Solving Modular Model Expansion: Case Studies

Shahab Tasharrofi, Xiongnan (Newman) Wu, Eugenia Ternovska

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

Abstract. Model expansion task is 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 AI planning, scheduling, logistics, supply chain management, etc., and are inherently modular. Recently, the model expansion framework was extended to deal with multiple modules to represent e.g. the task of constructing a logistics service provider relying on local service providers. In the current paper, we study existing systems that operate in a modular way in order to obtain general principles of solving modular model expansion tasks. We introduce a general algorithm to solve model expansion tasks for modular systems. We demonstrate, through several case studies, 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). We make our framework language-independent through a model-theoretic development.

1 Introduction

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. The next step in the development of the MX-based framework is adding modularity concepts. The following example clarifies our goals.

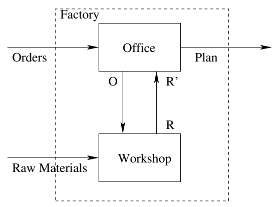

Fig. 1. Modular representation of a factory

Example 1 (Factory). Figure 1 shows a modular representation of a factory. This factory consists of an office and a workshop[1]. The office takes a list of goods needed by consumers and the workshop takes a list of raw materials. These two entities, i.e., the office and the workshop, can communicate with each other in order to plan the production of customers' orders according to both their internal constraints (such as their maximum throughput) and their external constraints (such as the cost of raw materials).

We would like to find a method for finding solutions to modular tasks such as production plan generation in the example. Modularity is incorporated through representing each part in the most suitable language. For example, the office is more easily specified in extended first-order logic, while the complex operation of the workshop module is perhaps most easily specified using ASP (answer set programming) in order to handle exceptions. In this paper, we take initial steps towards solving the underlying computationally complex task.

In a recent work [2], a subset of the authors extended the MX framework to represent a modular system. Under a model-theoretic view, an MX module can be viewed as a set (or class) of structures satisfying some axioms. An abstract algebra on MX modules was developed, 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.

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. 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 [3]. Our main challenge is to come up with an appropriate mathematical abstraction of "combined" solving. In [4], we propose an algorithm, which takes a modular system as input, and generate its solutions. In this paper, we perform several case studies of existing systems in our proposed framework. Our analysis is done from a model-theoretic perspective with the goal of making the approach language-independent.

Our contributions are as follows.

We show that, in the context of the model expansion task, our algorithm generalizes the work of solvers from different communities in a unifying and abstract way. In particular, we show that DPLL(T) framework [5], branch-and-cut based ILP solver [6] and state-of-the-art combination of ASP and CP [7] are all specializations of our algorithm.

[1] A more realistic example contains many more modules.

1. For each problem described in these frameworks, we design a compound modular system such that the set of structures in the modular system correspond to the set of solutions to the original problem.
2. We show how our algorithm can benefit from the techniques used in practical solver constructions to solve the modular system efficiently.
3. We show the feasibility of our algorithm for solving arbitrary modular systems by arguing that our algorithm on the constructed modular system models the solving procedure of the corresponding system.

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 [8].

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 $\mathcal{B}$ of ϕ. We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary[2].

Example 2. The following formula ϕ of first order logic constitutes an MX specification for Graph 3-colouring:

$$\begin{aligned} 1\{R(x), B(x), G(x)\}1 &\leftarrow V(x). \\ \bot &\leftarrow R(x), R(y), E(x, y). \\ \bot &\leftarrow B(x), B(y), E(x, y). \\ \bot &\leftarrow G(x), G(y), E(x, y). \end{aligned}$$

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

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 $\mathcal{G}$.

Example 3 (Factory as Model Expansion). In Figure 1, both the office box and the workshop box can be viewed as model expansion tasks. For example, the box labeled with "Workshop" can be abstractly viewed as an MX task with instance vocabulary $\sigma = \{RawMaterials\}$ and expansion vocabulary $\varepsilon = \{R\}$.

Moreover, in Figure 1, the bigger box with dashed borders can also be viewed as an MX task with instance vocabulary $\sigma' = \{Orders, RawMaterials\}$ and expansion vocabulary $\varepsilon' = \{Plan\}$. This task is a compound MX task whose result depends on the internal work of the office and the workshop.

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 given 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 [2] based on the initial development in [9]. As in [2], *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 [2] for details. We illustrate these operations by giving the following algebraic specification for the modular system in Example 1.

$$\text{Factory} := \pi_{\{\text{Goods},\text{RawMaterials},\text{Plan}\}}(\text{Office} \rhd \text{Factory})[R' = R]). \tag{1}$$

Considering Figure 1, symbol "Factory" refers to the whole modular system denoted by the box with dotted borders. The only important vocabulary symbols outside this box are "Goods", "RawMaterials" and "Plan". All other symbols are projected out. There is also a feedback from R to R'. In this paper, we only consider modular systems which do not use the union operator.

