may accept some bad partial structures. This kind of weak acceptance procedures are abundant in practice, e.g., Unit Propagation in SAT, Arc-Consistency Checks in CP, and computation of Founded and Unfounded Sets in ASP. As these examples show, such weak notions of acceptance can usually be implemented efficiently as they only look for local inconsistencies. Informally, oracles accept/reject a partial structure through a valid acceptance procedure for a set containing all possible instances of a problem and their solutions. We call this set a Certificate Set.

In theoretical computer science, a problem is a subset of $\{0,1\}^*$. In logic, a problem corresponds to a set of structures. Here, we use the logic notion.

Definition 5 (Certificate Set). *Let σ and ε be instance and expansion vocabularies. Let $\mathcal{P}$ be a problem, i.e., a set of σ-structures, and C be a set of $(\sigma \cup \varepsilon)$-structures. Then, C is a* $(\sigma \cup \varepsilon)$-certificate set for $\mathcal{P}$ *if for all σ-structures $\mathcal{A}$: $\mathcal{A} \in \mathcal{P}$ iff there is a structure $\mathcal{B} \in C$ that expands $\mathcal{A}$.*

Oracles are the interfaces between our algorithm and our modules. Next we present conditions that oracles should satisfy so that their corresponding modules can contribute to our algorithm.

Definition 6 (Oracle Properties). *Let $\mathcal{L}$ be a formalism with our desired properties. Let $\mathcal{P}$ be a problem, and let O be an oracle. We say that O is*

- Complete and Constructive (CC) wrt $\mathcal{L}$ *if O returns a reason $\psi_{\mathcal{B}}$ in $\mathcal{L}$ for each partial structure $\mathcal{B}$ that it rejects such that: (1) $\mathcal{B} \models \neg\psi_{\mathcal{B}}$ and, (2) all total structures accepted by O satisfy $\psi_{\mathcal{B}}$.*
- Advising (A) wrt $\mathcal{L}$ *if O provides a set of advices in $\mathcal{L}$ wrt $\mathcal{B}$ for all partial structures $\mathcal{B}$.*
- Verifying (V) *if O is a valid acceptance procedure for some certificate set C for P.*

Oracle O is *complete wrt* $\mathcal{L}$ because it ensures the existence of such a sentence and *constructive* because it provides such a sentence. Oracle O differs from the usual oracles in the sense that it does not only give yes/no answers, but also provides reasons for why the answer is correct. It is *advising* because it provides some facts that were previously unknown to guide the search. Finally, it is *verifying* because it guides the partial structure to a solution through a valid acceptance procedure. Although the procedure can be weak as described above, good partial structures are never rejected and O always accepts or rejects total structures correctly. This property guarantees the convergence to a total model. In the following sections, we use the term CCAV oracle to denote an oracle which is complete, constructive, advising, and verifying.

3.3 Requirements on the Solver

In this section, we discuss properties that a solver needs to satisfy. Although the solver can be realized by many practical systems, for them to work in an orderly fashion and for algorithm to converge to a solution fast, it has to satisfy certain properties. First, the solver has to be online since the oracles keep adding reasons and advices to it. Furthermore, to guarantee the termination, the solver has to guarantee progress, which means it either reports a proper extension of the previous partial structure or, if not, the solver is guaranteed to never return any extension of that previous partial structure later on. Now, we give the requirements on the solver formally.

