UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories

Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani

March 7, 2007

Technical Report # DIT-07-006

A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories

Alessandro Cimatti[1], Alberto Griggio[2], and Roberto Sebastiani[2]

[1] ITC-IRST, Povo, Trento, Italy. cimatti@itc.it
[2] DIT, Università di Trento, Italy. {griggio,rseba}@dit.unitn.it

Abstract. Finding small unsatisfiable cores for SAT problems has recently received a lot of interest, mostly for its applications in formal verification. However, propositional logic is often not expressive enough for representing many interesting verification problems, which can be more naturally addressed in the framework of Satisfiability Modulo Theories, SMT. Surprisingly, the problem of finding unsatisfiable cores in SMT has received very little attention in the literature; in particular, we are not aware of any work aiming at producing small unsatisfiable cores in SMT.

In this paper we present a novel approach to this problem. The main idea is to combine an SMT solver with an external propositional core extractor: the SMT solver produces the theory lemmas found during the search; the core extractor is then called on the boolean abstraction of the original SMT problem and of the theory lemmas. This results in an unsatisfiable core for the original SMT problem, once the remaining theory lemmas have been removed.

The approach is conceptually interesting, since the SMT solver is used to dynamically lift the suitable amount of theory information to the boolean level, and it also has several advantages in practice. In fact, it is extremely simple to implement and to update, and it can be interfaced with every propositional core extractor in a plug-and-play manner, so that to benefit for free of all unsat-core reduction techniques which have been or will be made available.

We have evaluated our approach by an extensive empirical test on SMT-LIB benchmarks, which confirms the validity and potential of this approach.

1 Motivations and goals

In the last decade we have witnessed an impressive advance in the efficiency of SAT techniques, which has brought large and previously intractable problems at the reach of state-of-the-art SAT solvers. As a consequence, SAT solvers are now a fundamental tool in most formal verification design flows for hardware systems, both for equivalence, property checking, and ATPG. In particular, and due to its importance in formal verification [10], the problem of finding small *unsatisfiable cores* in SAT —i.e., unsatisfiable subsets of unsatisfiable sets of clauses— has been addressed by many authors in the recent years [9, 11, 16, 12, 8, 4, 2, 7, 14].

The formalism of plain propositional logic, however, is often not suitable or expressive enough for representing many other real-world problems, including the verification of RTL designs, of real-time and hybrid control systems, and the analysis of proof obligations in software verification. Such problems are more naturally expressible as satisfiability problems in decidable first-order theories —Satisfiability Modulo Theories, SMT. Efficient SMT solvers have been developed in the last five years, called *lazy* SMT solvers, which combine DPLL with ad-hoc decision procedures for many theories of interest (see, e.g., [6, 1, 3, 13, 5]).

Surprisingly, the problem of finding unsatisfiable cores in SMT has received virtually no attention in the literature. Although some SMT tools do compute unsat cores, this is done either as a byproduct of the more general task of producing proofs, or by modifying the embedded DPLL solver so that to apply basic propositional techniques to produce an unsat core. In particular, we are not aware of any work aiming at producing *small* unsatisfiable cores in SMT.

In this paper we present a novel approach addressing this problem. The main idea is to combine an SMT solver with an external propositional core extractor. The SMT solver stores and returns the theory lemmas it had to prove in order to refute the input formula; the external core extractor is then called on the boolean abstraction of the original SMT problem and of the theory lemmas. The resulting boolean unsatisfiable core is cleaned from (the boolean abstraction of) all theory lemmas, and it is refined back into a subset of the original clauses. The result is an unsatisfiable core of the original SMT problem.

Although simple in principle, the approach is conceptually interesting: basically, the SMT solver is used to dynamically lift the suitable amount of theory information to the boolean level. Furthermore, the approach has several advantages in practice: first, it is extremely simple to implement and to update; second, it is effective in finding small cores; third, the core extraction is not prone to complex SMT reasoning; finally, it can be interfaced with every propositional core extractor in a plug-and-play manner, so that to benefit for free of all unsat-core reduction techniques which have been or will be made available.

We have evaluated our approach by an extensive empirical test on SMT-LIB benchmarks, in terms of both effectiveness (reduction in size of the cores) and efficiency (execution time). The results confirm the validity and potential of this approach.

The paper is organized as follows. In §2 we provide some background knowledge on techniques for SMT and for extraction of unsatisfiable cores in SAT and in SMT. In §3 we present and discuss our new approach and algorithm. In §4 we present and comment empirical tests. In §5 we conclude, suggesting some future developments.

2 Background

Given a decidable first-order theory $\mathcal{T}$, we call a *theory solver for* $\mathcal{T}$, $\mathcal{T}$*-solver*, any tool able to decide the satisfiability in $\mathcal{T}$ of sets/conjunctions of ground atomic formulas and their negations (*theory literals* or $\mathcal{T}$*-literals*) in the language of $\mathcal{T}$. If the input set of $\mathcal{T}$-literals μ is $\mathcal{T}$-unsatisfiable, then a typical $\mathcal{T}$*-solver* not only returns unsat, but it also returns the subset η of $\mathcal{T}$-literals in μ which was found $\mathcal{T}$-unsatisfiable. (η is hereafter called a *theory conflict set*, and $\neg\eta$ a *theory conflict clause*.) If μ is $\mathcal{T}$-satisfiable, then $\mathcal{T}$*-solver* not only returns sat, but it may also be able to discover one (or more) deductions in the form $\{l_1,...,l_n\} \models_{\mathcal{T}} l$, s.t. $\{l_1,...,l_n\} \subseteq \mu$ and l is an unassigned $\mathcal{T}$-literal. If so, we call $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ a *theory-deduction clause*. Importantly, notice that

```
SatValue $\mathcal{T}$-DPLL ($\mathcal{T}$-formula $\varphi$) {
    $\varphi^p = \mathcal{T}2\mathcal{P}(\varphi)$;
    while (DPLL($\varphi^p$,$\mu^p$) == sat) {
        $\langle\rho,\eta\rangle$ = $\mathcal{T}$-solver($\mathcal{P}2\mathcal{T}(\mu^p)$)
        if ($\rho$ == sat) then return sat;
        $\varphi^p = \varphi^p \wedge \mathcal{T}2\mathcal{P}(\neg\eta)$;
    };
    return unsat;
};
```

Fig. 1. A simplified schema for lazy *SMT*$(\mathcal{T})$ procedures.

both theory-conflict clauses and theory-deduction clauses are valid in $\mathcal{T}$. We call them *theory lemmas* or *$\mathcal{T}$-lemmas*.

