Refining Unsatisfiable Cores in Incremental SAT Solving

Norbert Manthey
Technische Universität Dresden
Knowledge Representation and Reasoning Group
Dresden, Germany
Email: norbert.manthey@tu-dresden.de

***Abstract*—Incremental SAT solving is used in many applications in the area of electronic design automation. The extraction of unsatisfiable subformulas of a propositional logic formula, as used in verification tools and MaxSAT algorithms, is an important feature. In this work we propose a simple refinement strategy for extracting unsatisfiable subformulas, which does not produce minimal subformulas, but can be computed easily. When implementing the proposed refinement, the circuit verification tool IC3 solves three more circuits from HWMCC 2014, improving its run time between 11 percent to 21 percent. Furthermore, the MaxSAT solver Open-WBO solves 32 more weighted partial industrial MaxSAT problems from the evaluations 2013 and 2014.**

I. Introduction

There exist many applications in electronic design automation (EDA) that are built on top of satisfiability testing. SAT solvers are used a lot in EDA to solve the major work load. These applications include bounded [1] and unbounded model checking [2]–[5], as well as combinatorial and sequential equivalence checking [6]–[10] or design debugging [11], [12], where the latter is based on the optimization variant of SAT solving, namely the MaxSAT problem. Most of the time, a SAT solver is not used to solve a single formula. Instead, related formulas are solved iteratively, and after each call the problem is refined based on the SAT solver's answer. The incremental interface of modern SAT solvers is motivated by [13]. The interface supports extracting a model for the current formula, as well as solving a formula under assumptions and it allows to extract an unsatisfiable subformula.

The latter feature is based on extracting an inconsistent subset of assumptions, and is actively used in abstraction-based algorithms to refine the abstraction, for example in [3]–[5]. Modern MaxSAT algorithms, which are capable of solving industrial EDA problems [14], also exploit the extraction of unsatisfiable subformulas based on assumptions [15]. The minimization of these unsatisfiable subsets is believed to improve the abstraction, as well as the verification time [16]. Hence, there have been attempts to apply the extraction of minimal unsatisfiable subformulas (MUS) to formal verification tasks, for example in [17]–[20]. However, the calculation of minimal subformulas is known to be a hard problem, as this problem is part of the problem class DP [21].

This work presents another, very light-weight, method to *refine* the extracted unsatisfiable core. Differently to the calculation of an MUS, the proposed procedure is rather simple. Assume, the set $C = \{a_1, \ldots, a_n\}$ of inconsistent assumption literals of a formula has been returned by an incremental SAT solver. Then, the solver represents this set as the clause $D = (\overline{a_1} \vee \ldots \vee \overline{a_n})$. We propose to run a limited *vivification* [22] on the clause D to obtain a shorter clause D', which then represents a smaller set of inconsistent assumptions C'. As the resulting clause D' is a subset of the clause D, $C' \subseteq C$ also holds.

We implemented a small patch to the SAT solvers MiniSAT and Glucose 3.0, because they are used in the open MaxSAT solver Open-WBO [23], as well as in the reference implementation of IC3 [4]. With the refined unsatisfiable cores, both tools can solve more problems of the MaxSAT evaluation 2014 [24] and the HWMCC'14 [25]. Furthermore, for IC3 the time to solve 33 percent of the benchmark is improved by 21 percent.

A. Related Work

Incremental SAT solving received much attention recently. Formula simplification techniques have been investigated, and methods to eventually undo simplification steps have been proposed in [26], [27].

Furthermore, handling assumption literals has been investigated in [28]. Instead of using these literals as decision literals, as proposed in [13], the assumption literals are added to the formula as unit clauses, such that simplification techniques become more powerful. To keep the benefit of incremental SAT solving, i.e. keeping learned clauses over multiple solver calls, the effects of learned unit clauses are traced and undone if necessary.

