Yaga

MCSat-based SMT Solver

Drahomír Hanák, Martin Blicha, and Jan Kofroň

June 13, 2023

1 Introduction

Boolean satisfiability problem (SAT) is to find a satisfying assignment of a propositional formula. It was among the first problems shown to be NP-complete. Despite its theoretical complexity, there has been a tremendous effort to create SAT solvers that could decide practical problems. This effort has been largely successful, as many modern SAT solvers are used today to solve problems in hardware, software verification, and bounded model checking, among others. They are usually based on the conflict-driven clause learning (CDCL) algorithm [14, 12, 15].

Satisfiability modulo theories (SMT) is an extension of the SAT problem to first-order logic [7]. SMT solvers typically utilize CDCL SAT solvers to find a satisfying assignment of a Boolean abstraction of the original formula. A theory-specific decision procedure is then used to either extend this assignment to theory variables or refute it. This approach is often called DPLL(T).

Model Constructing Satisfiability Calculus (MCSat) is an alternative to the DPLL(T) approach. It extends the CDCL framework used in many modern SAT solvers to SMT problems [7, 10]. Unlike DPLL(T), MCSat tries to construct a model directly in a specified theory. Advances in SAT solvers can thus be directly applied or adapted for MCSat.

Most SMT solvers are based on the DPLL(T) approach. MCSat has not attracted a lot of attention in the SMT community yet. The goal of this project is to create a prototype SMT solver based on the MCSat framework, implement a plugin for the theory of Linear real arithmetic (LRA), and evaluate the solver on the SMT-LIB benchmark [4].

1.1 Outline

Section 2 describes the MCSat framework, heuristic functions used by Yaga, and some experimental features we implemented. The architecture of the solver is described in Section 3. Section 4 evaluates Yaga on the SMT-LIB [4] benchmark for quantifier-free Linear real arithmetic (QF_LRA) and compares it with a DPLL(T)-based SMT solver.

1.2 Source code

Our prototype implementation of the SMT solver based on MCSat is publicly available[1] on GitHub.

The YAGA prerequisites include:

- C++ compiler with C++20 support
- `cmake` version at least 3.17
- `flex` version at least 2.6

To build YAGA, run the following commands:

1. `mkdir build-release`
2. `cd build-release`
3. `cmake -DCMAKE_BUILD_TYPE=Release ..`
4. `make`

1.3 Publication

We submitted YAGA to the SMT-COMP 2023 [3]. The solver implemented in our work is extensible to facilitate further research on decision procedures in the MCSat framework and the development of tools that use the solver as the back end.

[1] https://github.com/d3sformal/yaga

2 Model Constructing Satisfiability Calculus

The MCSat framework is split into several main components: the core solver, solver trail, clause database, theory-specific plugins, and heuristics (e.g., variable order, restart scheme, and clause minimization) [10]. The solver trail is a central solver component that records current progress. Theory plugins analyze the content of the trail and add new elements. They also detect inconsistencies and generate conflict clauses (false clauses in the current trail). Conflicts are analyzed by the core solver, which derives learned clauses and adds them to the clause database. The core solver also dispatches events in the system, like adding a new variable or a newly learned clause to all other components. Some decisions in the solver are controlled by heuristic functions (e.g., which variable should be decided next or whether we should restart).

2.1 Solver trail

The solver trail is a data structure that records the progress of the solver. It is a sequence of trail elements [10]:

- *decisions*: assignment of a value to a variable. We denote the decision of a variable x with $x \mapsto v$ where v is the new value of the variable x.
- *clausal propagations*: propagation of a literal by a Boolean constraint propagation (BCP). BCP propagates literal L as a clausal propagation if it detects a clause $C = L_1, \ldots, L_n, L$ such that $n \in \mathbb{N}_0$ and $\neg L_1, \ldots, \neg L_n$ are on the trail while L is unassigned (i.e., C is a unit clause).
- *semantic propagations* mark a literal, which represents a fully-assigned constraint. We say that a constraint is fully-assigned if all variables that appear in the constraint are assigned. For example, in Linear Real Arithmetic (LRA), a constraint $x < 0$, such that the rational variable x is assigned, can be evaluated. The LRA plugin has to propagate it to the trail as a semantic propagation.

In addition, we associate a decision level with each trail element. The decision level of a *decision* or *clausal propagation* is the number of decisions prior to and including the element [10]. The decision level of a *semantic propagation* is the highest decision level of any variable in the propagated literal. For example, in the trail $M = (x \mapsto 0, y \mapsto 1, z \mapsto 3)$, we can propagate $x + y \leq 2$ at the decision level 2 since that is the highest decision level of $\{x, y\}$.

In order to describe the properties of the trail, we borrow terminology used by De Moura et al. [7]. A trail is *consistent* if, for each literal on the trail, the value of the literal according to the current first-order model is true. For example, a trail $M = (x < 0, x \mapsto -1)$ is consistent because the literal $x < 0$ is true when $x \mapsto -1$. On the other hand, the trail would not be consistent if we replaced the decision $x \mapsto -1$ with $x \mapsto 0$. The solver trail may briefly become inconsistent when a plugin detects a conflict. In this case, the core solver restores consistency by backtracking (i.e., removing all trail elements above some decision level). A trail is *infeasible* if it contains a set of literals that is unsatisfiable given the current assignment of variables.

2.2 The core solver

The core solver implements an extension of the CDCL algorithm [10] (Algorithm 1).

Algorithm 1: The core solver loop

```
while true do
    conflicts ← propagate() ;
    if conflicts then
        learned, level ← analyze(conflicts) ;
        if any clause in learned is an empty clause then
            return unsat
        backtrack_with(learned, level) ;
    else
        if all variables have been decided then
            return sat
        else
            decide() ;
```

Theory plugins propagate literals to the trail in the `propagate()` method [10]. We run propagation in all plugins until no new elements are added to the trail. Plugins can also detect conflicts during propagation which are reported to the core

solver as conflict clauses. A conflict clause is a clause that is false in the current trail.

Conflict analysis

Plugins can return multiple conflict clauses. The core solver analyzes all conflicts in the `analyze()` method, which returns a list of newly learned clauses and a decision level to backtrack to. For the purposes of backtracking, we distinguish two types of learned clauses based on the top-level literals (set of literals in the learned clause with the highest decision level):

- *unique implication point* [14] (UIP) clause, which has exactly one top-level literal L. The core solver resolves UIP conflicts by backtracking to the second-highest decision level[1] and by propagating L.
- *semantic split* clause: all top-level literals are semantic propagations. Semantic split conflicts are resolved by backtracking to one level below the top level and deciding one of the top-level literals. Semantic split conflicts replace a decision of a non-Boolean variable with a decision of a Boolean variable [7].

