Abstract Modular Inference Systems and Solvers

Yuliya Lierler[1] and Miroslaw Truszczynski[2]

[1] University of Nebraska at Omaha
ylierler@unomaha.edu
[2] University of Kentucky
mirek@cs.uky.edu

Abstract. Integrating diverse formalisms into modular knowledge representation systems offers increased expressivity, modeling convenience and computational benefits. We introduce the concepts of *abstract inference modules* and *abstract modular inference systems* to study general principles behind the design and analysis of model-generating programs, or *solvers*, for integrated multi-logic systems. We show how modules and modular systems give rise to *transition graphs*, which are a natural and convenient representation of solvers, an idea pioneered by the SAT community. We illustrate our approach by showing how it applies to answer-set programming and propositional logic, and to multi-logic systems based on these two formalisms.

1 Introduction

Knowledge representation and reasoning (KR) is concerned with developing formal languages and logics to model knowledge, and with designing and implementing corresponding automated reasoning tools. The choice of specific logics and tools depends on the type of knowledge to be represented and reasoned about. Different logics are suitable for common-sense reasoning, reasoning under incomplete information and uncertainty, for temporal and spatial reasoning, and for modeling and solving boolean constraints, or constraints over larger, even continuous domains. In applications in areas such as distributed databases, semantic web, hybrid constraint modeling and solving, to name just a few, several of these aspects come to play. Accordingly, often diverse logics have to be accommodated together.

Modeling convenience is not the only reason why diverse logics are combined into modular hybrid KR systems. Another motivation is to exploit in reasoning the transparent structure that comes from modularity, computational strengths of individual logics, and synergies that arise when they are put together. Constraint logic programming [10] and satisfiability modulo theories (SMT) [20,2] are well-known examples of formalisms stemming directly from such considerations. More recent examples include constraint answer-set programming (CASP) [13] that integrates answer-set programming (ASP) [16,18] with constraint modeling languages [22], and "multi-logic" formalisms PC(ID) [17], SM(ASP) [14] and ASP-FO [4] that combine modules expressed as logic theories under the classical semantics with modules given as answer-set programs.

The key computational task arising in KR is that of *model generation*. Model-generating programs or *solvers*, developed in satisfiability (SAT) and ASP proved to be

M. Flatt and H.-F. Guo (Eds.): PADL 2014, LNCS 8324, pp. 49–64, 2014.

effective in a broad range of KR applications. Accordingly, model generation is of critical importance in modular multi-logic systems. Research on formalisms listed above resulted in fast solvers that demonstrate gains one can obtain from their heterogeneous nature. However, the diversity of logics considered and low-level technical details of their syntax and semantics obscure general principles that are important in the design and analysis of solvers for multi-logic systems.

In this paper we address this problem by proposing a language for talking about modular multi-logic systems that (i) abstracts away the syntactic details, (ii) is expressive enough to capture various concepts of inference, and (iii) is based only on the weakest assumptions concerning the semantics of underlying logics. The basic elements of this language are *abstract inference modules* (or just *modules*). Collections of abstract inference modules constitute *abstract modular inference systems* (or just *modular systems*). We define the semantics of abstract inference modules and show that they provide a uniform language capturing different logics, diverse inference mechanisms, and their modular combinations. Importantly, abstract inference modules and abstract modular inference systems give rise to *transition graphs* of the type introduced by Nieuwenhuis, Oliveras, and Tinelli [20] in their study of SAT and SMT solvers. As in that earlier work, our transition graphs provide a natural and convenient representation of solvers for modules and modular systems. In this way, abstract modular inference systems and the corresponding framework of transition graphs are useful conceptualizations clarifying computational principles behind solvers for multi-logic knowledge representation systems and facilitating systematic development of new ones.

We start the paper by introducing abstract inference modules.We then adapt transition graphs of Nieuwenhuis et al. [20] to the formalism of abstract inference modules and use them to describe algorithms for finding models of modules. In Section 4, we introduce abstract modular inference systems, extend the concept of a transition graph to modular systems, and show that transition graphs can be used to formalize search for models in this setting, too. We conclude by discussing related work, recapping our contributions, and commenting on future work. Throughout the paper, we illustrate our approach by showing how it applies to propositional logic and answer-set programming, and to multi-logic systems based on these two formalisms. A version of the paper containing proofs is available at `http://www.cs.uky.edu/ai/ams.pdf`.

2 Abstract Inference Modules

We start with some notation. Let σ be a fixed infinite vocabulary (a set of propositional atoms). We write $Lit(\sigma)$ for the set of all literals over σ. For a set $M \subseteq Lit(\sigma)$, we define $M^+ = \sigma \cap M$ and $M^- = \{a \in \sigma \colon \neg a \in M\}$. A literal $l \in Lit(\sigma)$ is *unassigned* by a set of literals $M \subseteq Lit(\sigma)$ if M contains neither l nor its dual literal $\overline{l}$. A set M of literals over σ is *consistent* if for every literal $l \in Lit(\sigma)$, $l \notin M$ or $\overline{l} \notin M$. We denote the set of all consistent subsets of $Lit(\sigma)$ by $C(\sigma)$.

Definition 1. *An* abstract inference module *over a vocabulary σ (or just a* module*, for short) is a finite set of pairs of the form (M, l), where $M \in C(\sigma)$, $l \in Lit(\sigma)$ and $l \notin M$. These pairs are called* inferences *of the module. For a module S, $\sigma(S)$ denotes the set of all atoms that appear in inferences of S.*

Intuitively, an inference (M, l) in a module indicates support for inferring l whenever all literals in M are given. We note that if (M, l) is an inference and $\overline{l} \in M$, the inference is an explicit indication of a contradiction. Figure 1(a) shows all inferences over the vocabulary $\{a\}$. Figures 1(b) and 1(c) give examples of modules over the vocabulary $\{a\}$. Here and throughout the paper, we present inferences as directed edges and modules as bipartite graphs.

