Journal of Artificial Intelligence Research 1 (1) 1 Submitted 1; published 1

Satisfiability modulo User Propagators

Katalin Fazekas KATALIN.FAZEKAS@TUWIEN.AC.AT
TU Wien, Vienna, Austria

Aina Niemetz NIEMETZ@CS.STANFORD.EDU
Stanford University, Stanford, USA

Mathias Preiner PREINER@CS.STANFORD.EDU
Stanford University, Stanford, USA

Markus Kirchweger MK@AC.TUWIEN.AC.AT
TU Wien, Vienna, Austria

Stefan Szeider SZ@AC.TUWIEN.AC.AT
TU Wien, Vienna, Austria

Armin Biere BIERE@CS.UNI-FREIBURG.DE
University of Freiburg, Freiburg, Germany

Abstract

Modern SAT solvers are often integrated as sub-reasoning engines into more complex tools to address problems beyond the Boolean satisfiability problem. Consider, for example, solvers for Satisfiability Modulo Theories (SMT), combinatorial optimization, model enumeration, and model counting. There, the SAT solver can often provide relevant information beyond the satisfiability answer and the domain knowledge of the embedding system, such as symmetry properties or theory axioms, may benefit the CDCL search. However, this knowledge can often not be efficiently represented in clausal form.

This paper proposes a general interface to inspect and influence the internal behaviour of CDCL SAT solvers. The aim is to capture the essential functionalities that simplify and improve use cases requiring a more fine-grained interaction with the SAT solver than provided via the standard IPASIR interface. For our experiments, the state-of-the-art SAT solver CaDiCaL is extended with the proposed interface and evaluated on two representative use cases: enumerating graphs within the SAT modulo Symmetries framework (SMS), and as the main CDCL($\mathcal{T}$) SAT engine of the SMT solver CVC5.

1. Introduction

Modern SAT solvers are often used as crucial sub-reasoning engines for more complex tools that address problems beyond the Boolean satisfiability problem. Use cases include Satisfiability Modulo Theories (SMT) (Barrett et al., 2021), combinatorial problems (Bacchus et al., 2021; Zhang, 2021), or model enumeration and counting (Gomes et al., 2021). The IPASIR interface (Balyo et al., 2016) has enabled the integration of off-the-shelf SAT solvers as a black box into larger systems. This is typically done to incrementally solve a sequence of similar propositional sub-problems. Many applications of SAT solvers, however, require a tighter integration with a more fine-grained interaction of the SAT solver with the rest of the system. A prominent example is the CDCL($\mathcal{T}$) framework for SMT solvers (Nieuwenhuis et al., 2006), where the search of the core SAT solver on the propositional abstraction of the input problem is guided by theory solvers. Other use cases include MaxSAT solvers,

which benefit from knowing if some literals imply others (Ignatiev et al., 2019), and solvers for symmetric combinatorial problems, where it is desired to add additional clauses during search (Devriendt et al., 2012). Currently, such use cases require either workarounds on the user level or non-trivial modifications of the SAT solver. As a result, replacing the underlying SAT solver is difficult, which hinders the adoption of recent advancements in SAT solving. Furthermore, non-standard extensions and modifications to the SAT solver may result in unintended performance issues if not implemented with care.

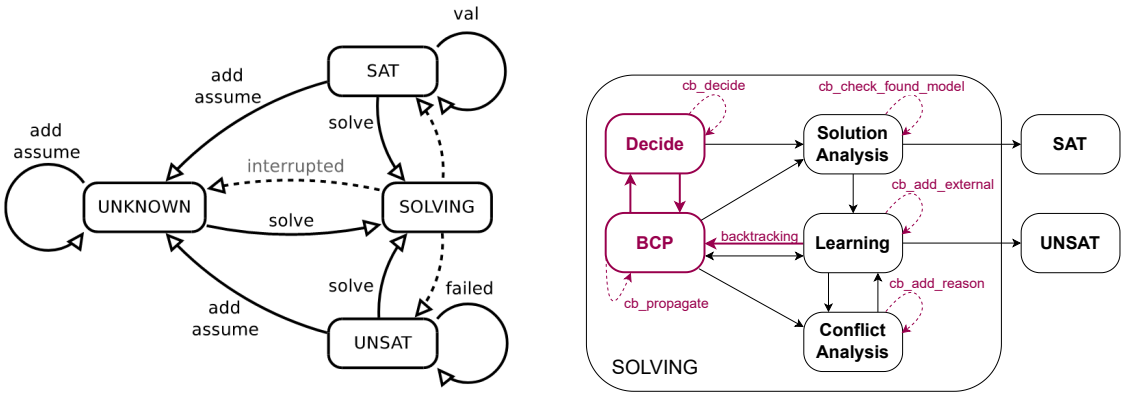
In this paper, we propose a generic interface able to capture the essential functionalities necessary to simplify and improve such use cases of SAT solvers. For this purpose, we extend the IPASIR interface (Balyo et al., 2016) with an interface to facilitate *external propagators*, also called *user propagators* (UP), yielding a new interface called IPASIR-UP.

Our extension allows users (1) to inspect and be notified about changes to the trail during search, (2) to add clauses to the problem during solving without restarting the search, (3) to propagate literals directly, based on external knowledge, without explicitly adding reason clauses (i.e., using delayed on-demand explanation) and to (4) guide the search by overwriting the internal decision heuristic of the solver based on external knowledge. Implementing support for such an interface is non-trivial in a state-of-the-art SAT solver, but enables a wide range of applications to efficiently use the solver without further, application-specific workarounds and modifications. To advocate our proposed interface, we implemented it in CaDiCaL (Biere et al., 2020, 2024), a state-of-the-art incremental SAT solver, on top of its implementation of the IPASIR interface.

Furthermore, we present two representative use cases of this extension of CaDiCaL in two different application contexts: integrating CaDiCaL via IPASIR-UP as the core SAT solver into (1) the CDCL-based SAT modulo Symmetries (SMS) framework and (2) a CDCL($\mathcal{T}$)-based Satisfiability Modulo Theories (SMT) solver. Our experiments present evidence that the IPASIR-UP interface provides a rich and concise interface for a modern, proof-producing, incremental SAT solver with inprocessing in such applications.

This paper extends the recent short tool paper (Fazekas et al., 2023) published at SAT'23 which introduced the IPASIR-UP interface. Based on comments and discussions with numerous developers, particularly developers of SAT, SMT and MaxSAT solvers, and users of SAT solvers, we have refined and slightly extended the IPASIR-UP interface with new features such as forgettable external clauses and batched assignment notifications[1]. Here, we describe this extended interface and provide some additional details that were not included in (Fazekas et al., 2023) due to space constraints: in Section 2.2 we explain some of the design decisions regarding the notification system of the interface, in Section 2.4 we present a possible schedule of the IPASIR-UP callbacks, and in a completely new Section 3 we provide additional details on how to combine IPASIR-UP with some of the common features of modern CDCL SAT solvers. Further, we extended our experimental evaluation with additional experiments. In the context of SMS, we include an experiment where we use CaDiCaL to generate all connected, cubic, claw-free graphs up to isomorphism. In the context of SMT, we additionally evaluate the performance of CaDiCaL as the CDCL($\mathcal{T}$) SAT engine of cvc5 on incremental SMT benchmarks.

1. The latest version of our IPASIR-UP implementation in CaDiCaL is available at https://github.com/arminbiere/cadical.



(a) The possible states of SAT solvers according to the IPASIR interface (see (Balyo et al., 2016)).

(b) The five additional states within state `Solving` according to the IPASIR-UP interface.

Figure 1: IPASIR model and its extension with states and transitions within CDCL solving.

2. An Interface beyond IPASIR

The IPASIR interface, as introduced in (Balyo et al., 2016), considers four possible states of a SAT solver (see Figure 1a). Initially, and while the formula is under construction, the solver is in state `UNKNOWN`. When function `solve()` is called, it transitions into state `Solving`. From that state, the solver can transition to either `SAT` or `UNSAT` (or, on interruption, back to `UNKNOWN`). Thus, IPASIR allows multiple calls to `solve()` while modifying the formula or querying details of the found solution (resp. refutation) *between* such calls. It is, however, not possible to interact with the solver while it is *in* the `Solving` state (except for interruptions, see dashed line in Figure 1a). Our goal is to extend the IPASIR interface with functions that can provide such interactions, and thereby allow to simplify and improve several use cases of modern incremental CDCL SAT solvers.

For this purpose, our interface IPASIR-UP refines the IPASIR state `Solving`, which implements the main CDCL loop, into *five* states, as shown in Figure 1b. CDCL combines unit propagation (`BCP`) with decisions (`Decide`) until either a clause becomes falsified by the current assignment or each variable is assigned a truth value. In the first case, the solver transitions into state `Conflict Analysis`, where it captures the reason of the contradiction as a derived driving clause, which is then learned in state `Learning`. If the learned clause is empty, the solver transitions to the `UNSAT` state. Otherwise, it backtracks to a lower decision level and unit propagation starts again. In the second case, as soon as a complete assignment is found, a standard CDCL solver will transition into the state `SAT`. In the presence of an *external propagator*, however, we introduce an artificial state called `Solution Analysis` as an intermediate state before transitioning to `SAT`.