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 briefly describe an algorithm for solving modular systems [4]. This algorithm takes a modular system M and a structure $\mathcal{A}$ and finds an expansion $\mathcal{B}$ of $\mathcal{A}$ in M. The algorithm uses an external solver as well as some oracles to "assist" the solver in finding a model (if one exists). The oracles correspond to the primitive modules of a modular system. In this section, we restate some of the properties that the solver and the so-called oracles have to satisfy in order for the algorithm to work correctly and efficiently. The algorithm works by interacting with the solver and the oracles at the same time. It repeatedly queries a solver S while adding "reasons" and "advices" from oracles to guide the solver. It does so 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. 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 $R^+ \cup R^- \neq (dom(\mathcal{B}))^n$.*

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 $\mathcal{B}$.*

Example 4. Consider a structure $\mathcal{B}$ with domain $\{0, 1, 2\}$ for vocabulary $\{I, R\}$, where I and R are unary relations, and $I^{\mathcal{B}} = \{\langle 0 \rangle, \langle 1 \rangle\}$, $\langle 0 \rangle \in R^{\mathcal{B}}$, and $\langle 1 \rangle \notin R^{\mathcal{B}}$, but it is unknown whether $\langle 2 \rangle \in R^{\mathcal{B}}$ or $\langle 2 \rangle \notin R^{\mathcal{B}}$. Then $\mathcal{B}$ is a $\{R\}$-partial structure over vocabulary $\{I, R\}$ where ${R^+}^{\mathcal{B}} = \{\langle 0 \rangle\}$ and ${R^-}^{\mathcal{B}} = \{\langle 1 \rangle\}$.

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

As expressed in the introduction, there is practical urge to solve complex computational tasks in a modular way so that full access to a complete axiomatization of the module is not assumed, i.e., the module is treated as a black box and accessed via controlled methods. However, clearly, as the solver does not have any information about the internals of the modules, it needs to be assisted by the modules themselves. 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 a 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.

The algorithm that we describe in this section (Algorithm 1), uses CCAV oracles which stand for: (1) complete and constructive, (2) advising and (3) verifying. In this paper, instead of formally defining these concepts, we give examples of how these concepts can be realized in practice. The formal definitions of these concepts can be found in [4]. Before giving these examples, we should point out one difference to the readers who are not accustomed to the logical approach to complexity: In theoretical computer science, a problem is a subset of $\{0, 1\}^*$. However, in descriptive complexity (the logical approach to complexity), the equivalent definition of a problem being a set of structures is adopted.

Example 5 (Certificates: Graph 3-coloring). Consider Example 2 of graph 3-coloring. There, $\sigma = \{E\}$ and $\varepsilon = \{R, G, B\}$. The problem P is the set of graphs $\mathcal{G} = (V^{\mathcal{G}}; E^{\mathcal{G}})$ which are 3-colorable. A certificate set C for problem P of graph 3-coloring is, as one might expect, the same as 3-coloring certificates in complexity theory, i.e., a partitioning of vertices into three sets R, G and B such that each partition is an independent set (i.e., there is no edge connecting two vertex of the same partition together). The certificate set C, as expected, should be so that $\mathcal{A} \in P$ (i.e., $\mathcal{A}$ is 3-colorable) iff C has at least one 3-coloring for $\mathcal{A}$ (i.e., there is at least one expansion $\mathcal{B}$ of $\mathcal{A}$ in C which interprets R, G and B correctly).

In general, certificate sets are similar to certificates in complexity theory. While it might be hard to recognize if a structure $\mathcal{A}$ belongs to a set of structures (i.e., a problem such as graph 3-coloring); it is generally easier to recognize if an expansion $\mathcal{B}$ of $\mathcal{A}$ is a good certificate for $\mathcal{A}$. That is, axiomatizing a certificate set is generally easier than axiomatizing the problem itself. The concept of a certificate set is used to define a verifying oracle:

Example 6 (Verifying Oracles: Graph 3-coloring). Consider a primitive module M which solves the 3-coloring problem P as in Example 5. Also, let C be a certificate set for P as in Example 5. Let O be an oracle such that, given a partial/total coloring $\mathcal{B}$ of a graph $\mathcal{G}$, (1) if $\mathcal{B}$ is total, O accepts $\mathcal{B}$ iff $\mathcal{B} \in C$, and (2) if $\mathcal{B}$ is partial and there is $\mathcal{B}' \in C$ extending $\mathcal{B}$, then O accepts $\mathcal{B}$, and (3) otherwise (i.e., if $\mathcal{B}$ is partial and there is no $\mathcal{B}' \in C$ extending $\mathcal{B}$), O can either reject or accept $\mathcal{B}$. We call this procedure the Valid Acceptance Procedure, and the oracle a verying oracle.