Satisfiability Modulo (the) Theory $\mathcal{T}$ (*SMT*$(\mathcal{T})$) is the problem of deciding the satisfiability of *boolean combinations* of propositional atoms and theory atoms. We call an *SMT*$(\mathcal{T})$ *tool* any tool able to decide *SMT*$(\mathcal{T})$. Notice that, unlike a *$\mathcal{T}$-solver*, an *SMT*$(\mathcal{T})$ tool must handle also boolean connectives.

Hereafter we adopt the following terminology and notation. The bijective function $\mathcal{T}2\mathcal{P}$ ("theory-to-propositional"), called *boolean abstraction*, maps propositional variables into themselves, ground $\mathcal{T}$-atoms into fresh propositional variables, and is homomorphic w.r.t. boolean operators and set inclusion. The function $\mathcal{P}2\mathcal{T}$ ("propositional-to-theory"), called *refinement*, is the inverse of $\mathcal{T}2\mathcal{P}$. The symbols φ, ψ denote $\mathcal{T}$-formulas, and μ, η denote sets of $\mathcal{T}$-literals; φ^p, ψ^p denote propositional formulas, μ^p, η^p denote sets of propositional literals (i.e., truth assignments) and we often use them as synonyms for the boolean abstraction of φ, ψ, μ, and η respectively, and vice versa (e.g., φ^p denotes $\mathcal{T}2\mathcal{P}(\varphi)$, μ denotes $\mathcal{P}2\mathcal{T}(\mu^p)$). If $\mathcal{T}2\mathcal{P}(\varphi) \models \bot$, then we say that φ is *propositionally unsatisfiable*.

2.1 Lazy techniques for SMT

The idea underlying every lazy *SMT*$(\mathcal{T})$ procedure is that (a complete set of) the truth assignments for the propositional abstraction of φ are enumerated and checked for satisfiability in $\mathcal{T}$; the procedure either returns sat if one $\mathcal{T}$-satisfiable truth assignment is found, or returns unsat otherwise.

Figure 1 presents a simplified schema of a lazy *SMT*$(\mathcal{T})$ procedure, called the *off-line schema*. The propositional abstraction φ^p of the input formula φ is given as input to a DPLL solver, which either decides that φ^p is unsatisfiable, and hence φ is $\mathcal{T}$-unsatisfiable, or returns a satisfying assignment μ^p; in the latter case, $\mathcal{P}2\mathcal{T}(\mu^p)$ is given as input to *$\mathcal{T}$-solver*. If $\mathcal{P}2\mathcal{T}(\mu^p)$ is found $\mathcal{T}$-consistent, then φ is $\mathcal{T}$-consistent. If not, *$\mathcal{T}$-solver* returns the conflict set η which caused the $\mathcal{T}$-inconsistency of $\mathcal{P}2\mathcal{T}(\mu^p)$; the abstraction of the $\mathcal{T}$-lemma $\neg\eta$, $\mathcal{T}2\mathcal{P}(\neg\eta)$, is then added as a clause to φ^p. Then the DPLL solver is restarted from scratch on the resulting formula.

Practical implementations follow a more elaborated schema, called the *on-line schema*. As before, φ^p is given as input to a modified version of DPLL, and when a satisfying assignment μ^p is found, the refinement μ of μ^p is fed to the $\mathcal{T}$*-solver*; if μ is found $\mathcal{T}$-consistent, then φ is $\mathcal{T}$-consistent; otherwise, $\mathcal{T}$*-solver* returns the conflict set η which caused the $\mathcal{T}$-inconsistency of $\mathcal{P}2\mathcal{T}(\mu^p)$. Then the clause $\neg\eta^p$ is added in conjunction to φ^p, either temporarily or permanently ($\mathcal{T}$*-learning*), and the algorithm backtracks up to the highest point in the search where a literal can be unit-propagated on $\neg\eta^p$ ($\mathcal{T}$*-backjumping*).

An important variant [6] is that of building a "mixed boolean+theory conflict clause", starting from $\neg\eta^p$ and applying the backward-traversal of the implication graph, until one of the standard conditions (e.g., 1st UIP [15]) is achieved .

Other important optimizations are *early pruning* and *theory propagation*: the $\mathcal{T}$-solver is invoked also on (the refinement of) an intermediate assignment μ: if it is found $\mathcal{T}$-unsatisfiable, then the procedure can backtrack, since no extension of μ can be $\mathcal{T}$-satisfiable; if not, and if the $\mathcal{T}$*-solver* performs a deduction $\{l_1,...,l_n\} \models_{\mathcal{T}} l$ s.t. $\{l_1,...,l_n\} \subseteq \mu$, then $\mathcal{T}2\mathcal{P}(l)$ can be unit-propagated, and the boolean abstraction of the $\mathcal{T}$-lemma $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ can be learned.

The on-line lazy *SMT*$(\mathcal{T})$ schema is a coarse abstraction of the procedures underlying all the state-of-the-art lazy *SMT*$(\mathcal{T})$ tools like, e.g., ARIO, BARCELOGIC, CVCLITE, MATHSAT, YICES. The interested reader is pointed to, e.g., [6, 1, 3, 13, 5], for details and further references.

2.2 Techniques for unsatisfiable-core extraction in SAT

Given an unsatisfiable (propositional) CNF formula φ, we say that an unsatisfiable CNF formula ψ is an *unsatisfiable core* of φ iff $\varphi = \psi \wedge \psi'$ for some (possibly empty) CNF formula ψ'. Intuitively, ψ is a subset of the clauses in φ causing the unsatisfiability of φ. An unsatisfiable core ψ is *minimal* iff the formula obtained by removing any of the clauses of ψ is satisfiable. A *minimum* unsat core is a minimal unsat core with the smallest possible cardinality.

