Propagators and Solvers for the Algebra of Modular Systems

Bart Bogaerts[1,2]*, Eugenia Ternovska[3], and David Mitchell[3]

[1] Department of Computer Science, Aalto University, Espoo, Finland
[2] KU Leuven, Department of Computer Science, Leuven, Belgium
[3] Computational Logic Laboratory, Simon Fraser University, Vancouver, Canada

Abstract

Solving complex problems can involve non-trivial combinations of distinct knowledge bases and problem solvers. The Algebra of Modular Systems is a knowledge representation framework that provides a method for formally specifying such systems in purely semantic terms. Many practical systems based on expressive formalisms solve the model expansion task. In this paper, we construct a solver for the model expansion task for a complex modular system from an expression in the algebra and black-box propagators or solvers for the primitive modules. To this end, we define a general notion of propagators equipped with an explanation mechanism, an extension of the algebra to propagators, and a lazy conflict-driven learning algorithm. The result is a framework for seamlessly combining solving technology from different domains to produce a solver for a combined system.

1 Introduction

Complex artifacts are, of necessity, constructed by assembling simpler components. Software systems use libraries of reusable components, and often access multiple remote services. In this paper, we consider systems that can be formalized as solving the model expansion task for some class of finite structures. A wide range of problem solving and query answering systems are so accounted for. We present a method for automatically generating a solver for a complex system from a declarative definition of that system in terms of simpler modules, together with solvers for those modules. The work is motivated primarily by "knowledge-intensive" computing contexts, where the individual modules are defined in (possibly different) declarative languages, such as logical theories or logic programs, but can be applied anywhere the model expansion formalization can.

The Algebra of Modular Systems (AMS) [48, 49], provides a way to define a complex module in terms of a collection of other modules, in purely semantic terms. Formally, each module in this algebra represents a class of structures, and a "solver" for the module solves the model expansion task for that class. That is, a solver for module $\mathscr{M}$ takes as input a structure $\mathscr{A}$ for a part of the vocabulary of $\mathscr{M}$, and returns either a set of expansions of $\mathscr{A}$ that are in $\mathscr{M}$, or the empty set. The operators of the algebra

*Bart Bogaerts is a postdoctoral fellow of the Research Foundation – Flanders (FWO). This work is supported by the Finnish Center of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170).

operate on classes of structures, essentially generalizing Codd's Relation Algebra [15] from tables to classes of structures.

While the AMS provides a good account of how to define a module in terms of other modules, little work has been done on combining solvers (see e.g., [38]). The purpose of this paper is to fill this gap. To this end, we give general definitions of propagators and solvers in terms of classes of partial structures, and define algebras that correspond to the AMS, but operate on propagators rather than classes of structures. The algebra provides a theoretical basis for the practical combination of solvers.

Intuitively, a propagator adds information to a partial structure without eliminating any solutions that extend that partial structure, while a solver searches through the set of more precise structures for a complete structure that is in the module. To formalize propagators and the operations of solvers, we use partial structures defined over the truth values $\mathbf{u}$, $\mathbf{f}$, $\mathbf{t}$, $\mathbf{i}$ (unknown, false, true and inconsistent, respectively), with the "precision order" $\mathbf{u} <_p \mathbf{t} <_p \mathbf{i}$, $\mathbf{u} <_p \mathbf{f} <_p \mathbf{i}$. Extending this order pointwise to structures, we obtain a complete lattice of partial structures ordered by precision. The least structure in the lattice carries no information (all formulas are "unknown"), while the greatest structure is inconsistent. (For simplicity, in this paper we consider only finite vocabularies and finite structures.)

Contributions. Our main contributions are as follows.

(1) We define a propagator for a module $\mathscr{M}$ to be an operator on the lattice of partial structures that is monotone ($\mathfrak{A} \geq_p \mathfrak{A}'$ implies $P(\mathfrak{A}) \geq_p P(\mathfrak{A}')$, and and information-preserving ($P(\mathfrak{A}) \geq_p \mathfrak{A}$). This concept generalizes many uses of the term "propagator" in the literature.

(2) We define an algebra of propagators that has the same operators and the same vocabulary as AMS, but the domain is propagators rather than modules. The propagator algebra is analogous to the AMS in the following sense. Each propagator P is for a unique module, $module(P)$). The function *module* is a surjective homomorphism from the algebra of propagators to the AMS. For example, if E_1, E_2 are modules, then $\pi_\delta(\sigma_{(Q \equiv R)} E_1 \times E_2)$ is a compound module. It represents the class of structures $\mathscr{A}$ such that some structure $\mathscr{A}'$ exists that coincides with $\mathscr{A}$ on δ and such that $\mathscr{A}'$ satisfies both E_1 and E_2 and interprets Q the same as R. If P_1 and P_2 are propagators for E_1 and E_2 respectively, then we show that in our extended version of the algebra, $\pi_\delta(\sigma_{(Q \equiv R)} P_1 \times P_2)$ is a propagator for $\pi_\delta(\sigma_{(Q \equiv R)} E_1 \times E_2)$.

(3) We show how solvers can be constructed from propagators, and vice versa.

(4) We study *complexity* of the combined propagators in terms of complexity of propagators for the individual modules and show that in general, our operations can increase complexity. This is useful, for example, to build a propagator for Quantified Boolean Formulas based on a propagator that (only) performs unit propagation for a propositional theory. We discuss how this can be done in Section 5.

(5) To model more interesting algorithms, we define *explaining propagators* to be propagators that return an "explanation" of a propagation in terms of simpler (explaining) propagators. That is, an explaining propagator P maps a partial structure to a more precise partial structure together with an "explanation" of that propagation in the form of an explaining propagator that is "simpler" than P. These propagators generalizes lazy clause generation [47], cutting plane generation [16] and counterexample-guided abstraction-refinement [14]. This generalization shows that many techniques are actually instances of the same fundamental principles.

(6) We extend the algebra of propagators to explaining propagators.

(7) We show how to construct a conflict-driven learning solver for a module from an explaining propagator for the module. The resulting algorithm is an abstract generalization of the conflict-driven clause learning (CDCL) algorithm for SAT, and other algorithms that can be found in the literature.

(8) We give several examples of techniques for applying the algebra in constructing solvers.

Our formal framework results effectively in a paradigm where pieces of information (modules) are accompanied with implemented technology (propagators) and where composing solving technology is possible with the same ease as composing modules. The algorithms we propose are an important step towards practical applicability of the algebra of modular system.

Related Work. The closest related work is research on technology integration. Examples include but are not limited to [23, 5, 42]. Combined solving is perhaps most developed in the *SAT modulo theories (SMT)* community, where theory propagations are tightly interleaved with satisfiability solving [41, 46]. The work we present in this paper differs from SMT in a couple of ways. In SMT, the problem associated with so-called *theories* (modules in our terminology) is the satisfiability problem. This has several ramifications. One of them is that two different decidable theories cannot always be combined into one decidable theories; this problem is known in the SMT community as the *combination problem*; a large body of work has been devoted to the study of this problem, see for instance [40, 52, 3, 20, 12, 13]. In our approach on the other hand, the focus is not on the satisfiability problem, but on the model expansion problem, which is, from a complexity point of view, a simpler task than satisfiability checking [30]. As a consequence, we do not encounter problems such as the combination problem: any combination of propagators always yields a propagator. Instead, in our work, we focus on *different ways to combine propagators*. In SMT, theories are always combined conjunctively. For instance, SMT provides no support for negation (it is possible to state that the negation of an atom is entailed by a theory, but not that a theory is not satisfiable) or projection (projecting out certain variables in a given theory). This kind of combinations, generalizing the AMS to an algebra of propagators, results in a setting where propagators for simple modules (or "theories" in SMT terminology) can be combined into more complex propagators for the combined module. These combinations constitute the essence of our work.

Recently, Lierler and Truszczyński [33] introduced a formalism with compositions (essentially, conjunctions) of modules given through solver-level inferences of the form (M,l), where M is a consistent set of literals and l is a literal not in M. Such pairs are called inferences of the module. Transition graphs for modules are constructed, with actions such as Propagate, Fail, Backtrack, and Decide. Solvers based on the transition graph are determined by the *select-edge-to-follow* function (search strategy). Solving templates are investigated for several formalisms, including SAT and ASP. From individual transition graphs, such graphs are constructed for conjunctions of modules, but more complex combinations of modules are not studied.

Combining propagators has been studied in constraint programming [9, 25, 29]. This research is often limited to a subset of the operations we consider here, for instance studying only conjunction [6, 1], disjunction [39, 55, 31] or connectives from propositional logic [32, 4]. In addition to these, we also consider selection and projection. The objectives of the current paper are similar to those considered the CP community; however, there are some key differences. First of all, we generalized the theory of propagators from constraint programming to the AMS. It can be applied in principle to every logic with a model semantics. As such, it can serve as a formal basis to transfer the rich body of work from constraint programming to other fields, such as for instance (Integer) Linear Programming or Answer Set Programming. Second, the traditional treatment of propagation in CP emphasizes tractability [25]: the focus is on propagators that can be computed in polynomial time. While it is often important to constrain the complexity of the propagators, it can be useful as well to allow for complexity increasing operations. In our framework, one of the operations (projection) increases complexity; as such, contrary to the propagators considered in constraint programming, it allows to construct propagators for compound modules for which membership checking is not polynomial. As explained above, this is useful to construct propagators for expressive logics such as QBF. Third, we equip our propagators with a *learning mechanism* that generalizes for instance lazy clause generation [47] from constraint programming and a *conflict analysis* mechanism that generalizes conflict-driven clause learning [36] from SAT [35].

In real-world applications multiple (optimization) problems are often tightly intertwined. This has been argued intensively by Bonyadi et al. [8], who designed the traveling thief problem as a prototype benchmarking problem of this kind. In the current state-of-the-art, such problems are tackled by special-purpose algorithms [11, 54], illustrating the need for a principled approach to combining solving technology from different research fields.

2 Modular Systems

Structures. A (relational)[1] *vocabulary* τ is a finite set of predicate symbols. A τ*-structure* $\mathscr{A}$ consists of a domain A and an assignment of an n-ary relation $Q^{\mathscr{A}}$ over A to all n-predicate symbols $Q \in \tau$. A *domain atom* is an expression of the form $Q(\overline{d})$ with $Q \in \tau$, and $\overline{d}$ a tuple of domain elements. The value of a domain atom $Q(\overline{d})$ in a structure $\mathscr{A}$ (notation $Q(\overline{d})^{\mathscr{A}}$) is true (**t**) if $\overline{d} \in Q^{\mathscr{A}}$ and false (**f**) otherwise. From now on, we assume that A is a fixed domain, shared by all structures. This assumption is not needed for the Algebra of Modular Systems in general, but it is convenient for the current paper since the input for the task we tackle (model expansion, see below) fixes the domain.