Definition 7 (Complete Online Solver). *A solver S is* complete and online *if the following conditions are satisfied by S:*

- *S supports the actions of initialization, adding sentences, and reporting its state as either $\langle UNSAT \rangle$ or $\langle SAT, \mathcal{B} \rangle$.*
- *If S reports $\langle UNSAT \rangle$ then the set of sentences added to S are unsatisfiable,*
- *If S reports $\langle SAT, \mathcal{B} \rangle$ then $\mathcal{B}$ does not falsify any of the sentences added to S,*
- *If S has reported $\langle SAT, \mathcal{B}_1 \rangle, \cdots, \langle SAT, \mathcal{B}_n \rangle$ and $1 \leq i < j \leq n$, then either $\mathcal{B}_j$ is a proper extension of $\mathcal{B}_i$ or, for all $k \geq j$, $\mathcal{B}_k$ does not extend $\mathcal{B}_i$.*

A solver as above is guaranteed to be sound (it returns partial structures that at least do not falsify any of the constraints) and complete (it reports unsatisfiability only when unsatisfiability is detected and not when, for example, some heuristic has failed to find an answer or some time limit is reached). Also, for finite structures, such a solver guarantees that our algorithm either reports unsatisfiability or finds a solution to modular system M and instance structure $\mathcal{A}$.

3.4 Lazy Model Expansion Algorithm

In this section, we present an iterative algorithm to solve model expansion tasks for modular systems. Algorithm 1 takes an instance structure and a modular system (and its CCAV oracles) and integrates them with a complete online solver to iteratively solve a model expansion task. The algorithm works by accumulating reasons and advices from oracles and gradually converging to a solution to the problem.

The role of the reasons is to prevent some bad structures and their extensions from being proposed more than once, i.e., when a model is deducted to be bad by an oracle, a new reason is provided by the oracle and added to the solver such that all models of the system satisfy that reason but the "bad" structure does not. The role of an advice is to provide useful information to the solver (satisfied by all models) but not yet satisfied by partial structure $\mathcal{B}$. Informally, an advice is in form "if Pre then Post", where "Pre" corresponds to something already satisfied by current partial structure $\mathcal{B}$ and "Post" is something that is always satisfied by all models of the modular system satisfying the "Pre" part, but not yet satisfied by partial structure $\mathcal{B}$. It essentially tells the solver that "Post" part is satisfied by all intended structures (models of the system) extending $\mathcal{B}$, thus helping the solver to accelerate its computation in its current direction.

The role of the solver is to provide a possibly good partial structure to the oracles, and if none of the oracles rejects the partial structure, keep extending it until we find a solution or conclude none exists. If the partial structure is rejected by any one of the oracles, the solver gets a reason from the oracle for the rejection and tries some other partial structures. The solver also gets advices from oracles to accelerate the search.

4 Examples: Modelling Existing Frameworks

In this section, we describe algorithms from three different areas and show that they can be effectively modelled by our proposed algorithm in the context of model expansion. Note that our purpose here is not to analyze other systems but to show the effectiveness of our algorithm in the absence of an implementation. We establish this claim by showing that our algorithm acts similar to the state-of-the-art algorithms when the right components are provided.

Data: Modular System M with each module M_i associated with a CCAV oracle O_i, input structure $\mathcal{A}$ and complete online solver S
Result: Structure $\mathcal{B}$ that expands $\mathcal{A}$ and is in M
begin
  Initialize the solver S using the empty expansion of $\mathcal{A}$;
  while *TRUE* **do**
    Let R be the state of S ;
    if $R = \langle UNSAT \rangle$ **then return** *Unsatisfiable* ;
    else if $R = \langle SAT, \mathcal{B} \rangle$ **then**
      Add the set of advices from oracles wrt $\mathcal{B}$ to S ;
      if M *does not accept* $\mathcal{B}$ **then**
        Find a module M_i in M such that M_i does not accept $\mathcal{B}|_{vocab(M_i)}$;
        Let ψ be the reason given by oracle O_i ;
        Add ψ to S ;
      else if $\mathcal{B}$ *is total* **then return** $\mathcal{B}$;
end

Algorithm 1: Lazy Model Expansion Algorithm

4.1 Modelling DPLL(T)

DPLL(T) [14] system is an abstract framework to model the lazy SMT approach. It is based on a general DPLL(X) engine, where X can be instantiated with a theory T solver. DPLL(T) engine extends the Decide, UnitPropagate, Backjump, Fail and Restart actions of the classic DPLL framework with three new actions: (1) **TheoryPropagate** gives literals that are T-consequences of current partial assignment, (2) T**-Learn** learns T-consistent clauses, and (3) T**-Forget** forgets some previous lemmas of theory solver.

