DCC831 Theory and Practice of SMT Solving
Python CDCL($\mathcal{T}_{LRA}$) SMT Solver

Augusto A. Mafra[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brazil

augustomafra@dcc.ufmg.br

1. Introduction

This project implements a Python CDCL($\mathcal{T}_{LRA}$) SMT solver. It integrates three main components as building blocks:

1. pySMT: used as the parser for SMT-LIB2.6 problems.
2. PySAT: used as the SAT solver for propositional formulas.
3. cvc5: used as the Quantifier-Free Linear Real Arithmetic (QF_LRA) Theory solver.

   All components are integrated using their Python APIs.

1.1. Goals

1. Provide a complete and sound Python CDCL($\mathcal{T}_{LRA}$) solver for QF_LRA problems.
2. Validate the solver on QF_LRA SMT-LIB benchmarks.
3. Compare the Python solver performance against cvc5 SMT solver.
4. Measure the performance impacts of varying the SAT solver selected via PySAT.

2. Implementation

2.1. Command-line Interface

The solver is implemented in the `cdcl_tlra_solver.py` file, providing the following command-line interface:

```
python3 cdcl_tlra_solver [-h] [--sat-solver SAT_SOLVER]
                         [--dump-models]
                         [--verbose VERBOSE]
                         smt_lib2_filename

  -h, --help            show this help message and exit
  --sat-solver SAT_SOLVER, -s SAT_SOLVER
                SAT solver used for solving propositional
                abstraction (Default: minisat22).
  --dump-models, -m
                Print models after every SAT response
  --verbose VERBOSE, -v VERBOSE
                Print verbose debugging log
```

The `--sat-solver` and `-s` options expect a SAT solver name as available for PySAT solvers. If not specified, the solvers uses `minisat22` as the default. The `--verbose` and `-v` options expect an integer for the verbosity level of the solver messages.

2.2. CDCL($\mathcal{T}_{LRA}$) Loop

The core CDCL($\mathcal{T}_{LRA}$) Loop is implemented in the `cdcl_tlra_check_sat` function. It performs three high-level steps:

1. Parse the input SMT-LIB 2.6 file using the pySMT parser API.
   Besides extracting the SMT formula, the parsing step also collects the expected result, which is commonly available in SMT-LIB benchmarks in the `(set-info :status sat/unsat/unknown)` command.
2. Clausify the SMT formula into an abstraction for propositional logic solvers.
   The clausifier is implemented in the `BooleanAbstraction` class, which encapsulates the mapping between SMT expressions and their corresponding propositional encodings.
3. Loop between calls to the SAT solver and the SMT solver.
   On each iteration, the current set of clauses is evaluated by the SAT solver. If the SAT solver returns `unsat`, then the loop halts returning `unsat` for the original problem.
   Otherwise, the asserted literals from the SAT model are converted back to SMT expressions and checked by the SMT solver.
   The SMT solver may either confirm the propositional model and return `sat`, or refute it in the $\mathcal{T}_{LRA}$ theory.
   In the latter case, the unsat core produced by SMT solver is mapped to a propositional conflict clause, which is fed to the SAT solver so the process repeats.

   Further details for the implementation steps are described below.

2.2.1. Clausifier

The propositional encoding for each SMT expression is represented by a non-zero integer variable in the SAT solver. Negated literals are represented by negative integers, and clauses are represented by lists of those literals. In order to produce a CNF representation for the SAT solver, the method `BooleanAbstraction.clausify` recursively applies Tseitin's encoding to the SMT subexpressions.

