The *Eos* SMT/SMA-Solver: A Preliminary Report (Extended Abstract)

Maria Paola Bonacina and Giulio Mazzi

Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15, I-37134 Verona, Italy, EU
mariapaola.bonacina@univr.it
giulio.mazzi@univr.it

Abstract

This is a preliminary report of work in progress on the development of the *Eos* SMT/SMA-solver. *Eos* is the first solver built from the start based on the CDSAT (*Conflict-Driven SATisfiability*) paradigm for solving satisfiability problems modulo theories and assignments. The latter means that assignments to first-order terms may appear in the input. CDSAT generalizes MCSAT (*Model-Constructing SATisfiability*), hence CDCL (*Conflict-Driven Clause Learning*), to theory combination. CDSAT reasons in a union of theories by combining in a conflict-driven manner theory inference systems, called *theory modules*. The current version of *Eos* has modules for propositional logic, equality with uninterpreted function symbols (UF), and linear real arithmetic. The module for propositional logic is a MiniSAT-inspired SAT solver. A key feature of MCSAT/CDSAT is theory conflict explanation by theory inferences: to this end, the *Eos* module for UF applies *congruence closure inferences*, and the *Eos* module for real arithmetic uses *Fourier-Motzkin resolution*; both rules may generate new (i.e., non-input) literals. The core solver in *Eos* implements the CDSAT transition system and several heuristics used in state-of-the-art CDCL-based SAT solvers. Some of these heuristics (e.g., random restarts) can be reused directly in the context of CDSAT, while others are adapted. *Eos* employs a *generalization of the VSIDS heuristics* to make decisions on both propositional and first-order terms, and the *watched literals scheme* for both BCP (Boolean Constraint Propagation) and deductions involving arithmetic terms and uninterpreted terms.

1 Introduction

CDSAT, which stands for *Conflict-Driven SATisfiability* [3, 4, 5], is a method designed for the problem of *satisfiability modulo theories and assignments* (SMT/SMA): given a quantifier-free formula φ and an assignment J of values to subterms of φ, determine whether there is a model of the theories that satisfies φ, while respecting the assignments in J. If J is empty, the problem is an SMT problem. Since most problems from applications involve multiple theories, CDSAT is conceived since the start for reasoning in a *union of disjoint theories*, that is, theories that may share only sorts and equality on shared sorts. CDSAT is a generalization of the MCSAT framework for *Model-Constructing SATisfiability* [7, 13, 12, 10, 2] to generic theory combinations. MCSAT integrates the CDCL (*Conflict-Driven Clause Learning*) procedure for propositional satisfiability (SAT) with a *conflict-driven theory satisfiability procedure* (e.g., [20, 16, 6, 14, 15]). Thus, CDSAT is also a generalization of the *equality sharing* method for theory combination [23, 22], also known as the *Nelson-Oppen scheme*, to include in the combination conflict-driven theory satisfiability procedures. This paper is a preliminary description of ongoing work on the brand new prototype *Eos*, which is the first SMT/SMA solver born to implement CDSAT.

2 CDSAT: An Exposition

Following the CDCL procedure for propositional satisfiability (SAT) [19, 18], solvers represent a candidate model by storing on a *trail* assignments of truth values to propositional variables. In SMT-solvers that integrate theory solvers with the CDCL procedure, propositional variables may be genuine propositional variables or abstractions of first-order atoms. MCSAT adds assignments of natural values to first-order variables (e.g., integer values for integer variables), and CDSAT generalizes this feature to allow such assignments to *first-order terms*.

Since SMA problems include assignments to first-order terms, and solvers work with assignments during the search, CDSAT views *all data as assignments*. A formula φ is the abbreviation of the assignment $\varphi \leftarrow \mathsf{true}$, logical connectives are seen as function symbols, and formulæ as Boolean terms. All theories have the sort of Boolean values. A *Boolean assignment* assigns a truth value to a Boolean term (e.g., $(x > 0) \leftarrow \mathsf{true}$), and a *first-order assignment* assigns to a first-order term a value of the appropriate sort, as in $f(x) \leftarrow 3$ if the function symbol f has the sort of the integers as output sort. CDSAT treats first-order assignments and Boolean assignments in a uniform way. With first-order assignments there are two ways to determine the truth value of an equality: either by assigning it directly (e.g., $(x \simeq y) \leftarrow \mathsf{true}$) or by assigning values to its sides (e.g., $\{x \leftarrow 0, y \leftarrow 0\}$ makes $x \simeq y$ true).

