Abstract—This paper is a description of the CVC5 SMT solver as entered into the 2022 SMT Competition. Here, we briefly summarize the main techniques implemented by CVC5 that are relevant. For more comprehensive information please refer to the tool paper about CVC5 [15], our website [7], and the source code on GitHub [6].

Overview

CVC5 is an open-source automatic theorem prover for SMT problems. It can be used to prove the validity or, dually, the satisfiability of first-order formulas in a large number of built-in logical theories and combinations thereof. For unsatisfiable formulas, CVC5 can be used to produce proofs in multiple formats [16]. CVC5 is an open and extensible SMT engine, and it can be used as a stand-alone tool or as a library. There is essentially no limit on its use for research or commercial purposes (see the section on its license below for more information).

Features

CVC5 is a CDCL(T)-based SMT solver that supports all theories standardized in SMT-LIB. It uses a modified version of MiniSat [28] as its CDCL(T) SAT solver. Theory combination is based on the polite combination framework [33, 41] using care graphs [34, 35].

Linear Arithmetic CVC5’s solver for linear arithmetic implements a Simplex procedure [27]. It includes heuristics proposed by Griggio [29]. Integers are handled by first solving the real relaxation of the constraints, and then using a combination of cuts from proofs of unsatisfiability [26] and branching to ensure integer solutions [30]. Additionally, the branch-and-bound method can optionally generate lemmas consisting of ternary clauses inspired by unit-cube tests [20].

Non-linear Arithmetic For non-linear arithmetic, we use strategies that are based on the combination of two independent subsolvers. The first subsolver is based on incremental linearization [22], where models are found for the linear abstraction of the input formula, i.e., treating multiplication as an interpreted function. Lemma schemas are then used to state properties of multiplication in a counterexample-guided fashion. Details on the lemma schemas used by this subsolver are described in [46]. The second subsolver implements cylindrical algebraic coverings [13] using the polynomial arithmetic and other algebraic routines from libpoly [36].

We primarily invoke incremental linearization for non-linear integer problems, and cylindrical algebraic decomposition for non-linear real problems. We additionally invoke incomplete techniques based on reductions to bit-vectors for non-linear integer problems, and combinations of the two solvers described above for non-linear real arithmetic.

Arrays The array solver implements a procedure inspired by the one described in de Moura and Bjørner [24]. Optionally, CVC5 reasons about arrays using an approach proposed by Christ and Hoenicke to lazily instantiate lemmas based on dependencies between arrays that differ in finitely many indices [21].

Bit-Vectors CVC5’s bit-vector solver uses bit-blasting and supports off-the-shelf SAT solvers such as CaDiCaL, or CryptoMiniSat [2] as SAT back-ends. In the current version, we use CaDiCaL [18] by default. The new bit-blasting solver seamlessly integrates into the CDCL(T) infrastructure of CVC5 and fully supports the combination of bit-vectors with any theory supported by CVC5.

Datatypes For handling quantifier-free constraints over datatypes, we use a rule-based procedure that follows the calculi described in [17, 42]. The procedure incorporates optimizations for sharing selectors over multiple constructors [49].

Floating-Point Arithmetic CVC5 eagerly translates floating-point expressions to the theory of bit-vectors. For that, it integrates SymFPU [19], a C++ library of bit-vector encodings of floating-point operations. Conversions between real and floating-point numbers are handled lazily.

Strings CVC5’s string solver consists of multiple components. At its core, the solver reasons about word equations [37]. The solver supplements reasoning about word equations with reasoning about code points to handle conversions between strings and integers efficiently [51]. The component responsible for extended functions such as string replacement, lazily reduces those functions to word equations after context-dependent simplifications [47]. Skolem variables in the lemmas
produced by the reductions reuse existing Skolem variables whenever possible for greater efficiency [52]. The regular expression component unfolds and computes derivatives of regular expressions [38]. The string solver incorporates aggre-
sive simplification rules that rely on abstractions to derive facts about string terms [50]. Finally, the solver detects conflicts eagerly on partial assignments from the SAT solver by computing the congruence-closure and constant prefixes and suffixes of string terms.

**Uninterpreted Functions** The theory solver for uninterpreted functions resembles Simplify’s approach [25] and remains largely unchanged. When combined with bit-vectors, CVC5 supports the Ackermannization and eager bit-blasting of constraints involving uninterpreted functions and sorts [31].

**Quantifiers** For handling logics where quantifiers are present, we rely on heuristic E-matching when they are combined with uninterpreted functions [23]. This technique is supplemented by conflict-based instantiation for detecting when an instantiation is in conflict with the current set of assertions [44]. Our strategy additionally incorporates finite model finding techniques, which are useful for finding satisfiable instances [43]. We additionally rely on enumerative approaches for instantiation when all other techniques are incomplete [48].

For quantifiers over linear arithmetic, we use a specialized counterexample-guided based approach for quantifier instantiation [45]. An extension of this technique is used for quantified bit-vector logics [39]. For other quantified logics in pure background theories, e.g., over floating-point or non-linear arithmetic, we use new techniques for syntax-guided quantifier instantiation [40].

**Decision Heuristic** In addition to MiniSat’s decision heuristic, CVC5 implements a separate heuristic that uses the original Boolean structure of the input to keep track of the justified parts of the input constraints, i.e., the parts where it can infer the value of terms based on a (partial) assignment to subterms. For decisions, it traverses assertions that are not satisfied under the current assignment, computing the desired values (starting with true as the desired value for the root) for each term until it finds a literal that has not been assigned and would contribute towards a desired value. The heuristic optionally prioritizes assertions that led to decisions that resulted in a conflict. This heuristic is a reimplemention and extension of a heuristic implemented in CVC4 [14].

