Abstract Modular 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 concepts of *abstract modules* and *abstract modular systems* to study general principles behind the design and analysis of model-finding programs, or *solvers*, for integrated heterogeneous multi-logic systems. We show how abstract modules and abstract modular systems give rise to *transition systems*, which are a natural and convenient representation of solvers 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. Similar issues arise in research on multi-context systems where the major task is to model *contextual* information and the flow of information among *contexts* [17, 7]. The contexts are commonly modeled by theories in some logics.

Modeling convenience is not the only reason why diverse logics are combined into modular hybrid KR systems. Another major motivation is to exploit in reasoning the transparent structure that comes from modularity, computational strengths of individual logics, and synergies that may arise when they are put together. Constraint logic programming [8] 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], which integrates answer set programming (ASP) [6, 15, 18]) with constraint modeling languages [22], and "multi-logic" formalisms PC(ID) [16], 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 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. The basic elements of this language are *abstract modules*. Collections of abstract modules constitute *abstract modular systems*. We define the semantics of abstract modules and show that they provide a uniform language capable of capturing different logics, diverse inference mechanisms, and their modular combinations. Importantly, abstract modules and abstract modular systems give rise to *transition systems* of the type introduced by Nieuwenhuis, Oliveras, and Tinelli [20] in their study of SAT and SMT solvers. We show that as in that earlier work, our transition systems provide a natural and convenient representation of solvers for abstract modules and abstract modular systems. We demonstrate that they lend themselves well to extensions that capture such important solver design techniques as learning (which here comes in two flavors: *local* that is limited to single modules, and *global* that is applied across modules). 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.

The results of our paper show that abstract modular systems and the corresponding abstract framework for describing and analyzing algorithms for modular declarative programming tools relying on multi-logics are useful and effective conceptualizations that can contribute to (i) clarifying computational principles of such systems and to (ii) the development of new ones.

The paper is organized as follows. We start by introducing one of the main concepts in the paper – abstract modules. We then proceed to formulating an algorithm (a family of algorithms) for finding models of such modules. We use an abstract transition system stemming from the framework by Nieuwenhuis et al. [20] for this purpose. Section 4 presents the definition of an abstract modular system and a corresponding solver based on backtrack search. We then discuss how this solver maybe augmented by such advanced SAT solving technique as learning. Section 6 provides an account on related work.

2 Abstract Modules

Let σ be a fixed finite vocabulary (a set of propositional atoms). A *module* over the vocabulary σ is a directed graph S whose nodes are $\perp$ and all consistent sets of literals,

and each edge is of the form $(M, \bot)$ or (M, Ml), where $l \notin M$ and Ml is a shorthand for $M \cup \{l\}$. If S is a module, we write $\sigma(S)$ for its vocabulary. For a set X of literals, we denote $X^+ = \{a\colon a \in X\}$ and $X^- = \{a\colon \neg a \in X\}$.

Intuitively, an edge (M, Ml) in a module indicates that the module supports inferring l whenever all literals in M are given. An edge $(M, \bot)$, $M \neq \emptyset$, indicates that there is a literal $l \in M$ such that a derivation of its dual $\bar{l}$ (and hence, a derivation of a contradiction) is supported by the module, assuming the literals in M are given. Finally, the edge $(\emptyset, \bot)$ indicates that the module is "explicitly" contradictory.

A node in a module is *terminal* if no edge leaves it. A terminal node that is consistent and complete is a *model node* of the module. A set X of atoms is a *model* of a module S if for some model node Y in S, $X \cap \sigma(S) = Y^+$. Thus, models of modules are not restricted to the signature of the module. Clearly, for every model node Y in S, Y^+ is a model of S.

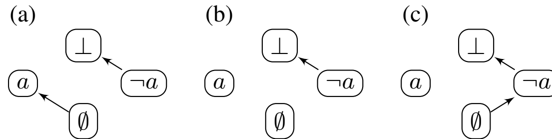

Fig. 1: Three modules over the vocabulary $\{a\}$.

A module S *entails* a formula φ, written $S \models \varphi$, if for every model I of S we have $I \models \varphi$. It is immaterial what logic the formula φ comes from as long as (i) the vocabulary of the logic is a subset of the vocabulary of S, and (ii) the semantics of the logic is given by a satisfiability relation $I \models \varphi$. A module S *entails* a formula φ wrt a set M of literals (over the same vocabulary as S), written $S \models_M \varphi$, if for every model I of S such that $M^+ \subseteq I$ and $M^- \cap I = \emptyset$, $I \models \varphi$.

Clearly, if two modules over the same signature have the same model nodes, they have the same models. Semantically the three modules in Figure 1 are the same. They have the same models (each has $\{a\}$ as its only model in the signature of the module) and so they entail the same formulas. We call modules with the same models *equivalent*.