In [29], the solver is adapted to store dependencies of assumption literals, such that learned clauses that are created together with assumption literals are shorter, and hence unit propagation can be executed faster. Based on the dependencies, the authors of [29] furthermore propose a minimization technique for learned clauses.

This work is supported by the DFG grant HO 1294/11-1

Differently, in [30], Audemard et al. treat the assumption literals specially during unit propagation and conflict analysis. Additionally, they propose to ignore assumption literals during the calculation of metrics to decide whether learnt clauses should be kept, similarly to the work by Belov et al. [31].

Our presented approach is orthogonal to the related work, as we suggest to reduce the unsatisfiable core. All other components of the solver remain untouched, such that the ideas of related work could also be incorporated.

B. Structure

We first give the preliminaries in Section II. Next, in Section III we present *reverse core refinement*, the main method of this work. Afterwards, we apply the idea to an implementation of IC3 and a MaxSAT solver, and discuss the empirical results in Section IV. We conclude the work in Section V.

II. Preliminaries

We assume the reader to be familiar with propositional logic, and present only the notation that is used throughout the paper. We assume a fixed infinite set V of Boolean *variables*. A *literal* is a variable v (*positive literal*) or a negated variable $\neg v$ (*negative literal*). The *complement* $\overline{x}$ of a positive (negative, resp.) literal x is the negative (positive, resp.) literal with the same variable as x. The complement $\overline{S}$ of a sequence S is the sequence $\overline{S} = (\overline{x} \mid x \in S)$. *Formulas* are conjunctions of clauses, and *clauses* are disjunctions of literals. Clauses do not contain duplicate literals. However, a clause is not a set of literals, as the order of the literals inside the clause matters in the solver implementation. For convenience, we will still use set operations for clauses and formulas to explain algorithms. The empty clause is denoted by $\bot$.

A sequence of literals M is *consistent*, if whenever $x \in M$, then $\overline{x} \notin M$. For simplicity, we view consistent sequences M as sets throughout this paper. An *interpretation* is a consistent set of literals I. The *reduct* $F|_I$ of a formula F with respect to I is the multiset $F|_I := \{C \setminus \overline{I} \mid C \in F, C \cap I = \emptyset\}$. An interpretation I *satisfies* a formula F, if $F|_I = \emptyset$. A formula is *satisfiable* if there is an interpretation that satisfies it. The SAT problem consists in deciding whether a formula is satisfiable.

Let C and D be clauses, with $x \in C$ and $\overline{x} \in D$; then, the *resolvent of the clauses C and D upon the literal x* is the clause $C \otimes_x D := (C \setminus \{x\}) \cup (D \setminus \{\overline{x}\})$.

A. SAT Solving

A major operation in modern SAT solvers is *unit propagation*, which is based on the fact that a unit clause $C = \{x\}$ can only be satisfied by interpretations that contain x. For this assignment of x the clause C is called the *reason*. Given a formula F and a literal x, then let $F \vdash_{\text{UP}} x$ denote that the literal x can be derived from the formula F via the repeated application of the unit rule.

Furthermore, SAT solvers rely on search. Literal assignments that are done due to search will be denoted with a dot on top, e.g. $\dot{x}$. Search literals do not have a reason. The *decision level* of a literal is the number of decision literals that have been added to the interpretation before this literal (including the literal itself).

A *conflict clause* C is a clause that is falsified by the current interpretation M, i.e. $C|_M = \emptyset$. By *conflict analysis* [32], a *learned clause* D is obtained from C, by applying resolution with the reason clauses of the literals of C. With D, *backjumping* is performed and then search can be continued.

Initially, $D = C$. Then, a literal $x \in D$ with the reason clause $R_{\overline{x}}$ for $\overline{x}$ is selected, and another clause D is obtained by $D := D \otimes R_x$. The literal $x \in D$ is selected as the literal with the highest index in M [13]. There exist two criteria to stop the routine. Modern SAT solvers learn a *first UIP clause* [33], which is reached when D contains only a single literal from the current decision level for the first time. Furthermore, *decision clauses* [34] can be learned, which are obtained by continuing resolution until no literal of D has a reason clause.