In the last few years, several algorithms for computing small, minimal or minimum unsatisfiable cores of propositional formulas have been proposed. In [16], they are computed as a byproduct of a DPLL-based proof-generation procedure. The computed unsat core is simply the collection of all the original clauses that the DPLL solver used to derive the empty clause by resolution. The returned core is not minimal in general, but it can be restricted by iterating the algorithm until a fixpoint, using as input of the *i*th iteration the core computed at the previous one. In [12], an algorithm to compute minimal unsat cores is presented. The technique is based on modifications of a standard DPLL engine, and works by adding some extra variables (selectors) to the original clauses, and then performing a branch-and-bound algorithm on the modified formula. The procedure presented in [8] extracts minimal cores using BDD manipulation techniques, removing one clause at a time until the remaining core is minimal. The algorithm of [7], instead, manipulates the resolution proof so that to shrink the size of the core, using also a fixpoint iteration as in [16] to further enhance the quality of the results. The construction of a minimal core in [4] also uses resolution proofs, and it works by iteratively removing from the proof one input clause at a time, until it is no longer possible to prove inconsistency. When a clause is removed, the resolution proof is modified to prevent future use of that clause.

As far as the the computation of *minimum* unsatisfiable cores is concerned, the algorithm of [9] searches all the unsat cores of the input problem; this is done by introducing selector variables for the original clauses, and by increasing the search space of the DPLL solver to include also such variables; then, (one of) the unsatisfiable subformulas with the smallest number of selectors assigned to true is returned. The approach described in [11] instead is based on a branch-and-bound algorithm that exploits the relation between maximal satisfiability and minimum unsatisfiability. The same relation is used also by the procedure in [14], which is instead based on a genetic algorithm.

2.3 Techniques for unsatisfiable-core extraction in SMT

To the best of our knowledge, there is no published work in the literature devoted to the computation of unsatisfiable cores in SMT. However, at least three SMT solvers support unsat core generation with techniques adapted from SAT. CVCLITE [1] and a recent extension of MATHSAT [3] can compute unsatisfiable cores as a byproduct of the generation of proofs, in a way similar to that in [16]. YICES [5] instead uses the following technique: a selector variable is introduced for each original clause, which is forced to false before starting the search. In this way, when a conflict at decision level zero is found, the conflict clause contains only selector variables, and the unsat core returned is the union of the clauses whose selectors appear in such conflict clause. [3]

We remark the fact that none of these solvers aims at producing minimal or minimum unsat cores, nor does anything to reduce their size.

3 A novel approach to building unsatisfiable cores in SMT

We present a novel approach in which the unsatisfiable core is computed *a posteriori* w.r.t. the execution of the SMT solver, and only if the formula has been found $\mathcal{T}$-unsatisfiable, by means of an external (and possibly optimized) propositional unsat-core extractor.

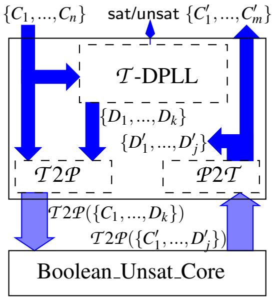
3.1 A novel approach

In the following we assume that a lazy $SMT(\mathcal{T})$ procedure has been run over a $\mathcal{T}$-unsatisfiable $SMT(\mathcal{T})$ CNF formula $\varphi =_{def} \{C_1, ..., C_n\}$, and that $D_1, ..., D_k$ denote all the $\mathcal{T}$-lemmas, both theory-conflict and theory-deduction clauses, which have been returned by the $\mathcal{T}$*-solver* during the run. In case of mixed boolean+theory-conflict clauses [6] (see § 2.1), the $\mathcal{T}$-lemmas are those which have been used to compute the mixed boolean+theory-conflict clause, including the initial theory-conflict clause and the theory-deduction clauses corresponding to the theory-propagation steps performed.

Our novel approach is based on two simple facts.

(i) Under the assumptions above, the conjunction of φ with all the $\mathcal{T}$-lemmas $D_1, ..., D_k$ is propositionally unsatisfiable: $\mathcal{T}2\mathcal{P}(\varphi \wedge \bigwedge_{i=1}^{n} D_i) \models \bot$.

[3] The description about the computation of unsat cores in CVCLITE and YICES reported here comes from private communications from the authors and from the user manual of CVCLITE.


$\{C_1,...,C_n\}$
sat/unsat
$\{C'_1,...,C'_m\}$
$\mathcal{T}$-DPLL
$\{D_1,...,D_k\}$
$\{D'_1,...,D'_j\}$
$\mathcal{T}2\mathcal{P}$
$\mathcal{P}2\mathcal{T}$
$\mathcal{T}2\mathcal{P}(\{C_1,...,D_k\})$
$\mathcal{T}2\mathcal{P}(\{C'_1,...,D'_j\})$
Boolean_Unsat_Core

```
$\langle$SatValue,Clause_set$\rangle$ $\mathcal{T}$-Unsat_Core(Clause_set $\varphi$) {
    // $\varphi$ is $\{C_1,...,C_n\}$
    if ($\mathcal{T}$-DPLL($\varphi$) == sat)
        then return $\langle$sat,$\emptyset\rangle$;
    // $D_1,...,D_k$ are the $\mathcal{T}$-lemmas stored by $\mathcal{T}$-DPLL
    $\psi^p$=Boolean_Unsat_Core($\mathcal{T}2\mathcal{P}(\{C_1,...,C_n,D_1,...,D_k\})$);
    // $\psi^p$ is $\mathcal{T}2\mathcal{P}(\{C'_1,...,C'_m,D'_1,...,D'_j\}))$;
    return $\langle$unsat,$\{C'_1,...,C'_m\}\rangle$;
}
```

Fig. 2. Schema of the $\mathcal{T}$-Unsat_Core procedure: architecture (left) and algorithm (right).

