Formal Methods in Computer-Aided Design 2021

Single Clause Assumption without Activation Literals to Speed-up IC3

Nils Froleyks
nils.froleyks@jku.at
Johannes Kepler University, Linz, Autstria

Armin Biere
biere@cs.uni-freiburg.de
Albert–Ludwigs–University, Freiburg, Germany

***Abstract*—We extend the well-established assumption-based interface of incremental SAT solvers to clauses, allowing the addition of a temporary clause that has the same lifespan as literal assumptions. Our approach is efficient and easy to implement in modern CDCL-based solvers. Compared to previous approaches, it does not come with any memory overhead and does not slow down the solver due to disabled activation literals, thus eliminating the need for algorithms like IC3 to restart the SAT solver. All clauses learned under literal and clause assumptions are safe to keep and not implicitly invalidated for containing an activation literal. These changes increase the quality of learned clauses, resulting in better generalization for IC3. We implement the extension in the SAT solver CaDiCaL and evaluate it with the IC3 implementation in the model checker ABC. Our experiments on the benchmarks from a recent hardware model checking competition show a speedup for the average SAT call and a reduction in number of calls per verification instance, resulting in a substantial improvement in model checking time.**

Introduction

Modern SAT solving is based on Conflict-Driven Clause Learning (CDCL) [1]. Many applications require solving a sequence of related SAT problems incrementally [2], [3], making use of inprocessing techniques [4], [5], [6] that make modern SAT solvers so efficient. Among those applications is the symbolic model checking algorithm IC3. In contrast to other incremental SAT-based techniques, such as bounded model checking (BMC) [7], [8] and k-induction [9], [10], IC3 does not rely on unrolling the transition function. As a result the SAT queries that IC3 poses are significantly smaller and faster to solve. However, the number of queries that IC3 makes over the course of one model checking procedure is significantly higher. We illustrate the kind of queries that IC3 makes in the following example.

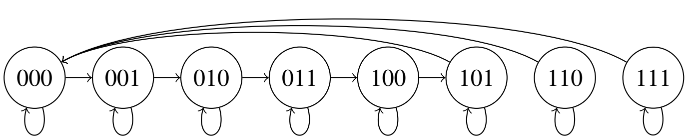

Fig. 1. Transition system

Consider the transition system of a three-bit ($b_2 b_1 b_0$) counter, encoding integers up to seven, in Fig. 1. Non-deterministically, the counter is incremented, remains unchanged or is reset to zero after reaching five. Suppose we want to ensure that starting at state zero, all states with values greater than five are unreachable. A typical query asks "is state six reachable from any other state?", expressed as $SAT?[T \wedge (\neg b_2 \vee \neg b_1 \vee b_0) \wedge b'_2 \wedge b'_1 \wedge \neg b'_0]$, where T encodes the transition system for one step from $b_2 b_1 b_0$ to $b'_2 b'_1 b'_0$. It is unsatisfiable, telling us that state six is in fact unreachable. We can try to generalize this result to a set of states by considering a *cube* – an assignment to a subset of variables. The query $SAT?[T \wedge (\neg b_1 \vee b_0) \wedge b'_1 \wedge \neg b'_0]$ is satisfiable because state two can be reached from state one and $SAT?[T \wedge (\neg b_2 \vee b_0) \wedge b'_2 \wedge \neg b'_0]$ is satisfiable due to the transition from state three to state four. However, the query $SAT?[T \wedge (\neg b_2 \vee \neg b_1) \wedge b'_2 \wedge b'_1]$ is unsatisfiable, allowing us to conclude that all states in the cube $b_2 \wedge b_1$ are not reachable from outside the cube. We can use that insight to strengthen T by adding $\neg b'_2 \vee \neg b'_1$ to all future queries. This is in contrast to the clauses we previously added for only one query.

The popular assumption-based interface pioneered by MiniSat [2], [8] allows the user to specify a set of literals that are assumed to be true and picked by the solver as the first decisions. This allows us to add the assumption that a state is within a certain cube after the transition ($b'_2 \wedge b'_1$), however we still need to assume an additional clause encoding that the state is currently not within said cube ($\neg b_2 \vee \neg b_1$). The most common way to implement clause assumption, is to simulate the desired behavior using activation literals [8], [11]. Let C be a clause to add temporarily and a, the activation literal, a free variable, *i.e.,* it does not occur in the formula. By adding $C \vee a$ to the formula and assuming $\neg a$, we achieve the same as adding C to the formula. After a solution is found, the clause a is added, effectively removing C from the formula.

