Boosting Minimal Unsatisfiable Core Extraction

Alexander Nadel
Intel Corporation, P.O. Box 1659, Haifa 31015 Israel
Email: alexander.nadel@intel.com

***Abstract*—A variety of tasks in formal verification require finding small or minimal unsatisfiable cores (subsets) of an unsatisfiable set of constraints. This paper proposes two algorithms for finding a minimal unsatisfiable core or, if a time-out occurs, a small non-minimal unsatisfiable core. Our algorithms can be applied to either standard clause-level unsatisfiable core extraction or high-level unsatisfiable core extraction, that is, an extraction of an unsatisfiable core in terms of "interesting" propositional constraints supplied by the user application. We demonstrate that one of our algorithms outperforms existing algorithms for clause-level minimal unsatisfiable core extraction on large well-known industrial benchmarks. We also show that our algorithms are highly scalable for the problem of high-level minimal unsatisfiable core extraction on huge benchmarks generated by Intel's proof-based abstraction refinement flow. In addition, we provide a comparative analysis of the impact of various algorithms on unsatisfiable core extraction.**

I. Introduction

Given an unsatisfiable formula in Conjunctive Normal Form (CNF), a *(clause-level) unsatisfiable core (UC)* is an unsatisfiable subset of its clauses. A *(clause-level) minimal unsatisfiable core (MUC)* is a clause-level UC that becomes satisfiable when any one of its clauses is removed. The problem for finding a small, a minimal, the smallest minimal, or all the minimal unsatisfiable cores has been addressed frequently in recent years [1]–[19], mainly due to the increasing importance of this problem in formal verification.

While clause-level UC extraction is widely used, the formulation of the problem of extracting a clause-level core implicitly assumes that a "good" core should contain as few clauses as possible, whereas many real-world applications require minimizing the number of high-level propositional *interesting constraints* in the core. A *high-level small/minimal unsatisfiable core* is a small/minimal subset of the interesting constraints, whose conjunction with the other constraints in the system is unsatisfiable.

In [13] an algorithm for finding all the high-level MUCs is proposed and applied during the refinement stage of the datapath abstraction refinement-based approach to formal equivalence verification (FEV) described in [20]. Specifically, an abstract counterexample is written as a set of interesting constraints. The abstract counterexample is encoded into CNF in order to find corresponding concrete bit-level counterexamples. If the CNF instance is unsatisfiable, then no such concretization exists, and the abstract counterexample is spurious. In this case, high-level MUCs are used to locate the source of infeasibility and refine the abstraction. The algorithm of [13] is reviewed in Section II.

High-level MUC extraction is used for compositional FEV [21], [22] in [23]. In compositional FEV, the design and the implementation are decomposed into pairs of corresponding slices. By proving the equivalence of all the pairs one can infer the equivalence of the models. It is essential for fast and correct FEV to allow the user (the designer) to specify assumptions that mimic the environment for each pair of slices. These assumptions can be used for the proof of equivalence, but the correctness of each assumption that impacts the proof must be proved separately afterwards. High-level MUC extraction, where the assumptions serve as the interesting constraints, is used to identify the assumptions that were relevant for the equivalence proof. The algorithm of finding a high-level MUC is only briefly sketched in [23] (in fact, a preliminary version of our Alg. 2 is used).

Another example where high-level UC extraction can be applicable is proof-based abstraction refinement for SAT-based hardware model checking, proposed independently in [24] and [25]. This algorithm uses bounded model checking (BMC) for increasing depths on the concrete design. When there is no counterexample up to a given depth, an UC is identified for this depth and an abstraction based on latches and/or gates is used to generate an abstract model which is then proved using complete model checking techniques. While the existing literature uses clause-level UC extraction for finding the abstraction, it would be more appropriate to use high-level UC extraction for this purpose, since the algorithm clearly needs UCs in terms of latches and/or gates, rather than clauses.

Finding one non-minimal core is the cheapest alternative in terms of run-time, but the least precise in terms of the size and accuracy of the core. Extracting all the minimal cores is the most precise, albeit the most costly, option. Finding one minimal core is a reasonable compromise between accuracy and run-time. In this paper we introduce two new algorithms applicable for both high-level and clause-level single MUC extraction. They can also return a small non-minimal core if a time-out occurs after the initial approximation stage, where the larger the time-out the smaller the core will be. One of the algorithms generalizes and improves the resolution-based approach to clause-level MUC extraction [6]–[8], while the other uses the selector variable-based approach to clause-level non-minimal UC extraction of [4], [11] as the starting point. We show that one of our algorithms, given large industrial benchmarks, yields empirically better results than previous approaches to clause-level MUC extraction. We demonstrate the scalability of our algorithms for high-level MUC core extraction using huge benchmarks generated by Intel's imple-

mentation of the proof-based abstraction refinement flow [24], [25]. Also, our work provides an extensive comparison between our new resolution-based and selector variable-based approaches to MUC extraction. Furthermore, we analyze the impact of the following on resolution-based MUC extraction: (1) different approaches to incremental SAT solving (pervasive clause reuse [26] versus reusing a single SAT instance [27]); (2) RRP (Resolution Refutation-based Pruning) [6]–[8]; (3) in-memory data structures with reference counters [9]–[11] versus on-disk data structures [1], [2].

The rest of the paper is organized as follows. Section II provides the necessary background and surveys the related work. Sections III and IV introduce our approaches (resolution-based and selector variable-based, respectively) to extracting a MUC. Section V presents and analyzes the experimental results. Section VI concludes our work.

II. Background and Related Work

We start this section with an overview of algorithms for incremental SAT solving, whose relevance to UC extraction will be explained shortly.

