Minimal unsatisfiable core extraction for SMT

Ofer Guthmann[1], Ofer Strichman[2], Anna Trostanetski[1]
[1] Computer science, Technion, Haifa, Israel. {ofer.guthmann,annat}@cs.technion.ac.il
[2] Information Systems Engineering, IE, Technion, Haifa, Israel. ofers@ie.technion.ac.il

***Abstract*—Finding a minimal (i.e., irreducible) unsatisfiable core (MUC), and high-level minimal unsatisfiable core (also known as group MUC, or GMUC), are well-studied problems in the domain of propositional satisfiability. In contrast, in the domain of SMT, no solver in the public domain produces a minimal or group-minimal core. Several SMT solvers, like Z3, produce a core but do not attempt to minimize it. The SMT solver MATHSAT has an option to try to make the core smaller, but does not guarantee minimality. In this article we present a method and tool, HSMTMUC, for finding MUC and GMUC for SMT solvers. The method is based on the well-known deletion-based MUC extraction that is used in most propositional MUC extractors, together with several new optimizations such as *theory-rotation*, and an adaptive activation strategy based on measurements, during execution, of the time consumed by various components, combined with exponential smoothing. We implemented HSMTMUC on top of Z3 and MATHSAT, and evaluated its performance with hundreds of SMT-LIB benchmarks.**

I. INTRODUCTION

Given an unsatisfiable formula in conjunctive normal form (CNF), an unsatisfiable core (UC) is any subset of these clauses that is still unsatisfiable. In the case of propositional formulas, the problem of finding a minimum core (or rather, the decision problem associated with it) is a Σ_2-complete problem [23] and there were several attempts to cope with it in practice [28], [39]. Given the high-complexity of this problem, there were several attempts to just find *small* cores, without a guarantee of minimality [42], [11], [20]. A large body of work has been dedicated to finding a *minimal* (i.e., irreducible) unsat core (MUC), e.g., [34], [30], [31], [32], which is easier than finding the minimum core, and at least gives the user the guarantee that no single constraint can be removed without making the formula satisfiable. Indeed the only competition ever held in this domain (as part of the SAT competition in 2011) focused on MUC extractions, and now there are several tools that provide this feature, such as MUSER2 [8], HAIFAMUC [31] and MCS-MUS [3]. Most applications of core extraction do not rely on the core being minimal or minimum per-ce, although a small core is desirable; hence striking a balance between efficiency and size of the core is a popular strategy.

The applications of minimal/minimum/small cores of propositional formulas are numerous, including abstraction refinement for model checking [2], [24], [6], formal equivalence verification [26], [16], decision procedures [12], bounded model-checking of multi-threaded systems [22] and functional bi-composition [13] — see [36], [30], [15] for extensive surveys.

When it comes to satisfiability Modulo Theories (SMT), we are aware of several SMT solvers that produce a core, including Z3 [18], CVC3 [5] and YICES [19] but do not attempt to minimize it. A method suggested by Cimatti et al. [15] and implemented in MATHSAT attempts to make the core smaller, but still does not guarantee minimality. We will describe this method in detail and our implementation and experiments with it in Sec. IV. We are aware of one tool, called DFS-FINDER [41], [40], for extracting minimal SMT cores. That tool and the benchmarks it was tested with are not in the public domain. It is based on a deletion-based strategy, and was implemented on top of the SMT solver ARGOLIB [29]. Their focus is on the order in which clauses are removed, which is orthogonal to the techniques we present here.

Whether minimality is important for SMT remains to be seen. As mentioned above most applications of propositional MUC do not *rely* on minimality, but still use a minimal core extractor, since they aspire to use a small core as a 'rule of thumb', namely it is assumed that a small core is better for the rest of the application. Since SMT is used now in many applications that require a core it is reasonable to expect that a tool that finds minimal cores reasonably fast will be used. Microsoft's tool YOGI, for example, a software property-checking tool based on static analysis and testing technology, uses unsat cores in its refinement process [21][1]. Another example is the UFO software model-checker, which uses it to generalize the proof before interpolation [1]. More generally minimality is essential to avoid unnecessary effort in analyzing constraints that are irrelevant for the conflict.

In the current article we show a method for finding a minimal core of SMT formulas, based on the popular deletion-based strategy that is used in several propositional MUC extractors. The basic idea is illustrated in Fig. 1. Given an initial core C, in which all the clauses are unmarked, we remove an unmarked clause $c \in C$ and check for satisfiability of the remaining formula. If the result is SAT, we mark c as necessary for the minimal core, and introduce it back to C. Otherwise (the remaining formula is unsat), we remove clauses outside the new core and continue.

There are many possible optimizations to this basic algorithm, as surveyed in [32], with varying relevance to the case of SMT. Most of them rely on access to a proof (e.g., resolution) and cannot be implemented without changing the SMT solver itself. An exception is Belov and Marques-Silva's recursive model rotation technique [7] (from hereon—rotation), which is both effective and does not require changes to the solver. Indeed we show in Sect. II a generalization of this technique,

[1]Private communication with the authors.

