On Conflict-Driven Reasoning

Position Paper

Maria Paola Bonacina*
Dipartimento di Informatica, Università degli Studi di Verona
Strada Le Grazie 15
Verona I-37134, Italy, EU
mariapaola.bonacina@univr.it

ABSTRACT

Automated formal methods and automated reasoning are interconnected, as formal methods generate reasoning problems and incorporate reasoning techniques. For example, formal methods tools employ reasoning engines to find solutions of sets of constraints, or proofs of conjectures. From a reasoning perspective, the expressivity of the logical language is often directly proportional to the difficulty of the problem. In propositional logic, Conflict-Driven Clause Learning (CDCL) is one of the key features of state-of-the-art satisfiability solvers. The idea is to restrict inferences to those needed to explain conflicts, and use conflicts to prune a backtracking search. A current research direction in automated reasoning is to generalize this notion of *conflict-driven satisfiability* to a paradigm of *conflict-driven reasoning* in first-order theories for satisfiability modulo theories and assignments, and even in full first-order logic for generic automated theorem proving. While this is a promising and exciting lead, it also poses formidable challenges.

CCS CONCEPTS

• **Theory of computation → Automated reasoning**;

KEYWORDS

Theorem proving, Satisfiability modulo theory, Model building, Theory combination, Equality reasoning, Satisfiability modulo assignment

ACM Reference format:
Maria Paola Bonacina. 2017. On Conflict-Driven Reasoning. In *Proceedings of Automated Formal Methods Workshop, Menlo Park, CA, USA, May 2017 (AFM 2017),* 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

*This paper was written while the author was visiting the School of Computer Science of the University of Manchester, in Manchester, England, UK, and the Computer Science Laboratory of SRI International, in Menlo Park, California, USA: support from both institutions is greatly appreciated. The research and the visits were funded in part by grants "CooperInt 2016" and "Ricerca di base 2015" both from the Università degli Studi di Verona.

AFM 2017, Menlo Park, CA, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automated reasoning and automated formal methods, for the specification, analysis, verification, or synthesis of systems, are interconnected, because logic is the calculus of computation, and reasoning about computer systems [4, 18, 50] may be more amenable to automation than other less machine-oriented domains.

In automated reasoning, problems are typically presented as *validity* queries. A validity query asks whether a conjecture φ follows from a set H of assumptions. Since mechanical methods preferably work refutationally, a validity query is usually reformulated in refutational form, by asking whether $H \cup \{\neg\varphi\}$ is *unsatisfiable*. Assumptions, conjectures, constraints are logical formulæ that express properties of an object of study, such as a system, a program, a data type, a circuit, a protocol, a mathematical structure. As mechanical methods usually adopt clausal form, $H \cup \{\neg\varphi\}$ is transformed into a set S of clauses, where a set is interpreted as a conjunction. Alternatively, one may be interested in knowing whether a constraint φ can be added to a set of constraints H so that $H \cup \{\varphi\}$ is still *satisfiable*. Once $H \cup \{\varphi\}$ has been turned into a set of clauses S, the problem is the same, namely determining whether S has a model or is unsatisfiable.

The answer is either a proof $S \vdash \Box$ that S is inconsistent, hence unsatisfiable, where $\Box$ is the empty clause, which represents a contradiction, or else a model of S. If the problem was originally formulated as a validity query, a proof means that φ follows from H, while a model represents a counter-example. If the problem was originally formulated as a satisfiability query, a model represents a solution, while a proof means that there is no solution.

Depending on the logic, these queries may be *decidable* (validity and satisfiability are both decidable), *semi-decidable* (validity is semi-decidable, satisfiability is not semi-decidable), or *undecidable* (validity and satisfiability are both undecidable).

An automated reasoning method or strategy is typically defined by an *inference system* and a *search plan*. The inference system is a set of inference rules, and the search plan is an algorithm equipped with heuristics to control the application of the inference rules. The application of an inference rule moves the system from one *state* of the *derivation* to the next.

When the problem is decidable, an automated reasoning strategy is expected to be a *decision procedure*, that requires it to be *sound*, *complete*, and *terminating*, returning a proof whenever the input is unsatisfiable and a model whenever the input is satisfiable. When the problem is semi-decidable, an automated reasoning strategy is expected to be a *semi-decision procedure*, that requires it to be *sound* and *complete*, returning a proof whenever the input is unsatisfiable. In practice, however, instances of decidable problems

may be too difficult for the available computational resources, or complete strategies may be too onerous, so that regardless of decidability, automated reasoning tools may return either a proof, or a model, or a "don't know" answer. The degree to which "don't know" answers may be tolerated depends on the application.

Similar to other subfields of artificial intelligence, problems in automated reasoning involve so much knowledge, that it is often too cumbersome or too inefficient to encode all of it in H and φ, hence in the set S of clauses. Therefore, a common paradigm is to reason *modulo* T, seeking proofs modulo T and restricting the attention to T-models. For example, if T is the theory of equality, we have *equational reasoning*, where the axioms of the theory of equality are built into the inference system. T may also contain additional axioms stating properties of symbols other than equality, such as *associativity* and *commutativity* of function symbols.

If T is a theory such that T-satisfiability is decidable, reasoning modulo T is known as *satisfiability modulo a theory* (SMT), and the knowledge about T is built in the algorithm implementing the decision procedure for T-satisfiability. For example, if T is the quantifier-free fragment of the theory of equality, a *congruence closure* algorithm decides the T-satisfiability of a set of equalities and inequalities (cf. Chapter 9 of [18]). An algorithm that decides the T-satisfiability of a set of ground literals in the signature of T, or T-literals for short, is called a *T-satisfiability procedure*. An algorithm that decides the T-satisfiability of a quantifier-free formula in the signature of T, or quantifier-free T-formula for short, is called a *T-decision procedure*. A quantifier-free formula φ is satisfiable if and only if its existential closure $\exists \bar{x}.\varphi$ is satisfiable, where $\bar{x}$ are all the variables in φ. Then, $\exists \bar{x}.\varphi$ is satisfiable if and only if $\hat{\varphi}$ is satisfiable, where $\hat{\varphi}$ is φ with all variables replaced by Skolem constants. Thus, the problem of deciding the T-satisfiability of a quantifier-free T-formula is equivalent to that of deciding the T-satisfiability of a ground T-formula.

A reasoning method is *model-based*, if the state of a derivation contains a representation of a candidate partial model, and inference and search for a model are intertwined, as inferences build and transform the model while the model drives the inferences [8]. In a model-based strategy, a *conflict* arises if one of the clauses of S is false in the current candidate model. The strategy is deemed *conflict-driven*, if it uses inferences to *explain* and *solve* the conflict repairing the model.

The rest of this paper is organized as follows. Section 2 is a necessarily incomplete overview of the state of the art in conflict-driven methods. Section 3 advertises two recent conflict-driven methods: *Semantically-Guided Goal-Sensitive reasoning* (SGGS), for full first-order logic [15–17], and *Conflict-Driven Satisfiability* (CDSAT), for satisfiability modulo a *generic* combination of theories, and for a new class of problems called *satisfiability modulo assignments* (SMA) [9, 10].

2 CONFLICT-DRIVEN METHODS

The conflict-driven paradigm was pioneered by *Conflict-Driven Clause Learning* (CDCL) for propositional satisfiability [41, 42, 45]. In conflict-driven methods that incorporate a CDCL-based SAT-solver as a black-box, the conflict-driven reasoning is propositional, even if the method applies to a more general logic. These methods are covered in Section 2.1. Other conflict-driven methods generalize the conflict-driven principle to satisfiability modulo a theory. These methods are treated in Section 2.2.

2.1 Conflict-driven propositional reasoning

This section summarizes methods whose conflict-driven component is restricted to propositional logic. In other words, the candidate model and the conflict-driven inferences are propositional, with an abstraction function mapping first-order atoms to propositional atoms.