Generally speaking, a verifying oracle is an oracle which never rejects a good partial structure. The intuition is that if an oracle rejects a partial structure, the algorithm can assume that it is futile to continue extending this partial interpretation. Not surprisingly, the concept of a verifying oracle is similar to the familiar concept of a verification procedure in complexity theory. As above, a verifying oracle is precise only about total interpretations. However, in order to be efficient, we need the oracle to tell us something about partial interpretations as well:

Example 7 (Reasons and Advices). Consider primitive module M and verifying oracle O as in Example 6. Also, consider a graph $\mathcal{G} = (V^{\mathcal{G}}; E^{\mathcal{G}})$ with $V^{\mathcal{G}} = \{a, b, c, d\}$ and $E^{\mathcal{G}} = \{(a, b), (b, a), (a, c), (c, a), (a, d), (d, a), (c, d), (d, c)\}$ and a partial 3-coloring $\mathcal{B} = (V^{\mathcal{G}}; E^{\mathcal{G}}, R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})$ of $\mathcal{G}$ which assigns color red to vertices a and b, color green to vertex c and no color (yet) to vertex d. We describe three different scenarios that O can follow with respect to $\mathcal{B}$ and the consequences of each scenario. Obviously, $\mathcal{B}$ is a bad partial 3-coloring and no matter what color we assign to d, we will obtain an invalid 3-coloring of $\mathcal{G}$. Therefore, one scenario for oracle O is to reject this partial coloring. Here, O can return a reason $\phi := \neg(R(a) \wedge R(b))$ which is falsified by $\mathcal{B}$ ($\mathcal{B} \models \neg\phi$) but satisfied by all valid 3-colorings of $\mathcal{G}$. One can think of ϕ as the reason why $\mathcal{B}$ is not a good partial structure. In this scenario, the algorithm will immediately understand that extending $\mathcal{B}$ is futile because it can never satisfy ϕ.

However, O is not required to recognize $\mathcal{B}$ as a bad partial structure right away. Therefore, another scenario for O is to accept $\mathcal{B}$ but still help the solver by giving the advice $\psi := (R(a) \wedge G(c)) \supset B(d)$. Formula ψ is such that it is satisfied by all valid 3-colorings of $\mathcal{G}$ but $\mathcal{B}$ neither satisfies nor falsifies ψ. Therefore, the algorithm can infer that instead of checking all 3 different color assignments to vertex d, it only needs to check one case when d is colored blue. The worst scenario, however, is when O accepts $\mathcal{B}$ and does not give any advice. In this case, the algorithm has to check all colors for d before finding that $\mathcal{B}$ is a bad partial structure.

Oracle O from Example 7 differs from the usual oracles in the sense that it does not only give yes/no answers, but also provides a reason for its "no" answers. Oracles such as O are called *complete and constructive*. Also, O is *advising* because it guide the search by revealing some facts about valid 3-colorings of $\mathcal{G}$. In what follows, CCAV stands for *complete, constructive, advising and verifying*.

3.3 Requirements on the Solver

In this section, we discuss properties that a solver has 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 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. Moreover, the solver has 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). Such a solver is called a *Complete Online Solver*. A formal definition can be found in [4].

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 solve a model expansion task in an iterative fashion. The algorithm works by accumulating reasons and advices from oracles and gradually converging to a solution to the problem.

4 Case Studies: 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. We show that our algorithm acts similar to the state-of-the-art algorithms when the right components are provided.

Notation 1 *We sometimes use a τ-structure $\mathcal{B}$ (which gives an interpretation to vocabulary τ) as the set of atoms of τ which are assigned by $\mathcal{B}$ to be true. For example, when*

```
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
    Let $\tau \subseteq vocab(M)$ be such that $\tau$ does not appear on the right hand of any feedback;
    Let $\rho : vocab(M) \to \tau$ be s.t. for all $\mathcal{B} \in M$ and $E \in vocab(M)$: $E^{\mathcal{B}} = [\rho(E)]^{\mathcal{B}}$;
    /* such $\rho$ can be found using feedback information      */
    Initialize the solver $S$ using the empty expansion of $\mathcal{A}$ to $\tau$;
    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
            Let $\mathcal{B}'$ be a structure such that $E^{\mathcal{B}'} = [\rho(E)]^{\mathcal{B}}$;
            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 $\mathcal{B}'|_{vocab(M_i)}$ ;
                Add the reason given by oracle $O_i$ to $S$ ;
            else if $\mathcal{B}'$ is total then return $\mathcal{B}'$ ;
end
```
Algorithm 1: Lazy Model Expansion Algorithm