To participate in DPLL(T) solving architecture, a theory solver provides three operations: (1) taking literals that have been set true, (2) checking if setting these literals true is T-consistent and, if not, providing a subset of them that causes inconsistency, (3) identifying some currently undefined literals that are T-consequences of current partial assignment and providing a justification for each. More details can be found in [14].

The modular system $DPLL(T)$ of the DPLL(T) system is the same as the one in Example 3, except that we have module M_P instead of SAT and M_T instead of ILP. In Figure 2, A corresponds to the result of TheoryPropagate action that contains some information about currently undefined values in L' together with their justifications; R is calculated from the T-Learn action and corresponds to reasons of M_T rejecting L'.

To model DPLL(T), we introduce a solver S to be any DPLL-based online SAT solver, so that it performs the basic actions of Decide, UnitPropagate, Fail, Restart, and also Backjump when the backjumping clause is added the solver. The two modules M_T and M_P are attached with oracles O_T and O_P respectively. They accept a partial structure $\mathcal{B}$ iff their respective module constraints is not falsified by $\mathcal{B}$. When rejecting $\mathcal{B}$, a reason "P_{in} then P_{out}" (true about all models of the module) is returned where P_{in} (resp. P_{out}) is a property about input (resp. output) vocabulary of the module satisfied (resp. falsified) by $\mathcal{B}$. They may also return advices of the same form but with P_{out} being neither satisfied nor falsified by $\mathcal{B}$. The constructions of these two modules are similar; so, we only give a construction for the solver S and module M_T:

Solver S is a DPLL-based online SAT solver (clearly complete and online).

Module M_T The associated oracle O_T accepts a partial structure $\mathcal{B}$ if it does not falsify the constraints described in Example 3 on L', M, A, and R for module M_T. If $\mathcal{B}$ is rejected, O_T returns a reason $\psi := \psi_{in} \supset \psi_{out}$ where $\mathcal{B}|_{\{L',M\}} \models \psi_{in}$ but $\mathcal{B}|_{\{A,R\}} \models \neg\psi_{out}$. Clearly, $\mathcal{B} \models \neg\psi$ and all models in M_T satisfy ψ. Thus, O_T is complete and constructive. O_T may also return some advices which are similar to the reason above except that ψ_{out} is neither satisfied nor falsified by $\mathcal{B}$. Hence, O_T is an advising oracle. Also, O_T always makes the correct decision for a total structure and rejects a partial structure only when it falsifies the constraints for M_T. O_T never rejects any good partial structure $\mathcal{B}$ (although it may accept some bad non-total structures). Therefore, O_T is a valid acceptance procedure for M_T and, thus, a verifying oracle.

Proposition 1. *1. Modular system $DPLL(T)$ models the DPLL(T) system. 2. Solver S is complete and online. 3. O_P and O_T are CCAV oracles.*

DPLL(T) architecture is known to be very efficient and many solvers are designed to use it, including most SMT solvers [9]. The DPLL(Agg) module [10] is suitable for all DPLL-based SAT, SMT and ASP solvers to check satisfiability of aggregate expressions in DPLL(T) context. All these systems are representable in our modular framework.

4.2 Modelling ILP Solvers

Integer Linear Programming solvers solve optimization problems. In this paper, we model ILP solvers which use general branch-and-cut method to solve *search* problems instead, i.e., when target function is constant. We show that Algorithm 1 models such ILP solvers. ILP solvers with other methods and Mixed Integer Linear Programming solvers use similar architectures and, thus, can be modelled similarly.