Modules represent more than just the set of their models. As already suggested above, the intended role of edges in a module is to represent allowed "local" inferences. For instance, given the empty set of literals, the first module in Figure 1 supports inferring a and the third module $\neg a$. In the latter case, the inference is not "sound" as it contradicts the semantic information in the module as that module does not entail $\neg a$ with respect to the empty set of literals.

Formally, an edge from a node M to a node M' in a module S is *sound* if $S \models_M M'$.[3] Clearly, if M' has the form Ml then $S \models_M M'$ if and only if $S \models_M l$. Similarly, if $M' = \bot$ then $S \models_M M'$ if and only if no model of S is consistent with M (that is, contains M^+ and is disjoint with M^-). A module is *sound* if all of its edges are sound,

[3] In the paper, we sometimes identify a set of literals with the conjunction of its elements. Here M' is to be understood as the conjunction of its elements.

that is, if all inferences supported by the module are sound with respect to the semantics of the module given by its set of models. The modules in Figures 1(a) and (b) are sound, the one in Figure 1(c) is not. Namely, the inference of $\neg a$ from $\emptyset$ is not sound.

Given two modules S and S' over the same vocabulary, we say that S is *equivalently contained* in S', $S \sqsubseteq S'$, if S and S' are equivalent (have the same model nodes) and the set of edges of S is a subset of the set of edges of S'. Maximal (wrt $\sqsubseteq$) sound modules are called *saturated*. We say that an edge from a node M to $\bot$ in a module S is *critical* if M is a complete and consistent set of literals over $\sigma(S)$. The following properties are evident.

Proposition 1. *Every two modules over the same signature and with the same critical edges are equivalent. For a saturated module S, every sound module with the same critical edges as S is equivalently contained in S. A module S is saturated if and only if it is sound and for every set M of literals and for every literal $l \notin M$, (M, Ml) is an edge of S whenever $S \models_M l$.*

Clearly, only the module in Figure 1(a) is saturated. The other two are not. The one in (b) is not maximal with respect to the containment relation, the one in (c) is not sound. We also note that all three modules have the same critical edges. Thus, by Proposition 1, they are equivalent, a property we already observed earlier. Finally, the module in Figure 1(b) is equivalently contained in the module in Figure 1(a).

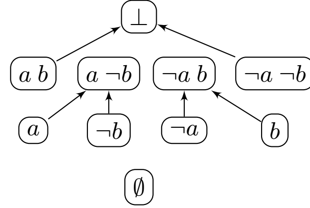

Fig. 2: An abstract module over the vocabulary $\{a, b\}$ related to the theory (1).

In practice, modules (graphs) are specified by means of theories and logics (more precisely, specific forms of inference in logics). For instance, a propositional theory T over a vocabulary σ and the inference method given by the classical concept of entailment determine a module over σ in which (i) (M, Ml) is an edge if and only if $T \cup M \models l$; and (ii) $(M, \bot)$ is an edge if and only if no model of T is consistent with M. Figure 1(a) shows the module determined in this way by the theory consisting of the clause a. Similarly, Figure 2 presents such a module for the theory

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

This module is saturated. Also, theory (1) and the inference method given by the unit propagate rule, a classical propagator used in SAT solvers, determines this module. In other words, for the theory (1) the unit propagation rule captures entailment.

We say that a module S is *equivalent* to a theory T in some logic if the models of S coincide with the models of T. Clearly, the module in Figure 2 is equivalent to the propositional theory (1).

Modules are not meant for modeling. Representations by means of logic theories are usually more concise (the size of a module is exponential in the size of its vocabulary). 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 to natural language.

The power of modules comes from the fact that they provide a uniform, syntax-independent way to describe theories and inference methods stemming from *different* logics. For instance, they represent equally well both propositional theories and logic programs under the answer-set semantics. Indeed, let us consider the logic program

$$\begin{array}{l} \{a\}, \\ b \leftarrow not\ a, \end{array} \tag{2}$$

where $\{a\}$ represents the so-called *choice rule* [23]. This program has two answer sets $\{a\}$ and $\{b\}$. Since these are also the only two models of the propositional theory (1), it is clear that the module in Figure 2 represents the program (2) and the reasoning mechanism of entailment with respect to its answer sets. Two other modules associated with program (2) are given in Figure 3. The module in Figure 3(a) represents program (2) and the reasoning on programs based on *forward chaining*; we call this module M_{fc}. We recall that given a set of literals, forward chaining supports the derivation of the head of a rule whose body is satisfied. We note that the module M_{fc} is not equivalent to program (2). Indeed, $\{a, b\}$ is a model of M_{fc} whereas it is not an answer set of (2). This is due to the fact that the critical edge from $a\,b$ to $\bot$ is unsupported by forward chaining and is not present in M_{fc}. On the other hand, all edges due to forward chaining are sound both in the module in Figure 2, which we call M_e, and M_{fc}. In the next section we discuss a combination of inference rules that yields a reasoning mechanism subsuming forward chaining and resulting in a module, shown in Figure 3(b), that is equivalently contained in M_e and so, equivalent to the program (2). This discussion indicates that the language of modules is flexible enough to represent not only the semantic mechanism of entailment, but also syntactically defined "proof systems" — reasoning mechanisms based on specific inference rules.