A. Incremental SAT Solving

Incremental SAT solving is intended to boost the solving of closely related SAT instances, which share clauses. It was noted in [26] that *pervasive clause reuse* (that is the reuse of learned clauses derived from shared input clauses in consecutive SAT invocations) provides a significant performance boost in SAT-based Automatic Test Pattern Generation. Another *single SAT instance-based* approach to incremental SAT solving was proposed in [27] in the context of incremental model checking and implemented in the Minisat SAT solver [28]. Minisat re-uses a single SAT instance for all the related invocations. After the solving is completed, one can add new clauses to Minisat and re-invoke the solver on the incremented instance. The single SAT instance-based approach is preferable to the pervasive clause reuse approach, since it reuses not only the relevant conflict clauses, but also all the information necessary for the decision and conflict clause deletion heuristics. However, it suffers from the drawback that it is not *decremental*, that is, it does not allow removing clauses between consecutive SAT invocations. *SAT solving under assumptions* [27] (also implemented in Minisat) provides a solution to this problem by allowing the user to supply a set of *assumptions* $Y = \{y_1, y_2, \ldots, y_m\}$ (where each assumption y_i is a literal) along with the input formula F. The solver returns "satisfiable" iff $F \wedge Y$ is satisfiable. The user application can augment related clauses that are expected to be removed with the negation of a literal l and assert these clauses when required by adding l to Y. An additional useful feature is that when $F \wedge Y$ is unsatisfiable, Minisat can return a small subset of the assumptions $Y' \subseteq Y$, called the *relevant assumptions*, such that $F \wedge Y'$ is still unsatisfiable [28]. The algorithm for returning the set of relevant assumptions is very cheap and requires only minimal changes to the solver. All the Y literals are picked as decision variables before all the other variables and are assigned true. Then standard SAT solving is used. The algorithm terminates when one of the assumptions y is forced to be false in clause C by Boolean Constraint Propagation (BCP). In this case the assumptions cannot hold together. Minisat resolves the C with all its predecessors in the implication graph until a clause containing the negations of Y's literals only is generated. The negation of this clause is returned as the set (conjunction) of relevant assumptions.

B. Unsatisfiable Core Extraction

The most scalable approach to extracting a small clause-level UC is the *resolution-based approach*. It is based on the ability of modern SAT solvers to store a resolution derivation during the process of solving and to generate a resolution refutation of a given unsatisfiable formula at the end. The basic resolution-based approach, discovered independently in [1] and [2], returns all the initial clauses connected to the empty clause $\square$ as the UC. This approach imposes little overhead on the SAT solver, hence it can handle huge instances having millions of clauses. Two methods for trimming the size of the core were proposed in [1] and [5], based on invoking the basic resolution-based approach until a fixed point is reached and manipulating the resolution refutation, respectively. Neither of these methods guarantees minimality.

A resolution-based algorithm for extracting a minimal UC, called Complete Resolution Refutation (CRR), was proposed in [6]–[8]. CRR first finds a resolution refutation π of the input formula and removes clauses that are not connected to the empty clause $\square$. Then, for each remaining input clause C, CRR removes the cone of C from π and invokes a SAT solver on the rest of the remaining clauses, including the conflict clauses. If the formula is satisfiable, then C belongs to a MUC; otherwise CRR removes all the clauses not connected to $\square$ from π and continues the loop until all the input clauses are either removed or are proved to belong to the MUC. CRR uses the pervasive clause reuse approach to incremental SAT solving: it invokes the SAT solver many times on related instances, re-using all the relevant conflict clauses. CRR's performance can be enhanced by applying a technique known as Resolution Refutation-based Pruning (RRP) [6]–[8], which is briefly described in Section III. CRR with RRP scales well for difficult industrial instances having up to one or two hundred clauses [6]–[8].

The early implementations of resolution-based algorithms for UC extraction stored the resolution derivation on disk [1], [2]. Several independent researches realized that the performance of UC extraction could be improved by storing the resolution derivation in memory. In [6] it was suggested as a direction for future work that storing the resolution derivation in memory could boost CRR. BooleForce [29] was the first solver to store the resolution derivation in memory (for extracting non-minimal UCs). An efficient implementation of the in-memory algorithm, based on reference counters, was proposed independently in [9]–[11]. The key observation is that if there are no references to the clause from either the

instance or the resolution derivation, it can safely be removed from the resolution derivation.

Now we describe another prominent approach to UC extraction–the *selector variable-based approach*–introduced in the AMUSE tool for non-minimal clause-level UC extraction [4]. This approach adds the negation of a fresh *selector variable* from a subset Y to each input clause. The SAT solver is then guided to assert the clauses by setting the selector variable to true whenever possible. In the end, the algorithm derives a Y-conflict clause containing a subset of the selector variables. The core consists of clauses the negation of whose corresponding selector variables belongs to the Y-conflict clause. AMUSE implementation requires changing the internals of the SAT solver. A very similar algorithm for non-minimal clause-level UC extraction which does not require changing the SAT solver was proposed in [11]. It provides the selector variables as assumptions, along with the formula augmented by selector variables, to Minisat. The UC consists of clauses whose selector variables are returned by Minisat as the relevant assumptions. Unlike the resolution-based approach, the selector variable-based approach does not need to store a resolution derivation. However, its major drawback is that adding selector variables causes the SAT solver to generate very long learned clauses, making it so that the algorithm does not scale well even to medium-size instances for clause-level MUC extraction. The selector variable-based approach to non-minimal clause-level UC extraction was shown to be inferior to the basic resolution-based approach in [11]. Moreover, AMUSE was shown to be much slower than the CRR algorithm in [6]–[8], even though AMUSE, unlike CRR, does not guarantee the minimality of the core. The selector variable-based approach can be extended for generating a number of clause-based UCs [4], the smallest clause-based MUC [12], and all the clause-based MUCs [13].

An algorithm for generating all the clause-level or all the high-level MUCs, called CAMUS, is proposed in [13]. First CAMUS computes the set of all the minimal correction subsets (MCSs) of a given unsatisfiable problem, where a correction subset is a subset of the constraints whose removal results in a satisfiable set of constraints. Then it finds the set of all the irreducible hitting sets of the MCSs, which is exactly the set of all the MUCs. The first stage of this algorithm is very costly, since it has to find *all* the MCSs. Yet, a version of CAMUS for finding all the high-level MUCs was successfully applied to formulas from the datapath abstraction domain [20] having more than one hundred clauses. The efficiency of the high-level MUCs extraction is achieved using Minisat's feature of SAT solving under assumptions with relevant assumption extraction as an underlying reasoning engine. In our context, it is important to note that the high-level MUCs extraction mode of CAMUS marks all the clauses that correspond to a particular interesting constraint with a particular selector variable. This operation allows CAMUS to use Minisat's features for reasoning about interesting constraints. Our Alg. 3 for finding a single MUC uses this operation as well, but, unlike CAMUS, we apply it to the problem of UC extraction

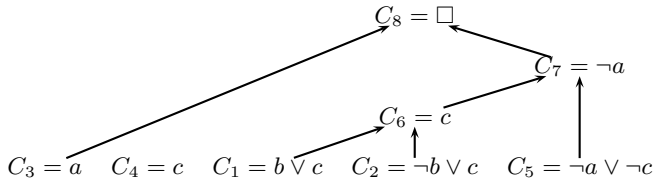

Fig. 1: An example. Assume $\Psi = \{R_1 = \{C_1, C_2\}, R_2 = \{C_3, C_4\}\}$; $\Omega = \{C_5\}$. Note that the only high-level MUC is $\{R_2\}$. A resolution refutation of *Clss*$(\Psi \wedge \Omega)$, addressed in the text, is shown.