The problem with IC3 specifically, is the large number of queries made over the course of a single verification procedure. After a few hundred calls the activation literals clutter up the variable space and slow down the SAT solvers propagation. The common solution to this problem is to fully restart the SAT solver by replacing it with a fresh instance periodically, thus also deleting all learned clauses and heuristic scores. How to schedule these restarts in IC3 specifically, has been the topic of a full journal paper [12]. Using the technique presented in this paper, restarts are not necessary at all. Additionally learned clauses are safe to keep and will not contain an activation literal, which would make them useless for future calls.

Other approaches to clause assumption have been explored: The logic solver Satire [13] supports pseudo-Boolean and

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_15

other constraints. It records the dependencies of learned constraints explicitly, thus allowing the deletion of arbitrary clauses. In the SMT community, an interface based on pushing and popping on the assertion stack is prevalent [14]. Since constraints are removed in order, it is possible to mark a point in the data structures that maintain learned knowledge and remove everything past it, when a pop operation is executed. The first implementation of IC3 [15] used the SAT solver Zchaff [16]. It assigns an additional 32-bit integer to each clause. When learning a clause the bits of all dependencies are combined. The user can delete a group of clauses with a certain bit. This approach mostly simulates the use of activation literals and comes with a significant memory overhead.

This paper presents an extension of the prevalent assumption mechanism to additionally allow the assumption of a single clause, called *constraint* in the following. The extension can be implemented by a simple modification to the decision mechanism in a CDCL-based SAT solver. We implemented it in under 100 lines of code in the state-of-the-art SAT solver CaDiCaL. To evaluate our implementation we modify the IC3 engine in the model checker ABC to use CaDiCaL and clause assumption. As a first result, the changes simplify SAT solver usage and eliminate the need for restarts as well as some book-keeping for activation literals. An empirical evaluation on the 2019 hardware model checking competition [17] benchmark set shows that ABC spends less time outside of computing SAT queries, the number of queries per verification is reduced and the average SAT call is faster. Overall using clause assumptions yields a substantial speedup in verification time.

INCREMENTAL SAT AND IC3

An *incremental* SAT solver solves a series of related formulas efficiently. It communicates with an application integrating it through an *interface* such as IPASIR [11]. It is implemented by all solvers participating in the incremental library track of the SAT Competition since 2015. The popular solver MiniSat along with all of its incremental descendants implement something very similar. We describe the relevant subset:

- `add(lit)` Add a literal to the current clause or if it equals 0, add the clause to the formula.
- `assume(lit)` Assume the literal to be true for the next solving attempt.
- `solve()` Return SAT if an assignment exists satisfying the formula and all assumptions, otherwise UNSAT.
- `val(lit)` Valid in SAT-case. Return the truth value of a literal in the satisfying assignment.
- `failed(lit)` Valid in UNSAT-case. Return *true* if the literal was assumed and used to prove unsatisfiability.

A prominent applications of incremental SAT-solving is the symbolic model checking algorithm IC3 by Bradley [15]. Given a transition system and a property P, IC3 tries to prove that it is not possible to reach a state that violates the property. It maintains a sequence of *frames* $F_0, F_1, \ldots F_k$, each frame F_i is a formula encoding an overapproximation of the set of states reachable in at most i steps. The frames are refined by adding additional clauses until one of the frames contains all reachable states and none violates the property or a counterexample is found. Each frame has its own SAT solver instance that is initialized with an encoding of the transition function and updated with the new frame clauses.

The solvers are used almost exclusively to answer queries for predecessors of the form $SAT?[T \land F_i \land \neg s \land s']$, where T is the transition function and s is a cube. To refine the frames, a state s in the last frame that violates the property is identified with the query $SAT?[F_k \land \neg P]$. If no such state exists, a new frame is appended, otherwise IC3 tries to prove that the state is not actually reachable. The frames are queried for predecessors until an initial state is reached, thus producing a counterexample, or one of the frames returns unsat. In the latter case `failed` can be used to generalize the unreachable state to a cube, the negation of which is added to the frame. IC3 is guaranteed to eventually terminate with two consecutive frames containing the same set of states.

ASSUMING CLAUSES

Our main contribution is an extension to incremental SAT solvers that allows the assumption of an additional clause, called *constraint*, which is only valid during the next satisfiability query. Two functions are added to the interface:

- `constrain(lit)` Adds a literal to constraint. If a finalized constraint exists, delete it. If the literal equals zero, finalizes the current constraint.
- `constraint_failed()` Valid in *UNSAT* case. Return whether constraint was used to prove unsatisfiability.

