A SAT-Based Decision Procedure for the Boolean Combination of Difference Constraints

Marco Maratea

j.w.w. A. Armando, C. Castellini, E. Giunchiglia

Mechanised Reasoning Group

Dipartimento di Informatica, Sistemistica e Telematica - Università di Genova

Motivation

Decision procedures able to decide quantifier-free first-order theories are becoming increasingly important in Artificial Intelligence and Formal Verification areas.

Several properties of hardware, timed automata, and software can be modeled in quantifier-free first-order theories as well as planning and scheduling problems.

Separation Logic (SL) is one of such decidable quantifier-free first-order theories that allows boolean combination of difference constraints.

Due to the boost that SAT solvers had in the last years, the SAT-Based approach to decide these formulas has become a suitable and efficient way.

Why Separation Logic?

SL seems to be a good compromise between efficiency and expressivity.

It combines propositional atoms with a restricted form of linear arithmetic via the standard boolean connectives.

Many available benchmarks, both random, hand-made and real-life (following the usual SAT Competition division in categories) are in SL and a lot of properties of systems and planning/scheduling constraints can be encoded in this logic.

SL: Definitions (1)

Fix a domain of interpretation D for the arithmetic variables (the set of real or the set of integer numbers).

An SL-atom is either a propositional variable or an SL-expression x - y $\leq$ c ($<, >, \geq, =, \neq$ can be (easily) recast in $\leq$), where x and y range on D and c is a numeric constant.

An SL-expression is also called difference constraint.

An SL-literal is an SL-atom or its negation.
An SL-clause is a finite disjunction of SL-literals.
An SL-formula is a finite conjunction of SL-clauses.

SL: Definitions (2)

We are restricting our attention on SL-formula in Conjunctive Normal Form (CNF): This is not a big problem as long as there are efficient algorithms for transforming a non-CNF in a CNF formula.

Deciding an SL-formula (Is there an SL-assignment to propositional atoms and arithmetic variables, such that the SL-formula ϕ is true?) is an NP-complete problem.

TSAT++: Yet another SAT-Based solver for SL?

Even TSAT++ follows the well-known (SAT-Based) lazy approach for deciding SL-formulas, it introduces new ideas/optimization techniques like:

1. SAT solver partial assignments check (early pruning)

2. detection of the "best" witness of inconsistency

3. propositional assignment reduction (via prime implicants)

Using these techniques and combining them, TSAT++ can reach state-of-the-art results in a very wide range of domains of benchmarks arising from the AI and Formal Verification communities.

Agenda

- TSAT++'s overview
- TSAT++'s optimization techniques
- Experimental analysis
- Conclusions and future work