The search version of general branch-and-cut algorithm [8] is as follows:

1. Initialization: $S = \{\text{ILP}^0\}$ with ILP^0 the initial problem.
2. Termination: If $S = \emptyset$, return UNSAT.
3. Problem Select: Select and remove problem ILP^i from S.
4. Relaxation: Solve LP relaxation of ILP^i (as a search problem). If infeasible, go to step 2. Otherwise, if solution X^{iR} of LP relaxation is integral, return solution X^{iR}.
5. Add Cutting Planes: Add a cutting plane violating X^{iR} to relaxation and go to 4.
6. Partitioning: Find partition $\{C^{ij}\}_{j=1}^{j=k}$ of constraint set C^i of problem ILP^i. Create k subproblems ILP^{ij} for $j = 1, \cdots, k$, by restricting the feasible region of subproblem ILP^{ij} to C^{ij}. Add those k problems to S and go to step 2. Often, in practice, finding a partition is simplified by picking a variable x_i with non-integral value v_i in X^{iR} and returning partition $\{C^i \cup \{x_i \leq \lfloor v_i \rfloor\}, C^i \cup \{x_i \geq \lceil v_i \rceil\}\}$.

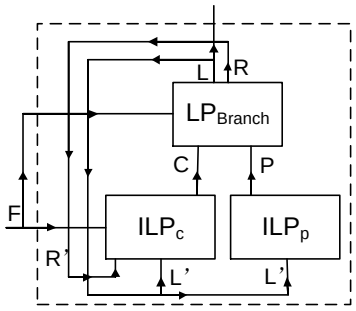

Fig. 3. Modular System Representing an ILP Solver

We use the modular system shown in figure 3 to represent the ILP solver. The ILP_c module takes the problem specification F, a set of assignments L' and a set of range information R' (which, in theory, describes assumptions about ranges of variables and, in practice, describes the branch information of an LP+Branch solver) as inputs and returns a set C. When all the assignments in L' are integral, C is empty, and if not, C represents a set of cutting planes conditioned by a subset of range information R', i.e., set of linear constraints that are violated by L' and all the set of assignments satisfy both F and R' also satisfy the set of cutting planes. The ILP_p module only takes the set of assignments L' as input and outputs a set of partitioning clauses P, such that when all the assignments in L' is integral, P is empty and when there is a non-integral assignment to a variable x, P is a set of partitioning clauses indicating that assignment to x should be either less than or equal to $\lfloor L'(x) \rfloor$ or greater than or equal to $\lceil L'(x) \rceil$. The LP_{Branch} module takes F, C and P as inputs and outputs the set of assignment L and the set of range information R such that L satisfies specification F, the range information R, the set of conditional cutting planes in C, and the set of partitioning clauses in P. We define the compound module ILP to be:

$$ILP := \pi_{\{F,L\}}(((ILP_c \cap ILP_p) \rhd LP_{Branch})[L = L'][R = R']).$$

The module ILP defined above is correct because all models satisfying it should have $C = B = \emptyset$ because, ILP_c contains structures in which, for every $(S, c) \in C$, which denotes cutting plane c under condition S, we have $S \subseteq R' = R$ and c is violated by $L' = L$. Furthermore, LP_{Branch} contains structures in which L satisfies both R and C, which indicates that either S is not the subset of R or L satisfies c. Thus C is empty. By a similar argument, one can prove that B also has to be empty.

We compute a model of this modular system by introducing a solver that interacts with all of the three modules above. We introduce an extended LP solver S which allows us to deal with disjunctions. S performs a depth-first-like search on disjunctive constraints, and runs its internal LP solver on non-disjunctive constraints plus the range information the search branch corresponds to. So, Partitioning action of ILP corresponds to adding a disjunctive constraint to the solver. All three modules above are associated with oracles O_c, O_p and O_{lp}, respectively. Exact constructions are similar to the ones in section 4.1. Here we only give a construction for the solver S:

Solver S is an extended LP solver, i.e., uses an internal LP solver. Let Br denote the set of branch constraints in S (constraints involving disjunctions) and L denote the set of pure linear constraints. When new constraint is added to S, S adds it to Br or L accordingly. S then performs a depth-first-like search on branch clauses, and, at branch i, passes $L \cup Br_i$ (a set of linear constraints specifying branch i) to its internal LP solver. Then, S returns $\langle SAT, \mathcal{B} \rangle$ if LP finds a model $\mathcal{B}$ in some branch, and $\langle UNSAT \rangle$ otherwise. Note that, by construction, S is complete and online.

