IPASIR-UP: User Propagators for CDCL

Katalin Fazekas
TU Wien, Vienna, Austria

Aina Niemetz
Stanford University, Stanford, USA

Mathias Preiner
Stanford University, Stanford, USA

Markus Kirchweger
TU Wien, Vienna, Austria

Stefan Szeider
TU Wien, Vienna, Austria

Armin Biere
University of Freiburg, Freiburg, Germany

Abstract

Modern SAT solvers are frequently embedded as sub-reasoning engines into more complex tools for addressing problems beyond the Boolean satisfiability problem. Examples include solvers for Satisfiability Modulo Theories (SMT), combinatorial optimization, model enumeration and counting. In such use cases, the SAT solver is often able to provide relevant information beyond the satisfiability answer. Further, domain knowledge of the embedding system (e.g., symmetry properties or theory axioms) can be beneficial for the CDCL search, but cannot be efficiently represented in clausal form. In this paper, we propose a general interface to inspect and influence the internal behaviour of CDCL SAT solvers. Our goal is to capture the most essential functionalities that are sufficient to simplify and improve use cases that require a more fine-grained interaction with the SAT solver than provided via the standard IPASIR interface. For our experiments, we extend CaDiCaL with our interface and evaluate it on two representative use cases: enumerating graphs within the SAT modulo Symmetries framework (SMS), and as the main CDCL(T) SAT engine of the SMT solver cvc5.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases SAT, CDCL, Satisfiability Modulo Theories, Satisfiability Modulo Symmetries

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.8

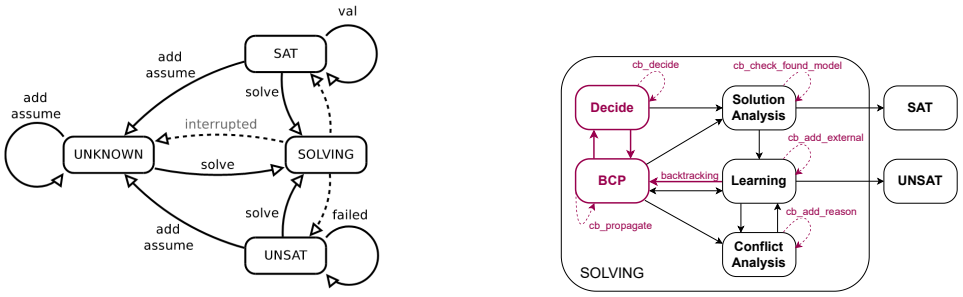
Category Tool Paper

Supplementary Material The evaluated tools and their most recent versions can be found here:
Software (Source Code): `https://doi.org/10.5281/zenodo.8003683`
Software (Source Code): `https://github.com/markirch/sat-modulo-symmetries/`
Software (Source Code): `https://github.com/cvc5/cvc5`
Software (Source Code): `https://github.com/arminbiere/cadical`

Funding 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).

1 Introduction

Modern SAT solvers frequently serve as crucial sub-reasoning engines of more complex tools for addressing problems beyond the Boolean satisfiability problem. Examples include solvers for Satisfiability Modulo Theories (SMT) [9], combinatorial problems [3, 37], or model



(a) The four possible states of SAT solvers according to the IPASIR interface (see [5]).

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

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

enumeration and counting [21]. The introduction of the IPASIR interface [5] enabled a relatively simple integration of off-the-shelf SAT solver as a black box into larger systems, typically to incrementally solve a sequence of similar propositional sub-problems.

Many use cases, 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 [35], 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 [22], and solvers for symmetric combinatorial problems, where it is desired to add additional clauses during search [15]. Currently, such use cases require either workarounds on the user level or non-trivial modifications of the SAT solver. As a consequence, it is non-trivial to replace the underlying SAT solver, which prevents taking advantage of recent advancements in SAT solving. Additionally, non-standard extensions and modifications of the SAT solver, if not done carefully, often come at the cost of an accidental performance hit.