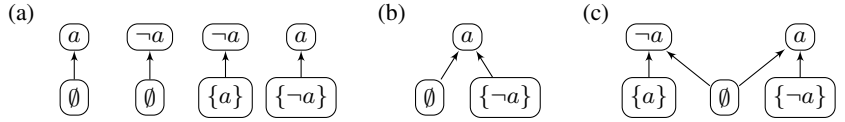

Fig. 1. All inferences and two inference modules over the vocabulary $\{a\}$

A set $M \subseteq Lit(\sigma)$ is *consistent with* a set $X \subseteq \sigma$ if $M^+ \subseteq X$ and $M^- \cap X = \emptyset$. A literal $l \in Lit(\sigma)$ is *consistent with* a set $X \subseteq \sigma$ if $\{l\}$ is consistent with X. Let S be an abstract inference module. A set $X \subseteq \sigma$ of atoms is a *model* of S if for every inference $(M, l) \in S$ such that M is consistent with X, l is consistent with X, too. For example, any set of atoms that contains a is a model of the module in Figure 1(b), whereas no set of atoms that does not contain a is such. The module in Figure 1(c) has no models due to inferences $(\emptyset, a)$ and $(\emptyset, \neg a)$. A module is *satisfiable* if it has models, and is *unsatisfiable* otherwise. The module in Figure 1(b) is satisfiable, the one in Figure 1(c) is unsatisfiable.

Two modules that have the same models are *equivalent*.

Proposition 1. *Abstract inference modules S_1 and S_2 are equivalent if and only if they have the same models contained in the set $\sigma(S_1) \cup \sigma(S_2)$.*

The semantics of modules is given by the set of their models. A module S over a vocabulary σ *entails* a literal $l \in Lit(\sigma)$, written $S \models\!\!\sim l$, if for every model X of S, l is consistent with X. Furthermore, S *entails* l with respect to a set $M \subseteq Lit(\sigma)$ of literals, written $S \models\!\!\sim_M l$, if whenever M is consistent with a model X of S, l is consistent with X, too. Modules are *sound* with respect to their semantics:

Proposition 2. *Let S be a module and (M, l) an edge in S. Then $S \models\!\!\sim_M l$.*

In the paper we often consider unions of (finitely many) modules. We use the symbol $\cup$ to denote the union of modules.

Proposition 3. *Let S_1 and S_2 be abstract inference modules. A set X of atoms is a model of $S_1 \cup S_2$ if and only if X is a model of S_1 and S_2.*

Modules are not meant for modeling knowledge. Representations by means of logic theories are usually more concise. Furthermore, the logic languages align closely with natural language, which facilitates modeling and makes the correspondence between logic theories and knowledge they represent direct. Modules lack this connection. The power of modules comes from the fact that they provide a uniform, syntax-independent

way to describe theories and inference methods from *different* logics. We illustrate this property of modules by showing that they can capture theories and inferences in classical propositional logic and in answer-set programming [8,16,18] (where theories are commonly called programs).

Let T be a finite CNF propositional theory over σ and let σ_T be the set of atoms that actually appear in T. We first consider the inference method given by the classical entailment. By $Ent(T)$ we denote the module consisting of pairs (M, l) that satisfy the following conditions: $M \in C(\sigma_T)$, $l \in Lit(\sigma_T) \setminus M$, and $T \cup M \models l$. Figure 1(b) shows the module $Ent(\{a\})$. Similarly, Figure 2 presents the module $Ent(T)$, where T is the theory:

$$\{a \vee b, \neg a \vee \neg b\}. \tag{1}$$

We note that $Ent(T)$ has two models contained in $\{a, b\}$: $\{a\}$ and $\{b\}$.[1] More generally, every model X of $Ent(T)$ contains exactly one of a and b.

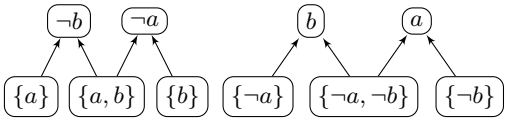

Fig. 2. Abstract module $Ent(T)$ for the theory T given by (1)

Focusing on specific inference rules of propositional logic also gives rise to abstract modules. *Unit Propagate* is a standard inference rule commonly used when reasoning with CNF theories. This inference rule is essential to all satisfiability (SAT) solvers, programs that compute models of CNF theories or determine that no models exist. The *Unit Propagate* rule gives rise to the module $UP(T)$ that consists of all pairs (M, l) that satisfy the following conditions: $M \in C(\sigma_T)$, $l \in Lit(\sigma_T) \setminus M$, and T has a clause $C \vee l$ (modulo reordering of literals) such that for every literal u of C, $\overline{u} \in M$.

Let T be the theory (1). The module $Ent(T)$ in Figure 2 coincides with $UP(T)$. Thus, for the theory (1) the *Unit Propagate* rule captures entailment.

We say that a module S is *equivalent* to a propositional theory T if they have the same models. Clearly, the module in Figure 2 is equivalent to the propositional theory (1). This is an instance of a general property.

Proposition 4. *For every propositional theory T (respectively, CNF formula T containing no empty clause), $Ent(T)$ (respectively, $UP(T)$) is equivalent to T.*

Unit Propagate is the primary inference rule of most SAT solvers. In the case of answer-set programming, most solvers rely on several inference rules associated with reasoning under the answer-set semantics. For instance, the classical answer-set solver SMODELS [19] exploits four inference rules: the *Unit Propagate* rule, the *Unfounded* rule, the *All Rules Cancelled* rule, and the *Backchain True* rule. To state these rules we introduce some definitions and notations commonly used in logic programming.

[1] We identify a model, an interpretation, of a propositional theory with the set of atoms that are assigned $True$ in the model.

