University of Nebraska at Omaha
DigitalCommons@UNO

Computer Science Faculty Publications | Department of Computer Science

7-2016

On Abstract Modular Inference Systems and Solvers

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

Miroslaw Truszczyński
University of Kentucky

Follow this and additional works at: http://digitalcommons.unomaha.edu/compscifacpub

Part of the Computer Sciences Commons

Recommended Citation
Lierler, Yuliya and Truszczyński, Miroslaw, "On Abstract Modular Inference Systems and Solvers" (2016). *Computer Science Faculty Publications.* Paper 22.
http://digitalcommons.unomaha.edu/compscifacpub/22

This Article is brought to you for free and open access by the Department of Computer Science at DigitalCommons@UNO. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

Artificial Intelligence 236 (2016) 65–89

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

On abstract modular inference systems and solvers ☆

Yuliya Lierler [a,*], Miroslaw Truszczynski [b,*]

[a] Department of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182, USA
[b] Department of Computer Science, University of Kentucky, Lexington, KY 40506-0633, USA

ARTICLE INFO

Article history:
Received 5 August 2014
Received in revised form 10 March 2016
Accepted 17 March 2016
Available online 29 March 2016

Keywords:
Knowledge representation
Model-generation
Automated reasoning and inference
SAT solving
Answer set programming

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. These graphs lend themselves well to extensions that capture such important solver design features as learning. In the paper, we consider two flavors of learning for modular formalisms, local and global. We illustrate our approach by showing how it applies to answer set programming, propositional logic, multi-logic systems based on these two formalisms and, more generally, to satisfiability modulo theories.

© 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Knowledge representation and reasoning (KR&R) 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 into 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&R 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. An early example of a successful integration of different types of reasoning is constraint logic programming (CLP) [28,29], which exploited computational properties of different theories of constraints in a formalism centered around logic programming. About two decades later a similar idea appeared in the area of propositional satisfiability. The resulting approach, known as satisfiability modulo theories (SMT) [49,4], consists of integrating diverse constraint theories around the "core" provided by propositional satisfiability. SMT solvers are currently among the most efficient automated reasoning tools and are widely used for computer-aided software verification [10]. Another, more recent example is constraint answer set programming (CASP) [45,20,2,31,35] that integrates answer set

☆ This paper is a substantially extended version of the paper presented at PADL 2014 [38].

* Corresponding authors.
E-mail addresses: ylierler@unomaha.edu (Y. Lierler), mirek@cs.uky.edu (M. Truszczynski).

http://dx.doi.org/10.1016/j.artint.2016.03.004
0004-3702/© 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

programming (ASP) [42,47] with constraint modeling and solving [51]. These approaches do not impose any strong *a priori* restrictions on the constraint theories they allow. However, some types of theories are particularly heavily studied (for instance, equality with uninterpreted functions, forms of arithmetic, arrays). Finally, more focused hybrid systems that combine modules expressed in classical logic with modules given as answer set programs have also received substantial attention lately. Examples include the "multi-logics" PC(ID) [43], SM(ASP) [36] and ASP-FO [11].[1] These multi-logic modular integrations facilitate modeling but also often lead to enormous performance gains. A good example is the problem of existence of Hamiltonian cycles in graphs. Known propositional logic encodings require that counter variables be used to represent reachability. That leads to representations of large sizes. Using propositional logic to represent non-recursive constraints and logic programs to represent reachability (which is much more direct than a counter-based propositional encoding) leads to concise encodings. East and Truszczynski [13] demonstrated that the performance of SAT solvers on propositional encodings of the problem lags dramatically behind that of the solver *aspps*, designed for handling together propositional and logic program modules on hybrid representations of the problem.[2] The "computational" motivation behind modular KR&R underlies our paper.

The key computational task arising in KR&R 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&R applications. Accordingly, model generation is of critical importance in modular multi-logic systems. Research on formalisms listed above resulted in fast solvers that demonstrate substantial gains that 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 representing modular multi-logic systems that aims to provide a general abstract view on solvers, to bring up key principles behind solver design, and to facilitate studies of their properties. As we are not concerned with the modeling aspect of a KR&R system but with solving, we design our language so that it (i) abstracts away the syntactic details, (ii) can capture diverse concepts of inference, and (iii) is based only on the weakest assumptions concerning the semantics of underlying logics, in particular, this language can capture any formalism whose semantics is determined by a set of models. The basic elements of this language are *abstract inference modules* (or just *modules*) that are defined to consist of *inferences*. 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 to capture inference mechanisms from different logics, 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 [49] 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. 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). 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. The design of transition systems based on *syntax-free* modules is what separates this work from earlier uses of graphs for describing model generation algorithms behind SAT, SMT, PC(ID), or ASP solvers [49,43,34,36]. These earlier transition graphs are language specific and based on the syntactic constructs typical of the respective formalisms. Adding a new level of abstraction allows one a bird's eye view on the landscape of solving techniques and their usage in hybrid settings.

To demonstrate the power of our approach, we show that it applies to answer set programming, propositional logic, multi-logic systems based on these two formalisms, and generally to satisfiability modulo theories. As SMT is a general framework for integrating diverse logics, the same expressivity claims hold true for our approach. However, in at least one aspect, our approach goes beyond the basic tenants of SMT. Namely, our modular systems have no central core, the role played by SAT in the case of SMT. Rather, all modules are viewed in exactly the same way and can pass on results of inferences directly to each other. In addition, all modules are presented in a uniform way as sets of inferences. In this way, we can ignore syntactical aspects of logics. Of course, that makes our formalism poorly tailored for modeling, as it is the syntax of logics that is typically used to provide concise representations of knowledge. Yet, our syntax-free modules make explicit the reasoning the logics of the modules support, and that is of central importance to our objective to support the design and analysis of solvers.

The paper is organized as follows. We start by introducing abstract inference modules. We then adapt the transition graphs of Nieuwenhuis et al. [49] to the formalism of abstract inference modules and use them to describe algorithms for finding and enumerating 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

[1] Logic PC(ID) is a propositional fragment of classical first-order logic with inductive definitions; SM(ASP) is a propositional language that merges classical logic expressions and logic programs under stable model semantics; ASP(FO) is a first-order language, which encapsulates modules stemming from classical logic and modules stemming from logic programming.

[2] We stress that in this discussion we simply aim at showing that combining modules coming from different logics may be beneficial. Here, limitations of propositional logic in modeling recursive constraints are overcome with modules designed specifically for this task. However, it is possible (in fact, straightforward) to design a single ASP program that efficiently handles the Hamiltonian cycle application.

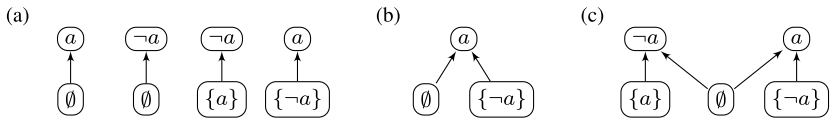


Fig. 1. All inferences and two inference modules over the vocabulary $\{a\}$.

for models in this setting, too. We also show in that section that abstract modular systems can be used to represent SMT. In Section 5, we discuss extensions to our framework that support representing model-finding algorithms exploiting "inference" learning, an abstract version of clause learning developed and studied in SAT. Throughout the paper, we illustrate our approach by showing how it applies to propositional logic, answer set programming, to multi-logic systems based on these two formalisms, and to SMT. We conclude by discussing related work, and recapping our contributions. All proofs are gathered in the appendix.

2. Abstract inference modules

We start with some notation. Let σ be a vocabulary (a set of propositional atoms). Elements of σ and their negations are *literals*. We write $Lit(\sigma)$ for the set of all literals over σ. For a literal l we define its *dual literal* $\bar{l}$ as $\neg a$, if $l = a$, and a, if $l = \neg a$. For a set $M \subseteq Lit(\sigma)$, we define $M^+ = \sigma \cap M$ and $M^- = \{a \in \sigma : \neg a \in M\}$. 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 $\bar{l}$. A set M of literals over σ is *consistent* if for every literal $l \in Lit(\sigma)$, $l \notin M$ or $\bar{l} \notin M$. We denote the set of all consistent subsets of $Lit(\sigma)$ by $C(\sigma)$. A set M of literals is *complete* over σ if for every atom $a \in \sigma$, either $a \in M$ or $\neg a \in M$.

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, σ_S denotes the set of all atoms that appear (possibly negated) 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 $\bar{l} \in M$, the inference is an explicit indication of a contradiction. Fig. 1(a) shows all inferences over the vocabulary $\{a\}$. Figs. 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.

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

Let S be an abstract module over some vocabulary σ. Clearly, S can also be viewed as a module over the vocabulary σ_S, as any inference of S is an inference constructed of literals in $Lit(\sigma_S)$. Moreover, it is clear from the definitions that for a module S viewed as a module over σ_S, a set X is a model of S if and only if $X \cap \sigma_S$ is a model of S. Thus, the semantics of a module S is fully determined by its models contained in σ_S. Following the same argument, we can view a module S over a vocabulary σ as a module over any vocabulary $\sigma' \supseteq \sigma$. In this respect, modules behave exactly as formulas and theories in classical logic.

Two modules (not necessarily over the same vocabulary) that have the same models are *equivalent*. Similarly to the observation made above, the following proposition shows that to test equivalence of modules one may restrict attention only to models consisting of atoms that occur in the modules in question.

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 their models. Let S be a module over a vocabulary σ and l a literal in $Lit(\sigma)$. We say that S *entails* l, written $S \mid\!\approx l$, 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 $S \mid\!\approx_M l$, 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, which we formally state below.

Proposition 2. *Let S be a module and (M, l) an inference in S. Then $S \mid\!\approx_M l$.*

In the paper, we often consider unions of (finitely many) modules. The union of modules is a well-defined operation as modules are sets (of inferences). Thus, the union of modules $M_1, \ldots, M_n$ is simply the module that consists precisely of all the inferences of $M_1, \ldots, M_n$. We use the symbol $\cup$ to denote the union of modules. The resulting module can be viewed as a module over *any* vocabulary σ that contains the vocabularies of all modules in the union.

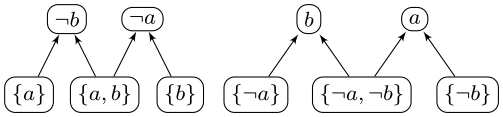


Fig. 2. Abstract module $Ent(T)$ for the theory T given by (1).

Proposition 3. *Let S_1 and S_2 be abstract inference modules. A set X 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, logic languages align closely with the 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 for *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 [23,42,47] (where logic programs are used as theories).

2.1. Propositional logic via abstract inference modules

Let T be a finite propositional theory (formula) over σ, and let σ_T be the set of atoms that 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 $T \cup M \models l$. Fig. 1(b) shows the module $Ent(\{a\})$. Fig. 1(c) shows the module $Ent(\{a \wedge \neg a\})$. Similarly, Fig. 2 presents the module $Ent(T)$, where T is the theory in conjunctive normal form (CNF theory)[3]:

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

We note that $Ent(T)$ has two models contained in $\{a, b\}$: $\{a\}$ and $\{b\}$.[4] More generally, every model X of $Ent(T)$ contains exactly one of a and b.

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. Let T be a finite propositional CNF theory over σ. 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 CNF theory (1). The module $Ent(T)$ in Fig. 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 Fig. 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.*

2.2. Answer set programming via abstract inference modules

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 [48] exploits four inference rules called 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.

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, $0 \le i \le m$, 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.

[3] We follow a common convention and represent CNF theories as sets of clauses.

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

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.

The concept of an *answer set* (stable model) was introduced in [22]. Saccà and Zaniolo [52] showed that answer sets can be characterized in terms of *unfounded sets* [56]. This characterization is the one we present here as it is especially useful in understanding inference rules of modern answer set solvers discussed here. A set U of atoms occurring in a program Π is *unfounded* on a consistent set M of literals with respect to Π if for every atom $a \in U$ and every $B \in Bodies(\Pi, a)$, there is $u \in s(B)$ such that $\overline{u} \in M$ or $U \cap s(B)^+ \neq \emptyset$. For a program Π over σ, a set X of atoms over σ is an *answer set* of Π if and only if X is a model of Π^{cl} and X contains no element of a set that is unfounded on $X \cup \{\neg a \colon a \in \sigma \setminus X\}$ with respect to Π. For a set M of literals and a program Π, we write $Unf(M, \Pi)$ to denote the family of all sets unfounded on M w.r.t. Π.

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 $a \in U$, for some $U \in Unf(M, \Pi)$;

All Rules 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 $B' \in Bodies(\Pi, a)$ such that $s(B') \neq s(B)$, there is $u \in s(B')$ such that $\overline{u} \in M$.