B. Incremental SAT Solving

In *incremental SAT solving* [13], [35], a solver instances is inizialized with a formula F and is used multiple times. Instead, based on the result for the current formula, more clauses can be added. Furthermore, search *under assumptions* is possible. Let A be a sequence of literals $(a_1 \dots a_n)$. Then, solving under the assumptions A is to evaluate the truth value of $F \wedge a_1 \wedge \dots \wedge a_n$, for short $F \wedge A$. This process is realized by using the literals of A as the first search decisions, before the usual heuristic driven search is performed. The decision literals are picked in exactly the order in which they appear in A.

If $F \wedge A \equiv \top$, then a model $M \models F \wedge A$ is returned. Otherwise, a learned clause D, which is created to be a decision clause, with $D \subseteq \overline{A}$, is generated, where $F \models D$ as we used only resolution to derive D. Hence, the conjunction of literals in $\overline{D}$ is inconsistent with F, which can be shown by unit propagation, i.e. $(F \wedge \overline{D}) \vdash_{\text{UP}} \bot$. Together with F, the literals $\overline{D}$ represent an *unsatisfiable core*. Depending on the application, the result is used to control the next step of the algorithm.

Example 1. *Consider the formula*

$$F = (\overline{a} \vee c) \wedge (\overline{b} \vee d) \wedge (\overline{b} \vee \overline{d}) \wedge (\overline{c} \vee d)$$

and let the assumptions be $A = (ab)$. As there are no unit clauses, search is triggered with the first decision literal a. By unit propagation c is implied with the reason $(\overline{a} \vee c)$, and d is implied with $(\overline{c} \vee d)$. Next, $\overline{b}$ is implied with the reason $(\overline{b} \vee \overline{d})$. Finally, the algorithm tries to assign b to $\top$, as b is an assumption. In this

step, the algorithm notices that b is assigned to $\bot$ already, and starts creating the learned clause D by starting with b's reason $D = (\overline{b} \vee \overline{d})$. By resolving D with $(\overline{c} \vee d)$ and afterwards resolving with $(\overline{a} \vee c)$, the final decision clause $D = (\overline{b} \vee \overline{a})$ is obtained. Note, that $F \wedge \overline{D}$ finds a conflict by unit propagation, as $F \wedge (b \wedge a) \vdash_{\text{UP}} \{d, \overline{d}\}$.

III. Reverse Core Refinement

The unsatisfiable core in Example 1 is $\overline{D} = (b \wedge a)$. This core contains exactly the same literals as the assumptions, they are just reversed. The conjecture of this paper is that an unsatisfiable core that is generated with the above routine contains redundant literals. A way to remove redundant literals is to apply minimization by checking whether for a literal $x \in D$ we can still find a conflict via unit propagation after removing the literal, i.e. $F \wedge (\overline{D \setminus \{x\}}) \vdash_{\text{UP}} \bot$. Then, x is redundant, and $\overline{D \setminus \{x\}}$ is a smaller unsatisfiable core. The above procedure is known as *vivification* [22], and is quadratic when being executed until reaching a fix point. Furthermore, no good selection heuristic for the literals x are known.

We propose to *reverse* the literals in $\overline{D}$, and use them as assumptions again, and to *not* execute vivification until a fix point is reached. With the new assumptions, we re-run the solver again. This way, literals, which might be present in $\overline{D}$ because they have been used as first decision literals, can now be dropped, as these literals are used last now.

