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 will not contain 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.

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 asks “is state six reachable from any other state?”, expressed as \(SAT?\left[ T \land \left(-b_2 \lor -b_1 \lor b_0\right) \land b_2' \land b_1' \land -b_0' \right] \), where \(T\) encodes the transition system for one step from \(b_2b_1b_0\) 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 \(SAT?\left[ T \land \left(-b_1 \lor b_0\right) \land b_1' \land -b_0' \right] \) is satisfiable because state two can be reached from state one and \(SAT?\left[ T \land \left(-b_2 \lor b_0\right) \land b_2' \land -b_0' \right] \) is satisfiable due to the transition from state three to state four. However, the query \(SAT?\left[ T \land \left(-b_2 \lor -b_1\right) \land b_2' \land b_1' \right] \) is unsatisfiable, allowing us to conclude that all states in the cube \(b_2 \land b_1\) are not reachable from outside the cube. We can use that insight to strengthen \(T\) by adding \(-b_2' \lor -b_1'\) 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 \((b_2' \land b_1')\), however we still need to assume an additional clause encoding that the state is currently not within said cube \((-b_2 \lor -b_1)\). 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 \(C \lor a\) to the formula and assuming \(-a\), 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
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 bookkeeping 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:

- **constraint(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
The returned models and failed assumption sets are checked to validate 1.2 billion test runs.

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 \(SAT^?\{T \land F_i \land \neg s \land s'\}\). 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

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

Fig. 2. Algorithm `decide()` picks the next decision to propagate.
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