A *four-valued* τ-structure $\mathfrak{A}$ is an assignment $Q(\overline{d})^{\mathfrak{A}}$ of a four-valued truth value (true (**t**), false (**f**), unknown (**u**) or inconsistent (**i**)) to each domain atom over τ. If $Q(\overline{d})^{\mathfrak{A}} \in \{\mathbf{t},\mathbf{f},\mathbf{u}\}$ for each $Q(\overline{d})$, we call $\mathfrak{A}$ *consistent*. If $Q(\overline{d})^{\mathfrak{A}} \in \{\mathbf{t},\mathbf{f}\}$ for each $Q(\overline{d})$, we call $\mathfrak{A}$ *two-valued* and identify it with the corresponding structure. A four-valued structures is sometimes also called a *partial* structure, as it provides partial information about values of domain atoms.

The precision order on truth values is induced by $\mathbf{u} <_p \mathbf{t} <_p \mathbf{i}$, $\mathbf{u} <_p \mathbf{f} <_p \mathbf{i}$. This order is pointwise extended to (four-valued) structures: $\mathfrak{A} <_p \mathfrak{A}'$ iff for all domain atoms $Q(\overline{d})$, $Q(\overline{d})^{\mathfrak{A}} <_p Q(\overline{d})^{\mathfrak{A}'}$. The set of all four-valued τ-structures forms a complete lattice when equipped with the order $\leq_p$. This means that every set $\mathscr{S}$ of (four-valued) structures has a greatest lower bound $\mathrm{glb}_{\leq_p}(\mathscr{S})$ and a least upper bound $\mathrm{lub}_{\leq_p}(\mathscr{S})$ in the precision order. Hence, there is a *most precise* four-valued structure $\mathrm{glb}(\varnothing)$, which we denote $\mathfrak{I}$; $\mathfrak{I}$ is the most inconsistent structure: it maps all domain atoms to **i**.

Four-valued structures are used to approximate structures. If $\mathscr{A}$ is a structure and $\mathfrak{A}$ a partial structure, we say that $\mathfrak{A}$ *approximates* $\mathscr{A}$ if $\mathfrak{A} \leq_p \mathscr{A}$. Below, we use four-valued structures to represent a *solver state*: certain domain atoms have been decided (they are mapped to **t** or **f**), other domain atoms atoms have not yet been assigned a value (they are mapped to **u**), and certain domain atoms are involved in an inconsistency (they are mapped to **i**). If a partial structure is inconsistent, it no longer approximates any structure. Solvers typically handle situations in which their state is inconsistent by backtracking.

If $Q(\overline{d})$ is a domain atom and $v \in \{\mathbf{t},\mathbf{f},\mathbf{i},\mathbf{u}\}$, we use $\mathfrak{A}[Q(\overline{d}):v]$ for the (four-valued) structure equal to $\mathfrak{A}$ except for interpreting $Q(\overline{d})$ as v. We use $\mathfrak{A}[Q:Q^{\mathfrak{A}'}]$ for the four-valued structure equal to $\mathfrak{A}$ on all symbols except for Q and equal to Q on $\mathfrak{A}'$. If $\delta \subseteq \tau$, we use $\mathfrak{A}|_\delta$ for the structure equal to $\mathfrak{A}$ on δ and mapping every other domain atom to **u**, i.e., $\mathfrak{A}|_\delta$ is the least precise structure that coincides with $\mathfrak{A}$ on δ.

Modules. Let $\tau_M = \{M_1, M_2, \dots\}$ be a fixed vocabulary of *atomic module symbols* (often just referred to as *atomic modules*) and let τ be a fixed vocabulary. *Modules* are built by the grammar:

$$E ::= \bot \mid M_i \mid E \times E \mid -E \mid \pi_\delta E \mid \sigma_{Q \equiv R} E. \tag{1}$$

We call $\times$ *product*, $-$ *complement*, π_δ *projection* onto δ, and $\sigma_{Q\equiv R}$ *selection*. Modules that are not atomic are called *compound*. Each atomic module symbol M_i has an associated vocabulary $voc(M_i) \subseteq \tau$. The *vocabulary* of a compound module is given by

[1]Without loss of generality, we restrict to relational vocabularies in this paper.

- $voc(\bot) = \tau$,
- $voc(E_1 \times E_2) = voc(E_1) \cup voc(E_2)$,
- $voc(-E) = voc(E)$,
- $voc(\pi_\delta E) = \delta$, and
- $voc(\sigma_\Theta E) = voc(E)$.

Semantics. Let $\mathscr{C}$ be the set of all τ-structures with domain A. Modules (atomic and compound) are interpreted by subsets of $\mathscr{C}$.[2] A *module interpretation* I assigns to each atomic module $M_i \in \tau_M$ a set of τ-structures such that any two τ-structures $\mathscr{A}_1$ and $\mathscr{A}_2$ that coincide on $voc(M_i)$ satisfy $\mathscr{A}_1 \in \mathscr{I}(M_i)$ iff $\mathscr{A}_2 \in \mathscr{I}(M_i)$. The value of a modular expression E in $\mathscr{I}$, denoted $[\![E]\!]^{\mathscr{I}}$, is defined as follows.

$$\begin{aligned}
&[\![\bot]\!]^{\mathscr{I}} := \varnothing.\\
&[\![M_i]\!]^{\mathscr{I}} := \mathscr{I}(M_i).\\
&[\![E_1 \times E_2]\!]^{\mathscr{I}} := [\![E_1]\!]^{\mathscr{I}} \cap [\![E_2]\!]^{\mathscr{I}}.\\
&[\![-E]\!]^{\mathscr{I}} := \mathscr{C} - [\![E]\!]^{\mathscr{I}}.\\
&[\![\pi_\delta(E)]\!]^{\mathscr{I}} := \{\mathscr{A} \mid \exists \mathscr{A}' \ (\mathscr{A}' \in [\![E]\!]^{\mathscr{I}} \text{ and } \mathscr{A}|_\delta = \mathscr{A}'|_\delta)\}.\\
&[\![\sigma_{Q\equiv R} E]\!]^{\mathscr{I}} := \{\mathscr{A} \in E^{\mathscr{I}} \mid Q^{\mathscr{A}} = R^{\mathscr{A}}\}.
\end{aligned}$$

We call $\mathscr{A}$ a *model* of E in $\mathscr{I}$ (denoted $\mathscr{A} \models_{\mathscr{I}} E$) if $\mathscr{A} \in [\![E]\!]^{\mathscr{I}}$.

In earlier papers [48, 49], the algebra was presented slightly differently; here, we restrict to a *minimal syntax*; this is discussed in detail in Section 5.

Example 2.1. Let $\tau = \{Edge, Trans\}$ be a vocabulary containing two binary predicates. Let $\mathscr{I}$ be a model interpretation, M_t a module with vocabulary τ such that $\mathscr{A} \models_{\mathscr{I}} M_t$ if and only if $Trans^{\mathscr{A}}$ is the transitive closure of $Edge^{\mathscr{A}}$ and M_f is a module with vocabulary $\{Trans\}$ such that $\mathscr{A} \models_{\mathscr{I}} M_f$ if and only if $Trans^{\mathscr{A}}$ is the full binary relation on A. Consider the following compound module

$$E := \pi_{\{Edge\}}(M_t \times (-M_f)).$$

Here, E is a module with vocabulary $\{Edge\}$ such that $\mathscr{A} \models_{\mathscr{I}} E$ if and only if $Edge^{\mathscr{A}}$ is a disconnected graph: the module M_t sets $Trans$ to be the transitive closure of $Edge$; the term $(-M_f)$ ensures that $Trans$ is not the full binary relation; these two modules are combined conjunctively and the result is projected onto $\{Edge\}$, i.e., the value of $Trans$ does not matter in the result. ▲

From now on, we assume that a module interpretation $\mathscr{I}$ is given and fixed. Slightly abusing notation, we often omit the reference to $\mathscr{I}$ and write, e.g., $\mathscr{A} \models E$ instead of $\mathscr{A} \models_{\mathscr{I}} E$.

Model expansion for modular systems. The *model expansion task* for modular systems is: given a (compound) module E and a partial structure $\mathfrak{A}$, find a structure $\mathscr{A}$ (or: find all structures $\mathscr{A}$) such that $\mathscr{A} \geq_p \mathfrak{A}$ and $\mathscr{A} \models_{\mathscr{I}} E$ (if one such exists).

Mitchell and Ternovska [38] have defined methods to apply the lazy clause generation (LCG) paradigm [19] to solve the model expansion problem for modular systems. In particular, given propagators P_i that explain their propagations by means of clauses for atomic modules M_i, they show how to build an LCG-solver for modules of the form $E = M_1 \times \cdots \times M_n$. In this paper, we generalize the above idea to a setting where E is an arbitrary (compound) module and the learning mechanism is not necessarily clause learning.

[2] If the assumption that A is fixed is dropped, modules are *classes* of structures instead of sets of structures, but this generality is not needed in this paper.

3 Propagators and Solvers

Now, we define a general notion of *propagator* and show how propagators for atomic modules can be composed into propagators for compound modules. Intuitively, a propagator is a blackbox procedure that refines a partial (four-valued) structure by deriving consequences of a given module.

Definition 3.1. A *propagator* is a mapping P from partial structures to partial structures such that the following hold:

- P is $\leq_p$-monotone: whenever $\mathfrak{A} \geq_p \mathfrak{A}'$, also $P(\mathfrak{A}) \geq_p P(\mathfrak{A}')$.
- P is information-preserving: $P(\mathfrak{A}) \geq_p \mathfrak{A}$ for each $\mathfrak{A}$.

Definition 3.2. Given a module E, a propagator P is an E*-propagator* if on two-valued structures, it coincides with E, i.e., whenever $\mathfrak{A}$ is two-valued, $P(\mathfrak{A}) = \mathfrak{A}$ iff $\mathfrak{A} \in E$.

Note that if $\mathfrak{A} \notin E$, and $\mathfrak{A}$ is two-valued, an E-propagator maps $\mathfrak{A}$ to an inconsistent partial structure since P is information-preserving. An E-propagator can never "lose models of E", as is formalised in the following lemma.