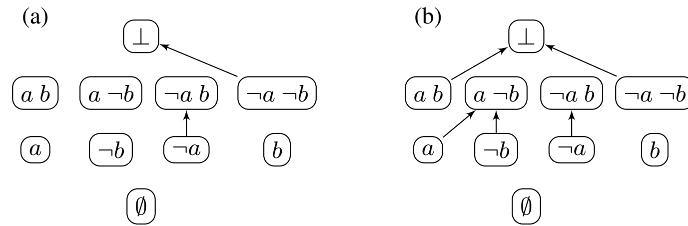

Fig. 3: Two abstract modules over the vocabulary $\{a, b\}$ related to the logic program (2).

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 systems to describe search procedures involved in model-finding programs commonly called *solvers*, and developed that approach for the case of SAT. Their transition system framework can express DPLL, the basic search procedure employed by SAT solvers, and its enhancements such as conflict driven clause learning. Lierler [12] proposed a similar framework for specifying an answer set solver SMODELS. Lierler and Truszczynski [14] extended that framework to capture such modern ASP solvers as CMODELS and CLASP, as well as a PC(ID) solver MINISAT(ID).

An abstract nature (independence from language and reasoning method selection) of modules introduced in this work and their relation to proof systems makes them a convenient, broadly applicable tool to study and analyze solvers. In this section, we adapt the transition system framework of Nieuwenhuis et al. [20] to the case of abstract modules. We then illustrate how it can be used to define solvers for instantiations of abstract modules such as propositional theories under the classical semantics and logic programs under the answer-set semantics.

A *state* relative to σ is either a special state $\bot$ (fail state) or an *ordered* consistent set M of literals over σ, some possibly annotated by Δ, which marks them as *decision* literals. For instance, the states relative to a singleton set $\{a\}$ of atoms are $\emptyset,\ a,\ \neg a,\ a^{\Delta},\ \neg a^{\Delta},\ \bot$.

Frequently, we consider a state M as a set of literals, ignoring both the annotations and the order between its elements. If neither a literal l nor its complement occur in M, then l is *unassigned* by M.

Each module S determines its *transition graph* AM_S: The set of nodes of AM_S consists of the states relative to the vocabulary of S. The edges of the graph AM_S are specified by the *transition rules* listed in Figure 4. The first three rules depend on the module, the fourth rule, *Decide*, does not. It has the same form no matter what module we consider. Hence, we omit the reference to the module from its notation.

Propagate$_S$: $\quad M \longrightarrow M\ l$ if S has an edge from M to $M\ l$

Fail$_S$: $\quad M \longrightarrow \bot$ if $\begin{cases} S \text{ has an edge from } M \text{ to } \bot, \\ M \text{ contains no decision literals} \end{cases}$

Backtrack$_S$: $\quad P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l}$ if $\begin{cases} S \text{ has an edge from } P\ l\ Q \text{ to } \bot, \\ Q \text{ contains no decision literals} \end{cases}$

Decide: $\quad M \longrightarrow M\ l^{\Delta}$ if l 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 sound 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 sound 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:

$$\emptyset \quad \xrightarrow{Decide} \quad b^{\Delta} \quad \xrightarrow{Propagate_S} \quad b^{\Delta}\ \neg a. \tag{3}$$

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

$$\emptyset \quad \xrightarrow{Decide} \quad \neg a^{\Delta} \quad \xrightarrow{Decide} \quad \neg a^{\Delta}\ b^{\Delta}. \tag{4}$$

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 as a description of a process of search for a model of module S by applying transition rules. Therefore, we can characterize a solver based on the transition system AM_S by describing a strategy for choosing a path in AM_S. Such a strategy can be based, in particular, on assigning priorities to some or all transition rules of AM_S, so that a solver will never apply a transition rule in a state if a rule with higher priority is applicable to the same state. For example, priorities

$$\mathit{Backtrack}_S, \mathit{Fail}_S >> \mathit{Propagate}_S >> \mathit{Decide}$$

on the transition rules of AM_S specify a solver that follows available inferences (modeled by edges in the module S) before executing a transition due to *Decide*. The path (3) in the transition graph of the module from Figure 2 follows that strategy, whereas the path (4) does not.

We now review the graph DP_F introduced for the classical DPLL algorithm by Nieuvenhuis et al. [20], adjusting the presentation to the form convenient for our purposes. We then demonstrate its relation to the AM_S graph. The set of nodes of DP_F consists of the states relative to the vocabulary of a CNF formula (a set of clauses) F. The edges of the graph DP_F are specified by the transition rule *Decide* of the graph AM_S and the rules presented in Figure 5. For example, let F_1 be the theory consisting of a single clause a. Figure 6 presents DP_{F_1}.