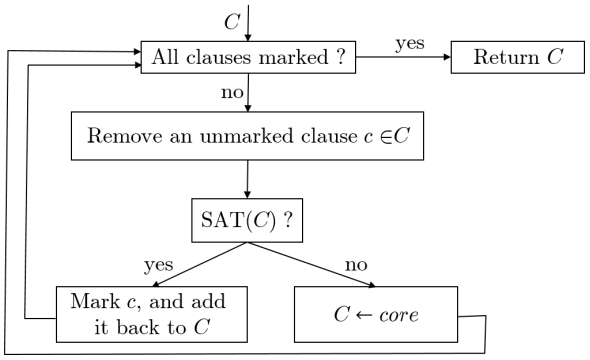

Fig. 1. Deletion-based core extraction. C is an inconsistent set of clauses, all of which are initially unmarked.

which we call *theory rotation*, to the case of SMT. It turns out to be not as effective in the SMT case owing to the cost of checking the T-consistency of assignments, as we will show in Sec. III, but it still improves the run time by about 10% on average, depending on the theory. We used a novel adaptive technique to decide when to activate it based on measurements, during execution, of the time consumed by various components.

We implemented our tool on top of Z3, while relying solely on its API.[2] We did not change Z3 itself, with the hope of supporting other SMT solvers in the future. Z3 has a large user-base and is one of the best solvers, for a large number of theories, according to the latest competitions results of the last few years, and hence is a natural choice. Our results, which we will present in Sect. III, show that our tool HSMTMUC, available from [25], reduces the size of the core by 45% on average. In addition, we support a route via MATHSAT, which turns out to be even faster. It is based on MATHSAT's implementation of Cimatti et al.'s method mentioned above [15], [14] for finding a small core, and then minimizing it with HSMTMUC. We will describe this hybrid approach towards the end of this article, in Sec. IV-E.

II. MINIMIZING SMT UNSATISFIABLE CORES

The standard input language of SMT solvers is that corresponding to the SMT-LIB2 standard [4], which is generally not clausal. A formula is stated via an `assert` statement. Multiple such statements are interpreted as a conjunction between their respective formulas. Z3 tracks which of the assertion statements were used in the proof, and this is what it returns upon a call to `(get-unsat-core)`.

We support two types of core minimizations: minimizing the set of assertions that are inconsistent, and minimizing the set of clauses that result from transforming them to CNF. The first of these goals can be thought of as an application of *High-level minimal unsatisfiable core* [30] (sometimes called *group-MUC*), a problem in which the goal is to minimize the number of high-level constraints in the core, where in this case each assertion is such a constraint. This variant is more useful for human-reading of the output, as it maintains the connection to the original input. The clausal variant can also be useful. For example, one can think of a refinement process that extracts the participating variables from the core, or an engine that extracts the interpolant from the last unsatisfiability proof rather than from the original one, which is likely to make the interpolant smaller. In such applications the mapping of the core to the original formula is not necessary.

Our implementation begins by simplifying the input formula and transforming it to CNF. This is done with the help of Z3's tactics ("simplify" and "tseitin-cnf"). The resulting CNF C is over theory literals and new auxiliary Boolean variables resulting from the Tseitin encoding. Each theory literal l is associated with a propositional variable that encodes whether this literal is true or false in the current assignment . To enjoy the incrementality of Z3, we build the formula such that each clause is guarded with an auxiliary variable, which is then passed as an assumption to the solver. We introduce the guard as a negated auxiliary variable, hence deleting a clause amounts to changing its associated assumption from TRUE to FALSE.

We follow the basic deletion-based method as explained in the introduction and illustrated in Fig. 1 (the center rectangle SAT(C) now corresponds to an SMT call). When solving the high-level variant, each time we remove the whole set of clauses that are associated with a single high-level constraint. On top of that, we implemented an optimization that we call *theory rotation*, for both the high-level and clausal variants of the problem. It does not require any change in Z3 itself. This optimization is the topic of the next subsection.

A. Theory rotation

Suppose that a set of T-clauses C is unsatisfiable, and removing an unmarked clause $c \in C$ makes the formula satisfiable. According to Fig. 1 at this point we should now mark c and put it back in C. In Alg. 1 we show a basic method by which additional clauses can potentially be marked without additional SMT calls. This method generalizes the *rotation* [37] and *recursive model rotation* [7] techniques, which were introduced and proven effective in the domain of propositional MUC extraction.

The idea is the following: given the (propositional) assignment α that satisfies $C \setminus \{c\}$, in line 3 we swap in α the value of one of the variables in c, and call the new assignment α'. This necessarily means that $\alpha' \models c$. If it happens to be the case that α' is T-satisfiable, and contradicts a single unmarked clause $c' \in C \setminus \{c\}$, then we can conclude that c' is also necessary for the core and hence can be marked. The reason is that while C is unsatisfiable, we found a T-satisfiable assignment α' such that $\alpha' \models C \setminus \{c'\}$. Hence in line 5 we mark c'. In the line that follows we call T-ROTATE$_b$ recursively with the new

[2]Specifically, we used the following API functions: Z3_solver_check_assumptions, Z3_solver_get_unsat_core, Z3_solver _reset, Z3_solver_assert, Z3_get_app_decl, Z3_get_app_num_args, Z3_get_app_arg, Z3_mk_not, Z3_mk_or, and Z3_is_eq_ast. To translate the input formula to CNF we apply simplification before and after: tactic t = tactic(ctx, "simplify"), tactic(ctx, "tseitin-cnf"), tactic(ctx, "simplify"); Z3_mk_tactic, Z3_tactic_and_then, Z3_tactic_apply.

assignment α'. The check in line 4 is done lazily, from left to right. Note that the fact that we check in line 4 that c' is unmarked guarantees termination, because 1) each clause can only be marked once, 2) a clause can never be unmarked, and 3) a recursive call happens only after marking a clause.