Lemma 3.3. *Let P be an E-propagator. If $\mathscr{A}$ is a model of E and $\mathscr{A} \geq_p \mathfrak{A}$, then also $\mathscr{A} \geq_p P(\mathfrak{A})$.*

Proof. Follows from $\leq_p$-monotonicity and the fact that $P(\mathscr{A}) = \mathscr{A}$ for two-valued structures $\mathscr{A}$. □

Example 3.4. Modern ASP solvers typically contain (at least) two propagators. One, which we call $P_{UP}^{\mathscr{P}}$, performs unit propagation on the completion of the program $\mathscr{P}$. The other, which we call $P_{UFS}^{\mathscr{P}}$ performs unfounded set propagation; that is: it maps a partial structure $\mathfrak{A}$ to

$$\mathrm{lub}_{\leq_p}(\mathfrak{A}, \mathfrak{A}[p : \mathbf{f} \mid p \in lUFS(\mathscr{P}, \mathfrak{A})]),$$

where $lUFS(\mathscr{P}, \mathfrak{A})$ is the largest unfounded set of $\mathscr{P}$ with respect to $\mathfrak{A}$ [53]. It is easy to see that these two propagators are information-preserving and $\leq_p$-monotone. ▲

Example 3.5. In several constraint solvers that perform bounds reasoning, (finite-domain) integer variables are represented by a relational representation of their bounds: a variable c is represented by a unary predicate $Q_{c\leq}$ with intended interpretation that $Q_{c\leq}(n)$ holds iff $c \leq n$. Consider in this setting a propagator $P_{c\leq d}$ that enforces bounds consistency for the constraint $c \leq d$. That is, $P_{c\leq d}$ maps a partial structure $\mathfrak{A}$ to a partial structure $\mathfrak{A}'$ such that for each n:

- if $Q_{d\leq}(n)^{\mathfrak{A}} \geq_p \mathbf{t}$, then $Q_{c\leq}(n)^{\mathfrak{A}'} = \mathrm{lub}_{\leq_p}(Q_{c\leq}(n)^{\mathfrak{A}}, Q_{d\leq}(n)^{\mathfrak{A}})$,
- otherwise, $Q_{c\leq}(n)^{\mathfrak{A}'} = Q_{c\leq}(n)^{\mathfrak{A}}$

and similar equations for $Q_{d\leq}(n)$. Intuitively, these equations state that if $d \leq n$ holds in $\mathfrak{A}$, then $P_{c\leq d}$ also propagates that $c \leq n$ holds. For instance, assume the domain $A = \{1, \dots, 100\}$ and that $\mathfrak{A}$ is such that

$$Q_{c\leq}(n)^{\mathfrak{A}} = \begin{cases} \mathbf{t} & \text{if } n \geq 90 \\ \mathbf{u} & \text{if } 90 > n \geq 10 \\ \mathbf{f} & \text{otherwise} \end{cases} \qquad Q_{d\leq}(n)^{\mathfrak{A}} = \begin{cases} \mathbf{t} & \text{if } n \geq 80 \\ \mathbf{u} & \text{if } 80 > n \geq 20 \\ \mathbf{f} & \text{otherwise} \end{cases}$$

This structure encodes that the value of c is in the interval $[10, 90]$ and d in the interval $[20, 80]$. In this case, $P_{c\leq d}$ propagates that also $c \leq 80$ without changing the value of d. Formally:

$$Q_{c\leq}(n)^{P_{c\leq d}(\mathfrak{A})} = \begin{cases} \mathbf{t} & \text{if } n \geq 80 \\ \mathbf{u} & \text{if } 80 > n \geq 10 \\ \mathbf{f} & \text{otherwise} \end{cases} \qquad Q_{d\leq}(n)^{P_{c\leq d}(\mathfrak{A})} = Q_{d\leq}(n)^{\mathfrak{A}}$$ ▲

Propagators and modules.

Lemma 3.6. *If P is a propagator, then there is a unique module E such that P is an E-propagator.*

Proof. Uniqueness follows immediately from Definition 3.2. Existence follows from the fact that we can define the module E such that $\mathscr{A} \models E$ if and only if $P(\mathscr{A}) = \mathscr{A}$. □

Lemma 3.6 gives rise to the following definition.

Definition 3.7. If P is a propagator, we define *module*(P) to be the unique module E such that P is an E-propagator.

Definition 3.8. If E is a module, the *E-checker* is the propagator P^E_{check} defined by:[3]

$$P^E_{check} : \mathfrak{A} \mapsto \begin{cases} \mathfrak{A} & \text{if } \mathfrak{A} \text{ is consistent but not two-valued} \\ \mathfrak{A} & \text{if } \mathfrak{A} \text{ is two-valued and } \mathfrak{A} \models E \\ \mathfrak{I} & \text{otherwise} \end{cases}$$

Lemma 3.9. *For each module E, P^E_{check} is an E-propagator.*

Proof. That P^E_{check} is a propagator follows from the fact that $\mathfrak{I}$ is more precise than any partial structure $\mathfrak{A}$. It follows immediately from the definition that P^E_{check} coincides with E on two-valued structures. □

The E-checker is the least precise E-propagator, as the following proposition states.

Proposition 3.10. *For each E-propagator P and each consistent structure $\mathfrak{A}$, $P(\mathfrak{A}) \geq_p P^E_{check}(\mathfrak{A})$.*

Propagators and Solvers.

Definition 3.11. Let E be a module. An *E-solver* is a procedure that takes as input a four-valued structure $\mathfrak{A}$ and whose output is the set $\mathscr{S}$ of all two-valued structures $\mathscr{A}$ with $\mathscr{A} \models E$ and $\mathscr{A} \geq_p \mathfrak{A}$.

Intuitively, a *solver* is a procedure that performs model expansion for a given module. Propagators can be used to create solvers and vice versa. We first describe how to build a simple generate-and-check solver from a propagator. Afterwards, we provide an algorithm that uses the solver in a smarter way. In the next section, we discuss how to add a learning mechanism to this solver.

Definition 3.12. Let P be an E-propagator. We define an E-solver S^P_{gc} as follows.[4] S^P_{gc} takes as **input** a structure $\mathfrak{A}$. The **state** of S^P_{gc} is a tuple $(\mathfrak{B}, \mathscr{S})$ of a structure and a set of two-valued structures $\mathscr{S}$; it is **initialised** as $(\mathfrak{A}, \varnothing)$. S^P_{gc} performs depth-first search on the search space of (four-valued) structures more precise than $\mathfrak{A}$. Choices consist of updating $\mathfrak{B}$ to $\mathfrak{B}[Q(\overline{d}) : \mathbf{t}]$ or to $\mathfrak{B}[Q(\overline{d}) : \mathbf{f}]$ for some domain atom with $Q(\overline{d})^{\mathfrak{B}} = \mathbf{u}$. Whenever $\mathfrak{B}$ is two-valued, the solver checks whether $P(\mathfrak{B}) = \mathfrak{B}$. If yes, $\mathfrak{B}$ is added to $\mathscr{S}$. After encountering a two-valued structure, it backtracks over its last choice. When the search space has been traversed, S^P_{gc} **returns** $\mathscr{S}$. Pseudo-code for this algorithm can be found in Algorithm 1.

Definition 3.13. Let P be a propagator. We define the solver S^P_p as follows. The solver S^P_p extends S^P_{gc} by updating $\mathfrak{B}$ to $P(\mathfrak{B})$ before each choice. If $\mathfrak{B}$ is inconsistent, S^P_p backtracks. Pseudocode for this solver can be found in Algorithm 2.

[3]Recall that $\mathfrak{I}$ denotes the most precise (inconsistent) structure.
[4]Here, *gc* stands for Generate-and-Check.

Algorithm 1 The solver S^P_{gc}

```
1:  function $S^P_{gc}(\mathfrak{A})$
2:      $(\mathfrak{B},\mathscr{S}) \leftarrow (\mathfrak{A},\varnothing)$
3:      while true do
4:          if $\mathfrak{B}$ is two-valued then
5:              if $P(\mathfrak{B}) = \mathfrak{B}$ then
6:                  $S \leftarrow S \cup \{\mathfrak{B}\}$
7:              if the last choice updated $\mathfrak{B}'$ to $\mathfrak{B}'[Q(\overline{d}) : \mathbf{t}]$ then
8:                  $\mathfrak{B} \leftarrow \mathfrak{B}'[Q(\overline{d}) : \mathbf{f}]$
9:              else                                  \\ i.e., if there were no more choices
10:                 return S
11:         else
12:             choose $\mathfrak{B} \leftarrow \mathfrak{B}[Q(\overline{d}) : \mathbf{t}]$ for some $Q(\overline{d})$ with $Q(\overline{d})^{\mathfrak{B}} = \mathbf{u}$
```

Algorithm 2 The solver S^P_p

```
1:  function $S^P_p(\mathfrak{A})$
2:      $(\mathfrak{B},\mathscr{S}) \leftarrow (\mathfrak{A},\varnothing)$
3:      while true do
4:          $\mathfrak{B} \leftarrow \mathrm{P}(\mathfrak{B})$
5:          if $\mathfrak{B}$ is two-valued or inconsistent then
6:              if $P(\mathfrak{B}) = \mathfrak{B}$ and $\mathfrak{B}$ is two-valued then
7:                  $S \leftarrow S \cup \{\mathfrak{B}\}$
8:              if the last choice updated $\mathfrak{B}'$ to $\mathfrak{B}'[Q(\overline{d}) : \mathbf{t}]$ then
9:                  $\mathfrak{B} \leftarrow \mathfrak{B}'[Q(\overline{d}) : \mathbf{f}]$
10:             else                                  \\ i.e., if there were no more choices
11:                 return S
12:         else
13:             choose $\mathfrak{B} \leftarrow \mathfrak{B}[Q(\overline{d}) : \mathbf{t}]$ for some $Q(\overline{d})$ with $Q(\overline{d})^{\mathfrak{B}} = \mathbf{u}$
```

Proposition 3.14. *If (the domain) A is finite and P is an E-propagator, then both S^P_{gc} and S^P_p are E-solvers.*

Sketch of the proof. Finiteness of A guarantees that depth-first search terminates. Correctness of S^P_{gc} follows from the fact that P is an E-propagator (since $\{\mathscr{A} \mid P(\mathscr{A}) = \mathscr{A}\} = \{\mathscr{A} \mid \mathscr{A} \models E\}$).

Correctness of S^P_p follows from Lemma 3.3 which states that no models are lost by propagation. □

In the above proposition, the condition that A is finite only serves to ensure termination of these two procedures that essentially traverse the entire space of structures more precise than $\mathfrak{B}$. All the concepts defined in this paper, for instance propagators, and operations on propagators, can also be used in the context of an infinite domain.