Let $\mathcal{U}$ denote the union of the theories and $\mathcal{T}$ stand for a component theory. An assignment is a set of distinct pairs $u \leftarrow \mathfrak{c}$, where u is a term in the global signature of theory $\mathcal{U}$ and $\mathfrak{c}$ is a value. A singleton assignment is a single such pair $u \leftarrow \mathfrak{c}$. All subterms of term u are said to *occur* in the assignment. Values are new constant symbols added to the signature of a theory to name the elements in the domains of interpretation of its sorts (e.g., truth values, integers, algebraic reals, but also generic values for generic sorts of uninterpreted symbols). A $\mathcal{T}$-assignment is an assignment where all values come from theory $\mathcal{T}$. A $\mathcal{U}$-assignment, or *global assignment*, or assignment for short, may mix values from different theories. We use A for generic singleton assignments, L for Boolean singleton assignments, J for $\mathcal{T}$-assignments, and E or H for $\mathcal{U}$-assignments. The flip of L, denoted $\overline{L}$, assigns to L the opposite truth value.

In *conflict-driven reasoning*, nontrivial inferences are performed only to *explain* conflicts. In CDCL, propositional resolution is applied to explain the Boolean conflict represented by a clause and the flips of all its literals. Resolvents can be learned as *lemmas* in a heuristically controlled manner. MCSAT generalizes conflict explanation to theory conflicts, requiring that the theory solver, termed *theory plugin*, has inference rules that explain theory conflicts. As these inference rules may generate *theory lemmas* that contain *new* (i.e., non-input) terms, termination requires that these terms come from a *finite basis*. CDSAT extends this approach from one theory to many, realizing *conflict-driven reasoning in a union of theories*. In CDSAT, each component theory is equipped with an inference system, called *theory module*, which provides inference rules for theory conflict explanation. Since conflict-driven reasoning happens in all theories, and not only in propositional logic, CDSAT regards propositional logic, if present, as one of the component theories. For termination, CDSAT restricts new terms to come from a *finite global basis* built from *finite local bases*, one per theory. A theory module is an abstraction of a theory satisfiability procedure; it is defined as a set of inference rules that work with assignments. A theory module $\mathcal{I}$ for theory $\mathcal{T}$ has inference rules of the form $J \vdash_{\mathcal{I}} L$, where L is a singleton Boolean assignment and J is a $\mathcal{T}$-assignment. Since all theories have equality symbols for their sorts, all modules include inference rules for reflexivity, symmetry, and transitivity of equality, as well as two inference rules to infer the truth value of an equality when both its sides are assigned values. CDSAT modules are required to be *sound*: if $J \vdash L$, then $J \models L$, which means that every $\mathcal{T}$-model that satisfies J satisfies L.

Given an input problem written as a global assignment H_0, CDSAT works with a trail Γ initialized with H_0 and shared by all theory modules, so that Γ contains a global assignment. A CDSAT trail is a sequence of distinct singleton assignments that can be either *decisions*, or *justified assignments*, that is, assignments with a justification. A trail can be seen as an assignment by forgetting the order of its elements. A decision can be either a Boolean or a first-order assignment. A justified assignment is a generalization of the concept of *implied literal* in CDCL: it can be either an input assignment, or the result of a theory inference, or an outcome of solving a conflict. The justification of an input assignment is empty, and the only first-order justified assignments are input assignments. All other justified assignments are Boolean. If a justified assignment depends on a theory inference $J \vdash L$, the justification of L is J. A *conflict* E is an unsatisfiable subset of the trail: $E \subseteq \Gamma$ and $H_0 \cup E \models\perp$. If a conflict $E \uplus \{L\}$, where $\uplus$ denotes disjoint union, is solved by flipping assignment L, that is, by placing $\overline{L}$ on the trail, $\overline{L}$ is a justified assignment with E as justification, as $H_0 \cup E \uplus \{\overline{L}\} \models\perp$ implies $H_0 \cup E \models L$.

