The Design and Implementation of the Model Constructing Satisfiability Calculus

Dejan Jovanović
New York University

Clark Barrett
New York University

Leonardo de Moura
Microsoft Research

***Abstract*—We present the design and implementation of the Model Constructing Satisfiability (MCSat) calculus. The MCSat calculus generalizes ideas found in CDCL-style propositional SAT solvers to SMT solvers, and provides a common framework where recent model-based procedures and techniques can be justified and combined. We describe how to incorporate support for linear real arithmetic and uninterpreted function symbols in the calculus. We report encouraging experimental results, where MCSat performs competitive with the state-of-the art SMT solvers without using pre-processing techniques and ad-hoc optimizations. The implementation is flexible, additional plugins can be easily added, and the code is freely available.**

I. Introduction

Considering the theoretical hardness of SAT, the astonishing adeptness of SAT solvers when attacking practical problems has changed the way we perceive the limits of algorithmic reasoning. Modern SAT solvers are based on the idea of *conflict-driven clause learning* (CDCL) [1]–[3]. The CDCL algorithm is a combination of an explicit backtracking search for a satisfying assignment complemented with a deduction system based on Boolean resolution. In this combination, the worst-case complexity of both components is circumvented by the components guiding and focusing each other.

Generalization of the SAT problem to the first-order domain is called satisfiability modulo theories (SMT). On the shoulders of efficient SAT solvers, and numerous successful applications, SMT gained momentum as a more expressive and equally performant framework. The common approach to solving SMT problems is to employ a SAT solver to enumerate assignments of the Boolean abstraction of the formula. The candidate (partial) Boolean assignments are then either confirmed or refuted by a *decision procedure* dedicated to reasoning about conjunctions of theory-specific constraints. If multiple theories are involved, satisfiability in the combination of such theories can be ensured by relying on high level combination frameworks in the spirit of Nelson and Oppen [4]–[6]. This style of reasoning is commonly called DPLL(T) [7], [8] and is employed by most of the SMT solvers today.

The Model-Constructing Satisfiability (MCSat) calculus [9] provides a more general alternative to DPPL(T), liting the idea of the CDCL-style model construction with conflict resolution to first-order domain. MCSat encompasses many recent model-based decision procedures for theories such as linear real arithmetic [10]–[12], linear integer arithmetic [13], nonlinear arithmetic [14], and floating-point [15] arithmetic.

Although the model-based decision procedures have proved effective for theories of high complexity, its was unclear how to approach the theory-combination problem, and if the approach can be competitive for "simple" theories where incumbent solutions seem to be satisfactory. In this paper we describe an implementation of the MCSat framework that can reason effectively in the combination of linear real arithmetic and uninterpreted functions, providing positive answers to both concerns. The procedure for linear arithmetic is a careful but conceptually simple implementation of a model-driven Fourier-Motzkin elimination, while the combination with uninterpreted functions is provided through model-driven Ackermanization [5], [16].

II. Preliminaries

We assume that the reader is familiar with the usual notions and terminology of propositional and first-order logic (see e.g. [17]).

As usual, we will denote the set of rational numbers as $\mathbb{Q}$ and use a, b, c to denote constants from $\mathbb{Q}$. We assume a finite set of Boolean and real variables, denoting them with letters x, y, z, and a finite set of uninterpreted function symbols which we denote with letters f, g. Each such function symbol f is associated with a fixed arity $k > 0$. We define a *pure term* inductively, with variables and constants being pure terms, and a function term $f(t_1, \ldots, t_k)$ being pure if each t_k is a pure term. We will refer to non-constant pure terms as *generalized variables* and, with abuse of notation, we will also refer to them with letters x, y, z.[1]

We use and p, q to denote linear polynomials over generalized variables with coefficients in $\mathbb{Q}$. All polynomials are assumed to be in sum-of-monomials normal form $a_1x_1 + \cdots + a_nx_n + c$, with a_i and c being constants, and x_i denoting generalized variables. For linear polynomials p and q, a linear constraint is a constraint of the form $p \bigtriangledown q$, where $\bigtriangledown \in \{<, \leq, =\}$.

An *atom* is either a Boolean variable or a linear constraint, and we consider atoms to be generalized variables of Boolean type. A *literal* L is an atom or a negation of an atom. A *clause* C is a disjunction of literals $(L_1 \vee \cdots \vee L_n)$, and we denote the empty clause with $\bot$.

[1] Intuitively, generalized variables are pure terms seeking an interpretation.

Example II.1. *Consider the constraints*

$$f(x+1) < y,\ \ x = y\ .$$

The term $f(x+1)$ is not pure, but we can purify the constraints by introducing a new variable s_1 obtaining

$$f(s_1) < y,\ \ s_1 = x+1,\ \ x = y\ .$$

The constraints above are satisfiable, for example, by the interpretation $x \mapsto 1,\ y \mapsto 1,\ s_1 \mapsto 2,\ f(s_1) \mapsto 0$.

A. Deduction Rules

MCSat as a proof system is a clausal deduction system based on clausal inference rules. Given a set of input clauses, MCsat either finds an assignment of variables that satisfies the clauses, or derives a proof of the unsatisfiability using the rules below.