The encoding to CNF literals follows from the known Tseitin representations for the tokens: `true`, `false`, `not`, `or`, `and`, `=>` and = (used for representing $\iff$ in Boolean formulas). Encoding the `ite` expressions is done by translating them to a set of equivalent SMT formulas using only conjunctions, negations and $\iff$, so that they can be recursively encoded by clausifying the equivalent formulas. In particular, this requires also encoding $Repr$ as a fresh SMT variable, which results in a sound representation as long as the following equivalence holds:

$$\begin{aligned}
\text{(ite cond then else)} \iff & ((Repr \iff \text{(ite cond then else)}) && (1)\\
& \wedge (Repr \wedge \text{cond} \iff \text{then}) && (2)\\
& \wedge (Repr \wedge \neg\text{cond} \iff \text{else})) && (3)
\end{aligned}$$

The equivalence can be proven by encoding it as the SMT-LIB problem below, which is solved to `unsat` using cvc5:

```
     (set-logic QF_UF)

(declare-const cond Bool)
(declare-const then Bool)
(declare-const else Bool)
(declare-const ite_result Bool)

; Encode the ite expression by introducing a fresh
; variable ite_result and asserting equivalence
(assert (= ite_result (ite cond then else)))

(declare-const Repr Bool)

; Equivalent representation using only <=>, not, and
; The assertions below are created by the clausifier
; when abstracting an ite expression
(assert (= (and Repr cond) then))
(assert (= (and Repr (not cond)) else))

; Prove that the alternative representation without
; ite is equivalent to the original one
(assert (distinct ite_result Repr))

; Results in unsat, completing the proof of equivalence
(check-sat)
```

A relevant observation for some big SMT-LIB benchmarks is that the `clausify` method recurses into the SMT subformulas, which fails due to max stack depth limit being reached when processing some formulas. However, the set of tests used in the experimental evaluation do not hit this issue.

2.2.2. SAT and SMT Solving

The clausifier needs to be run only once in the solver, because further iterations between the SAT and SMT backends will only reuse the literals already abstracted. Furthermore, in order to take advantage of the SMT solver state from previous iterations, the assertions are incrementally added using the `push` and `pop` SMT-LIB scopes. Thus the solver is configured with the `:incremental` option.

The CDCL($\mathcal{T}_{LRA}$) loop also requires that cvc5 produces unsat cores for each iteration. However, unsat cores are not currently supported on the wrapped cvc5 API on pySMT. To overcome this limitation, the underlying implementation of the cvc5 solver Python object is directly accessed to expose the unsat cores. This introduces the need for some extra book-keping, because the formula representation of SMT terms in the cvc5 API is not the same as the pySMT expressions. Whenever pySMT expressions are asserted by the SAT model, they are converted to the cvc5 API terms before being passed

to the solver and a dictionary stores the mapping between the two representations.

3. Experimental Evaluation

The solver is evaluated in selected benchmarks for the SMT-LIB QF_LRA theory. The result of the solver is collected for each test to assess soundness of the implementation. The benchmark runs are performed with a timeout of 32 minutes per test because runtimes of about 30 minutes have been observed when running the solver in the experimental setup. The benchmarks ran on a 11th Gen i7-1165G7 CPU with 2.80GHz, 8 GB RAM and on Ubuntu 22.04.5 LTS operating system.

Runtime is also measured for each benchmark run, and the CDCL($\mathcal{T}_{LRA}$) runtime is compared on setups using three different SAT solvers as propositional engines: MiniSat 2.2, Glucose 4.2 and CaDiCaL 1.9.5. Finally, the runtime of solving the SMT benchmarks using cvc5 is collected for comparison with the Python solver.

3.1. Functional Evaluation

The Python solver is able to solve 35 of the 50 selected benchmarks regardless of the SAT solver used, even though the backend propositional solver has impacted the runtimes observed. No soundness or other errors were observed in the results. The same 15 benchmarks failed with timeout for the three SAT configurations, and all were solved under 1s by cvc5.