Every assignment on a CDSAT trail has a *level*: the level of a decision is the successor of the level of the previous decision; the level of a justified assignment is the maximum among the levels of the elements of its justification. Therefore, unlike in CDCL, in CDSAT it is not granted that the levels of the assignments on the trail are in increasing order: a justified assignment L with justification H and level m may appear *after* an assignment A of level n, with $n > m$, if A does not belong to H. This situation and assignment L are termed a *late propagation*.

The CDSAT *transition system* comprises *trail rules* to search for a model and *conflict-state rules* to solve conflicts. The trail rules Decide, Deduce, Fail, and ConflictSolve transform the trail Γ. Rule Decide expands Γ with a decision $u \leftarrow \mathfrak{c}$, provided this assignment is *acceptable* for a module $\mathcal{I}$ of a component theory $\mathcal{T}$: Γ does not already assigns to u a value coming from $\mathcal{T}$; and if $u \leftarrow \mathfrak{c}$ is first-order its addition does not enable an $\mathcal{I}$-inference $J \cup \{u \leftarrow \mathfrak{c}\} \vdash_{\mathcal{I}} L$ for $J \subseteq \Gamma$ and $\overline{L} \in \Gamma$. In the Boolean case, L is acceptable if neither L nor $\overline{L}$ are on the trail. Acceptability also requires that the term u is *relevant* to theory $\mathcal{T}$: either u occurs in Γ and $\mathcal{T}$ has values for its sort, or u is an equality whose sides occur in Γ, and $\mathcal{T}$ does not have values for their sort; in the latter case $\mathcal{I}$ decides the truth value of the equality.

Assume that a theory module inference $J \vdash_{\mathcal{I}} L$ applies to the trail as $J \subseteq \Gamma$. If $\overline{L} \notin \Gamma$, a Deduce transition expands Γ with the justified assignment L. Deduce transitions cover propagations, including both *Boolean Constraints Propagation* (BCP) and *theory propagations*, and *theory conflict explanations*. If the $\mathcal{T}$-satisfiability procedure for theory $\mathcal{T}$ detects a conflict in Γ, its module $\mathcal{I}$ performs inferences framed as Deduce transitions, until the conflict surfaces on the trail with an inference $J \vdash_{\mathcal{I}} L$ and $\overline{L} \in \Gamma$. Deduce can be used in *conflict-driven style*, making sure that nontrivial inferences fire only for conflict explanation, or in a *forward-reasoning style* that enables nontrivial inferences to fire eagerly. The choice depends on theory and module, like the notion of what is a nontrivial inference. If Γ contains $\overline{L}$, the conflict $J \cup \{\overline{L}\}$ is detected: if its level is 0, rule Fail reports unsatisfiability; otherwise, rule ConflictSolve lets the conflict-state rules take control, resuming the search for a model from the trail they return.

The conflict-state rules UndoClear, Resolve, Backjump, and UndoDecide transform a trail Γ that contains a conflict. Assume that the conflict includes an assignment A that *stands out*, as its level is the maximum in the conflict and A is the *only* assignment of maximum level in the conflict. If A is a first-order assignment, rule UndoClear undoes A and clears Γ of all assignments of level greater than or equal to that of A. If A is a Boolean assignment L, let E be the rest of the conflict: rule Backjump unrolls the trail to the level of E and adds $\overline{L}$ with justification E. In CDSAT endowed with *lemma learning* [4], rule Backjump is replaced by a more general rule named LearnBackjump, that can flip any Boolean subset $H = \{\overline{L}_1, \ldots, \overline{L}_n\}$ of a conflict $H \uplus E$ to learn clause $L_1 \vee \ldots \vee L_n$, as $H_0 \cup E \uplus \{\overline{L}_1, \ldots, \overline{L}_n\} \models\perp$ implies $H_0 \cup E \models L_1 \vee \ldots \vee L_n$.