Satisfiability (SAT) is the problem of deciding the satisfiability of a set S of clauses in propositional logic. The DPLL (Davis-Putnam-Logemann-Loveland) procedure for SAT [20, 22, 23, 58] represents a candidate partial model by a *sequence* of literals, called a *trail*, and named M. The trail represents the partial model, also called M, where all literals on the trail are true. If a literal L is in M, its complement $\neg L$ is false in M. If neither L nor $\neg L$ is in M, both literals are *undefined*.

The procedure starts by putting in M all input unit clauses, and *propagating* their consequences in the form of *implications* and *conflicts*, an activity called *Boolean clausal propagation* (BCP). For implications, assume that all literals of a clause $C \in S$ but one, say L, are false in M. Then literal L is an *implied literal*, and is added to M with C as *justification*, because extending M with L is the only way to satisfy C. The discovery of an implied literal can be seen in terms of inferences as the result of a sequence of *unit resolution* steps using the literals in M as unit premises. For conflicts, whenever all literals of a clause $C \in S$ are false in M, a *conflict* emerges with C as the *conflict clause*. The discovery of a conflict can be seen in terms of inferences as the result of a sequence of unit resolution steps yielding the empty clause $\Box$.

When no more propagations are possible and in the absence of a conflict, the procedure *decides* that a literal L is true by adding it to M. A literal added to M by a decision is termed a *decided literal*. A decision is merely a guess to advance the search. This operation is also termed *case analysis* or *splitting*, because for a literal L there are two cases, as L is either true or false. After every decision, the procedure applies BCP to discover more implied literals or a conflict. When a conflict arises, the procedure backtracks chronologically, undoing the latest decision and all the propagations that depend on it. The procedure returns "satisfiable" if $M \models S$, and "unsatisfiable" if there is a conflict and no decision to undo.

The Conflict-Driven Clause Learning (CDCL) procedure [41, 42, 45] inherits from the DPLL procedure the representation of the candidate model, BCP, and decisions. It also maintains the initialization of the trail with input unit clauses, said to be stored at level 0. Then, every decision opens a subsequent *decision level* in the trail: the decision level numbered n contains the n-th decided literal in the current trail and all implied literals discovered by BCP as a consequence. The CDCL procedure behaves in a markedly different manner when a conflict arises.

Suppose that C is a conflict clause and contains a literal L, such that $\neg L$ is in M with justification D. Then propositional resolution is applied to resolve C and D upon L and $\neg L$. This inference is said to *explain* the conflict, as L is false because $\neg L$ is true, and $\neg L$ is true, because D is in S and all other literals of D appear negated in

M. The generated resolvent is still a conflict clause, since all other literals in C and D are false in M. A resolvent is a logical consequence of S and can be added to S as a *learned clause* or *lemma*. Such a step is called *learning*. In practice, S may be huge and the procedure learns one clause per conflict. How many resolutions to do and which resolvent to learn is a heuristic choice.

The *first unique implication point* (1UIP) heuristic prescribes to perform resolution until an *asserting* conflict clause C is generated. Assume that n is the number of the current decision level. A conflict clause $C = L_1 \vee \ldots \vee L_k \vee \ldots \vee L_m$ is asserting, or is an *assertion clause*, if for only one of its literals, say L_k, the complement appears in decision level n of the trail M. For all other literals L_j in C, with $1 \leq j \leq k-1$ or $k+1 \leq j \leq m$, the complement appears in a decision level smaller than n. As a special case, $\neg L_j$ appears in decision level 0, if $\neg L_j$ is a unit clause in the input set S. The 1UIP heuristic lets the procedure *learn* clause C and *backjump* to the smallest decision level where L_k is undefined and all other literals of C are false. Note that L_k is undefined in every level smaller than n. This smallest decision level is guaranteed to exist, because C is a conflict clause, and therefore for all its literals the complement is on the trail at some level. If this smallest decision level is $n-1$, backjumping reduces to backtracking. If this smallest decision level is 0, the procedure backjumps to a state where only input unit clauses are on the trail. After backjumping, the procedure adds L_k to the trail, so that C is satisfied and the conflict is solved. The CDCL procedure returns "satisfiable" if $M \models S$, and "unsatisfiable" if there is a conflict at level 0.

Satisfiability modulo theory (SMT) is the problem of deciding the satisfiability of a set S of *ground* clauses modulo a theory T. The DPLL(T) paradigm for SMT integrates a theory solver, or T-solver for short, and a CDCL-based SAT-solver [48]. Since the SAT-solver accepts only propositional clauses, first-order ground atoms are abstracted to propositional variables, sometimes called *proxy variables*. The interface between SAT-solver and T-solver consists essentially of two rules. The *T-conflict* rule detects that a set of literals $L_1, \ldots, L_k$ in M is inconsistent in T. The *T-propagation* rule discovers that a set of literals $L_1, \ldots, L_k$ in M derives in T a literal L, and adds L to M with the *T-lemma* $\neg L_1 \vee \ldots \vee \neg L_k \vee L$ as justification. DPLL(T) features *no creation of new atoms*, meaning atoms that do not appear in S. Indeed, the T-propagation rule requires that the atom of L occurs in the existing set of clauses, and clauses learned by CDCL are propositional resolvents made of input atoms.

If T is a combination of theories $T_1, \ldots, T_n$, the T_i-solvers need to agree on the interpretation of whatever is shared among the theories. If they are *disjoint*, meaning that they do not share function or predicate symbols other than equality, the theory solvers need to agree on the cardinalities of the domains for shared sorts and on an arrangement of shared constant symbols, that tells which are equal and which are not.

The *equality sharing* method is the standard approach to this combination problem (cf. [46, 47] and Chapter 10 of [18]). It requires the theories to be *stably infinite*, so that the common cardinality of the shared domains can be implicitly assumed to be countably infinite. An arrangement is computed by having each T_i-solver propagate any disjunction of equalities $c_1 \simeq d_1 \vee \ldots \vee c_n \simeq d_n$ between shared constants that is entailed in T_i by the T_i-subproblem. The case analysis for these disjunctions, as well as for any other disjunction generated by a T_i-solver, is entrusted to the SAT-solver.

If T is a combination of theories $T_1, \ldots, T_n$, the T-solver integrated in DPLL(T) is a combination of the T_i-solvers by equaliy sharing, and the notion that a disjunction $c_1 \simeq d_1 \vee \ldots \vee c_n \simeq d_n$ is handled by the SAT-solver is termed *splitting on demand* [3, 40]. The DPLL(T) framework is extended to allow the generation of a finite number of "new" atoms, namely the proxy variables for the equalities $c_1 \simeq d_1, \ldots, c_n \simeq d_n$.

Another way to implement equality sharing is *model-based theory combination* (MBTC) [25, 53]. It assumes that the T_i-solvers build partial T_i-models. Then, each T_i-solver propagates, by adding it to M, any equality $s \simeq t$ between ground terms that is true in the current candidate T_i-model, rather than entailed (disjunctions of) equalities between shared constants. Such an equality $s \simeq t$ may cause a conflict, precisely because it is not necessarily a logical consequence in T_i of the T_i-subproblem. If this happens, backjumping retracts it. Also MBTC *does not generate new atoms*, because the propagation of an equality $s \simeq t$ is allowed only if s and t appear in the existing set of clauses. MBTC applies mostly to fragments of arithmetic, where domain of interpretation and interpretation of theory symbols are fixed by an intended model (e.g., the integers), and algorithms that can update the candidate partial model after a conflict are known [25, 30].

MBTC is an approach to the implementation of equality sharing in the context of an SMT-solver built on top of a CDCL-based SAT-solver. Thus, conflicts are still handled and reasoned about in propositional logic. However, with its notion of allowing the propagation of equalities that are true in a current candidate partial theory model, but not necessarily in all models, MBTC prepares the ground for conflict-driven theory reasoning.

