Assuming Clauses

Nils Froleyks and Armin Biere

Johannes Kepler University Linz, Austria
{nils.froleyks,armin.biere}@jku.at

Abstract. We extend the well-established assumption-based interface of incremental SAT solvers to clauses, allowing the addition of temporary clauses that have the same lifespan as literal assumptions. Our approach is efficient and easy to implement in modern CDCL-based solvers and simplifies incremental SAT solver usage. Compared to previous approaches, it does not come with any memory overhead and does not slow down the solver due to disabled activation literals. All clauses learned under literal and clause assumptions are safe to keep and not implicitly invalidated for containing an activation literal. We implement the extension in the award-winning 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 considerable improvement in model checking time.

Introduction

Modern SAT solving is based on Conflict-Driven Clause Learning (CDCL) [23]. Many applications require solving a sequence of related SAT problems incrementally [16,2], making use of inprocessing techniques [13,21,17] that make modern SAT solvers so efficient. Among the applications that gain the most from incremental solvers are problems directly related to SAT, including MaxSAT [24] and minimal core extraction [26], planning [19,28] and in particular model checking [12], which is the focus of this paper. Starting with bounded model checking (BMC) [6,15], different SAT-based model checking techniques, such as k-induction [8,29], interpolation [25] and most recently IC3 [9], have been studied.

Motivated by the use of incremental SAT solving in `IC3`, consider the transition system of a three bit ($b_2b_1b_0$) 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 by `IC3` 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 $b_2b_1b_0$ to $b'_2b'_1b'_0$. 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

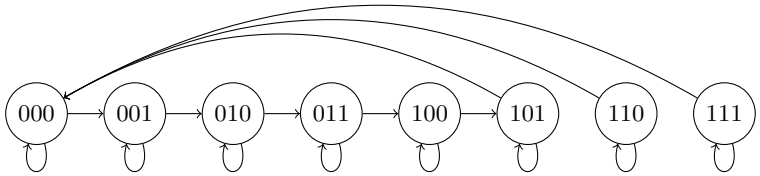

Fig. 1. Transition system

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 $\neg b_2' \vee \neg b_1'$ permanently to all future queries. This is in contrast to the clauses we previously added for only one query. To use an incremental SAT solver in this application, we need to add clauses temporarily.

The popular assumption-based interface pioneered by `MiniSat` [16,15] 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 handle the temporary clauses of length one, but we still need temporary addition of longer clauses. While assumption literals are implemented by nearly every incremental SAT solver, temporary clauses are less commonly supported.

The logic solver Satire [31] supports pseudo-Boolean and other constraints beyond Boolean clauses. It maintains a constraint hierarchy that records the dependencies between original and learned constraints. This allows the deletion of arbitrary constraints by identifying and recursively deleting all learned constraints dependent on them. Maintaining the hierarchy, however, is costly.

In the SMT community, an interface based on pushing and popping logical formulas on the assertion stack is prevalent [4]. Since the formulas 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. This might however delete a lot more learned knowledge than necessary.

The last SAT solver to explicitly support the removal of clauses is Zchaff [18]. It implements general clause deletion by associating every clause, learned or original, with an additional 32-Bit integer. The user can set one of the bits when adding a new clause, signaling that it is part of one of 32 groups that might be deleted later. Learned clauses collect the bits of all clauses they depend on. To delete a group, all clauses with the corresponding bit are removed. This approach comes with a high overhead in memory and implementation effort.

The most common way to implement temporary clauses is to simulate the desired behavior using activation literals [15,3]. Let C be a clause we wish to add temporarily and let a, the activation literal, be 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 solving the formula the clause a is added, effectively removing C from the formula. The main problem with

activation literals is, that they clutter up the variable space after disabling their clause and slow down propagation. This makes it necessary to restart the solver periodically. How to schedule these restarts in `IC3` specifically, has been explicitly investigated in [11]. Another problem that the last two approaches share is that they tend to learn clauses, that will not be used after the temporary clause has been removed. This is in contrast to clauses learned under literal assumptions, which are safe to keep and are useful after discarding the assumptions [15].

In this paper we present an extension to the assumption mechanism prevalent in modern SAT solvers, that allows the addition of a temporary clause. This clause, called *constraint* in the following, has the same lifespan as an assumption. 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 implementation of `IC3` provided by the model checker `ABC` to use `CaDiCaL` as the SAT solving back-end and use the extended interface where possible. As a first result, the changes simplify solver usage and eliminate the need for external restarts as well as some book-keeping for activation literals. An empirical evaluation on the 2019 hardware model checking competition [27] benchmark set shows that we gain a substantial increase in efficiency.