In each of the five states in Figure 1b, IPASIR-UP provides a callback (with prefix "`cb_`") to interact with the external propagator (dashed transitions in Figure 1b, see Section 2.3). Additionally, the propagator is being notified about changes to the trail (states and solid transitions highlighted in purple in Figure 1b, see Section 2.2). In the following, we briefly describe the main purpose of each function. Though we illustrate IPASIR-UP here with an example implementation in C++ (see Listings 1 and 2), the API is low-level enough to

Listing 1: Functions for Configuration and Management (see Section 2.1)

```
// VALID = UNKNOWN | SATISFIED | UNSATISFIED
//
// require (VALID) -> ensure (VALID)
//
void connect_external_propagator (ExternalPropagator * propagator);

// require (VALID) -> ensure (VALID)
//
void disconnect_external_propagator ();

// require (VALID_OR_SOLVING) /\ CLEAN(var) -> ensure (VALID_OR_SOLVING)
//
void add_observed_var (int var);

// require (VALID) -> ensure (VALID)
//
void remove_observed_var (int var);

// require (VALID_OR_SOLVING) -> ensure (VALID_OR_SOLVING)
//
bool is_decision (int observed_var);

// require (VALID_OR_SOLVING) -> ensure (VALID_OR_SOLVING)
//
void phase (int lit);

// require (VALID_OR_SOLVING) -> ensure (VALID_OR_SOLVING)
//
void unphase (int lit);

// require (VALID_OR_SOLVING) -> ensure (VALID_OR_SOLVING)
//
void force_backtrack (size_t new_level);
```

be supported in C with only minor modifications. The goal of our interface proposal is to explicitly identify certain possible interactions during the CDCL loop of a SAT solver. However, the precise timing and exact syntax of this interaction should always depend on the actual technical details of the SAT solver.

Note that many other (in this context) less relevant steps of the search (e.g., restart, reduce, and inprocessing) are, for now, ignored in the model of our interface. In Section 3, we will revisit some of these steps in the presence of an external propagator.

2.1 Configuration and Management

In order to be able to interact with the solver while in the `Solving` state, a user may connect and configure an *external propagator* through IPASIR-UP as follows.

Setup. When the solver is not in the `Solving` state, the user can connect an external propagator via the function `connect_external_propagator`. This propagator may

be disconnected outside of `Solving` via `disconnect_external_propagator`. There can be at most one external propagator connected to a solver. In practice this is not a limitation since, conceptually, multiple propagators on the user side can be managed and abstracted into one propagator facing the SAT solver.

Observed Variables. While an external propagator is connected, at any point in time (even during `Solving`), the user can notify the solver that a variable, that might be even new, is "relevant" by declaring it as an *observed variable* via `add_observed_var`. When not in state `Solving`, observed variables can be removed via `remove_observed_var`. Note that all IPASIR-UP calls involve observed variables only.

Additional Useful Functions. We propose two additional functions. First, function `phase` (as already implemented in some solvers) allows to force a particular phase of the specified variable when making a decision on that variable. Similarly, `unphase` allows the solver to fall back to its own preferred phase. Second, function `is_decision` can be queried for a given variable to determine if it is currently assigned by a decision. Third, users can force the SAT solver to backtrack to a certain decision level (only before a new decision would be made) via `force_backtrack`.

The complete signature of each of these functions is shown in Listing 1. The comments above the functions indicate the IPASIR state of the SAT solver when the function is allowed to be called (see Figure 1a for their relations). The union of states `UNKNOWN`, `SAT`, and `UNSAT` is referred to as `VALID` states here, while the state `VALID_OR_SOLVING` indicates that the function can be called also while the solver is in the `Solving` state.

2.2 Inspecting the Trail via Notifications

In order to be able to interfere with the search of a solver, it is important to know in which state the solver currently is. When it is in a valid state, most of its state is captured by the sets of redundant and irredundant clauses together with the reconstruction stack (see, e.g., (Järvisalo et al., 2012; Fazekas et al., 2019) for an abstract calculus based on such states). However, our focus here is on the `Solving` state, which requires more details to be sufficiently captured. The transition system introduced in (Nieuwenhuis et al., 2006) describes the `Solving` state of DPLL and CDCL SAT solvers as a pair of the form $M||F$, where F is a set of clauses and M is a partial assignment, representing the current *trail* of the solver. For proofs and solutions of SAT solving, the changes applied to F are the most important details. For external propagation, the changes of M must be captured properly.

There are several possible ways how these changes can be captured and communicated to users. A simple solver could make its trail directly accessible from outside. However, this would require a standardized way of organizing trails of SAT solvers, which may potentially disallow the use of a compacted internal representation of variables. Such an optimized representation is currently utilized in CaDiCaL and Kissat (Biere et al., 2020). Thus, instead of directly sharing the trail between user and solver, our goal is to hide trail internals but share enough information about its changes for users to sufficiently maintain relevant trail information outside of the solver.

The simplest view of the trail of a SAT solver is as a *stack* of literals, where each literal is assigned either by *decision* or *propagation* (Silva & Sakallah, 1999; Eén & Sörensson,

Listing 2: A C++ example implementation of functions for inspecting and influencing CDCL

```
class ExternalPropagator {
public:
  bool are_reasons_forgettable;

  virtual ~ExternalPropagator () { }

  virtual void notify_assignment (const std::vector<int>& lits) {};
  virtual void notify_new_decision_level () {};
  virtual void notify_backtrack (size_t new_level) {};

  virtual int cb_decide () { return 0; }
  virtual int cb_propagate () { return 0; }
  virtual int cb_add_reason_clause_lit (int propagated_lit) {
    return 0;
  }
  virtual bool cb_check_found_model (const std::vector<int> & model) {
    return true;
  }

  virtual bool cb_has_external_clause (bool & is_forgettable) {
    return false;
  }
  virtual int cb_add_external_clause_lit () { return 0; }
};
```

2003a). Whenever a literal is pushed to the trail due to a decision, it starts a new *decision level*. Then, based on that trail, unit propagation pushes literals, associated with that level, to the top of the trail stack. When the solver needs to backtrack or back-jump, it simply pops the top literals off the stack until it reaches the assignments of the desired decision level. This data structure (a *stack-like trail*) can be implemented efficiently and has a clean interface, where changes can be captured as a sequence of push and pop operations.

The trail of modern SAT solvers, however, may in practice not necessarily behave as a stack. Allowing both chronological and non-chronological backtracking during search may result in *out-of-order assignments* on the trail. On such *out-of-order trails*, the decision level of assignments may decrease during search. Even though there are certain invariants that can be assumed about such trails (Nadel & Ryvchin, 2018; Möhle & Biere, 2019), push and pop are not the most efficient ways to capture their changes (see, e.g., (Nadel, 2022), where the trail is actually a double linked list). The simplest way to communicate changes of an out-of-order trail is to leave behind the concept of decision levels and simply let users know when variables are assigned or unassigned during search.

In several application domains of SAT, e.g., in the context of SMS, such a notification strategy would suffice. However, one of the most prominent use cases of IPASIR-UP is the embedding of modern SAT solvers into SMT solvers. In the context of the CDCL($\mathcal{T}$) solving framework for SMT, the trail actually determines a conjunction of theory literals and is viewed as a stack-like trail, which is manipulated by specialized theory solvers via push and pop operations. To support also those use cases that require a stack-like view of

the trail, IPASIR-UP provides the following three notification functions (see Listing 2 for their example C++ signatures):

`notify_new_decision_level` The call of this function indicates to the user that on the trail a new decision level has started. The function does not report the actual decision that started this new level or the current decision level — it only reports that a decision happened and thus, the decision level is increased.

`notify_assignment` This function is called when observed variables are assigned (either by `BCP`, `Decide`, or `Learning` a unit clause). It has a single read-only argument containing literals that became satisfied by the new assignment. In case the notification reports more than one literal, it is guaranteed that all of the reported literals were assigned on the same (current) decision level.

`notify_backtrack` This function indicates that the solver backtracked to a lower decision level. Its single argument reports the new decision level. All assignments that were made above this target decision level must be considered as unassigned.

Expressing assignments and unassignments as a stack-like view can be challenging when the SAT solver employs several techniques leading to out-of-order assignments (see for example (Coutelier, 2023)). This challenge should thus be addressed by the developers of SAT solvers, who are aware of the details of the supported techniques, rather than the (usually unaware) users of the SAT solvers (see Section 3.5 for more details).

Developers of SAT solvers must also decide when to schedule the calls of these notification functions. It is acceptable to delay notifications as much as possible, e.g., to only notify the user about assignments after unit propagation has finished. It may also be necessary to not send notifications while the user is not allowed to interfere anyway (e.g., during preprocessing). However, it must be guaranteed that the user has a correct view of the current assignments whenever any of the callback functions of Section 2.3 are called.

2.3 Influencing CDCL via Callbacks

In Section 2.2, we focused on notifying the user about the changes to the trail of the SAT solver. Based on this information, in each of the five states of the search, IPASIR-UP allows the user to influence CDCL in various ways via the following callback functions (see Listing 2 for the function signatures).

`Decide`. Before the solver makes a decision, the callback `cb_decide` allows the user to enforce a user-specific choice of the selected variable and phase. Note that users can inject decisions only after all assumptions are satisfied.

This is also the state where users can enforce the solver to backtrack via the function `force_backtrack`. If this function is called at any other time, or if it leads to an invalid decision level, the query will be ignored by the solver. In the future, we plan to define a unified error code system that can be used to notify users when such errors occur while using the propagator interface. When backtracking to a level where not all assumptions are yet satisfied, the solver will first make its forced decisions based on the assumptions before allowing the user to interfere with the search again.

`BCP`. During unit propagation, the user can provide additional literals to be propagated through the `cb_propagate` callback. Note that this callback returns only a literal to be propagated. The propagating clause is not required at this point.