MBTC is applied also in the DPLL($\Gamma + T$) reasoning engine that integrates an *ordering-based inference system* Γ for first-order logic with equality in the DPLL(T) framework [12, 13, 24]. An ordering-based inference system assumes a *well-founded* ordering on terms, literals, and clauses, and comprises *expansion* inference rules, such as ordered resolution, ordered paramodulation, and superposition, and *contraction* inference rules, such as subsumption and simplification. The well-founded ordering is used to define the contraction rules and to restrict the expansion rules. Equipped with a fair search plan, such an inference system provides (1) a semi-decision procedure for validity in first-order logic with equality, and (2) T-satisfiability procedures for the quantifier-free fragments of the theory of equality and of several theories of data structures [1, 2, 6, 7], including arrays with or without extensionality.

DPLL($\Gamma + T$) is designed to determine the T-satisfiability of sets of clauses in the form $S = P \uplus R$, where T is a combination of theories $T_1, \ldots, T_n$, P is a set of *ground* clauses with occurrences of T-symbols, and R is a set of *non-ground* clauses where T-symbols do *not* occur. Variables in non-ground clauses are implicitly universally quantified. The set R may be the axiomatization of a theory for which we do not have a built-in satisfiability procedure. Thus, this kind of problem is more general than deciding the T-satisfiability of a set of ground clauses. The idea is to use the generic

inference system Γ to reason about the axiomatized theory, precisely because Γ offers complete quantifier reasoning, since it is refutationally complete for first-order logic with equality.

DPLL(Γ+T) integrates Γ into DPLL(T) by letting it use R-literals in M, including those propagated by MBTC, as premises of Γ-inferences. Since these literals may be withdrawn upon backjumping, they are memorized in clauses as *hypotheses*, and DPLL(Γ+T) works with *hypothetical clauses*. Conclusions of Γ-inferences inherit the hypotheses of their parents. When backjumping removes literals from M, the hypothetical clauses that depend on them are also removed.

Integrating an ordering-based inference system with a solver that performs a backtracking search presents both difficulties and opportunities. A difficulty is that one needs to prevent the unsound situation where a clause C is deleted by subsumption or simplification with a clause D and then D is removed upon backjumping. DPLL(Γ + T) solves this problem by adapting the contraction inference rules of Γ for hypothetical clauses in such a way that C is deleted, if D cannot be removed by backjumping before C, and only *disabled* otherwise. While deletion is final, a disabled clause C will be re-enabled, if D is removed by backjumping.

A distinctive opportunity is the possibility of allowing *speculative inferences*: the user can tentatively add to the set of clauses an arbitrary clause. The system will search for a model that satisfies both the input set S and the speculatively added clauses. DPLL(Γ+T) keeps track of clauses added by speculative inferences in the trail M, and if such a clause causes an inconsistency, it will be retracted upon backjumping. In this way, the speculative inferences are *reversible*. The crux is to add clauses that may induce termination on satisfiable inputs, such as equations that limit term depth by rewriting: if S is satisfiable, it may happen that the search for a model of S does not terminate, but the search for a model of S that also satisfies the speculatively added clauses terminates. DPLL(Γ+T) is (1) a semi-decision procedure for validity of generic problems in the form $S = P \uplus R$, and (2) a T-decision procedure with *speculative inferences* for problems $S = P \uplus R$ that satisfy additional hypotheses. Examples include *axiomatizations of type systems* [13].

A feature of DPLL(Γ + T) is that it applies each reasoner to handle the part of the problem that it is best for: DPLL(T) deals with ground clauses, while Γ sees non-ground R-clauses and ground unit R-clauses in M. The two engines communicate through M, making DPLL(Γ+T) *model-based*. However, the conflict-driven part is propositional as in DPLL(T). We consider next methods that lift conflict-driven reasoning to the theory level.

2.2 Conflict-driven theory reasoning

In conflict-driven theory reasoning, the mechanisms to *explain* a conflict, *learn* a lemma, and *solve* the conflict, work within the T-solver itself, and not only at the propositional level in the SAT-solver. In other words, the T-solver implements a *conflict-driven T-satisfiability procedure*. Such procedures exist for linear real arithmetic [21, 38, 44], linear integer arithmetic [36, 53, 55], non-linear arithmetic [37], and floating-point binary arithmetic [31].

Some progress has been made towards a conflict-driven T-satisfiability procedure for the theory of *arrays with extensionality* [19], by developing the notion of *lemmas on demand* [29]. The idea of *lemmas on demand* is that a theory solver should generate only theory lemmas that *explain* why some contents of the trail M is inconsistent with respect to the theory. In other words, theory propagation should be model-based and conflict-driven. In propositional logic, lemmas on demand is the same as CDCL, with propositional resolvents as lemmas.

Although there are decision procedures for the theory of *arrays with extensionality* [52], SMT-solvers often reason about it by reading the theory axioms as part of the input set S, and heuristically instantiating the universally quantified variables in the theory axioms. In this regard, the difference of approach between SMT-solvers and generic theorem provers that instantiate by unification the universally quantified variables in the theory axioms is less dramatic than commonly thought.

Of greater relevance to this analysis is the difference between generating potentially all lemmas, as in a saturation process, and generating lemmas in a conflict-driven manner. The decision procedure with lemmas on demand features rules that propagate read operations over arrays, and generate lemmas of the form $\neg L_1 \vee \ldots \vee \neg L_k \vee L$, where $L_1, \ldots, L_k$ are true and L is false in the current candidate model M, whereas L should be true according to the axioms of the theory [19]. The lemma reveals that M is not a theory model and tells why. Often lemmas are instances of axioms, so that lemmas on demand can be regarded as model-based conflict-driven axiom instantiation.

The next problem is how to get a *conflict-driven T-decision procedure*. Conflict-driven T-satisfiability procedures [21, 31, 36–38, 44] are not compatible in general with DPLL(T), and therefore one cannot get a conflict-driven T-decision procedure by plugging a conflict-driven T-satisfiability procedure into DPLL(T). A reason of incompatibility is precisely that DPLL(T) does not allow the creation of new atoms, whereas a conflict-driven T-satisfiability procedure may *explain* a conflict by generating a clause that contains *new* atoms [26]. Addressing this issue is a motivation for the design of MCSAT, that stands for *model-constructing satisfiability* [26]. MCSAT is a paradigm for *conflict-driven T-decision procedures* for satisfiability modulo a single generic theory T [26]. It has been instantiated to the combined theories of equality and linear real arithmetic [35], to non-linear integer arithmetic [34], and to the theory of bit-vectors [56].

MCSAT merges the propositional model of CDCL with the theory models of MBTC, by allowing the trail M to contain both literals and assignments of concrete values to free first-order variables. For example, the trail may contain a literal L, meaning the assignment $L \leftarrow true$, and assignments such as $x \leftarrow 3$. Therefore, the trail is viewed as carrying an *assignment* to represent a candidate partial model. Furthermore, MCSAT generalizes CDCL to any theory that can be equipped with clausal inference rules to *explain* theory conflicts. Thus, the existence of a *conflict-explanation inference mechanism* emerges as the key ingredient for a conflict-driven procedure. The possibility of learning a clause generated by the conflict-explanation inference, and using it to amend the candidate partial model follows.

The conflict-explanation inferences generate clauses that may contain *new* ground atoms in the signature of the theory, beyond

what is allowed by DPLL(T) even with splitting on demand. Assignments to first-order variables and new atoms are involved in decisions, propagations, conflict detections, and explanations, on a par with Boolean assignments and input atoms. This means that the conflict-driven T-satisfiability procedure is not integrated as a black-box satellite, but cooperates with the SAT-solver on the same level. The CDCL procedure itself is a conflict-driven T-satisfiability procedure where T is propositional logic.