TSAT++'s architecture

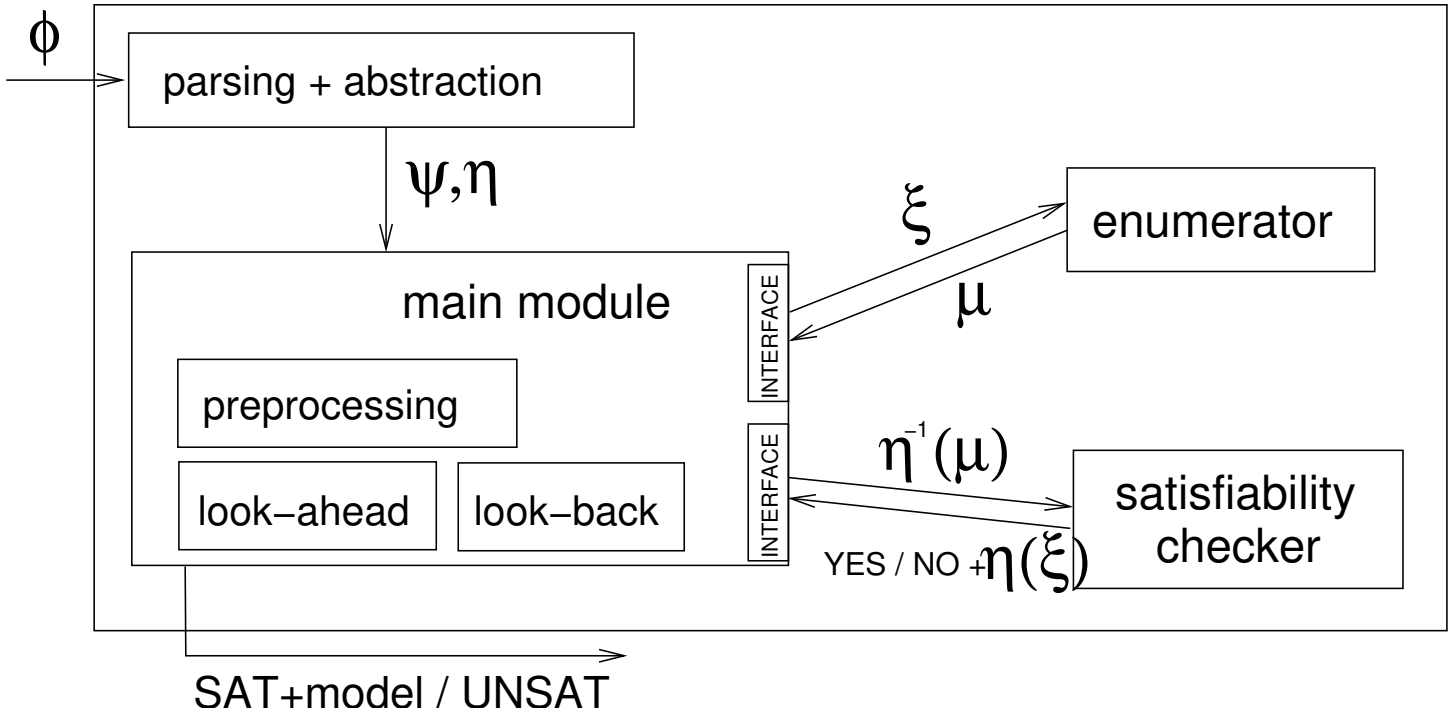

Figure 1: High-level view of TSAT++.

TSAT++'s approach

TSAT++ is implemented under the ACR (abstract-check-refine) paradigm:

1. first the SL-formula ϕ is abstracted to a boolean formula ψ;

2. the boolean models μs are checked for arithmetic consistency;

3. the boolean formula is refined using ξ in case of arithmetic inconsistency (and back to step. 2).

TSAT++ employs SIMO for the propositional part, and a modified version of the Bellman-Ford (BF) algorithm for the arithmetic part.

TSAT++'s basic algorithm

Given an SL-formula ϕ,

function TSAT++-Solve(ϕ)
  ψ= Abstract(ϕ)
  η= Map(ϕ)
  SIMO.LoadFormula(ψ)
  while ((μ = SIMO.Solve(ψ)) != NULL)
    if ((ξ = TSAT++.ConsistencyCheck(μ, η)) == NULL)
      return SAT
    SIMO.BacktrackWithReason(ξ)
  return UNSAT

Optimization 1: preprocessing

One drawback of the generate-and-test approach is that (exponentially) many trivially inconsistent valuations can be generated and then discarded (e.g., with $x - y \leq 3$ assigned to true and $x - y \leq 5$ to false)

To reduce this drawback, in TSAT++ for each pair c_1, c_2 of difference constraints in the same variables and occurring in ϕ, the consistency of all possible pairs of literals built out of them, i.e., $\{c_1, c_2\}$, $\{\neg c_1, c_2\}$, $\{c_1, \neg c_2\}$, and $\{\neg c_1, \neg c_2\}$, is checked.

Assuming, e.g., $\{c_1, c_2\}$ is inconsistent, the clause $\{\neg c_1, \neg c_2\}$ is added to ϕ before the search starts.

This dramatically speeds-up the search, especially on randomly generated problems.

TSAT++'s algorithm with opt 1.

Given an SL-formula ϕ,

function $\mathrm{TSAT++}$-Solve(ϕ)
  ψ= **Abstract**(ϕ)
  η= Map(ϕ)
  SIMO.LoadFormula(ψ)
  while ((μ = SIMO.Solve(ψ)) != NULL)
    if ((ξ = $\mathrm{TSAT++}$.ConsistencyCheck(μ, η)) == NULL)
      return SAT
    SIMO.BacktrackWithReason(ξ)
  return UNSAT

Optimization 2: early pruning

In CSP, this technique has been shown to be very effective on randomly generated problems.

The SAT solver must have the feature of returning also partial (but consistent) boolean model μ_p (even they do not satisfy yet ψ), other than total boolean model μ satisfying ψ.

If μ_p leads to an arithmetic inconsistence, there is no need to go on this branch, and the procedure can backtrack.

If μ_p does not lead to an arithmetic inconsistence, we have to go on this branch; if μ_p was total, we are done.

TSAT++'s algorithm with opt 2.