Proposition 2. *1. Modular system ILP models the branch-and-cut-based ILP solver. 2. S is complete and online. 3. O_c, O_p and O_{lp} are CCAV oracles.*

There are many other solvers in ILP community that use some ILP or MILP solver as their low-level solver. It is not hard to observe that most of them also have similar architectures that can be closely mapped to our algorithm.

4.3 Modelling Constraint Answer Set Solvers

The Answer Set Programming (ASP) community puts a lot of effort into optimizing their solvers. One such effort addresses ASP programs with variables ranging over huge domains (for which, ASP solvers alone perform poorly due to the huge memory the grounding uses). However, embedding Constraint Programming (CP) techniques into ASP solving is proved useful because grounding such variables is partially avoided.

In [15], the authors extend the language of ASP and its reasoning method to avoid grounding of variables with large domains by using constraint solving techniques. The algorithm uses ASP and CP solvers as black boxes and non-deterministically extends a partial solution to the ASP part and checks it with the CP solver. Paper [16] presents another integration of answer set generation and constraint solving in which a traditional DPLL-like backtracking algorithm is used to embed the CP solver into the ASP solving.

Recently, the authors of [11] developed an improved hybrid solver which supports advanced backjumping and conflict-driven no good learning (CDNL) techniques. They show that their solver's performance is comparable to state-of-the-art SMT solvers. Paper [11] applies a partial grounding before running its algorithm, thus, it uses an algorithm on propositional level. A brief description of this algorithm follows: Starting from an empty set of assignments and nogoods, the algorithm gradually extends the partial assignments by both unit propagation in ASP and constraint propagation in CP. If a conflict occurs (during either unit propagation or constraint propagation), a nogood containing the corresponding unique implication point (UIP) is learnt and the algorithm backjumps to the decision level of the UIP. Otherwise, the algorithm decides on the truth value of one of the currently unassigned atoms and continues to apply the propagation. If the assignment becomes total, the CP oracle queries to check whether this is indeed a solution for the corresponding constraint satisfaction problem (CSP). This step is necessary because simply performing constraint propagation on the set of constraints, i.e., arc-consistency checking, is not sufficient to decide the feasibility of constraints.

The modular model of this solver is very similar to the one in Figure 2, except that we have module ASP instead of SAT and CP instead of ILP. The compound module $CASP$ is defined as:

$$CASP := \pi_{\{F,M,L\}}((CP \rhd ASP)[L = L']).$$

As a CDNL-like technique is also used in SMT solvers, the above algorithm is modelled similarly to Section 4.1. We define a solver S to be a CDNL-based ASP solver. We also define modules ASP and CP to deal with the ASP part and the CP part. They are both associated oracles similar to those described in Section 4.1. We do not include the details here as they are similar to the ones in section 4.1.

Note that one can add reasons and advices to an ASP solver safely in the form of conflict rules because stable model semantics is monotonic with respect to such rules. Also, practical CP solvers do not provide reasons for rejecting partial structures. This issue is dealt with in [11] by wrapping CP solvers with a conflict analysis mechanism to compute nogoods based on the first UIP scheme.

5 Extension: Approximations

Almost all practical solvers use some kind of propagation technique. However, in a modular system, propagation is not possible in general because nothing is known in advance about a module. According to [6], it turns out that knowing only some general information about modules such as their totality and monotonicity or anti-monotonicity, one can hugely reduce the search space.