We now show how to construct a propagator from a solver. We call this propagator *optimal*, since it always returns the most precise partial structure any propagator could return (cf. Proposition 3.17). In this sense, this propagator performs skeptical reasoning.

Definition 3.15. Let S be an E-solver. We define an E-propagator $P^S_{opt} : \mathfrak{A} \mapsto \mathrm{glb}_{\leq_p} S(\mathfrak{A})$.

Notice that if S' is an E-solver as well, as a function, $P^{S}_{opt} = P^{S'}_{opt}$. However, we include S in the notation since for practical purposes, we need a way to compute $P^{S}_{opt}(\mathfrak{A})$; for this, a call to S is used.

Proposition 3.16. *If S is an E-solver, P^{S}_{opt} is an E-propagator.*

Proof. We first show that P^{S}_{opt} is a propagator. First, for each $\mathscr{A} \in S(\mathfrak{A})$, it holds that $\mathscr{A} \geq_p \mathfrak{A}$, hence also $P^{S}_{opt}(\mathfrak{A}) \geq_p \mathfrak{A}$. Second, notice that whenever $\mathfrak{A}_1 \geq_p \mathfrak{A}_2$, $S(\mathfrak{A}_1) \subseteq S(\mathfrak{A}_2)$, hence $P^{S}_{opt}(\mathfrak{A}_1) = \text{glb}_{\leq_p} S(\mathfrak{A}_1) \geq_p \text{glb}_{\leq_p} S(\mathfrak{A}_2) = P^{S}_{opt}(\mathfrak{A}_2)$. From these two properties, it follows that P^{S}_{opt} is indeed a propagator.

The fact that it is also an E-propagator follows from the property that for two-valued $\mathscr{A}$, $S(\mathscr{A}) = \{\mathscr{A}\}$ if $\mathscr{A} \models E$ and $S(\mathscr{A}) = \varnothing$ otherwise. □

Proposition 3.17. *Let P be any E-propagator and S an E-solver. For each structure $\mathfrak{A}$, it holds that $P^{S}_{opt}(\mathfrak{A}) \geq_p P(\mathfrak{A})$.*

Proof. From Lemma 3.3, we find that $P(\mathfrak{A}) \leq_p \mathscr{A}$ if $\mathscr{A} \in S(\mathfrak{A})$. Hence also $P(\mathfrak{A}) \leq_p \text{glb}_{\leq_p} S(\mathfrak{A}) = P^{S}_{opt}(\mathfrak{A})$. □

Combining propagators. First, we discuss how propagators for the same module can be combined. Afterwards, we extend the algebra of modular systems to propagators.

Proposition 3.18. *Composition $P_1 \circ P_2$ of two E-propagators is an E propagator. In particular, if P is an E-propagator, also P^n (an abbreviation for $P \circ P \circ \cdots \circ P$ (n times)) is an E-propagator. We use P^∞ for $\lim_{\leq_p} P^n = \text{lub}_{\leq_p}\{P^n \mid n \in \mathbb{N}\}$.*

Proposition 3.19. *If P_1 and P_2 are two E-propagators, then $(P_1 \circ P_2)(\mathfrak{A}) \geq_p P_1(\mathfrak{A})$ and $(P_1 \circ P_2)(\mathfrak{A}) \geq_p P_2(\mathfrak{A})$ for each $\mathfrak{A}$.*

Now, we show how checkers for compound expressions in the algebra can be built from propagators for atomic modules. These checkers are sufficient for defining the algebra on propagators: if propagators for atomic modules are given, Proposition 3.21 provides us with the means to obtain a propagator for compound expressions. However, for practical purposes, we are often interested in better, i.e., more precise propagators. Hence, after this proposition, we investigate for which operations better propagators can be defined.

Definition 3.20. Let P be an E-propagator, P' an E'-propagator and $\delta \subseteq \tau$. We define following checkers (we only define their behaviour on two-valued structures since otherwise the behaviour of checkers is trivial):

- $P^{\perp}_{check} : \mathscr{A} \mapsto \mathfrak{I}$
- $P^{E\times E'}_{check} : \mathscr{A} \mapsto \text{lub}_{\leq_p}\{P(\mathscr{A}), P'(\mathscr{A})\}$
- $P^{-E}_{check} : \mathscr{A} \mapsto \begin{cases} \mathscr{A} & \text{if } P(\mathscr{A}) = \mathfrak{I} \\ \mathfrak{I} & \text{otherwise} \end{cases}$
- $P^{\pi_\delta E}_{check} : \mathscr{A} \mapsto \begin{cases} \mathscr{A} & \text{if } S^{P}_{p}(\mathscr{A}|_\delta) \neq \varnothing \\ \mathfrak{I} & \text{otherwise} \end{cases}$
- $P^{\sigma_{Q\equiv R}E}_{check} : \mathscr{A} \mapsto \begin{cases} \mathscr{A} & \text{if } P(\mathscr{A}) = \mathscr{A} \text{ and } Q^{\mathscr{A}} = R^{\mathscr{A}} \\ \mathfrak{I} & \text{otherwise} \end{cases}$

Proposition 3.21. *The operations defined in Definition 3.20 define checkers. Furthermore for each compound module E'', $P^{E''}_{check}$ is an E''-checker.*

Sketch of the proof. Correctness for each of the above follows easily from the definition of the semantics of modular systems. □

Now, we present for several of the operations a better (more precise) propagator (compared to only checking).

Proposition 3.22. *Let P be an E-propagator, P' an E'-propagator and δ a sub-vocabulary of τ. We define the following operations:*

- $P \times P' : \mathfrak{A} \mapsto \mathrm{lub}_{\leq_p}\{P(\mathfrak{A}), P'(\mathfrak{A})\}$.
- $\pi_\delta P : \mathfrak{A} \mapsto \begin{cases} \mathfrak{I} & \textit{if } \mathfrak{A} \textit{ is inconsistent} \\ \mathfrak{I} & \textit{if } \mathfrak{A} \textit{ is two-valued on } \delta \textit{ and } S_p^P(\mathfrak{A}|_\delta) = \varnothing \\ \mathrm{lub}_{\leq_p}(P(\mathfrak{A}|_\delta)|_\delta, \mathfrak{A}|_{\tau\setminus\delta}) & \textit{otherwise.} \end{cases}$
- $\sigma_{Q\equiv R}P : \mathfrak{A} \mapsto (P(\mathfrak{A}))[Q : L, R : L]$ *where* $L = \mathrm{lub}_{\leq_p}(Q^{P(\mathfrak{A})}, R^{P(\mathfrak{A})})$.

It holds that $P \times P'$ is an $E \times E'$-propagator, $\pi_\delta P$ is a $\pi_\delta E$ propagator and $\sigma_{Q\equiv R}$ is a $\sigma_{Q\equiv R}E$-propagator.

Proof. We provide a proof for projection; the other operations are analogous.

We show that $\pi_\delta P$ is a **propagator**.

First, for each four-valued structure $\mathfrak{A}$, $P(\mathfrak{A}|_\delta) \geq_p \mathfrak{A}|_\delta$ since P is a propagator, hence also $\pi_\delta P(\mathfrak{A})|_\delta = P(\mathfrak{A}|_\delta)|_\delta \geq_p \mathfrak{A}|_\delta$. Furthermore, $\pi_\delta P(\mathfrak{A})|_{\tau\setminus\delta} = \mathfrak{A}|_{\tau\setminus\delta}$. Combining these two yields that $\pi_\delta P(\mathfrak{A}) \geq_p \mathfrak{A}$ and hence that $\pi_\delta P$ is indeed information preserving (the cases where $P(\mathfrak{A}) = \mathfrak{I}$ are trivial).

We show $\leq_p$-monotonicity of $\pi_\delta P$. Assume $\mathfrak{A}_1 \geq_p \mathfrak{A}_2$. If $\mathfrak{A}_2$ is inconsistent, then so is $\mathfrak{A}_1$ and thus $\pi_\delta P(\mathfrak{A}_1) = \pi_\delta P(\mathfrak{A}_2)$. If $\mathfrak{A}_2$ is two-valued on δ and $S_p^P(\mathfrak{A}_1|_\delta) = \varnothing$, then either $\mathfrak{A}_1$ is inconsistent, or $\mathfrak{A}_1|_\delta = \mathfrak{A}_2|_\delta$. In both cases, the result is trivial. If $\pi_\delta P(\mathfrak{A}_1) = \mathfrak{I}$, the result is trivial as well, hence we can assume that both $\mathfrak{A}_1$ and $\mathfrak{A}_2$ fall in the "otherwise" category in the definition of $\pi_\delta P$. The $\leq_p$-monotonicity of $\pi_\delta P$ now follows from the fact that if $\mathfrak{A}_1 \geq_p \mathfrak{A}_2$ then also **(1)** $\mathfrak{A}_1|_\delta \geq_p \mathfrak{A}_2|_\delta$ and thus $P(\mathfrak{A}_1|_\delta) \geq_p P(\mathfrak{A}_2|_\delta)$ and **(2)** $\mathfrak{A}_1|_{\tau\setminus\delta} \geq_p \mathfrak{A}_2|_{\tau\setminus\delta}$. Hence, we conclude that $\pi_\delta P$ indeed defines a propagator.

Now, we show that $\pi_\delta P$ is a π_δ**E-propagator**. Let $\mathscr{A}$ be a two-valued structure.

First suppose $\mathscr{A} \models \pi_\delta E$. In this case, there exists a two-valued $\mathscr{A}'$ such that $\mathscr{A}' \models E$ and $\mathscr{A}|_\delta = \mathscr{A}'|_\delta$. Thus, $S_p^P(\mathfrak{A}|_\delta) \neq \varnothing$ in this case. Also, in this case $P(\mathscr{A}|_\delta)$ is consistent and thus $P(\mathscr{A}|_\delta)|_\delta = \mathscr{A}|_\delta$. We conclude that in this case indeed $\pi_\delta P(\mathscr{A}) = \mathscr{A}$.