Example 2. *Consider the formula of Example 1 again. Now, let $A = (ba)$ be the refined assumptions. The new unsatisfiable core is $\overline{D} = b$, because after the search decision b we implied the literal d with $(\overline{b} \vee d)$ and $\overline{d}$ with $(\overline{b} \vee \overline{d})$. However, before the core is generated, the algorithm first learns the unit clause $\overline{b}$ from the conflict clause $(\overline{b} \vee \overline{d})$, according to the algorithm implemented in* MiniSAT 2.2 *and applies backtracking. Next, using b as search decision fails, and hence, the core $\overline{D} = b$ is returned.*

The steps in Example 2 show that a smaller core can be obtained. Furthermore, the algorithm might find further conflicts and learn further clauses before obtaining the final core, as Example 3 illustrates.

Example 3. *Consider the following formula*

$$(\overline{e} \vee a) \wedge (g \vee \overline{a} \vee \overline{e}) \wedge (\overline{g} \vee \overline{b}) \wedge (\overline{g} \vee f \vee \overline{d}) \wedge (\overline{c} \vee e) \wedge (\overline{g} \vee \overline{f}) \wedge (g \vee \overline{f})$$

with the assumptions $A = (abcd)$. Assuming literal a implies no literals. Next, after using b as decision, $\overline{h}$ and $\overline{f}$ are implied. With decision c, the literals e, g and $\overline{d}$ are implied. Hence, using d as next decision fails, and the unsatisfiable core $(dcba)$ is found without any intermediate conflicts.

When using this core as refined assumptions, i.e. $A' = (dcba)$, then two conflicts are necessary to obtain the refined core. Assuming d and c implies e with $(\overline{c} \vee e)$, a with $(\overline{e} \vee a)$, g with $(g \vee \overline{a} \vee \overline{e})$ and $\overline{f}$ with $(\overline{g} \vee \overline{f})$. Then, from the interpretation $M = (\dot{d}\dot{c}\dot{e}a\dot{g}\overline{f})$ the conflict clause $(\overline{g} \vee f \vee \overline{d})$ is found, from which the clause $D_1 = (\overline{g} \vee \overline{d})$ is learned.

With D_1 the interpretation M is changed to $M = (\dot{d}\dot{\overline{g}})$, which does not allow further propagation. Next, c is assumed again, resulting in e with $(\overline{c} \vee e)$ and a with $(\overline{e} \vee a)$. Then, with the interpretation $M = (\dot{d}\dot{\overline{g}}\dot{c}ea)$ the conflict clause $(g \vee \overline{a} \vee \overline{e})$ is found, from which the clause $D_2 = (\overline{e} \vee g)$ is learned.

After backtracking, the interpretation is changed to $M = (\dot{d}\dot{\overline{g}})$. With the clause D_2, $\overline{e}$ is implied, and $\overline{c}$ is implied with $(\overline{c} \vee e)$. Next, the attempt to assume c fails, and the resulting refined core is $(c \wedge d)$.

As the refinement procedure can be seen as overhead of the actual routine and because the effectiveness of the procedure is not known in advance, the number of additionally learned clauses might be limited. If this limit is reached, then the original unsatisfiable core is used.

A. Early Refined Cores

An alternative to improve the above routine is to stop as soon as a conflict with respect to the currently assigned assumptions is found. Instead of creating a decision clause if assigning an assumption fails, conflict analysis can also be applied to an intermediate conflict. This way, additional conflicts, and the potential overhead, can be avoided.

Example 4. *Consider the formula from Example 3 one more time. When using the refined assumptions, i.e. $A' = (dcba)$, then the interpretation $M = (\dot{d}\dot{c}\dot{e}a\dot{g}\overline{f})$ is created, and the conflict clause $(\overline{g} \vee f \vee \overline{d})$ is found. Learning a decision clause from this conflict results in the unsatisfiable core $(c \wedge d)$.*

In Example 4 the same unsatisfiable core is obtained as in Example 3. Furthermore, no additional conflicts have been encountered, and no additional learned clauses had to be created.