The core of MCSat is driven by the *Boolean resolution* rule. Given two clauses $C \vee L$ and $\neg L \vee D$, we can eliminate the literal L using the Boolean resolution rule

$$\frac{C \vee L \qquad \neg L \vee D}{C \vee D}$$

We denote the result of applying the resolution rule with $\mathsf{resolveB}(C, D, L)$.

For reasoning in linear arithmetic we use the *Fourier-Motzkin rule*. Given two inequalities $(p_L < x)$ and $(x < p_U)$, we can eliminate the variable x using the Fourier-Motzkin rule, obtaining a new inequality $(p_L < p_U)$. In clausal form, this rule can be stated as

$$\frac{}{\neg(p_L < x) \vee \neg(x < p_U) \vee (p_L < p_U)}$$

We denote this rule with $\mathsf{resolveFM}$. The rule above is applied to strict inequalities and, as expected, we overload $\mathsf{resolveFM}$ to cover non-strict inequalities and equalities.

In addition to the Fourier-Motzkin rule, in order to reason about dis-equalities we also use the *equality split rule*, which states that the relation between polynomials p and q can only be one of the three.

$$\frac{}{(p = q) \vee (q < p) \vee (p < q)}$$

We denote the split rule with $\mathsf{splitEq}$.

For reasoning about uninterpreted functions we use the *Ackermann expansion rule* which states that, for any uninterpreted function symbol f of arity k, if $x_i = y_i$ for $i = 1, \ldots, k$, then also $f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k)$, or, in clausal form

$$\frac{}{x_1 \neq y_1 \vee \cdots \vee x_k \neq y_k \vee f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k)}$$

We denote the Ackermann rule with $\mathsf{resolveCC}$.

We also assume a general "normalization" rule that performs simple semantics-preserving transformation on clauses, denoted with a dashed line, such as

$$\frac{\neg(p < q) \vee (x < 0) \vee (x < 0)}{(q \leq p) \vee (x < 0)}$$

The four rules above, together with the normalization rule, comprise the whole of our proof system, which speaks to the simplicity of the MCSat approach.

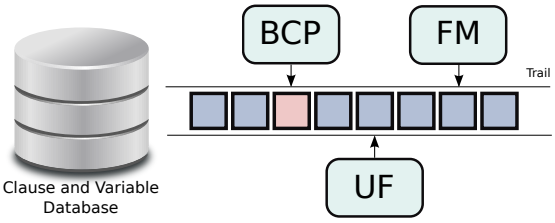

Fig. 1. Main solver components

III. Core Architecture

The MCSat architecture consists of a core solver that manages the solver components depicted in Figure 1. The main components are the clause and variable database, the solver trail, and the reasoning plugins. The core solver drives the solving process, and is responsible for dispatching notifications and handling requests from the plugins, while the plugins reason about the content of the trail. The most important duty of the core is to perform conflict analysis when the reasoning plugins detect a conflicting state.

The clause database contains a compact representation of all the clauses in the system (including unit clauses). The clause database contains both the input clauses and the clauses learned during the search. The variable database maintains the information about all the generalized variables in the system in an efficient index-based form. Both databases are occasionally compacted, by heuristically marking the clauses to keep[2], and then garbage-collecting unmarked clauses and variables using a mark-and-sweep approach. Maintaining a balance amount of clauses and variables is crucial in the MCSat setting as, in contrast to the DPLL(T) framework, model-based decision procedures can generate large amounts of new literals that would, if not removed, eventually overwhelm the system.

Plugins are the standard way of extending MCSat, and we currently have implementations of dedicated plugins that can reason about the Boolean structure, linear real arithmetic, and uninterpreted functions symbols, described in Section IV.

A. The Trail

The central data-structure in the framework is the solver trail. It is a generalized version of the trail found in modern SAT solvers. In our design of the interface to the trail, we make sure that the trail is used not only as the container of reasoning history, but also as the object ensuring formal progress towards termination of the search process.

The trail is a sequence of trail elements, where each element can be either a Boolean decision, a semantic decision, a clausal propagation, or a semantic propagation.

A *Boolean decision* is a literal L that we assume to be true, and is emphasized in the trail as $\mathbf{L}$. These elements are the equivalent of decided literals found in modern SAT solvers. A *semantic decision* is a decision on the value of a *non-Boolean* generalized variable. We write semantic decisions as $\mathbf{x} \mapsto \alpha$,

[2]We make sure that the GC does not affect termination by increasing the amount of clauses that are kept.

where α represents a value from the type of the variable x. For example, we may decide that the value of a real variable x is 1.[3] A *clausal propagation* is a literal L derived to be true through clause C using Boolean constraint propagation (BCP), which is marked as $L_{\downarrow C}$. If the clause $C = L$, i.e. it is a unit clause, we mark the propagation just as $L_{\downarrow}$. The *level* of a decision element, or a clausal propagation, is the number of decisions in the trail up to and including the element itself.

We say that a literal L (term t) *can be evaluated* if the generalized variables that appear in the literal L (term t) are all assigned values in the trail through semantic decisions. A *semantic propagation* marks a literal L that can be evaluated to **true**. We denote semantic propagation as $L_{\downarrow k}$, where the value k represents the level of the highest semantic decision used in evaluating L. The *level* of a semantic propagation $L_{\downarrow k}$ is k. If a literal L ($\neg L$) appears on the trail M as an element of level k, we say that L is **true** (**false**) in M, and the level of L is k.