Table 1: Benchmark Results on Python CDCL($\mathcal{T}_{LRA}$) solver and cvc5

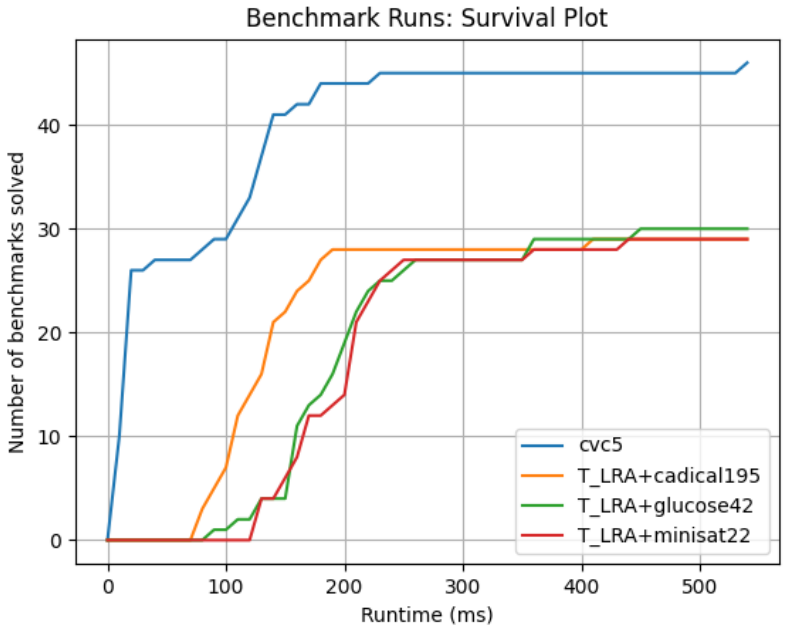
Result	$\mathcal{T}_{LRA}$+CaDiCaL 1.9.5	$\mathcal{T}_{LRA}$+Glucose 4.2	$\mathcal{T}_{LRA}$+MiniSat 2.2	cvc5
sat	16	16	16	25
unsat	19	19	19	25
timeout	15	15	15	0

3.2. Runtime Evaluation

The survival plot shows the number of benchmarks solved by each solver configuration up to 530ms, which was the maximum time needed for cvc5 to solve a single problem. Clearly, the Python CDCL($\mathcal{T}_{LRA}$) solvers do not achieve cvc5's performance, and they require longer timeouts in order to complete the solution for the 35 problems seen in Table 1.

Furthermore, the analysis indicates a significant performance impact of changing the underlying SAT solver. Despite having all runs with a timeout of 32 minutes, only the solver with MiniSat 2.2 required that large margin, completing its most difficult benchmark after 30 minutes. The solver using CaDiCal 1.9.5 solved the same benchmark in 12 minutes, and the one with Glucose 4.2 needed 16 minutes. This indicates that on the larger benchmarks, changing the SAT solver can result in a speed up of 2.5x.

On the tests that the CDCL($\mathcal{T}_{LRA}$) completed before 530ms, the survival plot indicates that setting the SAT solver also has significant effect. In the 100ms mark, the solver with CaDiCal 1.9.5 had completed 7 benchmarks, whereas Glucose 4.2 solved only one and Minisat 2.2 solved none. CaDiCal 1.9.5 keeps winning on the smaller benchmarks up to the 400ms mark, when Glucose 4.2 is able to solve one test beyond the other solvers.



4. Conclusion

The implemented Python CDCL($\mathcal{T}_{LRA}$) SMT solver for the QF_LRA theory produced sound results on the set of SMT-LIB tests evaluated. It successfully implements Tseitin encoding for clausifying QF_LRA SMT formulas, which allows to solve big benchmarks without hitting exponential increase on the number of clauses. Further enhancements on the clausifier would be to remove recursion when traversing the SMT-LIB subformulas, as it failed with stack-overflow on 31 tests of the QF_LRA non-incremental SMT-LIB 2024 benchmarks.

The Python solver does not match cvc5's performance on the benchmarks, but it provided a flexible interface for experimenting with different SAT solver configurations. In particular, the evaluation showed that, on the selected QF_LRA benchmarks, CaDiCal 1.9.5 was the best SAT solver overall for the biggest problems, reaching 2.5x time speedup against the implementation with Minisat 2.2, whereas Glucose 4.9 performed marginally better on the benchmarks solved around 400ms.