A *logic program*, or simply a *program*, over σ is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_\ell, not\ a_{\ell+1}, \ldots, not\ a_m, \tag{2}$$

where each a_i, $0 \leq i \leq m$, is an atom from σ. The expression a_0 is the *head* of the rule. The expression on the right hand side of the arrow is the *body*. For a program Π and an atom a, $Bodies(\Pi, a)$ denotes the set of the bodies of all rules in Π with the head a. We write σ_Π for the set of atoms that occur in a program Π.

For the body B of a rule (2), we define $s(B) = \{a_1, \ldots, a_\ell, \neg a_{\ell+1}, \ldots, \neg a_m\}$. In some cases, we identify B with the conjunction of the elements in $s(B)$, and we often interpret a rule (2) as the propositional clause

$$a_0 \vee \neg a_1 \vee \ldots \vee \neg a_\ell \vee a_{\ell+1} \vee \ldots \vee a_m. \tag{3}$$

For a program Π, we write Π^{cl} for the set of clauses (3) corresponding to all rules in Π. We assume the reader is familiar with the definition of an *answer set* [8], as well as the concept of *unfounded sets* [25]. For a set M of literals and a program Π, we write $U(M, \Pi)$ to denote a set that is unfounded on M w.r.t. Π (typically, such set will be identified by some algorithmic method, but a specific way in which we find it is immaterial for the purposes of this paper).

We are now ready to define the SMODELS inference rules. For a program Π, a set $M \in C(\sigma_\Pi)$ of literals, and a literal $l \in Lit(\sigma_\Pi) \setminus M$:

Unit Propagate: derive l if Π^{cl} contains clause $C \vee l$ such that for every $u \in C$, $\overline{u} \in M$;

Unfounded: derive l if $l = \neg a$ and $a \in U(M, \Pi)$;

All Rule Cancelled: derive l if $l = \neg a$ and for every $B \in Bodies(\Pi, a)$, there is $u \in s(B)$ such that $\overline{u} \in M$;

Backchain True: derive l, if for some rule $a \leftarrow B \in \Pi$, $a \in M$, $l \in s(B)$, and for every $B' \in Bodies(\Pi, a) \setminus \{B\}$, there is $u \in s(B')$ such that $\overline{u} \in M$;

Note that $UP(\Pi)$ and $UP(\Pi^{cl})$ are identical (and equivalent) even though they concern different logics.

The four rules above give rise to abstract inference modules $UP(\Pi)$, $UF(\Pi)$, $ARC(\Pi)$ and $BC(\Pi)$, respectively, each defined by taking the definition of the rule as the condition for (M, l) to be an inference of the module. We note that the inference rule *All Rule Cancelled* is subsumed by the inference rule *Unfounded*. That is, $ARC(\Pi) \subseteq UF(\Pi)$. This is the only inclusion relation between distinct modules in that set that holds for every program.

We say that a module S is *equivalent* to a program Π if for every $X \subseteq \sigma_\Pi$, X is a model of S if and only if X is an answer set of Π.[2] None of the four modules $UP(\Pi)$, $UF(\Pi)$, $ARC(\Pi)$ and $BC(\Pi)$ alone is equivalent to the underlying program Π. However, some combinations of these modules are. Let us define

$$UPUF(\Pi) = UP(\Pi) \cup UF(\Pi)$$

[2] This is not the standard concept of equivalence as it is restricted to models over the vocabulary of the program. It is sufficient, however, for our purpose of studying algorithms to compute answer sets.

and

$$smodels(\Pi) = UP(\Pi) \cup UF(\Pi) \cup ARC(\Pi) \cup BC(\Pi).$$

Since $ARC(\Pi) \subseteq UF(\Pi)$, it is not necessary to list the module $ARC(\Pi)$ explicitly in the union above. We do so, as the rule *All Rule Cancelled* is computationally cheaper than the rule *Unfounded* and in practical implementations the two are distinguished.

The following result restates well-known properties of the inference rules [12] in terms of equivalence of modules and programs.

Proposition 5. *Every logic program Π is equivalent to the modules $UPUF(\Pi)$ and $smodels(\Pi)$.*

Let Π be the program

$$\begin{array}{l} a \leftarrow not\ b \\ b \leftarrow not\ a. \end{array} \tag{4}$$

This program has two answer sets $\{a\}$ and $\{b\}$. Since these are also the only two models over the vocabulary $\{a, b\}$ of the module in Figure 2, the program and the module are equivalent. The module represents the reasoning mechanism of entailment with respect to the answer sets of the program. Furthermore, that module also represents the program (4) and the reasoning mechanism captured by the module $smodels(\Pi)$.

Two other modules associated with program (4) are given in Figure 3. Figure 3(a) shows the module $UP(\Pi)$, which represents the program (4) and the reasoning mechanism based on *Unit Propagate*. This module is not equivalent to program (4). Indeed, $\{a, b\}$ is its model but not an answer set of (4). Figure 3(b) shows the module $ARC(\Pi)$ (which in this case happens to coincide with both $UF(\Pi)$ and $BC(\Pi)$). Also this module is not equivalent to program (4) as $\emptyset$ is its model but not an answer set of Π. The union of the two modules in Figure 3 captures all four inference rules and is indeed equal to the module in Figure 2.

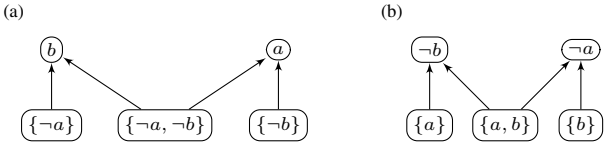

Fig. 3. Two abstract modules based on the program (4)

3 Abstract Modular Solver: AM_S

Finding models of logic theories and programs is a key computational task in declarative programming. Nieuwenhuis et al. [20] proposed to use *transition graphs* to describe search procedures involved in model-finding algorithms commonly called *solvers*, and developed that approach for the case of SAT. Their transition graph framework can express DPLL, the basic search procedure employed by SAT solvers, and its enhancements with techniques such as the conflict driven clause learning. Lierler and Truszczynski [12,14] proposed a similar framework to describe and analyze the answer-set programming solvers SMODELS, CMODELS [9] and CLASP [6], as well as a PC(ID) solver

MINISAT(ID) [17]. In the previous section, we argued that theories and programs can be represented by equivalent abstract inference modules (Propositions 4 and 5). We now show that the idea of a transition graph can be generalized to the setting of modules, leading to an abstract perspective on the problem of search for models of modules, and unifying the approaches to the model-finding task.