`Conflict Analysis`. If during conflict analysis a previous user propagation (see above) turns out to be relevant (i.e., necessary to derive the learnt clause), the solver asks for the corresponding reason clause via `cb_add_reason_clause_lit`, one literal at a time. The motivation for such delayed lazy explanation (see (Nieuwenhuis et al., 2006; Gent et al., 2010)) during conflict analysis is to generate and learn only useful clauses.

`Solution Analysis`. If the solver determines a full assignment without falsifying any present clauses (i.e., a SAT solution is found), `cb_check_found_model` is called. This function tells the solver if the SAT model is consistent with external user constraints. In case of inconsistency, additional clauses can be added to the problem without restarting the search (see below).

`Learning`. When the solver has finished `BCP` (right before `Decide`), or callback `cb_check_found_model` returned false, users can add new clauses to the problem. Callback `cb_has_external_clause` indicates if a new clause is to be added. A new clause is then added via `cb_add_external_clause_lit`, literal by literal. For proof generation, by default, the solver adds these clauses as irredundant original input clauses. However, in incremental SAT proofs this addition happens *during* derivation, not before, and so incremental proof checking can ignore them until that point (see (Fazekas et al., 2024b) and Section 3.6 for more details on it). In case the user sets the argument "is_forgettable" to true, the solver is allowed to delete the clause later (see Section 3.1). If the learned clause propagates (resp. is falsified) under the current trail, the solver transitions to `BCP` (resp. `Conflict Analysis`). When no more clauses are to be added, the solver continues the search.

2.4 An Example CDCL Flow with IPASIR-UP Callbacks

This section presents an example of a possible order of IPASIR-UP callbacks within the CDCL loop of a SAT solver. For the sake of keeping the presentation simple, we omit many implementation details and corner cases. Note that this example illustrates only one possible schedule of calls. The focus here is on showcasing the organization of the interactions with an external propagator on a possible order of calls. A complete example of an implementation can be found in the source code of CaDiCaL (Biere et al., 2020).

Algorithm 1 shows a simplified CDCL loop extended with calls to an external propagator via the IPASIR-UP functions. The SAT search always applies its own Boolean Constraint Propagation (`BCP`) function (`propagate`) first. If this leads to a conflict, it must be resolved and the solver must backtrack. These conflict handling and backtracking steps are captured by the call to `analyze_conflict` in Algorithm 1. In case a conflict cannot be resolved, `analyze_conflict` will return 20 (unsat) and the CDCL loop will terminate. If there is nothing to propagate by `BCP`, the solver asks the external propagator if there is anything the user wants to propagate (see Algorithm 2 below for details). If the external propagator has nothing to propagate, the solver asks if there is an external clause to add (via Algorithm 3). In case of no propagations and no external clauses to consider, the SAT solver checks if

Algorithm 1: CDCL loop with a connected external propagator (simplified).

```
Data: Solver trail τ, set of all variables V.
Result: 10 if formula is satisfiable, 20 if unsatisfiable.
res ← 0;
while res = 0 do
    if propagate() ≠ ok then
        res ← analyze_conflict();
    else if external_propagate() ≠ ok then
        res ← analyze_conflict();
    else if add_external_clauses() ≠ ok then
        res ← analyze_conflict();
    else if |τ| = |V| then
        if external_check_solution() ≠ ok then
            res ← analyze_conflict();
        else if |τ| = |V| then
            res ← 10;
    else
        external_decide();
return res;
```

there are any unassigned variables left. If not, a complete truth assignment has been found without falsifying any of the clauses, thus the trail is a solution. At this point, the external propagator must be asked to approve this solution. If the propagator indicates a conflict or introduces new variables while checking the solution, the search must continue. Otherwise, the solver returns with a satisfiable assignment. If none of the above applies, the solver must make a decision (see Algorithm 4).

Algorithm 2 shows the main steps of handling the external propagation of a literal l. If the literal is already falsified, the solver must start a conflict analysis. Details of the corresponding conflict analysis are omitted here, but note that any relevant unexplained propagation step must be explained during this analysis via function `cb_add_reason_clause_lit`. If the literal is already satisfied, the solver can ignore this propagation. If the literal is not yet assigned, the solver can assign it as an unexplained external propagation step and call `BCP` again. After that, once the user is notified of the new assignments, they can propagate more literals using the function `cb_external_propagate`. It is important to note here that external propagation happens literal-by-literal and not batch-wise, because in that way the user immediately sees when one of the propagations causes a conflict in the SAT solver.

Algorithm 3 shows the main steps of handling an external clause addition during the CDCL loop. First, the propagator must be notified of new assignments. Then, users can indicate via function `cb_has_external_clause` whether there is an external (potentially forgettable) clause to add. If there is such a clause, it is read into C literal by literal (in the same way as IPASIR allows clauses to be added). If clause C is satisfied by some fixed assignments, it can be ignored. If it is falsified by the current trail, conflict analysis must

Algorithm 2: External propagation within a CDCL SAT solver (simplified).

```
Function external_propagate()
    Data: Trail τ, literal l, l ∈ τ indicates that l is assigned true under τ.
    notify_assignments();
    l ← cb_external_propagate();
    while l ≠ 0 do
        switch l do
            case ¬l ∈ τ:
                // literal is already falsified
                return conflict
            case l ∈ τ:
                // literal is already satisfied
                skip;
            otherwise do
                // literal is yet unassigned
                τ ← τ ∪ {l} // add assignment to the trail
                if propagate() ≠ ok then
                    return conflict;
        notify_assignments();
        l ← cb_external_propagate();
    return ok
```

Algorithm 3: Addition of external clauses during the CDCL loop (simplified).

```
Function add_external_clauses()
    notify_assignments ();
    has_clause, is_forgettable ← cb_has_external_clause();
    while has_clause do
        C ← get_external_clause();
        switch C do
            case root-satisfied(C):
                skip;
            case falsified(C):
                return conflict;
            otherwise do
                if propagate() ≠ ok then
                    return conflict;
        notify_assignments();
        has_clause, is_forgettable ← cb_has_external_clause();
    return ok
```

start. Otherwise, the solver must call `BCP` to check for conflicts. External clause addition ends when either a given clause is falsified or the user has no more clauses to add.

Algorithm 4 shows in a simplified way how `cb_decide` can be intertwined with the internal decision process of a SAT solver. It is important to note that no external decision is allowed to take place as long as there are unassigned assumptions of the current SAT query. If all assumptions are already satisfied by the current trail, the solver can ask the external propagator for a decision literal (after notifying about the current state of the trail). If the propagator has nothing to decide, or if the given literal is already assigned, the solver can fall back on its own decision heuristic (function `decide` in Algorithm 4). Once a decision is made, regardless of the source of the decision, the propagator must be notified of the start of the new decision level by calling `notify_new_decision_level`.

Algorithm 4: Decision making with the external propagator (simplified).

```
Function external_decide()
    Data: The current set of assumptions A, with A \ τ the yet unsatisfied subset.
    if ∃a ∈ A \ τ: then
        // First all assumptions must be satisfied
        l ← a;
    else
        // Ask external propagator for a decision
        notify_assignments();
        l ← cb_decide(); // Returns 0 when no external decision was made
        if l = 0 ∨ l ∈ τ ∨ ¬l ∈ τ then
            // Ignore external decision if literal is already assigned
            l ← decide();
    notify_new_decision_level();
    τ ← τ ∪ l;
    notify_assignments();
```

3. Combining External Propagation with Features of SAT Solvers

In our SAT paper, we presented a proof-of-concept implementation of our interface that provides all the basic functionalities outlined above. Making external propagation compatible with all of the complex features of modern SAT solvers is challenging, but many of the common techniques can be supported relatively easily. In this section, we provide our insights, ideas and some open questions on how to combine IPASIR-UP with some of these common CDCL SAT features.

3.1 Clause Database Reduction

One of the distinguishing features of CDCL SAT solvers, compared to DPLL, is the use of clause learning upon conflict analysis (Silva & Sakallah, 1996). Such learned clauses are redundant, since they are derived from the original input problem by resolution. Thus, neither learning nor removing these clauses will change the satisfiability of the problem (but may make solving easier). However, allowing learned clauses to accumulate indefinitely can slow down the search process. To avoid this, solvers periodically delete a subset of these

redundant clauses. In contrast, the user-provided input clauses define the actual problem. Consequently, they are not allowed to be deleted unless they are *implied* (e.g., subsumed) by other input clauses. Thus, input (also called *irredundant*) and learned (also called *redundant*) clauses are distinguished and treated differently in the SAT solver.

When clauses are added via the external propagator during solving, the SAT solver does not know whether they should be considered as learned or input clauses—they could either be user-provided irredundant clauses or clauses learned by an external reasoning engine (e.g., a theory solver in SMT). Assuming that the external propagator is consistent and remains connected until the problem is solved, the SAT solver may be allowed to delete redundant clauses since the external propagator can always derive them again if necessary. However, if it is more expensive to repeatedly derive a clause than to remember it, it is undesirable to delete it. In most situations, only the user can determine whether this is the case, and thus IPASIR-UP allows to specify whether an external clause should be kept or may be forgotten (see parameter `is_forgettable` of `cb_has_external_clause` and the Boolean flag `are_reasons_forgettable` in Listing 2).

Forgettable external clauses are considered for deletion, just like the internal redundant clauses. Non-forgettable clauses are removed only if they are redundant w.r.t. the conjunction of the input and non-forgettable external clauses (like the internal irredundant clauses). However, it depends on the heuristics of the SAT solver's clause database management when exactly a forgettable clause is deleted. Currently, users cannot force clause deletion or get notified when certain external clauses are deleted. It remains future work to investigate whether there are use cases where these steps would be beneficial.