Given an SL-formula ϕ,

function $\mathrm{TSAT++}$-Solve(ϕ)
  ψ= Abstract(ϕ)
  η= Map(ϕ)
  SIMO.LoadFormula(ψ)
  while ((μ_p = SIMO.**Solve**(ψ)) != NULL)
    if ((ξ = $\mathrm{TSAT++}$.ConsistencyCheck(μ_p, η)) == NULL)
      if (**IsSatisfying**(μ_p))
        return SAT
    else
      SIMO.BacktrackWithReason(ξ)
  return UNSAT

Optimization 3: detecting reason (1)

Slightly modifying the BF algorithm, after the detection of a negative cycle, we are able to extract the "best" reason ξ under given condition looking among the available cycles.

The following three options are currently implemented in TSAT++:

- plain: pick the first reason non-deterministically
- shortest: pick the minimal reason under cardinality
- shallowest: pick the minimal reason under the order induced by the propositional stack

Optimization 3: detecting reason (2)

In the analysis, we have used the "shortest" option because it seems to lead to slight better results.

Nevertheless, the differences between reasons are not considerable, because there are very few available negative cycles for each failure.

This is due to the single-source nature of the BF.

TSAT++'s algorithm with opt 3.

Given an SL-formula ϕ,

function $\mathrm{TSAT++}$-Solve(ϕ)
  ψ= Abstract(ϕ)
  η= Map(ϕ)
  SIMO.LoadFormula(ψ)
  while ((μ = SIMO.Solve(ψ)) != NULL)
    if ((ξ = $\mathrm{TSAT++}$.**ConsistencyCheck**(μ, η)) == NULL)
      return SAT
    SIMO.BacktrackWithReason(ξ)
  return UNSAT

Optimization 4: assignment reduction

Given μ propositional satisfying ϕ, it may be the case that some of the literals in μ may be not necessary to satisfy ϕ. This is in particular true when SAT solvers use lazy data structure like watched literals.

We compute a *prime implicant* μ_r of the formula ϕ such that $\mu_r \subseteq \mu$.

We call the above procedure *reduction*, and it may be useful because

– if μ leads to a satisfying SL-assignment ,so is μ_r, and we are done;
– if μ does not lead to a satisfying SL-assignment, it may nevertheless be the case that μ_r does, and we can still interrupt the search because we are done;
– if μ and μ_r do not lead to a satisfying SL-assignment, checking the consistency of μ instead of μ_r can cause exponentially many more consistency checks.

TSAT++'s algorithm with opt 4.

Given an SL-formula ϕ,

function TSAT++-Solve(ϕ)
  ψ= Abstract(ϕ)
  η= Map(ϕ)
  SIMO.LoadFormula(ψ)
  while ((μ = SIMO.Solve(ψ)) != NULL)
    μ_r = TSAT++.**reduction**(μ)
    if ((ξ = TSAT++.ConsistencyCheck(μ_r, η)) == NULL)
      return SAT
    SIMO.BacktrackWithReason(ξ)
  return UNSAT

Experimental settings

For all the solvers:
TIME : 1000 sec
MEM : 512MB
"–" : segmentation fault

In TSAT++:
– j : preprocessing
– p : prime implicant generation (assignment reduction)
– 2 : shortest reason

SEP-m : SEP with internal conjunction matrix off (suggested by O. Strichman)

Disjunctive Temporal Problem (DTPs)

These are well-known random problems from the AI community.

DTPs are randomly generated by fixing the number k of expressions x - y $\leq$ c per SL-clause, the number n of arithmetic variables, a positive integer L such that all the constants are taken in $[-L, L]$. Then:

1. the number of clauses m is increased in order to range from satisfiable to unsatisfiable instances from 2*n to 14*n step n,

2. for each tuple of values of the parameters, 100 instances are generated and then given to the solvers, and

3. the median of the CPU time is plotted against the m/n ratio.