The four rules above give rise to abstract inference modules $UP(\Pi)$, $UF(\Pi)$, $ARC(\Pi)$ and $BC(\Pi)$, respectively, each obtained by taking the definition of the corresponding rule as the condition for (M, l), where $M \in C(\sigma_\Pi)$ and $l \in Lit(\sigma_\Pi) \setminus M$, to be an inference of the module. For instance, the module $UF(\Pi)$ consists of all inferences (M, l) such that $M \in C(\sigma_\Pi)$, $l \in Lit(\sigma_\Pi) \setminus M$, and $l = \neg a$, where a is any atom such that $a \in U$, for some $U \in Unf(M, \Pi)$.

We note that the inference rule *All Rules 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 also note that $UP(\Pi)$ and $UP(\Pi^{cl})$ are identical (and so, equivalent) even though they concern different logics.

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.[5] 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

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

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 Rules Cancelled* is computationally cheaper than the rule *Unfounded* and in practical implementations the two are distinguished.

The following result restates well-known properties of these inference rules 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

$$\begin{aligned} a &\leftarrow not\ b \\ b &\leftarrow not\ a. \end{aligned} \tag{4}$$

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 Fig. 2, the program and the module are equivalent. This module represents the program (4) and the reasoning mechanism captured by the module $smodels(\Pi)$. Two other modules associated with program (4) are given in Fig. 3. Fig. 3(a) shows the module $UP(\Pi)$, 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). Fig. 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 Fig. 3 captures all four inference rules and is indeed equal to the module in Fig. 2.

[5] 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.

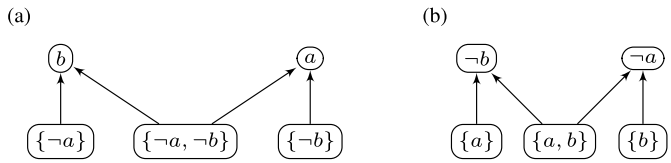


Fig. 3. Two abstract modules based on program (4).

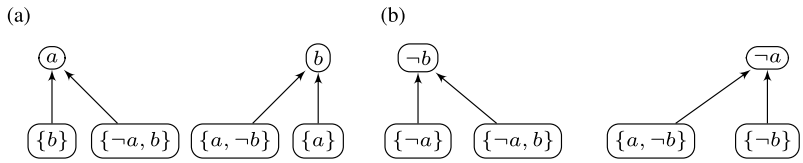


Fig. 4. Two abstract modules based on program (5).

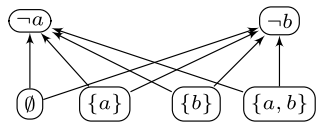


Fig. 5. An abstract module based on program (5).

We conclude this section with two more examples. In the first example, let Π be the program consisting of the rule

$$a \leftarrow not\ a.$$

This program has no answer sets. The module $UP(\Pi)$ consists of a single inference $(\{\neg a\}, a)$, whereas the modules $UF(\Pi)$, $ARC(\Pi)$, and $BC(\Pi)$ coincide and consist of a single inference $(\{a\}, \neg a)$. The module resulting from the union of two last mentioned inferences is unsatisfiable.

In the second example, we assume that Π is given by the rules

$$\begin{aligned} a &\leftarrow b \\ b &\leftarrow a. \end{aligned} \tag{5}$$

The empty set is the only answer set of this program. The inferences in Fig. 4(a) form the module $BC(\Pi)$ and those in Fig. 4(b) define the module $ARC(\Pi)$. The union of these two modules yields the module $UP(\Pi)$. The union of the inferences in Fig. 4(b) and in Fig. 5 gives the module $UF(\Pi)$.

3. Transition graphs — an abstraction of model finding algorithms

Finding models of logic theories and programs is a key computational task in declarative programming. Nieuwenhuis et al. [49] 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 conflict-driven clause learning. Lierler and Truszczynski [34,36] proposed a similar framework to describe and analyze the answer set solvers SMODELS, CMODELS [25] and CLASP [17,19], as well as a PC(ID) solver MINISAT(ID) [43]. 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 σ be a *finite* vocabulary. A *state over* σ is either a special state $\bot$ (the *fail* state) or a *sequence* M of *distinct* literals over σ, 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 $\bar{l}$, and
2. if $M = M'l^{\Delta}M''$, then l is unassigned in the set of the literals in M'.

For instance, if $\sigma = \{a, b\}$, then $\emptyset$, a, $\neg a^{\Delta}\, b$, $\neg a\, b^{\Delta}\, a$ and $\bot$ are examples of states over σ.

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

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

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

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

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

Fig. 6. The transition rules of the graph AM_S.

annotated by Δ denotes a current assumption: thus once a literal is assigned in a state, there is no point in 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 possible states relative to σ_S. The edges of the graph AM_S are specified by the *transition rules* listed in Fig. 6. 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 their notation. Moreover, even for the rule *Inference Propagate*, we often omit the reference to the module if it is implied by the context, or if the specific reference is immaterial. Finally, we call a node in a transition graph *terminal* if no edge originates in it (equivalently, no rule applies to it).

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

Theorem 6. *For every abstract inference 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 the module in Fig. 2. Below we show a path in the transition graph AM_S with every edge annotated by the corresponding transition rule:

$$\emptyset \ \xrightarrow{\textit{Decide}}\ b^{\Delta} \ \xrightarrow{\textit{Inference Propagate}_S}\ b^{\Delta}\ \neg a. \tag{6}$$

The state $b^{\Delta}\ \neg a$ is terminal. Thus, Theorem 6(b) asserts that $\{b\}$ is a model of S. There may be several paths determining the same model. For instance, the path

$$\emptyset \ \xrightarrow{\textit{Decide}}\ \neg a^{\Delta} \ \xrightarrow{\textit{Decide}}\ \neg a^{\Delta}\ b^{\Delta} \tag{7}$$

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 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 the "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 *Inference 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 a *Decide* edge.

This is still not a complete specification of a solver, as it offers no directions on how to select a decision literal (which of many possible *Decide* transitions to apply). Much of research on SAT solvers design 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 between *Inference Propagate* and *Decide* rules. Earlier, we selected an *Inference Propagate* edge and only if impossible, we selected a *Decide* edge. But that order can be reversed resulting in another class of algorithms. Finally, we could even consider more complicated selection functions that, when both *Decide* and *Inference Propagate* edges are available, in some cases select an *Inference Propagate* edge and in others a *Decide* one.

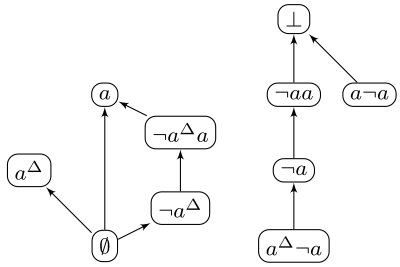


Fig. 7. The DP_F graph where $F = a$.

Before we proceed, we state several observations about the graph AM_S. First, for every model X of an abstract module S, there is a terminal state M in AM_S such that $X \cap \sigma_S = [M]^+$ and M is reachable from $\emptyset$ in the graph AM_S (in other words, every model of S is represented by some terminal state reachable from $\emptyset$ in AM_S). Indeed, we can take for M any state that contains annotated atoms x^Δ, for all $x \in X \cap \sigma_S$, and annotated negated atoms $\neg y^\Delta$, for all $y \in \sigma_S \setminus X$. Each such state M is reachable by a path whose edges are determined by the *Decide* rule. Moreover, if X is a model of S, then M is clearly a terminal state.

Second, generally each model of a module S is represented by many terminal states in the graph AM_S. Some are reachable from $\emptyset$ and some are not. However, as we just argued, the terminal states reachable from $\emptyset$ represent *all* models of the module. Thus, to decide satisfiability of a module (or, for satisfiable modules, to find a model) it is sufficient to consider only the states reachable from $\emptyset$. In this sense, the states reachable from $\emptyset$ determine the "essential" fragment of the transition graph. To illustrate these observations, let us consider the module $UP(\{a \vee b\})$ (which simulates Unit Propagate inferences based on $a \vee b$). Then, $a\,b$, $a^\Delta\,b$, $a\,b^\Delta$, and $a^\Delta\,b^\Delta$ all represent the same family of models — those that contain atoms a and b.[6] However, only one of these four states, $a^\Delta\,b^\Delta$, is reachable from $\emptyset$. The other three are not.

Third, one can generalize part (c) of Theorem 6 as follows. Let M be any state other than $\bot$. If a terminal state other than $\bot$ is reachable from M, then S has models that are consistent with $[M]$. Otherwise, $\bot$ is the only terminal state reachable from M and S has no models consistent with $[M]$ (but it may have other models). Thus, while only the fragment of the graph AM_S consisting of the states reachable from $\emptyset$ is needed to determine satisfiability and find models of the module, other parts of the graph are of interest too. Incidentally, similar generalizations are possible for parts (c) in Theorems 11, 16 and 17 that we state later.

Finally, we observe that the fragment of the graph AM_S reachable from $\emptyset$ contains inconsistent states also. Such states are, however, not terminal. For instance, the state $\neg a^\Delta\,\neg b^\Delta\,a$ is reachable from $\emptyset$ in the transition graph of the module $UP(\{a \vee b\})$ (the path is given by the edges determined by two applications of the *Decide* rule, followed by an application of the *Unit Propagate* rule). This state is not terminal as the *Backtrack* rule applies. Similarly, the fragment consisting of states that are not reachable may contain states that are consistent (as we mentioned above, the state $a\,b$ is not reachable from $\emptyset$ yet, it is consistent).

3.1. Abstract SAT solvers

We now show how the approaches proposed by Nieuwenhuis et al. [49] and Lierler [34] 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. [49], 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 vertexes 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. Fig. 7 presents DP_F.

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

Proposition 7. *For every CNF formula F with no empty clause,* $\text{DP}_F = \text{AM}_{UP(F)}$.

Theorem 6, Proposition 7, and the fact that a CNF formula F and the module $UP(F)$ are equivalent (Proposition 4) imply the following result.

[6] Four more states, with b preceding a, also represent this family of models.

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

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

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

Fig. 8. Transition rules of the graph SM_Π.

Corollary 8. *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. [49] 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. In our example, 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 algorithms 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.

3.2. Abstract answer set solvers

Our abstract approach to model generation in logics also applies to answer set programming [23,42,47]. Lierler [34] 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 $\text{DP}_{\Pi^{cl}}$ and the rules presented in Fig. 8.

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

Proposition 9. *For every logic program Π, $\text{SM}_\Pi = \text{AM}_{smodels(\Pi)}$.*

Indeed, this proposition, Theorem 6 and the fact that Π is equivalent to the module $smodels(\Pi)$ (Proposition 5) imply the result stemming from that of Lierler [34].

Corollary 10. *For every logic program Π,*

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

Since $UPUF(\Pi)$ is also equivalent to Π, we obtain a similar corollary for the transition graph $\text{AM}_{UPUF(\Pi)}$. Intuitively, this graph is characterized by the transition rules of the graph $\text{DP}_{\Pi^{cl}}$ as well as the rule *Unfounded* presented in Fig. 8. Thus, $\text{AM}_{UPUF(\Pi)}$ is an abstraction of another class of correct algorithms for finding answer sets of programs. In fact, it is so for any module S such that $UPUF(\Pi) \subseteq S \subseteq smodels(\Pi)$.

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 an edge is selected while in a consistent state.

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.

3.3. Model enumeration

We showed above how the transition graph AM_S can be used to conceptualize algorithms for deciding whether a module S has a model. Model enumeration is a related task of generating all models of a module S. Paper by Gebser et al. [18] is a good references for the problem. We now show that the transition graph approach can be adapted for the task of enumeration.

To account for model enumeration for a module S, we extend the graph AM_S to a graph AME_S. To this end, we introduce the transition rule

$$\textit{Enumerate}: \quad M \longrightarrow \begin{cases} P\,\bar{l} & \text{if no other rule applies to } M, \\ & M = P\,l^{\Delta}\,Q\text{, and } Q \text{ contains no decision literals} \\ \bot & \text{if no other rule applies to } M\text{, and} \\ & M \text{ contains no decision literals,} \end{cases}$$

and define the transition graph AME_S for an AM S as the graph AM_S extended with the transition rule *Enumerate*. The following theorem captures the main properties of this graph.

Theorem 11. *For every abstract inference module S,*

(a) the graph AME_S is finite and acyclic,
(b) the $\bot$ state is reachable from $\emptyset$,
(c) for every path from $\emptyset$ to $\bot$ in AME_S, the set of states in which the rule Enumerate applies is precisely the set of models of S over σ_S, and for each model X of S over σ_S there is exactly one state M on the path such that $X = [M]$.

This theorem assures us that if we follow a path from $\emptyset$ to $\bot$ we will encounter all models of S over σ_S.

Another related task is model counting where one wants to find the number of models of S, rather than what they are. Gomes at al. [27] provides a good account for the task. Since methods used for model counting aim to avoid explicit enumeration, it is not clear whether transition graphs can be useful for this task.