The core solver can backtrack only with a UIP or semantic split clause. However, a conflict clause returned by a plugin may not satisfy these requirements. In order to derive a conflict clause suitable for backtracking, the `analyze()` method resolves top-level literals that were added to the trail by clausal propagations [10]. Algorithm 2 summarizes the conflict analysis procedure. The `reason()` function returns the clause that propagated the provided literal. The `resolve(`C`,` D`,` L`)` function returns the resolvent of the clauses C and D with respect to the literal L. This conflict analysis corresponds to the first UIP strategy [16, 10] used in many modern CDCL SAT solvers if the returned clause is a UIP clause.

If there are several conflict clauses, we choose conflicts that backtrack to the lowest decision level. The rest of the clauses are discarded. Moreover, if there are UIP conflict clauses and semantic split clauses that would backtrack to the same level, we only use the UIP clauses. The rationale for this is that UIP clauses lead to a propagation instead of a decision which is more valuable for the progress of the solver (we refer to the proof of MCSat termination [7] for more information). Thus, if the core solver is about to backtrack, it only has a list of one type of conflict clause (UIP or semantic split clauses). UIP conflicts are resolved by propagating all implied literals (i.e., we propagate the top-level literal from each conflict clause). Semantic split clauses are resolved by deciding one of the top-level literals from conflict clauses.

[1] The solver backtracks to level 0 if the conflict clause has only one literal.

Algorithm 2: Derive conflict clause suitable for backtracking [10]

input : Conflict clause C and the current solver trail M.
output: Conflict clause suitable for backtracking.
Function `analyze`(C, M)**:**
    $k \leftarrow$ size of M;
    while C *is not empty, UIP clause, or a semantic split clause* **do**
        $k \leftarrow k - 1$;
        if $M[k]$ *is a clausal propagation of* L *and* $\neg L \in C$ **then**
            $C \leftarrow$ `resolve`(C, `reason`(L), L);
    return C

Heuristics

The core solver uses several heuristic functions. Heuristics listen to events and try to decide some problems for which we do not have an optimal answer.

- **Variable order** chooses the next variable to decide in the `decide()` method. We also use this heuristic to determine which one of the top-level literals in a semantic split conflict will be decided next. Choosing the value of the selected variable is delegated to the theory-specific plugin responsible for that variable.
- **Restart policy** has one method, `should_restart()`, which is called whenever the core solver encounters a conflict. If this method returns true, the solver restarts instead of backtracking (i.e., it removes all elements from the trail; learned clauses are retained).
- **Clause deletion** deletes learned clauses it deems unnecessary on restart. Clauses should only be deleted on restart because other components may keep pointers to clauses.

2.3 Theory plugins

Theory plugins manage constraints and variables of some type. The primary purpose of plugins is to propagate literals and to detect conflicts. They are also responsible for deciding the values of managed variables. These responsibilities are concentrated in two main methods:

- `propagate()`: propagates implied literals to the trail. The plugin generates conflict clauses in case a conflict is detected. If a plugin is responsible for

deciding values of variables of some type, it should be *unit-constraint complete* [10]. A plugin is unit-constraint complete if each `propagate()` call either returns conflict clauses or, after the call, for each unassigned variable managed by the plugin, there exists a decision of this variable such that if we append this decision to the current trail, the trail remains consistent.

- `decide()`: decides a value of the provided variable such that the trail is consistent if we add this decision.

Moreover, all theory plugins in YAGA are event listeners, so they receive all notifications about events in the system.

2.4 Boolean plugin

The primary purpose of the plugin is to perform Boolean constraint propagation (BCP) to exhaustion. BCP finds all unit clauses in the system and propagates the implied literals to the trail. Clause $C = L_1 \vee \cdots \vee L_n \vee L$, $n \in \mathbb{N}_0$ is unit if all literals $\neg L_1, \ldots, \neg L_n$ are on the trail and L is unassigned. The unassigned literal L from a unit clause C can be propagated as a clausal propagation to the trail using C as the reason for the propagation. It is sufficient to perform BCP to exhaustion to satisfy the unit-constraint completeness requirement [10].

We use the mechanism of watched literals [12] to detect unit clauses. Two literals in each clause are designated as watched literals. We try to maintain a property that each watched literal L is non-falsified (i.e., $\neg L$ is not on the trail). When a watched literal L is falsified, the plugin tries to find some non-falsified replacement in the clause. If there is no suitable replacement, we can detect unit or empty clauses:

- The other watched literal is unassigned. In this case, the clause is unit, and the plugin propagates the only unassigned literal in the clause to trail using a clausal propagation.
- The other watched literal is false. In this case, the clause is false, and the plugin stops the unit propagation process and returns this clause as a conflict.
- The other watched literal is true. In this case, we do not even try to replace L as the watched literal since the clause is already satisfied.

Similarly to MiniSat [15], we move the watched literals to the first two positions in each clause. We maintain a map which maps a literal to the list of clauses in which it is watched. Moreover, we cache the last checked position in each clause.

When a literal is falsified, the search for a new non-falsified literal to watch starts from the cached position. This is to avoid skipping past a long list of falsified literals every time we falsify a watched literal in a long clause.

2.5 Linear real arithmetic

The plugin for deciding Linear real arithmetic (LRA) is responsible for rational variables and linear constraints. Linear constraints are equalities and inequalities ($<, \leq, =, \neq, >, \geq$) on linear polynomials.

We internally represent linear constraints in a normalized form $\sum_{i=1}^{n} x_i c_i \triangledown c$ where $x_1, \ldots, x_n$ are rational variables with non-zero coefficients $c_1, \ldots, c_n \in \mathbb{Q} \setminus \{0\}$, $c \in \mathbb{Q}$ is the constant term in the linear polynomial, and $\triangledown$ is one of the predicates $<, \leq, =$. The rest of the predicates ($>, \geq, \neq$) can be obtained by negating a normalized linear constraint. Each constraint is associated with a Boolean variable and can be (uniquely) identified by a literal. While the general form of a linear constraint is normalized, the variables $x_1, \ldots, x_n$ can be stored in any order to facilitate finding unit linear constraints (i.e., constraints with exactly one unassigned variable) by swapping the variables as they are assigned.

Variable bounds

When a linear constraint on the trail becomes unit, it implies a bound for its only unassigned rational variable. We use a system of watched variables [10] similar to watched literals [12] (Section 2.4) to detect unit linear constraints. The first two variables in each constraint are the watched variables. If either of the watched variables is assigned, we try to find a new unassigned variable to watch. If there are no other unassigned variables, the constraint is either unit (in case the other watched variable is unassigned), or fully-assigned (in case the other watched variable is assigned). Similarly to watched literals, we also cache the last checked position in each constraint. The search for an unassigned variable starts from this cached position.