For a CNF formula F, by $\mu(\mathrm{DP}_F)$ we denote the graph (abstract module) constructed from DP_F by dropping all nodes that contain decision literals. We note that for the graph DP_{F_1} in Figure 6, the module $\mu(\mathrm{DP}_{F_1})$ coincides with the module in Figure 1(a). This is a manifestation of a general property.

Proposition 2. *For every CNF formula F, the graph $\mu(\mathrm{DP}_F)$ is a sound abstract module equivalent to F. Furthermore, the graphs $\mathrm{AM}_{\mu(\mathrm{DP}_F)}$ and DP_F are identical.*

$$\textit{UnitPropagate}_F: \quad M \longrightarrow M\, l \;\text{ if } \begin{cases} C \vee l \in F \text{ and } M \models \neg C, \\ l \text{ is unassigned by } M \end{cases}$$

$$\textit{Fail}_F: \quad M \longrightarrow \bot \;\text{ if } \begin{cases} C \in F \text{ and } M \models \neg C, \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_F: \quad P\, l^{\Delta}\, Q \longrightarrow P\, \bar{l} \;\text{ if } \begin{cases} C \in F \text{ and } P\, l^{\Delta}\, Q \models \neg C, \\ Q \text{ contains no decision literals} \end{cases}$$

Fig. 5: Three transition rules of the graph DP_F.

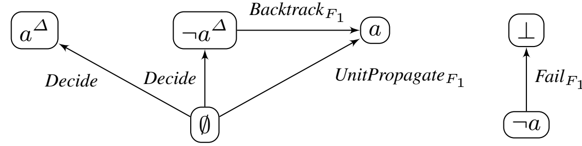

Fig. 6: The DP_{F_1} graph where F_1 is a single clause a.

Theorem 1 and the fact that the module $\mu(\text{DP}_F)$ is equivalent to a CNF formula F (Proposition 2) imply that the graph DP_F can be used for deciding the satisfiability of F. It is enough to find a path leading from node $\emptyset$ to a terminal node 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_1} are a and a^{Δ}. This translates into the fact that a is a model of F_1. This is exactly the result that Nieuwenhuis et al. [20] stated for the graph DP_F:

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).

We now introduce the graph AS_Π that extends the DPLL graph by Nieuwenhuis et al. so that the result can be used to specify an algorithm for finding answer sets of a program. The graph AS_Π can be used to form a sound module equivalent to a program Π in the same way as we used DP_F to form a sound module equivalent to a CNF formula F.

We assume the reader to be familiar with the concept of *unfounded sets* [26, 10]. For a set M of literals and a program Π, by $U(M, \Pi)$ we denote an unfounded set on M w.r.t. Π. It is common to identify logic rules of a program with sets of clauses. By Π^{cl} we denote the set of clauses corresponding to the rules of Π. For instance, let Π be (2), then Π^{cl} consists of clauses $a \vee \neg a, a \vee b$.

The set of nodes of AS_Π consists of the states relative to the vocabulary of program Π. The edges of the graph AS_Π are specified by the transition rules of the graph $\text{DP}_{\Pi^{cl}}$ and the rules presented in Figure 7.

For a program Π, by $\mu(\text{AS}_\Pi)$ we denote the graph (abstract module) constructed from AS_Π by removing all nodes that contain decision literals.

$$\textit{Unfounded}_{\Pi}: \quad M \longrightarrow M \neg a \text{ if } \begin{cases} a \in U(M, \Pi) \text{ and} \\ \neg a \text{ is unassigned by } M \end{cases}$$

$$\textit{Fail}_{\Pi}: \quad M \longrightarrow \bot \text{ if } \begin{cases} a \in U(M, \Pi), a \in M, \text{ and} \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_{\Pi}: \quad P\, l^{\Delta}\, Q \longrightarrow P\, \bar{l} \text{ if } \begin{cases} a \in U(P\, l\, Q, \Pi), a \in P\, l\, Q, \text{ and} \\ Q \text{ contains no decision literals} \end{cases}$$

Fig. 7: Transition rules of the graph AS_{Π}.

Proposition 3. *For every program Π, the graph $\mu(\mathrm{AS}_{\Pi})$ is a sound abstract module equivalent to a program Π under the answer set semantics. Furthermore, the graphs $\mathrm{AM}_{\mu(\mathrm{AS}_{\Pi})}$ and AS_{Π} are identical.*

From Theorem 1 and the fact that $\mu(\mathrm{AS}_{\Pi})$ is an abstract module equivalent to an answer-set program Π it follows that the graph AS_{Π} can be used for deciding whether Π has an answer set. It is enough to find a path in AS_{Π} leading from the node $\emptyset$ to a terminal node M. If $M = \bot$ then Π has no answer sets; otherwise, M is an answer set of Π.