Moreover, paper [6] proposes two procedures to approximate models of what are informally called positive and negative feedbacks. These procedures correspond to least fixpoint and well-founded model computations (but in modular setting). Here, we extend Algorithm 1 using these procedures. The extended algorithm prunes the search space of a model by propagating information obtained by these approximation procedures to the solver. First, let us define some properties that a module may satisfy.

Definition 8 (Module Properties [6]). *Let M be a module and τ, τ' and τ'' be some subsets of M's vocabulary. M is said to be:*

1. **τ-total over a class C of structures** *if by restricting models of M to vocabulary τ we can obtain all structures in C.*
2. **τ-τ'-τ''-monotone (resp. τ-τ'-τ''-anti-monotone)** *if for all structures $\mathcal{B}$ and $\mathcal{B}'$ in M we have that if $\mathcal{B}|_\tau \sqsubseteq \mathcal{B}'|_\tau$ and $\mathcal{B}|_{\tau'} = \mathcal{B}'|_{\tau'}$ then $\mathcal{B}|_{\tau''} \sqsubseteq \mathcal{B}'|_{\tau''}$ (resp. $\mathcal{B}'|_{\tau''} \sqsubseteq \mathcal{B}|_{\tau''}$).*

In [6], it is shown that these properties are fairly general and that, given such properties about basic MX modules, one can derive similar properties about complex modules.

Now, we can restate the two approximation procedures from [6]. For $\mathcal{B}$ and $\mathcal{B}'$ over the same domain, but distinct vocabularies, let $\mathcal{B}||\mathcal{B}'$ denote the structure over that domain and $voc(\mathcal{B}) \cup voc(\mathcal{B}')$ where symbols in $voc(\mathcal{B})$ [resp. $voc(\mathcal{B}')$] are interpreted as in $\mathcal{B}$ [resp. $\mathcal{B}'$]. We first consider the case of a positive feedback, i.e., when relation R which increases monotonically when S increases is fed back into S itself. This procedure is defined for a module $M' := M[S = R]$ and partial structure $\mathcal{B}$ where M is $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-monotone and $\mathcal{B}$ gives total interpretation to τ. It defines a chain L_i of interpretations for S as follows:

$$\begin{array}{l} L_0 := S^{+^{\mathcal{B}}}, \\ L_{i+1} := R^{M(\mathcal{B}|_\tau \,||\, \mathcal{L})} \text{ where } dom(\mathcal{L}) = dom(\mathcal{A}) \text{ and } S^{\mathcal{L}} = L_i. \end{array} \tag{2}$$

The procedure for a negative feedback is similar, but for M being $\{S\}$-τ-$\{R\}$-anti-monotone. It defines an increasing sequence L_i and a decreasing sequence U_i which say what should be in added to S^+ and S^-. Here, n is the arity of relations R and S:

$$\begin{array}{c} L_0 := S^+, U_0 := [dom(\mathcal{A})]^n \backslash S^-, \\ L_{i+1} := R^{M(\mathcal{B}|_\tau \,||\, \mathcal{U})} \text{ where } dom(\mathcal{U}) = dom(\mathcal{A}) \text{ and } S^{\mathcal{U}} = U_i, \\ U_{i+1} := R^{M(\mathcal{B}|_\tau \,||\, \mathcal{L})} \text{ where } dom(\mathcal{L}) = dom(\mathcal{A}) \text{ and } S^{\mathcal{L}} = L_i. \end{array} \tag{3}$$

Now, Algorithm 1 can be extended to Algorithm 2 which uses the procedures above to find new propagated information which has to be true under the current assumptions, i.e., the current partial structure. This information is sent back to the solver to speed up the search process. Algorithm 2 needs a solver which can get propagated literals.