.

in a straightforward manner which makes our high-level MUC extraction algorithm scalable to instances having millions of clauses. Note that although the second stage of CAMUS can easily be modified to return only one high-level MUC, this option does not seem to be practical, since CAMUS's first stage is clear overkill when only one MUC is required to be found.

III. Resolution-based Minimal Unsatisfiable Core Extraction

In this section we introduce a new resolution-based algorithm for high-level and clause-level minimal UC extraction. It may also return a non-minimal core if a time-out occurs after the initial approximation stage.

A. Definitions

We need to provide a number of well-known notions related to resolution. The *resolution rule* states that given clauses $D_1 = A \vee v$ and $D_2 = B \vee \neg v$, where A and B are also clauses, we can derive the clause $C = A \vee B$. The resolution rule application is denoted by $C = D_1 \otimes^v D_2$. A *resolution derivation* of a *target clause* C from a CNF formula F is a sequence $\pi = (C_1, C_2, \ldots, C_p \equiv C)$, where each clause C_i is either a clause of F (an initial clause) or derived by applying the resolution rule to C_j and C_k, where $j, k < i$ (a derived clause). A *resolution refutation* is a resolution derivation of the empty clause $\square$. A resolution derivation $\pi = (C_1, C_2, \ldots, C_p)$ can naturally be considered as a directed acyclic graph (dag) whose vertices correspond to all the clauses of π and in which there is an edge from a clause C_j to a clause C_i iff $C_i = C_j \otimes C_k$ (an example of such a dag appears in Fig. 1). Let π be a resolution derivation. A clause $D \in \pi$ is *reachable* from $C \in \pi$ if there is a path (of 0 or more edges) from C to D. The set of all vertices reachable from $C \in \pi$ (or from $\rho \subseteq \pi$), called the *cone* of C (or ρ), is denoted $Re(\pi, C)$ (or $Re(\pi, \rho)$). For the example in Fig. 1, $Re(\pi, \rho = \{C_1, C_2\}) = \{C_1, C_2, C_6, C_7, C_8\}$.

Now we provide definitions related to high-level UC extraction. Given a conjunction (set) of propositional formulas $\Psi = \{R_1, R_2, \ldots, R_m\}$ and a propositional formula Ω, such that $\Psi \wedge \Omega$ is unsatisfiable, $UC(\Psi, \Omega) \subseteq \Psi$ is a *high-level unsatisfiable core*, if $UC(\Psi, \Omega) \wedge \Omega$ is unsatisfiable. Each $R_i \in \Psi$ is an *interesting constraint (IC)* and the set Ω is the *remainder*. A high-level UC is *minimal* if removing any of its ICs makes its conjunction with the remainder satisfiable.

A *clause projection* $\mathit{Clss}(F)$ of a propositional formula F is a set of clauses equisatisfiable to F, generated by applying Tseitin encoding [30]. We sometimes refer to a formula F, meaning the associated clause projection $\mathit{Clss}(F)$.

Next we introduce our resolution-based algorithm for high-level MUC extraction. For clarity of presentation we start with a simple (yet novel) Alg. 1, which serves as the basis for the eventual Alg. 2.

B. The Basic Algorithm

Alg. 1 receives a set of ICs and the remainder. Its initial *approximation stage* (the first two lines) approximates a high-level MUC *muc* by placing in *muc* ICs whose intersection with a clause-level non-minimal UC is non-empty[1]. The clause-level non-minimal core is found using the basic resolution-based approach [1], [2]. The approximation stage of Alg. 1 corresponds to the "folk" algorithm for finding a high-level non-minimal UC. Note that even if the clause-level core is minimal, the high-level core is not necessarily minimal (this observation also holds for Alg. 2). Consider the example in Fig. 1. The set of clauses $\{C_1, C_2, C_3, C_5\}$ is a clause-level MUC of $\mathit{Clss}(\Psi \wedge \Omega)$. However, the corresponding set of interesting high-level constraints $\{R_1, R_2\}$ is not a high-level MUC.

Assume now that the algorithm enters the *minimization loop* (the "for all" loop). It simply goes over all the ICs remaining in *muc* and checks if a particular *removal candidate* R_i can be removed by invoking a SAT solver over the clause projection of the remainder and $\mathit{muc} \setminus \{R_i\}$. In the end, *muc* is a high-level MUC.

Note that if a time-out occurs during the minimization stage, the algorithm can still return a reduced, but not necessarily minimal, core. This property also holds for Alg. 2. We describe another property holding for both algorithms. This property is essential for guaranteeing that the algorithms indeed return a minimal core. Consider an IC R_j, such that $j \in \mathit{muc}$, but R_j is not the removal candidate for a certain minimization loop iteration. Note that all the clauses $\mathit{Clss}(R_j)$ are sent to the SAT solver, even if some of the clauses of $\mathit{Clss}(R_j)$ did not participate in the clause-level core returned by the SAT solver during the approximation stage. For example, suppose that the resolution refutation in Fig. 1 corresponds to the situation just after completion of the approximation stage. Assume that the removal candidate for the first iteration of the minimization loop is R_1. The clause $C_4 \in \mathit{Clss}(R_2)$ is not connected to $\square$. However, it must be sent to the SAT solver, otherwise the algorithm will erroneously conclude that R_1 must belong to the minimal core. Likewise, all the clauses in the clause projection of the remainder are sent to the SAT solver.

The main drawback of Alg. 1 is the lack of incrementality. The SAT solver is invoked each time on a new formula, while the learned conflict clauses and heuristical information are lost.