Let δ be a *finite* vocabulary. A *state over* δ is either a special state $\bot$ (the *fail* state) or a sequence M of *distinct* literals over δ, some possibly annotated by Δ, which marks them as *decision* literals, such that:

1. the set of literals in M is consistent or $M = M'l$, where the set of literals in M' is consistent and contains $\overline{l}$, and
2. if $M = M'l^{\Delta}M''$, then l is unassigned in M'.

For instance, if $\delta = \{a, b\}$, then $\emptyset$, a, $\neg a^{\Delta}\, b$, $\neg a\, b^{\Delta}\, a$ and $\bot$ are examples of states over δ. If M is a state, by $[M]$ we denote the *set* of the literals in M (that is, we drop annotations and ignore the order). Our definition of a state allows for inconsistent states. However, inconsistent states are of a very specific form — the inconsistence arises because of the last literal in the state. There is also a restriction on annotated (decision) literals. A decision literal must not appear in a state following another occurrence of that literal or its dual (annotated or not). Intuitively, a literal annotated by Δ denotes a current assumption: thus once a literal is assigned in a state, there is no point of later making an assumption concerning whether it holds or not.

Each module S determines its *transition graph* AM_S. The set of nodes of AM_S consists of all states relative to $\sigma(S)$. The edges of the graph AM_S are specified by the *transition rules* listed in Figure 4. The first rule depends on the module, the last three do not. They have the same form no matter what module we consider. Hence, we omit the reference to the module from its notation.

$$\textit{Propagate}_S: \quad M \longrightarrow Ml \text{ if } \begin{cases} [M] \text{ is consistent, } l \notin [M], \text{ and} \\ \text{for some } M' \subseteq [M],\ (M', l) \text{ is an inference of } S \end{cases}$$

$$\textit{Fail}: \quad M \longrightarrow \bot \text{ if } [M] \text{ is inconsistent and } M \text{ contains no decision literals}$$

$$\textit{Backtrack}: \quad P\, l^{\Delta}\, Q \longrightarrow P\, \overline{l} \text{ if } \begin{cases} [P\, l^{\Delta}\, Q] \text{ is inconsistent, and} \\ Q \text{ contains no decision literals} \end{cases}$$

$$\textit{Decide}: \quad M \longrightarrow M\, l^{\Delta} \text{ if } [M] \text{ is consistent and } l \text{ is unassigned by } [M]$$

Fig. 4. The transition rules of the graph AM_S

The graph AM_S can be used to decide whether a module S has a model. The following properties are essential.

Theorem 1. *For every module S,*

(a) graph AM_S is finite and acyclic,
(b) for any terminal state M of AM_S other than $\bot$, $[M]^+$ is a model of S,
(c) state $\bot$ is reachable from $\emptyset$ in AM_S if and only if S is unsatisfiable (has no models).

Thus, to decide whether a module S has a model it is enough to find in the graph AM_S a path leading from node $\emptyset$ to a terminal node M. If $M = \bot$, S is unsatisfiable. Otherwise, $[M]^+$ is a model of S. For instance, let S be a module in Figure 2. Below we show a path in the transition graph AM_S with every edge annotated by the corresponding transition rule:

$$\emptyset \quad \overset{\textit{Decide}}{\longrightarrow} \quad b^{\Delta} \quad \overset{\textit{Propagate}_S}{\longrightarrow} \quad b^{\Delta}\, \neg a. \tag{5}$$

The state $b^{\Delta}\, \neg a$ is terminal. Thus, Theorem 1(b) asserts that $\{b\}$ is a model of S. There may be several paths determining the same model. For instance, the path

$$\emptyset \quad \overset{\textit{Decide}}{\longrightarrow} \quad \neg a^{\Delta} \quad \overset{\textit{Decide}}{\longrightarrow} \quad \neg a^{\Delta}\, b^{\Delta}. \tag{6}$$

leads to the terminal node $\neg a^{\Delta}\, b^{\Delta}$, which is different from $b^{\Delta}\, \neg a$ but corresponds to the same model.

We can view a path in the graph AM_S starting in $\emptyset$ and ending in a terminal node as a description of a specific way to search for a model of module S. Each such path is determined by a function (strategy) selecting for each non-terminal state exactly one of its outgoing edges (exactly one applicable transition). Therefore, solvers based on the transition graph AM_S are uniquely determined by the "select-edge-to-follow" function. Such a function can be based, in particular, on assigning strict priorities to inferences in S. Below we describe an algorithm that captures "classical" DPLL strategy. Assuming M is the current state and it is not terminal, the algorithm proceeds as follows :

> if M is inconsistent and has no decision literals, follow the *Fail* edge (this is the only applicable transition); if M is inconsistent and has decision literals, follow the *Backtrack* edge (this is the only applicable transition); if M is consistent and $\textit{Propagate}_S$ applies, follow the edge implied by the highest priority inference of the form (M', l) in S such that $M' \subseteq [M]$; otherwise, follow the *Decide* edge.

This is still not a complete specification of a solver, as it offers no specification on how to select a decision literal (which of many possible *Decide* transitions to apply). Much of research on SAT solvers design, for example, has focused on this particular aspect and several heuristics were proposed over the years. Each such heuristics for selecting a decision literal when the *Decide* transition applies yields an algorithm.