Corollary 2. *For any program Π,*

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

Let Π be the program (2). Figure 3(b) presents the module $\mu(\mathrm{AS}_{\Pi})$. It is easy to see that this module is equivalently contained in the saturated module for Π presented in Figure 2. For program Π the inference rules of *UnitPropagate* and *Unfounded* are capable to capture all but one inference due to the entailment (the missing inference corresponds to the edge from b to $\neg a\ b$ in Figure 2).

Let us now consider the graph AS^{-}_{Π} constructed from AS_{Π} by either dropping the rules $\textit{Unfounded}_{\Pi}$, $\textit{Backtrack}_{\Pi}$, $\textit{Fail}_{\Pi}$ or the rules $\textit{UnitPropagate}_{\Pi^{cl}}$, $\textit{Backtrack}_{\Pi^{cl}}$, $\textit{Fail}_{\Pi^{cl}}$. In each case, the module $\mu(\mathrm{AS}^{-}_{\Pi})$ in general is not equivalent to a program Π. This demonstrates the importance of two kinds of inferences for the case of logic programs: (i) those stemming from unit propagate and related to the fact that an answer set of a program is also its classical model; as well as (ii) those based on the concept of "unfoundedness" and related to the fact that every answer set of a program contains no unfounded sets. We note that forward chaining mentioned in earlier section is subsumed by unit propagate.

The graph AS_{Π} is inspired by the graph SM_{Π} introduced by Lierler [11] for specifying an answer set solver SMODELS [19]. The graph SM_{Π} extends AS_{Π} by two additional transition rules (inference rules or propagators): *All Rules Cancelled* and *Backchain True*. We chose to start the presentation with the graph AS_{Π} for its simplicity. We now recall the definition of SM_{Π} and illustrate how a similar result to Proposition 3 is applicable to it.

If B is a conjunction of literals then by $\overline{B}$ we understand the set of the complements of literals occurring in B.

The set of nodes of SM_Π consists of the states relative to the vocabulary of program Π. The edges of the graph SM_Π are specified by the transition rules of the graph AS_Π and the following rules:

$$\textit{All Rules Cancelled}: \quad M \longrightarrow M\ \neg a \ \text{ if } \begin{cases} \overline{B} \cap M \neq \emptyset \text{ for all } B \in \mathit{Bodies}(\Pi, a) \\ \neg a \text{ is unassigned by } M \end{cases}$$

$$\textit{Fail ARC}: \quad M \longrightarrow \bot \ \text{ if } \begin{cases} \overline{B} \cap M \neq \emptyset \text{ for all } B \in \mathit{Bodies}(\Pi, a), \\ a \in M,\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack ARC}: \quad P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l} \ \text{ if } \begin{cases} \overline{B} \cap M \neq \emptyset \text{ for all } B \in \mathit{Bodies}(\Pi, a), \\ a \in P\ l\ Q,\ Q \text{ contains no decision literals} \end{cases}$$

$$\textit{Backchain True}: \quad M \longrightarrow M\ l \ \text{ if } \begin{cases} a \leftarrow B \in \Pi,\ a \in M,\ l \in B \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in \mathit{Bodies}(\Pi, a) \setminus B \\ l \text{ is unassigned by } M \end{cases}$$

$$\textit{Fail BT}: \quad M \longrightarrow \bot \ \text{ if } \begin{cases} a \leftarrow B \in \Pi,\ a \in M,\ l \in B \\ \overline{B'} \cap M \neq \emptyset \text{ for all } B' \in \mathit{Bodies}(\Pi, a) \setminus B \\ l \in M,\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack BT}: \quad P\ l^{\Delta}\ Q \longrightarrow P\ \bar{l} \ \text{ if } \begin{cases} a \leftarrow B \in \Pi,\ a \in P\ l\ Q,\ l' \in B \\ \overline{B'} \cap P\ l\ Q \neq \emptyset \text{ for all } B' \in \mathit{Bodies}(\Pi, a) \setminus B \\ l' \in P\ l\ Q,\ Q \text{ contains no decision literals} \end{cases}$$

The graph SM_Π shares the important properties of the graph AS_Π. Indeed, Proposition 3 and Corollary 2 hold if one replaces AS_Π with SM_Π. Corollary 2 in this form was one of the main results stated in [11][4].

Let Π be the program (2). Figure 3(b) presents the module $\mu(\text{AS}_\Pi)$. The module $\mu(\text{SM}_\Pi)$ coincides with the saturated module for Π presented in Figure 2. For program Π, the inference rule *Backchain True* captures the inference that corresponds to the edge from b to $\neg a\ b$, which the transition rules of the graph AS_Π are incapable to capture.