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 [5] 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 being notified about changes to the trail during search, (2) to add clauses to the problem during solving without restarting the search, and (3) to propagate literals directly, based on external knowledge, without explicitly adding reason clauses (i.e., using delayed on-demand explanation). 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 [10], 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.

2 An Interface beyond IPASIR

The IPASIR interface, as introduced in [5], 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). Our goal is to extend IPASIR with functions that can provide such interactions, and thereby allow to simplify and improve several use cases of modern incremental 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 by an example implementation in C++ (see Listing 1 and Listing 2), the API is low-level enough to be supported in C as well. Note that many other (in this context) less relevant steps of the search (e.g. restart, reduce, and inprocessing) are ignored in the model of our interface.

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.

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. Second, function `is_decision` can be queried for a given variable to determine if it is currently assigned by a decision.

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);
```

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 CDCL via Notifications

We introduce the following three notification functions to capture the changes to the trail (see Listing 2 for signatures). Note that it is acceptable for a SAT solver to delay these notifications (e.g., to notify on assignments only once BCP finished). However, all notifications must happen at the latest before any of the callback functions in Section 2.3 are called.

`notify_assignment`. This function is called when an observed variable is assigned (either by `BCP` or `Decide` or by `Learning` a unit clause). Its first argument is the literal that is satisfied by the assignment, and its second argument is a Boolean flag to indicate when an assignment is *fixed*. A fixed assignment is persistent and the user must ensure that it is *never* undone (even if backtracking would unassign it or some assumptions of the problem are changed).

`notify_new_decision_level`. This function is called on every decision, even if it does not involve an observed variable. It does not report the actual decision or the current decision level—it only reports that a decision happened and thus, the decision level is increased.

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 are not fixed and were made above this new decision level must be treated as unassigned.

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, IPASIR-UP allows the user to influence CDCL in various ways via the following callback functions in each of the five states of the search (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.

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 the user for the corresponding reason clause via `cb_add_reason_clause_lit`, one literal at a time. The motivation for such delayed lazy explanation (see [19, 35]) 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. If not, additional clauses can be added to the problem without restarting the search (see below).

Learning. Whenever the solver has finished `BCP` (right before `Decide`), or in case 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, which is then added via `cb_add_external_clause_lit`, literal by literal. For proof generation, the solver stores these clauses as irredundant original input clauses. In case 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 External Propagation with Inprocessing

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 solving. Ways to combine inprocessing with incremental clause addition was proposed in [17, 33], but their implementations assumed that many clauses are added all at once. Finding an efficient way to implement [17] when new clauses are added one by one during search (as in IPASIR-UP) is intriguing future work. For now, 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 guarantees that no restore step is necessary when external clauses are added.

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