Our approach is similar to the idea of model elimination [18]. We modify the decision heuristic to restrict the search to assignments that satisfy the constraint. The modified decision procedure is outlined in Fig. 2. The function decide is called initially at decision level 0. Decisions assigned to the trail are propagated outside of the function to assign truth values. Whenever a conflict arises, the decision level decreases and the assignments are backtracked [1]. Every assumption has a fixed decision level. In the case where an assumption is already satisfied, a *pseudo* decision level is introduced. Otherwise if an assumed literal is assigned to false at this point, the assignment is the result of propagating other assumptions together with original or learned clauses. Therefore the formula is proven unsatisfiable under the current assumptions if line 4 is reached.

At the first decision level after all assumptions have been assigned, three cases need to be considered: if one of the literals in the constraint is already satisfied, the search is not restricted. Otherwise one of the literals is picked as a decision to satisfy the constraint. In line 13 a variable selection heuristic can be used to pick the most promising literals first, similarly to [19], [20]. In the case where all literals are assigned to false, they are implied by the assumptions, thus cannot be assigned differently. The formula is therefore declared unsatisfiable under the assumptions and the constraint. This might only happen after additional clauses have been learned.

This approach to handle assumptions was pioneered by MiniSat [2]. It has been improved upon by collectively propagating the assumptions, using trail saving between incremental

```
decide ()
1  if level < |assumptions|
2      ℓ = assumptions[level]
3      if val(ℓ) = false
4          analyzeFinal()
5      else if val(ℓ) = true
6          level++ // pseudo decision level
7      else trail[level++] = ℓ
8  else if level = |assumptions|
9      unassignedLit = 0
10     for ℓ in constraint
11         if val(ℓ) = true
12             level++ // pseudo decision level
13         else if val(ℓ) = unassigned
14             unassigendLit = ℓ
15     if unassigendLit = 0
16         analyzeFinalConstraint() // cannot be satisfied
17     else trail[level++] = unassigendLit
18 else
19     ℓ = literalSelectionHeuristic()
20     trail[level++] = ℓ
```

Fig. 2. Algorithm decide picks the next decision to propagate.

calls [21] or factoring out assumptions [22]. These techniques can be combined with the presented constraint mechanism.

Modern SAT solvers not only report unsatisfiability as a result, but also allow the user to query whether a particular assumption failed, *i.e.,* was used to prove unsatisfiability. This concept, introduced as `analyzeFinal` by MiniSat [23], is essential for the efficiency of many applications. If an original or learned clause is inconsistent with the assumptions, the last assumption picked as a decision is already assigned to false. Using a simple breadth-first search, the reasons for this assignment can be traced back through the implication graph [1]. The assumptions at the leaves of the search tree are marked as failed. In line 16, a similar search is initialized with the negation of every literal in the constraint. Thus, all assumptions necessary to prove unsatisfiability of the constraint in conjunction with the formula are marked as failed.

Experiments

We implemented the constraint interface in CaDiCaL [24] version 1.3.1. To increase confidence in the correctness of the SAT solver and its new extension, we used the model-based tester [25] that is integrated with CaDiCaL. It generates random sequences of API calls including assumptions and constraints together with random configurations for the solver. The returned models and failed assumption sets are checked for correctness. We ran the tester on 8 cores for multiple days to validate 1.2 billion test runs.

To evaluate our approach, we integrated CaDiCaL into the bit-level model checker ABC[1] [26], replacing the integrated version of MiniSat [2]. There are two places where activation literals are used in ABC. The first is an alternative implementation of cube generalization, that is not used in the default configuration. In fact, it seems to not work correctly in the default version of ABC[1]. The other usage of activation literals is in the function that implements the predecessor query $SAT?[T \wedge F_i \wedge \neg s \wedge s']$. The transition function T and the frame F_i will only be extended with additional clauses, the cube s however changes at each query. The next-step cube s' is in conjunction with the rest of the formula and therefore translates to a set of unit clauses that can be implemented with assumptions. To combat the slowdown due to unused activation literals cluttering up the variable space, ABC replaces the SAT solver with a new instance after adding 300 activation literals. Using the extended interface, the negated cube $\neg s$ can be added as a constraint, thus eliminating the restarts.