(ii) As $\mathcal{T}$-lemmas D_i are valid in $\mathcal{T}$, they do not affect the $\mathcal{T}$-satisfiability of a formula: $(\psi \wedge D_i) \models_{\mathcal{T}} \bot \Longleftrightarrow \psi \models_{\mathcal{T}} \bot$.

Fact (i) is the implicit termination condition of all lazy SMT tools when the input formula is $\mathcal{T}$-unsatisfiable. E.g., in the off-line schema of Figure 1, the procedure ends when DPLL establishes that $\mathcal{T}2\mathcal{P}(\varphi \wedge \bigwedge_{i=1}^{n} D_i)$ is unsatisfiable, each D_i being the negation of the theory-conflict set η_i returned by the i-th call to the $\mathcal{T}$*-solver*. Fact (i) generalizes to the on-line schema, noticing that $\mathcal{T}$-backjumping on a theory-conflict clause D_i produces an analogous effect as re-invoking DPLL on $\varphi^p \wedge \mathcal{T}2\mathcal{P}(D_i)$, whilst theory propagation on a deduction $\{l_1,...,l_k\} \models_{\mathcal{T}} l$ can be seen as a form on unit propagation on the theory-deduction clause $\mathcal{T}2\mathcal{P}$ $(\bigvee_i \neg l_i \vee l)$.

These facts suggest the novel algorithm represented in Figure 2. The procedure $\mathcal{T}$-Unsat_Core receives as input a set of clauses $\varphi =_{def} C_1,...,C_n$ and it invokes on it a lazy $SMT(\mathcal{T})$ tool $\mathcal{T}$-DPLL, which is instructed to *store* somewhere the $\mathcal{T}$-lemmas returned by $\mathcal{T}$*-solver*, namely $D_1,...,D_k$. If $\mathcal{T}$-DPLL returns sat, then the whole procedure returns sat. Otherwise, the boolean abstraction of $\{C_1,...,C_n,D_1,...,D_k\}$, which is inconsistent because of (i), is passed to an external tool Boolean_Unsat_Core, which is able to return the boolean unsat core ψ^p of the input. By construction, ψ^p is the boolean abstraction of a clause set $\{C'_1,...,C'_m,D'_1,...,D'_j\}$ s.t. $\{C'_1,...,C'_m\} \subseteq \{C_1,...,C_n\}$ and $\{D'_1,...,D'_j\} \subseteq \{D_1,...,D_k\}$. As ψ^p is unsatisfiable, then $\{C'_1,...,C'_m,D'_1,...,D'_j\}$ is $\mathcal{T}$-unsatisfiable. By (ii), the $\mathcal{T}$-valid clauses $D'_1,...,D'_j$ have no role in the $\mathcal{T}$-unsatisfiability of $\{C'_1,...,C'_m,D'_1,...,D'_j\}$, so that the procedure returns unsat and the $\mathcal{T}$-unsatisfiable core $\{C'_1,...,C'_m\}$.

Notice that the resulting $\mathcal{T}$-unsatisfiable core is not guaranteed to be minimal, even if Boolean_Unsat_Core returns minimal boolean unsatisfiable cores. In fact, it might be the case that $\{C'_1,...,C'_m\} \setminus \{C'_i\}$ is $\mathcal{T}$-unsatisfiable for some C'_i even though $\mathcal{T}2\mathcal{P}(\{C'_1,...,C'_m\} \setminus \{C'_i\})$ is satisfiable, because all truth assignments μ^p satisfying the latter are such that $\mathcal{P}2\mathcal{T}(\mu^p)$ is $\mathcal{T}$-unsatisfiable.

The procedure can be implemented very simply by modifying the SMT solver so that to store the $\mathcal{T}$-lemmas [4] —if it doesn't already— and by interfacing it with some

[4] Notice that here "storing" does not mean "learning": the SMT solver is not required to add the $\mathcal{T}$-lemmas to the formula during the search. This imposes no constraint on the lazy strategy

state-of-the-art boolean unsat-core extractor used as an external black-box device (e.g., by a simple exchange of files in DIMACS format). Moreover, if the SMT solver can provide the set of all $\mathcal{T}$-lemmas as output, then the whole procedure may reduce to a control device interfacing with both the SMT solver and the boolean core extractor as black-box external devices.

3.2 Discussion

Though based on an extremely simple concept, we believe that the newly-proposed approach is appealing for several reasons.

First, it is extremely simple to implement. The building of unsat cores is demanded to an external device, which is fully decoupled from the internal DPLL-based enumerator. Therefore, there is no need to implement any internal unsat-core constructor nor to modify the embedded boolean device. Every possible external device can be interfaced in a plug-and-play manner by simply exchanging a couple of DIMACS files.

Second, it is trivial to update: once some novel, more efficient or more effective boolean unsat-core device is available, it can be used in a plug-and-play way. This does not require modifying the DPLL engine embedded in the SMT solver.

Third, from the perspective of effectiveness in reducing the size of unsat cores, every original clause which the boolean unsat-core device is able to drop is dropped also in the final formula. Therefore, this technique exploits for free all unsat-core reduction techniques which have been and will be conceived in the SAT community. Notably, this involves also boolean unsat-core techniques which could be difficult to adapt to SMT —like, e.g., those based on genetic algorithms [14].

From the perspective of execution time, we notice a few facts. On the one hand, when the input formula φ is $\mathcal{T}$-unsatisfiable, the technique requires the call of an external boolean unsat-core tool, involving an extra execution time. (Notably, these tools typically present a tradeoff between execution time and effectiveness in reducing the size of the cores. See §4.) This is compensated in part by the loss of the overhead due to the computation of proofs/unsat cores within SMT. [5] On the other hand, when φ is $\mathcal{T}$-satisfiable, neither the call of the external tool nor the internal building of proofs and unsat cores is activated. Therefore, we may expect a possible increase of execution time for $\mathcal{T}$-unsatisfiable formulas, depending on the efficiency/effectiveness ratio of the external tool, and a decrease for $\mathcal{T}$-satisfiable formulas.