Now suppose $\mathscr{A} \not\models \pi_\delta E$. In this case, there exists no structure $\mathscr{A}'$ such that $\mathscr{A}'|_\delta = \mathscr{A}|_\delta$ and $\mathscr{A}' \models E$. Thus $S_p^P(\mathscr{A}|_\delta) = \varnothing$ and $\pi_\delta(\mathscr{A})$ is indeed inconsistent. □

The intuitions in the above proposition are as follows. For $P \times P'$, P computes consequences of E, while P' computes consequences of E', given an input structure $\mathfrak{A}$. The propagator $P \times P'$ combines the consequences found by both: it returns the least upper bound of $P(\mathfrak{A})$ and $P'(\mathfrak{A})$ in the precision order. That is, it returns the structure in which all domain literals derived by any of the two separate propagators hold (and nothing more). For projection $\pi_\sigma P$, in the two-valued case, the solver S_p^P is used to check whether $\mathscr{A} \in \pi_\delta E$. For the three-valued case, P is used to propagate on $\mathfrak{A}|_\delta$, i.e., using only the information about the projected vocabulary δ. From this propagation, only the information that is propagated about δ is kept (this is $P(\mathfrak{A}|_\delta)|_\delta$). Indeed $\pi_\delta E$ enforces no restrictions on symbols in $\tau \setminus \delta$. Furthermore, we transfer all knowledge we previously had on symbols in $\tau \setminus \delta$ (this is some form of inertia); the resulting structure equals $P(\mathfrak{A}|_\delta)$ on symbols in δ and equals $\mathfrak{A}$ on symbols in $\tau \setminus \delta$. For selection $\sigma_{Q\equiv R}P$, propagation happens according to P. Afterwards, all propagations for Q are also transferred to R and vice versa. This is done by changing the interpretations of both Q and R to the least upper bound (in the precision order) of their interpretations in $P(\mathfrak{A})$.

Example 3.23 (Example 3.4 continued)**.** We already mentioned that typical ASP solvers have two propagators $P^{\mathscr{P}}_{UP}$ and $P^{\mathscr{P}}_{UFS}$. The actual propagation is done according to $P^{\mathscr{P}}_{ASP} = P^{\mathscr{P}}_{UP} \times P^{\mathscr{P}}_{UFS}$. Furthermore, typically, this propagation is executed until a fixed point is reached, hence the entire propagation is described by $\left(P^{\mathscr{P}}_{ASP}\right)^{\infty}$.

Now let $M^{\mathscr{P}}$ be the module such that $\mathscr{A} \models M^{\mathscr{P}}$ if and only if $\mathscr{A}$ is a stable model of $\mathscr{P}$. It is well-known that a structure is a stable model of $\mathscr{P}$ if and only if it is a model of the completion and it admits no non-trivial unfounded sets [45]. From this, it follows that $M^{\mathscr{P}} = M^{\mathscr{P}}_{UP} \times M^{\mathscr{P}}_{UFS}$, where $M^{\mathscr{P}}_{UP}$ is a module such that $\mathscr{A} \models M^{\mathscr{P}}_{UP}$ iff $\mathscr{A}$ is a model of the completion of $\mathscr{P}$ and $M^{\mathscr{P}}_{UFS}$ is a module such that $\mathscr{A} \models M^{\mathscr{P}}_{UFS}$ iff $\mathscr{A}$ admits no non-trivial unfounded sets with respect to $\mathscr{P}$. It is easy to see that $P^{\mathscr{P}}_{UP}$ and $P^{\mathscr{P}}_{UFS}$ are $M^{\mathscr{P}}_{UP}$- and $M^{\mathscr{P}}_{UFS}$-propagators respectively. From this it follows by Propositions 3.22 and 3.18 that $P^{\mathscr{P}}_{ASP}$ and also $\left(P^{\mathscr{P}}_{ASP}\right)^{\infty}$ are $M^{\mathscr{P}}$-propagators. ▲

Up to this point, we have described three different ways to construct E-propagators: P^{S}_{opt} is the most precise E-propagator if S is an E-solver, Proposition 3.21 describes how to build E-checkers (the least precise propagators) from propagators for subexpressions of E and Proposition 3.22 illustrates how to build more precise propagators for compound expressions. However, *precision* is not the only criterion for "good" propagators. In practice, we expect propagators to be efficiently computable. We now show that this is indeed the case for the propagators defined in Proposition 3.22.

Proposition 3.24. *Assume A is finite; furthermore assume access to an oracle that computes $P(\mathfrak{A})$ and $P'(\mathfrak{A})$. The following hold*

- $(P \times P')(\mathfrak{A})$ *can be computed in polynomial time (in terms of the size of A),*
- $(\sigma_{Q\equiv R}P)(\mathfrak{A})$ *can be computed in polynomial time,*
- $(\pi_{\delta}P)(\mathfrak{A})$ *can be computed in nondeterministic polynomial time.*

Sketch of the proof. The first two statements follow easily from the definitions. For instance $(P \times P')(\mathfrak{A})$ is defined as $\text{lub}_{\leq_p}\{P(\mathfrak{A}), P'(\mathfrak{A})\}$. Computing this least upper bound can be done by comparing $Q(\overline{d})^{P(\mathfrak{A}}$ and $Q(\overline{d})^{P'(\mathfrak{A}}$ for each domain atom $Q(\overline{d})$. There are only polynomially many domain atoms.

For the last statement, the complexity is dominated by a call to $S^{P}_{p}(\mathfrak{A}|_{\delta})$, which is essentially depth-first search. □

Proposition 3.24 shows that product and selection do not increase complexity when compared to the complexity of the propagators they compose. However, the situation for projection is different. That is not surprising, since Tasharrofi and Ternovska [50] already showed that the projection operation increases the complexity of the task of deciding whether a structure is a member of a given module or not. As such, Proposition 3.24 shows that our propagators for compound expressions only increase complexity when dictated by the complexity of checking membership of the underlying module.

4 Explanations and Learning

In many different fields, propagators are defined that *explain* their propagations in terms of simpler constructs. For instance in CDCL-based ASP solvers [22, 2, 17], the unfounded set propagator explains its propagation by means of clauses. Similar explanations are generated for complex constraints in constraint programming (this is the lazy clause generation paradigm [47]) and in SAT modulo theories [21]. The idea to generate clauses to explain complex constraints already exists for a long time, see e.g. [37]. In integer programming, the cutting plane method [16] is used to enforce a solution to be integer. In this methodology, when a (rational) solution is found, a cutting plane is learned that explains *why* this particular solution should be rejected. Similarly, in QBF solving, counterexample-guided abstraction-refinement (the CEGAR methodology) [14, 27, 43] starts from the idea to first solve a relaxed problem (an abstraction), and on-the-fly add explanations *why* a certain solution to the relaxation is rejected. More QBF algorithms are based on some of learning information on the fly [44, 28, 56, 24]. De Cat et al. [18] defined a methodology where complex formulas are grounded on-the-fly. This is a setting where inference made by complex formulas is explained in terms of simpler formulas (in this case, formulas with a lower quantification depth).

The goal of this section is to extract the essential building blocks used in all of the aforementioned techniques to arrive at an abstract, algebraic, framework with solvers that can learn new constraints/propagators during search. We generalize the common idea underlying each of the above paradigms by adding explanations and learning to our abstract framework. We present a general notion of an *explaining propagator* and define a method to turn such a propagator into a solver that learns from these explanations. An explaining propagator is a propagator that not only returns the partial structure that is the result of its propagations ($P(\mathfrak{A})$), but also an explanation ($C(\mathfrak{A})$). This explanation takes the form of a propagator itself. Depending on the application, explanations must have a specific form. For instance, for lazy clause generation, the explanation must be a (set of) clause(s); in integer linear programming, the explanation must be a (set of) cutting plane(s). In general, there are two conditions on the explanation. First, it should explain why the propagator made certain propagations: $C(\mathfrak{A})$ should derive at least what P derives in $\mathfrak{A}$. Second, the returned explanation should not be arbitrary, it should be a consequence of the module underlying the propagator P.

Definition 4.1. An *explaining propagator* is tuple (P,C) where P is a propagator and C maps each partial structure either to UNEXPLAINED (notation $\Diamond$) or to an explaining propagator $C(\mathfrak{A}) = (P',C')$ such that the following hold:

(1) (explains propagation) $P(\mathfrak{A}) \leq_p P'(\mathfrak{A})$.

(2) (soundness): $module(P') \subseteq module(P)$

Example 4.2. Integer linear programs are often divided into two parts: some solver performs search using linear constraints. When a solution is found, a *checker* checks whether this solution is integer-valued. If not, this checker propagates inconsistency and *explains* this inconsistency by means of a cutting plane. This process fits in our general definition of *explaining propagator*: a cutting plane can be seen itself as a propagator: during search it can propagate that its underlying constraint is violated . This propagator *explains* the inconsistency and is a consequence of the original problem (namely of the integrality constraint). ▲

As can be seen, we allow an explaining propagator to *not explain* certain propagations. For instance, whenever $P(\mathfrak{A}) = \mathfrak{A}$, nothing new is derived, hence there is nothing to explain. We say that (P,C) *explains propagation from* $\mathfrak{A}$ if either $P(\mathfrak{A}) = \mathfrak{A}$ or $C(\mathfrak{A}) \neq \Diamond$. Each propagator P as defined in Definition 3.1 can be seen as an explaining propagator $(P,C_\Diamond)$, where $C_\Diamond$ maps each partial structure to $\Diamond$.

Example 4.3 (Example 3.5 continued)**.** For each natural number n let cl_n denote the clause $Q_{c\leq}(n) \lor \neg Q_{d\leq}(n)$ and let P_n denote the propagator that performs unit propagation on cl_n. For each $\mathfrak{A}$, let $U_\mathfrak{A}$ denote the set of all n's such that at least one literal from cl_n is false in $\mathfrak{A}$. Furthermore, let $C_{c\leq d}$ denote the mapping that maps each four-valued structure $\mathfrak{A}$ to

$$\begin{cases} \Diamond & \text{if } P_{c\leq d}(\mathfrak{A}) = \mathfrak{A} \\ \left(\bigtimes_{n\in U_\mathfrak{A}} P_n, C_\Diamond\right) & \text{otherwise,} \end{cases}$$

where $\bigtimes_{n\in U_\mathfrak{A}} P_n$ denotes the product of all P_n with $n \in U_\mathfrak{A}$. In this case, $(P_{c\leq d}, C_{c\leq d})$ is an explaining propagator. It explains each propagation by means of a set of clauses (the product of propagators P_n for individual clauses). This particular explanation is used for instance in MinisatID [17] and many other lazy clause generation CP systems. ▲

In general, using *anything* as explanation is a bad idea: what we are hoping for is that a propagator explains its propagations in terms of *simpler* propagators (where the definition of "simple" can vary from field to field). In order to generalize this idea, in what follows we assume that $\prec$ is a strict well-founded order on the set of all explaining propagators, where smaller propagators are considered "simpler". In the following definition, we also require that all propagations need to be explained, except for $\prec$-minimal propagators.