Additional algorithms can be obtained by switching the preference over *Propagate* and *Decide* rules. Earlier, we selected a *Propagate* edge and only if impossible, we would select a *Decide* edge. But that order can be reversed resulting in another class of algorithms. Finally, we could even consider a more complicated selection functions that, when both *Decide* and *Propagate* edges are available, in some cases select a *Propagate* edge and in others a *Decide* one.

We now show how the approaches proposed by Nieuwenhuis et al. [20] and Lierler [12] to describe and analyze SAT and ASP solvers, respectively, fit in our abstract framework. Let F be a CNF formula that contains no empty clause. Nieuwenhuis et al. [20], defined the transition graph DP_F to capture the computation of the DPLL algorithm. We now review this graph in the form convenient for our purposes. All states

over the vocabulary of F form the vertices of DP_F. The edges of DP_F are specified by the three "generic" transition rules *Fail*, *Backtrack* and *Decide* of the graph AM_S, and the *Unit Propagate* rule below:

$$\textit{Unit Propagate}_F: \quad M \longrightarrow Ml \text{ if } \begin{cases} [M] \text{ is consistent, } l \notin [M], \text{ and} \\ \text{there is } C \vee l \in F, \text{ such that} \\ \text{for every } u \in C, \overline{u} \in [M] \end{cases}$$

For example, let F be the theory consisting of a single clause a. Figure 5 presents DP_F.

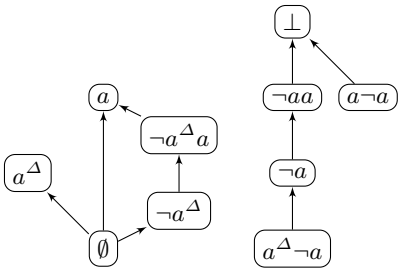

Fig. 5. The DP_F graph where $F = a$

It turns out that we can see the graph DP_F as the transition graph of the abstract module $UP(F)$.

Proposition 6. *For every CNF formula F with no empty clause,* $\text{DP}_F = \text{AM}_{UP(F)}$.

Theorem 1, Proposition 6, and the fact that a CNF formula F and the module $UP(F)$ are equivalent (Proposition 4) imply the following result.

Corollary 1. *For any CNF formula F,*

(a) graph DP_F is finite and acyclic,
(b) for any terminal state M of DP_F other than $\bot$, $[M]^+$ is a model of F,
(c) state $\bot$ is reachable from $\emptyset$ in DP_F if and only if F is unsatisfiable (has no models).

This is precisely the result stated by Nieuwenhuis et al. [20] and used to argue that the graph DP_F is an abstraction of the DPLL method. To decide the satisfiability of F (and to find a model, if one exists), it is enough to find a path leading from the state $\emptyset$ to a terminal state M. If $M = \bot$ then F is unsatisfiable; otherwise, $[M]^+$ is a model of F. For instance, the only terminal states reachable from the state $\emptyset$ in DP_F are a and a^Δ. This translates into the fact that $\{a\}$ is a model of F. Specific algorithm encapsulated by the graph DP_F (equivalently, $\text{AM}_{UP(F)}$) can be obtained by deciding on a way to select an edge while in a consistent state. Typical implementations of basic backtracking SAT solvers follow a *Unit Propagate*$_F$ edge whenever possible, choosing *Decide* edges only if nothing else applies. These algorithms differ from each other in the heuristics they use for the selection of a decision literal.

Next, we show that our abstract approach to model generation in logics applies to answer-set programming [8,16,18]. Lierler [12] introduced a transition system SM_Π to

describe and study the SMODELS solver. We first review the graph SM_{Π} and then show that Lierler's approach can be viewed as an instantiation of our general theory.

The set of nodes of the graph SM_{Π} consists of all states relative to the vocabulary of program Π. The edges of SM_{Π} are specified by the transition rules of the graph $\mathrm{DP}_{\Pi^{cl}}$ and the rules presented in Figure 6.

$$\textit{Unfounded}_{\Pi}: \quad M \longrightarrow M\neg a \text{ if } [M] \text{ is consistent, } \neg a \notin [M], \text{ and } a \in U([M], \Pi)$$

$$\textit{All Rule Cancelled}: \quad M \longrightarrow M\neg a \text{ if } \begin{cases} [M] \text{ is consistent, } \neg a \notin [M], \text{ and} \\ \text{for every } B \in \mathit{Bodies}(\Pi, a), \\ \text{there is } u \in s(B) \text{ such that } \overline{u} \in [M] \end{cases}$$

$$\textit{Backchain True}: \quad M \longrightarrow Ml \text{ if } \begin{cases} [M] \text{ is consistent, } l \notin [M] \\ \text{for some } a \leftarrow B \in \Pi, a \in [M], l \in s(B), \text{ and} \\ \text{for every } B' \in \mathit{Bodies}(\Pi, a) \setminus \{B\}, \\ \text{there is } u \in s(B') \text{ such that } \overline{u} \in [M] \end{cases}$$

Fig. 6. Transition rules of the graph SM_{Π}

The following result shows that Lierler's approach can be viewed as an instantiation of our general theory.

Proposition 7. *For every logic program Π, $\mathrm{SM}_{\Pi} = \mathrm{AM}_{\mathit{smodels}(\Pi)}$.*

Indeed, this proposition, Theorem 1 and the fact that Π is equivalent to the module $\mathit{smodels}(\Pi)$ (Proposition 5) imply the result stemming from Lierler [12].

Corollary 2. *For every logic program Π,*

(a) graph SM_{Π} ($\mathrm{AM}_{\mathit{smodels}(\Pi)}$) is finite and acyclic,
(b) for any terminal state M of SM_{Π} ($\mathrm{AM}_{\mathit{smodels}(\Pi)}$) other than $\bot$, M^+ is an answer set of Π,
(c) state $\bot$ is reachable from $\emptyset$ in SM_{Π} ($\mathrm{AM}_{\mathit{smodels}(\Pi)}$) if and only if Π has no answer sets.