```
class ExternalPropagator {
public:
  virtual ~ExternalPropagator () { }

  virtual void notify_assignment (int lit, bool is_fixed) {}
  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 () { return false; }
  virtual int cb_add_external_clause_lit () { return 0; }
};
```

3 Related Work

The main motivation for incremental reasoning is to allow reusing previously learnt information when a similar problem is solved. IPASIR [5] was introduced as a universal interface for incremental SAT solvers, which enables easy integration into applications to take advantage of incremental reasoning without specializing for 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. It not only gives the user more comprehensive access to information about the solver state during solving, but allows to influence, and thus, guide its behavior based on user-level information that is not available to the SAT solver.

For instance, adding clauses via IPASIR forces the SAT solver to restart search, delete assumptions, and discard the trail and the implication graph. On the the other hand, adding clauses via `cb_add_external_clause_lit` in IPASIR-UP allows to *continue* the search while keeping all assumptions and backtracking only when a conflict is encountered.

Our proposed interface captures and standardizes functionality that is required by a range of applications, and is thus partially implemented in some tools. An important use case for interaction with the SAT solver as outlined above is the CDCL($\mathcal{T}$) framework [35] for SMT solvers. State-of-the-art SMT solvers based on this framework (e.g., [6, 13, 14]) currently all implement a custom interaction layer with the SAT solver (see, e.g., the SAT worker interface in [13]), which makes replacing these legacy SAT solvers with a state-of-the-art SAT solver highly non-trivial.

The IntelSAT solver [32] implements efficient clause addition on arbitrary decision levels (using *reimplication* to guarantee that no implications are missed on lower decision levels), but does not support external decisions, lazy propagation explanation, nor notifications.

The state-of-the-art ASP solver clingo [18] 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) and proof generation. The concept of user propagators has also been introduced in the SMT solver z3 [11], mainly to enable users to implement custom theory support.

4 Empirical Evaluation

To show that our interface is effective and efficient in varied use cases, we extended CaDiCaL [10], 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 [27], and integrating CaDiCaL as the main CDCL($\mathcal{T}$) SAT engine in the SMT solver cvc5 [6].

4.1 Experiments with SMS

SAT modulo Symmetries (SMS) [23–27] is a recently introduced SAT-based framework for the exhaustive generation of combinatorial objects such as graphs, hypergraphs, or matroids with a given property while excluding isomorphic copies of the same object (isomorph-free). 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. Static SAT encodings of lexicographic minimality require an exponential number of clauses [30]. 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 [18], 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 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 two 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 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 0 and no monochromatic triangle with color 1.

These graphs are interesting for topics related to the famous Kochen-Specker Theorem from quantum mechanics [2]. For encoding the non-010-colorability, we follow previous work [29]. In contrast to the first task, the encoding is relatively large, even exponential in the number of vertices, and contains auxiliary non-object variables.

For CaDiCaL, the *default* configuration propagates literals 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 enumeration. Configuration *no-prop* corresponds to *default* without propagating literals but learning the clause immediately. Configuration *no-inpro* corresponds to *default* without

Table 1 Enumerating up to isomorphism: all graphs (top) and all KS candidates (bottom).

			CaDiCaL+IPASIR-UP [s]			Clingo [s]	
	#vertices	#graphs	*default*	*enum-IPASIR*	*no-prop*	*red*	*irred*
All graphs	6	156	0.01	0.02	0.01	0.02	0.01
	7	1044	0.09	0.13	0.09	0.10	0.09
	8	12346	0.95	1.59	1.00	1.15	1.07
	9	274668	34.24	64.27	34.31	81.67	94.65
	10	12005168	50815.60	109443.72	57616.47	213959.23	196576.58
	#vertices	#graphs	*default*	*no-inpro*	*no-prop*	*red*	*irred*
KS candidates	16	0	10.58	9.14	13.58	25.07	18.56
	17	1	39.82	31.48	44.58	122.28	87.92
	18	0	190.16	59.37	187.29	872.98	493.17
	19	8	1220.51	1253.96	1341.80	10542.41	3348.14
	20	147	13647.66	16449.50	13493.86	67728.42	82871.65

inprocessing on the non-observed variables. 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 #vertices. The number of generated graphs is given in column #graphs. All the here presented experiments 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 (Table 1, top): 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. The bottom part of Table 1 shows that the versions using CaDiCaL perform better. Inprocessing improves performance on the larger instances, but with less vertices it is more efficient to be turned off. 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.