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 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) over a vocabulary σ is a finite set $\mathcal{A}$ of abstract inference modules over vocabularies contained in σ. A set X, is a *model* of $\mathcal{A}$ if X is a model of every module $S \in \mathcal{A}$.

For an abstract modular inference system $\mathcal{A}$, by $\sigma_{\mathcal{A}}$ we denote the vocabulary $\bigcup_{S \in \mathcal{A}} \sigma_S$. Recalling our comments from Section 2, we note that an AMS $\mathcal{A}$ can be viewed as a modular system over any vocabulary extending $\sigma_{\mathcal{A}}$.

Let S_1 be the module presented in Fig. 1(b) and S_2 be the module in Fig. 2. The vocabulary $\sigma_{\mathcal{A}}$ of an 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 $\mathcal{A}$ over $\sigma_{\mathcal{A}}$ (more generally, a set X is a model of $\mathcal{A}$ if and only if X contains a and does not contain b). In Section 2, we observed that (i) $S_1 = Ent(T)$ (and also $= UP(T)$) for a propositional theory $T = \{a\}$, and (ii) $S_2 = smodels(\Pi)$ for a program Π given by (4). Thus, the AMS $\mathcal{A} = \{S_1, S_2\}$ illustrates how abstract modular systems can serve as an abstraction for heterogeneous multi-logic systems.

4.1. Modular logic programs via abstract modular inference systems

For a general example of a modular declarative formalism that can be seen as an abstract modular system we now discuss modular logic programs [37]. Modular logic programs generalize the formalism of lp-modules, an early approach to modular answer set programming proposed by Oikarinen and Janhunen [50].

The semantics of modular logic programs relies on the notion of an input answer set of a program [36]. 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." For instance, program $a \leftarrow b$ has two input answer sets that are subsets of set $\{a, b\}$: namely, $\emptyset$ and set $\{a, b\}$.

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 $l = \neg a$ and, for some $U \in Unf(M, \Pi)$, $a \in U$ and for every $b \in U$, $b \in Head(\Pi)$ or $\neg b \in M$.

The only difference from the *Unfounded* rule we discussed earlier is a restriction on unfounded sets that the new rule imposes.

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

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

Proposition 12. *Every program Π is input-equivalent to the module $UPUF'(\Pi)$.*

A *modular (logic) program* is a set of logic programs [37]. For a modular program $\mathcal{P}$, a set X of atoms is a *model* 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 models of $\mathcal{P}$ coincide with models of $\mathcal{A}$.

Proposition 13. *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) [37] and PC(ID) [43] can be viewed as abstract modular systems in the same manner.

Remarks on modularity in ASP We use modular logic programs in this paper only to illustrate the key aspects of our general framework. Thus, it is not the place for an extended discussion of the relationship between modular logic programs and more standard work on modularity in ASP. Nevertheless, a few comments might be in order. First, in ASP the thrust is on identifying a *decomposition* of a program into subprograms ("modules") so that there is a direct correspondence between the answer sets of the subprograms and the answer sets of the program. Finding such decompositions (either explicitly by preprocessing, or implicitly during search) may have a big impact on the performance of solvers. Because of the nature of the answer set semantics, such decompositions are only possible for programs with some hierarchical structure given by stratification [1] or, more generally, splitting [41]. In this work, we use the generic term "model" of a modular system to stress that our semantics of modular logic programs is not directly related to the semantics of answer sets of the union of modules.[7] In fact, our motivation is quite different. We assume an "inverse" scenario where the modular structure is given right from the beginning. The objective is to define a semantics of a collection of programs based on the semantics of the individual programs. The semantics of answer sets does not lend itself naturally to this purpose. For instance, consider a modular program

$$\{\{a \leftarrow\ \},\ \{b \leftarrow\ \}\} \tag{8}$$

There is no set of atoms that yields an answer set to both logic programs $\{a \leftarrow\ \}$ and $\{b \leftarrow\ \}$ simultaneously. Therefore, in our earlier work we proposed the input answer set semantics as the semantics for individual logic programs, that lends a way to defining a meaningful semantics of modular logic programs. In this example, the set $\{a, b\}$ of atoms is a model of modular logic program (8), as it is an input answer set of each of the component programs. For another example, let as consider a modular program

$$\{\{a \leftarrow b\},\ \{b \leftarrow a\}\}. \tag{9}$$

Clearly, the union of the rules in the modules of (9) is exactly the program given by (5). Each of the programs $\{a \leftarrow b\}$ and $\{b \leftarrow a\}$ has only one answer set, $\emptyset$. However, the modular program has two models $\emptyset$ and $\{a, b\}$. It is so because both sets are *input* answer sets of each module, even though only the first one is an answer set. To summarize, an important point behind input answer set semantics is that it allows us to avoid difficulties that arise in ASP in the context of substitutability of one subprogram by another [40] or when attempting to determine the answer sets of the union of programs [32].

4.2. Satisfiability modulo theories via abstract modular inference systems

Satisfiability modulo theories (SMT) [49,4] is a general, broadly used framework for integrating diverse logics. In this section we review the concept of SMT programs and illustrate how these programs can be seen as abstract modular systems presented in this paper.

We start by introducing the notion of a theory in SMT. A *signature* Σ is a set of predicate and function symbols, each with an associated nonnegative integer called *arity*. We call predicate symbols of arity 0 *propositional*. We call a signature

[7] We used the term "answer set" in our earlier work.

propositional if it only contains propositional symbols. (We note that elsewhere in the paper we refer to propositional signatures as vocabularies.) A *term* over Σ is either

- a function symbol of arity 0 from Σ, or
- an expression $f(t_1, \ldots, t_n)$, where f is a function symbol from Σ of arity $n > 0$ and $t_1, \ldots, t_n$ are terms over Σ.

An *atomic formula* is either

- a propositional symbol from Σ, or
- an expression $p(t_1, \ldots, t_n)$, where p is a predicate symbol from Σ of arity $n > 0$ and $t_1, \ldots, t_n$ are terms over Σ.

A *theory literal*, or *t-literal*, is either an atomic formula A or its negation $\neg A$. A *theory formula* (or a *t-formula*, for short) is a set of t-literals. An *interpretation for a signature* Σ (or a Σ*-interpretation* for short) is a pair I consisting of a non-empty set $|I|$, the *universe* of the interpretation, and a mapping $(\cdot)^I$ assigning

- to each function symbol f in Σ of arity 0, an element $f^I \in |I|$,
- to each function symbol f in Σ of arity $n > 0$, a total function $f^I : |I|^n \to |I|$,
- to each propositional symbol p in Σ, an element in $\{True, False\}$,
- to each predicate symbol p in Σ of arity $n > 0$, a total function $p^I : |I|^n \to \{True, False\}$.

Let I be a Σ-interpretation. We extend the mapping $(\cdot)^I$ to all terms over Σ by induction by setting for every function symbol f of arity $n > 0$ and every sequence $t_1, \ldots, t_n$ of terms

$$(f(t_1, \ldots, t_n))^I = f^I(t_1^I, \ldots, t_n^I),$$

where f^I is the function assigned to f by the interpretation I. Similarly, we extend the mapping $(\cdot)^I$ to all t-formulas over Σ. Namely, if ϕ is a t-literal $p(t_1, \ldots, t_n)$, we set

$$\phi^I = p^I(t_1^I, \ldots, t_n^I),$$

where p^I is the truth value function for p given by the interpretation I. Next, if ϕ is a t-literal $\neg A$, we set

$$\phi^I = (\neg A)^I = \begin{cases} True & \text{if } A^I = False, \\ False & \text{if } A^I = True \end{cases}$$

Finally, for a t-formula ϕ,

$$\phi^I = \begin{cases} True & \text{if for every t-literal } L \in \phi,\ L^I = True, \\ False & \text{otherwise} \end{cases}$$

When $\phi^I = True$ we say that the interpretation I satisfies ϕ.

For a signature Σ, a Σ*-theory* is a set of Σ-interpretations. We say that a t-formula ϕ over Σ is *satisfiable in a* Σ*-theory* Υ (or is Υ*-satisfiable*, for short) if there is an element of the set Υ that satisfies ϕ.

Clearly, t-formulas can be regarded simply as classical ground formulas with negation and conjunction as the only connectives allowed. Further, the semantics we introduced above is just the classical first-order logic semantics of such formulas. In the literature on SMT, a more sophisticated syntax of formulas is considered. Yet, SMT solvers often use so-called propositional abstractions of first-order formulas which, in their most commonly used case, are t-formulas of the kind discussed here [49, Section 3.1].

For a signature Σ, a disjoint propositional signature σ, and a Σ-theory Υ, a $[\Sigma, \sigma, \Upsilon]$*-abstraction* is a mapping from atomic formulas over Σ to σ. For a $[\Sigma, \sigma, \Upsilon]$-abstraction λ, a set M of propositional literals over the signature σ is a *model of* λ *(or a* λ*-model)* if a t-formula

$$\{A \mid \lambda(A) \in M\} \cup \{\neg A \mid \neg\lambda(A) \in M\}$$

is satisfiable in Σ-theory Υ. Clearly, λ-models are consistent sets of literals over σ.

An *SMT program* is a tuple $\langle T, \lambda_1, \ldots, \lambda_n \rangle$, where T is a propositional CNF formula that contains no empty clauses, and every λ_i, $1 \le i \le n$, is a $[\Sigma_i, \sigma_T, \Upsilon_i]$-abstraction. Recall that by σ_T we denote the set of atoms that appear in T. A consistent and complete set M of literals over σ_T is a *model* of an SMT program $\langle T, \lambda_1, \ldots, \lambda_n \rangle$ if M^+ is a model of T and M is a λ_i-model, for every i, $1 \le i \le n$.

We will now construct several abstract module systems that are equivalent to an SMT program $\langle T, \lambda_1, \ldots, \lambda_n \rangle$. The propositional formula T can be equivalently represented by modules $Ent(T)$ and $UP(T)$ introduced in Section 2.1. What remains is to construct modules to represent $[\Sigma_i, \sigma_T, \Upsilon_i]$-abstractions λ_i.

We first define the notion of *entailment* for $[\Sigma, \sigma, \Upsilon]$-abstractions. Let λ be a $[\Sigma, \sigma, \Upsilon]$-abstraction, $l \in \sigma$ a literal, and M a consistent set of literals over σ. We say that λ *entails* l w.r.t. M when for every consistent set M' of literals over σ that is

a superset of M it holds that if M' is a λ-model then $l \in M'$. We denote the fact that λ entails l w.r.t. M by $\lambda[M] \models l$. Note that if there is no single consistent set M' of literals over σ such that $M \subseteq M'$ and M' is a λ-model then, every literal $l \in \sigma$ is entailed by λ w.r.t. M.

For a $[\Sigma, \sigma, \Upsilon]$-abstraction λ, by $Ent(\lambda)$ we denote the module consisting of pairs (M, l) that satisfy the following conditions: M is a consistent set of literals over σ (in other words, $M \in C(\sigma)$), $l \in Lit(\sigma) \setminus M$, and $\lambda[M] \models l$.

For a $[\Sigma, \sigma, \Upsilon]$-abstraction λ, by $Min(\lambda)$ we denote the module consisting of pairs (M, l) that satisfy the following conditions: M is a consistent and complete set of literals over σ, $l \in Lit(\sigma) \setminus M$, and $\lambda[M] \models l$. The *Min* module differs from the *Ent* module by only including inferences (M, l), where M is complete in addition to being consistent. Thus, it serves the purpose of spotting that set M^+ is not a λ-model.

An AMS $\mathcal{A}$ and an SMT program $P = \langle T, \lambda_1, \ldots, \lambda_n \rangle$, are *equivalent* if for any consistent and complete set M of literals over σ_T, M is a model of P if and only if M^+ is a model of $\mathcal{A}$. We are now ready to state a formal result relating SMT programs and abstract modular inference systems composed of introduced modules.

Proposition 14. *Every SMT program $P = \langle T, \lambda_1, \ldots, \lambda_n \rangle$ is equivalent to any of the following abstract modular systems (over the vocabulary σ_T)*

1. $\{Ent(T), Ent(\lambda_1), \ldots, Ent(\lambda_n)\}$,
2. $\{UP(T), Ent(\lambda_1), \ldots, Ent(\lambda_n)\}$,
3. $\{Ent(T), Min(\lambda_1), \ldots, Min(\lambda_n)\}$,
4. $\{UP(T), Min(\lambda_1), \ldots, Min(\lambda_n)\}$.

It is interesting to note that replacing $Ent(\lambda_i)$ with $Min(\lambda_i)$ makes no difference for the semantics. However, the modular systems that result from such replacements capture different evaluation strategies of SMT solvers. In particular, the lazy evaluation strategy of SMT solvers [49] relies on the fact that the SMT program $P = \langle T, \lambda_1, \ldots, \lambda_n \rangle$ is equivalent to the fourth abstract modular system in the proposition above.