Since $\mathit{UPUF}(\Pi)$ is also equivalent to Π, we obtain a similar corollary for the transition graph $\mathrm{AM}_{\mathit{UPUF}(\Pi)}$. Intuitively, this graph is characterized by the transition rules of the graph $\mathrm{DP}_{\Pi^{cl}}$ as well as the rule *Unfounded* presented in Figure 6. Thus, $\mathrm{AM}_{\mathit{UPUF}(\Pi)}$ is a model of another correct algorithm for finding answer sets of programs. In fact, it is so for any module S such that $\mathit{UPUF}(\Pi) \subseteq S \subseteq \mathit{smodels}(\Pi)$.

Also the graph SM_{Π} describes a whole family of backtracking search algorithms for finding answer sets of programs. They differ from each other by the way we select an edge while in a consistent state. The selection function could be based on priorities of the propagation rules.

Our discussion of SAT and ASP solvers shows that the framework of modules uniformly encompasses different logics. Furthermore, it uniformly models diverse reasoning mechanisms (the logical entailment, reasoning under specific inference rules). Our results also show that transition graphs proposed earlier to represent and analyze SAT and ASP solvers are special cases of transition graphs for abstract inference modules.

4 Abstract Modular System and Solver AMS$_{\mathcal{A}}$

By capturing diverse logics in a single framework, abstract modules are well suited for studying modularity in declarative formalisms and for analyzing solvers for such modular formalisms. As illustrated by our examples, abstract inference modules can capture reasoning of various logics including classical reasoning with propositional theories and reasoning with programs under the answer-set semantics. Putting modules together provides an abstract uniform way to represent hybrid modular systems, in which modules represent theories from different logics.

We now define an abstract modular declarative framework that uses the concept of a module as its basic element. We then show how abstract transition graphs for modules generalize to the new formalism.

Definition 2. *An* abstract modular inference system *(AMS) is a finite set of abstract inference modules. The vocabulary of an AMS $\mathcal{A}$ is the union of the vocabularies of modules of $\mathcal{A}$ (they do not have to have the same vocabulary); we denote it by $\sigma(\mathcal{A})$. A set $X \subseteq \sigma$ is a* model *of $\mathcal{A}$, if X is a model of every module $S \in \mathcal{A}$.*

Let S_1 be a module presented in Figure 1(b) and S_2 be a module in Figure 2. The vocabulary of the AMS $\mathcal{A} = \{S_1, S_2\}$ consists of the atoms a and b. It is easy to see that the set $\{a\}$ is the only model of $\mathcal{A}$ contained in $\sigma(\mathcal{A})$ (more generally, a set X is a model of $\mathcal{A}$ if and only if X contains a). In Section 2, we observed that $S_1 = Ent(T)$ (and also $= UP(T)$), for a propositional theory T, and that $S_2 = smodels(\Pi)$, where Π is the program given by (4). This illustrates how abstract modular systems can serve as an abstraction for heterogeneous multi-logic systems.

For a general example of a modular declarative formalism that can be cast as an abstract modular system we now discuss the case of modular logic programs [15]. The semantics of modular logic programs relies on the notion of an input answer set of a program [14]. A set X of atoms is an *input answer set* of a logic program Π if X is an answer set of the program $\Pi \cup (X \setminus Head(\Pi))$, where $Head(\Pi)$ denotes the set of all head atoms of Π. Informally, input answer sets treat all atoms *not occurring* in the heads of program rules as *open* so that they can assume any logical value. These atoms are viewed as the "input." To capture the semantics of input answer sets in terms of inferences, we introduce a modified version of the propagation rule *Unfounded*:

Unfounded': derive l if $l = \neg a$, $a \in U(M, \Pi)$ and $a \in Head(\Pi)$.

This rule gives rise to an inference module $UF'(\Pi)$ defined by taking the condition of the rule as defining when (M, l) is to be an inference of the module. With the module $UF'(\Pi)$ in hand, we define $UPUF'(\Pi) = UP(\Pi) \cup UF'(\Pi)$.

An inference module S is *input-equivalent* to a logic program Π if input answer sets of Π coincide with models of S. We now restate Proposition 5 for the case of input-equivalence.

Proposition 8. *Every program Π is input-equivalent to the module $UPUF'(\Pi)$.*

A *modular (logic) program* is a set of logic programs [15]. For a modular program $\mathcal{P}$, a set X of atoms is an *answer set* of $\mathcal{P}$ if X is an input answer set of every program Π in $\mathcal{P}$. An AMS $\mathcal{A}$ is *equivalent* to a modular program $\mathcal{P}$ if answer sets of $\mathcal{P}$ coincide with models of $\mathcal{A}$.

Proposition 9. *Every modular program* $\{\Pi_1, \ldots, \Pi_n\}$ *is equivalent to the abstract modular system* $\{UPUF'(\Pi_1), \ldots, UPUF'(\Pi_n)\}$.

Theories in the logics SM(ASP) [15] and PC(ID) [17] can be cast as abstract modular systems in the same manner.

We now resume our study of general properties of abstract modular systems. For an AMS $\mathcal{A} = \{S_1, \ldots, S_n\}$, we define $\mathcal{A}^{\cup} = S_1 \cup \ldots \cup S_n$. We can now state the result showing that modular systems can be expressed in terms of a single abstract inference module.

Theorem 2. *Every abstract modular inference system* $\mathcal{A}$ *is equivalent to the abstract inference module* $\mathcal{A}^{\cup}$.

Each AMS $\mathcal{A}$ determines its *transition graph* $\text{AMS}_{\mathcal{A}}$, which we define by setting $\text{AMS}_{\mathcal{A}} = \text{AM}_{\mathcal{A}^{\cup}}$. Theorem 1 implies the following result.