If the conflict does not contain an outstanding assignment, rule Resolve picks a justified assignment A in the conflict and unfolds the conflict by replacing A with its justification H. In the Boolean case this transition rule emulates propositional resolution [5], as the conflict contains the negation $\{\overline{L}_1, \ldots, \overline{L}_n\}$ of a conflict clause $L_1 \vee \ldots \vee L_n$. This process continues until an outstanding literal emerges in the conflict, so that either UndoClear or Backjump applies. However, Resolve is inhibited if A is Boolean and its nonempty justification H contains a first-order decision A', whose level is maximum in the conflict: if Resolve were authorized in this case, UndoClear would undo A', but Decide could reiterate it, leading Deduce to infer A again, resulting in a loop. In this situation, UndoDecide removes both A and A' by backtracking like UndoClear and adds $\overline{A}$ as a decision. A discussion of the differences between CDSAT and MCSAT is available [5].

3 The *Eos* Main Solver

The main solver of *Eos* implements the CDSAT transition system, and it can be extended with an *arbitrary* number of theory modules. The current version of *Eos* integrates three modules: the SAT module for propositional logic, the LRA module for linear real arithmetic, and the UF module that handles uninterpreted function symbols and equality for uninterpreted sorts.

The trail in *Eos* is a CDSAT trail, where however the justifications of justified assignments are not stored on the trail. The system saves with every justified assignment L the identifier of the module responsible for the justification of L. When the justification of L is needed, a request is issued to that module. In this manner, every module can save this information in a convenient way relative to the reasoning in its theory and the module implementation. For a justified assignment L generated by a Deduce transition supported by an inference $J \vdash_{\mathcal{I}} L$, the module identifier saved with L is that of module $\mathcal{I}$. For a justified assignment L with justification E generated by a Backjump transition, the module identifier saved with L is that of the SAT module, because in the current implementation the SAT module builds this kind of justification. While functional, this solution is not especially flexible, and it is likely to change in the future.

The main functions of *Eos* are named `check_sat` and `conflict_analysis`. The `check_sat` function implements the search for a model of the input problem, covering the trail rules of the CDSAT transition system. The `conflict_analysis` function implements the conflict-state rules. The pseudocode of these two functions is provided in the figures labeled Algorithm 1 and Algorithm 2, respectively.

The `check_sat` function executes a loop, that it exits by firing Fail to report unsatisfiability, or when the assignment on the trail is satisfied. The function tries to propagate some truth value, by calling the specific propagation methods of the theory modules. This activity corresponds to Deduce transitions in CDSAT. This process continues until there is no more value to propagate or a conflict is found in the current assignment. If the conflict is at level zero the problem is unsatisfiable and the function returns unsatisfiable (rule Fail). Otherwise, the `conflict_analysis` function will take care of the conflict. If no more propagations are possible, a decision must be made. The main solver selects a term for a decision based on a heuristic (see Sect. 3.1), and then it asks the appropriate theory module to assign an acceptable value to the term. This is the implementation of the Decide rule. The `check_sat` function is only superficially similar to the analogous function in CDCL. For example, the `propagate` and `make_decision` procedures of *Eos* do not alter the trail themselves, they are only responsible for calling theory modules that work with the trail.

The `conflict_analysis` function (see the figure labeled Algorithm 2) begins by extracting

Algorithm 1 check_sat

```
function check_sat
    loop
        propagate( )                              ▷ rule Deduce
        if conflict then                          ▷ the propagation has generated a conflict
            if conflict at level zero then
                return unsatisfiable              ▷ rule Fail
            else
                conflict_analysis( )              ▷ rule ConflictSolve
        else                                      ▷ everything was propagated without conflict
            if decision order is empty then       ▷ every term has a value assigned?
                return satisfiable                ▷ SAT
            else
                make_decision( )                  ▷ rule Decide
```

from the trail the "reason" of the conflict, which is what CDSAT simply calls the conflict itself. Then, it retrieves the highest level among those of the assignments in the conflict: this level is called the *conflict level*. Since every level greater than the conflict level is inconsistent, a backjump to the conflict level is performed right away.

The procedure begins the resolution process between the trail and the conflict, implementing the Resolve rule. The last element of the trail that is at conflict level is removed from the conflict, and its justification is added to the conflict in its place (unless it is already there). If one of the elements of the justification is a first-order assignment that happens to be at the conflict level, rule UndoDecide is applied. Otherwise, the process continues until the conflict contains a single assignment at the conflict level. If this oustanding assignment is first-order, an *UndoClear* transition is performed. If this oustanding assignment is Boolean, its complement is a *Unique Implication Point* (UIP) as in CDCL, and Backjump applies. In the implementation, Boolean justified assignments are encoded as clauses. Thus, the justified assignment that Backjump extracts from the conflict is encoded as a clause with the UIP as implied literal. The SAT module is invoked to store the clause and propagate the UIP.