We continue by suggesting an improvement to this basic procedure, based on the observation that T-ROTATE$_b$ gives up once α' is not T-satisfiable. There is an obvious trade-off between the time invested in attempting to fix α' so that it becomes T-satisfiable and the time it saves by reducing the number of SMT calls. Algorithm T-ROTATE, which appears in Alg. 2, gives the user control over the amount of effort through the bound $flipThreshold$ (see line 6 and the fourth parameter of FLIP). In our winning strategy we set this threshold to two, meaning that we allow for flipping one additional variable in our attempt to make α' T-satisfiable. This strategy increases the number of marked clauses by close to 10% on average, as will be evident in the next section.

The functions T-ROTATE and FLIP in Alg. 2 are mutually recursive, and FLIP is also self-recursive. T-ROTATE is the one called from the main core minimization loop when the removal of c makes C satisfiable. T-ROTATE takes c as input and marks clauses (including c itself), i.e., mark that they belong to the MUC. For each literal $l \in c$, it calls FLIP in line 4.

The conditions checked by FLIP in line 8 are different than those checked in T-ROTATE$_b$, because we want to continue even if α' is not T-consistent (note that in such a case it is possible that $UnsatSet(C, \alpha') \equiv \emptyset$). In such a case we call FLIP recursively with each of the literals in the core (lines 12 – 13), with the hope that after flipping it the new assignment will satisfy all the conditions required for reaching line 10. This invokes FLIP for each literal in the core, which justifies the low value of $flipThreshold$ mentioned above.

Some implementation details: To compute $UnsatSet(C, \alpha')$, we maintain for each literal a set of clauses in which it appears. Hence if we flip l to $\neg l$, we check for (propositional) satisfiability of all the clauses that contain l, in addition to the clause that was unsatisfiable at the entrance to FLIP (when called from T-ROTATE, that clause is guaranteed to become satisfiable after flipping l, by construction. It is not necessarily the case when T-ROTATE is called recursively). Furthermore, since each clause is potentially checked multiple times, under similar assignments, we maintain a map from each clause cl that we check, to the literal lit that satisfies it. When revisiting cl, we first check if lit is still satisfied by the current assignment. If not, we revert to scanning the clause and update the map with a new satisfied literal, if one exists.

B. Theory rotation over high-level constraints

Our implementation of theory rotation for high-level theory constraints appears in Alg. 3. It is a generalization of Alg. 2, based on the propositional high-level rotation that was described in [35], [33]. If, after removing the clauses associated with a high-level constraint H, the formula becomes satisfiable, then we find the set of literals in the intersection of all the clauses in H that are unsatisfied by the current assignment α. We then flip the assignment of each of these literals separately, and check each time if it is both T-consistent and makes a single high-level constraint H' unsat, and H' is unmarked. If both conditions are true, then H' is marked as necessary. The FLIP function that is called in line 8 has the same code as in Alg. 2, except that C is now a set of high-level constraints.

C. Adaptive activation of theory rotation

Our initial experiments showed that T-rotation is frequently not cost-effective, despite the fact that it is polynomial and saves SMT calls which are worst-case exponential. This stands in contrast to the propositional case, where rotation is generally very cost-effective. Reasons for this difference include

- the theory check in line 9 is potentially expensive (yet for most theories still polynomial),
- the success rate is smaller than in the propositional case, because of the additional requirement that the assignment α is T-consistent,
- the attempts to fix α so it becomes T-consistent (line 13) may be expensive if $flipThreshold$ is not small.

Analyzing the logs of our experiments shows that T-rotation has a large impact in the beginning (by 'beginning' we mean after having the initial core from Z3), when there are still many unmarked clauses, but it diminishes through time. The overhead of calling T-rotation while being ineffective at later steps frequently outweighs the initial gain. To overcome this problem, we attempt to stop T-rotation when it is no longer cost-effective. We experimented with two strategies:

- **fail bound**: when x consecutive activations of T-ROTATE produce no marked clauses, we stop.
- **exponential smoothing**: Let t_{smt} be the average time it takes to check T-satisfiability of $C \setminus \{c\}$, t_r the average time it takes to run T-ROTATE, and n_r the average number of clauses that it marks (not including the initial clause c). Had these figures been known and constant throughout the run, we would use T-rotation only if

$$t_{smt} > \frac{t_r}{n_r} \,. \tag{1}$$

Since this is not the case, then as a second best choice we measure these figures at run time, and use them as the basis for estimating (1). They are not purely monotonic, however, and hence terminating T-rotation once (1) does not hold is not a good strategy. On the other hand the number of marked clauses n_r, as hinted before, has a clear trend: in practice we see that it is reduced to near 0 after a while, when the set of unmarked clauses becomes small. The solution we chose is based on *exponential smoothing*, a known technique in statistics that was also used recently in a SAT branching heuristic [27]. The input data can be seen as a stream of tuples $\langle t^0_{smt}, t^0_r, n^0_r\rangle, \langle t^1_{smt}, t^1_r, n^1_r\rangle, \ldots$, where the superscript denotes the time index. We define $T^0_{smt} = t^0_{smt}$