The LRA plugin keeps track of an interval of values that can be assigned to each rational variable [10] and a list of disequalities. Bounds of this interval may change when an inequality from the trail becomes unit since unit constraints imply a bound for the only unassigned variable (we treat equalities as two inequalities $\leq, \geq$). The plugin also propagates fully-assigned linear constraints to trail using semantic propagation.

$$\neg(p_L \bigtriangledown_L x) \vee \neg(x \bigtriangledown_U p_U) \vee p_L \bigtriangledown_R p_U$$

Figure 2.1: Explanation of a bound conflict [10].

Conflict analysis

If the interval of allowed values for a rational variable shrinks to an empty set, it has to be either due to a bound conflict or due to a disequality conflict [10]. We say that variable x is in a bound conflict if there are two linear constraints on the trail: a lower bound $p_L \bigtriangledown_L x$ and an upper bound $x \bigtriangledown_U p_U$ such that p_L, p_U are linear polynomials which evaluate to $l, u \in \mathbb{Q}$ in current trail, respectively, and $l > u$ or $l = u$ and at least one of the constraints $\bigtriangledown_L, \bigtriangledown_U$ is strict (i.e., $<$).

Bound conflicts can be explained by Fourier-Motzkin elimination of x from the bounds [10]. Figure 2.1 shows the explanation clause. The $\bigtriangledown_R$ predicate is a combination of $\bigtriangledown_L$ and $\bigtriangledown_U$ (i.e., $\bigtriangledown_R$ is $\leq$ if none of the bounds are strict and it is $<$ otherwise).

We say that a rational variable x is in a disequality conflict [10] if there are three inequalities on the trail: a lower bound $p_L \leq x$, an upper bound $x \leq p_U$, and a disequality $x \neq p_D$ such that p_L, p_U, and p_D evaluate to the same value in current trail. Disequality conflict can be explained using the disequality lemma (Figure 2.2).

$$x = p_D \vee \neg(p_L \leq x) \vee \neg(x \leq p_U) \vee p_L < p_D \vee p_D < p_U$$

Figure 2.2: Explanation of a disequality conflict [10].

A unit equality $x = p_E$ where p_E is a linear polynomial implies both a lower bound and an upper bound for the rational variable x. There is a special case of the disequality conflict if both the lower bound and the upper bound come from the same equality $x = p_E$ such that p_E and p_D evaluate to the same value in current trail. We explain this special case with the clause in Figure 2.3 which is equivalent to the clause in Figure 2.2 if we substitute p_L and p_U with p_E. However, it does not include any new literals that could not be evaluated. The first two literals are negations of literals that are on the trail. The rest of the clause $p_E < p_D \vee p_D < p_E$ is equivalent to $p_E \neq p_D$ in LRA and it evaluates to false in current trail since p_E, p_D evaluate to the same value.

$$x = p_D \vee x \neq p_E \vee p_E < p_D \vee p_D < p_E$$

Figure 2.3: Explanation of a disequality conflict if lower bound and upper bound of x come from the same equality.

2.6 Bound caching

The LRA plugin maintains a cache of computed bounds for each rational variable. It is implemented with a stack of computed lower bounds, upper bounds, and a list of disequalities for each rational variable. Equality is treated as a lower bound and an upper bound. New bounds are added to the appropriate stack when they are first computed. We lazily remove values from the top of each stack so that the solver does not have to do extra work when it backtracks.

New bounds are compared with the strongest bound that is already in the cache. A new bound is only added to the appropriate stack if the stack is empty or the new bound improves the old bound. We say that a new lower bound improves an old lower bound if the value of the new bound is strictly greater than the old bound or they are equal, and the old bound is not strict ($\geq$) while the new bound is strict ($>$). We use a symmetric predicate to order upper bounds. If a new bound does not improve the strongest old bound at the top of the stack, it is redundant, and it will not be needed to derive a conflict.

We use variable timestamps [10] to detect that a value of a variable has changed. The core solver has a global, ever-increasing integer which is incremented whenever any variable is assigned a value. Moreover, we store a timestamp of each variable. When a variable is assigned a value, its timestamp is set to the global timestamp. We update the timestamp of a variable even if it is assigned the same value. To detect whether a value of any variable from some set has changed, it is sufficient to cache the maximal timestamp of variables in the set and later compare the stored maximum with the current variable timestamps.

We use timestamps to detect obsolete bounds in Algorithm 3. It implements a function that returns *true* if the provided bound is obsolete. The `is_unassigned()` function checks whether the provided variable is assigned a value in the current trail. The `timestamp()` function returns the current timestamp of the variable. In addition to the computed value of the bound, we also keep the linear constraint from which we derived the bound (*reason*) in the normalized form (Section 2.5), its timestamp[2] (T_b), and the timestamp of the most recently assigned rational variable from *reason* (T_r). The second variable in a constraint is the second

[2]We use the timestamp of a linear constraint interchangeably with the timestamp of its Boolean variable.

watched variable, the most recently assigned rational variable in the constraint (we utilize an invariant of watched constraints from Section 2.5) when the bound was computed. The value of any assigned, rational variable in *reason* can only be changed after backtracking the decision of this variable. Consequently, we do not have to check any other variable from *reason* to determine whether the bound is obsolete.

Algorithm 3: Detect obsolete bounds

input : Linear constraint from which we derived the bound *reason*, cached timestamp of the Boolean variable of *reason*: T_b, and a cached timestamp of the second watched rational variable T_r

output: *true* if the bound is obsolete, $false$ otherwise.

Function `is_obsolete`(*reason*, T_b, T_r)**:**

```
if is_unassigned(reason) ∨ timestamp(reason) ≠ T_b then
    return true
vars ← variables(reason) ;
if size(vars) ≤ 1 then
    return false
return is_unassigned(vars_1) ∨ timestamp(vars_1) ≠ T_r
```

Algorithm 3 requires incrementing the variable timestamp even if the variable is assigned the same value. Consider the following trail $M = (x + y + z \leq 0, z \mapsto 0, y \mapsto 0)$ which implies an upper bound $x \leq 0$ with `timestamp(z) = 2` and `timestamp(y) = 3`. If we backtrack the last two decisions and replace them with $z \mapsto 1$ and $y \mapsto 0$, `timestamp(y)` would be unchanged, but the value of z had changed, and the cached bound is no longer valid. However, Algorithm 3 would not detect the change.

To determine the current lower bound or upper bound of a variable, we first have to remove obsolete bounds from the top of the stack using Algorithm 3. To check whether there is a disequality $x \neq v$ for some rational variable x and $v \in \mathbb{Q}$, we try to find v in the disequality list of x. If v is in the disequality list and it is not obsolete according to Algorithm 3, there is a valid disequality $x \neq v$. The LRA plugin does not have to do anything with variable bounds after backtracking, which simplifies the implementation.