The examples above show that the framework of abstract modules uniformly encompasses different logics. We illustrated this point by means of propositional logic and answer-set programming. Furthermore, it uniformly models diverse reasoning mechanisms (entailment and its proof theoretic specializations). The results also demonstrate that transition systems proposed earlier to represent and analyze SAT and ASP solvers are special cases of general transition systems for abstract modules introduced here.

[4] In [11], Lierler presented the SM_Π graph in a slightly different from: the states of that graph permitted inconsistent states of literals, which in turn allowed to unify the *Fail* and *Backtrack* transition rules for different propagators.

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

By capturing diverse logics in a single framework, abstract modules are well suited for studying modularity in declarative formalisms, and principles underlying solvers for modular declarative formalisms. 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 systems for modules generalize to the new formalism.

An *abstract modular system* (AMS) is a set of 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 $\sigma(\mathcal{A})$.

An interpretation I over $\sigma(\mathcal{A})$ (that is, a subset of $\sigma(\mathcal{A})$) is a *model* of $\mathcal{A}$, written $I \models \mathcal{A}$, if I is a model of every module $S \in \mathcal{A}$. An AMS $\mathcal{A}$ *entails* a formula φ (over the same vocabulary as $\mathcal{A}$), written $\mathcal{A} \models \varphi$, if for every model I of $\mathcal{A}$ we have $I \models \varphi$. We say that an AMS $\mathcal{A}$ is *sound* if every module $S \in \mathcal{A}$ is sound.

Let S_1 be a module presented in Figure 1(a) and S_2 be a module in Figure 3(b). The vocabulary of the AMS $\{S_1, S_2\}$ consists of the atoms a and b. It is easy to see that the interpretation $\{a, \neg b\}$ is its only model.

For a vocabulary σ and a set of literals M, by $M|_\sigma$ we denote the maximal subset of M consisting of literals over σ. For example, $\{\neg a, \neg b\}|_{\{a\}} = \{\neg a\}$.

Each AMS $\mathcal{A}$ determines its *transition system* $\text{AMS}_{\mathcal{A}}$. The set of nodes of $\text{AMS}_{\mathcal{A}}$ consists of the states relative to $\sigma(\mathcal{A})$. The transition rules of $\text{AMS}_{\mathcal{A}}$ comprise the rule *Decide* and the rules *Propagate*$_S$, *Fail*$_S$, and *Backtrack*$_S$, for all modules $S \in \mathcal{A}$. The latter three rules are modified to account for the vocabulary $\sigma(\mathcal{A})$ and are presented in Figure 8.

Propagate$_S$: $M \longrightarrow M\,l$ if S has an edge from $M|_{\sigma(S)}$ to $M\,l|_{\sigma(S)}$

Fail$_S$: $M \longrightarrow \bot$ if $\begin{cases} S \text{ has an edge from } M|_{\sigma(S)} \text{ to } \bot, \\ M \text{ contains no decision literals} \end{cases}$

Backtrack$_S$: $P\,l^{\Delta}\,Q \longrightarrow P\,\bar{l}$ if $\begin{cases} S \text{ has an edge from } P\,l\,Q|_{\sigma(S)} \text{ to } \bot, \\ Q \text{ contains no decision literals} \end{cases}$

Fig. 8: The transition rules of the graph $\text{AMS}_{\mathcal{A}}$.

Theorem 2. *For every sound AMS $\mathcal{A}$,*

(a) the graph $\text{AMS}_{\mathcal{A}}$ is finite and acyclic,
(b) any terminal state of $\text{AMS}_{\mathcal{A}}$ other than $\bot$ 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.

This theorem demonstrates that to decide a satisfiability of a sound AMS $\mathcal{A}$ it is sufficient to find a path leading from node $\emptyset$ to a terminal node. It provides a foundation for the development and analysis of solvers for modular systems.

For instance, let $\mathcal{A}$ be the AMS $\{S_1, S_2\}$. Below is a valid path in the transition graph $\text{AMS}_{\mathcal{A}}$ with every edge annotated by the corresponding transition rule:

$$\emptyset \stackrel{Decide}{\longrightarrow} \neg a^{\Delta} \stackrel{Propagate_{S_2}}{\longrightarrow} \neg\, a^{\Delta}\; b \stackrel{Backtrack_{S_1}}{\longrightarrow} a \stackrel{Decide}{\longrightarrow} a\; \neg b^{\Delta}.$$

The state $a\; \neg b^{\Delta}$ is terminal. Thus, Theorem 2 (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 (2) 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.

The results presented above imply, as special cases, earlier results on the logics PC(ID) and SM(ASP), and their solvers [14].

5 Learning in Solvers for AMSs.

Nieuwenhuis et al. [20, Section 2.4] defined the *DPLL System with Learning* graph to describe SAT solvers' learning, one of the crucial features of current SAT solvers responsible for rapid success in this area of automated reasoning. The approach of Nieuwenhuis, Oliveras, and Tinelli extends to our abstract setting. Specifically, the graph $\text{AMS}_{\mathcal{A}}$ can be extended with "learning transitions" to represent solvers for AMSs that incorporate learning.