[1]We assume here and elsewhere in the paper that the clause projection of each constraint (either an interesting constraint or the remainder) is created by appying Tseitin encoding which generates *new* auxiliary variables for each translated entity.

Algorithm 1 Basic high-level MUC extraction

Require: $\Psi = \{R_1, R_2, \ldots, R_m\} \wedge \Omega$ is unsatisfiable
1: Extract a clause-level non-minimal unsatisfiable core $\overline{F}$ using the basic resolution-based approach
2: $\mathit{muc} := \{i \mid \mathit{Clss}(R_i) \cap \overline{F} \neq \emptyset\}$
3: **for all** $i \in \mathit{muc}$ **do**
4: Invoke a SAT solver on $\mathit{Clss}(\Omega \wedge \{R_j \mid j \in \mathit{muc} \setminus \{i\}\})$
5: **if** the result is "unsatisfiable" **then**
6: $\mathit{muc} := \mathit{muc} \setminus \{i\}$
7: **return** $\{R_i \mid i \in \mathit{muc}\}$

We would like to extend Alg. 1 so that it would reuse the same SAT instance. To be able to check whether a removal candidate belongs to the core, we need to have the ability to *conditionally remove* the cone of the removal candidate (that is, to remove the cone while maintaining the possibility of returning it efficiently), since this cone corresponds exactly to the removal candidate and all its logical consequences. In addition, we need to support both the efficient *return* of conditionally removed clauses to the SAT instance for cases where the removal candidate belongs to the minimal core, and the efficient *unconditional removal* of clauses to support the operation of removing ICs from the core. We will describe how we implemented these operations after presenting the flow of Alg. 2.

C. The Final Algorithm

Alg. 2 uses an incremental SAT solver (which also maintains a resolution derivation) and assumes that it returns a triplet that contains the result (which can either be "satisfiable" or "unsatisfiable"), an updated SAT instance, and an updated resolution derivation. The approximation stage of Alg. 2 (from the beginning until line 7) invokes the SAT solver over the set of ICs and the remainder. The cones (of ICs) that do not include $\square$ are removed from the instance forever. The algorithm maintains a set of *minimal core candidates*, *muc_cands*, initialized with the indexes of ICs whose cone includes $\square$. It is not known whether the ICs in *muc_cands* belong to the minimal core. The algorithm also maintains a set of *minimal core habitants*, *muc*, which contains ICs that belong to the minimal UC. Consider the minimization loop (the "while" loop). Each iteration picks a removal candidate from the minimal core candidates. It conditionally removes the cone of the removal candidate from the SAT instance and invokes the SAT solver. If the instance is satisfiable, the removal candidate is guaranteed to belong to the minimal core, and hence it is moved from *muc_cands* to *muc* and its cone is returned to the instance. If the instance is unsatisfiable, the algorithm refines the minimal core candidates by keeping there only those ICs whose cone includes $\square$. Cones of other ICs are removed forever. Hence one iteration of the loop may remove more than one IC from the set of minimal core candidates. In the end, the algorithm returns the set of minimal core habitants as the high-level MUC.

Now we will discuss implementation details which are critical for performance. Conditionally removed clauses are

not deleted from the clause database, since this would make returning them cumbersome and costly. Rather, we make sure that these clauses are ignored by the solver's major algorithms, including Boolean Constraint Propagation (BCP) and clause-based heuristics (if such a heuristic, e.g., CBH [31], is used). This is done as follows. We remove the clauses from the WL data structure [32], which is used for BCP, then we mark the clauses and guide the heuristic to ignore the marked clauses. To return the conditionally removed clauses, it is sufficient to reinsert them into the WL data structure and unmark them for the clause-based heuristics. In addition, our implementation groups the following two operations into one pass over the clauses carried out just after executing line 10: (1) finding and conditionally removing the cone of the current removal candidate; (2) finding and either returning or unconditionally removing the cone of the previous removal candidate (or, for the first iteration only, unconditionally removing the cones of ICs found to be irrelevant during the approximation stage).

Algorithm 2 Resolution-based high-level MUC extraction

Require: $\Psi = \{R_1, R_2, \ldots, R_m\} \wedge \Omega$ is unsatisfiable
1: Initialize the SAT instance *SI* with $Clss(\Psi \wedge \Omega)$ and associate a resolution derivation π with *SI*
2: $\langle res, SI, \pi \rangle :=$ SAT(*SI*)
3: **for** $i \in 1 \ldots m$ **do**
4: **if** $\Box \notin Re(\pi, Clss(R_i))$ **then**
5: Remove $Re(\pi, Clss(R_i))$ from *SI* forever
6: *muc_cands* $:= \{i \mid \Box \in Re(\pi, Clss(R_i))\}$
7: *muc* $:= \{\}$
8: **while** *muc_cands* is non-empty **do**
9: $k :=$ a member of *muc_cands* $\setminus$ *muc*
10: Conditionally remove $Re(\pi, Clss(R_k))$ from *SI*
11: *muc_cands* $:=$ *muc_cands* $\setminus \{k\}$
12: $\langle res, SI, \pi \rangle :=$ SAT(*SI*)
13: **if** *res* $=$ satisfiable **then**
14: Return $Re(\pi, Clss(R_k))$ to *SI*
15: *muc* $:=$ *muc* $\cup \{k\}$
16: **else**
17: Remove $Re(\pi, Clss(R_k))$ from *SI* forever
18: **for** $i \in$ *muc_cands* **do**
19: **if** $\Box \notin Re(\pi, Clss(R_i))$ **then**
20: Remove $Re(\pi, Clss(R_i))$ from *SI* forever
21: *muc_cands* $:=$ *muc_cands* $\setminus \{R_i\}$
22: **return** $\{R_i \mid i \in$ *muc*$\}$

Standard clause-level MUC extraction is a particular case of high-level MUC extraction where each IC consists of a single clause and the remainder is empty. Consider Alg. 2 as an algorithm for clause-level MUC extraction and compare it to the CRR algorithm [6]–[8] described in Section II. The algorithms have a similar structure. Both try to reuse all the relevant conflict clauses between different iterations of the minimization loop. The main difference between them is that while CRR creates a new SAT instance for each minimization loop iteration, Alg. 2 reuses a single SAT instance. There is an additional difference between the implementation of CRR and the currently fastest implementation of Alg. 2. Alg. 2's fastest implementation uses the latest in-memory data structures with reference counters for storing the resolution derivation [9]–[11], while the CRR implementation of [6]–[8] uses the on-disk approach. Section V demonstrates that Alg. 2 empirically outperforms CRR for clause-level MUC extraction.