We tested five configurations: the original version of ABC (Og), disabled SAT solver restarts (Di), a version with CaDiCaL as backend using activation literals (Ca) and one also using CaDiCaL but the new constraint interface instead of activation literals (Co). As an additional result we present a slight modification to the last configuration that defers model reconstruction [6] in the SAT-case and failed literal collection in the UNSAT-case until a model or a failed literal is queried respectively (De). Using a heuristic to pick the literals from the constraint has not been successful. ABC uses a priority metric to order the literals of the cube s by default. Using this order for the constraint turned out to be superior to the heuristics available in CaDiCaL.

Our evaluation follows the principles laid out in SAT manifesto v1.0. [27]. The source code used for the evaluation and the generated log files are available on our website[2]. The experiments are run in parallel on 32 nodes of our cluster. Each node has access to two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) and 128 GB main memory. We allocate 4 instances of ABC to every node. The time limit is set to 1 hour of wall-clock time, memory is limited to 30GB per instance. The memory limit is the only aspect that differs from the setup used in the hardware model checking competition. However, the maximum memory consumption was observed to be below 1.5GB.

The evaluation is based on the benchmark set used in the 2019 model checking competition [17]. It contains 219 instances, 15 of which we removed because they were not solved by any tested configuration. We use PAR-2 scoring to compare the configurations. PAR-2 assigns the runtime in seconds or twice the time limit (7200) if an instance was not solved. The other columns list additional measurements for the two configurations using CaDiCaL, one with activation literals (Ca) and the other using constraints instead (Co). The number of restarts is zero if constraints are used and

[1] commit f87c8b4
[2] http://fmv.jku.at/assumingclauses

TABLE I
EXPERIMENTAL RESULTS.

	PAR-2					Res.	Calls		TpC	
	Di	Og	Ca	Co	De	Ca	Ca	Co	Ca	Co
Mean	80	46	16	8.93	**8.21**	61	19	**15**	0.61	**0.51**
beemTele6Int	136	7200	**53**	181	101	520	**157**	574	**0.24**	0.27
toyLock4	7200	483	1731	**357**	359	7459	2251	**1098**	0.42	**0.25**
visArraysField5	7200	1.6	**0.58**	51	34	1	**1**	113	0.53	**0.41**
nan	208	421	163	158	**140**	1381	**420**	423	**0.29**	0.32
beemColl6Int	241	258	322	133	**108**	398	123	**91**	2.31	**1.24**
cal110	213	168	130	**110**	122	191	59	**42**	**1.96**	2.39
cal109	179	197	102	117	**86**	110	**34**	44	2.71	**2.44**
cal93	186	136	121	**118**	140	206	63	**58**	**1.69**	1.8
cal94	127	160	115	**95**	131	171	52	**41**	**1.94**	2.1
cal100	112	**42**	67	67	54	148	45	**44**	**1.23**	1.29
cal131	46	**44**	77	58	60	136	42	**35**	1.58	**1.41**
cal146	47	39	71	42	**38**	131	41	**23**	**1.51**	1.55
cal136	**34**	46	59	43	35	100	31	**23**	1.62	**1.59**
cal128	52	38	46	**37**	40	99	31	**25**	1.29	**1.27**
beemExit5Int	51	17	26	16	**15**	357	110	**86**	0.18	**0.15**
cal134	38	47	50	48	**36**	79	**25**	26	1.72	**1.57**
cal132	39	36	48	42	**32**	83	26	**24**	1.57	**1.54**
cal144	**30**	34	41	33	42	64	20	**17**	1.7	**1.64**
beemLampNat5Int	26	23	**23**	35	31	193	**61**	102	**0.28**	0.3
cal89	16	**14**	32	33	25	68	22	**18**	**1.23**	1.6
beemRether4Bstep	13	**4.29**	16	7.16	6.99	91	29	**13**	**0.42**	0.49
beemBrp2Int	16	5.1	3.6	0.76	**0.74**	86	29	**7**	0.08	**0.07**
beemFrogs2Bstep	**2.47**	2.53	12	5.59	4.74	31	10	**4**	**1.12**	1.27
beemAdding5Int	1.78	3.9	2.07	1.12	**1.09**	53	17	**11**	0.08	**0.07**
visArraysTwo	1.35	2.89	3.89	0.57	**0.55**	99	30	**5**	0.09	**0.07**
Heap	2.02	1.9	3.38	1.68	**1.63**	57	22	**13**	0.11	**0.09**

Disable restarts, **Orig**inal version of ABC, **Ca**DiCaL backend, **Co**nstraint interface used, **De**fer model reconstruction

therefore not shown. Besides that, we list the number of SAT calls (in thousands), along with the average time per call in milliseconds. Table I presents the measured data for instances, where at least one configuration took more than two seconds, along with an average over all 204 instances.