2.7 Derivation of new bounds

New bounds are derived from unit linear constraints only (i.e., after most variables from the constraint are decided). It is possible to derive new bounds earlier by

eliminating some variables using Fourier-Motzkin (FM) elimination. For example, if $x + y = 0$ and $y = 0$ are on the trail, but both variables x and y have yet to be decided, we can derive $x = 0$. For another example with inequalities, if $x + y \leq 0$ and $y > 0$ are on the trail, we can derive $x < 0$. In the rest of this section, we assume that all linear constraints are inequalities ($<, \leq, >$, or $\geq$). Equalities can be treated as two inequalities ($\leq, \geq$). We do not derive new constraints from disequalities ($\neq$) even though it may be possible in some cases (e.g., if $x + y \neq 0$ and $y = 0$ are on the trail, we could derive $x \neq 0$).

When a linear constraint is added to the trail, we try to eliminate all bounded variables using bounds from the bound cache (Section 2.6). We say that a variable in a linear constraint is *bounded* if it is not assigned, and we have a lower bound or an upper bound in the bound cache that can be used to eliminate the variable from the constraint using FM elimination.

If this process creates a unit linear constraint (i.e., there is exactly one unassigned variable), we can derive a new bound. We only compute the bound value after FM elimination to avoid unnecessarily creating new linear constraints (Algorithm 4). The `rhs()` function returns the constant term in the linear constraint on the right-hand side of the inequality. The `predicate()` function returns the predicate of the linear constraint ($<, \leq, >, \geq, =$, or $\neq$). The `value()` function returns the cached bound value if applied to a bound or the value of a variable if applied to an assigned variable. The `lower_bound()` and `upper_bound()` functions return the best lower and upper bound from the bound cache, respectively. These functions return `nullptr` if there is no lower or upper bound.

Provided there is exactly one unassigned variable x in the linear constraint after FM elimination of bounded variables, Algorithm 4 returns the bound value (i.e., a constant $b \in \mathbb{Q}$ such that $x \triangledown b$ is implied in the current trail where $\triangledown$ is the predicate of the bound after FM elimination $<, \leq, >, \geq$).

Producing explanations

Each bound in the bound cache (Section 2.6) additionally has a list of other bounds which were used to eliminate some rational variables. These dependencies between bounds essentially form a directed acyclic graph. If a bound is involved in a conflict and it is not a leaf in this graph, we eliminate bounded variables using FM elimination to produce an explanation for the bound [7].

The FM elimination is applied recursively (i.e., in order to eliminate bounded variables in a bound we first have to eliminate all bounded variables in its dependencies). Let $x \leq u$ be an upper bound such that $u \in \mathbb{Q}$, $D_1, \ldots, D_n$ are all distinct linear constraints of bounds reachable in its dependency graph, and $x \leq p_U$ is the final result of FM elimination (i.e., p_U is a linear polynomial which evaluates to u in current trail). Then $D_1 \wedge \cdots \wedge D_n \rightarrow x \leq p_U$ is a valid explanation for the

Algorithm 4: Compute the value of a bound

input : Linear constraint c with predicate $<$, $\leq$, $>$, or $\geq$ which is on the trail.

output: Computed value of the bound provided there is exactly one unassigned variable in c after FM elimination of bounded variables.

```
Function compute_bound(c):
    ret ← rhs(c) ;
    for var, coef ← variables(c) do
        if is_unassigned(var) then
            bound ← nullptr;
            if ((predicate(c) is < or ≤) ∧ coef > 0) ∨
            ((predicate(c) is > or ≥) ∧ coef < 0) then
                bound ← lower_bound(var)
            else
                bound ← upper_bound(var)
            if bound ≠ nullptr then
                ret ← ret − coef ∗ value(bound)
        else
            ret ← ret − coef ∗ value(var)
    return ret
```

upper bound. This can be shown by applying the FM elimination rule [10] and Boolean resolution to resolve intermediate results. This explanation can be used even if there are no bounded variables in the linear constraint. In that case, the explanation is just a tautology $\neg(x \leq p_U) \vee x \leq p_U$. We can derive a similar clause for a lower bound.

For example, assume we have a trail $M = (x + y \leq 0,\ y + z \geq 0,\ z \leq 0)$. It implies an upper bound $z \leq 0$, a lower bound $y \geq 0$ which depends on $z \leq 0$, and an upper bound $x \leq 0$ which depends on $y \geq 0$. The following process produces an explanation for $x \leq 0$ (i.e., a clause that would make the trail $(M, \neg(x \leq 0))$ infeasible):

1. FM elimination of z from $y + z \geq 0$ (this is also an explanation for $y \geq 0$): $\neg(y + z \geq 0) \vee \neg(z \leq 0) \vee y \geq 0$
2. FM elimination of y from $x + y \leq 0$: $\neg(x + y \leq 0) \vee \neg(y \geq 0) \vee x \leq 0$
3. Boolean resolution of the previous two clauses using the literal $y \geq 0$ produces $\neg(x + y \leq 0) \vee \neg(y + z \geq 0) \vee \neg(z \leq 0) \vee x \leq 0$

The final clause in step 3 is the explanation for $x \leq 0$. The first three literals in the explanation are negations of literals that are on the trail. The final literal is the new bound $x \leq 0$.

Bound conflict

Rational variable x is in a bound conflict (Section 2.5) if there is a lower bound $l \in \mathbb{Q}$ and an upper bound $u \in \mathbb{Q}$ such that $l > u$ or $l = u$ and at least one of the bounds is strict. The following process shows how to find an explanation for the bound conflict in the presence of derived bounds. For brevity, we will assume that the bounds are not strict (the process is analogous for strict bounds).

1. Find an explanation for the upper bound as in the previous section. This produces a clause $\neg D_1 \vee \cdots \vee \neg D_n \vee x \leq p_U$ such that p_U is a linear polynomial which evaluates to u in the current trail and $D_1, \ldots, D_n$ are linear constraints from the dependency graph of the upper bound.

2. Find an explanation for the lower bound as in the previous section. This produces a clause $\neg D'_1 \vee \cdots \vee \neg D'_m \vee p_L \leq x$ such that p_L is a linear polynomial which evaluates to l in the current trail and $D'_1, \ldots, D'_m$ are linear constraints from the dependency graph of the lower bound.

3. FM elimination of x: $\neg(p_L \leq x) \vee \neg(x \leq p_U) \vee p_L \leq p_U$

4. Resolve the clause in step 1 with the clause in step 3 using the literal $x \leq p_U$

5. Resolve the clause in step 2 with the clause in step 4 using the literal $p_L \leq x$

The result of this process is the clause in Figure 2.4. All literals in this clause except the last one are negations of literals from the trail. Moreover, the last literal is a new constraint that does not contain unassigned variables, and it evaluates to false in the current trail since we assumed a bound conflict.

$$\neg D_1 \vee \cdots \vee \neg D_n \vee \neg D'_1 \vee \cdots \vee \neg D'_m \vee p_L \leq p_U$$

Figure 2.4: Explanation of a bound conflict if there are derived bounds.