IV. Experimental Results

We implemented the above modifications into the commonly used SAT solver MiniSAT, which is used as back end in several EDA tools. As a first tool, we chose the

Table I
NUMBER OF SOLVED CIRCUITS BY THE CORE-REFINING VARIANTS OF IC3, THE TIME IN SECONDS TO SOLVE 33 PERCENT OF THE BENCHMARK (Q-33), AND THE OBTAINED DEEP BOUND SCORE DBS.

Limit	IC3	1	1*	10	1000	100000	∞
Solved	84	**87**	**87**	77	77	85	77
q-33	712.9	640.2	670.0	1470	1598.1	**520.1**	1580.6
DBS	88.6	88.76	88.77	88.21	88.22	**88.84**	88.19

reference implementation of IC3[1] by Bradley, in which we determined the robustness of unsatisfiable core refinement. Next, we picked the MaxSAT solver OPEN-WBO, and applied the best modifications as well.

All experiments have been performed on a cluster with Intel Xeon CPUs, a timeout of 60 minutes, and each process was allowed to use up to 6 GB main memory.

A. Solving Hardware Verification with IC3

For hardware verification experiments we used the 206 publicly available circuits from the HWMCC 2014 [25]. We then modified the implementation of MINISAT and embedded it into the implementation of IC3. The results are summarized in Table I. Note that the implementation of IC3 uses two types of SAT solvers. A new solver is created for each frame during evaluating the circuit, and one solver is used for lifting. All these solvers are used multiple times while executing the IC3 algorithm.

The table presents the number of solved circuits, the time for solving the 33 percent quantile, and the deep bound score according to the HWMCC. The median is not presented, as none of the variants solves half the circuits. The table presents several variants of IC3: the original implementation of MINISAT, unlimited core refinement, and different limits for conflict refinement. Additionally, in column 1* we reordered the conflict clause again. In the above examples, the order of the literals in the unsatisfiable core is always reversed with respect to the original core. Variant 1* reverses the unsatisfiable core once more, to obtain the original order again. Apparently, the implementation of IC3 processes the unsatisfiable cores of the solver independently of their order.

The results in the table show that three more circuits can be solved if conflict refinement is used. Using a single conflict as limit results in the most effective variant of the tool, independently of the order of the unsatisfiable core. Furthermore, the run time for the 33 percent quantile is improved by 11 percent compared to the original implementation. When increasing the conflict limit for refinement, the performance, as well as the run time to solve most of the problems, decreases significantly. With a limit of 10 or 1000 conflicts, or with no limit at all, only 77 circuits can be solved. However, choosing a very large limit seems to be a good trade-off: when using at most 100000 conflicts for each conflict refinement the solving time for most problems decreases to 520.1 seconds for the 33 percent quantile, and the procedure is as robust as the original variant. With the high limit, the run time of IC3 is improved by 21 percent.

The result can be explained as follows: when using no limit, then there seem to be rare cases that require a lot of conflicts. To avoid these cases, a cut off is necessary. However, unsatisfiable core refinement pays off only if either the overhead is very small, or if a sufficient number of conflicts is allowed. This way, only the variants with a very low limit or a very large limit are as robust as the original version. Using no limit, or a medium limit results in a slowdown of the procedure.

Similarly, the score that evaluates the routine with respect to unrolling of unsuccessfully solved formulas supports the above findings. We picked the 70 publicly available circuits that have been used in the deep bound track of HWMCC 2014 and calculated the deep bound score (DBS) for each variant accordingly:

$$\mathrm{DBS} = \sum_{i=1}^{70} \left(1 - \frac{1}{2 + \mathrm{bound}_i}\right),$$

where bound_i is the last bound where the tool could show that no bug exists for the circuit within the given resource limits (allowing 100 as the maximum value). The last line of Table I shows that the fast variants with limit 1 and 100000 reach a higher score than the other limits.