One potential drawback of this approach is the fact that a *SMT*$(\mathcal{T})$ solver is required to store all the $\mathcal{T}$-lemmas returned by the $\mathcal{T}$*-solver*. However, we claim this is not a real problem. In fact, unlike with plain SAT, in lazy SMT the computational effort is typically dominated by the search in the theory $\mathcal{T}$, so that the number of clauses that can be stored with a reasonable amount of memory is typically much bigger than the number of calls to the $\mathcal{T}$*-solver* which can overall be accomplished within a reasonable amount of time. In our experience, even the hardest SMT formulas at the reach of current lazy

adopted (e.g., offline/online, permanent/temporary learning, usage of mixed boolean+theory conflict clauses, etc.).

[5] E.g., in MATHSAT the computation of proofs may require some overhead; moreover, some tricks and speedups are disabled in order to be able to produce proofs.

SMT solvers rarely need generating more than 10^5 $\mathcal{T}$-lemmas, which have very reasonable memory requirements to store. (E.g., notice that the default choice in MATHSAT is to learn all $\mathcal{T}$-lemmas permanently anyway, and we have never encountered memory overload problems due to this fact.)

Finally, one limitation of this approach is that the resulting $\mathcal{T}$-unsatisfiable core is not guaranteed to be minimal, even if Boolean_Unsat_Core returns minimal boolean unsatisfiable cores. However, to the best of our knowledge, not only the issue of the *minimality* of unsat cores in SMT has never been addressed or even discussed before, but also this is the first time that the problem of the *size* of unsat cores in SMT is addressed.

4 An empirical evaluation

The novel approach presented in §3 was implemented within the MATHSAT [3] system, and was experimentally evaluated. We have tried different external propositional unsat core extractors, including AMUSE [12], BOOLEFORCE [2], EUREKA [4], MUP [8], TRIMMER [7], ZCHAFF [16], [6] and interfaced them with MATHSAT by exchanging boolean formulas and relative cores in form of files in DIMACS format. We wanted to try also the procedure presented in [14], but we could not obtain the tool from the authors. We also had to stop the experiments with MUP because it used to run out of memory (1GB) even on small examples. We adopted BOOLEFORCE as our baseline choice because it turned out to be both the fastest and the least effective in reducing the size of the cores, so that to be the ideal starting point for evaluating the tradeoff between these two features. [7]

We also developed a simple script, here referred as "BOOLEFORCE-iter-X%", which calls BOOLEFORCE iteratively until the reduction performed in the last iteration is smaller than a given percentage $X \in [0,..,100]$, with a maximum of 10 iterations. [8] Notice that the parameter X allows for tuning the tradeoff between effectiveness and efficiency. (E.g., $X = 0$ is equivalent to run BOOLEFORCE until a fixpoint is reached, $X = 100$ is equivalent to call it only once.)

All the experiments have been performed on a subset of the SMT-LIB benchmarks. We collected a total of 885 problems, of which 561 are $\mathcal{T}$-unsatisfiable, taken from the QF_UF, QF_IDL, QF_RDL, QF_LIA and QF_LRA divisions, and selected by using the same scripts used in the last SMT competition. [9] We used a preprocessor to convert them into CNF, and in some cases to translate them from the SMT language to the native language of a particular SMT solver if needed. [10] All the tests were performed

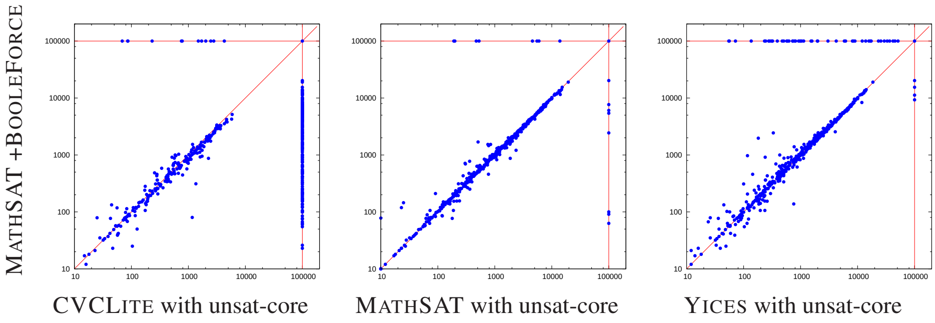
[6] For all these tools we have used the default configurations.

[7] This is not surprising because BOOLEFORCE was explicitly conceived for speeding up core generation with no claims of minimality, whilst all the other tools explicitly targeted size reduction or minimality.

[8] That is, it stops iterating either when $(|core_i| - |core_{i+1}|)/|core_i| < X/100$, $|core_i|$ being the size of the core after i iterated calls to BOOLEFORCE, or when $i > 10$.

[9] See http://www.csl.sri.com/users/demoura/smt-comp/.

[10] In particular, this was necessary for CVCLITE and YICES, since they can compute unsatisfiable cores only if the problems are given in their own native format.



	1^{st} quartile	median	mean	3^{rd} quartile
$\frac{\text{CVCLITE } w.u.c.}{\text{MATHSAT+BOOLEFORCE}}$	0.99	1.08	1.17	1.18
$\frac{\text{MATHSAT } w.u.c.}{\text{MATHSAT+BOOLEFORCE}}$	0.97	1.00	0.99	1.02
$\frac{\text{YICES } w.u.c.}{\text{MATHSAT+BOOLEFORCE}}$	0.94	1.00	1.00	1.04

Fig. 3. Size of the unsat cores computed by MATHSAT +BOOLEFORCE, y-axis, against those of CVCLITE, MATHSAT and YICES with proof-traces, with statistics on unsat core ratios. Points on the horizontal and vertical lines indicate timeouts/memouts.

on 3 GHz Intel Xeon machines with 4 GB of RAM running Linux. For each tested instance, the timeout was set to 600 seconds, and the memory limit to 1 GB. In order to make the experiments reproducible, the benchmarks and tools used are available at `http://dit.unitn.it/~griggio/papers/sat07_data.tar.bz2`. [11]

4.1 Effectiveness and efficiency of the baseline version

In this section we evaluate the baseline implementation of our novel approach in terms of effectiveness (size of cores) and efficiency (execution times).