For termination, MCSAT requires that new atoms come from a *finite basis*. A procedure that applies systematically the inference rules to enumerate all atoms in the finite basis would be too inefficient. The key point is that the inference rules are applied only to explain conflicts and amend the current partial assignment, so that the generation of new atoms is model-based and conflict-driven. In this sense, MCSAT is a faithful lifting of CDCL to SMT, with first-order inferences for theory explanation, beyond explanation by propositional resolution.

3 GENERAL CONFLICT-DRIVEN METHODS

While satisfiability in propositional logic is decidable, in first-order logic validity is semi-decidable and satisfiability is not even semi-decidable. Nonetheless, theorem-proving approaches often are conceived and understood first for propositional logic and then generalized to full first-order logic. Section 3.1 presents the main features of a method named SGGS that lifts CDCL to first-order logic [15–17]. An alternative approach is *conflict resolution* [33, 51]. Section 3.2 returns to SMT with a summary of an inference system named CDSAT, that generalizes MCSAT to *generic* combinations of theories [9, 10]. Section 3.3 discusses how also the SMT problem itself can be generalized to the SMA problem, for *satisfiability modulo assignments*.

3.1 SGGS

SGGS, or *Semantically-Guided Goal-Sensitive* reasoning, brings the conflict-driven style to first-order logic [15–17]. It is *simultaneously* first-order, *model-based*, *semantically-guided*, *goal-sensitive*, and *proof confluent*, a rare combination of features.

In first-order logic variables in clauses are implicitly universally quantified, atoms have infinitely many ground instances, and there are infinitely many interpretations, so that guessing truth values of atoms is too uninformed. SGGS adopts an *initial interpretation* I for *semantic guidance*, and employs *SGGS clause sequences* to represent first-order models. An SGGS clause sequence is a sequence of possibly constrained clauses with *selected literals*. A sequence Γ represents an interpretation $I[\Gamma]$, that is I modified to satisfy the selected literals in Γ. Thus, the SGGS clause sequence plays the role of the trail in CDCL, and *literal selection* is the first-order analogue of propositional decision.

Example 3.1. Assume that S includes the clauses $on(a,b)$, $on(b,c)$, $green(a)$, and $\neg green(c)$. If I is the all-negative interpretation, that makes all negative literals true, the SGGS-derivation starts with the SGGS clause sequence $\Gamma = on(a,b),\ on(b,c),\ green(a)$. In a unit clause its only literal is obviously selected. $I[\Gamma]$ is the interpretation that makes all positive literals false except $on(a,b)$, $on(b,c)$, and $green(a)$. If I is the all-positive interpretation, that makes all positive literals true, the SGGS-derivation starts with $\Gamma = \neg green(c)$, and $I[\Gamma]$ is the interpretation that makes all negative literals false except $\neg green(c)$. CDCL would put all input unit clauses on the trail. SGGS assumes a guiding interpretation and modifies it lazily, because dealing with first-order models is much heavier than dealing with propositional models.

SGGS generalizes BCP to *first-order clausal propagation*. BCP is based on the symmetry of truth values in propositional logic: if L is true, $\neg L$ is false, and if L is false, $\neg L$ is true. Since variables in first-order literals are implicitly universally quantified, if L is true, $\neg L$ is false, but if L is false, we only know that at least one ground instance of $\neg L$ is true. To address this discrepancy, SGGS introduces *uniform falsity*: a first-order literal is *uniformly false*, if *all* its ground instances are false. This stronger notion of falsity restores the symmetry: if L is true, $\neg L$ is uniformly false, and if L is uniformly false, $\neg L$ is true.

Every literal in an SGGS clause sequence Γ must be either I*-true* (true in I) or I*-false* (uniformly false in I), so that it represents the truth value in I of all its ground instances. Every clause C in Γ must have a *selected literal* L: the clause with L selected is written $C[L]$. I-false literals are preferred for selection. An I-true literal is selected only in a clause whose literals are all I-true; such a clause is termed I*-all-true*. SGGS aims at building a model of S: if $I \models S$, the search halts immediately; if $I \not\models S$, SGGS seeks to build an $I[\Gamma]$ that differs from I where needed to satisfy S, hence the preference for I-false literals.

Example 3.2. $S = \{R(x, f(x)),\ \neg R(x,x),\ \neg R(x,y) \vee R(y,x)\}$ presents an irreflexive, symmetric, reachability relation R such that every state x has a successor $f(x)$. If I is all negative, SGGS builds the sequence $\Gamma = [R(x, f(x))],\ \neg R(x, f(x)) \vee [R(f(x), x)]$: in the second clause, which is binary, the positive literal is preferred for selection, denoted by the square brackets, because it is I-false. Then SGGS halts as $I[\Gamma] \models S$.

A first-order clause is a *conflict clause* if all its literals are uniformly false in $I[\Gamma]$. A literal L is uniformly false in $I[\Gamma]$, if all its ground instances appear negated among those that a preceding selected literal M makes true in $I[\Gamma]$. In this sense, L *depends* on M.

Example 3.3. Given $S = \{P(x),\ \neg P(x) \vee R(a,x),\ \neg P(x) \vee \neg R(x,b)\}$ and I all negative, SGGS builds the sequence $\Gamma = [P(x)],\ \neg P(x) \vee [R(a,x)],\ \neg P(a) \vee [\neg R(a,b)]$: literals $\neg P(x)$ and $\neg P(a)$ are uniformly false in $I[\Gamma]$, because $P(x)$ is selected; literal $\neg R(a,b)$ is uniformly false in $I[\Gamma]$, because $R(a,x)$ is selected; and the last clause in Γ is in conflict with $I[\Gamma]$. Note that this clause is I-all-true.

A first-order literal L is *implied*, with clause C as *justification*, if L is the only literal of C that is not uniformly false in $I[\Gamma]$. SGGS ensures that every I-all-true clause in Γ is either a conflict clause or the justification of its selected literal. To this end, SGGS uses *assignment functions* to keep track of the dependence of I-true literals on I-false selected literals: an I-all-true clause whose literals are all assigned to I-false selected literals is a conflict clause; an I-all-true clause whose literals, except the selected one, are assigned, is a justification.

Example 3.4. Continuing Example 3.3, literals $\neg P(x)$ and $\neg P(a)$ are assigned to $[P(x)]$; literal $\neg R(a,b)$ is assigned to $[R(a,x)]$; and the last clause in Γ is in conflict with $I[\Gamma]$ as all its literals are assigned.

All SGGS clause sequences in the above examples are generated by applications of the *SGGS-extension* inference rule, that adds to the sequence an instance E of a clause $C \in S$ and selects one of its literals. The instance E is built in order to capture the ground instances of C such that $I[\Gamma] \not\models C$, so that the resulting sequence (e.g., ΓE) will satisfy them.

Example 3.5. In Example 3.2, clause $\neg R(x, f(x)) \vee [R(f(x), x)]$ is an instance of input clause $\neg R(x, y) \vee R(y, x)$. SGGS generates it by unifying literal $\neg R(x, y)$ in this input clause with the selected literal $[R(x, f(x))]$ which is already on the trail. Recall that every first-order clause has its own variables. Let us rename the variables of the input clause as $\neg R(u, v) \vee R(v, u)$. Then the applied most general unifier (mgu) is $\alpha = \{u \leftarrow x, v \leftarrow f(x)\}$. The meaning is as follows. Initially, because I is all negative, the second and the third clause in S are satisfied by I, but the first one is not. Thus, SGGS generates $\Gamma = [R(x, f(x))]$ by an SGGS-extension with empty mgu. At this point, $I[\Gamma]$ satisfies the first and the second clause, but not the third one. Which ground instances of the third clause have been lost? Precisely those where $\neg R(u, v)$ unifies with $[R(x, f(x))]$. Thus, SGGS extends the model to recapture these instances by adding $\neg R(x, f(x)) \vee [R(f(x), x)]$.