Constraint answer set programming [45,20,2,31,35] is another prominent multi-logic formalism. In SMT, theories are integrated with a propositional formula. In constraint answer set programming, theories (called constraints) are integrated with logic programs. A similar argument can be used to show that AMSs capture constraint answer set programs.

4.3. Abstract AMS solver

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. We say that an AMS $\mathcal{A}$ is *equivalent* to an abstract inference module S if $\mathcal{A}$ and S have the same models.

Theorem 15. *Every abstract modular inference system $\mathcal{A}$ is equivalent to the abstract inference module $\mathcal{A}^{\cup}$.*

This theorem shows the value of our abstraction. Concrete modular systems composed from theories of different logics cannot be easily combined into single theory. In particular, the operation of union cannot be applied since the result might not belong to any well-defined formal system. However, once all modules of the system are expressed as abstract modules, the problem disappears. The corresponding abstract modules can be combined and the union operator is the right one for the task.

We use Theorem 15 to define for each AMS $\mathcal{A}$ its *transition graph* $\textsc{ams}_{\mathcal{A}}$. Namely, we set $\textsc{ams}_{\mathcal{A}} = \textsc{am}_{\mathcal{A}^{\cup}}$. Theorem 6 implies the following result.

Theorem 16. *For every AMS $\mathcal{A}$,*

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

As in several other similar results before, Theorem 16 shows that the graph $\textsc{ams}_{\mathcal{A}}$ is an abstract representation of a class of algorithms to decide satisfiability of a modular system $\mathcal{A}$. An algorithm from the class searches for a path in $\textsc{ams}_{\mathcal{A}}$ that leads from node $\emptyset$ to a terminal node. In each step, it extends the path with a node reachable from the currently last node by some edge originating in it. Theorem 16(a) guarantees that the method terminates, the other two parts of that result ensure correctness.

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

$$\emptyset \xrightarrow{Decide} \neg a^{\Delta} \xrightarrow{Inference\ Propagate_{S_2}} \neg a^{\Delta}\ b \xrightarrow{Inference\ Propagate_{S_1}} \neg a^{\Delta}\ b\ a \xrightarrow{Backtrack} a \xrightarrow{Decide} a\ \neg b^{\Delta}.$$

The state $a\ \neg b^{\Delta}$ is terminal. Thus, Theorem 16(b) asserts that $\{a\}$ 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 all algorithms obtained by *integrating* algorithms represented by the modules S_1 and S_2, respectively.

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 modules in $\mathcal{A}$ are prioritized. Since modular systems do not assume any inherent priorities among modules, we assume the priorities are provided by the user as input (or control parameter) to the algorithm. 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 edge-selection strategy for the highest priority module.

Assuming that modules in $\mathcal{A}$ are enumerated $S_1, \ldots, S_k$ according to the descending priorities, 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 further 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 *Inference 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.

Our goal in this discussion is not to present a complete landscape of possible algorithmic instantiation of the graph $\text{AMS}_{\mathcal{A}}$ but simply to show an example of such an instantiation. Clearly, other possibilities exist. For instance, the preference order may be dynamic. That is, once a model M of modules $S_1, \ldots, S_i$ is found, the next module to drive the search might be selected based on M. We may also 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.

System DLVHEX Our results apply to a version of the DLVHEX[8] solver [15] 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.[9]

Model enumeration By Theorem 15, the problem of model enumeration for abstract modular systems can be reduced to the problem of model enumeration for a single module. Therefore, we do not discuss it here.

5. Learning in solvers for AMSs

Nieuwenhuis et al. [49, 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. Their approach

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

[9] A modular structure of a program was exploited by other solvers, too [8,19,53]. However, in those efforts modularity (represented by strongly connected components of the positive dependency graph) was used to improve the performance of a specific propagator, namely, the one based on unfounded sets. Our approach is not meant to model that level of granularity in solver's design.

$$\textit{Inference Propagate}_{S_i}: \quad M\|\mathcal{G} \longrightarrow Ml\|\mathcal{G} \quad \text{if} \begin{cases} [M] \text{ is consistent, } l \notin [M] \text{ and} \\ \text{for some } M' \subseteq [M], \\ (M', l) \text{ is an inference of } S_i^{\Gamma_i} \end{cases}$$

$$\textit{Learn Local}_{S_i}: \quad M\|\ldots,\Gamma_i,\ldots \longrightarrow M\|\ldots,\Gamma_i \cup E,\ldots \quad \text{if} \begin{cases} E \text{ is an } S_i\text{-safe set of} \\ \text{inferences over } \sigma_{S_i} \\ \text{such that } E \cup \Gamma_i \neq \Gamma_i \end{cases}$$

Fig. 9. The transition rules of $\textsc{amsl}_{\mathcal{A}}$ determined by a module $S_i \in \mathcal{A}$.

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

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

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

$$\textit{Learn Global}: \quad M\|\mathcal{G} \longrightarrow M\|\mathcal{G}^E \quad \text{if} \begin{cases} E \text{ is an } \mathcal{A}\text{-safe set of inferences over } \sigma_{\mathcal{A}} \\ \text{such that } \mathcal{G}^E \neq \mathcal{G} \end{cases}$$

Fig. 10. Global transition rules of $\textsc{amsl}_{\mathcal{A}}$.

extends to our abstract setting. Specifically, the graph $\textsc{ams}_{\mathcal{A}}$ 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 set of clauses as computation proceeds. These additional "learned" clauses give rise to new inferences to a SAT solver by enabling additional applications of *Unit Propagate*. In abstract modules, a similar effect can be obtained by extending them with new inferences (pairs (M, l)). These inferences give rise to new edges in the transition graph via the rule *Inference Propagate*. Thus, they can be seen as "shortcuts" in the original graph leading to shorter paths to a terminal state. We now state these intuitions formally for the case of abstract modular systems.

Let S be a module and E a set of inferences over σ_S. By S^E we denote the module constructed by adding to S the inferences in E. A set E of inferences over σ_S is *S-safe* if the module S^E is equivalent to S.

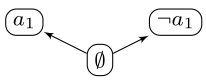
Let $\mathcal{A}$ be an AMS and E a set of inferences over $\sigma_{\mathcal{A}}$. For a module $S \in \mathcal{A}$, we define $E_{|S}$ to be the set of all inferences in E over the vocabulary σ_S, and set $\mathcal{A}^E = \{S^{E_{|S}} : S \in \mathcal{A}\}$. We say that E is *$\mathcal{A}$-safe* if $E = \bigcup_{S \in \mathcal{A}} E_{|S}$ (that is, if every inference in E is an inference over σ_S, for some $S \in \mathcal{A}$), and if the module $\mathcal{A}^E$ is equivalent to $\mathcal{A}$.

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 a state over the vocabulary $\sigma_{\mathcal{A}}$ and $\Gamma_1, \ldots, \Gamma_n$ is a sequence of sets of inferences over the vocabularies of the modules $S_1, \ldots, S_n$, respectively. Sometimes we denote the sequence $\Gamma_1, \ldots, \Gamma_n$ by $\mathcal{G}$. If E is a set of inferences over the vocabulary $\sigma_{\mathcal{A}}$ and $\mathcal{G} = \Gamma_1, \ldots, \Gamma_n$, we define $\mathcal{G}^E = \Gamma_1 \cup E_{|S_1}, \ldots, \Gamma_n \cup E_{|S_n}$.

Each AMS $\mathcal{A} = \{S_1, \ldots, S_n\}$ determines a graph $\textsc{amsl}_{\mathcal{A}}$. Its nodes are the augmented states relative to $\mathcal{A}$ and its transitions are specified in Figs. 9 and 10. The transitions in the first group are determined by individual modules, the transitions in the second group are "global."

To illustrate the rule *Learn Global*, consider an AMS consisting of two modules:

1. F is a module over the vocabulary $\{a_1, a_2\}$ with no inferences. (Every consistent and complete set of literals over $\{a_1, a_2\}$ is its model.)
2. S is a module over the vocabulary $\{a_1\}$ of the form:



The inferences $(\emptyset, a_1)$, $(\emptyset, \neg a_1)$, $(\{a_1\}, \neg a_1)$ and $(\{\neg a_1\}, a_1)$ are $\{F, S\}$-safe. Thus, we can apply the rule *Learn Global* with any subset of these inferences. This allows the future applications of *Inference Propagate*$_F$ to have access to these new inferences. Note that originally, *Inference Propagate*$_F$ has empty set of inferences to work with. Also, no $\{F\}$-safe inferences exist so that *Learn Local*$_F$ is inapplicable.

We refer to the transition rules *Inference Propagate*, *Backtrack*, *Decide*, and *Fail* of the graph $\textsc{amsl}_{\mathcal{A}}$ as *basic*. We say that a node in the graph is *semi-terminal* if no basic rule is applicable to it. The graph $\textsc{amsl}_{\mathcal{A}}$ can be used for deciding whether an AMS $\mathcal{A}$ has a model by constructing a path from $\emptyset\|\emptyset, \ldots, \emptyset$ to a semi-terminal node. We make this claim precise by stating the following theorem.

Theorem 17. *For every AMS $\mathcal{A}$,*

(a) the graph AMSL$_{\mathcal{A}}$ *is finite and acyclic,*
(b) for any semi-terminal state $M\|\mathcal{G}$ of 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 AMSL$_{\mathcal{A}}$ *if and only if $\mathcal{A}$ has no models.*

It follows that if we are constructing a path starting in $\emptyset\|\emptyset, \ldots, \emptyset$ then we will reach some semi-terminal state and at that point the task of finding a model of $\mathcal{A}$ is completed.

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 inferences 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. In this way, we provide a uniform framework for correctness proofs of multi-logic solvers incorporating learning.

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. Moreover, it is local, that is, 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. Moreover, being global, it 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 [46,44]. In fact, local (conflict-driven) learning has become standard not only in SAT solvers [26], but also in ASP solvers [19]. Nieuwenhuis et al. [49] described a transition rule *T-Learn* for SMT that can be seen as a precursor of the *Learn Global* rule. It is a standard practice in SMT solving to implement *T-Learn* [49]. Eiter et al. [14] implement *Learn Global* in the system DLVHEX and report that this significantly decreases the runtime of the system. We now present theoretical analysis showing that in some cases, *Learn Global* has a potential to yield substantial performance benefits for modular systems.

Let us consider a solver modeled within our abstract solving framework by imposing two restrictions. First, the transitions determined by the rule *Learn Global* are not allowed. We write AMSL$^-$ for the graph obtained from the corresponding graph AMSL by removing the edges corresponding to the application of the rule *Learn Global*. Second, we assume that the solver proceeds (applies the rules) according to the ranking given by an enumeration of modules in the input AMS $\mathcal{A}$, say $\mathcal{A} = \{S_1, \ldots, S_k\}$ (vide our discussion of solvers, and in particular of DLVHEX, in an earlier section). We will now show that a solver of this type can reap significant performance benefits by incorporating the rule *Learn Global*. To show that let us consider a family $\mathcal{A}_n = \{F_n, S\}$ $(n = 1, 2, \ldots)$ of AMSs, where:

1. F_n is a module over the vocabulary $\{a_1, \ldots, a_n\}$ with no inferences. In particular, every consistent and complete set of literals over $\{a_1, \ldots, a_n\}$ is its model.
2. S is a module as defined in the example prior to Theorem 17.

Assuming that the module F_n is higher ranked than the module S, solvers modeled by the graph AMSL$^-$ and the ranking-based rule application will start with the module F_n and by applying *Decide* to *all* its atoms, arrive at one of the models of F_n. Next, they will move the search to module S and in constant time discover a contradiction. Once the contradiction has been reached by means of rules in S, the algorithm backtracks to module F_n and generates another model of F_n. Then it moves on to S again, and again discovers a contradiction. The search terminates with failure only after *all* 2^n models of F_n have been inspected. Consequently, the search runs in time exponential in n.

There are two inferences that could be learned and added to S by means of the rule *Learn Local*: $(\{a_1\}, \neg a_1)$ and $(\{\neg a_1\}, a_1)$. Indeed, both are S-safe. However, incorporating these two inferences in the module S does not improve the performance of the solver. It will still try all models of F_n and fail only after all of them were considered. Thus, local learning does not help reduce the running time.

However, the inferences $(\emptyset, a_1)$, $(\emptyset, \neg a_1)$, $(\{a_1\}, \neg a_1)$ and $(\{\neg a_1\}, a_1)$ are $\{F_n, S\}$-safe. Applying the rule *Learn Global* with, say, $(\emptyset, a_1)$ and $(\{a_1\}, \neg a_1)$, and incorporating these two inferences into F_n allows the solver to immediately terminate the search (it will apply the inference $(\emptyset, a_1)$ followed by $(\{a_1\}, \neg a_1)$, and finally reach $\bot$ by applying the rule *Fail*). That search will run in time linear in n (essentially, the amount of time needed to reach the first conflict as all other steps take constant time). Thus, global learning results in an *exponential* speed up.