$\tau = \{R, S\}$ and $R^{\mathcal{B}} = \{(1,2)\}$ and $S^{\mathcal{B}} = \{(1,1),(2,2)\}$, then we may use $\mathcal{B}$ to represent the following set of atoms:

$$\mathcal{B} = \{R(1,2), S(1,1), S(2,2)\}.$$

We may also use a partial interpretation as a set of true atoms in a similar fashion. Sometimes, we also use $\mathcal{B}$ to represent a formula, i.e., conjunction of the atoms in above set. Complement of a set is defined as usual, e.g., $R^{\mathcal{B}^c} = dom(\mathcal{B})^2 \setminus R^{\mathcal{B}}$. Negation of a set S of literals is also defined such that $l \in S$ if and only if $\neg l \in \neg S$.

4.1 Modelling DPLL(T)

DPLL(T) [5] 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 [5].

The modular system representing the DPLL(T) system on the input formula $\phi \wedge \psi$ is shown in figure 2, where $\sigma = I$, $\varepsilon = E$, and $E^+ \cup E^- \cup E_1^+ \cup E_1^- \cup E_2^+ \cup E_2^-$

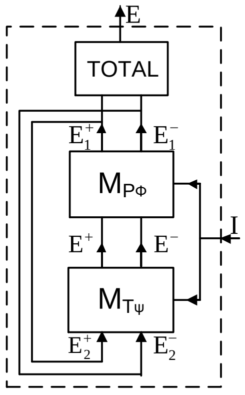

Fig. 2. Modular System Representing the DPLL(T) System on Input Formula $\phi \wedge \psi$

is the internal vocabulary of the module. Also, there are feedbacks from E_1^+ to E_2^+ and from E_1^- to E_2^-. The set of symbols in E^+ and E^- (same for E_1^+ and E_1^-, E_2^+ and E_2^-) semantically represents a partial interpretation of the symbols in the expansion vocabulary, i.e., E^+ (resp. E^-) represents the positive (resp. negative) part of the partial interpretation.

There are three MX modules in $DPLL(T)_{\phi\wedge\psi}$. The modules M_{P_ϕ} and M_{T_ψ} work on different parts of the specification. The formula ϕ in M_{P_ϕ} is CNF representation of the problem specification with all non-propositional literals replaced by propositional ones, and the formula ψ in M_{T_ψ} is the formula $\bigwedge_i d_i \Leftrightarrow l_i$ where l_i and d_i are, respectively, an atomic formula in theory T and its associated propositional literal used in M_{P_ϕ}. The module M_{P_ϕ} is the set of structures $\mathcal{B}$ such that:

$$(E_1^{+\mathcal{B}}, E_1^{-\mathcal{B}}) = \begin{cases} (D, D) & \text{if } R^+ \cap R^- \neq \emptyset \\ (R^+, R^{+c}) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \models \phi \\ (R^+, R^-) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \not\models \phi \end{cases},$$

where $D = [dom(\mathcal{B})]^n$, n is the arity of E^+, and (R^+, R^-) is the result of Unit Propagation on ϕ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup E^{+\mathcal{B}} \cup \neg E^{-\mathcal{B}}$.

Similarly, the module M_{T_ψ} is defined as the set of structures $\mathcal{B}$ such that:

$$(E^{+\mathcal{B}}, E^{-\mathcal{B}}) = \begin{cases} (D, D) & \text{if } R^+ \cap R^- \neq \emptyset \\ (D, D) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \models_T \neg\psi \\ (R^+, R^{+c}) & \text{if } R^+ \cap R^- = \emptyset, I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^- \models_T \psi \\ (R^+, R^-) & \text{if } R^+ \cap R^- = \emptyset, \text{T-satisfiability unknown} \end{cases},$$

where D is as before and (R^+, R^-) is the result of Theory Propagation on ψ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup E_2^{+\mathcal{B}} \cup \neg E_2^{-\mathcal{B}}$, and $R \models_T \psi$ denotes that ψ is T-satisfiable under the set of facts R. Note that the satisfiability test is not necessarily complete. It can be done in different degrees depending on the complexity of different theories.