Eos has an implementation of the LearnBackjump rule [4], but it is under testing and still subject to change. Another feature of *Eos* that is implemented, but requires more testing, is a procedure for *conflict minimization*, inspired by a technique used in SAT solvers to minimize the length of conflict clauses [25]. *Eos* generalizes it to handle also first-order assignments. *Eos* can restart the search process by backjumping to level 0. After a certain number of conflicts, the search is stopped and *Eos* performs a restart. Similar to MiniSAT [9], *Eos* employs the Luby sequence to determine the number of conflicts after which a restart is issued.

3.1 Heuristics for Decisions

When no more deductions are possible, the solver must make a decision. The information required to determine the *acceptability* of a decision for a theory module is stored in the module. The main solver is responsible for calling the appropriate module for every term that requires a decision. The selection of terms for decisions is based on a generalization of the *VSIDS heuristic* [21] to handle both Boolean and first-order terms. Similar to CDCL, the current implementation of *Eos* increases the *activity* of both Boolean and first-order terms only during conflict analysis.

Algorithm 2 conflict_analysis

```
procedure conflict_analysis
    conflict ← get_reason()                                  ▷ get the reason of the conflict
    conflict_level ← get_max_level(conflict)                 ▷ higher level of conflict values
    backjump(conflict_level)                                 ▷ undo everything after the conflict
    while conflict has two or more terms at conflict_level do
        last ← pop_from_trail( )                             ▷ get the last Boolean propagation on the trail
        if last.level() = conflict_level and last is in conflict then    ▷ rule Resolve
            conflict.remove(last)                            ▷ resolve this value with the conflict
            ▷ get the justification of this propagation
            justification ← get_justification(last)
            for all Term just in justification do
                ▷ this propagation is justified by a first order decision at the conflict level?
                if just is non-Boolean and at conflict level then
                    new_value ← ¬ trail.get_value(last)      ▷ flip the value of the propagation
                    backjump_one_level( )
                    add_decision(last,new_value)             ▷ rule UndoDecide
                    return
                else
                    conflict.add(just)                       ▷ add just to the conflict
    ▷ here, the conflict has a single term assigned at the level of the conflict
    topmost_var ← get_unassigned(conflict)
    if topmost_var is non-Boolean then
        backjump_one_level( )                                ▷ rule UndoClear
        return
    clause ← create_clause(conflict)                         ▷ learn a new clause
    bt_level ← compute_backjump_level(conflict)
    backjump(bt_level)                                       ▷ rule Backjump
    learn_new_clause(clause)
```

When *Eos* undoes an assignment $u \leftarrow \mathfrak{c}$ by removing it from the trail, it stores it in a cache. In this manner, if term u gets selected again for a decision, *Eos* checks whether u appears in the cache. If this is the case, and term u is relevant to component theory $\mathcal{T}$, the main solver asks the module for $\mathcal{T}$ to determine whether value $\mathfrak{c}$ is still acceptable. If it is, the next decision about term u reuses the cached value.

3.2 Knowledge Representation in *Eos*

Eos stores all the information about sorts and terms in a hash-consed database. A term is represented by an integer that acts as an index in the database, allowing the system to retrieve from the database whatever information about the term is required. All formulæ are kept reduced to a *canonical form*, so that it is possible to assign the same index to all formulæ that have the same canonical form. These formulæ are equivalent formulæ written in different ways. The implementation of this database is based on the one used in Yices [8]. *Eos* is written in C++; it accepts problems written in SMT-LIB 2.6 notation [1]; and it can handle problems in the *QF_UF, QF_LRA* and *QF_UFLRA* logics.

4 The SAT Module

The SAT module handles propositional logic. All Boolean formulæ are reduced to equisatisfiable conjunctive normal form by applying the Tseitin transformation [26]. The implementation of the SAT module is based on the MiniSAT solver [9], with a focus on performing Boolean Clausal Propagation (BCP) very efficiently. This is achieved by using a *watched literals scheme* that watches two literals per clause, in order to identify implied literals (also known as unit clauses), and fully assigned clauses. In addition, the SAT module uses a specialized memory manager to accelerate the BCP by saving clauses in a compact region in memory.