3.2 Finding Fixed Assignments

Depending on the use case, it can often be beneficial for users to know when an assignment made by the SAT solver will never change in the future (i.e., the assignment is fixed). This allows users to perform further optimizations to simplify their own constraints and problem representation based on these fixed assignments.

To support such use cases, our first proposed version of IPASIR-UP classified explicitly each assignment made by the SAT solver as either *fixed* or *non-fixed*. However, this forced users to care about fixed assignments, even in applications where it added no value. Therefore, we refined our design such that the current notification feature of IPASIR-UP simply reports (un-)assignments, without an explicit flag indicating fixed variables assignments.

However, for those users who are interested in fixed assignments, we have introduced an alternative interface class (called `FixedAssignmentListener` in CaDiCaL) that eagerly sends notifications when *any* of the variables are fixed. Since this feature may be useful even in use cases where no external propagator is required, its class is independent from the propagator, and thus does not restrict notifications to those variables that are relevant to the external propagator. Supporting this interface is therefore completely optional, and users and developers can decide if it is worth the development effort for their use case.

3.3 Inprocessing and Solution Reconstruction

Our interface IPASIR-UP enables a more fine-grained way of incremental SAT solving, where new clauses may be added not only between two `solve` calls, but also during solv-

ing. Ways to combine inprocessing with incremental clause addition was proposed, e.g., by (Fazekas et al., 2019) and (Nadel et al., 2012). However, both techniques assumed that many clauses are added all at once between each solve call.

It is not hard to see that the `RestoreAddClauses` method of (Fazekas et al., 2019) can be used *during* solving, using the same tainting and cleanness check as in (Fazekas et al., 2019). This allows the solver to recognize when a clause added by the external propagator triggers the undoing of some of the previous clause elimination steps. In such cases, the restored clauses can be added along with the external clause. Note that traversing the entire reconstruction stack each time the external propagator provides clauses may be too costly. It remains intriguing future work to address this technical challenge.

However, there is another issue with combining inprocessing SAT solving with an external propagator. Some of the clause elimination techniques used in inprocessing (such as blocked clause elimination) only preserve satisfiability, while potentially introducing additional models to the problem. These techniques require an extra *solution reconstruction* step once a satisfying solution to the simplified formula is found. This step, however, not only extends the found solution with further assignments (e.g., by assigning a value to the completely eliminated variables) but may also change the truth value of some variables that were assigned previously. The variables that have been flipped may be on any decision level in the trail. Since the external propagator has a stack-like view on this trail (see Sect. 2.2), users cannot unassign and reassign them without backtracking to their corresponding level.

Further investigation is required to evaluate the practical overhead of these flips during solution reconstruction. Currently, we assume that observed variables are internally frozen, and whenever a variable is added via `add_observed_var`, it is clean w.r.t. the reconstruction stack. This ensures that no restore step is necessary when external clauses are added and no variable assignments are flipped in found solutions.

3.4 Decision Heuristics

The IPASIR-UP interface enables users to influence the decision heuristics of SAT solvers. Users can suggest a decision via the `cb_decide` callback function, which will then be executed instead of the solver's own decision. Further, the `phase` and `unphase` functions allow the user to share additional knowledge about variables with the solver. Note, however, that overwriting the decision heuristics of a SAT solver carries a high risk of ruining its performance. Nevertheless, there are use cases, e.g. when the solver is used as a simple unit propagation engine, where these functionalities are very helpful.

However, information flow in the opposite direction is currently unavailable. Currently, there is no standardized method for users to access the score or order of variables in the solver. Standardizing access to this information with low overhead remains an open problem that may warrant further investigation in the future.

3.5 The Lack of Backtracking

Backtracking is a standard feature of CDCL SAT solvers that modifies their trail. In certain situations, such as when a conflict arises or a restart is scheduled, backtracking is expected. If an external propagator is connected, it should be notified of a backtracking event to

maintain an up-to-date view of the current trail (see Section 2.2). However, some of the new features of SAT solvers make these interactions with the propagator more complex.

As previously mentioned, assignments that are out of order (e.g., due to chronological backtracking, see Section 2.2) or that are changed later (e.g., due to solution reconstruction, see Section 3.3) require additional attention to ensure proper notification while maintaining a stack-like external view of the trail. The simplest solution is to report a backtrack to the lowest decision level where a change occurred and re-notify every decision level starting from there. This allows the SAT solver to maintain its behaviour while the external propagator rebuilds its stack-like trail with the changes. Alternatively, the trail can be used as a stack within the SAT solver. For instance, out-of-order assignments can be reassigned after each backtrack as long as the backtracked level is higher than their actual assignment level.

Techniques such as incremental lazy backtracking (ILB) (Nadel, 2022) and trail reuse (van der Tak et al., 2011) aim to prevent repeated work by identifying when parts of the trail can be reused from the previous search. If a prefix of the previous trail is reused, it can be easily communicated to the propagator by backtracking to the last kept decision level. Otherwise, the previously described dedicated stack-like notification of the changes must be implemented.

Generally, users do not need to be aware of the incremental techniques used by a SAT solver to use them correctly. However, when an external propagator is connected, some insights can be helpful. For instance, users may clean up their propagator between two solve calls, while the solver may not necessarily restart the search due to ILB. Such uncommunicated expectations are error-prone and can lead to time-consuming debugging. Insightful users can also determine when it is not beneficial to use some techniques. The necessary backtrack and reassign steps of the stack-like view can be costly for the user. Therefore, it is important to weigh the benefits of these incremental techniques on the SAT side against the costs on the external propagator's side.

3.6 Proof Production

The certification of the returned answers is essential for SAT solvers that are used in safety-critical systems or as a sub-reasoning engine in tools of formal methods. This feature is supported by every modern SAT solver in non-incremental stand-alone applications. However, in incremental use cases, the current standard proof generation and checking techniques may not suffice (Kiesl-Reiter & Whalen, 2023). For example, during incremental inprocessing there is a distinction between 'deleting' or 'weakening' a clause (Fazekas et al., 2019), which cannot be expressed in the current standard proof formats of SAT solvers. In our recent works (Fazekas et al., 2024c, 2024b) we proposed systematic approaches that support efficient certification and checking of incremental inprocessing SAT solving. Moreover, we proposed ICNF (Fazekas et al., 2024c), an extension of DIMACS, to capture interactions and incremental SAT queries in a single file. These novel interaction files and proof formats can capture clause addition steps of the user propagator during the CDCL loop as well. Therefore, certifying and verifying the use cases of IPASIR-UP is relatively simple.

However, there may be complications due to the difference of forgettable and propositionally redundant clauses in the presence of an external propagator (see Sect. 3.1). If a user defines a clause as forgettable and a solver decides to delete it, it will appear in the proof

as a normal deletion step of a redundant clause. Current proof checkers for SAT solvers do not verify the correctness of such clause deletion steps since they cannot turn a satisfiable problem into an unsatisfiable one. Therefore, the current checking techniques allow for such deletion steps, even if the removed clause is not necessarily redundant in propositional logic. However, other proof systems, such as veriPB (Bogaerts et al., 2022), may wish to verify clause deletion steps as well. In these cases, it will be necessary to distinguish between externally forgettable clauses and real redundant ones in the produced proofs.

4. Related Work

The primary objective of incremental reasoning is to reuse previously learned information when solving similar problems. Incremental SAT solvers have a long history (Hooker, 1993; Kim et al., 2000; Whittemore et al., 2001; Eén & Sörensson, 2003a, 2003b) with numerous improvements in the last decade (e.g., Kupferschmid et al., 2011; Nadel & Ryvchin, 2012; Nadel et al., 2012; Audemard et al., 2013; Nadel et al., 2014; Hickey & Bacchus, 2019). IPASIR (Balyo et al., 2016) is a universal C interface for incremental SAT solvers, allowing easy integration of incremental SAT solvers into applications, without the need to specialize in a specific SAT solver. IPASIR-UP extends IPASIR for use cases that require more fine-grained interaction between the application and the SAT solver during solving. Not only does it provide the user with more comprehensive access to information about the solver state during solving, but it also allows the user to influence and guide the solver's behaviour based on user-level information that is not available to the SAT solver.

The proposed interface standardises functionality that is already used by various applications. The CDCL($\mathcal{T}$) framework (Nieuwenhuis et al., 2006) for SMT solvers is an important use case of interacting with the SAT solver as described above. Current state-of-the-art SMT solvers, such as CVC5 (Barbosa et al., 2022), Z3 (de Moura & Bjørner, 2008), MathSat (Cimatti et al., 2013), OpenSMT (Bruttomesso et al., 2010), and veriT (Bouton et al., 2009) all implement a custom interaction layer with the SAT solver (see, e.g., the SAT worker interface described in (Cimatti et al., 2013)). Recently, in the context of SMT, (Bjørner et al., 2023) introduced an interface to seamlessly integrate theory solvers into Z3. This enables extending the solving capabilities of Z3 beyond its default features (Eisenhofer et al., 2023). However, it does not facilitate the replacement of the internal SAT solver.

The desire to combine SAT solving with external reasoning engines is not new. The Lynx SAT solver already provided a programmatic callback interface a decade ago (Ganesh et al., 2012). It allowed users to add new clauses, based on the current partial solution, during the search. Following that work, the programmatic SAT schema was exploited many more times, for example in MathCheck (Zulkoski et al., 2016; Bright et al., 2016), or later in other methods that combine SAT with computer algebra systems (see e.g. (Bright et al., 2019, 2020, 2020), and (Bright et al., 2021)). Another extended SAT solver was implemented in (Erez & Nadel, 2015), where users could *interact* with the solver to combine and guide the search with graph-aware reasoning. A more recent tool, the IntelSAT solver (Nadel, 2022), implements efficient clause addition at arbitrary decision levels, while using *reimplication* to guarantee that no implications are missed at lower decision levels. However, none of these tools support lazy propagation explanation, or stack-like notifications.