Algorithm 1 A basic theory rotation algorithm.

```
1: function T-ROTATE_b(clause-set C, clause c, assignment α)
2:     for each l ∈ c do
3:         α' := α[l ← ¬l];
4:         if UnsatSet(C, α') ≡ {c'} and c' is unmarked and T-SAT(α') then
5:             mark c';
6:             T-ROTATE_b (C, c', α');
```

Algorithm 2 Theory rotation, in which certain effort is invested in fixing the assignment α so it becomes T-consistent and consequently leads to marking of additional clauses.

Require: C is unsat, and $\alpha \models C \setminus c$

```
1: void T-ROTATE(clause-set C, clause c, assignment α)
2:     mark c;
3:     for each literal l ∈ c do
4:         FLIP(C, l, α, 0);
```

Require: α does not satisfy zero or one clauses from C

```
5: void FLIP(clause-set C, lit l, assignment α, int depth)
6:     if depth ≥ flipThreshold then return ;                    ▷ User-defined threshold
7:     α' := α[l ← ¬l];
8:     if (UnsatSet(C, α') ≡ {c'} and c' is unmarked) or UnsatSet(C, α') ≡ ∅ then
9:         if (T-SAT(α')) then                                   ▷ Theory-checking of α'
10:            T-ROTATE (C, c', α');                             ▷ c' must exist here
11:        else
12:            for each literal l' in core do                    ▷ core = unsat core of line 9
13:                FLIP(C, l', α', depth + 1);
```

Algorithm 3 High-level theory rotation, in which certain effort is invested in fixing the assignment α so it becomes T-consistent and consequently leads to marking of additional clauses.

Require: C is unsat, and $\alpha \models C \setminus c$

```
1: void T-ROTATE(constraint-set C, constraint c, assignment α)
2:     mark c;
3:     intersection := Lit(c)          ▷ Lit(c) is the set of literals that appear in c
4:     for each clause cl ∈ c do
5:         if α ⊭ cl then              ▷ at least one such clause exists since α ⊭ c
6:             intersection := intersection ∩ Lit(cl)   ▷ Lit(cl) is the set of literals that appear in cl
7:     for each literal l ∈ intersection do
8:         FLIP(C, l, α, 0);
```

and

$$T^i_{smt} = \alpha \cdot t^i_{smt} + (1 - \alpha) \cdot T^{i-1}_{smt} \ , \tag{2}$$

where α is a parameter in the range $[0..1]$: the closer it is to 1, the closer the value of T^i_{smt} is to the current input t^i_{smt}. Conversely the closer α is to 0, the more 'smooth' T^i_{smt} becomes. Similarly we define T^i_r and N^i_r, with respect to the t_r and n_r sequences, respectively. We continue with T-rotation while

$$T^i_{smt} > \frac{T^i_r}{N^i_r} \ . \tag{3}$$

In our experiments we used $\alpha = 0.1$.

III. EXPERIMENTS

We experimented with the same 561 benchmarks used in [15], which were selected from SMT-LIB, and include instances from the quantifier-free theories LRA, UF, RDL, LIA and IDL. From those we removed 63 instances that Z3 cannot solve in 10 minutes (our timeout), i.e., solve the formula itself augmented with the auxiliary guard variables. This left us with 498 benchmarks. All experiments and graphs in this section were conducted via HBENCH [38], a performance-benchmarking platform.

For the clausal variant of the problem, Fig. 2 (left) shows a comparison of the size of the default core given by Z3 and the minimal core that our tool, HSMTMUC, emits. Overall

the average reduction in core size is 45%. The plot on the right shows the impact of theory rotation on the number of iterations. Overall the average reduction in the number of iterations is 34%. Although the diagram shows that rotation always reduces the number of iterations (or at least does not increase it), in theory a point to the left of the diagonal *is* possible. This is because rotation may change the order in which clauses are chosen for removal. This, in turn, may impact the run of the SMT solver and produce a different core if the formula is unsatisfiable.

Table I shows more detailed results for the clausal variant. The 'base' configuration is a simple deletion-based strategy, and all others include rotation. The 'b x' label means that we activate the 'fail bound' strategy with a bound x, and the 'exp' label means that we activate the exponential smoothing strategy, both explained in the previous section. Overall, all the rotation strategies that we used improve upon the base configuration, with the 'exp' strategy having the least number of fails within the 10 min. timeout. The 'T-conflict resolved' column presents the number of cases that lines 12–13 in T-ROTATE led to additional marked clauses. Whereas the avg. time it takes to compute the minimal core (the Time column) is close to 30 sec., the avg. time it takes Z3 to compute the initial core is ≈ 2.5 sec.

A detailed analysis shows that the effect of theory rotation depends on the theory itself (to the extent that the benchmarks themselves are representative of the theory). We refer the reader to [25] for detailed results.