However, it is not always the case that an SGGS-extension adds a clause E whose selected literal extends $I[\Gamma]$, because E may be a conflict clause. In such a case, SGGS *explains* the conflict by a restricted form of first-order resolution, called *SGGS-resolution*. SGGS-resolution resolves an I-false literal L in E with the implied I-true literal M, whose selection in Γ makes L uniformly false in $I[\Gamma]$. Thus, SGGS-resolution resolves the conflict clause E with the I-all-true clause D that is the justification of M in Γ. The resolvent is still in conflict. This series of *explanation inferences* by SGGS-resolution terminates when either the empty clause $\square$ or an I-all-true conflict clause is generated.

The generation of $\square$ signals that the input set S is unsatisfiable. Otherwise, SGGS *moves* the I-all-true conflict clause, say $E[L]$, to the left of the clause $D[M]$ whose selected I-false literal M makes E's I-true selected literal L uniformly false in $I[\Gamma]$. The effect of this *SGGS-move* is to *learn* $E[L]$ and *solve* the conflict by *flipping* the truth value of *all* ground instances of L. At this point, $D[M]$ is in conflict, so that SGGS-resolution intervenes to resolve $E[L]$ and $D[M]$ upon L and M. Prior to the move, SGGS may *partition* $D[M]$ by $E[L]$ as in the following:

Example 3.6. Continuing Example 3.4, we can see why $\neg R(a, b)$ is selected in conflict clause $\neg P(a) \vee [\neg R(a, b)]$: in an I-all-true conflict clause, SGGS prescribes to select the literal that is assigned rightmost, so that when the clause moves left to solve the conflict, the only literal in the clause that will be unassigned is the selected one, and the clauses changes status from conflict clause to learned justification of an implied literal. The move consists of moving $\neg P(a) \vee [\neg R(a, b)]$ to the left of $\neg P(x) \vee [R(a, x)]$. However, SGGS does not do that, because changing the truth value of all ground instances of $[R(a, x)]$ in order to satisfy $[\neg R(a, b)]$ is too much. The philosophy of SGGS is to be *conflict-driven* and change $I[\Gamma]$ only as far as it is needed to solve the conflict. SGGS partitions $\neg P(x) \vee [R(a, x)]$ by $\neg P(a) \vee [\neg R(a, b)]$ producing $\Gamma = [P(x)],\ x \not\equiv b \rhd \neg P(x) \vee [R(a, x)],\ \neg P(b) \vee [R(a, b)],\ \neg P(a) \vee [\neg R(a, b)]$. Next, SGGS-move yields $\Gamma = [P(x)],\ x \not\equiv b \rhd \neg P(x) \vee [R(a, x)],\ \neg P(a) \vee [\neg R(a, b)],\ \neg P(b) \vee [R(a, b)]$. SGGS-resolution resolves $\neg P(a) \vee [\neg R(a, b)]$ and $\neg P(b) \vee [R(a, b)]$ to generate $\Gamma = [P(x)],\ x \not\equiv b \rhd \neg P(x) \vee [R(a, x)],\ \neg P(a) \vee [\neg R(a, b)],\ \neg P(b) \vee [\neg P(a)]$, where the resolvent $\neg P(b) \vee [\neg P(a)]$ is another I-all-true conflict clause. The selection of $\neg P(a)$ is arbitrary, since both $\neg P(b)$ and $\neg P(a)$ are assigned to $[P(x)]$.

As shown in the above example, in SGGS-resolution, the resolvent *replaces* the parent that is not I-all-true. All clauses that have literals assigned to the deleted resolution parent are also deleted. In other words, the resolvent replaces the conflict clause, not the justification, like in CDCL. Partitioning a clause $D[M]$ by a clause $E[L]$ replaces $D[M]$ by a *partition*, $D_1[M_1], \ldots, D_n[M_n]$, that is, a set of clauses that together represent the same ground instances as $D[M]$, but have *disjoint* selected literals. Furthermore, the set of ground instances of $atom(L)$ is equal to the set of ground instances of $atom(M_j)$ for some j, $1 \leq j \leq n$, where $atom(L)$ denotes the atom of literal L. In other words, partitioning $D[M]$ by $E[L]$ splinters $D[M]$ in such a way to expose the non-empty intersection between the ground instances of L and those of M, where intersection ignores sign. Partitioning introduces *constraints*, that are a kind of *Herbrand constraints* [14, 17].

Example 3.7. Continuing Example 3.6, clause $\neg P(b) \vee [\neg P(a)]$ partitions clause $[P(x)]$, yielding $\Gamma = x \not\equiv a \rhd [P(x)],\ [P(a)],\ \neg P(a) \vee [\neg R(a, b)],\ \neg P(b) \vee [\neg P(a)]$, where $x \not\equiv b \rhd \neg P(x) \vee [R(a, x)]$ has been deleted: SGGS allows us to delete a clause that has a literal (here $\neg P(x)$) assigned to a clause (here $[P(x)]$) that gets partitioned. The alternative is to recursively partition $x \not\equiv b \rhd \neg P(x) \vee [R(a, x)]$ into $\neg P(a) \vee [R(a, a)]$ and $x \not\equiv b, x \not\equiv a \rhd \neg P(x) \vee [R(a, x)]$, and assign $\neg P(a)$ to $[P(a)]$ and $x \not\equiv b, x \not\equiv a \rhd \neg P(x)$ to $x \not\equiv a \rhd [P(x)]$. Note that $x \not\equiv b \rhd \neg P(x) \vee [R(a, x)]$ cannot simply remain in Γ, because $x \not\equiv b \rhd \neg P(x)$ has nowhere to be assigned after $[P(x)]$ has been partitioned. By SGGS-move we get $\Gamma = x \not\equiv a \rhd [P(x)],\ \neg P(b) \vee [\neg P(a)],\ [P(a)],\ \neg P(a) \vee [\neg R(a, b)]$. Then SGGS-resolution resolves $\neg P(b) \vee [\neg P(a)]$ and $[P(a)]$ to give $\Gamma = x \not\equiv a \rhd [P(x)],\ \neg P(b) \vee [\neg P(a)],\ [\neg P(b)]$, where $\neg P(a) \vee [\neg R(a, b)]$ is deleted, because its literal $\neg P(a)$ was assigned to the deleted resolution parent.

Another reason for deleting $\neg P(a) \vee [\neg R(a, b)]$ in the above example is that it is *disposable*: in SGGS a clause C in $\Gamma C \Gamma'$ is *disposable*, if it is satisfied by $I[\Gamma]$. *SGGS-deletion* deletes disposable clauses eagerly.

Example 3.8. Continuing Example 3.7, clause $[\neg P(b)]$ partitions clause $x \not\equiv a \rhd [P(x)]$, generating $\Gamma = x \not\equiv a, x \not\equiv b \rhd [P(x)],\ [P(b)],\ \neg P(b) \vee [\neg P(a)],\ [\neg P(b)]$. By SGGS-move we get $\Gamma = x \not\equiv a, x \not\equiv b \rhd [P(x)],\ [\neg P(b)],\ [P(b)],\ \neg P(b) \vee [\neg P(a)]$. Then SGGS-resolution yields $\Gamma = x \not\equiv a, x \not\equiv b \rhd [P(x)],\ [\neg P(b)],\ \square,\ \neg P(b) \vee [\neg P(a)]$.