The state-of-the-art ASP solver clingo (Gebser et al., 2016) provides a generic interface to augment the tool with *theory propagators*. It extends the CDCL loop at four locations, with notifications and the ability to add clauses during the search and upon checking the found model. It does not, however, support lazy propagation explanation (i.e, `cb_propagate` with delayed clause addition) nor proof generation.

5. Empirical Evaluation

To show that our interface is effective and efficient in varied use cases, we extended CaDiCaL (Biere et al., 2020, 2024), a state-of-the-art incremental SAT solver which implements the IPASIR interface, with IPASIR-UP. Our extension required ∼800 lines of C++ code in CaDiCaL, accompanied with another ∼700 lines in its model based tester. We provide an evaluation on two representative use cases: enumerating graphs with certain properties via SAT Modulo Symmetries (Kirchweger & Szeider, 2021), and integrating CaDiCaL as the main CDCL($\mathcal{T}$) SAT engine in the SMT solver cvc5 (Barbosa et al., 2022). All evaluated tools of our experiments are available at Zenodo (Fazekas et al., 2024a).

5.1 Experiments with SMS

SAT modulo Symmetries (SMS) (Kirchweger & Szeider, 2021, 2024) is a recently introduced SAT-based framework for the exhaustive generation of graphs with a given property while excluding isomorphic copies of the same object (isomorph-free). The framework was extended to matroids (Kirchweger et al., 2022), hypergraphs (Kirchweger et al., 2023b), planar graphs (Kirchweger et al., 2023) and was utilized for improving the known lower-bound on a smallest Kochen-Specker vector system (Kirchweger et al., 2023a) and for computing universal graphs (Zhang & Szeider, 2023). In contrast to a generate-and-test approach, which quickly becomes infeasible due to the extremely fast-growing number of candidate objects, SMS directly generates isomorph-free objects with the desired property. At its core, SMS runs a CDCL solver on a propositional formula that encodes the desired property using object variables.

For instance, if the object is a graph, the graph property is expressed using variables $e_{u,v}$ for each vertex pair u, v indicating existence of an edge between u and v. Isomorphic copies are avoided by guiding the solver to generate canonical objects, e.g., by requiring the adjacency matrix to be lexicographically minimal. There is no known complete symmetry breaking which isn't exponential in the number of clauses, even if other canonical forms than being lexicographically minimal are used (Codish et al., 2016; Itzhakov & Codish, 2020; Heule, 2019). Hence SMS delegates the *minimality check* to an external algorithm invoked whenever the SAT solver decides on an object variable. SMS can perform the minimality check even when many object variables are undecided. This check tests if a minimal object is consistent with the current partial truth assignment. A symmetry-breaking clause is sent back to the CDCL solver if the check fails.

In previous work, SMS used clingo (Gebser et al., 2016), an ASP solver with support for adding custom propagators. The IPASIR-UP interface enables us to replace clingo in SMS with CaDiCaL. We use `cb_has_external_clause` to indicate if we have a symmetry-breaking clause to add and `cb_propagate` to propagate literals. To exhaustively generate all isomorph-free objects with the given property, we add a clause forbidding each object

Table 1: Enumerating up to isomorphism: all graphs (top), claw-free, cubic, connected graphs (mid), and all KS candidates (bottom).

			CaDiCaL +IPASIR-UP [s]					**Clingo [s]**	
	n	#graphs	*default*	*enum-IPASIR*	*no-inpro*	*forgettable*	*no-prop*	*clingo-red*	*clingo-irred*
All graphs	6	156	0.01	0.01	0.01	0.01	0.01	0.01	0.01
	7	1044	0.06	0.08	0.05	0.05	0.05	0.06	0.05
	8	12346	0.52	0.78	0.53	0.55	0.52	0.63	0.63
	9	274668	13.73	23.76	13.86	14.93	14.06	51.36	47.85
	10	12005168	3916.74	3102.31	3260.63	3758.26	3702.24	165697.81	150790.36
CCC graphs	34	1502	24.26	31.15	26.51	28.26	26.29	255.32	50.06
	36	3187	50.83	63.67	52.04	62.52	48.91	271.87	106.46
	38	6946	94.53	112.04	101.59	118.24	103.52	564.13	197.70
	40	15025	210.04	243.07	210.99	254.44	209.06	1668.35	519.69
	42	33687	500.21	574.40	522.62	594.37	504.57	6378.04	1531.99
	44	77450	1082.26	1275.57	1154.77	1274.17	1132.15	13658.87	5687.90
	46	177465	2599.75	3269.27	2776.63	3225.35	2604.54	26874.47	15926.36
	48	418112	7433.26	9087.24	7487.61	7951.47	7869.27	167149.83	75481.11
KS cand.	16	0	5.97	5.57	4.36	4.35	3.89	10.88	8.02
	17	1	16.51	16.96	13.91	13.27	19.14	49.04	37.68
	18	0	81.06	77.52	57.96	89.52	87.89	354.49	310.62
	19	8	493.16	580.32	478.22	549.39	529.54	3737.21	3598.72
	20	147	5574.00	7237.05	7036.95	6047.77	6685.28	58076.57	50048.51

found so far. We can do this via the standard IPASIR interface or IPASIR-UP using callback `cb_check_found_model`.

In the following, we compare the performance of SMS between CaDiCaL +IPASIR-UP and clingo on three graph generation tasks. The first task is to generate up to isomorphism all graphs for a given number n of vertices without additional restrictions, i.e., the formula describing the graph is empty. The second task is to generate all connected, cubic, claw-free graphs up to isomorphism. A graph is cubic if all vertices have degree 3 and a graph is claw-free if it doesn't contain a claw (a complete bipartite graph $K_{1,3}$) as an induced subgraph. We use auxiliary non-object variables to ensure that the graph is cubic and for each possible embedding of the $K_{1,3}$ in the resulting graph we add a clause ensuring that it is not the case. Another SAT-based approach using static symmetry breaking was used to enumerate all up to 36 vertices (Itzhakov & Codish, 2023). With SMS using the new IPASIR-UP interface we are able to enumerate all connected, cubic, claw-free graphs up to 48 vertices. The third task is to generate up to isomorphism all non-010-colorable graphs with a minimum degree of at least three not containing a cycle of length 4. A graph is 010-colorable if the vertices can be colored with 0 and 1 such that there is no monochromatic edge with color 1 and no monochromatic triangle with color 0. These graphs are interesting for topics related to the famous Kochen-Specker Theorem from quantum mechanics (Arends et al., 2011). For encoding the non-010-colorability, we follow previous work (Li et al., 2022). In contrast to the first task, the encoding is relatively large, even exponential in the number of vertices.

For CaDiCaL, the *default* configuration propagates literals, makes the symmetry breaking clauses not forgettable, and exhaustively enumerates all graphs using the IPASIR-UP interface when possible. Configuration *enum-IPASIR* propagates literals and adds symmetry-

breaking clauses via IPASIR-UP but uses IPASIR for solution enumeration. Configuration *no-prop* corresponds to *default* without propagating literals but learning the clause immediately. Configuration *no-inpro* corresponds to *default* without inprocessing on the non-observed variables (i.e., option *inprocessing* of CaDiCaL is turned off). Configuration *forgettable* corresponds to *default* but clauses added via the IPASIR-UP interface are forgettable. For clingo, we either add the clauses as redundant (configuration *red*), i.e., the symmetry-breaking clauses are part of the clause-deletion policy, or the clauses are irredundant (configuration *irred*).

Table 1 summarizes the results given the number of vertices in column n. The number of generated graphs is given in column #graphs. All experiments with SMS ran on a cluster equipped with Intel Xeon E5-2640v4 CPUs at 2.40 GHz.

For enumeration, the new interface gives a speedup over IPASIR for most instances: with IPASIR, the search is started at the root level after a model has been found, while with IPASIR-UP, the current trail is preserved and backtracked. CaDiCaL outperforms Clingo in all the applications shown here. On other SMS applications, we observed Clingo and CaDiCaL performing similarly. However, CaDiCaL with IPASIR-UP shows the potential to solve problems outside the other solver's reach.

5.2 Experiments with SMT

Satisfiability Modulo Theories (SMT) solvers serve as the back-end reasoning engine for a variety of applications (e.g., (Leino, 2010; Alur et al., 2013; Cadar et al., 2008; Niemetz et al., 2018; Godefroid et al., 2012; Backes et al., 2020)). The majority of state-of-the-art SMT solvers are based on the CDCL($\mathcal{T}$) framework (Nieuwenhuis et al., 2006), which tightly integrates theory solvers with a CDCL SAT solver at its core. The CDCL($\mathcal{T}$) SAT engine is queried to find a satisfying assignment of the propositional abstraction of the input formula, which is then iteratively refined until either the assignment is $\mathcal{T}$-consistent or the SAT engine determines unsat.

The CDCL($\mathcal{T}$) framework requires a tight integration with the SAT solver in a way that allows the theory layer to interact with the SAT solver during search, i.e., in an *online* fashion. This is in contrast to other lazy SMT approaches based on the same abstraction/refinement principle that integrate a SAT solver as a *black box*, e.g., lemmas on demand (Barrett et al., 2002; Moura & Rueß, 2002). That is, rather than querying the SAT solver for a full satisfying assignment of the propositional abstraction, the theory layer guides the search of the SAT solver until a $\mathcal{T}$-consistent assignment is found or the formula becomes unsatisfiable.