Example III.1. *Consider the clause*

$$C \equiv (0 < x) \vee (0 < y) \vee (1 < x + y)$$

and the trail

$$M = [\![\overset{1}{\mathbf{x} \mapsto \mathbf{0}},\ \neg(\overset{2}{\mathbf{0} < \mathbf{y}}),\ \neg(0 \overset{1}{<} x)_{\downarrow 1},\ (1 < \overset{2}{x} + y)_{\downarrow C},]\!] \ .$$

The level of each element is marked above it in the trail. The first two elements of M are a semantic decision assigning the variable x to the value 0, and a Boolean decision of the literal $\neg(0 < y)$. The next element of M is a semantic propagation of $\neg(0 < x)$. Note that the semantic propagation is marked at level 1, as this is the level at which x is assigned. The last element of M is a clausal propagation of the literal $(1 < x+y)$ due to clause C.

Adding new elements to the end of the trail, by performing a decision, or propagating a literal, is restricted as follows.

- No literal L can be added to the trail (either by decision or propagation), if the literal L or $\neg L$ already appears on the trail.
- No semantic decisions $\mathbf{x} \mapsto \alpha$ can be added to the trail if it invalidates a literal L that is already on the trail, i.e. if $\neg L$ would be a semantic propagation after the decision.
- No Boolean decision $\mathbf{L}$ can be added to the trail if it invalidates a clause C in the system, i.e. if the literal $\neg L$ appears in C, and all other literals of C are already **false**.
- A clausal propagation $L_{\downarrow C}$ can only be added to the trail if the literal L appears in C, and all all other literals of C are already **false**.

Besides adding elements to the trail, a trail can also be *backtracked*. Backtracking amounts to retracting some decisions and their consequences from the trail. We require that any backtracking of the trail be accompanied with a clausal *backtracking reason* that is used to maintain the invariance of progress. The reason for backtracking is always a clause C that evaluates to **false** in the trail, thus a signal that the search must be revised. In addition to being false, the clause C should also be either a *unique implication point* [2] (UIP) clause or a *semantic split* clause, as explained below, and we denote this with the predicate $\mathsf{canBacktrackWith}(M, C)$.

Let $\mathsf{topLevel}(M, C)$ denote the highest level (in M) of a literal from C, and let $\mathsf{topLiterals}(M, C)$ denote the set of literals from C at this highest level. Clause C is a UIP clause if there is only one literal L in $\mathsf{topLiterals}(M, C)$. A UIP clause C can thus be used to propagate L at the second highest level of literals in C, or level 0 if C is unit, and we denote this level as $\mathsf{uipLevel}(M, C)$. Clause C is a semantic split if each $L \in \mathsf{topLiterals}(M, C)$ is a semantic propagation $L_{\downarrow k}$ in M.

Algorithm 1: MCSAT::BACKTRACKWITH(M, C)

Data: trail M, clause C evaluates to **false** in M
if *C is a UIP clause in M* **then**
  $level \leftarrow \mathsf{uipLevel}(M, C)$
else C is a semantic split clause in M
  $level \leftarrow \mathsf{topLevel}(M, C)$ - 1
remove from M all elements of level $> level$
if *C was a UIP clause* **then**
  for the unassigned $L \in C$, add $L_{\downarrow C}$ to M
else C was a semantic split clause
  for an unassigned $L \in C$, add decision $\mathbf{L}$ to M

The backtracking procedure `backtrackWith` is presented in Algorithm 1. We call propagations, decisions, and backtracking *valid* if they conform to the restrictions outlined above.

Example III.2. *Consider again the clause C and trail M from Example III.1. Using the Fourier-Motzkin rule, we can deduce the following valid clause*

$$\frac{}{\neg(1 - x < y) \vee \neg(y \leq 0) \vee (1 - x < 0)}$$
$$R_1 \equiv \neg(1 < x + y) \vee (0 < y) \vee (1 < x)$$

The first two of the literals from R_1 are already **false** *in the trail M, and the last literal $(1 < x)$ is a new literal. The new literal can be evaluated in M, and we can propagate it semantically, obtaining a new trail*

$$[\![\overset{1}{\mathbf{x} \mapsto \mathbf{0}}, \neg(\overset{2}{\mathbf{0} < \mathbf{y}}), \neg(0 \overset{1}{<} x)_{\downarrow 1}, (1 < \overset{2}{x} + y)_{\downarrow C}, \neg(1 \overset{1}{<} x)_{\downarrow 1}]\!] \ .$$

The clause R_1 is valid and evaluates to **false** *at M, i.e. the search needs to be revised. But, R_1 is not suitable for backtracking since it contains two literals of level 2, and none of them is a semantic propagation. Fortunately, we can use the Boolean resolution rule to remove the literal propagated by C and deduce*

$$\frac{C \qquad R_1}{R_2 \equiv (1 < x) \vee (0 < x) \vee (0 < y)}$$

The clause R_2 only contains one literal at the highest level, so it is an UIP clause suitable for backtracking. The effect of

[3]Boolean decisions are in fact just a particular case of a semantic decision, convenient for presentation and implementation purposes.

backtracking the trail with `backtrackWith(`R`)` *is a new trail*

$$[\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0}, \neg(0 \overset{1}{<} x)_{\downarrow 1}, \neg(1 \overset{1}{<} x)_{\downarrow 1}, (0 \overset{1}{<} y)_{\downarrow R_2}]\!] \ .$$

Note that, in addition to the propagated UIP literal, the backtracking procedure also kept the late semantic propagations.