The module $TOTAL$ is the set of structures $\mathcal{B}$ such that $E_1^{+\mathcal{B}} \cap E_1^{-\mathcal{B}} = \emptyset$, $E_1^{+\mathcal{B}} \cup E_1^{-\mathcal{B}} = D$, and $E_1^{+\mathcal{B}} = E^{\mathcal{B}}$.

We define the modular system $DPLL(T)_{\phi\wedge\psi}$ as:

$$DPLL(T)_{\phi\wedge\psi} := \pi_{\{I,E\}}(((M_{T_\psi} \rhd M_{P_\phi})[E_1^+ = E_2^+][E_1^- = E_2^-]) \rhd TOTAL). \quad (2)$$

To show that the combined module $DPLL(T)_{\phi\wedge\psi}$ is correct, consider any model of the modular system. Note that for both modules M_{P_ϕ} and M_{T_ψ}, the outputs always contain all the information that the inputs have, i.e., for any structure $\mathcal{B}$ in the module M_{P_ϕ}, we have ${E_1^+}^{\mathcal{B}} \supseteq {E^+}^{\mathcal{B}}$ and ${E_1^-}^{\mathcal{B}} \supseteq {E^-}^{\mathcal{B}}$, and for any structure $\mathcal{B}$ in M_{T_ψ}, we have ${E^+}^{\mathcal{B}} \supseteq {E_2^+}^{\mathcal{B}}$ and ${E^-}^{\mathcal{B}} \supseteq {E_2^-}^{\mathcal{B}}$. Furthermore, from the semantics of the feedback operator, we know that ${E_1^+}^{\mathcal{B}} = {E_2^+}^{\mathcal{B}}$ and ${E_1^-}^{\mathcal{B}} = {E_2^-}^{\mathcal{B}}$. Thus, we have ${E^+}^{\mathcal{B}} = {E_1^+}^{\mathcal{B}} = {E_2^+}^{\mathcal{B}}$ and ${E^-}^{\mathcal{B}} = {E_1^-}^{\mathcal{B}} = {E_2^-}^{\mathcal{B}}$. Moreover, from the definition of module $TOTAL$, we know that (${E_1^+}^{\mathcal{B}}$, ${E_1^-}^{\mathcal{B}}$) represents a total interpretation of the symbols in E and $E^{\mathcal{B}} = {E_1^+}^{\mathcal{B}}$. Finally, from the definitions of M_{P_ϕ} and M_{T_ψ} on encodings of total interpretations, we can conclude that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$. On the other hand, it is easy to see that for any structure $\mathcal{B}$ such that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$, $\mathcal{B}$ is in $DPLL(T)_{\phi\wedge\psi}$.

So, there is a one-to-one correspondence between models of $DPLL(T)_{\phi\wedge\psi}$ and the propositional part of the solutions to the DPLL(T) system on input formula $\phi \wedge \psi$. To find a solution, one can compute a model of this modular system.

To solve $DPLL(T)_{\phi\wedge\psi}$, 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 to the solver. The three modules $TOTAL$, M_{T_ψ} and M_{P_ϕ} are attached with oracles O_{TOTAL}, O_T and O_P respectively. They accept a partial structure $\mathcal{B}$ iff their respective module constraints are not falsified by $\mathcal{B}$. As the constructions of modules O_T and O_P are similar to each other, we only give constructions for the solver S, oracle O_{TOTAL}, and oracle O_T:

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

Oracle O_{TOTAL} accepts a partial structure $\mathcal{B}$ iff ${E_1^+}^{\mathcal{B}} \cap {E_1^-}^{\mathcal{B}} = \emptyset$, ${E_1^+}^{\mathcal{B}} \cup {E_1^-}^{\mathcal{B}} = D$, and $E^{\mathcal{B}} = {E^+}^{\mathcal{B}}$. If $\mathcal{B}$ is rejected, O_{TOTAL} returns $\bigwedge_{\omega\in\Omega'} \omega$ as the reason, where Ω' is any non-empty subset of the set $\Omega = \{E_1^+(d) \Leftrightarrow \neg E_1^-(d) \mid d \in D, \mathcal{B} \not\models E_1^+(d) \Leftrightarrow \neg E_1^-(d)\} \cup \{E(d) \Leftrightarrow E_1^+(d) \mid d \in D, \mathcal{B} \not\models E(d) \Leftrightarrow E_1^+(d)\}$. O_{TOTAL} returns the set Ω as the set of advices when $\mathcal{B}$ is the empty expansion of the instance structure, and the empty set otherwise.[3] Clearly, O_{TOTAL} is a CCAV oracle.