We also experimented with the high-level variant. We removed benchmarks that have a single `assert` statement, which left us with 395 benchmarks, out of which 41 timed-out. The last line of the table shows our results. Note that the number of iterations in the high-level variant is an order of magnitude smaller comparing to the clausal variant, which is expected given the nature of the problem (i.e., rather than removing a single clause in each iteration, we remove many). In our experiments rotation had negligible effect in this variant, which is expected given the low number of iterations. We again refer the reader to [25] for detailed results.

Unfortunately we cannot make a full comparison to [41], [40] because neither the tool nor the benchmarks that were used to evaluate it are in the public domain. We could only compare the 15 sample benchmarks for which detailed results were published in [40]. The results show that our tool is over two orders of magnitude faster. Detailed results are available in [25]. We emphasize that [41], [40] is based on a different, less competitive SMT solver (ARGOLIB), and that they used different hardware, hence the comparison only approximates the relation between the tools, not the MUC extraction algorithms themselves.[3]

[3]It seems that they also used a different conversion to CNF, because the number of clauses that they report is different than ours (to both directions), despite the fact that we start from the same SMT-LIB benchmark.

IV. A COMPARISON TO A MINIMIZATION OF THE BOOLEAN ENCODING, AND A HYBRID APPROACH

We now describe in detail our implementation and experiments with a method suggested by Cimatti et. al in [15] that was implemented in MATHSAT, which finds a small SMT core, that is not necessarily minimal. As we will show, our implementation of this method based on Z3 is not competitive with the one in MATHSAT, but a hybrid approach, in which we run HSMTMUC to minimize the result of MATHSAT, is the best configuration we found.

A. Boolean-encoding minimization

Recall that an SMT solver combines a propositional SAT solver and a decision procedure DP_T for a conjunction of T-terms, for each supported theory T. It begins by associating with each T-literal l a new propositional variable which we denote by $e(l)$. Overloading the notation, we denote by $e(\varphi)$ a T-formula φ after all of its literals are encoded this way. Hence $e(\varphi)$ is a propositional *abstraction* of φ. We call $e(\varphi)$ the *propositional skeleton* of φ.

Cimatti et al.'s method for extracting a small unsat SMT core, is based on using a *propositional* MUC extractor for minimizing $e(\varphi) \wedge e(L)$, where L denotes the lemmas generated during the run of the SMT solver. The $e(L)$ clauses are discarded from the core, because L corresponds to theory lemmas that are by construction T-valid, and hence can always be conjoined to the formula. The rest of the core can be mapped back to a set of clauses $\varphi' \subseteq \varphi$, which is guaranteed to be unsatisfiable. This method has a major practical advantage as it leverages existing tools for minimizing propositional cores and is easy to implement if the SMT solver can emit $e(L)$ (Z3 does not support such an option, but we will explain how this can be achieved with Z3 later on). Nevertheless, as noted by the authors, this process does not guarantee minimality of the SMT core. We demonstrate this fact with two examples.

Example 1. *Quoting example 5 from [15], consider*

$$\begin{aligned}\varphi \doteq\ & ((x = 0) \vee (x = 1)) \wedge (\neg(x = 0) \vee (x = 1)) \wedge \\ & ((x = 0) \vee \neg(x = 1)) \wedge (\neg(x = 0) \vee \neg(x = 1)) \ .\end{aligned} \tag{4}$$

It is clear that $e(\varphi)$ is unsatisfiable, and further that all the clauses in $e(\varphi)$ are necessary for maintaining unsatisfiability. Nevertheless the last clause is not necessary and hence this is not a minimal core of φ. □

The example above demonstrates the fact that whereas T-valid clauses *cannot* be part of a *minimal* core (because they are always implied anyway and therefor can be removed), the information that they are valid is lost once the search for a core focuses on the propositional abstraction of φ. This particular problem can be easily fixed by removing T-valid clauses from the resulting core (by calling DP_T for each clause separately), but this still does not guarantee minimality as we show next.

A bigger problem is that once we minimize $e(\varphi) \wedge e(L)$, we are restricted to the lemmas in L, which are not necessarily

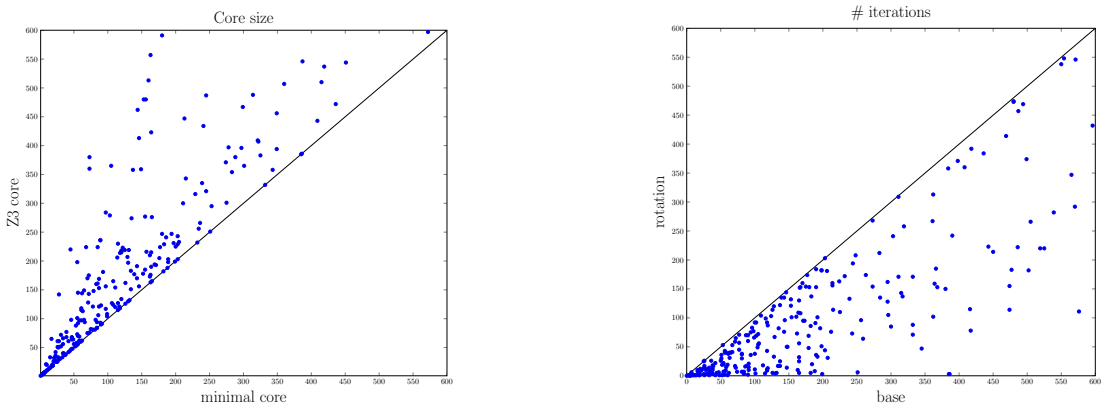

Fig. 2. (left) Z3's default core vs. the minimal core that our tool HSMTMUC emits. (right) The number of iterations with and without theory-rotation.