It is important to note that we diverge slightly from the original MCSat presentation [9], in that the trail as presented here contains explicit semantic propagation elements. In [9], all semantic propagation were implicit in the trail and available through evaluation. This change was guided by implementation reasons since it allows, for example, a more efficient implementation of Boolean constraint propagation. The change is a minor one, and the main theoretical results from [9] hold in this setting and we restate the most important ones as lemmas.

It was shown in [9] (Theorem 1) that using valid trail operations is enough to ensure termination, if we assume that all literals that the system will ever see come from a finite set of literals $\mathcal{B}$, that we call the *finite basis*.

Lemma III.1. *Starting from an empty trail* $[\![]\!]$*, any procedure that only uses valid trail operations, while using only literals from the finite basis* $\mathcal{B}$*, can only make a finite number of such trail operations.*

In Section IV we will show how the finite basis assumption can be ensured for problems combining linear arithmetic and uninterpreted functions.

B. Conflict Analysis

As in CDCL SAT solvers, conflict analysis is used to learn from the conflicting clauses encountered during the search – clauses with all literals **false** in the trail. As seen in Example III.2, it is possible to identify clausal conflicts that can not directly be used for backtracking. Conflict analysis takes a conflicting clause and transforms it into a new clause that is suitable for backtracking. This newly learned clause is used in the main solver loop to revise the search.

A conflicting clause is not suitable for backtracking if it contains more than one literal at the top level, and these literals are not semantic propagations. It is easy to show that the problematic literals are clausal propagations, and conflict analysis can eliminate them by performing Boolean resolution with the clauses that propagated them. Since the analysis uses existing valid clauses and the resolution rule, the result of such conflict analysis will be a valid deduction. Therefore, if conflict analysis learns an empty clause $\bot$, this will be a signal to the main solver that the problem is unsatisfiable. The conflict analysis procedure is presented in Algorithm 2.

Note that the analysis procedure only uses the resolveB rule. Other deduction rules are "axiom" rules used to create new clauses which can be used to identify conflicting situations or for propagation purposes. Also, as the analysis procedure concludes as soon it finds a clause suitable for backtracking, if this is the case due to a UIP clause, the analysis style corresponds to the 1st UIP SAT strategy [18].

Algorithm 2: MCSAT::ANALYZECONFLICT(M, C)

Data: solver trail M, clause C inconsistent with M

```
k ← M.size()
while C ≠ ⊥ and ¬ canBacktrackWith(M, C) do
    k ← k − 1
    if M[k] = L_↓D and ¬L ∈ C then
        C ← resolveB(C, D, L)
return C
```

Lemma III.2. *Given a valid trail* M *and a clause* C *that is* **false** *in* M*, the* `analyzeConflict(`M, C`)` *procedure always terminates with a clause* R *that is a valid deduction and is either the empty clause or is suitable for backtracking.*

C. Main Search Loop

The algorithm behind MCSat is based on the search-and-resolve loop common in modern SAT solvers (e.g. [19]). The main loop of the solver performs a "smart" search for a satisfying assignment and terminates either by finding the assignment that satisfies the original problem, or deduces that the problem is unsatisfiable. The main `check()` method of the solver is presented in Algorithm 3.

Algorithm 3: MCSAT::CHECK()

Data: solver trail M, variables to assign in *queue*

```
1  while true do
2      propagate()
3      if detected conflict with clause C then
4          R ← analyzeConflict(M, C)
5          if R = ⊥ then return unsat
6          backtrackWith (M, R)
7      else
8          if queue.empty() then return sat
9          x ← queue.pop()
10         decideValue(x)
```

The search process goes forward, making continuous progress, either through propagation, conflict analysis, or by making a decision. The `propagate()` procedure invokes the propagation procedures provided by the enabled plugins. Each plugin is allowed to propagated new information to the top of the trail. If a plugin detects an inconsistency this is communicated to the solver by producing a conflicting clause. This is recorded by the solver and allows the solver to analyze the conflict using the `analyzeConflict()` procedure. If conflict analysis learns the empty clause $\bot$, the problems is proven unsatisfiable, otherwise the learned clause is used to backtrack the search.

On the other hand, if the plugins have performed propagation to exhaustion, and no conflict was detected, the procedure makes progress by deciding an unassigned variable to a value. The solver picks an unassigned variable x to be assigned, and

relegates the choice of the value to the plugin responsible for assigning x. A choice of value for the selected unassigned variable should exist, as otherwise a plugin should have detected the inconsistency. MCSat uses a uniform heuristic to select the next variable, regardless of their type. The heuristic is based on how often a variable is used in conflict resolution, and is popularly used in CDCL-style SAT solvers [3]. Note that, as explained in the preliminaries, every atom (e.g., $x < 2$) is treated as a generalized Boolean variable. If all the variables are assigned to a value, this is a satisfying assignment for the original problem.

IV. Plugins

The reasoning engines in MCSat are organized in modules that we call *plugins*. The plugins can register listeners for notification about important events in the system, such as new assertion formulae, creation of new clauses and generalized variables, and garbage-collection events. Plugins participate in the solving process by performing propagation and detecting conflicts, with dedicated plugins also taking part in selecting values for variables.