Our current implementation of resolution-based algorithms uses reference counters for efficiently removing unreferenced nodes in the in-memory resolution derivation. However, we noticed that using reference counters for this purpose is redundant, since the same effect can be achieved by removing unreferenced nodes during the standard interprocessing required for the clause deletion heuristic as follows. The solver stores the list L of all the clauses deleted by the clause deletion heuristic. Note that only clauses that appear in L should be considered for removal from the resolution derivation. When L becomes larger than some threshold, the algorithm removes from the resolution derivation all the clauses in L whose predecessors in the resolution derivation also appear in L. The exact implementation details are solver-specific. We have been working on implementing this idea and experimenting with it in the hope that it will result in further memory footprint reduction.

RRP [6]–[8], used to enhance CRR, is directly applicable to Alg. 2. The underlying idea is that a model for *SI* during any minimization loop iteration can only be found under such a partial assignment that falsifies every clause in some path in $Re(\pi, Clss(R_k))$ from a clause in $Clss(R_k)$ to $\Box$. The claim is correct, since finding a model for *SI* that satisfies every path in $Re(\pi, Clss(R_k))$ would mean that there is a satisfiable vertex cut in π, contradicting the assumption that π is a resolution refutation. For example, consider again Fig. 1 and suppose that R_1 is picked as the first removal candidate by the minimization loop. A model for *SI* can be found either when $b, c = 0; a = 1$ for the path C_1, C_6, C_7, C_8 or $c = 0; b, a = 1$ for the path C_2, C_6, C_7, C_8. RRP takes advantage of the described property during the minimization loop by guiding the decision heuristic and the backtracking engine of the SAT solver to falsify paths in $Re(\pi, Clss(R_k))$ in a systematic manner. We analyze the impact of RRP on Alg. 2's performance in Section V.

IV. Selector Variable-based Minimal Unsatisfiable Core Extraction

This section proposes a new selector variable-based algorithm for extracting a single high-level or clause-level MUC or, if a time-out occurs after the initial approximation stage, a non-minimal core. Our algorithm takes advantage of the ability of modern SAT solvers to solve the problem under assumptions and to return a set of relevant assumptions for proving the unsatisfiability [27], [28], explained in Section II.

Consider Alg. 3. It is composed of the approximation stage and the minimization loop, exactly like Algs. 1 and 2. First, the algorithm allocates a fresh selector variable s_i for each IC R_i and augments each clause in the clause projection of R_i with $\neg s_i$. At every stage of the algorithm every conflict clause in the cone of R_i will contain the literal $\neg s_i$. Hence, assigning some s_i to true means asserting the IC R_j, while assigning s_j to false means removing the associated IC by satisfying all the relevant clauses. Our algorithm takes advantage of

these properties to support the conditional and unconditional removal of the ICs from the instance, as well as their return to it, without explicitly maintaining a resolution derivation.

The approximation stage of the algorithm launches a SAT solver, providing it the input formula (updated with the selector variables) and the set of selector variables as the assumptions. Suppose that the solver returns: (1) the satisfiability status; (2) the updated incremental CNF instance *SI*; and (3) the set of relevant assumptions *rel_asm*. After invoking the solver, the set of ICs which corresponds to the selector variables in *rel_asm* is a (not necessarily minimal) high-level UC. The cones of the other ICs are removed from the instance. This is done by adding unit clauses to the instance, each of which contains the negation of a selector variable corresponding to one particular IC. One could physically remove the clauses, but this would require the expensive operation of going over the clauses explicitly and rebuilding the clause database. In our implementation, the SAT solver identifies and removes the satisfied clauses, at no additional cost, as part of the standard interprocessing required for the clause deletion heuristic.

The minimization loop of our algorithm maintains the sets of minimal core candidates and minimal core habitants like Alg. 2. Each iteration of the minimization loop removes a particular removal candidate from the set of minimal core candidates and launches the SAT solver on the formula, supplying it a set of assumptions that does not include the removal candidate. If the result is satisfiable, the removal candidate belongs to the MUC, otherwise it does not belong to it. In the latter case, based on the set of relevant assumptions returned by the SAT solver, the algorithm refines the set of minimal core candidates and removes the cones of unnecessary ICs. At the end, the set of minimal core habitants is returned as the high-level MUC.

Compare selector variable-based Alg. 3 to resolution-based Alg. 2. The selector variable-based approach saves the overhead of maintaining a resolution derivation and making additional passes over the clause database. Alg. 3 is also much simpler to implement. However, the selector variable-based approach has the drawback that adding new variables to the formula makes the conflict clauses larger and the solver slower. An empirical comparison of Alg. 3 and Alg. 2 is provided in Section V.

Algorithm 3 Selector variable-based high-level MUC extraction

Require: $\Psi = \{R_1, R_2, \ldots, R_m\} \wedge \Omega$ is unsatisfiable
1: For each $R_i \in \Psi$: $Sel(R_i) := \{\neg s_i \vee C | C \in Clss(R_i)\}$, where s_i is a new variable
2: $SI := (\wedge_{i:=1\ldots m} Sel(R_i)) \wedge Clss(\Omega)$
3: $\langle res, rel\_asm, SI\rangle := SATAsm(SI; \{s_1, s_2, \ldots s_m\})$
4: For each $j \notin rel\_asm$: Add a unit clause $\neg s_j$ to *SI*
5: $muc\_cands := \{i \mid s_i \in rel\_asm\}$
6: $muc := \{\}$
7: **while** *muc_cands* is non-empty **do**
8: $k :=$ a member of $muc\_cands \setminus muc$
9: $muc\_cands := muc\_cands \setminus \{k\}$
10: $\langle res, rel\_asm, SI\rangle := SATAsm(SI; \{s_i \mid i \in muc\_cands \cup muc\})$
11: **if** *res* = satisfiable **then**
12: $muc := muc \cup \{k\}$
13: **else**
14: $muc\_cands := \{i \mid s_i \in rel\_asm\}$
15: For each $j \notin rel\_asm$: Add a unit clause $\neg s_j$ to *SI*
16: **return** $\{R_i \mid i \in muc\}$

V. Experimental Results

In this section we empirically compare algorithms for clause-level MUC extraction and high-level MUC extraction.