Further, throughout this process, a backward communication channel allows the SAT solver to notify the theory layer about variable assignments, decisions, and backtracks. The theory layer uses this information to derive conflicts, propagate theory literals, or suggest decision variables based on theory-guided heuristics. If theory propagations are involved in deriving a conflict in the SAT solver, the theory layer must provide explanations for the propagated theory literals. If a partial assignment of the propositional abstraction is $\mathcal{T}$-inconsistent, the theory layer sends a lemma to the SAT solver to refine the abstraction.

CVC5 is a state-of-the-art CDCL($\mathcal{T}$) SMT solver widely used in industry and academic projects (Barbosa et al., 2022). It relies on a highly customized version of MiniSat (Eén &

Table 2: Number of solved instances and runtime on non-incremental SMT-LIB benchmarks with a 300 seconds time limit.

	CVC5		CVC5-IPASIRUP	
Division	solved	time [s]	solved	time [s]
Arith (6,931)	6,361	175,490	**6,362**	177,677
BitVec (6,185)	**5,656**	164,392	5,635	170,636
Equality (12,159)	5,347	2,053,167	**5,351**	2,051,612
Equality+LinearArith (56,562)	**45,987**	3,183,536	45,972	3,187,670
Equality+MachineArith (10,912)	1,077	2,956,148	**1,086**	2,955,064
Equality+NonLinearArith (21,450)	**13,491**	2,437,631	13,488	2,445,284
FPArith (3,979)	3,169	262,984	**3,171**	262,844
QF_Bitvec (46,191)	43,868	932,547	**43,878**	931,884
QF_Datatypes (8,903)	8,300	197,540	**8,363**	175,047
QF_Equality (8,054)	8,048	4,633	8,048	4,004
QF_Equality+Bitvec (16,809)	15,943	339,387	**16,167**	232,135
QF_Equality+LinearArith (3,644)	3,487	55,058	**3,536**	38,944
QF_Equality+NonLinearArith (1,118)	**773**	113,295	772	111,708
QF_FPArith (76,252)	**76,117**	67,160	76,111	61,681
QF_LinearIntArith (16,469)	12,122	1,466,880	**12,599**	1,332,706
QF_LinearRealArith (2,008)	1,799	103,513	**1,870**	72,126
QF_NonLinearIntArith (25,446)	13,636	3,823,539	**14,360**	3,517,103
QF_NonLinearRealArith (12,154)	11,234	308,073	**11,323**	277,400
QF_Strings (103,405)	99,936	1,193,661	**100,368**	1,096,961
Total (438,631)	376,351	19,838,643	**378,460**	19,102,495

Sörensson, 2003a) as its core SAT engine, which was extended to support the production of resolution proofs, pushing and popping of assertion levels, and custom theory-guided decision heuristics. The interaction with CVC5's theory layer is directly implemented in MiniSat by various callbacks.

These customizations make it difficult to replace CVC5's version of MiniSat with a state-of-the-art SAT solver to take advantage of improvements in SAT solving. However, replacing this customized version of MiniSat with a SAT solver that implements IPASIR-UP enables us to easily switch it out with any other solver that implements the interface. It has the additional advantage that the interaction with the SAT layer is standardized and clean. That is, no CDCL($\mathcal{T}$)-specific modifications of the SAT solver, which may accidentally impact performance, are required.

We integrated CaDiCaL with the IPASIR-UP extension as main CDCL($\mathcal{T}$) SAT engine of CVC5 while fully utilizing the IPASIR-UP notification and callback interface: `notify_assignment` is used to construct the current partial assignment for the observed theory literals; the incremental solver state of CVC5 is managed via `notify_new_decision_level` and `notify_backtrack`, which are utilized to restore its internal state when backtracking decisions; `cb_propagate` and `cb_add_reason_clause_lit` are used for theory propagations and explanations; `cb_decide` to implement custom decision heuristics; `cb_add_external_clause_lit` for adding lemmas and conflicts; and `cb_check_found_model` to check whether

Table 3: Number of solved satisfiability queries and runtime on incremental SMT-LIB benchmarks with a 300 seconds time limit. Numbers (i/q) show the total number of instances i and the total number of satisfiability queries q for these instances.

	CVC5		CVC5-IPASIRUP	
Division	solved	time [s]	solved	time [s]
Arith (11/41,362)	41,362	88	41,362	104
BitVec (18/38,856)	36,125	2,565	36,125	2,572
Equality (4,067/711,896)	**46,586**	592,148	46,574	593,739
Equality+LinearArith (1,894/1,346,978)	435,916	29,787	**436,107**	30,834
Equality+MachineArith (4/2,269)	818	302	818	303
Equality+NonLinearArith (4,374/739,582)	89,345	615,636	**91,082**	609,840
FPArith (10/6,055)	**3,422**	1,826	3,421	1,828
QF_Bitvec (2,590/53,684)	**51,526**	55,030	51,477	56,490
QF_Equality (1,778/29,990)	29,986	2,484	29,986	2,517
QF_Equality+Bitvec (3,633/9,870)	**7,351**	138,470	7,323	146,137
QF_Equality+Bitvec+Arith (1,710/34,795)	33,403	59,159	**33,418**	51,784
QF_Equality+LinearArith (3,947/6,039,485)	**3,693,170**	106,937	2,400,530	113,273
QF_Equality+NonLinearArith (1,018/164,755)	104,275	14,038	**105,398**	13,683
QF_FPArith (19,188/588,878)	552,699	757,344	**567,287**	616,019
QF_LinearIntArith (69/20,041,214)	**2,325,341**	17,562	1,521,214	16,172
QF_LinearRealArith (10/1,515)	515	3,003	**700**	2,652
QF_NonLinearIntArith (12/4,219,215)	**771,648**	3,603	479,184	3,603
Total (44,333/34,070,399)	**8,223,488**	2,399,991	5,852,006	2,261,556

the SAT assignment is $\mathcal{T}$-satisfiable. CVC5 further uses `phase` to set the phase for specific variables, and `is_decision` to query if a specific variable was used to make a decision.

The full integration of CaDiCaL as CDCL($\mathcal{T}$) SAT engine of CVC5 required about 600 lines of C++ code on top of CVC5 1.1.1. In the following, we refer to this version of CVC5 with CaDiCaL as the CDCL($\mathcal{T}$) SAT engine as CVC5-IPASIRUP. Note that both configurations utilize a CaDiCaL instance that is independent from the CDCL($\mathcal{T}$) SAT engine as the SAT solver backend of the bit-vector theory solver, which implements bit-blasting. Further note that proof production (Barbosa et al., 2022, 2023) is not yet supported in CVC5-IPASIRUP, since this requires an extension of the proof infrastructure of CVC5 to support DRAT proofs (MiniSat was customized to emit resolution proofs).

We evaluate the overall performance of CVC5-IPASIRUP against CVC5 version 1.1.1 on all non-incremental and incremental benchmarks of the 2024 release of SMT-LIB (Preiner et al., 2024a, 2024b). We ran this experiment on a cluster equipped with AMD Ryzen 9 7950X CPUs and allocated one CPU core, 8GB of RAM and a time limit of 300 seconds for each solver and benchmark pair (unknown answers were treated as timeouts). Table 2 shows the number of solved benchmarks and runtime on non-incremental divisions, and Table 3 gives the number of solved satisfiability queries and runtime in incremental divisions in SMT-LIB. The results in both tables are grouped into the divisions defined in SMT-COMP 2024 (Bobot et al., 2024).

On the non-incremental benchmarks, CVC5-IPASIRUP solves 2,109 more benchmarks and improves over CVC5 in 13 out of 19 divisions overall. On the 373,311 commonly solved

benchmarks, CVC5-IPASIRUP (692,554s) is 1.43× faster than CVC5 (990,307s). On commonly solved quantifier-free (quantified) instances, CVC5-IPASIRUP improves in terms of runtime by a factor of 1.48× (1.08×). For quantifier-free divisions, CVC5-IPASIRUP significantly improves in terms of solved instances over CVC5 in arithmetic divisions (+1,361), in QF_Strings (+432) and in QF_Equality+BitVec (+224), which contains logics that combine bit-vectors with arrays. In quantified divisions, CVC5-IPASIRUP's performance is comparable to CVC5, solving 23 less benchmarks overall. Interestingly, CVC5-IPASIRUP exceeds the memory limit on 91 (62) more quantifier-free (quantified) instances than CVC5.

On the incremental benchmarks, CVC5-IPASIRUP improves over CVC5 in 6 out of 17 divisions. However, overall CVC5-IPASIRUP solves 29% less incremental satisfiability queries. The majority of these unsolved queries are in the arithmetic divisions QF_Equality+LinearArith, QF_LinearIntArith and QF_NonLinearIntArith. All of these divisions contain benchmarks with a large number of satisfiability queries, some with over one million queries. One of the main differences of CVC5-IPASIRUP and CVC5 on incremental benchmarks is that the customized version of MiniSat in CVC5 has native push/pop support for adding and deleting clauses in the SAT solver. In CVC5-IPASIRUP, push/pop is implemented via activation literals and solving under assumptions. Preliminary experiments showed that activation literals can have a significant performance overhead compared to native push/pop support, even for simple benchmarks that contain a large number of incremental queries. We believe that this is the main reason for CVC5-IPASIRUP's performance degradation. This is rather surprising since activation literals are a widely used approach in incremental SAT solving. As future work, we will investigate ways to improve activation literals for this use case. If this is not successful we intend to add native push/pop support in CaDiCaL.