A unit clause is one where all literals are assigned except one, which is the implied literal. When a clause is discovered to be a unit clause with implied literal L, if L occurs in the clause with positive polarity (e.g., L is P), the assignment $P \leftarrow \mathsf{true}$ is added to the trail; if L occurs in the clause with negative polarity (e.g., L is $\neg P$), the assignment $P \leftarrow \mathsf{false}$ is added to the trail. A fully assigned clause is one such that the trail contains assignments for all its literals. If a clause is fully assigned the module checks whether the clause is satisfied; if not, it means that it is a conflict clause and a conflict is raised. The conflict is composed of the conflict clause and the complements of all its literals.

Learned clauses are removed periodically in order to keep the BCP fast. Similar to MiniSAT, the quality of a learned clause is measured by an activity-based heuristic. Those learned clause whose measure is lower than a given threshold are eliminated by the garbage collection mechanism of *Eos*. The garbage collection process removes clauses with low activity and new terms generated for explaining conflicts that only appears in low-activity clauses. Otherwise, the system could be overwhelmed by the number of newly introduced terms.

5 The LRA Module

The *Fourier-Motzkin algorithm* decides the satisfiability of a set of linear disequalities over the reals, by applying exhaustively *Fourier-Motzkin (FM) resolution*, and therefore has very high complexity (see [24], Ch. 12, and [17], Ch. 5). The LRA module of *Eos*, the LRA plugin of MCSAT [13], and previous procedures [6, 20, 16], use FM-resolution *only* to explain conflicts, like CDCL does with propositional resolution. Thus, these procedures stand to the Fourier-Motzkin algorithm like CDCL stands to a procedure that decides propositional satisfiability by applying resolution exhaustively.

Given polynomials t_1 and t_2 and a free variable x that is not free in t_1 and t_2, FM-resolution derives a new relation between t_1 and t_2:

$$t_1 \lessdot_1 x, x \lessdot_2 t_2 \vdash t_1 \lessdot_3 t_2 \tag{1}$$

where $\lessdot_1, \lessdot_2 \in \{<, \leq\}$, and $\lessdot_3$ is $\leq$ if both $\lessdot_1$ and $\lessdot_2$ are $\leq$, and it is $<$ otherwise. Another rule is required to handle a special case. Given the polynomials t_0, t_1, t_2, and a free variable x that is not free in t_0, t_1, t_2, the *Disequality Elimination* rule can be applied to explain a conflict:

$$t_1 \leq x, x \leq t_2, t_1 \simeq t_0, t_2 \simeq t_0, x \not\simeq t_0 \vdash \bot \tag{2}$$

The LRA module propagates truth values to arithmetic formulæ and keeps a set of acceptable assignments for every real variable in the system. This is accomplished by using a mechanism similar to the two watched literals scheme of the SAT module. A polynomial $a_1 \cdot x_1 + \cdots + a_n \cdot x_n \simeq c$ is *watched* in a clause that contains all the variables $x_1 \ldots x_n$ plus the whole formula. When all the variables $x_1 \ldots x_n$ are assigned, it is possible to assign a truth value to the formula: this is an *evaluation* inference in the description of the module as a set of inference rules [5]. When a truth value is assigned to the formula and all variables $x_1 \ldots x_n$ are assigned values except for one variable x_i, we have a *unit constraint* [13]. Thus, it is possible to update the information on the set of acceptable values for x_i. For example, given the assignment $\{x \leftarrow 1, (2x < y) \leftarrow \mathsf{true}\}$, an acceptable value for y must be greater than 2.

For every variable, the module saves lower and upper bounds, a list of disequalities, and possibly an equality. When two bounds are incompatible, rules (1) and (2) are applied to explain the conflict, possibly by generating a new term not already present in the input.