Oracle O_T accepts a partial structure $\mathcal{B}$ iff it does not falsify the constraints described above for module M_{T_ψ} on I, E^+, E^-, E_2^+, and E_2^-. Let (R^+, R^-) denote the result of the Theory Propagation on ψ under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup {E_2^+}^{\mathcal{B}} \cup \neg {E_2^-}^{\mathcal{B}}$. Then, if $\mathcal{B}$ is rejected,

1. If $R^+ \cap R^- \neq \emptyset$ or ψ is T-unsatisfiable under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^-$, O_T returns a reason ω of the form $\bigwedge_{d\in D_1} E_2^+(d) \wedge \bigwedge_{d\in D_2} E_2^-(d) \supset \bigwedge_{d\in D_3}(E^+(d) \wedge E^-(d))$ with $D_1 \subseteq D, D_2 \subseteq D, \emptyset \subsetneq D_3 \subseteq D, T \models \bigvee_{d\in D_1} \neg l(d) \vee \bigvee_{d\in D_2} l(d), \mathcal{B} \models \neg\omega$, where $l(d)$ denotes the atomic formula l in ψ whose associated propositional atom is d. Note that from the advices and reasons from oracles, the solver can understand

[3] This makes sure that Ω is returned only once at the beginning.

that right hand side of the implication is inconsistent, and thus the reason corresponds to the set of T-inconsistent literals from the theory solver in the DPLL(T) system.
2. Else if ψ is T-satisfiable under $I^{\mathcal{B}} \cup \neg I^{\mathcal{B}^c} \cup R^+ \cup \neg R^-$, O_T returns a reason ω of the form $\bigwedge_{d \in D_1} E_2^+(d) \wedge \bigwedge_{d \in D_2} E_2^-(d) \supset \bigwedge_{d \in R^+} E^+(d) \wedge \bigwedge_{d \in R^{+c}} E^-(d)$, where $D_1 \subseteq D, D_2 \subseteq D, \mathcal{B} \models \neg\omega$.
3. Else, O_T returns a reason similar to the second case except that it uses R^- instead of ${R^+}^c$.

By the definition of M_{T_ψ}, we know that $\mathcal{B}$ falsifies the reason and all models of M_{T_ψ} satisfy the reason. Thus, O_T is complete and constructive. O_T may also return some advices in the same form as any ω above such that $\mathcal{B}$ satisfies the left hand side of the implication, but not the right hand side. Also, since the outputs of M_{T_ψ} always subsume the inputs, O_T may also return the set $\{E_2^+(d) \supset E^+(d) \mid d \in D, \mathcal{B} \models E_2^+(d), \mathcal{B} \not\models E^+(d)\} \cup \{E_2^-(d) \supset E^-(d) \mid d \in D, \mathcal{B} \models E_2^-(d), \mathcal{B} \not\models E^-(d)\}$ as the set of advices.[4] Clearly, all the structures in M_{T_ψ} satisfy all sets of advices. Hence, O_T is an advising oracle. Finally, 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_ψ}. O_T never rejects any good partial structure $\mathcal{B}$ (although it may accept some bad non-total structures). Therefore, O_T is a verifying oracle.

Proposition 1. *1. Modular system $DPLL(T)_{\phi \wedge \psi}$ is the set of structures $\mathcal{B}$ such that $\mathcal{B} \models \phi$ and $\mathcal{B} \models_T \psi$.*
2. Solver S is complete and online.
3. O_P, O_T, and O_{TOTAL} are CCAV oracles.
4. Algorithm 1 on modular system $DPLL(T)_{\phi \wedge \psi}$ associated with oracles O_P, O_T, O_{TOTAL}, and the solver S models the solving procedure of the DPLL(T) system on input formula $\phi \wedge \psi$.

DPLL(T) architecture is known to be very efficient and many solvers are designed to use it, including most SMT solvers [10]. The DPLL(Agg) module [11] 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 [6] is as follows:
1. Initialization: $S = \{\text{ILP}^0\}$ with ILP^0 the initial problem.
2. Termination: If $S = \emptyset$, return UNSAT.

[4] Again O_T only returns this set when $\mathcal{B}$ is the empty expansion of the instance structure.

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 $\{C^i \cup \{x_i \leq \lfloor v_i \rfloor\}, C^i \cup \{x_i \geq \lceil v_i \rceil\}\}$.