Fairness of an SGGS-derivation ensures that inferences that are infinitely often possible are not neglected. It also ensures that every conflict is solved before further SGGS-extensions. SGGS is *refutationally complete*: if the input S is unsatisfiable, any fair SGGS-derivation from S is a refutation. It is also *model complete* in the limit: if S is satisfiable, the limiting sequence of any fair SGGS-derivation from S represents a model of S, where both limiting

sequence and derivation may be infinite. SGGS is flexible with respect to *goal-sensitivity*: it is goal-sensitive, if I satisfies the clauses issued from the assumptions H, but not those from the negation $\neg\varphi$ of the conjecture. SGGS is proof confluent, because it gets out of conflict by moving a clause in Γ, without undoing steps by backtracking or backjumping. This suggests that a backtracking search may not be an essential ingredient of conflict-driven reasoning.

3.2 CDSAT

CDSAT, for Conflict-Driven Satisfiability, extends MCSAT to *generic* combinations of disjoint theories [9, 10]. This involves clarifying the requirements that the theories $T_1, \ldots, T_n$ and their solvers need to fulfill, in order to ensure the soundness, completeness, and termination of the combined system.

CDSAT pushes further the philosophy of MCSAT of considering the CDCL procedure as one of n conflict-driven T-satisfiability procedures that cooperate to build a model in the trail. Accordingly, CDSAT regards atoms, literals, clauses and even formulæ as terms of sort prop (from proposition), a special sort that every theory signature is required to have. The assertion of a literal L in the trail is viewed as an assignment $L \leftarrow \mathit{true}$, and assignments such as $L \leftarrow \mathit{true}$ and $x \leftarrow 3$, where x is a free first-order variable, are treated in a completely uniform manner.

The trail is defined as a sequence of assignments, rather than as a sequence of literals. Furthermore, the notion of assignment is generalized to allow assignments to non-variable terms for both Boolean and first-order assignments. Thus, also input problems are written as assignments: in order to determine the satisfiability of a set of clauses $S = \{C_1, \ldots, C_m\}$, CDSAT is given as input the assignment $\{C_1 \leftarrow \mathit{true}, \ldots, C_m \leftarrow \mathit{true}\}$.

Concrete values such as 3 are not necessarily in the signatures of the theories. Therefore, CDSAT assumes *theory extensions* that add to the signatures as many constants as needed to name the concrete values (e.g., all the integers), including truth values. These extensions are required to be *conservative*, meaning that reasoning in the extension does not change the problem. Formally, the extension T_k^+ of theory T_k is *conservative*, if any T_k^+-unsatisfiable set S of clauses is also T_k-unsatisfiable. Thus, if CDSAT discovers T_k^+-unsatisfiability, the problem is T_k-unsatisfiable; if the problem is T_k-satisfiable, there is a T_k^+-model that CDSAT can build.

The combination of theories $T_1, \ldots, T_n$ is realized by the cooperation of *theory modules* $I_1, \ldots, I_n$, where theory modules are inference systems that work on assignments. Theory modules are the abstract counterpart of theory solvers or theory plugins [35]. Theory modules for propositional logic, and the quantifier-free fragments of the theory of equality, linear rational arithmetic, and the theory of arrays with extensionality are provided as examples [9, 10]. Every theory module I_k is responsible for deciding assignments, and for the inferences that lead to propagate assignments, and detecting, explaining, and solving conflicts, in its theory T_k.

The cooperation among the theory modules consists of having them all contribute to transform the trail. Since the theories have different signatures and the signatures are mixed on the trail, every theory module has its own *theory view* of the trail. CDSAT develops further the intuition, already in MCSAT and SGGS, that the essence of a conflict-driven approach is the *explanation* of conflicts. In CDSAT a conflict is a set of assignments. For the purpose of explanation, every assignment A in the trail that is not a decision is associated with a set of preceding assignments J that A was inferred from. Thus, if A becomes part of a conflict, it can be explained away by replacing it with J. Propositional resolution, as in a CDCL explanation, is a special instance of the CDSAT explanation mechanism. The inference system of CDSAT is parametrized by a *global basis*, which is the source of new terms that theory modules can employ in their inferences. CDSAT is sound and complete for combinations of disjoint theories, assuming that at least one of the theories has information about the cardinalities of the domains to interpret the shared sorts. Assuming that all theories are stably infinite is a special way of having this information. Similar to MCSAT, finiteness of the global basis ensures termination.

Clearly, there is no reason to restrict CDSAT to inputs of the form $\{C_1 \leftarrow \mathit{true}, \ldots, C_m \leftarrow \mathit{true}\}$. CDSAT accepts input problems containing both Boolean and first-order assignments. For example, one may need to decide the satisfiability of a quantifier-free formula φ in a combination of theories, given an assignment to some of the free variables in φ, whether propositional or first-order. Therefore, CDSAT addresses a more general problem than SMT, that we call SMA for *satisfiability modulo assignments*. For SMA problems, the input format presupposes the theory extensions.

3.3 Satisfiability modulo assignments

During the search, a conflict-driven reasoner maintains a partial candidate model represented by an assignment. This suggests the more general problem of *satisfiability modulo assignments* (SMA), defined as the problem of deciding the satisfiability of a set S of clauses modulo a theory T with respect to an initial assignment J to some of the terms in S, including *both* propositional and first-order terms. If J is empty, SMA reduces to SMT; if both J and T are empty, SMA reduces to SAT, while an intermediate state of a SAT or SMT search is an SMA instance. In CDSAT, there is no distinction between S and J, that are united to form the input assignment.

The answer to an SMA problem is either "satisfiable" with a model of S extending J, or "unsatisfiable" with a set of clauses E that follows from S and is false in J. The set E is an *explanation*, because it explains why S is unsatisfiable under J. The concept of *explanation* generalizes those of *unsatisfiable core* and *interpolant*. In SAT, an *unsatisfiable core* of S is a set of clauses that follows from S and is unsatisfiable. An unsatisfiable core explains why S is unsatisfiable, and the smaller it is with respect to the subset ordering $\subseteq$, the more precise it is regarded. If J is also written as a set of clauses, a *(reverse) interpolant* of S and J is a formula that follows from S and is inconsistent with J (cf. [11] for a survey of interpolation systems for ground proofs). MCSAT uses interpolants in arithmetic as explanations [26].

SMA arises in several contexts, such as *enumeration* of models, *parallelization*, and *optimization*. The models of a SAT or SMT problem can be enumerated by solving a series of SMA problems where each initial assignment J excludes the models already found. Approaches to parallel SAT by distributed search (cf. Section 4.1 in [5]) solve a SAT problem with input set S, by solving in parallel multiple instances of SMA with input set S and initial assignments

J each containing a distinct *guiding path* [57] or *cube* [32]. An optimization problem can be approached by solving a series of SMA problems where each initial assignment J contains information generated by the previous runs, in such a way that the series converges towards an optimal solution. For example, this concept appeared in the presentation of [28] about adapting to optimization the satisfiability procedure of [27, 37] for the theory of algebraic reals.

4 DISCUSSION

The big picture sees various approaches to extend conflict-driven reasoning to the first-order level. From the SMT side, the process started with generalizations of Conflict-Driven Clause Learning (CDCL) from propositional logic to several fragments of arithmetic [21, 31, 36–38, 44]. These methods offer *conflict-driven T-satisfiability procedures*. By being generic with respect to the theory, *Model-Constructing Satisfiability* (MCSAT) encompasses these predecessors, and by integrating theory reasoning and propositional reasoning in all aspects of deduction and search, it provides a paradigm for *conflict-driven T-decision procedures* [26, 34, 35, 56]. In turn, *Conflict-Driven Satisfiability* (CDSAT) generalizes MCSAT to generic combinations of theories and satisfiability modulo assignments (SMA) problems, where a partial assignment may also be part of the input problem [9, 10].