Definition 4.4. We say that an explaining propagator (P,C) *respects* $\prec$ if

- Whenever $C(\mathfrak{A}) = \Diamond$, $P(\mathfrak{A}) = \mathfrak{A}$ or (P,C) is $\prec$-minimal,
- In all other cases, $C(\mathfrak{A}) \prec (P,C)$ and $C(\mathfrak{A})$ respects $\prec$.

Example 4.5. For lazy clause generation, the order on propagators would be $(P,C) \prec (P',C')$ if P performs unit propagation for a set of clauses and P' does not. The conditions in Definition 4.4 guarantee that each non-clausal propagator explains its propagation in terms of these $\prec$-minimal propagators; i.e., in terms of clauses. ▲

Example 4.6. When grounding lazily [18], one can consider propagators P_φ that perform some form of propagation for a first-order formula φ. A possible order $\prec$ is then: $(P_\varphi, C_\varphi) \prec (P_{\varphi'}, C_{\varphi'})$ if φ has strictly smaller quantification depth then φ'. For instance, a propagator for a formula $\forall x : \exists y : \psi(x,y)$ can explain its propagations by means of a propagator for the formula $\exists y : \psi(d,y)$, where d is an arbitrary domain element. ▲

We now show how explaining propagators can be used to build solvers.

Definition 4.7. Let (P,C) be an explaining propagator that respects $\prec$. We define a *learning solver* $ls^{(P,C)}$ as follows. The **input** of $ls^{(P,C)}$ is a partial structure $\mathfrak{A}$. The **state** of $ls^{(P,C)}$ is a triple $(\mathscr{P}, \mathfrak{B}, \mathscr{S})$ where $\mathscr{P}$ is a set of explaining propagators, $\mathfrak{B}$ is a (four-valued) structure, and $\mathscr{S}$ is a set of (two-valued) structures. The state is **initialised** as $(\{(P,C)\}, \mathfrak{A}, \varnothing)$. The solver performs depth-first search on the structure $\mathfrak{B}$, where each choice point consists of assigning a value to a domain atom unknown in $\mathfrak{B}$. Before each choice point, until a fixed point is reached, $\mathfrak{B}$ is updated to $P^*(\mathfrak{B})$ and $\mathscr{P}$ to $\mathscr{P} \cup \{C^*(\mathfrak{B})\}$, where (P^*, C^*) is $\prec$-minimal among all elements of $\mathscr{P}$ that have $P^*(\mathfrak{B}) \neq \mathfrak{B}$. If no such element exists, no more propagation is possible and the solver makes another choice. Whenever $\mathfrak{B}$ is inconsistent, the solver backtracks over its last choice. If this search encounters a model (a two-valued structure $\mathscr{A}$ with $\mathscr{A} = P(\mathscr{A})$), it stores this model in $\mathscr{S}$ and adds $(P_{check}^{-\{\mathscr{A}\}}, C_\Diamond)$ to $\mathscr{P}$. After the search space has been traversed (i.e., inconsistency is derived without any choice points left), it returns $\mathscr{S}$. Pseudocode for this solver can be found in Algorithm 3.

Example 4.8 (Examples 3.23 and 4.5 continued)**.** Consider the order $\prec$ from Example 4.5. Note that $P_{UP}^{\mathscr{P}}$ performs unit propagation on a set of clauses and hence is $\prec$-minimal. In many modern ASP solvers [17, 22, 2], $P_{UFS}^{\mathscr{P}}$ explains its propagations in terms of clauses as well, resulting in an explanation mechanism $C_{UFS}^{\mathscr{P}}$. These clauses are typically obtained by applying acyclicity algorithms to the dependency graph of the program; for details, see for instance [34]. In ls, the order $\prec$ is then used to prioritize unit propagation over unfoundedness propagation, conform modern ASP practices. ▲

Proposition 4.9. *Assume A is finite. If (P,C) is an E-explaining propagator, then $ls^{(P,C)}$ is an E-solver.*

Sketch of the proof. As before, termination follows from the fact that A is finite. It follows from the second condition in Definition 4.1 that during execution of $ls^{(P,C)}$, for all $P' \in \mathscr{P}$, it holds that $module(P') \subseteq module(P)$, hence propagation is indeed correct for E. □

Algorithm 3 The solver $ls^{(P,C)}$

```
 1: function $S_p^P(\mathfrak{A})$
 2:     $(\mathscr{P},\mathfrak{B},\mathscr{S}) \leftarrow (\{(P,C)\},\mathfrak{A},\varnothing)$
 3:     while true do
 4:         while At least one $(P^*,C^*) \in \mathscr{P}$ has $P^*(\mathfrak{B}) \neq \mathfrak{B}$ do
 5:             Let $(P^*,C^*)$ be a $\prec$-minimal propagator with $P^*(\mathfrak{B}) \neq \mathfrak{B}$ in $\mathscr{P}$
 6:             $\mathfrak{B} \leftarrow P^*(\mathfrak{B})$
 7:             $\mathscr{P} \leftarrow \mathscr{P} \cup \{C^*(\mathfrak{B})\}$
 8:         if $\mathfrak{B}$ is two-valued or inconsistent then
 9:             if $P(\mathfrak{B}) = \mathfrak{B}$ and $\mathfrak{B}$ is two-valued then
10:                 $S \leftarrow S \cup \{\mathfrak{B}\}$
11:                 $\mathscr{P} \leftarrow \mathscr{P} \cup \{(P_{check}^{-\{\mathscr{A}\}}, C_\Diamond)\}$
12:             if the last choice updated $\mathfrak{A}'$ to $\mathfrak{A}'[Q(\overline{d}):\mathbf{t}]$ then
13:                 $\mathfrak{B} \leftarrow \mathfrak{B}'[Q(\overline{d}):\mathbf{f}]$
14:             else                                         \\ i.e., if there were no more choices
15:                 return S
16:         else
17:             choose $\mathfrak{B} \leftarrow \mathfrak{B}[Q(\overline{d}):\mathbf{t}]$ for some $Q(\overline{d})$ with $Q(\overline{d})^{\mathfrak{B}} = \mathbf{u}$
```

The next question that arises is: *how can explaining propagators for individual modules be combined into explaining propagators for compound modules?* The answer is not always simple. As for regular propagators, each operation can be defined trivially. In Section 3, being defined trivially meant simply defining the checker. In this case, additionally, it means that for C we take $C_\Diamond$. Below, we discuss some more interesting cases. We will use the following notations. If $\overline{d}$ is a tuple of domain elements, we use $P^{\overline{d}}_{Q\equiv R}$ for the propagator that maps each structure $\mathfrak{A}$ to a structure equal to $\mathfrak{A}$ except that it interprets $Q(\overline{d})$ and $R(\overline{d})$ both as $\mathrm{lub}_{\leq_p}(Q(\overline{d})^{\mathfrak{A}}, R(\overline{d})^{\mathfrak{A}})$. We use $P_{Q\equiv R}$ for the propagator $\bigtimes_{\{\overline{d}\in A^n\}} P^{\overline{d}}_{Q\equiv R}$ where A is the domain and n is the arity of Q and R. Furthermore, we use $C_{Q\equiv R}$ for the mapping that sends $\mathfrak{A}$ to

$$\begin{cases} \Diamond & \text{if } P_{Q\equiv R}(\mathfrak{A}) = \mathfrak{A} \\ \left(\bigtimes_{\{\overline{d}\in A^n \mid Q(\overline{d})^{\mathfrak{A}} \neq R(\overline{d})^{\mathfrak{A}}\}} P^{\overline{d}}_{Q\equiv R}, C_\Diamond\right) & \text{otherwise} \end{cases}$$

Definition 4.10. Assume (P,C) and (P',C') are explaining propagators. We define the following explaining propagators:

- **Product** of explaining propagators: $(P,C)\times(P',C') = (P\times P', C\times C')$ where

$$C\times C' : \mathfrak{A} \mapsto \begin{cases} C(\mathfrak{A})\times C'(\mathfrak{A}) & \text{if both } (P,C) \text{ and } (P',C') \\ & \text{explain propagation from } \mathfrak{A} \\ \Diamond & \text{otherwise} \end{cases}$$

- **Projection** of an explaining propagator: $\pi_\delta(P,C) = (\pi_\delta P, \pi_\delta C)$, where

$$\pi_\delta C : \mathfrak{A} \mapsto \begin{cases} \Diamond & \text{if } \mathfrak{A} \text{ is inconsistent or } C(\mathfrak{A}|_\delta) = \Diamond \\ \Diamond & \text{if } \mathfrak{A} \text{ is two-valued on } \delta \text{ and } s(P)(\mathfrak{A}|_\delta) = \varnothing \\ \pi_\delta(C(\mathfrak{A})) & \text{otherwise} \end{cases}$$

- **Selection** of an explaining propagator: $\sigma_{Q\equiv R}(P,C) = (P,C)\times(P_{Q\equiv R}, C_{Q\equiv R})$.

Proposition 4.11. *The mappings in Definition 4.10 indeed define explaining propagators.*

Proof. The proof is similar for all cases. We only give the proof for projection. The proof is by induction on the structure of C. First assume that $C = C_\lozenge$. In this case, $\pi_\delta(P,C) = (\pi_\delta P, C_\lozenge)$, which is indeed an explaining propagator.

For the induction case, we can assume that for each $\mathfrak{A}$ with $C(\mathfrak{A}) \neq \lozenge$, $\pi_\delta C(\mathfrak{A})$ is an explaining propagator. We show that $\pi_\delta(P,C)$ is an explaining propagator. Choose some $\mathfrak{A}$ with $(\pi_\delta C)(\mathfrak{A}) \neq \lozenge$. Let (P',C') denote $(\pi_\delta C)(\mathfrak{A})$ and (P'',C'') denote $C(\mathfrak{A})$. From Definition 4.10, we know that $P' = \pi_\delta P''$.

First, we show that $\pi_\delta(P,C)$ explains propagation, i.e., that $\pi_\delta P(\mathfrak{A}) \leq_p P'(\mathfrak{A})$. We know that $P(\mathfrak{A}) \leq_p P''(\mathfrak{A})$ since (P,C) is an explaining propagator. It follows immediately from the definition of $\pi_\delta P$ that also $\pi_\delta P(\mathfrak{A}) \leq_p \pi_\delta P''(\mathfrak{A}) = P'(\mathfrak{A})$.