Disequality conflict

To explain a disequality conflict (Section 2.5), we follow a similar procedure as for bound conflicts. We resolve explanation of the upper bound $\neg D_1 \vee \cdots \vee \neg D_n \vee x \leq$

p_U with the disequality lemma (clause in Figure 2.2) derived by Jovanović et al. [10] (using the literal $x \leq p_U$) and then we resolve the result with the explanation of the lower bound $\neg D'_1 \vee \cdots \vee \neg D'_m \vee p_L \leq x$ (using the literal $p_L \leq x$). The final explanation is the clause in Figure 2.5.

$$x = p_D \vee \neg D_1 \vee \cdots \vee \neg D_n \vee \neg D'_1 \vee \cdots \vee \neg D'_m \vee p_L < p_D \vee p_D < p_L$$

Figure 2.5: Explanation of a disequality conflict if there are derived bounds.

2.8 Heuristics

Variable order

YAGA uses a generalization of the variable state independent decaying sum (VSIDS) heuristic [15, 12] to order variables. We associate a score with each variable in the solver regardless of its type. When a variable participates in a conflict, its score is increased. Specifically, we increase the score of each variable in the learned clause and any other clause that was resolved with the conflict clause in conflict analysis. Variable scores decay (i.e., they are decreased) after each learned clause.

To avoid decreasing the scores of all variables after each learned clause, we instead increase the amount by which variable scores are increased [15] by 5%. This is roughly equivalent to the method described in the previous paragraph. An advantage of this approach is that we only have to change the scores of variables involved in the conflict. However, since we use a finite precision type for the scores, they would eventually overflow, so we have to rescale all variable scores when they become too large.

We implemented a heap to find the variable with the highest VSIDS score in a logarithmic time. Moreover, the data structure has a table with the position of each variable in the heap so we can update variable scores efficiently. Initially, all variables are added to the heap, and the VSIDS score is initialized to the number of occurrences of that variable in the input formula. Assigned variables from the top of the heap are removed in the `decide()` method (Algorithm 1). Variables are re-added to the heap when the solver backtracks.

Clause deletion

Each conflict in the solver generates at least one new learned clause. We delete subsumed clauses [8] when the solver restarts to keep the clause database size

manageable. Clause A *subsumes* clause B if $A \subseteq B$. In that case, we also say that B is subsumed (by A).

YAGA indexes all learned clauses in the clause database by building a map `occur` from literals to the list of clauses in which the literal occurs. It additionally computes signatures of all learned clauses [8]. A signature is a 64-bit unsigned integer defined in Equation 2.1. The $h(L)$ function computes a hash of the literal L, and $\oplus$ is the bitwise or operation.

$$sig(\{L_1, \ldots, L_n\}) = 2^{h(L_1) \mod 64} \oplus \cdots \oplus 2^{h(L_n) \mod 64} \tag{2.1}$$

To check whether $A \subseteq B$, YAGA first compares the sizes of the input clauses and their signatures. If $|A| > |B|$, A cannot be a subset of B. Similarly, if $sig(A)\,\&\sim sig(B) \neq 0$ [3], there is a literal in A, which is not in B, so $A \not\subseteq B$. We have to perform the expensive subset check otherwise.

Algorithm 5: Mark all subsumed clauses [8].

input : Clause C.
output: All clauses subsumed by C are marked.
Function `mark_subsumed(`C`):`
  if C *is marked* **then**
    return
  $L \leftarrow$ literal from C with the shortest `occur` list;
  for $C' \in$ `occur(`L`)` **do**
    if $C \neq C' \wedge$ `subsumes(`C`, `C'`)` **then**
      mark that C' is a subsumed clause;

Clauses are deleted using a mark-and-sweep approach. The `mark_subsumed` function from Algorithm 5 marks all clauses subsumed by clause C. Subsumed clauses are marked by making them empty. Hence, if a marked clause is used as the second argument of `subsumes()`, the function quickly returns false without going through the literals of the first clause. The final phase then deletes all marked clauses from the database.

Let N be the set of newly learned clauses since the last restart and O the set of old learned clauses (i.e., the rest of the learned clauses). For all old clauses $A, B \in O$, `subsumes(`A, B`)` returns false because we delete all subsumed clauses. Consequently, we only have to check whether a new clause subsumes an old clause or whether any clause subsumes a new clause. This is achieved by the following process (the index operation creates the `occur` map from specified clauses):

[3] & is the bitwise and operation, and $\sim$ is the bitwise negation

1. Mark old clauses subsumed by a new clause: index all clauses in O and then run `mark_subsumed()` for all clauses in N.

2. Mark new clauses subsumed by any clause: index all clauses in N and then run `mark_subsumed()` for all learned clauses in $O \cup N$.

Learned clause minimization

We use self-subsuming resolution [15, 8] to minimize learned clauses. Clause A is *self-subsumed* by clause B with respect to the literal L if `resolve(`A, B, L`)` $\subseteq A$ where `resolve(`A, B, L`)` is the resolvent of A and B.

Yaga minimizes learned clauses using self-subsuming resolution by removing all literals L such that the learned clause is self-subsumed by `reason(`$\neg L$`)` [15] where the `reason()` function returns the reason clause for the propagation of $\neg L$. For example, if we have a conflict clause $L \vee L_1 \vee L_2$ and the reason for propagating $\neg L$ is the clause $\neg L \vee L_1$, then the resolvent of these two clauses is $L_1 \vee L_2$ which is a strict subset of the conflict clause $L \vee L_1 \vee L_2$, and so the literal L is redundant.

Restart strategy

Yaga's restart strategy is based on the heuristic used by the Glucose solver [5]. The Glucose solver computes clause glucose level (LBD) when it learns a new clause. LBD is the number of distinct decision levels of Boolean variables in the clause. The main idea is to try to only learn good clauses with respect to their LBD (i.e., clauses with a small LBD). It maintains a global average of all LBDs and a moving average of recent LBDs. The solver restarts when the moving average exceeds the global average by some threshold.

We use a more straightforward implementation, which maintains two exponential moving averages of LBDs with different parameters [6]. Both averages are initialized with a zero. The formula to update the exponential moving average ema given LBD of the current learned clause x is $ema \leftarrow ema + \alpha \cdot (x - ema)$ where α is a parameter. We use $\alpha = 2^{-13}$ for the global average and $\alpha = 2^{-5}$ for the recent average. Yaga restarts when the recent average exceeds the global average by 30%. However, we also have a minimal number of conflicts since the last restart (50) before the solver can restart again.

Variable values

The LRA plugin caches previously decided values of rational variables. Algorithm 6 shows the process of selecting a value for an unassigned rational variable x if the current solver trail is M. The `lower_bound()` and `upper_bound()` functions return

the strongest cached lower bound and upper bound, respectively. The `value()` functions returns computed value of a cached bound.