Config.	# fails	Time (sec.)	Iterations	Rotation Calls	Cls. Marked By Rotation	T-Conflicts Resolved	T-check Time (Sec.)	Init core	Final core
(base)	108	30.5	559.2	0.0	0.0	0.0	0.0	820.2	454.2
T-ROTATE	108	29.7	372.0	472.2	203.7	20.8	1.4	820.2	454.4
T-ROTATE b 5	108	**28.9**	435.9	168.8	130.8	10.2	1.0	820.2	454.5
T-ROTATE b 7	109	29.2	417.1	194.4	143.2	12.3	1.2	820.2	454.4
T-ROTATE exp	**107**	29.6	424.3	244.8	151.8	11.2	1.2	820.2	454.5
HL (base)	41	7.2	45.4	0.0	0.0	0.0	0.0	41.8	26.7

TABLE I

THE TOP FIVE ROWS REFLECT RESULTS WITH DIFFERENT CONFIGURATIONS, OVER 498 SMT BENCHMARKS. OTHER THAN THE # FAILS COLUMN, THE DATA REFLECTS AVERAGES. THE LAST TWO COLUMNS REFERS ONLY TO BENCHMARKS SOLVED BY ALL CONFIGURATIONS. 'INIT CORE' IS THE SIZE OF THE INITIAL CORE EMITTED BY Z3. THE LAST ROW REFERS TO THE HIGH-LEVEL VARIANT, AND IS ONLY OVER THE (395) BENCHMARKS THAT HAVE MORE THAN ONE `assert` STATEMENT.

minimal themselves. The following example demonstrates this problem.

Example 2. *For $x_1, \ldots, x_4 \in \mathcal{R}$, let*

$$\varphi \doteq (x_1 = x_2) \wedge (x_2 = x_4) \wedge (x_1 = x_3) \wedge (x_3 = x_4) \wedge \neg(x_1 = x_4)\ . \tag{5}$$

Suppose now that the following lemma, which is simply a negation of φ, was learned during the search

$$L \doteq \neg(x_1 = x_2) \vee \neg(x_2 = x_4) \vee \neg(x_1 = x_3) \vee \neg(x_3 = x_4) \vee (x_1 = x_4)\ . \tag{6}$$

This is a T-valid statement, although it is not minimal. Now $e(\varphi) \wedge e(L)$ is unsatisfiable, and a minimal core at the propositional *level, after discarding the $e(L)$ clauses, is all the clauses of φ. This core is not minimal with respect to φ, however, because, e.g., the first two clauses can be removed. This* could *have been prevented had the solver inferred the shorter lemma L':*

$$L' \doteq \neg(x_1 = x_3) \vee \neg(x_3 = x_4) \vee (x_1 = x_4)\ , \tag{7}$$

but there is no guarantee for this to happen. □

B. Z3's lemmas and proofs

As described in [14], MATHSAT has a built-in support for logging all the T-Lemmas produced during φ's satisfiability check. However, in the case of Z3, this logging is bypassed, and instead Z3 maintains proof objects during conflict resolution, as described in [17].

A detailed description of Z3's language and proofs has been given in [17], [9], [10]. Z3*'s language* is a many-sorted FOL based on the SMT-LIB language. Z3*'s proof terms* represent natural deduction proof currently using 34 *axioms* and *inference rules*. These inference rules range from simple rules such as **MP** (modus ponens), to complex rules that abbreviate multiple reasoning steps such as **Rewrite** for standard simplification rules, and other theory-specific reasoning, such as **Transitivity**.

Given an unsatisfiable formula φ, Z3*'s proof* is a *directed acyclic graph* (DAG) with a single root. Each node is labeled with a formula: leafs are labeled with either a T-valid formula or one of the original clauses in φ, internal nodes are labeled with a consequent of some T-inference rule, and the root is labeled with $\bot$, i.e., $false$. In the discussion below we will not make the distinction between a node and its label. An edge from a node n to a node n' in the proof represents the fact that n' was used as a premise of an inference rule whose consequent is n. Hence, if n has k children $n_1..n_k$, then

$$(e(n_1) \wedge \cdots \wedge e(n_k)) \rightarrow e(n) \tag{8}$$

represents an encoding of a valid T-implication. Let $e(L)$ be the set of implications of the form (8) corresponding to the entire set of internal nodes and the set of $T-$valid leafs in the proof graph. Then $e(\varphi) \wedge e(L)$ must be unsatisfiable.