Incremental SAT and IC3

We introduce the Boolean satisfiability problem in the context of incremental solvers. A *literal* is a Boolean variable or its negation. Variables are identified with a non-zero integer (1) and negative literals with the negative variable (-1). A *clause* is a logical disjunction of literals. Its negation is a conjunction of literals called a *cube*. A *formula* is a conjunction of clauses. An *assignment* maps each variable that occurs in a formula to `true` or `false`. An assignment satisfies a positive literal if the variable is assigned `true`, a negative literal if it is assigned `false`, a clause if one of the literals is satisfied and a formula if all of the clauses are satisfied. An *incremental* SAT solver solves a series of related formulas efficiently. It communicates with an application integrating it through an *interface* such as IPASIR [3]. All solvers participating in the incremental library track of the SAT Competition since 2015 implement it. The popular solver `MiniSat` [16] along with all of its incremental descendants implement something very similar. We describe the relevant subset:

`add(int lit)` Used to add clauses. Add `lit` to the current clause or if `lit` equals 0, finalize the current clause and add it to the formula.

`assume(int lit)` Assume `lit` to be true for the next `solve` call. Assumptions are discarded after the next call to `solve`.

`solve()` If an assignment A exists that satisfies the formula and all assumptions return `SAT`. Otherwise return `UNSAT`.

`val(int lit)` Valid in `SAT` case. Return the truth value of `lit` in A.

`failed(int lit)` Valid in `UNSAT` case. Return `true` if assumption `lit` was used to prove unsatisfiability. Otherwise return `false`.

A prominent applications of incremental SAT-solving is the symbolic model checking algorithm `IC3` by Bradley [9]. 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 $SAT?[T \wedge F_i \wedge \neg s \wedge s']$, 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 $SAT?[F_k \wedge \neg P]$. 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 addition of a temporary clause, called *constraint*, which is only valid during the next satisfiability query. We add two functions to the incremental interface:

constrain(int lit) Adds `lit` to the current constraint. If a finalized constraint exists, delete it. If `lit` equals zero, finalizes the constraint.

constraint_failed() Valid in `UNSAT` case. Return `true` if the constraint was used to prove unsatisfiability. Otherwise return `false`.