From the theorem-proving side, *Semantically-Guided Goal-Sensitive* (SGGS) reasoning [15–17] and *conflict resolution* [33, 51] lift CDCL to first-order logic. A comparison between SGGS and ordering-based theorem provers (e.g., [39, 43, 49, 54]) is premature, because SGGS still needs to be implemented and extended to first-order logic with equality. The point of SGGS is not to reprove the theorems that other approaches have already conquered, but rather to explore new domains or hard problems, where its conflict-driven character may be rewarding. The identification of such classes of problems is also an objective. Similarities between SGGS and CDSAT include the notion of mapping a literal, in SGGS, or an assignment, in CDSAT, to the literals, or assignments, respectively, that it depends on, and the notion that a model be part of the input problem, as SGGS assumes an initial interpretation for semantic guidance, while CDSAT accepts SMA problems. The future may witness further convergence.

REFERENCES

[1] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. 2005. On a rewriting approach to satisfiability procedures: extension, combination of theories and an experimental appraisal. In *Proceedings of the Fifth International Workshop on Frontiers of Combining Systems (FroCoS) (Lecture Notes in Artificial Intelligence)*, Bernhard Gramlich (Ed.), Vol. 3717. Springer, Berlin, Germany, EU, 65–80.

[2] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. 2009. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic* 10, 1 (2009), 129–179.

[3] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Splitting on demand in SAT modulo theories. In *Proceedings of the Thirteenth International Conference on Logic, Programming and Automated Reasoning (LPAR) (Lecture Notes in Artificial Intelligence)*, Miki Hermann and Andrei Voronkov (Eds.), Vol. 4246. Springer, Berlin, Germany, EU, 512–526.

[4] Maria Paola Bonacina. 2010. On theorem proving for program checking – Historical perspective and recent developments. In *Proceedings of the Twelfth International Symposium on Principles and Practice of Declarative Programming (PPDP)*, Maribel Fernández (Ed.). ACM, New York, New York, USA, 1–11.

[5] Maria Paola Bonacina. 2017. Parallel theorem proving. In *Handbook of Parallel Constraint Reasoning*, Youssef Hamadi and Lakhdar Sais (Eds.). Lecture Notes in Computer Science, Vol. 177–233. Springer, Berlin, Germany, EU, 177–233.

[6] Maria Paola Bonacina and Mnacho Echenim. 2007. Rewrite-based satisfiability procedures for recursive data structures. In *Proceedings of the Fourth Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR), held at the Fourth Federated Logic Conference (FLoC), August 2006*, Byron Cook and Roberto Sebastiani (Eds.). Electronic Notes in Theoretical Computer Science, Vol. 174(8). Elsevier, Amsterdam, The Netherlands, EU, 55–70.

[7] Maria Paola Bonacina and Mnacho Echenim. 2008. On variable-inactivity and polynomial $\mathcal{T}$-satisfiability procedures. *Journal of Logic and Computation* 18, 1 (2008), 77–96.

[8] Maria Paola Bonacina, Ulrich Furbach, and Viorica Sofronie-Stokkermans. 2015. On first-order model-based reasoning. In *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer*, Narciso Martí-Oliet, Peter Olveczky, and Carolyn Talcott (Eds.). Lecture Notes in Computer Science, Vol. 9200. Springer, Berlin, Germany, EU, 181–204.

[9] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. 2016. *A model-constructing framework for theory combination.* Technical Report 99/2016. Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy, EU. Also Technical Report of SRI International and INRIA - CNRS - École Polytechnique; revised April 2017.

[10] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. 2017. Satisfiability modulo theories and assignments. In *Proceedings of the Twenty-Sixth Conference on Automated Deduction (CADE) (Lecture Notes in Artificial Intelligence)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, Berlin, Germany, EU, 42–59.

[11] Maria Paola Bonacina and Moa Johansson. 2015. Interpolation systems for ground proofs in automated deduction: a survey. *Journal of Automated Reasoning* 54, 4 (2015), 353–390.

[12] Maria Paola Bonacina, Christopher A. Lynch, and Leonardo de Moura. 2009. On deciding satisfiability by DPLL($\Gamma + \mathcal{T}$) and unsound theorem proving. In *Proceedings of the Twenty-second International Conference on Automated Deduction (CADE) (Lecture Notes in Artificial Intelligence)*, Renate Schmidt (Ed.), Vol. 5663. Springer, Berlin, Germany, EU, 35–50.

[13] Maria Paola Bonacina, Christopher A. Lynch, and Leonardo de Moura. 2011. On deciding satisfiability by theorem proving with speculative inferences. *Journal of Automated Reasoning* 47, 2 (2011), 161–189.

[14] Maria Paola Bonacina and David A. Plaisted. 2014. Constraint manipulation in SGGS. In *Proceedings of the Twenty-Eighth Workshop on Unification (UNIF) at the Sixth Federated Logic Conference (FLoC) (Technical Reports of the Research Institute for Symbolic Computation)*, Temur Kutsia and Christophe Ringeissen (Eds.). Johannes Kepler Universität Linz, Linz, Austria, EU, 47–54.

[15] Maria Paola Bonacina and David A. Plaisted. 2015. SGGS theorem proving: an exposition. In *Proceedings of the Fourth Workshop on Practical Aspects in Automated Reasoning (PAAR) at the Sixth Federated Logic Conference (FLoC), July 2014 (EasyChair Proceedings in Computing (EPiC))*, Stephan Schulz, Leonardo De Moura, and Boris Konev (Eds.), Vol. 31. EasyChair, Manchester, England, UK, 25–38.

[16] Maria Paola Bonacina and David A. Plaisted. 2016. Semantically-guided goal-sensitive reasoning: model representation. *Journal of Automated Reasoning* 56, 2 (2016), 113–141.

[17] Maria Paola Bonacina and David A. Plaisted. 2017. Semantically-guided goal-sensitive reasoning: inference system and completeness. *Journal of Automated Reasoning* 59, 2 (2017), 165–218.

[18] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation - Decision Procedures with Applications to Verification.* Springer, Berlin, Germany, EU.

[19] Robert Brummayer and Armin Biere. 2009. Lemmas on demand for the extensional theory of arrays. *Journal on Satisfiability, Boolean Modeling and Computation* 6 (2009), 165–201.

[20] Chin-Liang Chang and Richard Char-Tung Lee. 1973. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, Cambridge, England, UK.

[21] Scott Cotton. 2010. Natural domain SMT: A preliminary assessment. In *Proceedings of the Eighth International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS) (Lecture Notes in Computer Science)*, Krishnendu Chatterjee and Thomas A. Henzinger (Eds.), Vol. 6246. Springer, Berlin, Germany, EU, 77–91.

[22] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.

[23] Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. *J. ACM* 7 (1960), 201–215.

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Engineering DPLL(T) + saturation. In *Proceedings of the Fourth International Conference on Automated Reasoning (IJCAR) (Lecture Notes in Artificial Intelligence)*, Alessandro Armando, Peter Baumgartner, and Gilles Dowek (Eds.), Vol. 5195. Springer, Berlin, Germany, EU, 475–490.

[25] Leonardo de Moura and Nikolaj Bjørner. 2008. Model-based theory combination. In *Proceedings of the Fifth International Workshop on Satisfiability Modulo Theories (SMT 2007) (Electronic Notes in Theoretical Computer Science)*, Sava Krstić and Albert Oliveras (Eds.), Vol. 198(2). Elsevier, Amsterdam, The Netherlands, EU, 37–49.

[26] Leonardo de Moura and Dejan Jovanović. 2013. A model-constructing satisfiability calculus. In *Proceedings of the Fourteenth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.), Vol. 7737. Springer, Berlin, Germany, EU, 1–12.

[27] Leonardo de Moura and Grant Olney Passmore. 2013. Computation over real closed infinitesimal and transcendental extensions of the rationals. In *Proceedings of the Twenty-Fourth Conference on Automated Deduction (CADE) (Lecture Notes in Artificial Intelligence)*, Maria Paola Bonacina (Ed.), Vol. 7898. Springer, Berlin, Germany, EU, 177–191.