6 The UF Module

The UF module handle equalities and inequalities between terms made of uninterpreted symbols, building in the congruence axioms of equality for all non-nullary uninterpreted function symbols. This module performs propagations by using an extension of the watching literals scheme analogous to the one used in the LRA module. Every equality clause $u_1 \simeq u_2$ has three components: the two sides u_1 and u_2 of the equality, and the Boolean term t that represents the equality itself. If the two sides are assigned, it is possible to assign a truth value to the equality. If one of the sides, say u_1, and the truth value of the equality are known, it is possible to glean some information on what is an acceptable value for the other side u_2. If the equality is assigned true, the value of u_2 is also determined. If the equality is assigned false, the value of u_1 can be excluded from the set of acceptable values for u_2.

Given m-tuples of terms $t_1 \ldots t_m$ and $u_1 \ldots u_m$, where t_i and u_i have the same sort for all i, $1 \leq i \leq m$, the following inference rule embodies the congruence axiom for all function symbols f in the signature:

$$(t_i = u_i)_{i=1\ldots m} \vdash f(t_1, \ldots, t_m) = f(u_1, \ldots, u_m) \tag{3}$$

It is possible that some of the equalities are generated as new terms. This rule is implemented via another variant of the watched literals scheme [13]. In this case, the arguments of the function symbol are watched, and when they are all assigned, the module checks that the congruence property holds.

7 Current and Future Work

Eos is a prototype under active development: the code is at https://gitlab.com/GiuMaz/eos_smt, with the more recent updates under the *develop* branch. Current work includes a revision of the conflict analysis procedure to incorporate the LearnBackjump rule [4], and a modification of the mechanism to select terms for decisions that takes into account that some decisions are *forced*. For a first-order term u, the decision $u \leftarrow \mathfrak{c}$ is *forced* if $\mathfrak{c}$ is the only value left for u (e.g., if $\{u \leq t,\ t \leq u,\ t \leftarrow \mathfrak{c}\} \subseteq \Gamma$ in arithmetic). *Eos* will make forced decisions eagerly, reserving its VSIDS-inspired heuristics to all other decisions, those that are true guesses. The distinction between forced decisions and true guesses changes the CDCL-inherited notion that the system makes a decision only when no more propagations are possible, as forced decisions should be made as soon as possible, together with Boolean propagations.

Another topic for current work is the performance of *Eos* on QF_LRA and QF_UFLRA benchmarks. The implementation of an approach that reduces the need for recomputing values by using time-stamps [13] is expected to decrease the time spent in computing the values of polynomials. *Eos* shows promising results in the QF_UF category: given a time limit of 10 minutes, it can solve nearly 90% of the problems in this category and it performs especially well in the `eq_diamond` family of benchmarks [11].

The best architecture for a CDSAT-based SMT/SMA-solver is a main issue for current and future work, and the development of *Eos* allows us to investigate this issue without the burden of legacy code. CDSAT treats all theories and all modules as peers, although propositional logic still retains a few advantages (e.g., the module inferences infer only Boolean assignments). In *Eos*, the SAT solver and the other modules are not exactly peers for historical reasons (the SAT solver in *Eos* was developed first and as a stand-alone tool), but this may change in the future. An MCSAT architecture placing the module for the theory of equality at the center was preliminarily explored [2]. While inferring only Boolean assignments is not a restriction in theory, a variant of MCSAT that allows the system to deduce first-order assignments exists [12] and is implemented in Yices [8]: this extension is future work for both CDSAT and *Eos*.

A main direction for future work is the extension of *Eos* with modules for more theories, such as *arrays with extensionality* [3, 5], *bit-vectors* [27, 10], other *datatypes*, and *nonlinear arithmetic* [14, 12], so that it can handle more problems. It seems especially interesting to consider theories where conflict-driven reasoning is known or expected to have a significant impact, such as nonlinear arithmetic [14, 12].

Acknowledgments Part of this work was done when the first author was visiting the Computer Science Lab of SRI International, whose support is greatly appreciated. This work was funded in part by grant "Ricerca di base 2017" of the Università degli Studi di Verona.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories library (SMT-LIB). www.SMT-LIB.org, 2016.

[2] François Bobot, Stéphane Graham-Lengrand, Bruno Marre, and Guillaume Bury. Centralizing equality reasoning in MCSAT. In Vijay D'Silva and Rayna Dimitrova, editors, *Workshop on Satisfiability Modulo Theories (SMT-16)*, 2018.