As discussed above, the search for an unsatisfiable core could also be aborted earlier. We tested three additional variants: (i) always abort early without refinement, (ii) abort early only during refinement, and (iii) abort early and abort early during refinement. Note that no conflict limit is necessary when using early abort during refinement, as early abort will stop on the first conflict. The obtained results are all worse than the original variant: the number of solved circuits is 75, 78, 76, the q-33 time is always above 1500 seconds, and the DBS is 87.83, 88.17 and 88.07, respectively.

B. Solving Industrial MaxSAT Problems with Open-WBO

The MaxSAT solver OPEN-WBO [23] is an open solver that supports multiple SAT solvers. For simplicity, we picked the implementation of MINISAT 2.2, and applied the above modifications. Furthermore, we selected the solver GLUCOSE 3.0, which is also included in the package of OPEN-WBO, and which has been optimized for incremental SAT solving [30]. However, as GLUCOSE 3.0 uses a dynamic restart schedule, we allow the search for a refined unsatisfiable core only until the next restart that would be triggered by the solver.

For the evaluation we used all industrial problems of the MaxSAT evaluation 2013 and 2014. These problems can further be partitioned into partial MaxSAT problems (PMS), where all soft clauses have the same weight and

[1]The original source code is available at https://github.com/arbrad/IC3ref, accessed 10th August 2015.

Table II
NUMBER OF SOLVED MAXSAT PROBLEMS AND THE CORRESPONDING MEDIAN TIME BY THE CORE-REFINING VARIANTS OF OPEN-WBO, SEPARATED INTO THE PROBLEM TYPES PMS AND WPMS

Variant	MS2.2	1	100000	∞	REA	GL3	RST
Solved Industrial MaxSAT Problems							
ALL	1715	1719	1720	**1725**	1709	1726	**1747**
PMS	**1041**	**1041**	1026	1029	1026	**1046**	1035
WPMS	674	678	694	**696**	683	680	**712**
WMSU3	398	398	**399**	397	395	**400**	**400**
WBO	276	280	295	**299**	288	280	**312**
Median Run Time							
ALL	**6.7**	6.7	11.4	9.6	11.7	**6.3**	6.4
PMS	13.7	**13.6**	23.0	22.1	23.6	**10.9**	12.7
WPMS	**2.0**	2.1	4.2	3.9	4.3	**2.3**	2.7

there exist further hard clauses, and weighted partial MaxSAT (WPMS), where the weights of the soft clauses are allowed to vary. In the benchmark there are 2001 industrial problems, from which 1195 are PMS, and the remaining 806 are WPMS problems. We evaluated the modification for these categories separately.

Table II shows the results for the modification. The first columns are for the SAT solver MINISAT 2.2 (MS2.2) with the variants to refine the unsatisfiable core with different limits, and with the variant to abort early combined with refinement (REA). The last two columns show the results for OPEN-WBO with GLUCOSE 3.0 (GL3), and with core refinement until the next restart (RST).

For the discussion of the results a detail of OPEN-WBO is important: the MaxSAT algorithm is picked depending on the problem instance, and whenever possible an incremental variant is selected. For PMS problems, always the algorithm MSU3 [36] is chosen. Weighted problems are solved either with WMSU3 [36], or with the WBO [37] algorithm. In the algorithm WBO, the used SAT solver is rebuilt in each iteration of the algorithm. Hence, learned clauses from previous calls are lost. Differently, in MSU3, one solver is kept and used for all calls with incremental updates, similarly to the used implementation of IC3. Likewise, the incremental variant of WMSU3 also keeps one solver object for the whole search and performs optimization updates to the formula incrementally.