A. Clause-Level Minimal Unsatisfiable Core Extraction

We used the same benchmarks that were used in [6]–[8]. The instances were taken from well-known unsatisfiable families from bounded model checking (*barrel*, *longmult*) [33] and microprocessor verification (*fvp-unsat.2.0, pipe_unsat_1.0*) [34]. The size of the instances ranged from 6,069 to 189,109 clauses, the average size being 49,986 clauses. Detailed information regarding these instances appears in [6]–[8]. All the algorithms were implemented in the Eureka SAT solver [35]. All experiments were carried out on a machine with 4Gb memory and two Intel Xeon CPU 3.06 processors.

Note that CRR with RRP is the best existing algorithm for extracting a clause-level MUC, given large difficult formal verification benchmarks. It was shown in [6]–[8] that CRR with RRP convincingly outperforms AMUSE [4] and MUP [36]. There exist a number of other approaches to UC extraction, such as those based on adaptive core search [3], Brouwer's fixed-point approximation algorithm [14], local search [15], a combination of local search and complete search [16], [19], a branch and bound algorithm [17], and genetic algorithms [18]. However, none of these approaches has been shown to scale well to large-size or even medium-size benchmarks. The instances considered in the experimental results sections of the papers mentioned above rarely exceed 10,000 clauses; in most cases the instances considered have at most a few thousand or even a few hundred clauses. This is not surprising, since these algorithms do not utilize the power of DPLL-based SAT solvers, currently the only approach that can solve large and difficult CNFs. Note that the problem of finding a MUC is DP-complete [37], hence in general extracting a MUC is at least as difficult as SAT solving; moreover, unless NP=co-NP, it is more difficult.

Consider Table I. Columns 1MR, 1MN, and 1DN (the names are explained in the caption of Table I) correspond to various implementations of Alg. 2. Columns PDR and PDN correspond to the CRR+RRP and CRR algorithms. Column PMN can be thought of as an in-memory implementation of CRR or as a modification of Alg. 2 that creates multiple SAT instances with pervasive clause reuse between them. Column SV corresponds to the selector variable-based approach of Alg. 3. Compare the best implementation of our Alg. 2 (1MN) to the best previous approach CRR with RRP (PDR). 1MN is clearly preferable, as it is faster overall and faster for thirteen out of sixteen instances. In addition, it manages to find the

TABLE I: Comparing algorithms for clause-level MUC extraction. The first column contains instance names (where, p/b/l stand for pipe/barrel/longmult). Each cell in the next seven columns contains the execution time in seconds on the top, the core size on the bottom-left, and the number of clauses whose status was not determined within the time-out of 2 hours on the bottom-right (the core is minimal iff 0 appears on the bottom-right of the corresponding cell). The first letter of the abbreviated algorithm names corresponds to the approach to incremental SAT solving: "P"/"1" stand for pervasive clause reuse/single SAT instance-based, respectively. The second letter corresponds to the data structures: "D"/"M" stand for on-disk/in-memory with reference counters, respectively. The third letter corresponds to RRP invocation: "R"/"N" stand for turning RRP on/turning RRP off, respectively. Bold times are the best times.

	Resolution-based												**SV**	
Inst	**1MR**		**1MN**		**PMN**		**1DN**		**PDR**		**PDN**			
4p	7200		**1417**		2326		7200		3791		4055		2021	
	25399	25341	18164	0	17928	0	18622	4050	18897	0	18609	0	17472	0
4p_1_ooo	7200		**1528**		2593		7200		2928		4579		4323	
	20445	20443	12213	0	12211	0	12444	8417	12246	0	12226	0	12887	0
4p_2_ooo	4718		**2383**		3428		7200		4566		7062		4999	
	14456	0	14438	0	14572	0	16360	11047	14553	0	14569	0	14560	0
4p_3_ooo	5053		**2560**		3694		7200		4465		6285		5357	
	15844	0	15850	0	15811	0	16141	7168	15892	0	15899	0	16177	0
4p_4_ooo	4768		**2432**		4706		7200		5865		7200		6354	
	17625	0	17558	0	17633	0	19215	13680	17872	0	17916	370	17793	0
3p_k	343		**167**		310		810		469		540		239	
	6784	0	6784	0	6787	0	6786	0	6783	0	6783	0	7074	0
4p_k	7200		**1426**		2243		7200		2938		3261		3097	
	21459	21459	17045	0	17039	0	17218	3403	17055	0	17075	0	18786	0
5p_k	7200		7200		7200		7200		7200		7200		7200	
	45406	45404	36423	8946	37479	22827	37523	36363	39336	19888	38800	28221	49134	47179
b5	286		68		71		869		115		128		**48**	
	2653	0	2653	0	2653	0	2653	0	2653	0	2653	0	2653	0
b6	1514		**348**		433		7200		436		552		402	
	4437	0	4437	0	4437	0	4498	659	4437	0	4437	0	4437	0
b7	1802		849		800		7200		1081		1108		**700**	
	6877	0	6877	0	6877	0	7324	2927	6877	0	6877	0	6877	0
b8	7200		4115		4479		7200		**4110**		4923		5758	
	10260	1390	10077	0	10076	0	12452	11382	10075	0	10075	0	10076	0
l4	23		14		14		25		**12**		**12**		78	
	972	0	972	0	972	0	972	0	972	0	972	0	972	0
l5	191		143		130		321		100		**97**		642	
	1518	0	1518	0	1518	0	1518	0	1518	0	1518	0	1520	0
l6	1121		**968**		1072		3129		1760		1615		5705	
	2189	0	2189	0	2190	0	2189	0	2189	0	2189	0	2194	0
l7	7200		**5099**		7200		7200		7200		7200		7200	
	2982	36	2982	0	2994	649	3203	1814	3454	1993	3071	973	3494	2895
Total	*63019*		***30717***		*40699*		*84354*		*47036*		*55817*		*54123*	
	199306	*114073*	*170180*	*8946*	*171177*	*23476*	*179118*	*100910*	*174809*	*21881*	*173669*	*29564*	*186106*	*50074*

minimal UC within the time-out for one more instance.