6. Related work and future research

In an important development, Brewka and Eiter [6] 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 has allowed researchers to focus on semantic aspects of multi-context systems. Since their inception, multi-context systems have received substantial attention and inspired implementations of hybrid reasoning systems including DLVHEX [15] and DMCS [16]. There are some similarities between AMSs and MCSs. However, there are also differences. First, MCSs provide an abstract framework to define seman-

tics of hybrid systems. In contrast, AMSs explicitly represent inferences of a logic and provide an abstract framework for studying model generation algorithms.

Second, the two formalisms differ in how they share information among modules. MCSs use to this end the so-called "bridge rules." In AMSs information sharing is implemented by a simple notion of sharing parts of the vocabulary between the modules. Rather non-surprisingly, bridge rules can simulate it. More interestingly, as our recent research on model-based abstract modular systems shows, despite its simplicity, information sharing through vocabulary sharing is expressive enough to capture the effects of bridge rules [39].

Modularity is one of the key techniques in principled software development. The importance of modularity has also been recognized in declarative programming languages rooted in KR&R such as answer set programming. In particular, Oikarinen and Janhunen [50] 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. Dao-Tran et al. [9] extend modularity to programs under the answer set semantics, whose models may have contextually dependent input provided by other modules. Janhunen [30] proposed the composition of hybrid reasoning systems using a general modular architecture that allows to combine propositional formulas as well as logic programs. Järvisalo, Oikarinen, Janhunen, and Niemelä [33], and Tasharrofi and Ternovska [54] studied different collections of operators to combine elementary abstract modules into more complex ones. We focused on building simple (flat-structured) modular systems that can be obtained from abstract modules by means of only one composition operator, the union (which implements the standard notion of the logical conjunction connective). In contrast to the work by Järvisalo et al. [33] and Tasharrofi and Ternovska [54], the conjunction (union) can be applied to any modules, no matter their internal structure and interdependencies between them. Whether our "union-based" modular systems can represent modular systems arising when other operators to combine modules are allowed is an interesting open question.

Tasharrofi, Wu, and Ternovska [55] proposed an algorithm for modular model expansion tasks, in particular, for the task of model generation, in the abstract multi-logic system setting developed by Tasharrofi and Ternovska [54]. They describe their algorithm by standard pseudocode and do not propose any abstract representations. Giunchiglia et al. [24] analyzed pseudocode descriptions of algorithms to study and relate several backtrack search procedures behind answer set solvers. In this work, we adapt an abstract 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. [49], Lierler [34], and Lierler and Truszczynski [36]. 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 [21] describe a form of a tableaux system to capture 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.

Brain [5] introduces the concept of *I-spaces* meant to uniformly capture states of computation of search algorithms stemming from different logical formalisms. This way search algorithms can be uniformly described and compared. Similarly, D'Silva, Haller and Kroening [12] introduce a *lattice-theoretic* generalization of several logic-based formalisms including propositional satisfiability. They show that a conflict-driven-clause-learning algorithm of modern satisfiability solvers can be considered and analyzed in lattice-theoretic terms. It is an interesting question whether I-spaces of Brain or the lattice-theoretic approach of D'Silva et al. could be used to study modularity of multi-logic systems.

Brochenin et al. [7] illustrated how the graph-based framework in spirit of Nieuwenhuis et al. can be lifted from capturing DPLL-like procedures to decision procedures at the second level of polynomial hierarchy. In the future we intend to investigate the applicability of the ideas by Brochenin et al. in the context of abstract modular inference systems and solvers.

Barrett et al. [3] proposed the transition system $DPLL(T_1, \ldots, T_n)$ that captures the following architecture of an SMT solver: DPLL-based SAT solver plays the role of the master system coordinating the search process of distinct specialized solvers for theories $T_1, \ldots, T_n$. The $\text{AMSL}_{\mathcal{A}}$ transition system can be seen as a generalization of the $DPLL(T_1, \ldots, T_n)$ framework that (a) removes SAT solving as the distinguished component of an SMT solver, and (b) is agnostic about which theory solver plays the role of the master system.

7. Conclusions

In this paper, 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 propositional theories and logic programs can be expressed as abstract inference modules, and that collections of propositional theories and logic programs can be represented as abstract modular systems. Even more importantly, we showed that satisfiability modulo theories can be translated to and studied in the language of abstract modular systems, too, thus demonstrating a broad scope of applicability of our formal framework.

Next, we showed that transition graphs determined by modules and modular systems provide an elegant and effective unifying representation of model generating algorithms, or solvers, and simplify reasoning about such issues as correctness or termination. Our discussion of inference learning identified two types of learning relevant to computing models of modular systems — local and global. The former corresponds to learning studied before in SAT and SMT and shown both theoretically and practically to be essential for good performance. The latter, the global learning, is a new concept that arises in the context of modular systems. It concerns learning *across* modules and, as local learning, promises to lead to performance gains. In the future, we will conduct a systematic study of global learning and its impact on solvers for practical multi-logic formalisms.

The paper provides evidence that abstract inference modules, abstract modular systems and their transition graphs can be useful in theoretical studies of solver properties, and in the development of solvers for modular systems that combine theories from different logic formalisms.

Acknowledgements

We are grateful to the reviewers for numerous comments that helped us to significantly improve the paper.

Appendix A. Proofs

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 $\sigma_{S_1} \cup \sigma_{S_2}$.*

Proof. ($\Rightarrow$) Evident.

($\Leftarrow$) Assume that S_1 and S_2 have the same models contained in the set $\sigma_{S_1} \cup \sigma_{S_2}$. To prove the implication, it suffices to show that if X is a model of S_1 then X is a model of S_2. To simplify the notation, we define $\delta = \sigma_{S_1} \cup \sigma_{S_2}$ and $X_\delta = X \cap \delta$.

Let (M, l) be an inference of S_1 such that M is consistent with X_δ. Since $M \subseteq Lit(\delta)$, M is consistent with X. It follows that l is consistent with X. Since $l \in Lit(\delta)$, l is consistent with X_δ. Thus, X_δ is a model of S_1. By the assumption, X_δ is a model of S_2.

Let (M', l') be an inference of S_2 such that M' is consistent with X. Since $M' \subseteq Lit(\delta)$, M' is consistent with X_δ. We recall that X_δ is a model of S_2. Thus, l' is consistent with X_δ and, since $l' \in Lit(\delta)$, also with X. It follows that X is a model of S_2. □

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

Proof. Let X be a model of S such that M is consistent with X. By the definition of a model of a module, l is consistent with X and the result follows. □

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

Proof. The assertion is an immediate consequence of definitions. Let X be a model of $S_1 \cup S_2$ and (M, l) be an inference of S_1 such that M is consistent with X. Since (M, l) is an inference of $S_1 \cup S_2$, l is consistent with X and so, X is a model of S_1. The case of (M, l) being an inference of S_2 and the converse implication can be proved in a similar way. □

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

Proof. We denote by σ_T the vocabulary that consists of atoms occurring in T (and $Ent(T)$).

Statement 1: For every propositional theory T, $Ent(T)$ is equivalent to T. Let X be a model of T and (M, l) an inference of $Ent(T)$ such that M is consistent with X. Clearly, X is a model of M. Since $T \cup M \models l$ and X is a model of $T \cup M$, X is model of l, that is, l is consistent with X. We derive that X is a model of $Ent(T)$.

Conversely, let X be a model of $Ent(T)$ and let us define $M = (X \cap \sigma_T) \cup \{\neg a : a \in \sigma_T \setminus X\}$. Clearly, M is consistent with X. We now proceed by contradiction. Assume that X is not a model of T. Then, $T \cup M$ is inconsistent. Let l be any literal in M and $M' = M \setminus \{l\}$. It follows that $T \cup M' \models \bar{l}$ (indeed, since $T \cup M$ is inconsistent, every model of $T \cup M'$, must be consistent with $\bar{l}$). By definition, $(M', \bar{l}) \in Ent(T)$. From the fact that M is consistent with X and $M' \subset M$, we derive that M' is consistent with X. Since X is a model of $Ent(T)$, $\bar{l}$ is consistent with X. On the other hand, $l \in M$ and M is consistent with X. Thus, l is consistent with X, a contradiction.

Statement 2: For every CNF formula T containing no empty clause, $UP(T)$ is equivalent to T. Let X be a model of T and (M, l) an inference of $UP(T)$ such that M is consistent with X. It follows that T has a clause $C \vee l$ such that for every literal u of C, $\overline{u} \in M$. Thus, all literals $\overline{u}$, $u \in C$, are consistent with X, that is, X is a model of $\neg C$. Since X is a model of T, X is a model of l, that is, l is consistent with X. We derive that X is a model of $UP(T)$.

Conversely, let X be a model of $UP(T)$. We proceed by contradiction. Assume that X is not a model of T. Then there is a clause C in T such that X is a model of $\neg C$. Let us assume that $C = u_1 \vee \ldots \vee u_k$. It follows that the set $\{\overline{u}_1, \ldots, \overline{u}_k\}$ is consistent with X. Since C is a clause of T and T contains no empty clause, $k \geq 1$. Moreover, since C is not a tautology (we recall that X is a model of $\neg C$), $u_k \notin \{\overline{u}_1, \ldots, \overline{u}_{k-1}\}$. By the definition of $UP(T)$, $(\{\overline{u}_1, \ldots, \overline{u}_{k-1}\}, u_k) \in UP(T)$. Since $\{\overline{u}_1, \ldots, \overline{u}_{k-1}\}$ is consistent with X and X is a model of $UP(T)$, u_k is consistent with X, a contradiction. □

We recall that if Π is a program, σ_Π denotes the vocabulary that consists of atoms occurring in Π. It is obvious that σ_Π also coincides with the set of atoms occurring in the modules $UP(\Pi)$, $UF(\Pi)$, $UPUF(\Pi)$ and $smodels(\Pi)$.

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

Proof. *Statement 1: Every logic program Π is equivalent to the module $UPUF(\Pi)$.* Let X be an answer set of Π. By definition, X is a model of Π^{cl} and X does not have any non-empty subset that is unfounded on X with respect to Π. We start by showing that X is a model of $UP(\Pi)$. We then demonstrate that X is a model of $UF(\Pi)$. By Proposition 3, it will immediately follow that X is a model of $UPUF(\Pi)$.

Let (M, l) be any inference in $UP(\Pi)$ such that M is consistent with X. Since $(M, l) \in UP(\Pi)$, Π^{cl} has a clause $C \vee l$ (modulo a reordering of literals) such that for every literal $u \in C$, $\overline{u} \in M$. Thus, all literals $\overline{u}$, $u \in C$, are consistent with X, that is, X is a model of $\neg C$. Since X is a model of Π^{cl}, X is a model of l, that is, l is consistent with X. Thus, X is a model of $UP(\Pi)$.

Let (M, l) be any inference in $UF(\Pi)$ such that M is consistent with X. By the definition of $UF(\Pi)$, there is an atom $a \in \sigma_\Pi$ such that $l = \neg a$ and $a \in U$, for some set $U \in Unf(M, \Pi)$. Since M is consistent with X, U is also unfounded on $X \cup \{\neg a \colon a \in \sigma_\Pi \setminus X\}$ with respect to Π. By the definition of an answer set, $a \notin X$. Consequently, $l = \neg a$ is consistent with X. It follows that X is a model of $UF(\Pi)$ and, by the comment above, a model of $UPUF(\Pi)$.

Conversely, let $X \subseteq \sigma_\Pi$ be a model of $UPUF(\Pi)$. By Proposition 3, X is a model of $UP(\Pi)$ and a model of $UF(\Pi)$. Let us assume that X is not an answer set of Π.

Case 1: X is not a model of Π^{cl}. Then there is a clause C in Π^{cl} such that X is a model of $\neg C$. Let us assume that $C = u_1 \vee \ldots \vee u_k$. It follows that the set $\{\overline{u}_1, \ldots, \overline{u}_k\}$ is consistent with X. Since C is a clause of Π^{cl}, $k \geq 1$. Moreover, since C is not a tautology, $u_k \notin \{\overline{u}_1, \ldots, \overline{u}_{k-1}\}$. By the definition of $UP(\Pi)$, $(\{\overline{u}_1, \ldots, \overline{u}_{k-1}\}, u_k) \in UP(\Pi)$. Since $\{\overline{u}_1, \ldots, \overline{u}_{k-1}\}$ is consistent with X and X is a model of $UP(\Pi)$, u_k is consistent with X, a contradiction.

Case 2: X contains an element, say a, that belongs to a set that is unfounded on $X \cup \{\neg a \colon a \in \sigma_\Pi \setminus X\}$ with respect to Π. By the definition of the rule *Unfounded*, we conclude that $(X \cup \{\neg a \colon a \in \sigma_\Pi \setminus X\}, \neg a) \in UF(\Pi)$. Since X is a model of $UF(\Pi)$ and is consistent with $X \cup \{\neg a \colon a \in \sigma_\Pi \setminus X\}$, $\neg a$ is consistent with X, a contradiction.