Algorithm 6: Decide a value of a rational variable.

input : An unassigned rational variable x and the current solver trail M.
output: Value $v \in \mathbb{Q}$ such that $(M, x \mapsto v)$ is consistent.
Function `decide(`x, M`):`

```
if we have a cached value v and (M, x ↦ v) is consistent then
    return v
if x can be assigned an integer then
    return the smallest integer v ∈ ℤ (according to the absolute value)
      such that (M, x ↦ v) is consistent.
l ← value(lower_bound(x));
v ← value(upper_bound(x));
while (M, x ↦ v) is not consistent do
    v ← (l + v)/2 ;
return v
```

3 Architecture

Yaga consists of two main parts: frontend and backend. The frontend part contains an abstract representation of terms in a tree structure. Another part of the frontend parses formulas in the SMT-LIB format to this abstract representation. SMT-LIB commands like checking satisfiability are delegated to the backend.

The backend part contains the core solver and theory plugins. It uses a different theory-specific representation of constraints that is more restrictive. This allows us to have a representation of constraints most suitable for each plugin, whereas the abstract representation can be used for preprocessing and high-level optimizations.

3.1 Integration

The backend can be used through a facade interface `YAGA` which hides initialization and implementation details. The facade contains methods for creating variables of specified type, constraints, and clauses. The only types returned by the facade are `Variable` (internal representation of a variable: ordinal number and a variable type), `Literal` (Boolean variable or its negation), or a `Clause` (disjunction of literals). Functions that create a specialized constraint return a literal representing the constraint.

The facade constructor has one argument, which is an initializer. Initializer instantiates all the necessary types for the specified theory. We predefined a couple of initializes:

- `logic::qf_lra` creates an MCSat backend for quantifier-free linear real arithmetic.
- `logic::propositional` creates a SAT solver based on MCSat.

3.2 Backend

Figure 3.1 summarizes Yaga's backend architecture. The core solver implements Algorithm 1. It maintains the current solver state (solver trail and clause database)

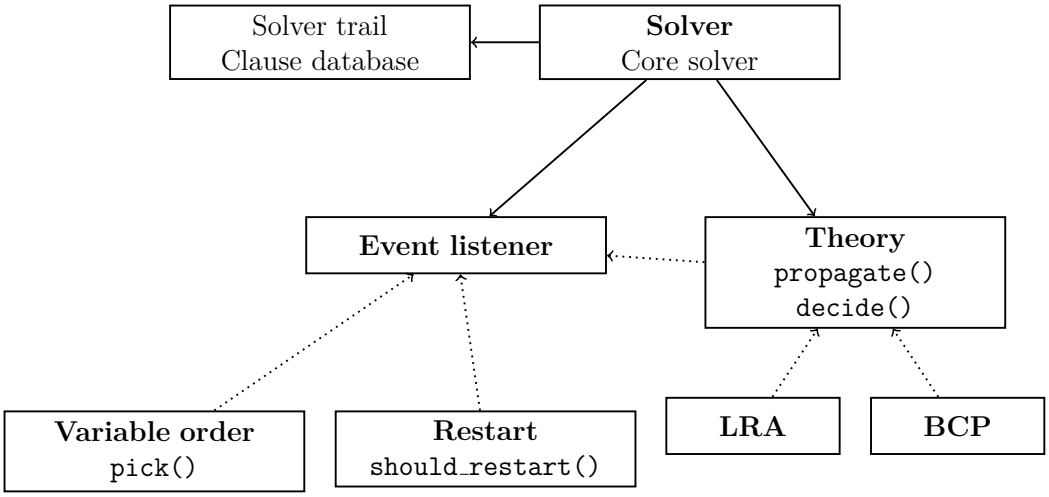

Figure 3.1: Overview of the backend architecture.

which is passed to event listeners and theory plugins. The way plugins and event listeners interface with the solver is through the solver state by analyzing the content of the trail added by other components and adding new elements.

The core solver calls `propagate()` in all plugins. If a plugin returns some conflict clauses, they are analyzed by the `Conflict_analysis` class as described in Section 2.2. If no conflict is detected during propagation, YAGA finds an unassigned variable to decide using the `Variable_order` heuristic and calls `decide()` with the selected variable in all plugins.

The core solver also dispatches events to all event listeners in the system. Heuristic functions such as variable order (`Variable_order`) and restart strategy (`Restart`) as well as theory plugins (`Theory`), are all event listeners.

3.3 Solver state

The solver state is the solver trail and the database of asserted and learned clauses. Solver trail records all decisions and propagations. We store variable values separately from trail elements. We have a table for each variable type that assigns a value to a variable (`Model`). The `trail.model()` method returns the current assignment of variable values of a specified type. In order to propagate or decide a value of a variable, a trail element has to be added, and a new value has to be set in the appropriate model. Figure 3.2 shows how to propagate a Boolean variable. Adding a new decision is similar, except we call the `decide()` method instead of the `propagate()` method.

Clausal and semantic propagations use the same representation on the trail.

```
trail.propagate(Variable{var_ord, Variable::boolean}, reason, level);
trail.model<bool>(Variable::boolean).set_value(var_ord, true);
```

Figure 3.2: Propagation of the boolean variable `var_ord` at the decision level `level` with a reason clause `reason`.

They can be distinguished by querying the reason for propagation. The method `trail.reason()` returns `nullptr` for all semantic propagations and a non-null pointer to a clause for any clausal propagation.

The clause database is a list of asserted and learned clauses. Asserted clauses are clauses from the input formula, and the backend should not modify them. When a plugin detects a conflict, it returns a conflict clause which the core solver subsequently analyzes. The result of this analysis is added to the learned clauses by the core solver. Components can keep references to clauses in the database. Adding new clauses does not invalidate references to any clause. However, references to all clauses are invalidated when the solver restarts.

3.4 Theory plugins

All theory plugins extend the `Theory` class. The core solver has a `set_theory()` method, which creates a new plugin used by the core solver. The core solver works with only one plugin. However, we implemented `Theory_combination` to combine several plugins, which is itself a `Theory`. It delegates method calls to all plugins. Moreover, the `propagate()` method performs propagation to exhaustion (i.e., it calls `propagate()` in all plugins until no new elements are added to the trail or until a plugin detects a conflict).

The `decide()` method is called when an unassigned variable is selected by the core solver to be decided. It is called in all plugins regardless of the type of the variable. The plugin responsible for the selected variable decides a value in the current solver trail.

Boolean plugin

The `Bool_theory` class implements the Boolean constraint propagation. It is also responsible for all Boolean variables. The default value used for all variables in the `decide()` method is true. However, we also implemented the phase-saving [13] heuristic, which caches values of boolean variables and uses the cached value in the `decide()` method. It can be enabled by calling the `set_phase(Phase::cache)` method.