Comparing 1MN and SV clearly shows that our resolution-based approach is preferable to our selector variable-based approach for clause-level MUC extraction. The overhead of adding a variable per clause is too high.

The single SAT instance-based approach to incrementality results in better performance for the in-memory version of the resolution-based approach to UC extraction (compare 1MN to PMN). This is not surprising, as it takes advantage of the information gathered by the decision variable and clause deletion heuristics. However, it deteriorates the performance of the on-disk algorithm (compare 1DN to PDN). The problematic aspect of such a combination is that the single SAT instance-based approach stores the entire resolution derivation in a single file whose growing size does not allow it to be processed efficiently.

While RRP is helpful for CRR (compare PDR and PDN), it deteriorates the performance of Alg. 2 (compare 1MR and 1MN). We can also report that PMN outperforms PMR (PMR's results are not reported in the table due to space limitations). Hence RRP does not work well when the resolution derivation is stored in-memory. The reasons for this could be related to higher memory consumption, since RRP requires more memory due to additional bookkeeping. We plan to investigate and optimize the performance of RRP for Alg. 2 in the future.

B. High-Level Minimal Unsatisfiable Core Extraction

Compare the algorithms for high-level MUC extraction. As far as we know, this paper is the first to address the problem of extracting a single minimal (or small, if a time-out occurs) high-level UC. A much more expensive algorithm, called CAMUS, for the much more difficult problem of extracting all the high-level MUCs, is proposed in [13]. We provided a description of CAMUS in Section II-B. We used 49 instances generated from the abstraction stage of Intel's implementation of the proof-based abstraction refinement flow for model checking [24], [25]. All the instances are available from the author. The abstraction is in terms of latches. Each instance consists of two files: the standard CNF file in DIMACS format, and a file that maps latches to their clause projection. The goal was to minimize the number of latches in the core in order to create a more accurate abstraction. Consider Table II. Note that our instances have on average more than 1,850,000 clauses, while the largest benchmark has more than 5,000,000 clauses. Such instances are beyond the reach of modern algorithms

TABLE II: Statistics for instances used for testing high-level MUC extraction algorithms. The remainder fraction is the fraction of the remainder out of all the clauses.

	Clauses	Remainder Fraction	ICs Num.	Mean IC size
Min.	136878	0.955	584	3.89
Average	1853500	0.968	3367	17.27
Max.	5136873	0.977	4030	48.6

TABLE III: Comparing algorithms for high-level UC extraction.

	1MR	**1MN**	**PMN**	**1DN**	**PDR**	**PDN**	**SV**
UC time	**416**	**416**	**416**	766	766	766	588
UC size	28.9	28.9	28.9	28.9	28.9	28.9	27.7
MUC time	3843	3797	4731	9238	44699	44345	**3278**
MUC size	9.6	9.6	9.6	9.6	9.6	9.6	9.5

for clause-level MUC extraction. Note also that the fraction of the remainder among the clauses is very high, while the number of ICs (latches) is 3367 on average. We compared the same 7 algorithms that were compared for clause-level UC extraction. Consider Table III. The table summarizes the performance of our algorithms for both high-level non-minimal UC extraction (which corresponds to the approximation stage only) and high-level MUC extraction. The overall run-time and the average core sizes are displayed. Column PMN can be considered either as a modification of Alg. 2 that creates a new SAT instance for each minimization loop iteration or as a generalization of CRR for high-level minimal UC extraction.

While the best resolution-based approach 1MN is preferable to the selector variable-based approach for non-minimal UC extraction, the selector variable-based approach is preferable for minimal UC extraction. Hence, while adding selector variables does not pay off for non-minimal UC extraction, even when the number of ICs is relatively low, it turns out to be useful for the minimization loop. This result hints that the overhead of additional passes over the clauses is greater than the overhead of maintaining additional variables for high-level UC extraction (at least for our benchmarks). Also note that: (1) RRP is not helpful for the high-level UC extraction. The reason is, apparently, that $Re(\pi, Clss(R_k))$ is too large to be efficiently explored, since it has too many source clauses. (2) Not surprisingly, the in-memory data structure is clearly preferable to the on-disk one. (3) The single SAT instance-based approach to incrementality pays off even if the on-disk data structure is used (as opposed to the situation in clause-level UC extraction). The apparent reason is that considerably fewer operations of extracting the core using the file are required for high-level UC extraction.

VI. Conclusion and Future Work

We introduced two new algorithms (resolution-based and selector variable-based) for finding a minimal unsatisfiable core (or a small non-minimal core, if a time-out occurs). Our algorithms can be applied to either standard clause-level minimal unsatisfiable core extraction or high-level unsatisfiable core extraction, that is, extraction of a minimal unsatisfiable subset in terms of "interesting" propositional constraints provided by the user application.

We demonstrated that our resolution-based algorithm outperforms existing algorithms for standard clause-level minimal unsatisfiable core extraction on large, difficult industrial benchmarks. We also demonstrated the empirical usefulness and scalability of both our algorithms for high-level minimal unsatisfiable core extraction on huge benchmarks generated by Intel's proof-based abstraction refinement flow.

In addition, we provided a detailed comparison of various algorithms and heuristics for minimal unsatisfiable core extraction. An important conclusion is that while our resolution-based approach is clearly preferable to our selector variable-based approach for standard clause-level minimal unsatisfiable core extraction, the latter approach is faster for high-level minimal unsatisfiable core extraction. We found that the single SAT instance-based approach to incremental SAT solving results in better performance than the pervasive clause reuse approach. Furthermore, the in-memory data structure with reference counters for storing the resolution derivation is preferable to the on-disk data structure. Finally, RRP was not found to be helpful for the newly proposed algorithms.

We plan to investigate how to efficiently integrate RRP into our algorithms in the future. Furthermore, we have been working on improving the in-memory data structures for storing the resolution derivation by removing the redundant usage of reference counters (as described in Section III-C). In addition, we plan to study the impact of our algorithms within various applications, such as proof-based abstraction refinement and compositional FEV. Finally, we plan to study how to enhance our algorithms to extract more than one MUC.

Acknowledgments

The author would like to thank Amit Palti for supporting this work, Paul Inbar for editing the paper, and Vadim Ryvchin and Iddo Tzameret for providing useful comments.