4.2 Experiments with SMT

Satisfiability Modulo Theories (SMT) solvers serve as the back-end reasoning engine for a variety of applications (e.g., [1, 4, 12, 20, 28, 34]). The majority of state-of-the-art SMT solvers are based on the CDCL($\mathcal{T}$) framework [35], 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 [8, 31]. 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

Table 2 SMT-LIB benchmarks solved by CVC5 and CVC5-IPASIRUP with a 300 seconds time limit.

	CVC5		CVC5-IPASIRUP	
Division	solved	time [s]	solved	time [s]
Arith (6,865)	**6,303**	173,628	6,299	176,278
BitVec (6,045)	**5,552**	153,899	5,529	161,482
Equality (12,159)	5,320	2,062,804	**5,322**	2,061,758
Equality+LinearArith (53,453)	45,902	2,288,230	**45,906**	2,288,352
Equality+MachineArith (6,071)	983	1,533,646	**987**	1,532,782
Equality+NonLinearArith (21,104)	**13,314**	2,419,535	13,053	2,486,588
FPArith (3,965)	3,145	268,628	**3,155**	266,245
QF_Bitvec (42,472)	**40,321**	984,880	40,320	985,946
QF_Datatypes (8,403)	8,077	110,704	**8,168**	82,878
QF_Equality (8,054)	8,044	9,394	**8,047**	7,169
QF_Equality+Bitvec (16,585)	15,817	307,558	**16,015**	234,369
QF_Equality+LinearArith (3,442)	**3,388**	23,041	3,381	23,465
QF_Equality+NonLinearArith (709)	627	27,428	**629**	27,598
QF_FPArith (76,238)	76,054	94,487	**76,081**	76,700
QF_LinearIntArith (16,387)	11,670	1,575,635	**12,004**	1,512,696
QF_LinearRealArith (2,008)	1,721	130,408	**1,766**	113,919
QF_NonLinearIntArith (25,361)	13,037	4,094,712	**13,682**	3,840,933
QF_NonLinearRealArith (12,134)	11,166	333,933	**11,238**	316,728
QF_Strings (69,908)	**69,357**	203,677	69,296	230,918
Total (391,363)	339,798	16,796,234	**340,878**	16,426,813

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 [6]. It relies on a highly customized version of MiniSat [16] 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 this version of MiniSat with a state-of-the-art SAT solver and take advantage of improvements in SAT solving. Replacing this customized 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 further has the additional advantage that interaction with the SAT layer is standardized and clean, i.e., no "hacks" have to be added to the SAT solver that may accidentally impact performance.

We integrated CaDiCaL with the IPASIR-UP extension as main CDCL($\mathcal{T}$) SAT engine 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 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 700 lines of C++ code on top of cvc5 1.0.5. In the following, we refer to this version of cvc5 with CaDiCaL as the CDCL($\mathcal{T}$) SAT engine as CVC5-IPASIRUP. Note that proof production 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 1.0.5 on all non-incremental benchmarks of the 2022 release of SMT-LIB [7]. We ran this experiment on a cluster equipped with Intel Xeon E5-2650v4 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 grouped into the divisions defined in SMT-COMP 2022 [36].

Overall, CVC5-IPASIRUP solves 1080 more benchmarks than CVC5 and improves over CVC5 in 13 out of 19 divisions. On the 336,533 commonly solved benchmarks, CVC5-IPASIRUP (947,053s) is 1.16× faster than CVC5 (1,096,092s). For quantifier-free divisions, CVC5-IPASIRUP significantly improves over CVC5 in arithmetic logics (+1091) and in logics that combine bit-vectors with arrays (+198). On quantified divisions, CVC5-IPASIRUP's performance is similar to CVC5 except for the UFNIA logic (in division Equality+NonLinearArith), where CVC5-IPASIRUP solves 251 less benchmarks than CVC5. The overall results of CVC5-IPASIRUP are very encouraging, given the fact that the CVC5 code base is tuned for its custom version of MiniSat. This particularly applies to the quantifiers module in CVC5, explaining the UFNIA performance regression. However, the CVC5-IPASIRUP implementation provides a solid baseline to tune and improve cvc5's internals for the IPASIR-UP interface.

5 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. Further, in our current implementation proof production is only experimental. In future work it needs to be evaluated and extended to support further features such as distinction of redundant and irredundant external clauses.

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. For example, users might want to decide when to restart the search or where to backtrack upon a conflict. Sharing more information about the internal search statistics, or variable and clause scores could also be valuable. 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.

Acknowledgements. The authors would like to thank the reviewers for their thoughtful comments and helpful suggestions on improving this paper.

References

1 Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013. URL: `https://ieeexplore.ieee.org/document/6679385/`.

2 Felix Arends, Joël Ouaknine, and Charles W. Wampler. On searching for small Kochen-Specker vector systems. In Petr Kolman and Jan Kratochvíl, editors, *Graph-Theoretic Concepts in Computer Science - 37th International Workshop, WG 2011, Teplá Monastery, Czech Republic, June 21-24, 2011. Revised Papers*, volume 6986 of *LNCS*, pages 23–34. Springer, 2011. `doi:10.1007/978-3-642-25870-1_4`.

3 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. `doi:10.3233/FAIA201008`.