**Unsat Cores** CVC5 implements two approaches to compute unsatisfiable cores: (i) assumption-based unsat cores (ii) proof-based unsat cores. Both approaches use CVC5’s proof infrastructure. CVC5’s proof infrastructure generates fine-grained proofs for unsatisfiable problems. The assumption-based approach uses MiniSat’s support for computing unsatisfiable assumptions. CVC5 uses the proof infrastructure to track the preprocessing of assertions, sends the constraints as assumptions to MiniSat, and retrieves the list of unsatisfiable assumptions after running its regular solving procedure. The proof-based approach uses the proof infrastructure to track preprocessing and the reasoning done by the SAT solver. After the main solving procedure finishes, it extracts the unsat core from the proof.

**CONFIGURATIONS**

CVC5 is entering all divisions of the single query, the incremental, the unsat-core, and the model-validation tracks of SMT-COMP 2022. We also enter two variants to the proof exhibition track as explained below.

The branch used for all configurations is smtcomp2022 [11]. We use a binary optimized for reading input from files in a competition setting for all tracks but the incremental track. For the incremental track, we use a binary optimized for reading from interactive inputs. For each track, we use a dedicated run script, which calls CVC5 with parameters that depend on the logic used in the input. For details about the parameters used for each logic, please refer to the run scripts in the competition branch [8, 9, 10, 12]. All configurations are compiled with the optional dependencies CLN [1], glpk-cut-log [4] (a fork of GLPK [5]), CaDiCaL (commit 8b2c2b3b), SymFPU (commit 8bfe139), and libpoly (commit 1383809).

**Single Query Track (CVC5)** For the Single Query track, we configure CVC5 for optimized reading from non-interactive inputs. For certain logics, we try different options sequentially (see runscript at [10]).

**Incremental Track (CVC5-inc)** For the Incremental track, we configure CVC5 for optimized reading from interactive inputs and use the default options for most logics. See the runscript [8] for more details.

**Unsat-Core Track (CVC5)** For the Unsat Core track, we configure CVC5 for optimized reading from non-interactive inputs and use options similar to the ones used for the Single Query Track (see runscript [12] for details). The submission uses assumption-based unsat cores.

**Model-Validation Track (CVC5)** For the model-validation track, we use a similar configuration as for the Single Query track (see runscript [9] for details). For QF_LRA, we disable the simplification of unconstrained terms since it is not compatible with model generation.

**Proof Exhibition Track (CVC5 and CVC5-lfsc)** For the proof exhibition track, CVC5 uses a new flexible proof-producing architecture [16]. CVC5 can check proofs internally or produce proofs in a format that can be checked using an external tool. The internal proof checker checks proofs during construction. If a user does not want to trust CVC5’s internal checker and desires to check the proofs with an external checker, they can use the other formats that CVC5 supports, namely LFSC (described below) and Alethe.1 The internal proofs can also be printed in a DOT format to facilitate visualization.2 CVC5 participates with two entries:

1See https://cvc5.github.io/docs/cvc5-1.0.0/proofs/output_alethe.html
2See https://cvc5.github.io/docs/cvc5-1.0.0/proofs/output_dot.html. One can use a default DOT visualizer or the dedicated proof visualizer available at https://ufmg-smite.github.io/proof-visualizer/
• the default cvc5 entry produces proofs in the *internal proof format* of cvc5.

The format contains proof rules that very closely represent how cvc5 is solving the problem. A proof rule application has the form

```
(RULENAME F1 ... Fn :args t1 ... tm)
```

where F1 ... Fn are the *children* proof rule applications, t1 ... tm are the *arguments* terms, and RULENAME is a function from the children and arguments to a *conclusion* term. A proof is directed acyclic graph of proof rule applications. Seen as a tree, the root necessarily has the negation of the input formula as the conclusion, proven to be unsatisfiable.

No external checker is used. The proofs are rather checked during construction in cvc5 itself, so that if an invalid proof is produced, cvc5 fails with an error message.

• the cvc5-lfsc entry produces proofs using the LFSC framework [53]. LFSC is a logical framework, based on Edinburgh LF [32], which was explicitly designed to facilitate the production and checking of fine-grained proofs in SMT. It comes with a small and efficient proof checker, which is generic in the sense that it takes as input both a proof term p and a *proof signature*, a definition of the data types and proof rules used to construct p. The checker verifies that p is well-formed with respect to the provided signature. We have defined proof signatures for most of the theories supported by cvc5. These definitions can be combined together as needed to define a proof system for any combination of those theories. When emitting proofs in LFSC, cvc5 includes all the relevant signatures as a preamble to the proof term.

**COPYRIGHT AND LICENSE**

CVC5 is copyright 2022 by its authors and contributors and their institutional affiliations. For a full list of authors, refer to the AUTHORS and THANKS files distributed with the source code [6].

The source code of cvc5 is open and available to students, researchers, software companies, and everyone else to study, to modify, and to redistribute original or modified versions; distribution is under the terms of the modified BSD license. Please note that cvc5 can be configured (however, by default it is not) to link against some GPLed libraries, and therefore the use of these builds may be restricted in non-GPL-compatible projects. For more information about cvc5’s license refer to the actual license text as distributed with its source code [6].

**ACKNOWLEDGMENTS**

CVC5 is supported in part by the organizations listed on our website [3].

---

3 The rules are described in detail at [https://cvc5.github.io/docs/cvc5-1.0.0/proofs/proof_rules.html](https://cvc5.github.io/docs/cvc5-1.0.0/proofs/proof_rules.html).

---

**REFERENCES**


[40] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. Syntax-guided quantifier instantiation. In Jan Friso Groote and Kim Guldb...