From the distribution of the problems in the industrial category, one can easily see, that by using core refinement, the performance of OPEN-WBO can be improved. With MINISAT, 1725 problems can be solved, instead of 1715, while using core refinement almost always increases the median run time of the tool, except when only a single conflict is allowed during refinement. With GLUCOSE, the number of solved problems increases from 1726 to 1747. A detailed analysis reveals, that the increase in performance comes from the weighted problems, more precisely from the WBO algorithm, whose performance jumps from 276 to 299 and 280 to 312 solved problems for MINISAT and GLUCOSE, respectively. The reduced unsatisfiable cores seem to help the tool, and the additionally learned clauses during conflict refinement cannot influence future search steps, because the SAT solver is re-initialized before the next call. The numbers for WMSU3 remain almost constant. For plain partial MaxSAT problems, the performance of the tool even decreases: up to 15 less problems can be solved. We assume that the additionally learned clauses, as well as the additionally search steps prevent the tool from following the search path of the original variant. Nevertheless, except for the early abort modification all variants of core refinement are more robust than the original variant of OPEN-WBO for the whole industrial benchmark. The modification is only useful when being used inside the WBO algorithm. Still, this modification is outperformed by most limited core refinements. All other tested variants of early refinement aborts resulted in worse results.

V. CONCLUSION

Incremental SAT solving is widely applied in EDA applications to solve verification problems or to debug designs. Modern algorithms for both problems, i.e. the IC3 algorithm, as well as unsatisfiable core based MaxSAT algorithms, are heavily based on extracting unsatisfiable subformulas from a propositional formula. This paper presents reverse core refinement, which allows to reduce these subformulas with little overhead. The effectiveness of the approach has been demonstrated for hardware verification problems as well as industrial MaxSAT problems. Both the robustness and the run time of the modified tools have been improved significantly.

Neither modern verification tool implementations nor modern MaxSAT solvers use the full features that are available in modern SAT solvers, most importantly *inprocessing*. Currently, SAT solvers are used as a black box with a simple interface to incrementally solve specified problems. As future work, the effects of formula simplifications during the verification or debugging process should be investigated, especially because the complexity of verification or MaxSAT is higher than the complexity of solving the satisfiability problem. Most simplification techniques are developed for solving the SAT problem and the complexity of each technique is lower than NP to not solve an unnecessarily hard problem. However, for verification and debugging, also the more complex simplification techniques should be considered, as they might improve the overall performance even when a higher simplification time is required.

REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '99. Springer, 1999, pp. 193–207.

[2] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, ser. LNCS, W. A. H. Jr. and S. D. Johnson, Eds., vol. 1954. Springer, 2000, pp. 108–125.

[3] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

[4] A. R. Bradley, "SAT-based model checking without unrolling," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'11. Springer, 2011, pp. 70–87.

[5] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '11. FMCAD Inc, 2011, pp. 125–134.

[6] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[7] E. I. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *DATE*, 2001, pp. 114–121.

[8] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *International Conference on Computer-Aided Design*, S. Hassoun, Ed. ACM, 2006, pp. 836–843.

[9] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *ICCD*, 2006.

[10] D. Kaiss, M. Skaba, Z. Hanna, and Z. Khasidashvili, "Industrial strength SAT-based alignability algorithm for hardware equivalence verification," in *FMCAD*, 2007, pp. 20–26.

[11] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 24, no. 10, pp. 1606–1621, Nov. 2006.

[12] Y. Chen, S. Safarpour, J. P. Marques-Silva, and A. G. Veneris, "Automated design debugging with maximum satisfiability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 11, pp. 1804–1817, 2010.

[13] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT 2003.*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[14] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, and K. A. Sakallah, "Improved design debugging using maximum satisfiability," in *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007*, 2007, pp. 13–19.

[15] J. Marques-Silva and J. Planes, "Algorithms for maximum satisfiability using unsatisfiable cores," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. ACM, 2008, pp. 408–413.

[16] A. Belov, H. Chen, A. Mishchenko, and J. Marques-Silva, "Core minimization in SAT-based abstraction," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. EDA Consortium, 2013, pp. 1411–1416.

[17] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification, 15th International Conference, CAV 2003*, ser. LNCS, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 1–13.