TSAT++'s performances (1): DTPs

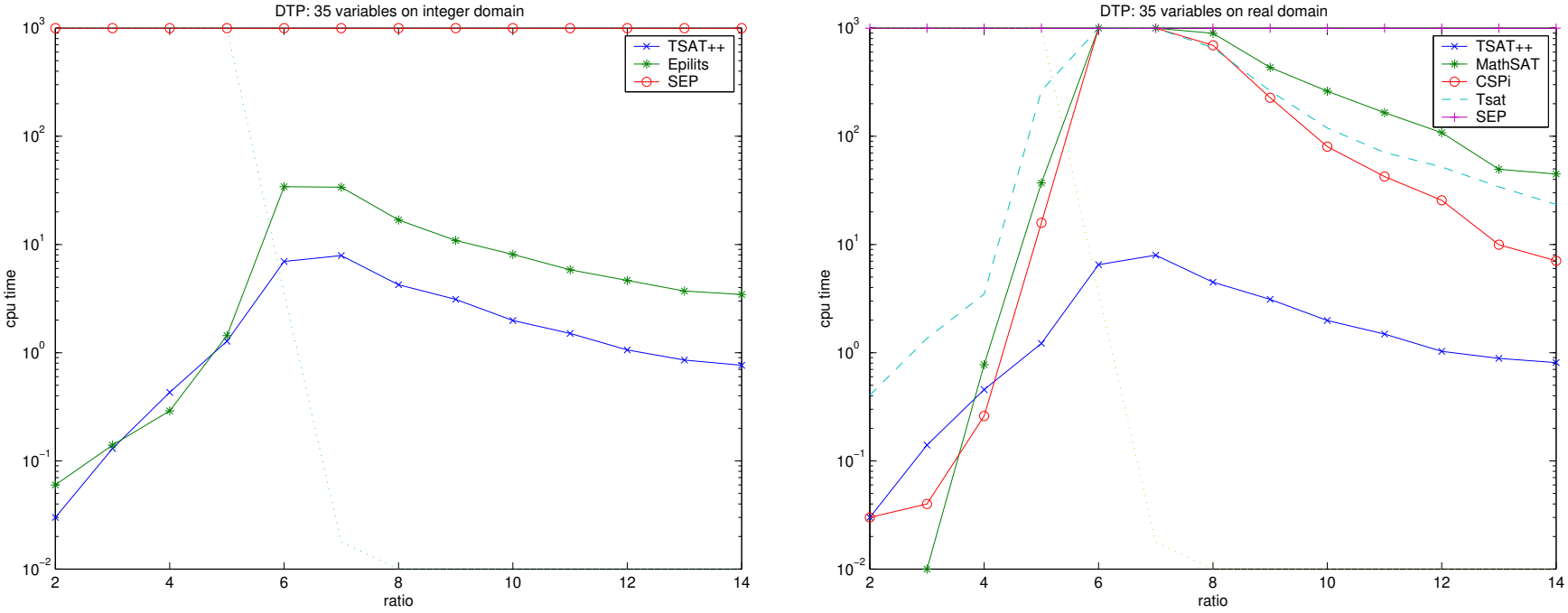

Figure 2: Evaluation on the DTP on 35 variables. Integer domain (left) and real domain (right). Setting: $k = 2$, $L = 100$.

TSAT++'s performance (2): postoffice problems

Instance	SAT?	TSAT++ jp2	MathSAT	SEP
P04-6-P04	NO	0.07	0.36	16.02
P04-7-P04	NO	0.11	0.36	134.21
P04-11-P04	NO	1.01	2.13	TIME
P04-12-P04	YES	0.58	0.91	TIME
P05-10-P05	NO	2.41	5.32	–
P05-11-P05	NO	3.44	9.23	–
P05-12-P05	NO	4.79	22.06	–
P05-13-P05	NO	8.88	54.17	–
P05-14-P05	YES	2.99	11.36	–

Diamonds problems

Given a parameter D (number of diamonds), these problems are characterized by an exponentially large (2^D) number of boolean models μ, some of which correspond to satisfying SL-assignments; hard instances with a unique satisfying SL-assignment can be generated.

A second parameter, S (related to the number of edge in each diamond), is used to make μ larger, further increasing the difficulty.

Variables range over the reals.

TSAT++'s performance (3): diamonds problems

Instance			Lazy				Eager	
D	S	u?	TSAT++ p2	M.SAT	ICS	CVC	SEP	SEP-m
250	5	NO	0.08	5.40	0.05	MEM	52.20	0.95
250	5	YES	0.21	TIME	150.02	3.26	0.77	288.30
500	5	NO	0.29	21.22	0.11	MEM	742.99	5.92
500	5	YES	1.05	TIME	MEM	6.99	4.85	TIME
1000	5	NO	1.07	–	0.28	MEM	TIME	27.52
1000	5	YES	6.45	–	MEM	15.68	22.53	TIME
2000	5	NO	3.76	–	0.82	MEM	–	–
2000	5	YES	29.90	–	MEM	37.53	–	–

Conclusions

TSAT++ is a SAT-Based solver for the boolean combination of difference constraints that follows the generate-and-test approach.

It introduces various ideas/optimization techniques; most of them are theory-independent.

Using and combining these techniques, TSAT++ is faster (sometimes significantly) than other state-of-the-art solvers in different domains arising from AI and Formal Verification communities.

Future work

From the point of view of the basic research:

- Extending TSAT++'s theory with (full) linear arithmetic, uninterpreted functions, arrays

and, on the "applications" side, using TSAT++ as an effective back-end solver for:

- Software Model Checking
- Planning/Scheduling