We now show that $\pi_\delta(P,C)$ only derives consequences, i.e., that $module(P') \models module(\pi_\delta P)$. We know that $module(P'') \models module(P)$. From the definition of the semantics of modular systems, it follows that then also $\pi_\delta module(P'') \models \pi_\delta module(P)$. Furthermore, from Proposition 3.22, we know that $\pi_\delta module(P'') = module(\pi_\delta P'') = module(P')$ and $\pi_\delta module(P) = module(\pi_\delta P)$. The result then follows. □

A Conflict-Driven Learning Algorithm

The CDCL algorithm for SAT lies at the heart of most modern SAT solvers, and also many SMT solvers, ASP solvers, and others. We now give an algorithm scheme that generalizes the CDCL algorithm to modular systems.

Definition 4.12. The solver $cdl^{(P,C)}$ is obtained by modifying $ls^{(P,C)}$ as follows. Each time propagation leads to an inconsistent state, we update $\mathfrak{B}, \mathscr{P} \leftarrow \mathfrak{B}', \mathscr{P} \cup \{(P',C')\}$, where $(\mathfrak{B}',(P',C')) =$ HandleConflict$(\mathfrak{B},\mathscr{P})$, and HandleConflict is a function such that

(1) (P',C') is an explaining propagator that respects $\prec$,
(2) $\mathfrak{A} \leq_p \mathfrak{B}' <_p \mathfrak{B}$ (backjumping),
(3) $(\mathscr{P} \cup \{(P',C')\})(\mathfrak{B}') >_p \mathscr{P}(\mathfrak{B}')$ (learning),
(4) $module(\mathscr{P}) \subseteq module(P')$ (consequence)

After executing HandleConflict, it is optional to restart by re-setting $\mathfrak{B}$ to $\mathfrak{A}$.

Pseudocode for this solver can be found in Algorithm 4.

The intuition is that HandleConflict is some function that returns a state to backtrack to, and a new propagator to add to the set of propagators. This new propagator should, in the structure to which we backtrack, propagate something that was not propagated before. Thus, by analyzing the conflict, we obtain better information and avoid ending up in the same situation again.

Proposition 4.13. *Assume A is finite. If (P_E,C_E) is an E-explaining propagator that respects $\prec$, then $cdl^{(P_E,C_E)}$ is an E-solver.*

Sketch of the proof. Correctness of $cdl^{(P_E,C_E)}$ follows from correctness of $ls^{(P,C)}$ combined with the fourth condition for HandleConflict in Definition 4.12. The hardest thing to prove is termination of this algorithm in case restarts are involved. It can be seen that this algorithm terminates by the fact that after each conflict, by the third condition in HandleConflict, for at least one partial structure (namely for $\mathfrak{B}'$), strictly more is propagated by $(\mathscr{P} \cup \{(P',C')\})$ than by $\mathscr{P}$. Since the number of propagators is finite, there cannot be an infinite such sequence, hence only a finite number of conflicts can occur. □

The purpose of HandleConflct is to perform a conflict analysis analogous to that in standard CDCL. This procedure can be anything; in practice, it will depend on the form $\prec$-minimal propagators take and on the proof system used for these minimal propagators. Below, we present a sufficient restriction on $\prec$-minimal propagators to ensure that a procedure HandleConflict exists.

Algorithm 4 The solver $ls^{(P,C)}$

```
1:  function $S_p^P(\mathfrak{A})$
2:      $(\mathscr{P},\mathfrak{B},\mathscr{S}) \leftarrow (\{(P,C)\},\mathfrak{A},\varnothing)$
3:      while true do
4:          while At least one $(P^*,C^*) \in \mathscr{P}$ has $P^*(\mathfrak{B}) \neq \mathfrak{B}$ do
5:              Let $(P^*,C^*)$ be a $\prec$-minimal propagator with $P^*(\mathfrak{B}) \neq \mathfrak{B}$ in $\mathscr{P}$
6:              $\mathfrak{B} \leftarrow P^*(\mathfrak{B})$
7:              $\mathscr{P} \leftarrow \mathscr{P} \cup \{C^*(\mathfrak{B})\}$
8:          if $\mathfrak{B}$ is two-valued then
9:              $S \leftarrow S \cup \{\mathfrak{B}\}$
10:             $\mathscr{P} \leftarrow \mathscr{P} \cup \{(P_{check}^{-\{\mathscr{A}\}}, C_{\Diamond})\}$
11:         else if $\mathfrak{B}$ is inconsistent then
12:             if No choices were made then return S
13:             $(\mathfrak{B}',(P',C')) \leftarrow$ HandleConflict$(\mathfrak{B},\mathscr{P})$
14:             $(\mathfrak{B},\mathscr{P}) \leftarrow (\mathfrak{B}',\mathscr{P} \cup \{(P',C')\}$
15:         else
16:             choose $\mathfrak{B} \leftarrow \mathfrak{B}[Q(\bar{d}) : \mathbf{t}]$ for some $Q(\bar{d})$ with $Q(\bar{d})^{\mathfrak{B}} = \mathbf{u}$
```

Proposition 4.14. *Suppose that there exists a function F that takes as arguments two $\prec$-minimal explaining propagators (P_1,C_1) and (P_2,C_2) that respect $\prec$, and a partial structure $\mathfrak{B}$, and returns an explaining propagator (P,C) that respects $\prec$, such that the following hold.*

> *If $\mathfrak{A} <_p P_1(\mathfrak{B}) <_p P_2(P_1(\mathfrak{B}))$ and $\mathfrak{B} <_p P_2(\mathfrak{B}) <_p P_2(P_1(\mathfrak{B}))$, then $module(P_1) \times module(P_2) \subseteq module(P)$ and there exists a structure $\mathfrak{B}' \leq_p \mathfrak{B}$ such that $P(\mathfrak{B}') >_p P_2(P_1(\mathfrak{B}'))$.*

In that case, a procedure HandleConflict that satisfies the restrictions in Definition 4.12 exists.

Sketch of the proof. The idea is that it suffices to be able to combine $\prec$-minimal propagators since all propagations can (by iterated calls to the explanation mechanism) be explained in terms of these propagators. Furthermore, the above condition can be applied iteratively to combine more than two $\prec$-minimal propagators. □

The intuition for F is that it effectively analyses the source of a conflict found by a sequence of propagations. We want to be able to determine a minimum collection of points in the partial structure relevant to the conflict. For this, it suffices that we can take two $\prec$-minimal propagators and “resolve” them to obtain one with stronger propagation power. Observe that, if we assume that all $\prec$-minimal propagators have a representation as clauses, this function can be implemented by means of the standard resolution used in CDCL conflict analysis process. In general, other resolution mechanisms might be used. The chosen implementation for F essentially determines the *proof system* that will be used in the solver.

Iterated applications of F, starting from the last two propagators that changed state and working back to earlier propagators allow us to handle conflicts. Note that F is only defined on $\prec$-minimal propagators. However, the explanation mechanism in explaining propagators allows us to always reduce propagators to $\prec$-minimal propagators by means of calling the explanation method until a minimal propagator is obtained (this is possible since $\prec$ is a well-founded order).

5 Modular patterns

Sometimes, defining a propagator compositionally does not yield the best result. We identify three patterns for which we can define a better (more precise) propagator by exploiting a global structure. The first two optimizations consist of direct implementations for propagators for expressions in the algebra of modular systems that are not in the minimal syntax (for details, see, e.g., [49]). The third optimization is based on techniques that were recently used to nest different SAT solvers to obtain a QBF solver.

Disjunction of Modules. The *disjunction* of two modules is defined as $E_1 + E_2 = -(-E_1 \times -E_2)$. Such an operation can be used for instance in the context of a web-shop offering different shipping option. If the constraints related to each shipping option are specified by the shipping company itself, possibly in different languages, say in modules M_{S_i} for different shipping companies and the desires of the user are specified in a module M_U, then $M_U \times (M_{S_1} + \cdots + M_{S_n})$ represents a module in which the users desires are satisfied by at least one shipping company. This can be used to see if the web shop can satisfy the request or not.

Definition 5.1. Let P_1 and P_2 be an E_1-, respectively E_2-, propagator. We define a propagator

$$P_1 + P_2 : \mathfrak{A} \mapsto \mathrm{glb}_{\leq_p}(P_1(\mathfrak{A}), P_2(\mathfrak{A})).$$

Intuitively, this propagator only propagates what holds in both $P_1(\mathfrak{A})$ and $P_2(\mathfrak{A})$. As such, it indeed only derives consequences of the disjunction.

Proposition 5.2. *The following hold:*

- *$P_1 + P_2$ is a $-(-E_1 \times -E_2)$ propagator*
- *For each consistent partial structure $\mathfrak{A}$,*

$$(P_1 + P_2)(\mathfrak{A}) \geq_p -(-P_1 \times -P_2)(\mathfrak{A}).$$

Sketch of the proof. It is easy to see that $\mathscr{A} \models -(-E_1 \times -E_2)$ iff $\mathscr{A} \models E_1$ or $\mathscr{A} \models E_2$. The first point now follows directly from the definition of $P_1 + P_2$. The second point follows from Proposition 3.10 since $-(-P_1 \times -P_2)$ is a checker. □

Extended selection. It is also possible to allow expressions of the form $\sigma_\Theta E$ where Θ consists of expressions of the form $Q \equiv R$ or $Q \not\equiv R$ and propositional connectives applied to them (for semantics, see, e.g., [49]). Each such expression can be rewritten to the minimal syntax used in this paper, for instance $\sigma_{P \not\equiv Q} E$ is equivalent to $E \times -\sigma_{P \equiv Q} E$. By taking an entire such formula into account at once, more precise reasoning is possible.

Proposition 5.3. *Let P be a $\sigma_\Theta E$-propagator with Θ an expression as above. If $\Theta \models Q \equiv R$, then $\sigma_{Q \equiv R} P$ is also a $\sigma_\Theta E$ propagator and for all partial structures $\mathfrak{A}$, $(\sigma_{Q \equiv R} P)(\mathfrak{A}) \geq_p P(\mathfrak{A})$.*

Proof. Follows directly from the fact that in this case $\sigma_{Q \equiv R}(\sigma_\Theta E) = \sigma_\Theta E$. □

Proposition 5.3 states that we can use (symbolic) equality reasoning on Θ to derive more consequences. The following example illustrates the extra propagation power Proposition 5.3 yields.