In order to ensure completeness in the system, if a plugin is dedicated to selecting values for a particular type T (such as Boolean or real), it must be unit-constraint complete. We call a plugin *unit-constraint complete* for type T if, after a call to `propagate()`, either the plugin has identified a conflicting clause C, or, for each unassigned variable x of type T there exists a valid decision $\mathbf{x} \mapsto \alpha$ (or a Boolean decision if T is Boolean). Note that unit-constraint completeness *does not* require that the plugin ensures consistency of all assertions, only the constraints with a single unassigned variable are satisfiable (grouped by variable) – a much easier property to check.

Example IV.1. *Consider the clauses*

$$C_1 \equiv ((x + y \leq 0) \vee (0 \leq y) \vee z)$$
$$C_2 \equiv ((x + y \leq 0) \vee (0 \leq y) \vee \neg z) \ ,$$

where variables x and y are of real type, and the variable z a Boolean, with the corresponding trail

$$M = [\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0},\ \neg(\mathbf{x} + \mathbf{y} \overset{2}{\leq} \mathbf{0}),\ \neg(\mathbf{0} \overset{3}{\leq} \mathbf{y}) \ ]\!] \ .$$

In the trail M, the clauses C_1 and C_2 have all but one literal **false***, i.e. they are unit constraints that can propagate a literal. M does not allow a value for variable z, since assigning z to* **true** *invalidates clause C_1, and assigning z to* **false** *invalidates C_2. This kind of unit constraint conflict can be detected with exhaustive Boolean constraint propagation. For example, using C_1, we can propagate z obtaining*

$$M' = [\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0},\ \neg(\mathbf{x} + \mathbf{y} \overset{2}{\leq} \mathbf{0}),\ \neg(\mathbf{0} \overset{3}{\leq} \mathbf{y}),\ z^{3}_{\downarrow C_1} \ ]\!] \ .$$

In the trail M' the clause C_2 is **false** *and is a conflicting clause.*

In addition to the Boolean conflict above, the original trail M does not allow a selection of value for the real variable y. With respect to y there are two unit constraints in M – the constraint $\neg(x + y \leq 0) \equiv (0 < x + y)$ (that evaluates to $0 < y$) and the constraint $\neg(y \geq 0) \equiv y < 0$ – and they are in conflict.

The conflicting unit constraints can be resolved using the resolveFM *rule obtaining the clause*

$$\frac{\neg(-x < y) \vee \neg(y < 0) \vee (-x < 0)}{R \equiv (x + y \leq 0) \vee (0 \leq y) \vee (0 < x)}$$

We can semantically propagate that the new literal $(0 < x)$ is **false***, obtaining a trail*

$$M'' = [\![\mathbf{x} \overset{1}{\mapsto} \mathbf{0},\ \neg(\mathbf{x} + \mathbf{y} \overset{2}{\leq} \mathbf{0}),\ \neg(\mathbf{0} \overset{3}{\leq} \mathbf{y}),\ \neg(0 \overset{1}{<} x)_{\downarrow 1} \ ]\!] \ .$$

In the trail M'' the clause R is **false** *and is a conflicting clause.*

A. BCP And Watchlists

As hinted in Example IV.1, in order to ensure unit-completeness for the Boolean variables in a clausal setting, it is enough to perform Boolean constraint propagation (BCP) to exhaustion. We've implemented the customary efficient BCP loop in a dedicated BCP plugin, with the basic mechanics built upon the important concept of watchlists.

In the SAT literature, the two-literal watchlist was first introduced in [3] as an efficent mechanism to detect when a clause becomes unit. In the MCsat approach, the concept of watchlists is more generally applicable and we use it in other plugins too. The key insights is the following. If we are interested in detecting when, of a set V of variables, exactly k variables are left unassigned, it is enough to "watch" a set $W \subseteq V$ of $(k + 1)$ variables by maintaining the invariant that all variables in W are unassigned. If a variable $x \in W$ becomes assigned, then we try to replace x with another unassigned variable $y \in V \setminus W$. If we can't find an unassigned variable y to replace x, this means that in V there are now exactly k variables unassigned.

B. Linear Real Arithmetic

The plugin for reasoning about linear arithmetic (FM plugin) is responsible for reasoning about linear constraints and deciding values for variables of real type. In order to maintain unit completeness for variables of the real type, we should ensure that for each real variable x, the set of all linear constraints from the trail M, that are unit in variable x, is consistent.

We call a literal $L \in M$ a linear constraint unit in x, if the atom of L is a linear constraint, and all variables of L different from x are unassigned in M. Any linear constraint $L \in M$, unit in x, can be equivalently written as one of

$$x \neq p, \qquad p \triangledown x, \qquad x \triangledown p \ ,$$

with $\triangledown \in \{<, \leq, =\}$. Since the constraint is unit, the polynomial p can be evaluated in M and takes some value $v \in \mathbb{Q}$.

Example IV.2. *Consider the trail*

$$M = [\![\neg(x+y<0),\ \mathbf{x}\mapsto\mathbf{0}, \neg(x+z=0), (0<y+z)]\!]\ .$$

The trail M contains two unit linear constraints. The literal $\neg(x+y<0)$ is unit in variable y, is equivalent to $(-x \leq y)$, and evaluates to $0 \leq y$. The literal $\neg(x+z=1)$ is unit in z, is equivalent to $z \neq 1-x$, and evaluates to $z \neq 1$. The linear constraint $(0<y+z)$ is not unit in M.