Statement 2: Every logic program Π is equivalent to the module $smodels(\Pi)$. Let X be an answer set of Π. By Statement 1, X is a model of $UP(\Pi)$ and $UF(\Pi)$. Since $ARC(\Pi) \subseteq UF(\Pi)$, X is a model of $ARC(\Pi)$. The proof that X is also a model of $BC(\Pi)$ uses the well-known fact that if X is an answer set of Π, X is also a supported model of Π that is, for every $a \in X$, there is $B \in Bodies(\Pi)$ such that X is a model of $s(B)$. Let us consider an inference $(M, l) \in BC(\Pi)$ such that M is consistent with X. By the definition of $BC(\Pi)$, there is a rule $a \leftarrow B$ in Π such that $a \in M$, $l \in s(B)$, and for every $B' \in Bodies(\Pi, a)$ such that $s(B') \neq s(B)$, there is $u \in s(B')$ such that $\overline{u} \in M$. Let $a \leftarrow B'$ be a rule in Π such that $s(B') \neq s(B)$ and let $u \in s(B')$ be such that $\overline{u} \in M$. Since M is consistent with X, X is not a model of u (if $u = b$, for some atom b, $\neg b \in M$ and so, $b \notin X$; if $u = \neg b$, for some atom b, $b \in M$ and so, $b \in X$). It follows that X is not a model of $s(B')$ for any rule $a \leftarrow B'$ in Π, where $s(B') \neq s(B)$. Since X is a supported model of Π and $a \in X$, X must be a model of $s(B)$. In particular, X is a model of l, that is, l is consistent with X. Thus, X is a model of the inference (M, l) and so, a model of $BC(\Pi)$. By Proposition 3, X is a model of $smodels(\Pi)$.

Conversely, let $X \subseteq \sigma_\Pi$ be a model of $smodels(\Pi)$. By the definitions of $UPUF(\Pi)$ and $smodels(\Pi)$, $UPUF(\Pi) \subseteq smodels(\Pi)$. Thus, X is a model of $UPUF(\Pi)$. By Statement 1, X is an answer set of Π. □

Since states are *sequences* of literals, we will often refer to *prefixes* of states. Formally, given a state $l_1 l_2 \ldots l_n$, every sequence $l_1 l_2 \ldots l_k$, where $0 \leq k \leq n$, is its prefix. In particular, each state is its own prefix.

Lemma 18. *Let S be an abstract inference module and N a non-fail state in the transition system AM_S reachable in AM_S from $\emptyset$. For every prefix M' of N and for every model X of S, if every decision literal of M' is consistent with X, then $[M']$ is consistent with X.*

Proof. We proceed by induction on the length of a path from $\emptyset$ to N in AM_S. If that length is 0, $N = \emptyset$ and the claimed property trivially holds. Let us consider a non-fail state N reachable from $\emptyset$ by a path p of length $k > 0$ and let us assume that every non-fail state reachable in AM_S from $\emptyset$ by a path of length at most $k - 1$ has the claimed property. Let M' be the state preceding N on p. Since M' is reachable from $\emptyset$ by a path of length $k - 1$, it follows by the induction hypothesis that M' has the claimed property. That is, for every prefix M'' of M' (including $M'' = M'$) and for every model X of S, if every decision literal in M'' is consistent with X then $[M'']$ is consistent with X.

Let N be of the form $l_1 \dots l_n$.[10] Since N is reached from M' by an edge resulting from *Inference Propagate*, *Backtrack* or *Decide* (applying the rule *Fail* results in the state $\bot$), $l_1 \dots l_{n-1}$ is a prefix of M'.

Let X be a model of S and M'' a prefix of N such that all decision literals in M'' are consistent with X. If M'' is a prefix of $l_1 \dots l_{n-1}$, then M'' is a prefix of M'. By the observation above, $[M'']$ is consistent with X. In particular, if all decision literals in $l_1 \dots l_{n-1}$ are consistent with X, $[l_1 \dots l_{n-1}]$ is consistent with X.

The only other case is $M'' = N$. Since all decision literals in N are consistent with X, all decision literals of $l_1 \dots l_{n-1}$ are consistent with X. Thus, by the observation above, $[l_1 \dots l_{n-1}]$ is consistent with X. To complete the proof, we have to show that l_n is consistent with X. The edge connecting M' to $N = l_1 \dots l_n$ in AM_S is not generated by the rule *Fail*. This leaves us with three cases to consider.

Inference Propagate: In this case, $M' = l_1 \dots l_{n-1}$, $[M']$ is consistent, $l_n \notin [M']$ and for some $M'' \subseteq [M']$, (M'', l_n) is an inference of S. By Proposition 2, $S \not\approx_{M''} l_n$. Since $[M']$ is consistent with X, M'' is consistent with X and so, l_n is consistent with X.

Backtrack: In this case, M' has the form $l_1 \dots l_{n-1} \bar{l}_n^{\Delta}\ Q$, where Q contains no decision literals, and $[M'] = [l_1 \dots l_{n-1} \bar{l}_n^{\Delta}\ Q]$ is inconsistent. Let us assume that l_n is not consistent with X. It follows that $\bar{l}_n$ is consistent with X. Consequently, all decision literals of M' are consistent with X. By the induction hypothesis, $[M']$ is consistent with X, a contradiction. Thus, l_n is consistent with X.

Decide: In this case, $M' = l_1 \dots l_{n-1}$ and l_n is a decision literal. Since all decision literals of M are consistent with X then, trivially, $[l_n]$ is consistent with X. □

Theorem 6. *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).

Proof. Part (a) can be proved following the argument for Proposition 1 in the paper by Lierler [34]. It also follows as a corollary from Theorem 11(a), to which we provide an explicit proof later on.

(b) Let M be a terminal state of AM_S other than $\bot$. Since neither *Fail* nor *Backtrack* is applicable, $[M]$ is consistent. Since *Decide* is not applicable, $[M]$ assigns all literals, that is, $[M]$ is complete. Let (M', l) be an inference of S such that M' is consistent with $[M]^+$. It follows that $M' \subseteq [M]$. Since *Inference Propagate* is not applicable, $l \in M$. Thus, l is consistent with $[M]^+$. It follows that $[M]^+$ is a model of S.

(c) Left-to-right: Since $\bot$ is reachable from $\emptyset$, there is a state M without decision literals such that there is a path in AM_S from $\emptyset$ to M, and there is an edge from M to $\bot$ in AM_S due to the application of *Fail*. It follows that $[M]$ is inconsistent. By Lemma 18, $[M]$ is consistent with every model of S (indeed, M has no decision literals). Since $[M]$ is inconsistent, S has no models.

Right-to-left: From (a) it follows that there is a path in AM_S from $\emptyset$ to some terminal state. Since S has no models, (b) implies that this state must be $\bot$. □

Proposition 7. *For every CNF formula F with no empty clause,* $\text{DP}_F = \text{AM}_{UP(F)}$.

Proof. It is clear that the two graphs have the same sets of nodes ($\bot$ and all states over the vocabulary σ_F). It is also clear that both graphs have the same edges arising from the generic rules *Fail*, *Backtrack* and *Decide*. Thus, let us consider an edge in DP_F implied by the rule *Unit Propagate*$_F$. By definition, this edge has the form (M, Ml), where $[M] \subseteq Lit(\sigma_F)$ is consistent, $l \in Lit(\sigma_F) \setminus M$, and there is a clause $C \vee l \in F$ such that for every literal u of C, $\overline{u} \in [M]$. It follows that $([M], l) \in UP(F)$ and, by the definition of *Inference Propagate*$_{UP(F)}$, (M, Ml) is an edge of $\text{AM}_{UP(F)}$.

Conversely, let us consider an edge of $\text{AM}_{UP(F)}$ implied by *Inference Propagate*$_{UP(F)}$. This edge is of the form (M, Ml), where $[M] \subseteq Lit(\sigma_F)$ is consistent, $l \in Lit(\sigma_F) \setminus M$, and for some $M' \subseteq [M]$, (M', l) is an inference of $UP(F)$. It follows that there is a clause $C \vee l \in F$, such that for every literal u in C, $\overline{u} \in M'$. Consequently, for every literal u in C, $\overline{u} \in [M]$ and so, (M, Ml) is an edge of DP_F. □

Proposition 9. *For every logic program Π,* $\text{SM}_\Pi = \text{AM}_{smodels(\Pi)}$.

Proof. The proof follows the same line of argument as the previous one and is an immediate consequence of the definitions. □

We say that a sequence M' *extends* a sequence M if M is a prefix of M'; moreover, M' *properly* extends M if M' extends M and $M \neq M'$. We recall that states other than the fail state $\bot$ are sequences and note that the following lemma follows directly from the definitions of the transition rules.

[10] As we move from a state to a non-fail state in the transition, the state can grow or shrink, the latter when the rule *Backtrack* is used. Thus, we have that $n \leq k$ and, in general, the inequality is strict.

Lemma 19. *If $M \neq \bot$ is a state in AME_S and M' is a successor of M such that $M' \neq \bot$, then M' is a proper extension of M or $M' = P\,\bar{l}$, where l is the last decision literal in $M = P\,l^{\Delta}\,Q$.*

Lemma 20. *Let S be an abstract module. For every path p in AME_S starting in a state M, every state that follows M on p is equal to $\bot$, or is a proper extension of M, or contains a literal $\bar{l}$, for some decision literal l^{Δ} in M.*

Proof. We prove the statement by induction on the number of decision literals k in M. Let $k = 0$. Then, $M = \bot$ or M is a sequence of non-decision literals. In the first case, path p consists of M only and the assertion is trivially true (there are no nodes on p that *follow* M). In the second case, Lemma 19 and a simple inductive argument imply that every state on p that follows M, other than $\bot$, has M as its proper prefix.

Thus, let $k \geq 1$ and let us assume that the assertion holds for every path originating in a state with at most $k-1$ decision literals. For the induction step, let us consider a state $M = P_1 l_1^{\Delta} P_2 l_2^{\Delta} \ldots P_k l_k^{\Delta} P_{k+1}$, where $P_1, \ldots, P_{k+1}$ contain no decision literals. If all states on p contain l_k^{Δ} then, by Lemma 19 and a simple induction, all states that follow M on p are of the form $M\,Q$, for some non-empty sequence Q of literals (possibly annotated) that are unassigned in M. Thus, the assertion follows. Otherwise, let M' be the first state on p not containing l_k^{Δ}. By Lemma 19 and a simple inductive argument, all states on p strictly between M and M' properly extend M and $M' = P_1 l_1^{\Delta} P_2 l_2^{\Delta} \ldots P_k \bar{l}_k$. In particular, $\bar{l}_k \in M'$. Moreover, by the induction hypothesis, every state that follows M' on p is equal to $\bot$, is an extension of M' and, consequently, contains $\bar{l}_k$, or contains $\bar{l}_i$, for some i, $1 \leq i \leq k-1$. Thus, the assertion follows in this case, too. □

Theorem 11. *For every abstract inference module S,*

(a) the graph AME_S is finite and acyclic,
(b) the $\bot$ state is reachable from $\emptyset$,
(c) for every path from $\emptyset$ to $\bot$ in AME_S, the set of states in which the rule Enumerate applies is precisely the set of models of S over σ_S, and for each model X of S over σ_S there is exactly one state M on the path such that $X = [M]$.

Proof. (a) Finiteness of AME_S is evident. Let us assume that there is a cycle in AME_S. Then, there is a path in AME_S that starts in a state M and returns to M after traversing a positive number of edges. That contradicts Lemma 20.
(b) It is easy to see that every path ending in a state other than $\bot$ can be extended. Since AME_S is acyclic, following outgoing edges of nodes in AME_S (breaking ties in an arbitrary way, if more than one rule applies) eventually takes us to $\bot$.
(c) We note that (i) the states of graphs AM_S and AME_S coincide, and (ii) each edge of the graph AM_S is also an edge of the graph AME_S. Also, for every terminal state M of AM_S other than $\bot$, the rule *Enumerate* is the only transition rule applicable to M in AME_S. By Theorem 6(b), it follows that $[M]^+$ is a model of S. Thus, if M is a state on a path p from $\emptyset$ to $\bot$ in AME_S, and the edge on p leading from M out to the next state on the path is determined by *Enumerate*, then $[M]^+$ is a model of S.

To conclude the proof, we show that every model of S over σ_S will eventually be reached by any path from $\emptyset$ to $\bot$. Let X be a model of S over σ_S. Consider any path p from $\emptyset$ to $\bot$ in AME_S. Let P denote the longest prefix of a state on p such that $[P] \subseteq X$. Let M denote the first state on p such that P is a prefix of M. We will show that $M = P$ and $[P] = X = [M]$.