The intuition behind learning in SAT is to allow new propagations by extending the original clause database as computation proceeds. These "learned" clauses provide new "immediate derivations" to a SAT solver by enabling additional applications of *UnitPropagate*. In the framework of abstract modules, immediate derivations are represented by edges. Adding edges to modules captures the idea of learning by supporting new propagations that the transition rule *Propagate* may take an advantage of. We now state these intuitions formally for the case of abstract modular systems.

Let S be a module and E a set of edges between nodes of S. By S^E we denote the module constructed by adding to S the edges in E. A set E of edges is S-*safe* if the module S^E is sound and equivalent to S. For an AMS $\mathcal{A}$ and a set of edges E over the vocabulary of $\mathcal{A}$, we define $\mathcal{A}^E = \{S^{E_{|S}} : S \in \mathcal{A}\}$ (where $E_{|S}$ is the set of those edges in E that connect nodes in S). We say that E is $\mathcal{A}$-*safe* if $\mathcal{A}$ and $\mathcal{A}^E$ are *equivalent*, and each module S^E in $\mathcal{A}^E$ is sound.

An *(augmented) state* relative to an AMS $\mathcal{A} = \{S_1, \ldots, S_n\}$ is either a distinguished state $\bot$ or a pair of the form $M||\Gamma_1, \ldots, \Gamma_n$ where M is an *ordered* consistent set M of literals over σ, some possibly annotated by Δ; and $\Gamma_1, \ldots, \Gamma_n$ are sets of edges between nodes of modules $S_1, \ldots, S_n$, respectively. Sometimes we denote $\Gamma_1, \ldots, \Gamma_n$ by $\mathcal{G}$. For any AMS $\mathcal{A} = \{S_1, \ldots, S_n\}$, we define a graph $\text{AMSL}_{\mathcal{A}}$. Its nodes are the augmented states relative to $\mathcal{A}$. The rule *Decide* of the $\text{AMS}_{\mathcal{A}}$ graph extends to $\text{AMSL}_{\mathcal{A}}$ as follows

$$\textit{Decide}: \quad M||\mathcal{G} \longrightarrow M\; l^{\Delta}||\mathcal{G} \;\text{ if }\; l \text{ is unassigned by } M.$$

Figure 9 presents the transition rules of $\text{AMSL}_{\mathcal{A}}$ that are specific to each module S_i in $\mathcal{A}$. We note that the set E of edges in the rule *Learn Local*$_{S_i}$ is required to consist of

edges that run between the nodes of S_i. The transition rule

$$\text{Learn Global:} \quad M||\ldots,\Gamma_j,\ldots \;\longrightarrow\; M||\ldots,\Gamma_j \cup E_{|S_i},\ldots \text{ if } E \text{ is } \mathcal{A}\text{-safe}$$

where E is a set of edges between nodes over the vocabulary $\sigma(\mathcal{A})$, concludes the definition of $\text{AMSL}_{\mathcal{A}}$.

$$\textit{Propagate}_{S_i}: \quad M||\mathcal{G} \;\longrightarrow\; M\; l||\mathcal{G} \text{ if } S_i^{\Gamma_i} \text{ has an edge from } M \text{ to } M\; l$$

$$\textit{Fail}_{S_i}: \quad M||\mathcal{G} \;\longrightarrow\; \bot \text{ if} \begin{cases} S_i^{\Gamma_i} \text{ has an edge from } M \text{ to } \bot, \\ M \text{ contains no decision literals} \end{cases}$$

$$\textit{Backtrack}_{S_i}: \quad P\; l^{\Delta}\; Q||\mathcal{G} \longrightarrow P\; \bar{l}||\mathcal{G} \text{ if} \begin{cases} S_i^{\Gamma_i} \text{ has an edge from } P\; l\; Q \text{ to } \bot, \\ Q \text{ contains no decision literals} \end{cases}$$

$$\textit{Learn Local}_{S_i}: \quad M||\ldots,\Gamma_i,\ldots \;\longrightarrow\; M||\ldots,\Gamma_i \cup E,\ldots \text{ if } E \text{ is } S_i\text{-safe}$$

Fig. 9: Transition rules of $\text{AMSL}_{\mathcal{A}}$ for module $S_i \in S$.

We refer to the transition rules *Propagate*, *Backtrack*, *Decide*, and *Fail* of the graph $\text{AMSL}_{\mathcal{A}}$ as *basic*. We say that a node in the graph is *semi-terminal* if no basic rule is applicable to it. The graph $\text{AMSL}_{\mathcal{A}}$ can be used for deciding whether an AMS $\mathcal{A}$ has an answer set by constructing a path from $\emptyset||\emptyset,\ldots,\emptyset$ to a semi-terminal node.