Using the watchlist mechanism, we can efficiently maintain an up-to-date set of linear constraints that are unit. The unit constraints in the trail impose constraints on the unit variables, and for each variable x, the FM plugin tracks the following

- the strongest lower bound of x implied by a unit linear constraint $L \in M$;
- the strongest upper bound of x implied by a unit linear constraint $L \in M$;
- a set of values v_D such that x is implied to be different from v_D by unit linear constraint $L \in M$.

Having the information above, for each variable x, we can now effectively reason about its unit feasibility by inspecting if there is a value within its upper and lower bound that is not disallowed by a disequality constraint. If, for some variable x, there is no such value, it must be due to a *bound conflict* or a *disequality conflict*.

Variable x is in a bound conflict if the trail contains two unit linear constraints $L_L \equiv (p_L\ \triangledown_L\ x)$ and $L_U \equiv (x\ \triangledown_U\ p_U)$, with p_L and p_U evaluating to v_L and v_U, where either $v_L > v_U$, or $v_L = v_U$ but at least one of the bounds $\triangledown_L$ or $\triangledown_U$ is strict ($<$). This conflict can be resolved using the resolveFM rule

$$\frac{}{\neg(p_L\ \triangledown_L\ x) \vee \neg(x\ \triangledown_U\ p_U) \vee (p_L\ \triangledown\ p_U)}$$

where $\triangledown$ is the result of combining $\triangledown_L$ and $\triangledown_U$. This clause can be used as an explanation of the conflict since the first two literals evaluate to **false**, and the last literal doesn't contain x and can be semantically propagated as **false**.

Variable x is in a disequality conflict if the trail contains a unit disequality constraint $(x \neq p_D)$, and two unit linear constraints $(p_L \leq x)$ and $(x \leq p_U)$, with p_D, p_L and p_U all evaluating to the same value v. This conflict can be resolved using a derived rule we call resolveDiseq with the derivation presented in Figure 2. This rule produces a clause that can be used as an explanation of the conflict since the first three literals evaluate to **false** and the last two literals can be semantically propagated as **false**. The resolveDiseq rule is applicable for disequality conflicts with unit equality constraints too, and although a more precise rule exists, we use this one for simplicity.

In addition to detecting conflicts, the FM plugin also eagerly performs semantic propagation. Using the same watch-list mechanism, the FM plugin tracks all linear constraints in the system, and can detect when a linear constraint becomes fully assigned. Such constraints are evaluated using the assignment in the trail and added to the trail as semantic propagations.

Computing bounds implied by unit constraints and performing semantic propagation of fully assigned linear constraints can be very expensive. The propagation loop of the FM plugin spends 90% or more of its time evaluating these constraints. In order to improve performance we use *value time-stamping* feature of the main solver. The main solver maintains a global ever-increasing time-stamp for decision values. Each variable x is associated with its own time-stamp, and the time-stamp of x gets assigned to the global time-stamp every time x is assigned to a value different from the value x was assigned in some previous attempt. This allows us to detect when a set of variables (say variables of a linear constraints) are assigned to the same values as some previous time we considered the set, by keeping the maximal time-stamp of those variables. This in turn allows us to cache bound computations and semantic evaluations of linear constraints, in cases when the same values were chosen.

If no conflicts were detected, the FM plugin is dedicated to picking the values of the real variables. In order to improve performance of arithmetic operations, when deciding on a value for a variable, if possible, we always choose the values to be dyadic rationals.[4] Additionally, if allowed by the bounds, when picking a value for a variable x, we try to use the value that was used for x previously (value-caching). This is a strategy similar to phase-caching in SAT solvers [20], and allows for better evaluation cache performance when using value time-stamping described above.

C. Uninterpreted Functions

Most decision procedures for uninterpreted functions are based on fast union-find algorithms complemented with congruence-closure reasoning [21], [22]. Instead, we adopt a very simple approach to reasoning about uninterpreted functions that detects direct conflicts in term assignments.

We say that a function term $f(x_1, \ldots, x_n)$ has an *evaluation representative* $f(\alpha_1, \ldots, \alpha_n)$ in a trail M, if each x_i is either the constant α_i, or is assigned by M to value α_i. For each uninterpreted functions term that appears in the input formula (generalized variables), we maintain a single-variable watchlist of it's non-constant arguments. This allows as to detect when all of the arguments of the function application have been assigned, and the term therefore has an evaluation representative. The UF plugin can then detect a conflict if two terms with the same evaluation representative are ever assigned to different values.

Example IV.3. *Consider the unit constraint $f(x) < f(y)$ and assume that the trail is in the state*

$$M = [\![(f(x) \overset{0}{<} f(y))_\downarrow,\ \mathbf{f(x)}\overset{1}{\mapsto}\mathbf{0},\ \mathbf{f(y)}\overset{2}{\mapsto}\mathbf{1},\ \mathbf{x}\overset{3}{\mapsto}\mathbf{0}]\!]\ .$$

In this state, the UF plugin knows that the arguments of $f(x)$ are fully assigned, with the evaluation representative $f(0)$, and is assigned to 0.