References

[1] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable Boolean formula." in *Preliminary Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.

[2] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, 2003, pp. 886–891.

[3] R. Bruni, "Approximating minimal unsatisfiable subformulae by means of adaptive core search," *Discrete Applied Mathematics*, vol. 130, no. 2, pp. 85–100, 2003.

[4] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov, "AMUSE: a minimally-unsatisfiable subformula extractor," in *Proceedings of the 41th Design Automation Conference (DAC'04)*, 2004, pp. 518–523.

[5] R. Gershman, M. Koifman, and O. Strichman, "Deriving small unsatisfiable cores with dominators," in *CAV*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 109–122.

[6] N. Dershowitz, Z. Hanna, and A. Nadel, "A scalable algorithm for minimal unsatisfiable core extraction," *CoRR*, vol. abs/cs/0605085, 2006.

[7] ——, "A scalable algorithm for minimal unsatisfiable core extraction." in *SAT*, ser. Lecture Notes in Computer Science, A. Biere and C. P. Gomes, Eds., vol. 4121. Springer, 2006, pp. 36–41. [Online]. Available: http://dblp.uni-trier.de/db/conf/sat/sat2006.html#DershowitzHN06

[8] A. Nadel, "Understanding and improving a modern SAT solver," Ph.D. dissertation, Tel Aviv University, Tel Aviv, Israel, August 2009.

[9] A. Biere, "PicoSAT essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[10] O. Shacham and K. Yorav, "On-the-fly resolve trace minimization," in *DAC*. IEEE, 2007, pp. 594–599.

[11] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Efficient generation of unsatisfiability proofs and cores in SAT," in *LPAR*, ser. Lecture Notes in Computer Science, I. Cervesato, H. Veith, and A. Voronkov, Eds., vol. 5330. Springer, 2008, pp. 16–30.

[12] I. Lynce and J. P. M. Silva, "On computing minimum unsatisfiable cores," in *SAT*, 2004.

[13] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[14] H. van Maaren and S. Wieringa, "Finding guaranteed MUSes fast," in *SAT*, ser. Lecture Notes in Computer Science, H. K. Büning and X. Zhao, Eds., vol. 4996. Springer, 2008, pp. 291–304.

[15] É. Grégoire, B. Mazure, and C. Piette, "Using local search to find MSSes and MUSes," *European Journal of Operational Research*, vol. 199, no. 3, pp. 640–646, 2009.

[16] C. Piette, Y. Hamadi, and L. Sais, "Efficient combination of decision procedures for MUS computation," in *FroCos*, ser. Lecture Notes in Computer Science, S. Ghilardi and R. Sebastiani, Eds., vol. 5749. Springer, 2009, pp. 335–349.

[17] M. H. Liffiton, M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. Marques-Silva, and K. A. Sakallah, "A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas," *Constraints*, vol. 14, no. 4, pp. 415–442, 2009.

[18] J. Zhang, S. Li, and S. Shen, "Extracting minimum unsatisfiable cores with a greedy genetic algorithm," in *Australian Conference on Artificial Intelligence*, ser. Lecture Notes in Computer Science, A. Sattar and B. H. Kang, Eds., vol. 4304. Springer, 2006, pp. 847–856.

[19] J. Zhang, S. Shen, and S. Li, "Tracking unsatisfiable subformulas from reduced refutation proof," *JSW*, vol. 4, no. 1, pp. 42–49, 2009.

[20] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Refinement strategies for verification methods based on datapath abstraction," in *ASP-DAC*, F. Hirose, Ed. IEEE, 2006, pp. 19–24.

[21] S.-Y. Huang and K.-T. Cheng, *Formal equivalence checking and design debugging*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[22] Z. Khasidashvili, D. Kaiss, and D. Bustan, "A compositional theory for post-reboot observational equivalence checking of hardware," in *FMCAD*. IEEE, 2009, pp. 136–143.

[23] 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.

[24] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples." in *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, 2003, pp. 2–17.

[25] A. Gupta, M. K. Ganai, Z. Yang, and P. Ashar, "Iterative abstraction using SAT-based BMC with proof analysis," in *ICCAD*. IEEE Computer Society / ACM, 2003, pp. 416–423.

[26] J. P. M. Silva and K. A. Sakallah, "Robust search algorithms for test pattern generation," in *FTCS*, 1997, pp. 152–161.

[27] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, 2003.

[28] ——, "The MiniSat page." [Online]. Available: http://www.minisat.se/

[29] A. Biere, "Booleforce," http://fmv.jku.at/booleforce. [Online]. Available: http://fmv.jku.at/booleforce

[30] G. S. Tseitin, "On the complexity of derivations in the propositional calculus," *Studies in Mathematics and Mathematical Logic*, vol. Part II, pp. 115–125, 1968.

[31] N. Dershowitz, Z. Hanna, and A. Nadel, "A clause-based heuristic for SAT solvers." in *SAT*, ser. Lecture Notes in Computer Science, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, 2005, pp. 46–60. [Online]. Available: http://dblp.uni-trier.de/db/conf/sat/sat2005.html#DershowitzHN05

[32] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *DAC*. ACM, 2001, pp. 530–535.

[33] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, 1999, pp. 193–207. [Online]. Available: http://www.inf.ethz.ch/personal/biere/papers/BiereCimattiClarkeZhu-TACAS99.ps.gz

[34] M. Velev and R. Bryant, "Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," in *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001, pp. 226–231. [Online]. Available: http://www.ece.cmu.edu/~mvelev/DAC01.ps.gz

[35] A. Nadel, M. Gordon, A. Palti, and Z. Hanna, "Eureka-2006 SAT solver," http://fmv.jku.at/sat-race-2006/descriptions/4-Eureka.pdf. [Online]. Available: http://fmv.jku.at/sat-race-2006/descriptions/4-Eureka.pdf

[36] J. Huang, "MUP: A minimal unsatisfiability prover," in *Proceedings of the Tenth Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, 2005, pp. 432–437.

[37] C. H. Papadimitriou and M. Yannakakis, "The complexity of facets (and some facets of complexity)," in *Proceedings of the Fourteenth Annual ACM Symposium on the Theory of Computing (STOC'82)*, 1982, pp. 255–260.