Example 5.4. Consider the module $\sigma_{(Q \not\equiv R \vee R \equiv U) \wedge (Q \equiv R)} E$. It is easy to see that $R \equiv U$ is a consequence of the selection expression in this module. As such, Proposition 5.3 guarantees that we are allowed to improve propagators, to also propagate equality between Q and R. ▲

Improved Negation. Janhunen et al. [26] recently defined a solver that combines two SAT solvers. The essence of their algorithm can be translated into our theory as follows. Let τ and δ be vocabularies, E a $\tau \cup \delta$-module and S an E-solver. Assume that there is a procedure `Explain` such that for each two-valued τ-structure $\mathscr{A}$ such that $S(\mathscr{A}) \neq \varnothing$, $\mathfrak{A} = \texttt{Explain}(\mathscr{A})$ is a partial τ-structure such that

- $\mathfrak{A} \leq_p \mathscr{A}$
- For each two-valued τ structure $\mathscr{B} \geq_p \mathfrak{A}$, $S(\mathscr{B}) \neq \varnothing$.

Thus, `Explain` explains *why* a certain module is satisfiable. Given this, Janhunen et al. defined an explaining propagator P for $-\pi_\tau E$. If $\mathfrak{A}$ is two-valued on τ, P calls $S(\mathfrak{A}|_\tau)$. If the result of this call is not empty, it propagates a conflict and generates an explanation using $\texttt{Explain}(\mathfrak{A}|_\tau)$; this explanation is a propagator that performs unit propagation for a clause, the negation of the returned partial interpretation (see [26] for details). Otherwise, P maps $\mathfrak{A}$ to itself.

This idea has been generalized to work for arbitrary QBF formulas [7]. It forms the essence of many SAT-based QBF algorithms [44, 56, 24, 28]. Janhunen et al. [26] improved this method by introducing a notion of an *underapproximation*. That is, instead of using an E-solver S, they use an $\bar{E}$-solver $\bar{S}$, where $\bar{E}$ is some module derived from E. This allows them to run $\bar{S}$ *before* $\mathfrak{A}$ is two-valued on τ. The module $\bar{E}$ is constructed in such a way that from runs of $\bar{S}$, a lot of information can already be concluded without having a two-valued $\mathfrak{A}$. Researching how these underapproximations generalize to modular systems is a topic for future work.

6 Conclusion and Future Work

In this paper, we defined general notions of *solvers* and *propagators* for modular systems. We extended the algebra of modular systems to *modular propagators* and showed how to build solvers from propagators and vice versa. We argued that our notion of propagator generalizes notions from various domains. Furthermore, we added a notion of *explanations* to propagators. These explanations generalize concepts from answer set programming, constraint programming, linear programming and more. We used these explanations to build *learning solvers* and discussed how learning solvers can be extended with a conflict analysis method, effectively resulting in a generalization of CDCL to other learning mechanisms and hence also to other proof systems. Finally, we discussed several patterns of modular expressions for which more precise propagation is possible than what would be obtained by creating the propagators following the compositional rules.

The main contribution of the paper is that we provide an abstract account of propagators, solvers, explanations, learning and conflict analysis, resulting in a theory that generalizes many existing algorithms and allows integration of technology of different fields. Our theory allows one to build *actual solvers* for modular systems and hence provides an important foundation for the practical usability of modular systems.

Several topics for future work remain. While the current theory provides a strong foundation, an *implementation* is still needed to achieve practical usability. We intend to research more *patterns* for which improved propagation is possible, and generalize the aforementioned *underapproximations* to our framework. The Algebra of Modular Systems has been extended with a *recursion* operator (see for instance [51]); the question "What are good propagators for this operator?" remains an open challenge.

References

[1] Slim Abdennadher and Thom W. Frühwirth. Integration and optimization of rule-based constraint solvers. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, volume 3018 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2003.

[2] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In Cabalar and Son [10], pages 54–66.

[3] Franz Baader, Silvio Ghilardi, and Cesare Tinelli. A new combination procedure for the word problem that generalizes fusion decidability results in modal logics. *Inf. Comput.*, 204(10):1413–1452, 2006.

[4] Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 35–40. Professional Book Center, 2005.

[5] Marcello Balduccini, Yuliya Lierler, and Peter Schüller. Prolog and ASP inference under one roof. In Cabalar and Son [10], pages 148–160.

[6] Frédéric Benhamou. Heterogeneous constraint solving. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96, Aachen, Germany, September 25-27, 1996, Proceedings*, volume 1139 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 1996.

[7] Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Solving QBF instances with nested SAT solvers. In Adnan Darwiche, editor, *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016.*, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016.

[8] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Luigi Barone. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2013, Cancun, Mexico, June 20-23, 2013*, pages 1037–1044. IEEE, 2013.

[9] Sebastian Brand and Roland H. C. Yap. Towards "propagation = logic + control". In Sandro Etalle and Mirosław Truszczyński, editors, *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4079 of *LNCS*, pages 102–116. Springer, 2006.

[10] Pedro Cabalar and Tran Cao Son, editors. *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *LNCS*. Springer, 2013.

[11] Shelvin Chand and Markus Wagner. Fast heuristics for the multiple traveling thieves problem. In Tobias Friedrich, Frank Neumann, and Andrew M. Sutton, editors, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, pages 293–300. ACM, 2016.

[12] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A gentle non-disjoint combination of satisfiability procedures. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2014.

[13] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A polite non-disjoint combination method: Theories with bridging functions revisited. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 419–433. Springer, 2015.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[15] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[16] G Dantzig, R Fulkerson, and S Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[17] Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion in the presence of function symbols using constraint programming. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, pages 1068–1075. IEEE Computer Society, 2013.

[18] Broes De Cat, Marc Denecker, Maurice Bruynooghe, and Peter J. Stuckey. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. (JAIR)*, 52:235–286, 2015.

[19] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal,*

September 20-24, 2009, Proceedings, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.

[20] Pascal Fontaine. Combinations of theories for decidable fragments of first-order logic. In Silvio Ghilardi and Roberto Sebastiani, editors, *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, volume 5749 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2009.

[21] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[22] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

[23] Michael Gelfond, Veena S. Mellarkod, and Yuanlin Zhang. Systems integrating answer set programming and constraint programming. In Marc Denecker, editor, *Second Workshop on Logic and Search, 2008*, pages 145–152. ACCO, 2008.

[24] Alexandra Goultiaeva, Martina Seidl, and Armin Biere. Bridging the gap between dual propagation and cnf-based QBF solving. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 811–814. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[25] Martin James Green and Christopher Jefferson. Structural tractability of propagated constraints. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2008.

[26] Tomi Janhunen, Shahab Tasharrofi, and Eugenia Ternovska. SAT-to-SAT: Declarative extension of SAT solvers with new propagators. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 978–984. AAAI Press, 2016.

[27] Mikolás Janota, William Klieber, Joao Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016.

[28] Mikolás Janota and Joao Marques-Silva. Solving QBF by clause selection. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 325–331. AAAI Press, 2015.

[29] Christopher Jefferson, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Implementing logical connectives in constraint programming. *Artif. Intell.*, 174(16-17):1407–1429, 2010.

[30] Antonina Kolokolova, Yongmei Liu, David G. Mitchell, and Eugenia Ternovska. On the complexity of model expansion. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 447–458. Springer, 2010.

[31] Olivier Lhomme. An efficient filtering algorithm for disjunction of constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *LNCS*, pages 904–908. Springer, 2003.

[32] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2004.

[33] Yuliya Lierler and Miroslaw Truszczynski. On abstract modular inference systems and solvers. *Artif. Intell.*, 236:65–89, 2016.

[34] Maarten Mariën. *Model Generation for ID-Logic*. PhD thesis, Department of Computer Science, KU Leuven, Belgium, February 2009.

[35] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin

Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[36] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[37] David G. Mitchell. Hard problems for CSP algorithms. In Jack Mostow and Chuck Rich, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 398–405. AAAI Press / The MIT Press, 1998.

[38] David G. Mitchell and Eugenia Ternovska. Clause-learning for modular systems. In Francesco Calimeri, Giovambattista Ianni, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *Lecture Notes in Computer Science*, pages 446–452. Springer, 2015.

[39] Tobias Müller and Jörg Würtz. Constructive disjunction in oz. In *WLP*, pages 113–122, 1995.

[40] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[41] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[42] Max Ostrowski and Torsten Schaub. ASP modulo CSP: The clingcon system. *TPLP*, 12(4–5):485–503, 2012.

[43] Markus N. Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In Roope Kaivola and Thomas Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, pages 136–143. IEEE, 2015.

[44] Darsh P. Ranjan, Daijue Tang, and Sharad Malik. A comparative study of 2QBF algorithms. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, Online Proceedings*, 2004.

[45] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217. ACM Press, 1990.

[46] Roberto Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.

[47] Peter J. Stuckey. Lazy clause generation: Combining the power of SAT and CP (and mip?) solving. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, pages 5–9. Springer, 2010.

[48] Shahab Tasharrofi and Eugenia Ternovska. A semantic account for modularity in multi-language modelling of search problems. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2011.

[49] Shahab Tasharrofi and Eugenia Ternovska. Three semantics for modular systems. In Sébastien Konieczny and Hans Tompits, editors, *Proceedings of the Fifteenth International Workshop on Non-monotonic Reasoning (NMR-14)*, number RR-1843-14-01 in INFSYS, pages 59–68. Institut fur Informationssysteme, 2014.

[50] Shahab Tasharrofi and Eugenia Ternovska. Modular systems: Semantics, complexity. In *Proceedings of HR workshop*, 2015.

[51] Eugenia Ternovska. Static and dynamic views on the algebra of modular systems. In *Proceedings of NMR*, 2016.

[52] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In *FroCoS*, pages 103–119, 1996.

[53] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[54] Markus Wagner. Stealing items more efficiently with ants: A swarm intelligence approach to the travelling thief problem. In Marco Dorigo, Mauro Birattari, Xiaodong Li, Manuel López-Ibáñez, Kazuhiro Ohkura,

Carlo Pinciroli, and Thomas Stützle, editors, *Swarm Intelligence - 10th International Conference, ANTS 2016, Brussels, Belgium, September 7-9, 2016, Proceedings*, volume 9882 of *Lecture Notes in Computer Science*, pages 273–281. Springer, 2016.

[55] Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In Günther Görz and Steffen Hölldobler, editors, *KI-96: Advances in Artificial Intelligence, 20th Annual German Conference on Artificial Intelligence, Dresden, Germany, September 17-19, 1996, Proceedings*, volume 1137 of *Lecture Notes in Computer Science*, pages 377–386. Springer, 1996.

[56] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Proceedings*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.