Linear real arithmetic

The `Linear_arithmetic` class implements the Linear real arithmetic (LRA) plugin. The plugin itself consists of several smaller parts:

- `Linear_arithmetic`: implements the system of watched constraints (Section 2.5) to detect the unit and fully assigned constraints.
- `Linear_constraints`: is a repository of normalized linear constraints. It stores data of all linear constraints in the system. It also detects duplicates. The `Linear_constraint` class is a lightweight type containing a reference to the data in this repository and a literal identifying the constraint.
- `Bounds`: is a map from rational variables to implied bounds `Variable_bounds` (Section 2.6). It also contains methods for deducing new bounds.
- `Lra_conflict_analysis`: implements the Fourier-Motzkin elimination and procedures for explaining bound and disequality conflicts.

The numbers in YAGA are represented by the `Rational` class. This is a wrapper class for two classes – `Fraction` and `Long_fraction`. While the `Fraction` template class can represent numbers of a limited precision, in particular as a ratio of two instances of the template type, `Long_fraction` employs the GMP library [1] which allows for representation of rational numbers of unbounded precision.

To be efficient,`Long_fraction` attempts to store the fraction as two 32-bit integers. If either integer overflows, it switches to a representation from the GMP library.

Both `Fraction` and `Long_fraction` implement arithmetic operators to make their instances easy to use, such as addition, subtraction, multiplication, division, floor and ceil operations, as well as comparison operators.

New linear constraints are created by calling the `constraint()` method of the `Linear_arithmetic` class. This method creates a new Boolean variable if the provided constraint is not in the system and adds it to the list of watched constraints. The LRA plugin adds new constraints using this method only when there is a bound or disequality conflict.

3.5 Events

Event listeners inherit from the base class `Event_listener`. The core solver currently dispatches the following events:

- `on_init`: called when the core solver starts a new check.

- `on_variable_resize`: called when some variables are added or removed.
- `on_before_backtrack`: called before the solver backtracks.
- `on_learned_clause`: called when a new clause is learned and added to the clause database.
- `on_conflict_resolved`: called for each clause resolved with a conflict clause.
- `on_restart`: called after the solver restarts.

Heuristic functions

Heuristic functions extend the event listener interface. The `Variable_order` class additionally has a `pick()` method, which finds an unassigned variable to decide. The variable order used by the heuristic is determined by its `is_before(a, b)` method, which should return true if the variable `a` should be decided before the variable `b`. The core solver uses the `is_before()` method in a semantic split conflict to find the best literal to decide among the top-level literals in a conflict clause (Section 2.2). The order of variables can dynamically change.

We implemented a generalization of the VSIDS heuristic [12] for MCSat in the `Generalized_vsids` class. It internally uses a d-ary heap to order all variables by their score (`Variable_priority_queue`).

The `Restart` class is the interface for all restart strategies. It has only one method `shoud_restart()`, which is called when the core solver encounters a conflict. If it returns true, the solver restarts instead of backtracking. We implemented a couple of restart strategies:

- `No_restart` disables restarts.
- `Luby_restart` uses the Luby sequence [11, 9] multiplied by an integer constant (a parameter of the heuristic) to determine the number of conflicts before a restart.
- `Glucose_restart` implements a variant of the restart strategy used in the Glucose SAT solver [5, 6] (Section 2.8).

Clause deletion and minimization are implemented in the `Subsumption` class. Clause deletion does not have a specialized interface. Clauses are deleted in the `on_restart()` event handler method. Clause minimization is done explicitly by calling the `minimize()` method with a learned clause. It uses self-subsuming resolution (Section 2.8) to remove redundant literals from the learned clause.

4 Evaluation

In this section, we evaluate Yaga on the quantifier-free Linear real arithmetic (QF_LRA) SMT-LIB benchmark [4] and compare it with the OpenSMT solver [2] (a DPLL(T)-based solver).

We use a randomly selected sample (256 problems) with equal amounts of satisfiable and unsatisfiable instances from the QF_LRA category to evaluate the experimental features (propagation of rational variables and derivation of bounds). The final configuration of Yaga is evaluated on the whole benchmark. We use a timeout of 1200 seconds in all experiments.

4.1 Rational variable propagation

It is common in some problems to have rational variables with only one allowed value (e.g., there could be $0 \leq x$ and $x \leq 0$ on the trail or even $x = 0$). It might be beneficial to prioritize these variables in the `decide()` method (Algorithm 1) since decided variables could lead to more variable bounds and, thus, earlier conflict detection. We extended the LRA plugin and the variable order heuristic to track these variables. The variable order heuristic (Section 2.8) has another heap with rational variables whose bounds allow only one value. The variables in the additional heap are ordered by the VSIDS score. All unassigned variables from the new heap are decided before any other variable.

Figure 4.1 compares computation time with rational variable propagation (the y-axis) and without it (the x-axis). We achieved a speed-up above 40% on satisfiable problems that did not time out. However, the heuristic did not work as well on unsatisfiable inputs. Some new problems (mostly unsatisfiable) could not be solved with this heuristic within the time limit of 20 minutes.

4.2 Derivation of new bounds

We tried deriving new bounds using Fourier-Motzkin elimination from the bounds on the trail (Section 2.7). Stronger variable bounds help the LRA plugin with

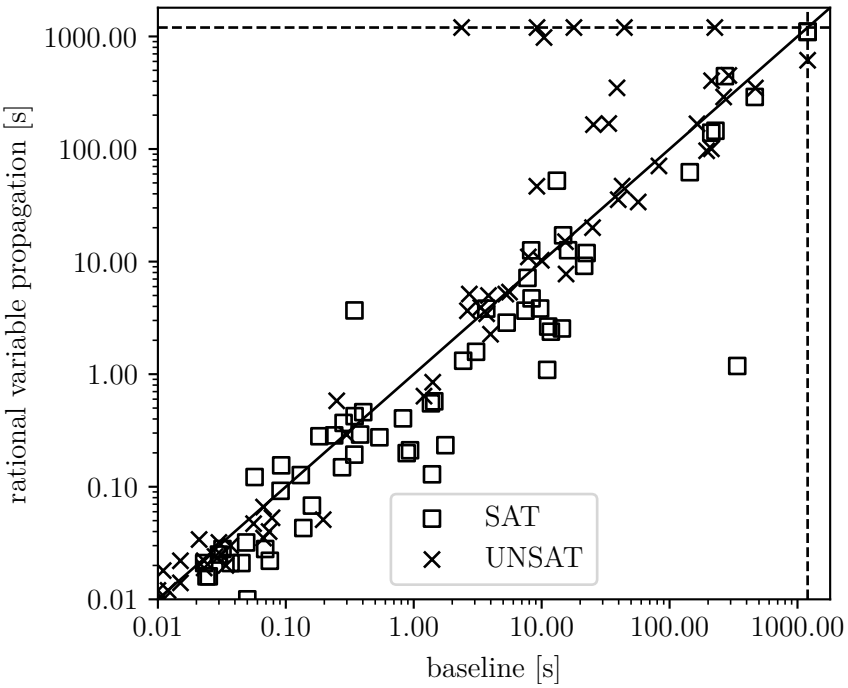

Figure 4.1: Evaluation of rational variable propagation.

deciding a proper value for a rational variable, and some conflicts may be discovered earlier.