Case 1. $M = P\ Q$, where Q is a nonempty sequence of literals. It follows that $M \neq \emptyset$ and that M was obtained from its predecessor on p, say M', by means of one of the rules of AME_S other than the *Fail* rule. It follows that P is the prefix of M'. This contradicts the fact that M is the first state on p such that P is a prefix of M.

Case 2. $M = P$. It remains to show that $[P] = X$. We recall that $[P] \subseteq X$. Towards a contradiction, let us assume that $[P] \subset X$. It follows that (i) P is consistent and (ii) $X \setminus [P]$ is nonempty and contains literals that are unassigned by $[P]$. By (i), rules *Fail* and *Backtrack* are not applicable. By (ii), rule *Decide* is applicable and hence rule *Enumerate* is not applicable. By P' we denote the successor state for P on p. Let us assume that P' is generated by the *Unit Propagate* rule. Then, $P' = Pl$ and since P is consistent with X and X is a model of S, l is consistent with X and so, $l \in X$. This contradicts the fact that P is the longest prefix of any state on p such that $[P] \subseteq X$. Thus, P' is obtained from P by the *Decide* rule and so, $P' = Pl^{\Delta}$. Since P is the longest prefix of any state on p such that $[P] \subseteq X$, $l \notin X$. The path p can only terminate by entering $\bot$ from a state with no decision literals by an application of either *Enumerate* or *Fail* rule. Let M' be the first state on p after M that does not contain l^{Δ}. By Lemma 19 and a simple inductive argument, $M' = P\bar{l}$. We recall that $l \notin X$. Thus, $\bar{l} \in X$ and $[P\bar{l}] \subseteq X$, a contradiction.

From Lemma 20 it immediately follows that we will not encounter two states encoding the same model on any path in AME_S. □

Proposition 12. *Every program Π is input-equivalent to the module $UPUF'(\Pi)$.*

Proof. By definition, X is an input answer set of Π if and only if X is an answer set of $\Pi \cup (X \setminus Head(\Pi))$. By Proposition 5, X is an answer set of $\Pi \cup (X \setminus Head(\Pi))$ if and only if X is a model of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$. Thus, to complete the proof it suffices to show that X is a model of $UPUF'(\Pi)$ if and only if X is a model of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$.

Let us assume that X is a model of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$. Let (M, l) be an inference of $UPUF'(\Pi)$ such that M is consistent with X. To prove that X is a model of $UPUF'(\Pi)$ we need to show that l is consistent with X. If (M, l) is implied by the rule *Unit Propagate* then, clearly, (M, l) is also an inference of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$. Since X is a model of that module and M is consistent with X, l is consistent with X.

Thus, let us assume that (M, l) is implied by the rule *Unfounded'*. It follows that $l = \neg a$ and that for some set U of atoms, U is unfounded on M w.r.t. Π, $a \in U$, and for every $b \in U$, $b \in Head(\Pi)$ or $\neg b \in M$. Let us assume that $a \in X$. Since M is consistent with X and X is a model of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$, $(M, \neg a)$ is not an inference of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$. In particular, it follows that U is not unfounded on M w.r.t. $\Pi \cup (X \setminus Head(\Pi))$. Since U is unfounded on M w.r.t. Π, we obtain that $U \cap (X \setminus Head(\Pi)) \neq \emptyset$. Let $b \in U \cap (X \setminus Head(\Pi))$. Then $b \in U$, $b \in X$ and $b \notin Head(\Pi)$. By the properties of U, $\neg b \in M$. Since M is consistent with X, $b \notin X$, a contradiction. Thus, $a \notin X$ and so, $\neg a$ (that is, l) is consistent with X.

Conversely, let us assume that X is a model of $UPUF'(\Pi)$ and let (M, l) be an inference of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$ such that M is consistent with X. We will show that l is consistent with X. This property will imply that X is a model of $UPUF(\Pi \cup (X \setminus Head(\Pi)))$, thus completing the argument.

Case 1. The inference (M, l) is determined by the rule *Unit Propagate* applied to a clause from Π^{cl}. It follows that (M, l) is also an inference of the module $UPUF'(\Pi)$. Since X is a model of $UPUF'(\Pi)$ and M is consistent with X, l is consistent with M.

Case 2. The inference (M, l) is determined by the rule *Unit Propagate* applied to a single-atom clause a, where $a \in X \setminus Head(\Pi)$. It follows that $l = a$. Since $a \in X \setminus Head(\Pi)$, then a (that is, l) is consistent with X.

Case 3. The inference (M, l) is determined by the rule *Unfounded*, that is, it is of the form $(M, \neg a)$, where $a \in \sigma_\Pi$, a is unassigned by M, and a belongs to some set U of atoms that is unfounded on M w.r.t. $\Pi \cup (X \setminus Head(\Pi))$. It is clear that $U \cap (X \setminus Head(\Pi)) = \emptyset$. In particular, U is unfounded on M also w.r.t. Π.

Let us define $M' = M \cup \{\neg b \colon b \in \sigma_\Pi \setminus X,\ b \neq a\}$. First, it is evident that a is unassigned by M'. Second, M' is consistent with X and so, M' is consistent. Finally, U is unfounded on M' w.r.t. Π (since $M \subseteq M'$). Let $b \in U$. If $b \notin Head(\Pi)$, then $b \notin X$ (otherwise, we would have $b \in U \cap (X \setminus Head(\Pi))$). Thus, $\neg b \in M'$. Since a is unassigned in M' $(M', \neg a)$ is an inference of $UPUF'(\Pi)$ implied by the rule *Unfounded'* (with U as an unfounded set underlying it). Since X is a model of $UPUF'(\Pi)$ and M' is consistent with X, $\neg a$ (that is, l) is consistent with X. □

Proposition 13. *Every modular program $\{\Pi_1, \ldots, \Pi_n\}$ is equivalent to the abstract modular system $\{UPUF'(\Pi_1), \ldots, UPUF'(\Pi_n)\}$.*

Proof. A set X of atoms is a model of a modular program $\{\Pi_1, \ldots, \Pi_n\}$ if and only if X is an input answer set of every Π_i, $1 \leq i \leq n$. Similarly, X is a model of the abstract modular system $\{UPUF'(\Pi_1), \ldots, UPUF'(\Pi_n)\}$ if and only if X is a model of every abstract module $UPUF'(\Pi_i)$, $1 \leq i \leq n$. Thus, the result follows from Proposition 12. □

Proposition 14. *Every SMT program $P = \langle T, \lambda_1, \ldots, \lambda_n \rangle$ is equivalent to any of the following abstract modular systems (over the vocabulary σ_T)*

1. $\{Ent(T), Ent(\lambda_1), \ldots, Ent(\lambda_n)\}$,
2. $\{UP(T), Ent(\lambda_1), \ldots, Ent(\lambda_n)\}$,
3. $\{Ent(T), Min(\lambda_1), \ldots, Min(\lambda_n)\}$,
4. $\{UP(T), Min(\lambda_1), \ldots, Min(\lambda_n)\}$.

Proof. *Statement 1.* Let M be a consistent and complete set of literals over σ_T such that M is a model of P. By the definition of a model of an SMT program, M^+ is a model of T and, for every i, $1 \leq i \leq n$, M is a λ_i-model. By Proposition 4, the former implies that M^+ is a model of $Ent(T)$. We will now use the latter to show that for every i, $1 \leq i \leq n$, M^+ is a model of $Ent(\lambda_i)$. To this end, let us consider an inference $(L, l) \in Ent(\lambda_i)$ such that L is consistent with M^+. We need to show that l is consistent with M^+. Since L is consistent with M^+, $L^+ \subseteq M^+$ and $L^- \cap M^+ = \emptyset$. By the assumption that both M and L are sets of literals over σ_T, we obtain $L \subseteq M$. From the construction of $Ent(\lambda_i)$ it follows that $\lambda[L] \models l$. Since M is consistent and complete, and is also a λ_i-model such that $L \subseteq M$, $l \in M$. Consequently, l is consistent with M^+.

Conversely, let M be a consistent and complete set of literals over σ_T such that M^+ is a model of $\{Ent(T), Ent(\lambda_1), \ldots, Ent(\lambda_n)\}$. By the definition of a model of an AMS, M^+ is a model of $Ent(T)$ and, for every i, $1 \leq i \leq n$, a model of $Ent(\lambda_i)$. By Proposition 4, M^+ is a model of T. It remains to show that for every i, $1 \leq i \leq n$, M is a λ_i-model. Let us fix an arbitrary i, $1 \leq i \leq n$, and proceed by contradiction. That is, let us assume that M is not a λ_i-model. Since M is a consistent and complete set of literals over σ_T, for every literal $l \in \sigma_T$, $\lambda_i[M] \models l$. Let us consider any literal l over σ_T such that $l \notin M$ (since M is consistent, such literals exist). From the definition of $Ent(\lambda_i)$ it follows that $(M, l) \in Ent(\lambda_i)$. Since M is consistent with M^+ and M^+ is a model of $Ent(\lambda_i)$ it follows that l is consistent with M^+. By the completeness of M, $l \in M$, a contradiction.

Statement 2. The proof follows that of Statement 1. The only difference is that we now use the module $UP(T)$ as an equivalent abstract inference module representation of T (cf. Proposition 4).

Statement 3. Let M be a consistent and complete set of literals over σ_T such that M is a model of P. We reason as before and obtain that M^+ is a model of T. In the proof of Statement 1, we showed that M^+ is a model of $Ent(\lambda_i)$. Since $Min(\lambda_i) \subseteq Ent(\lambda_i)$, M^+ is a model of $Min(\lambda_i)$.

The converse implication can be proved exactly as in Statement 1, as the only elements of $Ent(\lambda_i)$ we used in the reasoning also belong to $Min(\lambda_i)$.

Statement 4. The proof follows the lines of the argument of Statement 3 with the same proviso that we used in Statement 2. □

Theorem 15. *Every abstract modular inference system $\mathcal{A}$ is equivalent to the abstract inference module $\mathcal{A}^{\cup}$.*

Proof. By definition, X is a model of $\mathcal{A} = \{S_1, \ldots, S_n\}$ if and only if X is a model of every module S_i, $1 \leq i \leq n$. By Proposition 3, that latter holds if and only if X is a model of the module $\mathcal{A}^{\cup}$. □

Theorem 16. *For every AMS $\mathcal{A}$,*

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

Proof. By definition, $\text{AMS}_{\mathcal{A}} = \text{AM}_{\mathcal{A}^{\cup}}$. By Theorem 6(a), $\text{AM}_{\mathcal{A}^{\cup}}$ is finite and acyclic. Thus, the graph $\text{AMS}_{\mathcal{A}}$ is finite and acyclic, too. Next, by Theorem 6(b), any terminal state of $\text{AM}_{\mathcal{A}^{\cup}}$ other than $\perp$ is a model of $\mathcal{A}^{\cup}$. Thus, by Theorem 15, each such state is a model of $\text{AMS}_{\mathcal{A}}$. Part (c) follows by a similar argument. □

Theorem 17. *For every AMS $\mathcal{A}$,*

(a) the graph $\text{AMSL}_{\mathcal{A}}$ is finite and acyclic,
(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 $\perp$ is reachable from $\emptyset \| \emptyset, \ldots, \emptyset$ in $\text{AMSL}_{\mathcal{A}}$ if and only if $\mathcal{A}$ has no models.

Proof (Sketch). (a) The set of augmented states is obviously finite as augmented states are defined over a finite vocabulary. Thus, the graph $\text{AMSL}_{\mathcal{A}}$ is finite. Let us assume that $\text{AMSL}_{\mathcal{A}}$ contains a cycle, say C. Since transition rules either keep the second component of a state the same or extend it, C is of the form $M_0 \| \mathcal{G}, M_1 \| \mathcal{G}, \ldots, M_p \| \mathcal{G}$, where $\mathcal{G}$ is a sequence of sets of inferences, and each M_i is a state over $\sigma_{\mathcal{A}}$. Let $G = \bigcup \mathcal{G}$. One can show that $M_0, M_1, \ldots, M_p$ is a cycle in $\text{AMS}_{\mathcal{A}^G}$, a contradiction with Theorem 16(a).

(b) Let us assume that $M \| \mathcal{G}$ is reachable from $\emptyset \| \emptyset, \ldots, \emptyset$. It follows that $G = \bigcup \mathcal{G}$ is $\mathcal{A}$-safe. Using this observation, one can show that M is reachable from $\emptyset$ in $\text{AMS}_{\mathcal{A}^G}$. Moreover, since $M \| \mathcal{G}$ is a semi-terminal state in $\text{AMSL}_{\mathcal{A}}$, M is a terminal state in $\text{AMS}_{\mathcal{A}^G}$. By Theorem 16(b), $[M]^+$ is a model of $\mathcal{A}^G$. Since G is $\mathcal{A}$-safe, $\mathcal{A}^G$ and $\mathcal{A}$ are equivalent and so, $[M]^+$ is a model of $\mathcal{A}$.