Size of the unsat cores. First, we compare the baseline implementation of our novel approach, MATHSAT +BOOLEFORCE, against CVCLITE [1], YICES [5] and MATHSAT with unsat-core generation, with respect to the size of the unsat cores returned. Figure 3 shows the resulting scatter-plots, showing also some statistics about the ratios of the unsat core sizes computed by two different solvers. CVCLITE, MATHSAT, MATHSAT +BOOLEFORCE and YICES solved within the timeout respectively 233, 423, 420 and 472 out of the 561 $\mathcal{T}$-unsatisfiable problems; in every plot of Figure 3 only instances for which both solvers terminated successfully have been considered. Figure 4 shows the absolute reduction in size performed by MATHSAT +BOOLEFORCE.

[11] With the exception of EUREKA, which has been provided to us by the authors under a non-disclosure agreement.

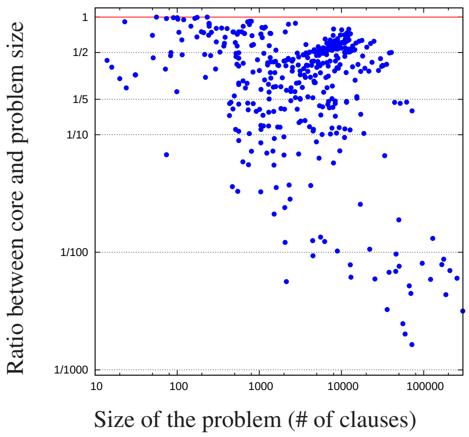
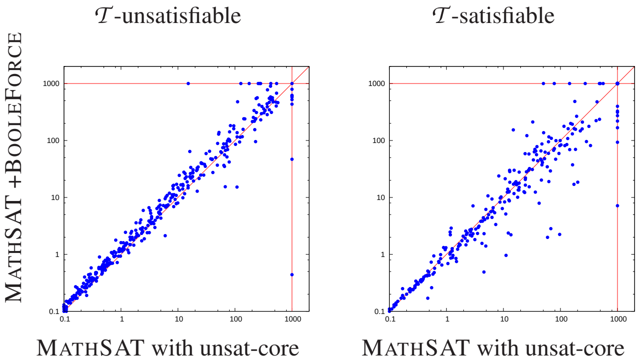


Fig. 4. Ratio between the size of the original formula and that of the unsat core computed by MathSAT +BooleForce.



MathSAT / MathSAT + BooleForce	1^{st} **quartile**	**median**	**mean**	3^{rd} **quartile**
$\mathcal{T}$-unsatisfiable	0.73	0.85	0.89	1.00
$\mathcal{T}$-satisfiable	0.91	1.01	1.46	1.23
overall	0.77	0.91	1.09	1.03

Fig. 5. Comparison of the run times between MathSAT +BooleForce and MathSAT with unsat-core, on $\mathcal{T}$-unsatisfiable (left) and $\mathcal{T}$-satisfiable (right) formulas, with respective statistics. Points on the horizontal and vertical lines indicate timeouts/memouts.

The results presented show that, even using the external tool which is the least effective in reducing the size of the cores (BooleForce), the effectiveness of the baseline version of our tool is almost equivalent to those of the other tools.

Execution times. Second, we compare our novel approach against the proof-based one with respect to the run time. We compare MathSAT+BooleForce against Math-

SAT on both $\mathcal{T}$-unsatisfiable and $\mathcal{T}$-satisfiable formulas. [12] Figure 5 shows the scatter-plots for $\mathcal{T}$-unsatisfiable (left) and $\mathcal{T}$-satisfiable (right) instances, together with statistics about the execution time ratios of the two solvers. For MATHSAT +BOOLEFORCE, the execution time considered is the sum of the times of MATHSAT and BOOLEFORCE. The experiments show that our proposed method is slightly slower for unsatisfiable problems, but a bit faster for satisfiable problems, and comparable on average.

4.2 Using different propositional unsat core extractors

We have already stressed that one of the advantages of our method is that it benefits for free of all the advances in propositional unsat core extraction. In this last part of our experimental evaluation, we compare the results obtained using different tools. We show that by using different core extractors it is possible to reduce significantly the size of cores and to trade core quality for speed of execution (and vice versa) with no implementation effort. We compare the baseline configuration of our novel system, MATHSAT +BOOLEFORCE, against five other configurations, each calling a different propositional core extractor.

Results are collected in Figures 6 and 7. The first column shows scatter-plots comparing the sizes of the unsat cores, while the third shows those comparing execution times. Plots in the second column are meant to show a more detailed view of the relative performance of the different configurations with respect to core quality; they have the following meaning: the x-axis displays the size (number of clauses) of the problem, whilst the y-axis displays the ratio between the size of the unsat core computed by MATHSAT +BOOLEFORCE and that computed by the other configuration. For instance, a point with y value of $1/2$ means that the unsat core computed by the current configuration is half the size of that computed by MATHSAT +BOOLEFORCE; values above 1 mean the tool behaves worse than plain MATHSAT +BOOLEFORCE.

In figure 6 we evaluated the five configurations which use, respectively, AMUSE [12], EUREKA [4], TRIMMER [7], ZCHAFF [16], and BOOLEFORCE-iter-0%. Looking at the second column, we notice that EUREKA, followed by ZCHAFF and BOOLEFORCE-iter-0%, seems to be the most effective in reducing the size of the final unsat cores, up to $1/2$ the size of those obtained with plain BOOLEFORCE. Looking at the third column, we notice that with AMUSE and ZCHAFF, and in part with EUREKA, efficiency degrades drastically, and many problems cannot be solved within the timeout. With BOOLEFORCE-iter-0% and TRIMMER the execution times improve relevantly, but still we have up to an order magnitude performance gaps w.r.t. the baseline version.