[4]Dyadic rationals $\mathbb{D} = \{\frac{p}{2^k} \mid p \in \mathbb{Z},\ n \in \mathbb{N}\}$ are a convenient dense sub-ring of $\mathbb{Q}$, allowing more efficient ring operations ($+$, $\times$) due to less gcd computation.

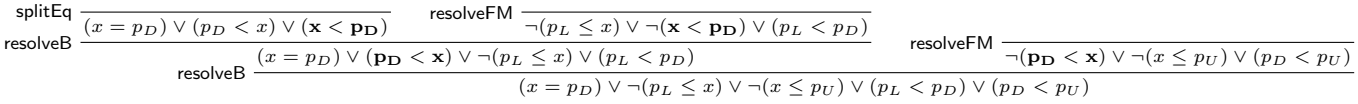

Fig. 2. Derivation of the disequality lemma.

We continue from this state to assign the next unassigned variable y, and the responsible plugin (FM) can assign it to any value, including

$$[\![M, \mathbf{y} \overset{4}{\mapsto} \mathbf{0}]\!] \ .$$

The UF Plugin now has enough information to detect a conflicting state: the term $f(y)$ has all arguments assigned, with the representative $f(0)$ that is already assigned to the value $0 \neq 1$. We can explain the conflict using the resolveCC *rule to obtain the explanation clause*

$$\frac{}{R \equiv \neg(x = y) \vee (f(x) = f(y))}$$

We can propagate the new literals semantically, adding $(x = y)_{\downarrow 4}$ and $\neg(f(x) = f(y))_{\downarrow 2}$ to the trail and marking a conflict with the clause R. The single top literal of R being **false** *makes this clause an UIP clause and the solver can then backtrack to resolve the conflict, obtaining*

$$[\![M, \neg(f(x) \overset{2}{=} f(y))_{\downarrow 2}, \ \neg(x \overset{3}{=} y)_{\downarrow R}]\!] \ .$$

With the new trail, the FM plugin can now make a more informed decision on the value of y, which will satisfy the constraints.

D. Finite Basis

In order to guarantee termination through Lemma III.1, we need to guarantee that starting from the initial problem, the literals that the procedure operates on can be bound to a finite set. New literals are only created by the plugins, particularly the FM and UF plugins, as part of clauses that explain conflicting situations. The UF plugin only creates new literals using the resolveCC rule, introducing new equalities over function terms and their arguments. Given that the number of function terms in the input problem is finite, and no new function terms are ever introduced, the number of new literals that the UF plugin can introduce is finite. As already shown in [10]–[12], fixing the decision order on variables of real type ensures that the number of new literals introduced by the FM plugin is also finite. The argument follows from the fact that the FM rules always introduces linear constraints from existing ones, with the top variable eliminated. In practice, however, we do not enforce a fixed variable order, as the flexibility in deciding variables is crucial for performance.

V. Experimental Results

We implemented the MCSat framework as an independent engine in the CVC4 [23] solver (reusing the basic infrastructure and the parser) with the code freely available, and we refer to this implementation as mcsat. [5]

In order to evaluate the new approach, we compared our implementation with several SMT solvers that support linear arithmetic and uninterpreted functions, namely cvc4 1.2 [23], z3 4.3.1 [24], mathsat 5.1.12 [25], and yices 1.0.38 [26]. All of theses solvers are DPPL(T) based and implement a variant of the simplex algorithm described in [27]. All experiments were preformed on AMD Opteron 250 2.4GHz machines with a timelimit of 30 minutes and memory limited to 2GB.

We first compared the solvers on a set of pure arithmetic benchmarks from the QF_LRA category of the SMTLIB library.[6] The results are presented in Table I and show that the new mcsat implementation is competitive with the other solvers, and even excels on some problems that are hard for the DPLL(T)-based simplex solvers (such as the clocksynchro examples).

We then evaluated the solvers on the benchmarks that combine linear arithmetic and uninterpreted functions. For this we combined the benchmarks from the QF_UFLRA and the QF_UFLIA categories of the STMLIB library, while changing all the integer problems into their real-relaxation counterpart.[7] The results are presented in Table II. The results on this set show a very robust performance of mcsat, with our implementation solving all problems in the least amount of time. Again, there is a category of problems (wisas) hard for the DPLL(T)-based solvers where mcsat excels.

VI. Conclusion

We presented the design and implementation of the model-based satisfiability calculus. The new solver can effectively reason in linear real arithmetic and uninterpreted functions, and performs competitive with existing solvers. We proposed a simple combination mechanism for uninterpreted functions, based on model filtering, that has proven to be competitive with more sophisticated theory-combination frameworks.

We see many exciting directions for future work. In addition to integrating and developing further the existing model-based decision procedures for integer and non-linear real arithmetic, we plan to develop a decision procedure for the theory of arrays based on [28]. We also plan to work on implementing theory propagation algorithms that have proved effective in the DPPL(T) framework, and to work on integration of existing DPLL(T) decision procedures (such as simplex) into the MCSat framework.

[5]Source code of the revision used in experiments is available at `https://github.com/dddejan/CVC4/tree/mcsat-fmcad2013` in the `src/mcsat` directory. Use with `cvc4 --enable-mcsat`.

[6]Available at `http://www.smt-lib.org/`.