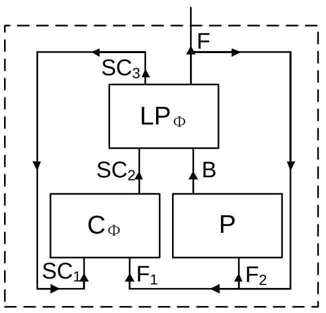

Fig. 3. Modular System Representing an ILP Solver

We use the modular system shown in figure 3 to represent the ILP solver. The module C_ϕ takes a set of variable assignments F_1 and a set of cutting planes SC_1 as inputs and returns another set of cutting planes SC_2. When all the assignments in F_1 are integral, SC_2 is equal to SC_1, and if not, SC_2 is the union of SC_1 and a cutting plane violated by F_1 w.r.t. the set of linear constraints $SC_1 \cup \phi$. The module P takes a set of assignments F_2 as input and outputs a set of range constraints $B = \{B_x \mid F_2(x) \notin \mathcal{Z}\}$, where B_x is non-deterministically chosen from the set $\{x \leq \lfloor F_2(x) \rfloor,\ x \geq \lceil F_2(x) \rceil\}$. The module LP_ϕ takes the set of cutting planes SC_2 and the set of range constraints B as inputs and outputs the set of cuttings planes SC_3 and the set of assignments F in a deterministic way such that SC_3 is the union of SC_2 and B, and F is a total assignment satisfying $SC_2 \cup B \cup \phi$. LP_ϕ is undefined when $SC_2 \cup B \cup \phi$ is inconsistent. We define the compound module ILP_ϕ to be:

$$ILP_\phi := \pi_{\{F\}}(((C_\phi \cap P) \rhd LP_\phi)[SC_3 = SC_1][F = F_1][F = F_2]).$$

To show that the combined module ILP_ϕ is correct, consider any model of the modular system. By the definition of LP_ϕ, we know that F satisfies ϕ. Furthermore, the set B is empty in the model because F satisfies all the linear constraints in B, but F_2 (which is equal to F by the semantics of feedback operator) falsifies those constraints. Thus by the definition of the module P, we know that F_2 (also F) is integral. Thus F is an integral solution to ϕ. On the other hand, for any integral solution S to ϕ, consider a structure $\mathcal{B}$ such that $F^{\mathcal{B}} = F_1^{\mathcal{B}} = F_2^{\mathcal{B}} = S$, $B^{\mathcal{B}} = \emptyset$, and $SC_1^{\mathcal{B}} = SC_2^{\mathcal{B}} = SC_3^{\mathcal{B}} =$

$\bigcup_x \{x \leq F(x),\ x \geq F(x)\}$. Then clearly, $\mathcal{B}$ is in the module ILP_ϕ, i.e., $\mathcal{B}$ is the model of the module ILP_ϕ.

So there is one-to-one correspondence between the solutions of the ILP problem with input ϕ, and the models of the modular system ILP_ϕ. We compute a model of this modular system by associating modules with oracles (O_c, O_p and O_{lp}) and introducing a solver S that interacts with those oracles. Each oracle rejects a partial structure $\mathcal{B}$ if it contradicts corresponding module definition and in this case, the reason for the rejection is provided. For example, when $F_2^{\mathcal{B}}$ is non-integral, O_p rejects $\mathcal{B}$ and gives the reason $\lfloor F_2(x)^{\mathcal{B}} \rfloor < F_2(x) < \lceil F_2(x)^{\mathcal{B}} \rceil \supset B(\text{“}F_2(x) \leq \lfloor F_2(x)^{\mathcal{B}} \rfloor\text{”}) \vee B(\text{“}F_2(x) \geq \lceil F_2(x)^{\mathcal{B}} \rceil\text{”})$, for some non-integral variable x, and O_c rejects $\mathcal{B}$ with the reason $(\bigwedge_{l \in L} SC_1(l)) \wedge (\bigwedge_x F_1(x) = F_1(x)^{\mathcal{B}}) \supset SC_2(c)$, where c is the cutting plane that violates F_1, and L is a subset of SC_1 such that F_1 is the intersection of some constraints in $L \cup \phi$. Full details of the oracles are omitted due to the space consideration. The solver S accepts full propositional language with atomic formulas being either boolean variables or range constraints. In addition, S can assign numerical values (for F) according to the set of derived range constraints.

Proposition 2. *1. Modular system ILP_ϕ is the set of structures representing the sets of integral solutions of ϕ.*
2. S is complete and online.
3. O_c, O_p and O_{lp} are CCAV oracles.
4. Algorithm 1 on modular system ILP_ϕ, associated with oracles O_c, O_p, O_{lp}, and the solver S models the branch-and-cut-based ILP solver on input formula ϕ.

There are many other solvers in the 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 needed). However, embedding Constraint Programming (CP) techniques into ASP solving is proved useful because complete grounding can be avoided.