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

(a) there is an integer m such that every path in $\text{AMSL}_{\mathcal{A}}$ contains at most m edges due to basic transition rules,
(b) for any semi-terminal state $M||\mathcal{G}$ of $\text{AMSL}_{\mathcal{A}}$ reachable from $\emptyset||\emptyset,\ldots,\emptyset$, M is a model of $\mathcal{A}$,
(c) state $\bot$ is reachable from $\emptyset||\emptyset,\ldots,\emptyset$ in $\text{AMSL}_{\mathcal{A}}$ if and only if $\mathcal{A}$ has no models.

It follows that if we are constructing a path starting in $\emptyset||\emptyset,\ldots,\emptyset$ in a way that guarantees that every sequence of consecutive edges of the path labeled with *Learn Local* and *Learn Global* eventually ends (is finite), then the path will reach some semi-terminal state. As soon as a semi-terminal state is reached the problem of finding a model is solved.

There is an important difference between *Learn Local* and *Learn Global*. The first one allows new propagations within a module but does not change its semantics as the models of the module stay the same (and it is local, other modules are unaffected by it). The application of *Learn Global* while preserving the overall semantics of the system may change the semantics of individual modules by eliminating some of their models (and, being global, affects in principle all modules of the system). SAT researchers have demonstrated that *Learn Local* is crucial for the success of SAT technology both in practice and theoretically. Our initial considerations suggest that under some circumstances, *Learn Global* offers additional substantial performance benefits.

We stress that our discussion of learning does not aim at any specific algorithmic ways in which one could perform learning. Instead, we formulate conditions that learned edges are to satisfy (S-safety for learning local to a module S, and $\mathcal{A}$-safety for the global learning rule), which ensure the correctness of solvers that implement learning so that to satisfy them. In this way, we provide a uniform framework for correctness proofs of multi-logic solvers incorporating learning.

6 Related Work

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 and hence contexts that rely on such abstract logics. The independence of contexts from syntax promoted focus on semantic aspect of modularity exhibited by 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]. Abstract modular systems introduced here are similar to MCSs as they too do not rely on any particular syntax for logics assumed in modules (a counterpart of a context). What distinguishes them is that they encapsulate some semantic features stemming from inferences allowed by the underlying logic. This feature of abstract modules is essential for our purposes as we utilize them as a tool for studying algorithmic aspects of multi-logic systems. Another difference between AMS and MCS is due to “bridge rules.” Bridge rules are crucial for defining the semantics of an MCS. They are also responsible for “information sharing” in MCSs. They are absent in our formalism altogether. In AMS information sharing is implemented by a simple notion of a shared 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 [9], and Tasharrofi and Ternovska [24] 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. [9] and Tasharrofi and Ternovska [24], we define such conjunction for any modules disregarding their internal structure and interdepencies between each other.

Tasharrofi, Wu, and Ternovska [25] developed and studied an algorithm for processing modular model expansion tasks in the abstract multi-logic system concept developed by Tasharrofi and Ternovska [24]. 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 [11], 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.

7 Conclusions

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. Learning, a fundamental technique in solver design, displays itself in two quite different flavors, local and global. The former corresponds to learning studied before in SAT and SMT and demonstrated both theoretically and practically to be essential for good performance. Global learning is a new concept that we identified in the context of modular systems. It concerns learning *across* modules and, as local learning, promises to lead to performance gains. In the future work we will conduct a systematic study of global learning in abstract modular systems and its impact on solvers for practical multi-logic formalisms.

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: 12th European Conference on Logics in Artificial Intelligence (JELIA). pp. 352–355 (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 (2005)
6. 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)

7. Giunchiglia, F.: Contextual reasoning. Epistemologia XVI, 345–364 (1993)
8. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19(20), 503–581 (1994)
9. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 155–168. LPNMR '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04238-6_15
10. Lee, J.: A model-theoretic counterpart of loop formulas. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). pp. 503–508. Professional Book Center (2005)
11. Lierler, Y.: Abstract answer set solvers. In: Proceedings of International Conference on Logic Programming (ICLP). pp. 377–391. Springer (2008)
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. Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue 11, issue 4-5 (2011)
15. 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 Verlag (1999)
16. Mariën, M., Wittocx, J., Denecker, M., Bruynooghe, M.: SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In: SAT. pp. 211–224 (2008)
17. McCarthy, J.: Generality in Artificial Intelligence. Communications of the ACM 30(12), 1030–1035 (1987), reproduced in [?]
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. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138, 181–234 (2002)
24. Tasharrofi, S., Ternovska, E.: A semantic account for modularity in multi-language modelling of search problems. In: Frontiers of Combining Systems, 8th International Symposium (FroCoS). pp. 259–274 (2011)
25. Tasharrofi, S., Wu, X.N., Ternovska, E.: Solving modular model expansion tasks. CoRR abs/1109.0583 (2011)
26. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of ACM 38(3), 620–650 (1991)