Figure 4.2 shows a comparison of computation time in seconds with bound derivation (the y axis) and without it (the x axis). There are some problems where bound derivation improved the computation time. The average number of conflicts on problems that did not timeout was lower with bound derivation. However, the overhead of computing new bounds outweighed any benefit from a slightly lower number of conflicts, and the solver was, on average, slower with bound derivation.

4.3 Comparison

Figure 4.3 shows a comparison of the computation time of Yaga and OpenSMT2 [2] (a DPLL(T)-based solver) on the whole QF_LRA SMT-LIB benchmark [4]. Yaga solved 1429 problems within the 20 minute time limit. OpenSMT solved 1709 within the same time limit.

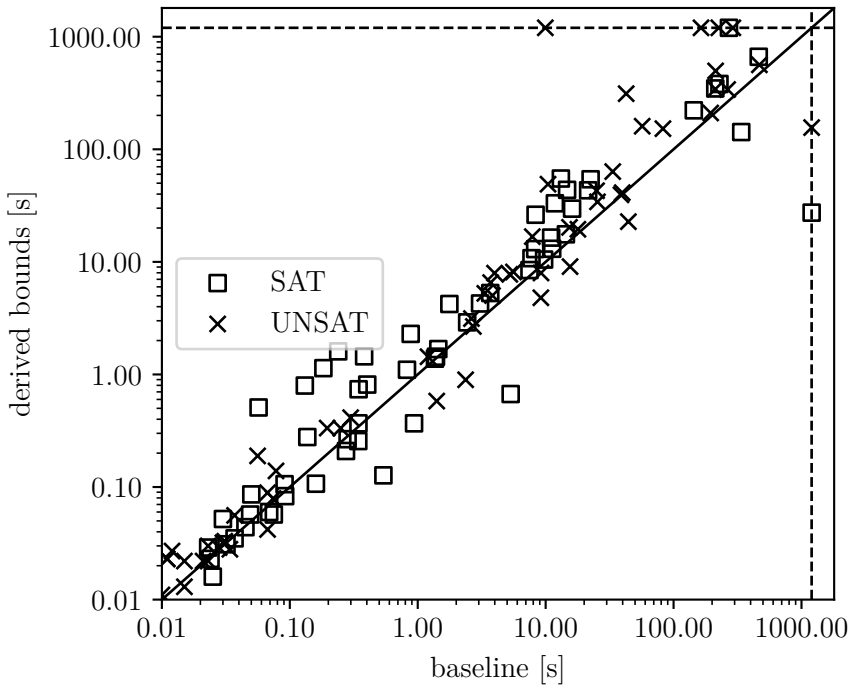

Figure 4.2: Derivation of new bounds.

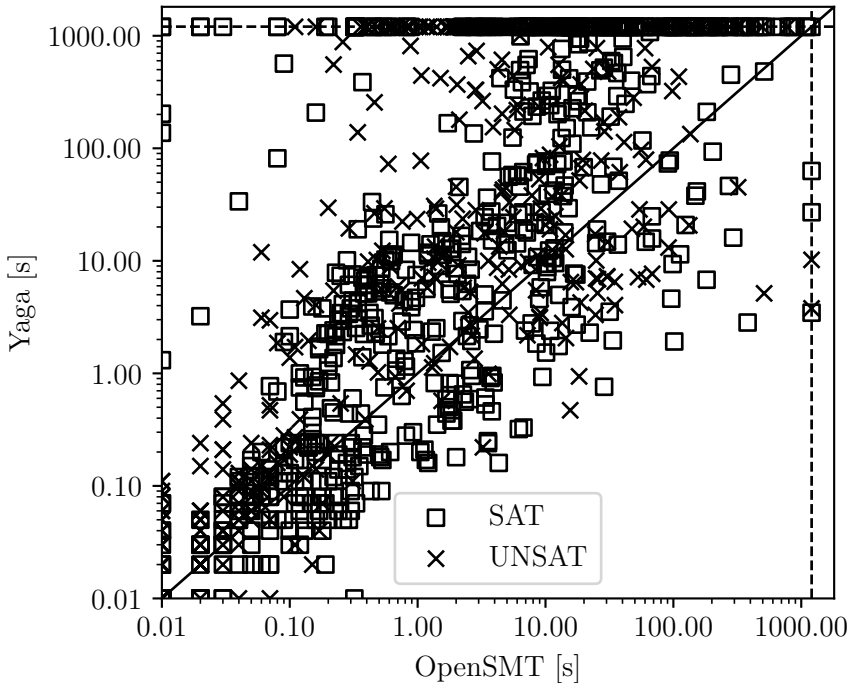

Figure 4.3: Comparison of Yaga and OpenSMT on the whole SMT-LIB QF_LRA benchmark.

5 Conclusion

We implemented an SMT solver Yaga based on the MCSat framework [7, 10] with two plugins for Boolean variables and the theory of quantifier-free Linear real arithmetic (LRA). The solver has an extensible interface that can be used for further development of theory plugins and heuristics for the MCSat framework. We evaluated Yaga on the quantifier-free LRA category of the SMT-LIB benchmark [4], and we compared it with a DPLL(T)-based SMT solver OpenSMT [2].

Bibliography

[1] GMP: The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`. Accessed: 2023-06-03.

[2] OpenSMT2. `https://github.com/usi-verification-and-security/opensmt`. Accessed: 2023-06-11.

[3] SMT-COMP 2023. `https://smt-comp.github.io/2023/`. Accessed: 2023-06-12.

[4] SMT-LIB: the satisfiability modulo theories library. `http://smtlib.cs.uiowa.edu/`. Accessed: 2023-06-09.

[5] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[6] Armin Biere. Weaknesses of CDCL solvers. In *Fields Institute Workshop on Theoretical Foundations of SAT Solving*, 2016.

[7] Leonardo De Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation: 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings 14*, pages 1–12. Springer, 2013.

[8] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. *SAT*, 3569:61–75, 2005.

[9] Jinbo Huang et al. The Effect of Restarts on the Efficiency of Clause Learning. In *IJCAI*, volume 7, pages 2318–2323, 2007.

[10] Dejan Jovanovic, Clark Barrett, and Leonardo De Moura. The design and implementation of the model constructing satisfiability calculus. In *2013 Formal Methods in Computer-Aided Design*, pages 173–180. IEEE, 2013.

[11] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[12] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[13] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing–SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings 10*, pages 294–299. Springer, 2007.

[14] Joao P Marques Silva and Karem A Sakallah. GRASP-a new search algorithm for satisfiability. In *ICCAD*, volume 96, pages 220–227, 1996.

[15] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 53(2005):1–2, 2005.

[16] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*, pages 279–285. IEEE, 2001.