In [12], 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 [13] 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 [7] developed an improved hybrid solver which supports advanced backjumping and conflict-driven nogood learning (CDNL) techniques. They show that their solver's performance is comparable to state-of-the-art SMT solvers. Paper [7] 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_ϕ instead of SAT_ϕ and CP_ψ instead of ILP_ψ. The compound module $CASP_{\phi\wedge\psi}$ is defined as:

$$CASP_{\phi\wedge\psi} := \pi_{\{I,E\}}(((CP_\psi \rhd ASP_\phi)[E_1^+ = E_2^+][E_1^- = E_2^-]) \rhd TOTAL).$$

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_ϕ and CP_ψ 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 [7] by wrapping CP solvers with a conflict analysis mechanism to compute nogoods based on the first UIP scheme.

5 Related Work

There are many papers on modularity in declarative programming, we only review the most relevant ones. The authors of [9] proposed a multi-language framework for constraint modelling. That work was the initial inspiration of [2], but the authors extended the ideas significantly by developing a model-theoretic framework and introducing a feedback operator that adds a significant expressive power.

An early work on adding modularity to logic programs is [14]. The authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. The ideas are further generalized in [15] 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 [16] to define modularity for disjunctive logic programs. There are also other approaches to adding modularity to ASP languages and ID-Logic as described in [17–19].

The works mentioned earlier focus on the theory of modularity in declarative languages. However, there are also papers that focus on the practice of modular declarative programming and, in particular, solving. These 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 [20] and ASP-PROLOG [21] extend Prolog with ASP, CP techniques are incorporated into ASP solving in [12], [13], and [7]. Also, ESRA [22], ESSENCE [8] and Zinc [23] 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. The second category is related to multi-context systems. In [24], 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 [25–28]. However, these papers 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 is an important future research direction.

6 Conclusion

We took a language-independent view on iterative modular problem solving. Our algorithm is designed to solve combinatorial search problems described as modular systems in the context of model expansion. This model-theoretic approach allows us to abstract away from particular languages of the modules. We performed several case studies of our algorithm in relation to existing systems such as DPLL(T), ILP, ASP-CP. We demonstrated that, in the context of the model expansion task, our algorithm generalizes the work of these solvers. As a side effect of this analysis, we demonstrated how Valid Acceptance Procedures from different communities could be used to implement oracles for modules to achieve efficient solving. For example, the procedures of Well-Founded Model computation and Arc-Consistency checking can be used to implement oracles for the ASP and CP languages to construct an efficient combined solver, which corresponds to the state-of-the-art combination of ASP and CP described in [7].

Our general approach for solving modular systems can be applied to systems such as Business Process Planners in different areas and their variants including Logistics Service Provider, Manufacturer Supply Chain Management, Mid-size Businesses Relying on External Web Services and Cloud Computing. With the increasing use of service-oriented architecture, such modular systems will become increasingly more applicable. We believe we are taking important initial steps addressing the core aspect of this complex multi-dimensional problem, namely the underlying computationally complex task. As a future direction, we plan to develop a prototype implementation of our algorithms.

References

1. Mitchell, D.G., Ternovska, E.: A framework for representing and solving NP search problems. In: Proc. AAAI. (2005) 430–435

2. 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)
3. Niemelä, I.: Integrating answer set programming and satisfiability modulo theories. In: LPNMR. Volume 5753 of LNCS., Springer-Verlag (2009) 3–3
4. Tasharrofi, S., Wu, X.N., Ternovska, E.: Language-independent modular problem solving. (2012) Under Review.
5. 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
6. Pardalos, P., Resende, M.: Handbook of applied optimization. Volume 126. Oxford University Press New York; (2002)
7. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proc. of ICLP'09. LNCS, Springer-Verlag (2009) 235–249
8. 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
9. 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
10. Sebastiani, R.: Lazy Satisfiability Modulo Theories. JSAT **3** (2007) 141–224
11. De Cat, B., Denecker, M.: DPLL(Agg): An efficient SMT module for aggregates. In: LaSh'10 Workshop. (2010)
12. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. (2005) 52–66
13. 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
14. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Proc. of LPNMR, Springer-Verlag (1997) 290–309
15. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: the Proc. of NMR'06. (2006) 10–18
16. 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
17. 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
18. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: SEA. (2007) 41–55
19. Denecker, M., Ternovska, E.: A logic of non-monotone inductive definitions. Transactions on Computational Logic **9**(2) (2008) 1–51
20. Swift, T., Warren, D.S.: The XSB System. (2009)
21. 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
22. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. Logic Based Program Synthesis and Transformation (2004) 214–232
23. de la Banda, M., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. Principles and Practice of Constraint Programming-CP 2006 700–705
24. 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

25. 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
26. 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
27. 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)
28. 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