Comparing the first two columns, it is evident that if activation literals are used, solver restarts are necessary. It has been suggested [12] that because the queries posed by IC3 are small but numerous, IC3 implementations should prefer faster SAT solvers to more powerful ones. Comparing the original with the CaDiCaL version shows that while using MiniSat is faster on a number of instances, using CaDiCaL seems to be an advantage on the harder instances. In fact, using the newer SAT solver, one additional instance can be verified. Over all instances a speedup of 2.82 is observed.

With the version using CaDiCaL and activation literals as a baseline, we observe a speedup of 1.84 when switching to constraints. The time spend outside the SAT solver is reduced to below 20%, by eliminating the actual SAT solver restarts and the repeated loading of the transition relation [28]. Beyond that, the average SAT call is 16% faster. This can partially be explained by the solver not being slowed down by activation literals. We conjecture that, more importantly, the “quality” of the learned clauses in the solvers database is higher. Since clauses are not deleted by restarts and none of the learned clauses are implicitly disabled for containing an activation literal, the solver can profit from shorter and more useful clauses. Measuring this quality however, is outside the scope of this paper. An additional effect is that these clauses allow conflicts earlier in the search tree, resulting in fewer failed literals and thus allows for better generalization in IC3. This can explain why 21% fewer calls are made.

The last two columns listing PAR-2 scores reflect small changes in the solver. Deferring the model reconstruction results in an additional speedup of 9%, increasing the total speedup compared to the original version to 5.64.

CONCLUSION

We present a simple extension to the commonly used incremental SAT solver interface IPASIR that simplifies solver usage and is easy to implement by modern SAT solvers. The extension gives an alternative to the techniques described in the journal paper [12] and partially implemented in ABC. Our experiments using the new technique with ABC show a substantial improvement in model checking time. Compared to the original IC3 engine, our final implementation is more than five times faster.

Handling more than one constraint can be achieved by using a complete model elimination search over the constraints. This would however increase the implementation effort. Additionally, inprocessing techniques cannot be applied, therefore model elimination might be less effective than using activation literals, if the number of temporary clauses is high. We leave this investigation to future work.

Acknowledgements: This work is supported by the Austrian Science Fund (FWF); projects W1255-N23 / S11408-N23 as well as the LIT AI Lab funded by the State of Upper Austria.

References

[1] Marques-Silva, Joao and Lynce, Ines and Malik, Sharad, "Chapter 4. Conflict-Driven Clause Learning SAT Solvers," in *Handbook of Satisfiability: Second Edition*, Biere, Armin and Heule, Marijn and van Maaren, Hans and Walsh, Toby, Ed. IOS Press, feb 2021.

[2] Eén, Niklas and Sörensson, Niklas, "An Extensible SAT-Solver," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, Giunchiglia, Enrico and Tacchella, Armando, Ed. Berlin, Heidelberg: Springer, 2004, pp. 502–518.

[3] Audemard, Gilles and Lagniez, Jean-Marie and Simon, Laurent, "Improving Glucose for Incremental SAT Solving with Assumptions," in *Theory and Applications of Satisfiability Testing – SAT 2013*, ser. Lecture Notes in Computer Science, Järvisalo, Matti and Van Gelder, Allen, Ed. Berlin, Heidelberg: Springer, 2013, pp. 309–317.

[4] Eén, Niklas and Biere, Armin, "Effective Preprocessing in SAT Through Variable and Clause Elimination," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, Bacchus, Fahiem and Walsh, Toby, Ed. Berlin, Heidelberg: Springer, 2005, pp. 61–75.

[5] Järvisalo, Matti and Heule, Marijn J. H. and Biere, Armin, "Inprocessing Rules," in *Automated Reasoning*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 355–370.

[6] Fazekas, Katalin and Biere, Armin and Scholl, Christoph, "Incremental Inprocessing in SAT Solving," in *Theory and Applications of Satisfiability Testing – SAT 2019*, ser. Lecture Notes in Computer Science, Janota, Mikoláš and Lynce, Inês, Ed. Cham: Springer International Publishing, 2019, pp. 136–154.

[7] Biere, Armin and Cimatti, Alessandro and Clarke, Edmund and Zhu, Yunshan, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, Cleaveland, W. Rance, Ed. Berlin, Heidelberg: Springer, 1999, pp. 193–207.