(c) Let us first assume that $\perp$ is reachable from $\emptyset \| \emptyset, \ldots, \emptyset$ in $\text{AMSL}_{\mathcal{A}}$. Let $M \| \mathcal{G}$ be the direct predecessor of $\perp$ on one of those reachability paths. It follows that M is inconsistent and contains no decision literals. Reasoning as before, we can show that M is reachable from $\emptyset$ in $\text{AMS}_{\mathcal{A}^G}$ (where $G = \bigcup \mathcal{G}$). Since M is inconsistent and contains no decision literals, $\perp$ is reachable from $\emptyset$ in $\text{AMS}_{\mathcal{A}^G}$. By Theorem 16(c), $\mathcal{A}^G$ is not satisfiable. Since G is $\mathcal{A}$-safe, $\mathcal{A}^G$ and $\mathcal{A}$ are equivalent. Consequently, $\mathcal{A}$ is unsatisfiable (has no models).

Next, let us assume that $\perp$ is not reachable from $\emptyset \| \emptyset, \ldots, \emptyset$ in $\text{AMSL}_{\mathcal{A}}$. Then $\perp$ is not reachable from $\emptyset$ in $\text{AMS}_{\mathcal{A}}$. By Theorem 16(c), $\mathcal{A}$ has models. □

References

[1] K. Apt, H. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988, pp. 89–142.

[2] M. Balduccini, Representing constraint satisfaction problems in answer set programming, in: Proceedings of ICLP Workshop on Answer Set Programming and Other Computing Paradigms, ASPOCP, 2009, https://www.mat.unical.it/ASPOCP09/.

[3] C. Barrett, R. Nieuwenhuis, A. Oliveras, C. Tinelli, Splitting on demand in sat modulo theories, in: M. Hermann, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning, in: Lecture Notes in Computer Science, vol. 4246, Springer, Berlin, 2006, pp. 512–526.

[4] C. Barrett, R. Sebastiani, S. Seshia, C. Tinelli, Satisfiability modulo theories, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2008, pp. 737–797.

[5] M. Brain, An algebra of search spaces, in: A.M. Frisch, B. O'Sullivan (Eds.), Proceedings of the ERCIM Workshop on Constraint Solving and Constraint Logic Programming, 2011, pp. 72–86, http://csclp2011.cs.st-andrews.ac.uk/csclp2011proceedings.pdf.

[6] G. Brewka, T. Eiter, Equilibria in heterogeneous nonmonotonic multi-context systems, in: Proceedings of the 22nd National Conference on Artificial Intelligence, AAAI 2007, AAAI Press, 2007, pp. 385–390.

[7] R. Brochenin, Y. Lierler, M. Maratea, Abstract disjunctive answer set solvers, in: T. Schaub, G. Friedrich, B. O'Sullivan (Eds.), Proceedings of the 21st European Conference on Artificial Intelligence, ECAI 2014, in: Frontiers in Artificial Intelligence and Applications, vol. 263, IOS Press, 2014, pp. 165–170.

[8] F. Calimeri, W. Faber, G. Pfeifer, N. Leone, Pruning operators for disjunctive logic programming systems, Fundam. Inform. 71 (2–3) (2006) 183–214.
[9] M. Dao-Tran, T. Eiter, M. Fink, T. Krennwallner, Modular nonmonotonic logic programming revisited, in: P. Hill, D. Warren (Eds.), Logic Programming, in: Lecture Notes in Computer Science, vol. 5649, Springer, Berlin, 2009, pp. 145–159.
[10] L.M. de Moura, N. Bjørner, Satisfiability modulo theories: introduction and applications, Commun. ACM 54 (9) (2011) 69–77, http://doi.acm.org/10.1145/1995376.1995394.
[11] M. Denecker, Y. Lierler, M. Truszczynski, J. Vennekens, A Tarskian informal semantics for answer set programming, in: A. Dovier, V.S. Costa (Eds.), ICLP (Technical Communications), in: LIPIcs, vol. 17, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012, pp. 277–289.
[12] V. D'Silva, L. Haller, D. Kroening, Abstract conflict driven learning, in: R. Giacobazzi, R. Cousot (Eds.), The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, ACM, 2013, pp. 143–154.
[13] D. East, M. Truszczynski, Predicate-calculus-based logics for modeling and solving search problems, ACM Trans. Comput. Log. 7 (1) (2006) 38–83.
[14] T. Eiter, M. Fink, T. Krennwallner, C. Redl, Conflict-driven ASP solving with external sources, Theory Pract. Log. Program. 12 (4–5) (Sep. 2012) 659–679, http://dx.doi.org/10.1017/S1471068412000233.
[15] T. Eiter, G. Ianni, R. Schindlauer, H. Tompits, A uniform integration of higher-order reasoning and external evaluations in answer-set programming, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005, Morgan Kaufmann, San Francisco, CA, USA, 2005, pp. 90–96.
[16] S. El-Din Bairakdar, M. Dao-Tran, T. Eiter, M. Fink, T. Krennwallner, The DMCS solver for distributed nonmonotonic multi-context systems, in: T. Janhunen, I. Niemelä (Eds.), Proceedings of the 12th European Conference on Logics in Artificial Intelligence, JELIA 2010, in: LNCS, vol. 6341, Springer, Berlin, 2010, pp. 352–355.
[17] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Conflict-driven answer set solving, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007, Morgan Kaufmann, San Francisco, CA, USA, 2007, pp. 386–392.
[18] M. Gebser, B. Kaufmann, T. Schaub, Solution enumeration for projected boolean search problems, in: W.-J. van Hoeve, J. Hooker (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, in: Lecture Notes in Computer Science, vol. 5547, Springer, Berlin, 2009, pp. 71–86.
[19] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: from theory to practice, Artif. Intell. 187 (2012) 52–89.
[20] M. Gebser, M. Ostrowski, T. Schaub, Constraint answer set solving, in: Proceedings of 25th International Conference on Logic Programming, ICLP, Springer, Berlin, 2009, pp. 235–249.
[21] M. Gebser, T. Schaub, Tableau calculi for answer set programming, in: S. Etalle, M. Truszczynski (Eds.), Proceedings of the 22nd International Conference on Logic Programming, ICLP 2006, in: LNCS, vol. 4079, Springer, Berlin, 2006, pp. 11–25.
[22] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R. Kowalski, K. Bowen (Eds.), Proceedings of International Logic Programming Conference and Symposium, MIT Press, 1988, pp. 1070–1080.
[23] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (1991) 365–385.
[24] E. Giunchiglia, N. Leone, M. Maratea, On the relation among answer set solvers, Ann. Math. Artif. Intell. 53 (1–4) (2008) 169–204.
[25] E. Giunchiglia, Y. Lierler, M. Maratea, Answer set programming based on propositional satisfiability, J. Autom. Reason. 36 (2006) 345–377.
[26] C.P. Gomes, H. Kautz, A. Sabharwal, B. Selman, Satisfiability solvers, in: F. van Harmelen, V. Lifschitz, B. Porter (Eds.), Handbook of Knowledge Representation, Elsevier, 2008, pp. 89–134.
[27] C.P. Gomes, A. Sabharwal, B. Selman, Model counting, in: Handbook of Satisfiability, IOS Press, 2009, pp. 633–654.
[28] J. Jaffar, J. Lassez, Constraint logic programming, in: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21–23, 1987, ACM Press, 1987, pp. 111–119.
[29] J. Jaffar, M. Maher, Constraint logic programming: a survey, J. Log. Program. 19 (20) (1994) 503–581.
[30] T. Janhunen, Modular equivalence in general, in: 19th European Conference on Artificial Intelligence, ECAI, 2008, pp. 75–79.
[31] T. Janhunen, G. Liu, I. Niemelä, Tight integration of non-ground answer set programming and satisfiability modulo theories, in: Working Notes of the 1st Workshop on Grounding and Transformations for Theories with Variables, 2011.
[32] T. Janhunen, E. Oikarinen, H. Tompits, S. Woltran, Modularity aspects of disjunctive stable models, in: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'07, Springer, Berlin, 2007, pp. 175–187, http://dl.acm.org/citation.cfm?id=1758481.1758499.
[33] M. Järvisalo, E. Oikarinen, T. Janhunen, I. Niemelä, A module-based framework for multi-language constraint modeling, in: E. Erdem, F. Lin, T. Schaub (Eds.), Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2009, in: LNCS, vol. 5753, Springer, Berlin, 2009, pp. 155–168.
[34] Y. Lierler, Abstract answer set solvers with backjumping and learning, Theory Pract. Log. Program. 11 (2011) 135–169.
[35] Y. Lierler, Relating constraint answer set programming languages and algorithms, Artif. Intell. 207 (2014) 1–22.
[36] Y. Lierler, M. Truszczynski, Transition systems for model generators — a unifying approach, in: 27th International Conference on Logic Programming, ICLP 2011, Theory Pract. Log. Program. 11 (4–5) (2011), Special Issue.
[37] Y. Lierler, M. Truszczynski, Modular answer set solving, in: Late-Breaking Developments in the Field of Artificial Intelligence, in: AAAI Workshops, vol. WS-13-17, AAAI, 2013.
[38] Y. Lierler, M. Truszczynski, Abstract modular inference systems and solvers, in: M. Flatt, H.-F. Guo (Eds.), Practical Aspects of Declarative Languages, in: Lecture Notes in Computer Science, vol. 8324, Springer, Berlin, 2014, pp. 49–64.
[39] Y. Lierler, M. Truszczynski, An abstract view on modularity in knowledge representation, in: B. Bonet, S. Koenig (Eds.), Proceedings of the 29th AAAI Conference on Artificial Intelligence, 2015, pp. 1532–1538.
[40] V. Lifschitz, D. Pearce, A. Valverde, Strongly equivalent logic programs, ACM Trans. Comput. Log. 2 (4) (2001) 526–541.
[41] V. Lifschitz, H. Turner, Splitting a logic program, in: P.V. Hentenryck (Ed.), Proceedings of the 11th International Conference on Logic Programming, ICLP 1994, 1994, pp. 23–37.
[42] V. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: The Logic Programming Paradigm: a 25-Year Perspective, Springer, Berlin, 1999, pp. 375–398.
[43] M. Mariën, J. Wittocx, M. Denecker, M. Bruynooghe, SAT(ID): satisfiability of propositional logic extended with inductive definitions, in: H.K. Büning, X. Zhao (Eds.), Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing, SAT 2008, in: LNCS, vol. 4996, Springer, Berlin, 2008, pp. 211–224.
[44] J.P. Marques Silva, I. Lynce, S. Malik, Conflict-driven clause learning SAT solvers, in: Handbook of Satisfiability, IOS Press, 2009, pp. 131–153.
[45] V.S. Mellarkod, M. Gelfond, Y. Zhang, Integrating answer set programming and constraint logic programming, Ann. Math. Artif. Intell. 53 (1–4) (2008) 251–287.
[46] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings DAC-01, 2001.
[47] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Ann. Math. Artif. Intell. 25 (1999) 241–273.
[48] I. Niemelä, P. Simons, Extending the Smodels system with cardinality and weight constraints, in: J. Minker (Ed.), Logic-Based Artificial Intelligence, Kluwer, 2000, pp. 491–521.
[49] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T), J. ACM 53 (6) (2006) 937–977.

[50] E. Oikarinen, T. Janhunen, Modular equivalence for normal logic programs, in: G. Brewka, S. Coradeschi, A. Perini, P. Traverso (Eds.), Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006, IOS Press, Amsterdam, The Netherlands, 2006, pp. 412–416.
[51] F. Rossi, P. van Beek, T. Walsh, Constraint programming, in: F. van Harmelen, V. Lifschitz, B. Porter (Eds.), Handbook of Knowledge Representation, Elsevier, 2008, pp. 181–212.
[52] D. Saccà, C. Zaniolo, Stable models and non-determinism in logic programs with negation, in: D.J. Rosenkrantz, Y. Sagiv (Eds.), Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1990, ACM, New York, NY, USA, 1990, pp. 205–217.
[53] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artif. Intell. 138 (2002) 181–234.
[54] S. Tasharrofi, E. Ternovska, A semantic account for modularity in multi-language modelling of search problems, in: C. Tinelli, V. Sofronie-Stokkermans (Eds.), Proceedings of the 8th International Symposium on Frontiers of Combining Systems, FroCoS 2011, in: LNCS, vol. 6989, Springer, Berlin, 2011, pp. 259–274.
[55] S. Tasharrofi, X.N. Wu, E. Ternovska, Solving modular model expansion tasks, CoRR, arXiv:1109.0583, 2011.
[56] A. Van Gelder, K. Ross, J. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (3) (1991) 620–650.