To address this problem, in Figure 7 we evaluated the configurations with BOOLEFORCE-iter-X% for different values of the threshold parameter X ($X = 0$, 2, 5, 10, 20). (Notice that the baseline configuration is equivalent to BOOLEFORCE-iter-100%.) By tuning X we can trade effectiveness for efficiency at will. Remarkably, even $X = 2$ is sufficient to drastically reduce the performance gaps down to an acceptable level, with very limited reduction in core quality.

[12] It would have been interesting to try the same comparison also within YICES, so that to check a slightly different way of internally generating the unsat cores. Unfortunately, the API of YICES does not provide the possibility of retrieving the $\mathcal{T}$-lemmas generated during the search.

It is important to notice that, due to limited time and know-how of the external tools used, we restricted our experiments to their default configurations. Therefore we are confident that even better results, in terms of both effectiveness and efficiency, can be obtained by means of a more accurate tuning of the parameters of the tools.

5 Conclusions

We have presented a novel approach to generating small unsatisfiable cores in SMT, that computes them a posteriori, relying on an external propositional unsat core extractor. The technique is very simple in concept, and straightforward to implement and update. Moreover, it benefits for free of all the advancements in propositional unsat core computation. Our experimental results have shown that, by using different core extractors, it is possible to reduce significantly the size of cores and to trade core quality for speed of execution (and vice versa), with no implementation effort. As a byproduct, our evaluation gives also some insights on the relative performances of these tools.

Future work will include experiments with more fine-tuning of the external boolean unsat core devices, as well as experiments of a "mixed" technique, in which to use the unsat core computed by MathSAT as a starting point. We also plan to investigate the possibility of applying a similar technique to reduce the size of proofs of unsatisfiability, in particular in the context of interpolant generation.

References

1. C. Barrett and S. Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Proceedings of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
2. BOOLEFORCE, `http://fmv.jku.at/booleforce/`.
3. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P.van Rossum, S. Schulz, and R. Sebastiani. An incremental and Layered Procedure for the Satisfiability of Linear Arithmetic Logic. In *Proc. TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
4. N. Dershowitz, Z. Hanna, and A. Nadel. A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In *Proc. SAT'06*, volume 4121 of *LNCS*. Springer, 2006.
5. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
6. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. CAV'04*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
7. R. Gershman, M. Koifman, and O. Strichman. Deriving Small Unsatisfiable Cores with Dominators. In *Proc. CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
8. J. Huang. MUP: a minimal unsatisfiability prover. In *Proc. ASP-DAC '05*. ACM Press, 2005.
9. I. Lynce and J. P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.
10. K. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Proc. CAV '02*, number 2404 in LNCS. Springer, 2002.
11. M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. Marques Silva, and K. A. Sakallah. A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In *Proc. SAT'05*, volume 3569 of *LNCS*. Springer, 2005.
12. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: A Minimally-Unsatisfiable Subformula Extractor. In *Proc. DAC'04*. ACM/IEEE, 2004.

13. H. M. Sheini and K. A. Sakallah. A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic. In *Proc. SAT'05*, volume 3569 of *LNCS*. Springer, 2005.
14. J. Zhang, S. Li, and S. Shen. Extracting Minimum Unsatisfiable Cores with a Greedy Genetic Algorithm. In *Proc. ACAI*, volume 4304 of *LNCS*. Springer, 2006.
15. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. ICCAD '01*. IEEE Press, 2001.
16. Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Proc. of SAT*, 2003.