[8] Eén, Niklas and Sörensson, Niklas, "Temporal Induction by Incremental SAT Solving," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 4, pp. 543–560, jan 2003.

[9] Bjesse, Per and Claessen, Koen, "SAT-Based Verification without State Space Traversal," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, Hunt, Warren A. and Johnson, Steven D., Ed. Berlin, Heidelberg: Springer, 2000, pp. 409–426.

[10] Sheeran, Mary and Singh, Satnam and Stålmarck, Gunnar, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, Hunt, Warren A. and Johnson, Steven D., Ed. Berlin, Heidelberg: Springer, 2000, pp. 127–144.

[11] Balyo, Tomáš and Biere, Armin and Iser, Markus and Sinz, Carsten, "SAT Race 2015," *Artificial Intelligence*, vol. 241, pp. 45–65, dec 2016.

[12] Cabodi, G. and Camurati, P. E. and Mishchenko, A. and Palena, M. and Pasini, P., "SAT Solver Management Strategies in IC3: An Experimental Approach," *Formal Methods in System Design*, vol. 50, pp. 39–74, mar 2017.

[13] Whittemore, J. and Kim, J. and Sakallah, K., "SATIRE: A New Incremental Satisfiability Engine," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, jun 2001, pp. 542–545.

[14] Barrett, Clark and Stump, Aaron and Tinelli, Cesare and others, "The Smt-Lib Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[15] Bradley, Aaron R., "SAT-Based Model Checking without Unrolling," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, Jhala, Ranjit and Schmidt, David, Ed. Berlin, Heidelberg: Springer, 2011, pp. 70–87.

[16] Fu, Zhaohui and Marhajan, Yogesh and Malik, Sharad, "Zchaff Sat Solver," 2004.

[17] Preiner, Mathias and Biere, Armin, "Hardware Model Checking Competition 2019," http://fmv.jku.at/hwmcc19/, 2019.

[18] Van Gelder, Allen, "Autarky Pruning in Propositional Model Elimination Reduces Failure Redundancy," *Journal of Automated Reasoning*, vol. 23, no. 2, pp. 137–193, aug 1999.

[19] Goldberg, Evguenii I. and Novikov, Yakov, "BerkMin: A Fast and Robust Sat-Solver," in *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*. IEEE Computer Society, 2002, pp. 142–149.

[20] Gershman, Roman and Strichman, Ofer, "HaifaSat: A New Robust SAT Solver," in *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, ser. Lecture Notes in Computer Science, Ur, Shmuel and Bin, Eyal and Wolfsthal, Yaron, Ed., vol. 3875. Springer, 2005, pp. 76–89.

[21] Hickey, Randy and Bacchus, Fahiem, "Speeding Up Assumption-Based SAT," in *Theory and Applications of Satisfiability Testing – SAT 2019*, ser. Lecture Notes in Computer Science, Janota, Mikoláš and Lynce, Inês, Ed. Cham: Springer International Publishing, 2019, pp. 164–182.

[22] Lagniez, Jean-Marie and Biere, Armin, "Factoring Out Assumptions to Speed Up MUS Extraction," in *Theory and Applications of Satisfiability Testing – SAT 2013*, ser. Lecture Notes in Computer Science, Järvisalo, Matti and Van Gelder, Allen, Ed. Berlin, Heidelberg: Springer, 2013, pp. 276–292.

[23] Eén, Niklas and Sörensson, Niklas, "MiniSat Page," http://minisat.se/.

[24] Biere, Armin, "Cadical, Lingeling, Plingeling, Treengeling and Yalsat Entering the Sat Competition 2018," *Proceedings of SAT Competition*, pp. 14–15, 2017.

[25] Artho, Cyrille and Biere, Armin and Seidl, Martina, "Model-Based Testing for Verification Back-Ends," in *Tests and Proofs*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 39–55.

[26] Brayton, Robert and Mishchenko, Alan, "ABC: An Academic Industrial-Strength Verification Tool," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, Touili, Tayssir and Cook, Byron and Jackson, Paul, Ed. Berlin, Heidelberg: Springer, 2010, pp. 24–40.

[27] Biere, Armin and Järvisalo, Matti and Le Berre, Daniel and Meel, Kuldeep S. and Mengel, Stefan, "The SAT Practitioner's Manifesto," sep 2020.

[28] Vizel, Y. and Grumberg, O. and Shoham, S., "Lazy Abstraction and SAT-Based Reachability in Hardware Model Checking," in *2012 Formal Methods in Computer-Aided Design (FMCAD)*, oct 2012, pp. 173–181.