C. Implementation using Z3's proof

To implement Cimatti et. el.'s method on top of Z3, we traverse the proof graph produced by Z3[4], and replace each inference with a corresponding propositional lemma (8). Having extracted $e(L)$, it is now possible to apply propositional MUC extraction on $e(\varphi) \wedge e(L)$, as described in Sect. IV-A. Finally, given the propositional MUC, we translate it back to the original T-clauses and check whether it is a minimal core with HSMTMUC. Since we are only interested in the question whether the core is already minimal, for this experiment we terminate HSMTMUC early with "not minimal" once we find a clause that can be removed.

D. Experimental results

We ran our implementation on top of Z3 of the method of [14], with the same 498 benchmarks that were mentioned in Sect. III. For the propositional MUC extractor we used HMUC [32]. We note that the fact that we ask Z3 to log the proof, has an overhead. In experiments reported in [17], it was shown that the memory overhead is $\times 3$ to $\times 40$ greater, with corresponding slowdowns of $\times 1.1$ to $\times 3$. The detailed results appear in Table II. Overall it is not competitive with the implementation in MATHSAT, as will be seen next.

E. Experiments with a hybrid solution

We tested two more configurations: the original implementation of [14] in MATHSAT, and another version in which after running MATHSAT we invoke HSMTMUC to minimize the resulting core. We refer to the two stages of this *hybrid* solution as Hybrid-M (MATHSAT) and Hybrid-H (HSMTMUC). The number of fails are:

HSMTMUC (base)	Hybrid
171	138

Note that those numbers are out of the full set of 561 benchmarks. Hence the 171 fails of HSMTMUC is made of the 108 fails reported in Table I + 63 cases in which Z3 could not produce the initial core within the time limit. The 138 fails of the hybrid approach include 98 fails of MATHSAT itself. Hence, we can see that from the perspective of the number of fails, the hybrid approach is better than HSMTMUC alone for finding a minimal core. In Table III we examine more closely the cases that all three approaches succeeded. As can be seen, we achieve a reduction of 20.9% on average in core size with the hybrid approach, comparing to MATHSAT alone (which, recall, is not necessarily minimal), and a reduction of 9% comparing to HSMTMUC. The total average time of the hybrid approach (11.2 sec. + 16.7 sec.) is larger, however, than invoking HSMTMUC alone (22.9 sec).

Comparing MATHSAT to our implementation on top of Z3, we see that the former is better: it has less fails (98 vs. 164), better run-time on those instances it completes (11.2 sec. vs. 40.4 sec) and smaller average core size (523.0 vs. 723.7). It seems that MATHSAT simply finds proofs that use a smaller number of facts from the original formula φ. It also does not have the overhead of reconstructing the proof as explained in Sec. IV-C, and it uses a different propositional extractor (MUSER2 vs. HMUC).

[4]Using the methods expr::num_args(), expr::arg(i), expr::decl(), fun_decl::decl_kind()

V. CONCLUSIONS AND FUTURE WORK

We presented an algorithm for extracting a minimal unsatisfiable core from SMT unsatisfiable formulas, which is based on a combination of a deletion-based strategy and theory rotation. Many other optimizations exist for the propositional case, such as those published in [8], [32], but they can only be used in the context of SMT if the SMT solver itself is changed. We refrained so far from such changes, with the hope of supporting other SMT solvers that provide a similar API. A highly desirable situation is one in which the initial run of the SMT solver is already biased towards a small core, the same way that the SAT solver is biased towards finding a minimal core in HaifaMuc [32]. For example, make the theory solver return lemmas that contradict as few unmarked clauses as possible. Such an optimization requires theory-specific changes, however, which we leave for future research.

REFERENCES

[1] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Computer Aided Verification - 24th International Conference, CAV*, volume 7358 of *LNCS*, pages 672–678. Springer, 2012.

[2] N. Amla and K. McMillan. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *9th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, 2003.

[3] F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *Computer Aided Verification - 27th International Conference, CAV*, volume 9207 of *LNCS*, pages 70–86. Springer, 2015.

[4] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at http://www.SMT-LIB.org.

[5] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[6] A. Belov, H. Chen, A. Mishchenko, and J. Marques-Silva. Core minimization in SAT-based abstraction. In *DATE*, pages 1411–1416, 2013.

[7] A. Belov and J. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, pages 37–40, 2011.

[8] A. Belov and J. Marques-Silva. MUSer2: An efficient MUS extractor. *J. on Satisfiability, Boolean Modeling and Computation (JSAT)*, 8(1/2):123–128, 2012.

[9] S. Böhme. Proof reconstruction for z3 in isabelle/hol. In *7th International Workshop on Satisfiability Modulo Theories (SMT09)*, 2009.

[10] S. Böhme and T. Weber. Fast lcf-style proof reconstruction for z3. In *Interactive Theorem Proving*, pages 179–194. Springer, 2010.

[11] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Appl. Math.*, 130(2):85–100, 2003.

[12] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *Software Tools for Technology Transfer (STTT)*, 11:95 – 104, 2009.

[13] H. Chen and J. Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In *VLSI-SoC (Selected Papers)*, pages 52–72, 2011.

[14] A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 334–339. Springer, 2007.