Data: Similar to Algorithm 1, but with modules' totality, monotonicity and anti-monotonicity properties given
Result: Structure $\mathcal{B}$ expands $\mathcal{A}$ and is in M
begin
  Initialize the solver S using the empty expansion of $\mathcal{A}$;
  while *true* **do**
    Let R be the state of S ;
    if $R = \langle UNSAT \rangle$ **then return** *Unsatisfiable* ;
    else if $R = \langle SAT, \mathcal{B} \rangle$ **then**
      Add the set of advices from oracles wrt $\mathcal{B}$ to S ;
      if M *does not accept* $\mathcal{B}$ **then**
        Find a module M_i in M such that M_i does not accept $\mathcal{B}|_{vocab(M_i)}$;
        Let ψ be the reason given by oracle O_i ;
        Add ψ to S ;
      else if $\mathcal{B}$ *is total* **then return** $\mathcal{B}$;
      else
        foreach *applicable positive feedback* $M_1 := M_2[T = T']$ **do**
          Let L^* be the limit of series L_i in Equation 2 ;
          Propagate $L^* \backslash {T^+}^{\mathcal{B}}$ to S ;
        foreach *applicable negative feedback* $M_1 := M_2[T = T']$ **do**
          Let $\langle L^*, U^* \rangle$ be the limit of series $\langle L_i, U_i \rangle$ in Equation 3 ;
          Propagate $(L^* \backslash {T^+}^{\mathcal{B}}) \cup \mathbf{not}([dom(\mathcal{A})]^n \backslash (U^* \cup {T^-}^{\mathcal{B}}))$ to S ;
end

Algorithm 2: Lazy Model Expansion with Approximation (Propagation)

6 Related Works

This paper is a continuation of [6] and proposes an algorithm for solving model expansion tasks in the modular setting. The modular framework of [6] expands the idea of model theoretic (and thus language independent) modelling of [13] and introduces the feedback operator and discusses some of the consequences (such as complexity implications) of this new operator. There are many other works on modularity in declarative programming that we only briefly review.

An early work on adding modularity to logic programs is [17]. The authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. This is further generalized in [18] by considering the concept of modules in declarative programming and introducing modular equivalence in normal logic programs under the stable model semantics. This line of work is continued in [19] to define modularity for disjunctive logic programs. There are also other approaches to adding modularity to ASP languages and ID-Logic as described in [20–22].

The works mentioned earlier focus on the theory of modularity in declarative languages. However, there are also works that focus on the practice of modular declarative programming and, in particular, solving. These works generally fall into one of the two following categories:

The first category consists of practical modelling languages which incorporate other modelling languages. For example, X-ASP [23] and ASP-PROLOG [24] extend prolog with ASP. Also ESRA [25], ESSENCE [12] and Zinc [26] are CP languages extended with features from other languages. However, these approaches give priority to the host language while our modular setting gives equal weight to all modelling languages that are involved. It is important to note that, even in the presence of this distinction, such works have been very important in the development of this paper because they provide guidelines on how a practical solver deals with efficiency issues. We have emphasized on this point in Section 4.

The second category consists of the works done on multi-context systems. In [27], the authors introduce non-monotonic bridge rules to the contextual reasoning and originated an interesting and active line of research followed by many others for solving or explaining inconsistencies in non-monotonic multi-context systems [28–31]. However, these works do not consider the model expansion task. Moreover, the motivations of these works originate from distributed or partial knowledge, e.g., when agents interact or when trust or privacy issues are important. Despite these differences, the field of multi-context systems is very

relevant to our research. Investigating this connection as well as incorporating results from the research on multi-context system into our framework is our most important future research direction.

7 Conclusion

We addressed the problem of finding solutions to a modular system in the context of model expansion and proposed an algorithm which finds such solutions. We defined conditions on modules such that once satisfied, modules described with possibly different languages can participate in our algorithm. We argued that our algorithm captures the essence of practical solvers, by showing that DPLL(T) framework, ILP solvers and state-of-the-art combinations of ASP and CP are all specializations of our modular framework. We believe that our work bridges work done in different communities and contributes to cross-fertilization of the fields.