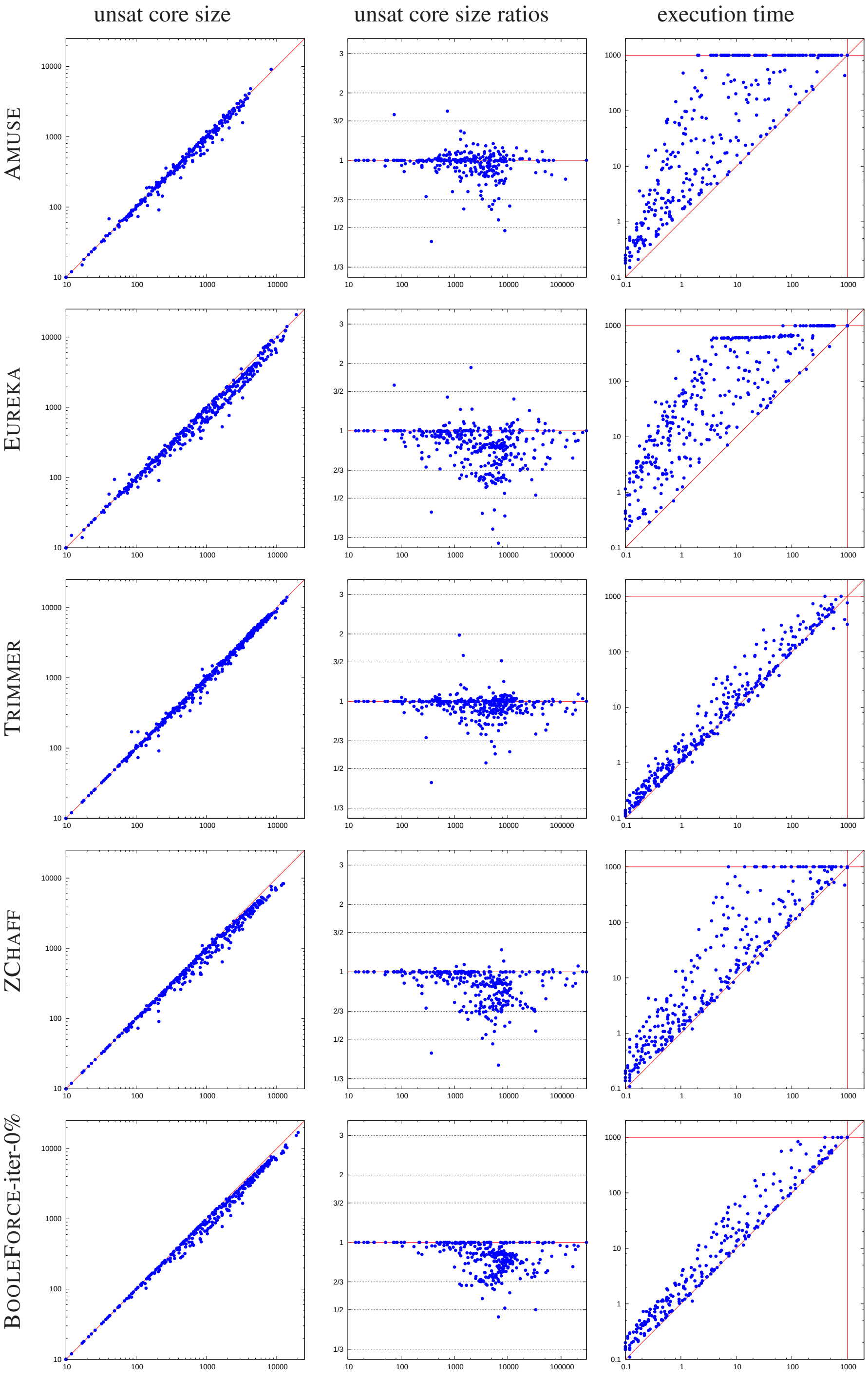


Fig. 6. Comparison of the core sizes (left), core ratios (middle) and run times (right) using different propositional unsat core extractors. For the scatter plots (1st and 3rd column), the baseline system (MATHSAT +BOOLEFORCE) is always on the x-axis. For the ratio plots (2nd column), on the x-axis there's the size of the problem, and on the y-axis the ratio between the size of the cores computed by the two systems: a point above the middle line means better quality for the baseline system.

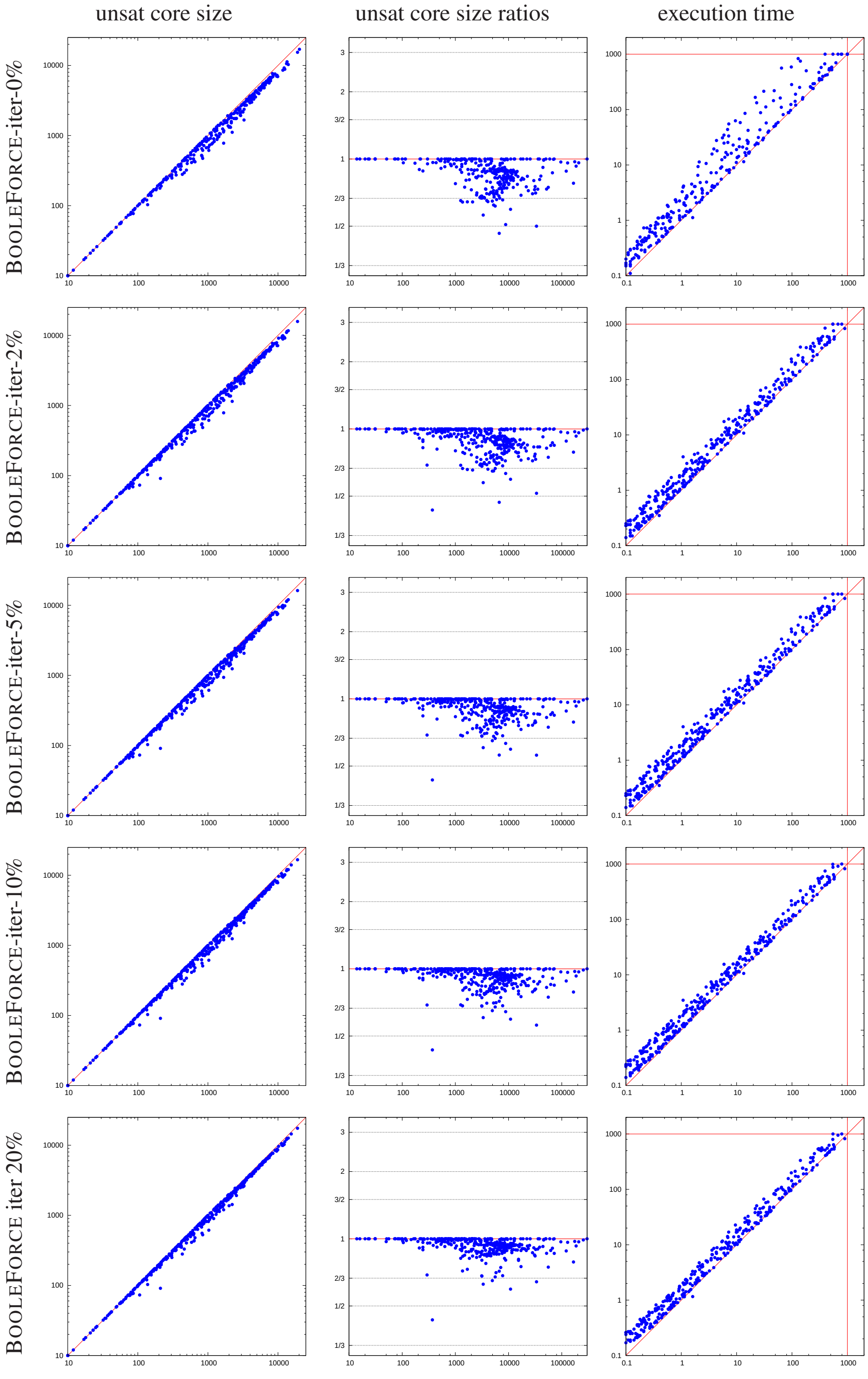


Fig. 7. Comparison of the core sizes (left), core ratios (middle) and run times (right) with BooleForce-iter-X% for different values of the parameter *X*. For the scatter plots (1^{st} and 3^{rd} column), the baseline system (MathSAT +BooleForce) is always on the x-axis. For the ratio plots (2^{nd} column), on the x-axis there's the size of the problem, and on the y-axis the ratio between the size of the cores computed by the two systems: a point above the middle line means better quality for the baseline system.