Family	# Benchmarks	# Fails	# Minimal	Time (Sec.)	# lemmas extracted	Init core	Core size
QF_IDL	71	4	43	6.2	3109.5	1120.7	529.9
QF_LIA	115	15	20	13.5	3670.1	1306.4	824.6
QF_LRA	116	14	26	46.	14134.0	1094.8	794.9
QF_RDL	81	20	14	72.2	9281.0	2398.6	1257.3
QF_UF	115	48	2	76.2	54394.6	242.4	172.6
Total	498	101	105	40.4	15686.6	1208.9	723.7

TABLE II
RESULTS OF OUR IMPLEMENTATION OF CIMATTI'S ET. AL.'S METHOD ON TOP OF Z3, OVER 498 SMT BENCHMARKS. COLUMNS 4 – 8 REFER ONLY TO BENCHMARKS THAT RAN TO COMPLETION.

Family	# Benchmarks	Original size	Core size			Time			Core Reduction %
			Hybrid-M	Hybrid-H	HSMTMUC	Hybrid-M	Hybrid-H	HSMTMUC	
QF_IDL	68	30659.5	534.3	525.8	515.9	4.5	6.0	8.5	1.6
QF_LIA	100	3490.1	614.1	496.0	567.3	18.7	13.6	11.9	22.5
QF_LRA	83	1908.3	448.7	330.2	405.5	2.2	20.8	47.6	31.7
QF_RDL	53	13353.3	872.1	779.2	813.6	26.5	16.6	28.2	15.7
QF_UF	71	2007.3	209.9	105.0	107.7	5.8	26.6	19.7	50.0
Total	375	9180.0	523.0	428.8	470.0	11.2	16.7	22.9	20.9

TABLE III
CHECKING THE HYBRID APPROACH, OVER THE 375 (OUT OF 561) BENCHMARKS SOLVED BY ALL CONFIGURATIONS TO COMPLETION. THE LAST COLUMN REFERS TO THE REDUCTION IN CORE SIZE BY THE HYBRID APPROACH, COMPARING TO MATHSAT ALONE.

[15] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Intell. Res. (JAIR)*, 40:701–728, 2011.

[16] O. Cohen, M. Gordon, M. Lifshits, A. Nadel, and V. Ryvchin. Designers work less with quality formal equivalence checking. In *Design and Verification Conference (DVCon)*, 2010.

[17] L. M. de Moura and N. Bjørner. Proofs and refutations, and z3. In *LPAR Workshops*, volume 418, pages 123–132, 2008.

[18] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[19] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification (CAV), 18th International Conference*, volume 4144 of *LNCS*, pages 81–94, 2006.

[20] R. Gershman, M. Koifman, and O. Strichman. An approach for extracting a small unsatisfiable core. *Formal Methods in System Design (FMSD)*, 33:1 – 27, 2008.

[21] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 43–56. ACM, 2010.

[22] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL*, pages 122–131. ACM Press, 2005.

[23] A. Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon University, 2006.

[24] A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *ICCAD*, pages 416–423, 2003.

[25] O. Guthmann, O. Strichman, and A. Trostanetski. HSmtMuc:. http://ie.technion.ac.il/~ofers/haifasolvers/site/site.html.

[26] Z. Khasidashvili, D. Kaiss, and D. Bustan. A compositional theory for post-reboot observational equivalence checking of hardware. In *FMCAD*, pages 136–143, 2009.

[27] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In N. Piterman, editor, *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, volume 9434 of *LNCS*, pages 225–241. Springer, 2015.

[28] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 305–310, 2004.

[29] F. Maric and P. Janicic. Argo-Lib: A generic platform for decision procedures. In D. A. Basin and M. Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR*, volume 3097 of *LNCS*, pages 213–217. Springer, 2004.

[30] A. Nadel. Boosting minimal unsatisfiable core extraction. In R. Bloem and N. Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 221–229. IEEE, 2010.

[31] A. Nadel, V. Ryvchin, and O. Strichman. Efficient mus extraction with resolution. In *FMCAD*, pages 197–200, 2013.

[32] A. Nadel, V. Ryvchin, and O. Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:27–51, 2014.

[33] A. Nadel, V. Ryvchin, and O. Strichman. Ultimately incremental SAT. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, volume 8561 of *LNCS*, pages 206–218. Springer, 2014.

[34] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.

[35] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *SAT*, pages 174–187, 2011.

[36] J. P. M. Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL'10*, pages 9–14, 2010.

[37] J. P. M. Silva and I. Lynce. On improving MUS extraction algorithms. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference*, volume 6695 of *LNCS*, pages 159–173, 2011.

[38] O. Strichman. HBench – a platform for performance benchmarking. http://strichman.net.technion.ac.il/hbench/.

[39] J. Zhang, S. Li, and S. Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In *Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence*, volume 4304 of *LNCS*, pages 847–856. Springer, 2006.

[40] J. Zhang, S. Shen, J. Zhang, W. Xu, and S. Li. Extracting minimal unsatisfiable subformulas in satisfiability modulo theories. *Comput. Sci. Inf. Syst.*, 8(3):693–710, 2011.

[41] J. Zhang, W. Xu, J. Zhang, S. Shen, Z. Pang, T. Li, J. Xia, and S. Li. Finding first-order minimal unsatisfiable cores with a heuristic depth-first-search algorithm. In *Intelligent Data Engineering and Automated Learning - IDEAL*, volume 6936 of *LNCS*, pages 178–185, 2011.

[42] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas, 2003.