6. Summary and Future Work

In this paper, we proposed an extension of the IPASIR interface of SAT solvers to facilitate interactions with the solver *during* the search. We demonstrated the usage and benefits of such an interface in two representative use cases. However, to enable all functionalities of modern SAT solvers, some restrictions were introduced. For example, to enable inprocessing, external clauses can have only observed (i.e., frozen) variables.

We believe that both developers of more complex reasoning tools and end-users of SAT solvers can strongly benefit from a unified interface that provides access and control over the details of CDCL methods during incremental problem solving. Though the proposed IPASIR-UP interface provides a sufficient set of functions to cover a very wide range of applications, there are many possible extensions to consider in the future. We hope that further discussions and further use cases of IPASIR-UP, for instance in MaxSAT, knowledge compilation or in QBF reasoning, will make it clear what kind of extensions and refinements would be the most practical to implement.

Acknowledgements. The authors would like to thank the organizers, visitors and participants of the program "Satisfiability: Theory, Practice, and Beyond" at the Simons Institute for the Theory of Computing for their many valuable comments and fruitful discussions. We also thank Mathias Fleury and Florian Pollitt for the several discussions about CaDiCaL.

This work was supported in part by the Stanford Center for Automated Reasoning, the Stanford Center for Blockchain Research, a gift from Amazon Web Services, by the Austrian

Science Fund (FWF) under projects No. T-1306 and P-32441, and by the Vienna Science and Technology Fund (WWTF) under project No. ICT19-065 (Reveal-AI).

References

Alur, R., Bodík, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., & Udupa, A. (2013). Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 1–8. IEEE.

Arends, F., Ouaknine, J., & Wampler, C. W. (2011). On searching for small Kochen-Specker vector systems. In Kolman, P., & Kratochvíl, J. (Eds.), *Graph-Theoretic Concepts in Computer Science - 37th International Workshop, WG 2011, Teplá Monastery, Czech Republic, June 21-24, 2011. Revised Papers*, Vol. 6986 of *LNCS*, pp. 23–34. Springer.

Audemard, G., Lagniez, J., & Simon, L. (2013). Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In Järvisalo, M., & Gelder, A. V. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, Vol. 7962 of *Lecture Notes in Computer Science*, pp. 309–317. Springer.

Bacchus, F., Järvisalo, M., & Martins, R. (2021). Maximum satisfiabiliy. In Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 929–991. IOS Press.

Backes, J., Berrueco, U., Bray, T., Brim, D., Cook, B., Gacek, A., Jhala, R., Luckow, K. S., McLaughlin, S., Menon, M., Peebles, D., Pugalia, U., Rungta, N., Schlesinger, C., Schodde, A., Tanuku, A., Varming, C., & Viswanathan, D. (2020). Stratified abstraction of access control policies. In Lahiri, S. K., & Wang, C. (Eds.), *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, Vol. 12224 of *Lecture Notes in Computer Science*, pp. 165–176. Springer.

Balyo, T., Biere, A., Iser, M., & Sinz, C. (2016). SAT race 2015. *Artif. Intell.*, *241*, 45–65.

Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., & Zohar, Y. (2022). cvc5: A versatile and industrial-strength SMT solver. In Fisman, D., & Rosu, G. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, Vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442. Springer.

Barbosa, H., Barrett, C. W., Cook, B., Dutertre, B., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Tinelli, C., & Zohar, Y. (2023). Generating and exploiting automated reasoning proof certificates. *Commun. ACM*, *66*(10), 86–95.

Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., & Barrett, C. W. (2022). Flexible proof production in an industrial-strength SMT solver. In Blanchette, J., Kovács, L., & Pattinson, D. (Eds.), *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, Vol. 13385 of *Lecture Notes in Computer Science*, pp. 15–35. Springer.

Barrett, C. W., Dill, D. L., & Stump, A. (2002). Checking satisfiability of first-order formulas by incremental translation to SAT. In Brinksma, E., & Larsen, K. G. (Eds.), *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, Vol. 2404 of *Lecture Notes in Computer Science*, pp. 236–249. Springer.

Barrett, C. W., Sebastiani, R., Seshia, S. A., & Tinelli, C. (2021). Satisfiability modulo theories. In Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 1267–1329. IOS Press.

Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., & Pollitt, F. (2024). Cadical 2.0. In Gurfinkel, A., & Ganesh, V. (Eds.), *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, Vol. 14681 of *Lecture Notes in Computer Science*, pp. 133–152. Springer.

Biere, A., Fazekas, K., Fleury, M., & Heisinger, M. (2020). CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., & Suda, M. (Eds.), *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, Vol. B-2020-1 of *Department of Computer Science Report Series B*, pp. 51–53. University of Helsinki.

Bjørner, N. S., Eisenhofer, C., & Kovács, L. (2023). Satisfiability modulo custom theories in Z3. In Dragoi, C., Emmi, M., & Wang, J. (Eds.), *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston, MA, USA, January 16-17, 2023, Proceedings*, Vol. 13881 of *Lecture Notes in Computer Science*, pp. 91–105. Springer.

Bobot, F., Bromberger, M., & Jonás, M. (2024). The International Satisfiability Modulo Theories Competition (SMT-COMP)..

Bogaerts, B., Gocht, S., McCreesh, C., & Nordström, J. (2022). Certified symmetry and dominance breaking for combinatorial optimisation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022 Virtual Event, February 22 - March 1, 2022*, pp. 3698–3707. AAAI Press.

Bouton, T., Oliveira, D. C. B. D., Déharbe, D., & Fontaine, P. (2009). veriT: An open, trustable and efficient SMT-solver. In Schmidt, R. A. (Ed.), *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, Vol. 5663 of *Lecture Notes in Computer Science*, pp. 151–156. Springer.

Bright, C., Cheung, K. K. H., Stevens, B., Kotsireas, I. S., & Ganesh, V. (2020). Nonexistence certificates for ovals in a projective plane of order ten. In Gasieniec, L., Klasing,

R., & Radzik, T. (Eds.), *Combinatorial Algorithms - 31st International Workshop, IWOCA 2020, Bordeaux, France, June 8-10, 2020, Proceedings*, Vol. 12126 of *Lecture Notes in Computer Science*, pp. 97–111. Springer.

Bright, C., Dokovic, D. Z., Kotsireas, I. S., & Ganesh, V. (2019). The SAT+CAS method for combinatorial search with applications to best matrices. *Ann. Math. Artif. Intell.*, *87*(4), 321–342.

Bright, C., Ganesh, V., Heinle, A., Kotsireas, I. S., Nejati, S., & Czarnecki, K. (2016). Mathcheck2: A SAT+CAS verifier for combinatorial conjectures. In Gerdt, V. P., Koepf, W., Seiler, W. M., & Vorozhtsov, E. V. (Eds.), *Computer Algebra in Scientific Computing - 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*, Vol. 9890 of *Lecture Notes in Computer Science*, pp. 117–133. Springer.

Bright, C., Kotsireas, I. S., & Ganesh, V. (2020). Applying computer algebra systems with SAT solvers to the williamson conjecture. *J. Symb. Comput.*, *100*, 187–209.

Bright, C., Kotsireas, I. S., Heinle, A., & Ganesh, V. (2021). Complex golay pairs up to length 28: A search via computer algebra and programmatic SAT. *J. Symb. Comput.*, *102*, 153–172.

Bruttomesso, R., Pek, E., Sharygina, N., & Tsitovich, A. (2010). The opensmt solver. In Esparza, J., & Majumdar, R. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, Vol. 6015 of *Lecture Notes in Computer Science*, pp. 150–153. Springer.

Cadar, C., Dunbar, D., & Engler, D. R. (2008). KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Draves, R., & van Renesse, R. (Eds.), *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224. USENIX Association.

Cimatti, A., Griggio, A., Schaafsma, B. J., & Sebastiani, R. (2013). The MathSAT5 SMT solver. In Piterman, N., & Smolka, S. A. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, Vol. 7795 of *Lecture Notes in Computer Science*, pp. 93–107. Springer.

Codish, M., Gange, G., Itzhakov, A., & Stuckey, P. J. (2016). Breaking symmetries in graphs: The nauty way. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, Vol. 9892 of *Lecture Notes in Computer Science*, pp. 157–172. Springer.

Coutelier, R. (2023). Chronological vs. non-chronological backtracking in satisfiability modulo theories. Master's thesis, Université de Liège, Liège, Belgique.

de Moura, L. M., & Bjørner, N. (2008). Z3: an efficient SMT solver. In Ramakrishnan, C. R., & Rehof, J. (Eds.), *Tools and Algorithms for the Construction and Analysis*

of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, Vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Springer.

Devriendt, J., Bogaerts, B., Cat, B. D., Denecker, M., & Mears, C. (2012). Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pp. 49–56. IEEE Computer Society.

Eén, N., & Sörensson, N. (2003a). An extensible SAT-solver. In *SAT*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer.

Eén, N., & Sörensson, N. (2003b). Temporal induction by incremental SAT solving. In Strichman, O., & Biere, A. (Eds.), *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, Vol. 89 of *Electronic Notes in Theoretical Computer Science*, pp. 543–560. Elsevier.

Eisenhofer, C., Kovács, L., & Rawson, M. (2023). Embedding the connection calculus in satisfiability modulo theories. In Otten, J., & Bibel, W. (Eds.), *Proceedings of the 1st International Workshop on Automated Reasoning with Connection Calculi (AReCCa 2023), Prague, Czech Republic, September 18, 2023*, Vol. 3613 of *CEUR Workshop Proceedings*, pp. 54–63. CEUR-WS.org.

Erez, A., & Nadel, A. (2015). Finding bounded path in graph using SMT for automatic clock routing. In Kroening, D., & Pasareanu, C. S. (Eds.), *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, Vol. 9207 of *Lecture Notes in Computer Science*, pp. 20–36. Springer.