Theorem 3. *For every AMS* $\mathcal{A}$*,*

(a) the graph $\text{AMS}_{\mathcal{A}}$ *is finite and acyclic,*
(b) any terminal state M *of* $\text{AMS}_{\mathcal{A}}$ *other than* $\bot$ $[M]^+$ *is a model of* $\mathcal{A}$*,*
(c) the state $\bot$ *is reachable from* $\emptyset$ *in* $\text{AMS}_{\mathcal{A}}$ *if and only if* $\mathcal{A}$ *is unsatisfiable.*

As in other cases, Theorem 3 shows that the graph $\text{AMS}_{\mathcal{A}}$ is an abstract representation of an algorithm to decide satisfiability of a modular system $\mathcal{A}$. Such algorithm searches in $\text{AMS}_{\mathcal{A}}$ for a path leading from node $\emptyset$ to a terminal node by moving from a node to node, selecting any edge originating in the current node. Theorem 3 guarantees that the method terminates, the other two parts of that component ensure correctness.

For instance, let $\mathcal{A}$ be the AMS $\{S_1, S_2\}$ where S_1 is a module in Figure 1(b) and S_2 is a module in Figure 2. Below is a valid path in the transition graph $\text{AMS}_{\mathcal{A}}$ with every edge annotated by the corresponding transition rule:

$$\emptyset \overset{Decide}{\longrightarrow} \neg a^{\Delta} \overset{Propagate_{S_2}}{\longrightarrow} \neg\, a^{\Delta}\, b \overset{Propagate_{S_1}}{\longrightarrow} \neg\, a^{\Delta}\, b\, a \overset{Backtrack}{\longrightarrow} a \overset{Decide}{\longrightarrow} a\, \neg b^{\Delta}.$$

The state $a\, \neg b^{\Delta}$ is terminal. Thus, Theorem 3 (b) asserts that $\{a, \neg b\}$ is a model of $\mathcal{A}$. Let us interpret this example. Earlier we demonstrated that module S_1 can be regarded as a representation of a propositional theory consisting of a single clause a whereas S_2 corresponds to the logic program (4) under the semantics of answer sets. We then illustrated how modules S_1 and S_2 give rise to particular algorithms for implementing search procedures. The graph $\text{AMS}_{\mathcal{A}}$ represents the algorithm obtained by *integrating* the algorithms supported by the modules S_1 and S_2 separately.

We will now discuss some classes of algorithms captured by the graph $\text{AMS}_{\mathcal{A}}$. As before, they are more specifically determined by a strategy of selecting an outgoing edge from the current state. Let us assume that such a strategy is available for each module $S \in \mathcal{A}$. Let us also assume that the modules in $\mathcal{A}$ are prioritized. This leads to an algorithm that proceeds as follows (assuming M is the current state and it is not terminal):

> if M is inconsistent, we always select the *Fail* or *Backtrack* edge (whichever is applicable); if M is consistent then we select an edge determined by the highest priority inference from the highest priority module.

Assuming that modules in $\mathcal{A}$ are enumerated $S_1, \ldots, S_k$ from the highest priority one to the lowest, the described algorithm works as follows. It starts by moving along edges implied by inferences of the module S_1 (according to the selection strategy for that module). If we reach $\bot$, the entire search is over with failure. Otherwise, we reach a consistent state, in which no inference from module S_1 is applicable (that state represents a model of S_1). The phase of search involving module S_1 gets suspended and we continue in the same way but now following edges determined by inferences in module S_2. In other words, we start the phase of the search involving module S_2. If we reach $\bot$, the search is over with failure. If we reach an inconsistent state that contains decision literals, we apply the *Backtrack* rule. If that rule backtracks to a literal introduced after we moved to module S_2, we remain in the module S_2 phase and continue. If the backtrack takes us back to a literal introduced while a higher priority module was considered (in this case, that must be module S_1), we resume the module S_1 phase of the search suspended earlier. If *Propagate* or *Decide* edges in module S_2 are available, we select one of them following the strategy for module S_2. If we reach a consistent state with no outgoing edges implied by inferences of S_2 (that state represents a model of both S_1 and S_2) we suspend the module S_2 phase and start the module S_3 phase, and continue in that way until a terminal state is reached.

The main advantage of such an algorithm is that each phase is concerned only with inferences coming from a single module and state changes involve only literals from the vocabulary of that module. The literals established during phases involving higher priority modules remain fixed. Thus, the search space in each phase is effectively limited to that of the module involved in that phase.

Clearly, other specializations of the graph $\text{AMS}_{\mathcal{A}}$ are possible. For instance, we may alternate between modules in a more arbitrary way, possibly switching from the current module to another even in situations when the current state has outgoing edges implied by the inferences of the current module. However, such algorithms may have to work with search spaces that are larger than the search space for a single module.

DLVHEX: Our results apply to a version of the DLVHEX[3] solver [5] restricted to logic programs. DLVHEX computes models of *HEX-programs* by exploiting their modularity, that is, representing programs as an equivalent modular program. Answer set programs consisting of rules of the form (2) form a special class of HEX-programs. Therefore, DLVHEX restricted to such programs can be seen as an answer-set solver that exploits their modularity. Given a program Π, DLVHEX starts its operation by constructing a modular program $\mathcal{P} = \{\Pi_1, \ldots, \Pi_n\}$ so that (i) $\Pi = \Pi_1 \cup \cdots \cup \Pi_n$ and (ii) answer sets of $\mathcal{P}$ coincide with answer sets of Π. It then processes modules one after another according to an order determined by the structure of a program. That process can be modeled in abstract terms described above. In particular, the graph $\text{AMS}_{\{UPUF'(\Pi_1),\ldots,UPUF'(\Pi_n)\}}$ can be seen as an abstraction capturing the family of DLVHEX-like algorithms based on *Unit Propagate* and *Unfounded′* inferences.