[18] A. Gupta, M. Ganai, Z. Yang, and P. Ashar, "Iterative abstraction using SAT-based BMC with proof analysis," in *Proceedings of the 2003 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '03. IEEE Computer Society, 2003, pp. 416–423.

[19] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla, "GLA: Gate-level abstraction revisited," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. EDA Consortium, 2013, pp. 1399–1404.

[20] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *FMCAD 2010*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 221–229.

[21] C. H. Papadimitriou and D. Wolfe, "The complexity of facets resolved," *J. Comput. Syst. Sci.*, vol. 37, no. 1, pp. 2–13, Aug. 1988.

[22] C. Piette, Y. Hamadi, and L. Saïs, "Vivifying propositional clausal formulae," in *Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence*. IOS Press, 2008, pp. 525–529.

[23] R. Martins, V. Manquinho, and I. Lynce, "Open-WBO: A modular MaxSAT solver,," in *Theory and Applications of Satisfiability Testing – SAT 2014*, ser. LNCS, C. Sinz and U. Egly, Eds. Springer, 2014, vol. 8561, pp. 438–445.

[24] "MaxSAT evaluations," http://www.maxsat.udl.cat/, Oct. 2014.

[25] "Hardware model checking competition," http://fmv.jku.at/hwmcc14/, Oct. 2014.

[26] A. Nadel, V. Ryvchin, and O. Strichman, "Ultimately incremental SAT," in *SAT 2014*, ser. LNCS, C. Sinz and U. Egly, Eds., vol. 8561. Springer, 2014, pp. 206–218.

[27] ——, "Preprocessing in incremental SAT," in *SAT 2012*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 256–269.

[28] A. Nadel and V. Ryvchin, "Efficient SAT solving under assumptions," in *SAT 2012*, ser. LNCS, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 242–255.

[29] J. Lagniez and A. Biere, "Factoring out assumptions to speed up MUS extraction," in *SAT 2013*, ser. LNCS, M. Järvisalo and A. V. Gelder, Eds., vol. 7962. Springer, 2013, pp. 276–292.

[30] G. Audemard, J. Lagniez, and L. Simon, "Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction," in *Theory and Applications of Satisfiability Testing - SAT 2013*, ser. LNCS, M. Järvisalo and A. V. Gelder, Eds., vol. 7962. Springer, 2013, pp. 309–317.

[31] A. Belov, N. Manthey, and J. P. Marques-Silva, "Parallel MUS extraction," in *SAT 2013*, ser. LNCS, M. Järvisalo and A. V. Gelder, Eds., vol. 7962. Springer, 2013, pp. 133–149.

[32] J. P. Marques-Silva and K. A. Sakallah, "GRASP – a new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM international conference on computer-aided design*, ser. ICCAD '96. IEEE Computer Society, Washington, DC, USA, 1996, pp. 220–227.

[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th annual Design Automation Conference*, ser. DAC '01. ACM, 2001, pp. 530–535.

[34] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '01. IEEE Press, 2001, pp. 279–285.

[35] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.

[36] J. Marques-Silva and J. Planes, "On using unsatisfiability for solving maximum satisfiability," *CoRR*, vol. abs/0712.1097, 2007.

[37] V. Manquinho, J. Marques-Silva, and J. Planes, "Algorithms for weighted boolean optimization," in *Theory and Applications of Satisfiability Testing - SAT 2009*, ser. LNCS, O. Kullmann, Ed. Springer, 2009, vol. 5584, pp. 495–508.

[38] M. Järvisalo and A. V. Gelder, Eds., *Theory and Applications of Satisfiability Testing - SAT 2013*, ser. LNCS, vol. 7962. Springer, 2013.

[39] A. Cimatti and R. Sebastiani, Eds., *Theory and Applications of Satisfiability Testing - SAT 2012*, ser. LNCS, vol. 7317. Springer, 2012.