[3] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Satisfiability modulo theories and assignments. In Leonardo de Moura, editor, *Conference on Automated Deduction (CADE-26)*, volume 10395 of *LNAI*, pages 42–59. Springer, 2017.

[4] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Proofs in conflict-driven theory combination. In June Andronick and Amy Felty, editors, *Certified Programs and Proofs (CPP-7)*, pages 186–200. ACM Press, 2018.

[5] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Conflict-driven satisfiability for theory combination: transition system and completeness. *Journal of Automated Reasoning*, in press:1–31, 2019. Available at http://doi.org/10.1007/s10817-018-09510-y.

[6] Scott Cotton. Natural domain SMT: A preliminary assessment. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS-8)*, volume 6246 of *LNCS*, pages 77–91. Springer, 2010.

[7] Leonardo de Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI-14)*, volume 7737 of *LNCS*, pages 1–12. Springer, 2013.

[8] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification (CAV-26)*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[9] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT-7)*, volume 9710 of *LNCS*, pages 502–518. Springer, 2004.

[10] Stéphane Graham-Lengrand and Dejan Jovanović. An MCSAT treatment of bit-vectors. In Martin Brain and Liana Hadarean, editors, *Workshop on Satisfiability Modulo Theories (SMT-15)*, 2017.

[11] Dejan Jovanović. Model constructing satisfiability calculus – A model-based approach to SMT. Lecture at the SAT/SMT Summer School, 2015.

[12] Dejan Jovanović. Solving nonlinear integer arithmetic with MCSAT. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI-18)*, volume 10145 of *LNCS*, pages 330–346. Springer, 2017.

[13] Dejan Jovanović, Clark Barrett, and Leonardo de Moura. The design and implementation of the model-constructing satisfiability calculus. In Barbara Jobstman and Sandip Ray, editors, *Formal Methods in Computer Aided Design (FMCAD-13)*. ACM and IEEE, 2013.

[14] Dejan Jovanović and Leonardo de Moura. Solving non-linear arithmetic. In Bernhard Gramlich, Dale Miller, and Ulrike Sattler, editors, *International Joint Conference on Automated Reasoning (IJCAR-6)*, volume 7364 of *LNAI*, pages 339–354. Springer, 2012.

[15] Dejan Jovanović and Leonardo de Moura. Cutting to the chase: solving linear integer arithmetic. *Journal of Automated Reasoning*, 51:79–108, 2013.

[16] Konstantin Korovin, Nestan Tsiskaridze, and Andrei Voronkov. Conflict resolution. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming (CP-15)*, volume 5732 of *LNCS*. Springer, 2009.

[17] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. Springer, 2008.

[18] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marjin Heule, Hans Van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in AI and Applications*, pages 131–153. IOS Press, 2009.

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

[20] Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing DPLL to richer logics. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification (CAV-21)*, volume 5643 of *LNCS*, pages 462–476. Springer, 2009.

[21] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In David Blaauw and Luciano Lavagno, editors, *Annual Design Automation Conference (DAC-38)*, pages 530–535, 2001.

[22] Greg Nelson. Combining satisfiability procedures by equality sharing. In Woodrow W. Bledsoe and Don W. Loveland, editors, *Automatic Theorem Proving: After 25 Years*, pages 201–211. American Mathematical Society, 1983.

[23] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[24] Alexander Schrijver. *Theory of Linear and Integer Programming*. Interscience Series in Discrete Mathematics and Optimization. Wiley, 1998.

[25] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing (SAT-12)*, volume 5584 of *LNCS*, pages 237–243. Springer, 2009.

[26] G.S. Tsetin. On the complexity of derivation in propositional calculus. In A.O. Slisenko, editor, *Studies in constructive mathematics and mathematical logic*, volume 2, pages 115–125. Consultants Bureau, 1970. Presented at the Leningrad Seminar on Mathematical Logic 1966; reprinted in J. Siekmann and G. Wrightson (eds.), *Automation of reasoning*, Vol. 2, 466–483, Springer, 1983.

[27] Aleksandar Zeljić, Christoph M. Wintersteiger, and Philipp Rümmer. Deciding bit-vector formulas with `mcSAT`. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing (SAT-19)*, volume 9710 of *LNCS*, pages 249–266. Springer, 2016.