Our approach is similar to the idea of model elimination [30]. 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 [23]. 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 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 of the constraint is already satisfied, the search is not further restricted. Otherwise one of the literals is picked as a decision to satisfy the constraint. In the case where all literals are assigned to

```
decide ( )
1  if level < |assumptions|
2      ℓ = assumptions[level]
3      if val(ℓ) = false
4          failingAssumption()
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         failingConstraint() // 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.

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` [16]. It has been improved upon by collectively propagating the assumptions and using trail saving even between incremental calls [20] 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` [14], 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 [23]. 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` version 1.3.1 [5]. To increase confidence in the correctness of the SAT solver and its new extension, we used the model-based tester [1] `mobical` 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] [10]. By default `ABC` uses a version of `MiniSat` [16] that is tightly integrated. Therefore not all functionality is fully implemented with `CaDiCaL`. Firstly, restarts of the solver without reallocating everything are not supported. Secondly, the on-demand compression of the clause database, which in `MiniSat` simply means unit propagation, is not implemented.

Temporary clauses are implemented with activation literals in `ABC`. To deal with the accumulating useless activation literals, the solver is restarted after adding 300 of them. A syntactic search on the source code of `ABC` indicated two places where temporary clauses are used. The first is an alternative implementation of cube generalization, that is not used in the default configuration. In fact, this implementation seems to not work correctly in our version of `ABC`[1]. We will not use it for our experiments. The other occurrence is in the function `CheckCube` that implements the predecessor query $SAT?[T \wedge F_i \wedge \neg s \wedge s']$. While the transition function T and the frame F_i will only be extended with additional clauses, the cube s changes every 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. Using the extended interface, the negated cube $\neg s$ can be added as a constraint. Since no activation literals are used, the solver will not be restarted. For one of the tested configurations (see Table 1), we disable the solver restarts explicitly. No further modifications have been made to the source code or the default settings of `ABC`.

Our evaluation follows the principles laid out in SAT manifesto v1.0. [7]. 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, each solving a different benchmark. 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 [27]. The results are presented in Table 1.

[1] commit f87c8b4

[2] http://fmv.jku.at/assumingclauses

Comparing the first two columns, it is evident that if activation literals are used, solver restarts are necessary. It has been suggested [11] 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. Overall, using the mean PAR2 score as a metric, 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. Besides the actual time it takes to restart the SAT solver, the average SAT call is 16% faster. This can be explained by the solver not being slowed down by activation literal clutter and additional clauses at all. We conjecture that, more importantly, the "quality" of the learned clauses 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, when using constraints.

As an additional result we present a modification to `CaDiCaL` that defers the collection of failed literals and the model reconstruction [17] until a literal is queried or a model is requested. Adding this minor change in the SAT solver to the configuration using constraints, results in an additional speedup of 9%, increasing the total speedup compared to the original version to 5.64.

Conclusion

We presented 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. Our experiments show that a considerable speedup can be achieved by using constraints. Since the development of new SAT encodings depends on the interface available, we hope that new applications will arise as more solvers implement the extended interface. Handling more than one constraint can be achieved using a complete model elimination search over the constraints. This would however increase the implementation effort and might be less effective than using activation literals, if the number of temporary clauses is high. We leave this to future work.

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

Table 1. The benchmark set 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 performance of the five tested configurations (explained below). 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 and the other using constraints instead. The number of restarts is zero if constraints are used and therefore not shown. Besides that, we list the number of SAT calls (in thousands), along with the average time per call in milliseconds. We display the measured data for all instances, where at least one configuration took more than two seconds, along with an average over all 204 instances.

	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**

Di Disable SAT solver restarts in the original version **Og** Original version of `ABC`
Ca `CaDiCaL` is used as the solver **Co** Constraints are used instead of activation literals
De Defers the model reconstruction and only executes it if a model is needed

References

1. Artho, C., Biere, A., Seidl, M.: Model-Based Testing for Verification Back-Ends. In: Tests and Proofs. pp. 39–55. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_3
2. Audemard, G., Lagniez, J.M., Simon, L.: Improving Glucose for Incremental SAT Solving with Assumptions. In: Theory and Applications of Satisfiability Testing – SAT 2013. pp. 309–317. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_23
3. Balyo, T., Biere, A., Iser, M., Sinz, C.: SAT Race 2015. Artificial Intelligence **241**, 45–65 (dec 2016). https://doi.org/10.1016/j.artint.2016.08.007
4. Barrett, C., Stump, A., Tinelli, C., et al.: The smt-lib standard: Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, England). vol. 13, p. 14 (2010)
5. Biere, A.: Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. Proceedings of SAT Competition pp. 14–15 (2017)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
7. Biere, A., Järvisalo, M., Le Berre, D., Meel, K.S., Mengel, S.: The SAT Practitioner's Manifesto (sep 2020). https://doi.org/10.5281/zenodo.4500928
8. Bjesse, P., Claessen, K.: SAT-Based Verification without State Space Traversal. In: Hunt, W.A., Johnson, S.D. (eds.) Formal Methods in Computer-Aided Design. pp. 409–426. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_23
9. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
10. Brayton, R., Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 24–40. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
11. Cabodi, G., Camurati, P.E., Mishchenko, A., Palena, M., Pasini, P.: SAT solver management strategies in IC3: An experimental approach. Formal Methods in System Design **50**, 39–74 (mar 2017). https://doi.org/10.1007/s10703-017-0272-0
12. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking, vol. 10. Springer (2018)
13. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing. pp. 61–75. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11499107_5
14. Eén, N., Sörensson, N.: MiniSat Page. `http://minisat.se/`
15. Eén, N., Sörensson, N.: Temporal Induction by Incremental SAT Solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (jan 2003). https://doi.org/10.1016/S1571-0661(05)82542-3
16. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing. pp. 502–518. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37

17. Fazekas, K., Biere, A., Scholl, C.: Incremental Inprocessing in SAT Solving. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing – SAT 2019. pp. 136–154. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_9
18. Fu, Z., Marhajan, Y., Malik, S.: Zchaff sat solver (2004)
19. Gocht, S., Balyo, T.: Accelerating SAT Based Planning with Incremental SAT Solving. Proceedings of the International Conference on Automated Planning and Scheduling **27**(1) (jun 2017)
20. Hickey, R., Bacchus, F.: Speeding Up Assumption-Based SAT. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing – SAT 2019. pp. 164–182. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_11
21. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing Rules. In: Automated Reasoning. pp. 355–370. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_28
22. Lagniez, J.M., Biere, A.: Factoring Out Assumptions to Speed Up MUS Extraction. In: Järvisalo, M., Van Gelder, A. (eds.) Theory and Applications of Satisfiability Testing – SAT 2013. pp. 276–292. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_21
23. Marques-Silva, J., Lynce, I., Malik, S.: Chapter 4. Conflict-Driven Clause Learning SAT Solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Frontiers in Artificial Intelligence and Applications. IOS Press (feb 2021). https://doi.org/10.3233/FAIA200987
24. Martins, R., Joshi, S., Manquinho, V., Lynce, I.: On Using Incremental Encodings in Unsatisfiability-based MaxSAT Solving. Journal on Satisfiability, Boolean Modeling and Computation **9**(1), 59–81 (jan 2014). https://doi.org/10.3233/SAT190102
25. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Computer Aided Verification. pp. 1–13. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
26. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: Formal Methods in Computer Aided Design. pp. 221–229 (oct 2010)
27. Preiner, M., Biere, A.: Hardware model checking competition (HWMCC) 2019. http://fmv.jku.at/hwmcc19/ (2019)
28. Schreiber, D., Pellier, D., Fiorino, H., Balyo˘, T.: Tree-REX: SAT-Based Tree Exploration for Efficient and High-Quality HTN Planning. Proceedings of the International Conference on Automated Planning and Scheduling **29**, 382–390 (2019)
29. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Hunt, W.A., Johnson, S.D. (eds.) Formal Methods in Computer-Aided Design. pp. 127–144. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
30. Van Gelder, A.: Autarky Pruning in Propositional Model Elimination Reduces Failure Redundancy. Journal of Automated Reasoning **23**(2), 137–193 (aug 1999). https://doi.org/10.1023/A:1006143621319
31. Whittemore, J., Kim, J., Sakallah, K.: SATIRE: A new incremental satisfiability engine. In: Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232). pp. 542–545 (jun 2001). https://doi.org/10.1145/378239.379019