[3] http://www.kr.tuwien.ac.at/research/systems/dlvhex/

5 Related Work and Conclusions

In an important development, Brewka and Eiter [3] introduced an abstract notion of a *heterogeneous nonmonotonic multi-context system* (MCS). One of the key aspects of that proposal is its abstract representation of a logic that allows one to study MCSs without regard to syntactic details. The independence of contexts from syntax promoted focus on semantic aspect of modularity in multi-context systems. Since their inception, multi-context systems have received substantial attention and inspired implementations of hybrid reasoning systems including DLVHEX [5] and DMCS [1]. There are some similarities between AMSs and MCSs. However, there is also a key difference. MCSs provide an abstract framework to define semantics of hybrid systems. In contrast, AMSs explicitly represent inferences of a logic and provide an abstract framework for studying model-generation algorithms. On a more technical level, another notable difference concerns information sharing among modules. MCSs use to this end the so-called "bridge rules." In AMS information sharing is implemented by a simple notion of sharing parts of the vocabulary between the modules.

Modularity is one of the key techniques in principled software development. This has been a major trigger inspiring research on modularity in declarative programming paradigms rooting in KR languages such as answer-set programming, for instance. Oikarinen and Janhunen [21] proposed a modular version of answer-set programs called lp-modules. In that work, the authors were primarily concerned with the decomposition of lp-modules into sets of simpler ones. They proved that under some assumptions such decompositions are possible. Järvisalo, Oikarinen, Janhunen, and Niemelä [11], and Tasharrofi and Ternovska [23] studied the generalizations of lp-modules. In their work the main focus was to abstract lp-modules formalism away from any particular syntax or semantics. They then study properties of the modules such as "joinability" and analyze *different ways* to join modules together and the semantics of such a join. We are interested in building simple modular systems using abstract modules – the only composition mechanism that we study is based on conjunction of modules. Also in contrast to the work by Järvisalo et al. [11] and Tasharrofi and Ternovska [23], we define such conjunction for any modules disregarding their internal structure and interdependencies between each other.

Tasharrofi, Wu, and Ternovska [24] developed and studied an algorithm for processing modular model expansion tasks in the abstract multi-logic system concept developed by Tasharrofi and Ternovska [23]. They use the traditional pseudocode method to present the developed algorithm. In this work we adapt the graph-based framework for designing backtrack search algorithms for abstract modular systems. The benefits of that approach for modeling families of backtrack search procedures employed in SAT, ASP, and PC(ID) solvers were demonstrated by Nieuwenhuis et al. [20], Lierler [12], and Lierler and Truszczynski [14]. Our work provides additional support for the generality and flexibility of the graph-based framework as a finer abstraction of backtrack search algorithms than direct pseudocode representations, allowing for convenient means to prove correctness and study relationships between the families of the algorithms.

Gebser and Schaub [7] describe a form of a tableaux system to describe inferences involved in computing answer sets. Several rules used in their approach are closely

related to those we discussed in the context of modules designed to represent reasoning on logic programs. However, the two approaches are formally different. Most notably, the concepts of states in a tableaux and in an abstract module are different. Still, there seems to be a connection between them, which we plan to investigate in our future work.

We introduced abstract modules and abstract modular systems and showed that they provide a framework capable of capturing diverse logics and inference mechanisms integrated into modular knowledge representation systems. In particular, we showed that transition graphs determined by modules and modular systems provide a unifying representation of model-generating algorithms, or solvers, and simplify reasoning about such issues as correctness or termination. We believe they can be useful in theoretical comparisons of solver effectiveness and in the development of new solvers.

References

1. 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.) JELIA 2010. LNCS, vol. 6341, pp. 352–355. Springer, Heidelberg (2010)
2. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsch, T. (eds.) Handbook of Satisfiability, pp. 737–797. IOS Press (2008)
3. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp. 385–390 (2007)
4. Denecker, M., Lierler, Y., Truszczynski, M., Vennekens, J.: A Tarskian informal semantics for answer set programming. In: Dovier, A., Costa, V.S. (eds.) International Conference on Logic Programming (ICLP). LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
5. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer set programming. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), pp. 90–96. Professional Book Center (2005)
6. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 386–392. MIT Press, Cambridge (2007)
7. Gebser, M., Schaub, T.: Tableau calculi for answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 11–25. Springer, Heidelberg (2006)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium, pp. 1070–1080. MIT Press (1988)
9. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36, 345–377 (2006)
10. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19(20), 503–581 (1994)
11. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 155–168. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04238-6_15

12. Lierler, Y.: Abstract answer set solvers with backjumping and learning. Theory and Practice of Logic Programming 11, 135–169 (2011)
13. Lierler, Y.: On the relation of constraint answer set programming languages and algorithms. In: Proceedings of the AAAI Conference on Artificial Intelligence. MIT Press (2012)
14. Lierler, Y., Truszczynski, M.: Transition systems for model generators — a unifying approach. In: Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP 2011), Special Issue 11(4-5) (2011)
15. Lierler, Y., Truszczynski, M.: Modular answer set solving. In: Proceedings of Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2013) (2013)
16. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer (1999)
17. Mariën, M., Wittocx, J., Denecker, M., Bruynooghe, M.: SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 211–224. Springer, Heidelberg (2008)
18. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
19. Niemelä, I., Simons, P.: Extending the Smodels system with cardinality and weight constraints. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 491–521. Kluwer (2000)
20. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
21. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: 17th European Conference on Artificial Intelligence (ECAI), pp. 412–416 (2006)
22. Rossi, F., van Beek, P., Walsh, T.: Constraint programming. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 181–212. Elsevier (2008)
23. Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 259–274. Springer, Heidelberg (2011)
24. Tasharrofi, S., Wu, X.N., Ternovska, E.: Solving modular model expansion tasks. CoRR abs/1109.0583 (2011)
25. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of ACM 38(3), 620–650 (1991)