4 John Backes, Ulises Berrueco, Tyler Bray, Daniel Brim, Byron Cook, Andrew Gacek, Ranjit Jhala, Kasper Søe Luckow, Sean McLaughlin, Madhav Menon, Daniel Peebles, Ujjwal Pugalia, Neha Rungta, Cole Schlesinger, Adam Schodde, Anvesh Tanuku, Carsten Varming, and Deepa Viswanathan. Stratified abstraction of access control policies. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 165–176. Springer, 2020. `doi:10.1007/978-3-030-53288-8\_9`.

5 Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016. `doi:10.1016/j.artint.2016.08.007`.

6 Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *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*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. `doi:10.1007/978-3-030-99524-9\_24`.

7 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2023. URL: `http://smt-lib.org`.

8 Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2002. `doi:10.1007/3-540-45657-0\_18`.

9 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. `doi:10.3233/FAIA201017`.

10 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

11 Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. Satisfiability modulo custom theories in Z3. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation - 24th International Conference, VMCAI 2023, Boston,*

MA, USA, January 16-17, 2023, Proceedings, volume 13881 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2023. doi:10.1007/978-3-031-24950-1_5.

12 Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

13 Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *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*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013. doi:10.1007/978-3-642-36742-7_7.

14 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *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*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

15 Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. 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*, pages 49–56. IEEE Computer Society, 2012. doi:10.1109/ICTAI.2012.16.

16 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

17 Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019. doi:10.1007/978-3-030-24258-9_9.

18 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos, editors, *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, volume 52 of *OASIcs*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.ICLP.2016.2.

19 Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010. doi:10.1007/978-3-642-11503-5_19.

20 Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012. doi:10.1145/2093548.2093564.

21 Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 993–1014. IOS Press, 2021. doi:10.3233/FAIA201009.

22 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient maxsat solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019. doi:10.3233/SAT190116.

23 Markus Kirchweger, Tomáš Peitl, and Stefan Szeider. Co-certificate learning with SAT modulo symmetries. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence, IJCAI 2023*. AAAI Press/IJCAI, 2023. To appear.

24 Markus Kirchweger, Tomáš Peitl, and Stefan Szeider. A SAT solver's opinion on the Erdős-Faber-Lovász conjecture. In Meena Mahajan and Friedrich Slivovsky, editors, *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, 2023. to appear.

25 Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. A SAT attack on Rota's basis conjecture. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.4.

26 Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. SAT-based generation of planar graphs. In Meena Mahajan and Friedrich Slivovsky, editors, *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, 2023. to appear.

27 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 34:1–34:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.34.

28 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

29 Zhengyu Li, Curtis Bright, and Vijay Ganesh. A SAT solver + computer algebra attack on the minimum Kochen–Specker problem. Technical report, School of Computer Science at the University of Windsor, November 2022. https://cbright.myweb.cs.uwindsor.ca/reports/nmi-ks-preprint.pdf.

30 Eugene M Luks and Amitabha Roy. The complexity of symmetry-breaking formulas. *Annals of Mathematics and Artificial Intelligence*, 41(1):19–45, 2004.

31 Leonardo De Moura and Harald Rueß. 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*, 2002.

32 Alexander Nadel. Introducing intel(r) SAT solver. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.8.

33 Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in incremental SAT. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 256–269. Springer, 2012. doi:10.1007/978-3-642-31612-8_20.

34 Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, *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*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018. doi:10.1007/978-3-319-96145-3_32.

35 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006. doi:10.1145/1217856.1217859.

36 The International Satisfiability Modulo Theories Competition (SMT-COMP), 2022. URL: https://smt-comp.github.io/2022.

37 Hantao Zhang. Combinatorial designs by SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 819–858. IOS Press, 2021. doi:10.3233/FAIA201005.