References

1. Mitchell, D.G., Ternovska, E.: A framework for representing and solving NP search problems. In: Proc. AAAI. (2005) 430–435
2. Kolokolova, A., Liu, Y., Mitchell, D., Ternovska, E.: On the complexity of model expansion. In: Proc., 17th Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-17), Springer (2010) 447–458 LNCS 6397.
3. Ternovska, E., Mitchell, D.: Declarative programming of search problems with built-in arithmetic. In: Proc. of IJCAI. (2009) 942–947
4. Tasharrofi, S., Ternovska, E.: Built-in arithmetic in knowledge representation languages. In: NonMon at 30 (Thirty Years of Nonmonotonic Reasoning). (October 2010)
5. Tasharrofi, S., Ternovska, E.: PBINT, a logic for modelling search problems involving arithmetic. In: Proc. 17th Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-17), Springer (October 2010) LNCS 6397.
6. Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: 8th International Symposium Frontiers of Combining Systems (FroCoS). (October 2011)
7. Niemelä, I.: Integrating answer set programming and satisfiability modulo theories. In: LPNMR. Volume 5753 of LNCS., Springer-Verlag (2009) 3–3
8. Pardalos, P., Resende, M.: Handbook of applied optimization. Volume 126. Oxford University Press New York; (2002)
9. Sebastiani, R.: Lazy Satisfiability Modulo Theories. JSAT **3** (2007) 141–224
10. De Cat, B., Denecker, M.: DPLL(Agg): An efficient SMT module for aggregates. In: LaSh'10 Workshop. (2010)
11. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proc. of ICLP'09. LNCS, Springer-Verlag (2009) 235–249
12. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13** (2008) 268–306
13. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Proc. of LPNMR. Volume 5753 of LNCS., Springer-Verlag (2009) 155–168
14. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (November 2006) 937–977
15. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. (2005) 52–66
16. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence **53** (2008) 251–287
17. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Proc. of LPNMR, Springer-Verlag (1997) 290–309
18. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: the Proc. of NMR'06. (2006) 10–18
19. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. In: Proc. of LPNMR. Volume 4483 of LNAI. (2007) 175–187
20. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: ICLP'06. LNCS. (2006) 376–390
21. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: SEA. (2007) 41–55

22. Denecker, M., Ternovska, E.: A logic of non-monotone inductive definitions. Transactions on Computational Logic **9**(2) (2008) 1–51
23. Swift, T., Warren, D.S.: The XSB System. (2009)
24. Elkhatib, O., Pontelli, E., Son, T.: ASP- PROLOG: A System for Reasoning about Answer Set Programs in Prolog. In: the Proc. of PADL'04. (2004) 148–162
25. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. Logic Based Program Synthesis and Transformation (2004) 214–232
26. de la Banda, M., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. Principles and Practice of Constraint Programming-CP 2006 700–705
27. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proceedings of the 22nd national conference on Artificial intelligence - Volume 1, AAAI Press (2007) 385–390
28. Bairakdar, S.E.D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The dmcs solver for distributed nonmonotonic multi-context systems. In Janhunen, T., Niemelä, I., eds.: Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings. Volume 6341 of Lecture Notes in Computer Science., Springer (2010) 352–355
29. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The mcs-ie system for explaining inconsistency in multi-context systems. In Janhunen, T., Niemelä, I., eds.: Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings. Volume 6341 of Lecture Notes in Computer Science., Springer (2010) 356–359
30. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in multi-context systems. In Lin, F., Sattler, U., Truszczynski, M., eds.: Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010, AAAI Press (2010)
31. Eiter, T., Fink, M., Schüller, P.: Approximations for explanations of inconsistency in partially known multi-context systems. In Delgrande, J.P., Faber, W., eds.: Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings. Volume 6645 of Lecture Notes in Computer Science., Springer (2011) 107–119