Fazekas, K., Biere, A., & Scholl, C. (2019). Incremental inprocessing in SAT solving. In Janota, M., & Lynce, I. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 136–154. Springer.

Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., & Biere, A. (2023). IPASIR-UP: user propagators for CDCL. In Mahajan, M., & Slivovsky, F. (Eds.), *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, Vol. 271 of *LIPIcs*, pp. 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., & Biere, A. (2024a). Satisfiability modulo user propagators experiments..

Fazekas, K., Pollitt, F., Fleury, M., & Biere, A. (2024b). Certifying incremental SAT solving. In Bjørner, N. S., Heule, M., & Voronkov, A. (Eds.), *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*, Vol. 100 of *EPiC Series in Computing*, pp. 321–340. EasyChair.

Fazekas, K., Pollitt, F., Fleury, M., & Biere, A. (2024c). Incremental proofs for bounded model checking. In Kunz, W. (Ed.), *27th GMM/ITG/GI Workshop on Methods*

and Description Languages for Modelling and Verification of Circuits and Systems (MBMV'24), Kaiserslautern, Germany, Vol. 314 of *ITG-Fachberichte*, pp. 133–143. VDE Verlag.

Ganesh, V., O'Donnell, C. W., Soos, M., Devadas, S., Rinard, M. C., & Solar-Lezama, A. (2012). Lynx: A programmatic SAT solver for the RNA-folding problem. In Cimatti, A., & Sebastiani, R. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Vol. 7317 of *Lecture Notes in Computer Science*, pp. 143–156. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Wanko, P. (2016). Theory solving made easy with clingo 5. In Carro, M., King, A., Saeedloei, N., & Vos, M. D. (Eds.), *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, Vol. 52 of *OASIcs*, pp. 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Gent, I. P., Miguel, I., & Moore, N. C. A. (2010). Lazy explanations for constraint propagators. In Carro, M., & Peña, R. (Eds.), *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, Vol. 5937 of *Lecture Notes in Computer Science*, pp. 217–233. Springer.

Godefroid, P., Levin, M. Y., & Molnar, D. A. (2012). SAGE: whitebox fuzzing for security testing. *Commun. ACM*, *55*(3), 40–44.

Gomes, C. P., Sabharwal, A., & Selman, B. (2021). Model counting. In Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 993–1014. IOS Press.

Heule, M. J. H. (2019). Optimal symmetry breaking for graph problems. *Math. Comput. Sci.*, *13*(4), 533–548.

Hickey, R., & Bacchus, F. (2019). Speeding up assumption-based SAT. In Janota, M., & Lynce, I. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 164–182. Springer.

Hooker, J. N. (1993). Solving the incremental satisfiability problem. *J. Log. Program.*, *15*(1&2), 177–186.

Ignatiev, A., Morgado, A., & Marques-Silva, J. (2019). RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.*, *11*(1), 53–64.

Itzhakov, A., & Codish, M. (2020). Incremental symmetry breaking constraints for graph search problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 1536–1543. AAAI Press.

Itzhakov, A., & Codish, M. (2023). Breaking symmetries with high dimensional graph invariants and their combination. In Ciré, A. A. (Ed.), *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 20th International Conference, CPAIOR 2023, Nice, France, May 29 - June 1, 2023, Proceedings*, Vol. 13884 of *Lecture Notes in Computer Science*, pp. 133–149. Springer.

Järvisalo, M., Heule, M., & Biere, A. (2012). Inprocessing rules. In Gramlich, B., Miller, D., & Sattler, U. (Eds.), *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, Vol. 7364 of *Lecture Notes in Computer Science*, pp. 355–370. Springer.

Kiesl-Reiter, B., & Whalen, M. W. (2023). Proofs for incremental SAT with inprocessing. In *FMCAD*, pp. 132–140. IEEE.

Kim, J., Whittemore, J., & Sakallah, K. A. (2000). On solving stack-based incremental satisfiability problems. In *Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors, ICCD '00, Austin, Texas, USA, September 17-20, 2000*, pp. 379–382. IEEE Computer Society.

Kirchweger, M., Peitl, T., & Szeider, S. (2023a). Co-certificate learning with SAT modulo symmetries. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence, IJCAI 2023*. AAAI Press/IJCAI. To appear.

Kirchweger, M., Peitl, T., & Szeider, S. (2023b). A SAT solver's opinion on the Erdős-Faber-Lovász conjecture. In Mahajan, M., & Slivovsky, F. (Eds.), *The 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), July 04-08, 2023, Alghero, Italy*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Kirchweger, M., Scheucher, M., & Szeider, S. (2022). A SAT attack on Rota's basis conjecture. In Meel, K. S., & Strichman, O. (Eds.), *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, Vol. 236 of *LIPIcs*, pp. 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Kirchweger, M., Scheucher, M., & Szeider, S. (2023). SAT-based generation of planar graphs. In Mahajan, M., & Slivovsky, F. (Eds.), *The 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), July 04-08, 2023, Alghero, Italy*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Kirchweger, M., & Szeider, S. (2021). SAT modulo symmetries for graph generation. In Michel, L. D. (Ed.), *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, Vol. 210 of *LIPIcs*, pp. 34:1–34:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Kirchweger, M., & Szeider, S. (2024). Computing small rainbow cycle numbers with SAT modulo symmetries (short paper). In Shaw, P. (Ed.), *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, Vol. 307 of *LIPIcs*, pp. 37:1–37:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Kupferschmid, S., Lewis, M., Schubert, T., & Becker, B. (2011). Incremental preprocessing methods for use in BMC. *Formal Methods Syst. Des.*, *39*(2), 185–204.

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In Clarke, E. M., & Voronkov, A. (Eds.), *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, Vol. 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer.

Li, Z., Bright, C., & Ganesh, V. (2022). A SAT solver + computer algebra attack on the minimum Kochen–Specker problem. Tech. rep., School of Computer Science at the University of Windsor. https://cbright.myweb.cs.uwindsor.ca/reports/nmi-ks-preprint.pdf.

Möhle, S., & Biere, A. (2019). Backing backtracking. In Janota, M., & Lynce, I. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 250–266. Springer.

Moura, L. D., & Rueß, H. (2002). Lemmas on demand for satisfiability solvers. In *The 5th International Symposium on the Theory and Applications of Satisfiability Testing, SAT 2002, Cincinnati, USA, May 15, 2002*.

Nadel, A. (2022). Introducing Intel(R) SAT solver. In Meel, K. S., & Strichman, O. (Eds.), *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, Vol. 236 of *LIPIcs*, pp. 8:1–8:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Nadel, A., & Ryvchin, V. (2012). Efficient SAT solving under assumptions. In Cimatti, A., & Sebastiani, R. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Vol. 7317 of *Lecture Notes in Computer Science*, pp. 242–255. Springer.

Nadel, A., & Ryvchin, V. (2018). Chronological backtracking. In Beyersdorff, O., & Wintersteiger, C. M. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, Vol. 10929 of *Lecture Notes in Computer Science*, pp. 111–121. Springer.

Nadel, A., Ryvchin, V., & Strichman, O. (2012). Preprocessing in incremental SAT. In Cimatti, A., & Sebastiani, R. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Vol. 7317 of *Lecture Notes in Computer Science*, pp. 256–269. Springer.

Nadel, A., Ryvchin, V., & Strichman, O. (2014). Ultimately incremental SAT. In Sinz, C., & Egly, U. (Eds.), *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, Vol. 8561 of *Lecture Notes in Computer Science*, pp. 206–218. Springer.

Niemetz, A., Preiner, M., Wolf, C., & Biere, A. (2018). Btor2, BtorMC and Boolector 3.0. In Chockler, H., & Weissenbacher, G. (Eds.), *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, Vol. 10981 of *Lecture Notes in Computer Science*, pp. 587–595. Springer.

Nieuwenhuis, R., Oliveras, A., & Tinelli, C. (2006). Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, *53*(6), 937–977.

Preiner, M., Schurr, H.-J., Barrett, C., Fontaine, P., Niemetz, A., & Tinelli, C. (2024a). SMT-LIB release 2024 (incremental benchmarks)..

Preiner, M., Schurr, H.-J., Barrett, C., Fontaine, P., Niemetz, A., & Tinelli, C. (2024b). SMT-LIB release 2024 (non-incremental benchmarks)..

Silva, J. P. M., & Sakallah, K. A. (1996). GRASP - a new search algorithm for satisfiability. In Rutenbar, R. A., & Otten, R. H. J. M. (Eds.), *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pp. 220–227. IEEE Computer Society / ACM.

Silva, J. P. M., & Sakallah, K. A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, *48*(5), 506–521.

van der Tak, P., Ramos, A., & Heule, M. (2011). Reusing the assignment trail in CDCL solvers. *J. Satisf. Boolean Model. Comput.*, *7*(4), 133–138.

Whittemore, J., Kim, J., & Sakallah, K. A. (2001). SATIRE: A new incremental satisfiability engine. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pp. 542–545. ACM.

Zhang, H. (2021). Combinatorial designs by SAT solvers. In Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 819–858. IOS Press.

Zhang, T., & Szeider, S. (2023). Searching for smallest universal graphs and tournaments with SAT. In *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, Vol. 280 of *LIPIcs*, pp. 39:1–39:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Zulkoski, E., Ganesh, V., & Czarnecki, K. (2016). MATHCHECK: A math assistant via a combination of computer algebra systems and SAT solvers. In Kambhampati, S. (Ed.), *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pp. 4228–4233. IJCAI/AAAI Press.