[28] Leonardo de Moura and Grant Olney Passmore. 2013. Exact global optimization on demand (Presentation only). In *Notes of the Third Workshop on Automated Deduction: Decidability, Complexity, Tractability (ADDCT)*, Silvio Ghilardi, Viorica Sofronie-Stokkermans, and Ashish Tiwari (Eds.). 50–50. Available at https://userpages.uni-koblenz.de/~sofronie/addct-2013/, last seen on May 9, 2017.

[29] Leonardo de Moura and Harald Rueß. 2002. Lemmas on demand for satisfiability solvers. In *Proceedings of the Fifth International Symposium on the Theory and Application of Satisfiability Testing (SAT) (Lecture Notes in Computer Science)*. Springer, Berlin, Germany, EU, 244–251.

[30] Bruno Dutertre and Leonardo de Moura. 2006. A fast linear arithmetic solver for DPLL(T). In *Proceedings of the Eighteenth International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science)*, Tom Ball and R. B. Jones (Eds.), Vol. 4144. Springer, Berlin, Germany, EU, 81–94.

[31] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. 2012. Deciding floating-point logic with systematic abstraction. In *Proceedings of the Twelfth International Conference on Formal Methods in Computer Aided Design (FMCAD)*, Gianpiero Cabodi and Satnam Singh (Eds.). ACM and IEEE, New York, New York, USA.

[32] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. 2012. Cube and conquer: guiding CDCL SAT solvers by lookaheads. In *Proceedings of the Seventh Haifa Verification Conference (HVC) (Lecture Notes in Computer Science)*, K. Eder, J. Lourenço, and O. Shehory (Eds.), Vol. 7261. Springer, Berlin, Germany, EU, 50–65.

[33] Daniyar Itegulov, John Slaney, and Bruno Woltzenlogel Paleo. 2017. Scavenger 0.1: a theorem prover based on conflict resolution. In *Proceedings of the Twenty-Sixth Conference on Automated Deduction (CADE) (Lecture Notes in Artificial Intelligence)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, Berlin, Germany, EU.

[34] Dejan Jovanović. 2017. Solving Nonlinear Integer Arithmetic with MCSAT. In *Proceedings of the Eighteenth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, Ahmed Bouajjani and David Monniaux (Eds.), Vol. 10145. Springer, Berlin, Germany, EU, 330–346.

[35] Dejan Jovanović, Clark Barrett, and Leonardo de Moura. 2013. The design and implementation of the model-constructing satisfiability calculus. In *Proceedings of the Thirteenth Conference on Formal Methods in Computer Aided Design (FMCAD)*, Barbara Jobstman and Sandip Ray (Eds.). ACM and IEEE, New York, New York, USA.

[36] Dejan Jovanović and Leonardo de Moura. 2011. Cutting to the chase: solving linear integer arithmetic. In *Proceedings of the Twenty-Third International Conference on Automated Deduction (CADE) (Lecture Notes in Artificial Intelligence)*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.), Vol. 6803. Springer, Berlin, Germany, EU, 338–353.

[37] Dejan Jovanović and Leonardo de Moura. 2012. Solving non-linear arithmetic. In *Proceedings of the Sixth International Joint Conference on Automated Reasoning (IJCAR) (Lecture Notes in Artificial Intelligence)*, Bernhard Gramlich, Dale Miller, and Ulrike Sattler (Eds.), Vol. 7364. Springer, Berlin, Germany, EU, 339–354.

[38] Konstantin Korovin, Nestan Tsiskaridze, and Andrei Voronkov. 2009. Conflict resolution. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP) (Lecture Notes in Computer Science)*, Ian P. Gent (Ed.), Vol. 5732. Springer, Berlin, Germany, EU, 509–523.

[39] Laura Kovàcs and Andrei Voronkov. 2013. First order theorem proving and Vampire. In *Proceedings of the Twenty-Fifth International Conference on Computer-Aided Verification (CAV) (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, Berlin, Germany, EU, 1–35.

[40] Sava Krstić and Amit Goel. 2007. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In *Proceedings of the Sixth International Symposium on Frontiers of Combining Systems (FroCoS) (Lecture Notes in Artificial Intelligence)*, Frank Wolter (Ed.), Vol. 4720. Springer, Berlin, Germany, EU, 1–27.

[41] João P. Marques Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans Van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, Amsterdam, The Netherlands, EU, 131–153.

[42] João P. Marques Silva and Karem A. Sakallah. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.* 48, 5 (1999), 506–521.

[43] William W. McCune. 2005–2010. Prover9 and Mace4. (2005–2010). http://www.cs.unm.edu/~mccune/prover9/, last seen on May 10, 2017.

[44] Kenneth L. McMillan, A. Kuehlmann, and Mooly Sagiv. 2009. Generalizing DPLL to richer logics. In *Proceedings of the Twenty-First International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, Berlin, Germany, EU, 462–476.

[45] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-Ninth Design Automation Conference (DAC)*, David Blaauw and Luciano Lavagno (Eds.). ACM and IEEE, New York, New York, USA, 530–535.

[46] Greg Nelson. 1983. Combining satisfiability procedures by equality sharing. In *Automatic Theorem Proving: After 25 Years*, Woodrow W. Bledsoe and Donald W. Loveland (Eds.). American Mathematical Society, Providence, Rhode Island, USA, 201–211.

[47] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems* 1, 2 (1979), 245–257.

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

[49] Stephan Schulz. 2013. System description: E 1.8. In *Proceedings of the Nineteenth International Conference on Logic, Programming and Automated Reasoning (LPAR) (Lecture Notes in Artificial Intelligence)*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.), Vol. 8312. Springer, Berlin, Germany, EU, 735–743.

[50] Natarajan Shankar. 2009. Automated deduction for verification. *Comput. Surveys* 41, 4 (2009), 40–96.

[51] John Slaney and Bruno Woltzenlogel Paleo. 2017. Conflict resolution: a first-order resolution calculus with decision literals and conflict-driven clause learning. *Journal of Automated Reasoning* in press (2017), 1–27.

[52] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy Levitt. 2001. A decision procedure for an extensional theory of arrays. In *Proceedings of the Sixteenth IEEE Symposium on Logic in Computer Science (LICS)*, Joseph Halpern (Ed.). IEEE Computer Society Press, Los Alamitos, California, USA.

[53] Chao Wang, Franjo Ivančić, Malay Ganai, and Aarti Gupta. 2005. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *Proceedings of the Twelfth International Conference on Logic, Programming and Automated Reasoning (LPAR) (Lecture Notes in Artificial Intelligence)*, Geoff Sutcliffe and Andrei Voronkov (Eds.), Vol. 3835. Springer, Berlin, Germany, EU, 322–336.

[54] Christoph Weidenbach, Dylana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS version 3.5. In *Proceedings of the Twenty-Second International Conference on Automated Deduction (CADE) (Lecture Notes in Artificial Intelligence)*, Renate Schmidt (Ed.), Vol. 5663. Springer, Berlin, Germany, EU, 140–145.

[55] Steven A. Wolfman and Daniel S. Weld. 1999. The LPSAT engine and its application to resource planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Thomas Dean (Ed.), Vol. 1. Morgan Kaufmann Publishers, San Francisco, California, USA, 310–316.

[56] Aleksandar Zeljić, Christoph M. Wintersteiger, and Philipp Rümmer. 2016. Deciding bit-vector formulas with `mcSAT`. In *Proceedings of the Nineteenth International Conference on Theory and Applications of Satisfiability Testing (SAT) (Lecture Notes in Computer Science)*, Nadia Creignou and Daniel Le Berre (Eds.), Vol. 9710. Springer, Berlin, Germany, EU, 249–266.

[57] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. 1996. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation* 21, 4–6 (1996), 543–560.

[58] Hantao Zhang and Mark E. Stickel. 2000. Implementing the Davis-Putnam method. *Journal of Automated Reasoning* 24, 1/2 (2000), 277–296.