[7]`sed -e s/Int/Real/g -e s/QF_UFLIA/QF_UFLRA/g`

TABLE I
COMPARISON OF MCSAT WITH OTHER SOLVERS ON QF_LRA BENCHMARKS.

	mcsat		cvc4		z3		mathsat5		yices	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
clocksynchro (36)	**36**	**123.11**	36	1166.55	36	1828.74	36	1732.59	36	1093.80
DTPScheduling (91)	**91**	**31.33**	91	72.92	91	100.55	89	1980.96	91	926.22
miplib (42)	8	97.16	**27**	**3359.40**	23	3307.92	19	5447.46	23	466.44
sal (107)	107	12.68	107	13.46	107	6.37	107	7.99	**107**	**2.45**
sc (144)	144	1655.06	144	1389.72	144	954.42	144	880.27	**144**	**401.64**
spiderbenchmarks (42)	42	2.38	42	2.47	42	1.66	42	1.22	**42**	**0.44**
TM (25)	25	1125.21	25	82.12	**25**	**51.64**	25	1142.98	25	55.32
ttastartup (72)	70	4443.72	72	1305.93	72	1647.94	72	2607.49	**72**	**1218.68**
uart (73)	73	5244.70	73	1439.89	73	1379.90	73	1481.86	**73**	**679.54**
	596	12735.35	**617**	**8832.46**	613	9279.14	607	15282.82	613	4844.53

TABLE II
COMPARISON OF MCSAT WITH OTHER SOLVERS ON QF_UFLRA BENCHMARKS.

	mcsat		cvc4		z3		mathsat5		yices	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
EufLaArithmetic (33)	33	39.57	33	49.11	**33**	**2.53**	33	20.18	33	4.61
Hash (198)	198	34.81	198	10.60	198	7.18	198	1330.88	**198**	**2.64**
RandomCoupled (400)	400	68.04	400	35.90	400	31.44	**400**	**18.56**	384	39903.78
RandomDecoupled (500)	500	34.95	500	40.63	500	30.98	**500**	**21.86**	500	3863.79
Wisa (223)	223	9.18	223	87.35	223	10.80	223	65.27	**223**	**2.80**
wisas (108)	**108**	**40.17**	108	5221.37	108	443.36	106	1737.41	108	736.98
	1462	**226.72**	1462	5444.96	1462	526.29	1460	3194.16	1446	44514.60

REFERENCES

[1] S. Malik and L. Zhang, "Boolean satisfiability from theoretical hardness to practical success," *Communications of the ACM*, vol. 52, no. 8, pp. 76–82, 2009.

[2] J. P. M. Silva and K. A. Sakallah, "GRASP – a new search algorithm for satisfiability," in *ICCAD*, 1997.

[3] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Design Automation Conference*, 2001, pp. 530–535.

[4] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 2, pp. 245–257, 1979.

[5] L. de Moura and N. Bjørner, "Model-based Theory Combination," in *Satisfiability Modulo Theories*, ser. ENTCS, vol. 198, 2008, pp. 37–49.

[6] D. Jovanović and C. Barrett, "Being careful about theory combination," *Formal Methods in System Design*, pp. 1–24, 2012.

[7] S. Krstić and A. Goel, "Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL," *Frontiers of Combining Systems*, pp. 1–27, 2007.

[8] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[9] L. de Moura and D. Jovanović, "A Model-Constructing Satisfiability Calculus," in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, vol. 7737, 2013, pp. 1–12.

[10] S. Cotton, "Natural domain SMT: A preliminary assessment," in *FORMATS*, 2010.

[11] K. L. McMillan, A. Kuehlmann, and M. Sagiv, "Generalizing DPLL to richer logics," in *Computer Aided Verification*, 2009, pp. 462–476.

[12] K. Korovin, N. Tsiskaridze, and A. Voronkov, "Conflict resolution," *Principles and Practice of Constraint Programming*, pp. 509–523, 2009.

[13] D. Jovanović and L. de Moura, "Cutting to the chase: Solving linear integer arithmetic," in *Automated Deduction*, 2011, pp. 338–353.

[14] ——, "Solving non-linear arithmetic," *Automated Reasoning*, pp. 339–354, 2012.

[15] L. Haller, A. Griggio, M. Brain, and D. Kroening, "Deciding floating-point logic with systematic abstraction," in *Formal Methods in Computer-Aided Design*, 2012, pp. 131–140.

[16] W. Ackermann, *Solvable cases of the decision problem*, 1954, vol. 12.

[17] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009.

[18] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *Computer-aided Design*, 2001, pp. 279–285.

[19] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and applications of satisfiability testing*, 2004, pp. 502–518.

[20] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *Theory and Applications of Satisfiability Testing*, 2007, pp. 294–299.

[21] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *Journal of the ACM*, vol. 27, no. 2, pp. 356–364, 1980.

[22] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *Journal of the ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[23] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*. Springer, 2011, pp. 171–177.

[24] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[25] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2013, pp. 93–107.

[26] B. Dutertre and L. D. Moura, "The yices SMT solver," *Tool paper at* `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[27] ——, "A fast linear-arithmetic solver for DPLL(T)," in *Computer Aided Verification*, 2006, pp. 81–94.

[28] R. Brummayer and A. Biere, "Lemmas on Demand for the Extensional Theory of Arrays," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 165–201, 2009.