Smt-Switch: A Solver-Agnostic C++ API for SMT Solving

Makai Mann[1(✉)], Amalee Wilson[1], Yoni Zohar[1], Lindsey Stuntz[1], Ahmed Irfan[1], Kristopher Brown[1], Caleb Donovick[1], Allison Guman[3], Cesare Tinelli[2], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
{makaim,amalee,yoniz,lstuntz,irfan,donovick}@cs.stanford.edu, ksb@stanford.edu

[2] The University of Iowa, Iowa City, USA
cesare-tinelli@uiowa.edu

[3] Columbia University, New York City, USA
ag3910@columbia.edu

Abstract. This paper presents `Smt-Switch`, an open-source, solver-agnostic API for SMT solving. `Smt-Switch` provides simple, uniform, and high-performance access to SMT solving for applications in areas such as automated reasoning, planning, and formal verification. It defines an abstract interface, which can be implemented by different SMT solvers. The interface allows the user to create, traverse, and manipulate terms, as well as dynamically dispatch queries to various underlying SMT solvers.

1 Introduction

`Smt-Switch` is an open-source, solver-agnostic C++ API for interacting with SMT-LIB-compliant SMT solvers. While SMT-LIB [1] provides a standard textual interface for SMT solving, there are limitations to that interface. In particular, applications that need to manipulate solver formulas or respond to solver output are easier and more efficient with an integrated API. Common approaches for addressing these limitations include committing to a specific solver (and its API) or using a custom internal expression representation, which is then translated to SMT-LIB and sent to a solver. In contrast, `Smt-Switch` provides a generic in-memory API, but without a custom representation, instead providing a lightweight wrapper around the underlying solver expressions. `Smt-Switch` already has support for many prominent SMT solvers and a variety of theories, and it provides an extensible abstract interface which makes it easy to add new solvers and theories. `Smt-Switch` is open-source and uses the permissive BSD license. It is available at https://github.com/makaimann/smt-switch.

The remainder of the paper is organized as follows. We start by describing the architecture of the tool in Sect. 2. Section 3 illustrates how to use the API with a simple example. We cover related work in Sect. 4 and give an experimental evaluation in Sect. 5. Finally, Sect. 6 concludes.

C.-M. Li and F. Manyà (Eds.): SAT 2021, LNCS 12831, pp. 377–386, 2021.
https://doi.org/10.1007/978-3-030-80223-3_26

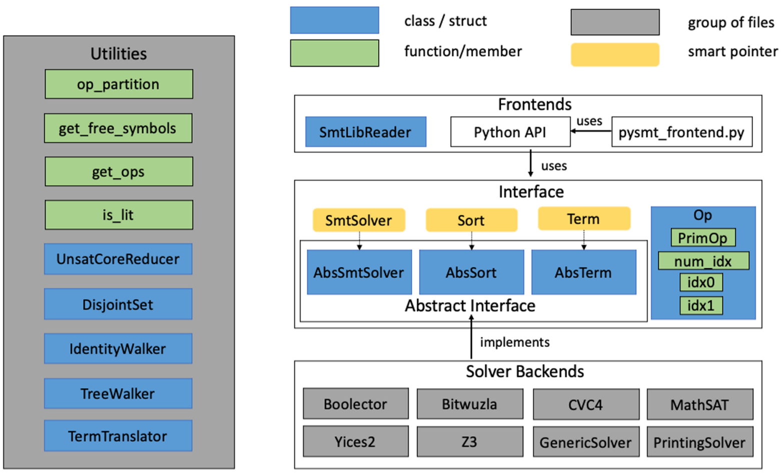

Fig. 1. Architecture diagram

2 Design

Figure 1 depicts an overview of the `Smt-Switch` architecture. After some general comments, we explain the various components in the figure. Throughout this paper, we refer to the external code of some SMT solver used by `Smt-Switch` as an *underlying* solver, and we use *backend* to refer to the `Smt-Switch` wrapper for an underlying solver. `Smt-Switch` delegates as much of the functionality to the underlying solvers as possible. This reduces redundancy and results in simpler implementations and lower memory overhead. The API is implemented in C++, and `Smt-Switch` also provides Python bindings using Cython [4].

Building and Linking. `Smt-Switch` uses CMake [13]. The build infrastructure is designed to be modular with respect to backend solvers. This allows the user to build `Smt-Switch` once and then link solver backends to their project as needed. The build configuration script also has options to enable static and debug builds.

Testing. We use *GoogleTest* [10] and *Pytest* [14] for the C++ test infrastructure and the Python test infrastructure, respectively. Tests are parameterized by solver so that each test can easily be run over all solvers.

Custom Exceptions. `Smt-Switch` defines its own set of exceptions inherited from `std::exception`. Each of them has a `std::string` message. The defined exceptions are: i) `SmtException` – the generic base class exception, ii) `NotImplementedException`, iii) `IncorrectUsageException`, and iv) `InternalSolverException`.

License. The `Smt-Switch` code is distributed under the BSD 3-clause license and provides setup scripts for building underlying solvers with similarly liberal open-source licenses. For solvers with more restrictive licenses, users are responsible for obtaining the underlying solver libraries themselves.

2.1 Interface

Smt-Switch provides abstract classes that define an interface for interacting with an underlying SMT solver. The interface corresponds closely to SMT-LIB version 2.6 [1], making it straightforward to connect solvers that are SMT-LIB compliant. At the Smt-Switch API level, the user interacts with smart pointers to the abstract classes. The virtual method functionality of C++ allows the interface to be agnostic to the underlying solver. The three primary abstract classes are: i) AbsSort; iii) AbsTerm; and iii) AbsSmtSolver. The Op class is not abstract and does not need to be implemented by the backend. However, the backend must interpret an Op when building terms.

AbsSort. The AbsSort abstract class represents logical sorts in Smt-Switch. A Sort is a pointer to an AbsSort. An enum called SortKind is used to represent built-in SMT-LIB sorts. In some cases, additional parameters are needed to create a sort. For example, bitvector sorts all have SortKind BV, and to create a bitvector sort, an additional parameter for bit-width is needed. Smt-Switch currently supports the following sorts, as they are defined in the SMT-LIB standard: i) Booleans, ii) integers, iii) reals, iv) fixed-width bitvectors, v) uninterpreted functions, vi) arrays, vii) uninterpreted sorts, and viii) algebraic datatypes. Each backend is responsible for creating an AbsSort object, given a SortKind (and its parameters if any). The backend must also be able to provide the SortKind and parameter information from a given sort.

Op. Op is a struct that represents logical function operators in Smt-Switch. As with sorts, there is an enum (called PrimOp) that contains built-in SMT-LIB functions from various theories. An Op stores a PrimOp and up to two integer indices. Unindexed operators are defined only by their PrimOp. Indexed operators use one or two indices. For example, integer addition is represented as PrimOp::Plus without any indices, while bitvector extraction uses PrimOp::Extract together with two indices specifying the most and least significant bits of the extracted slice. Smt-Switch uses a simple naming scheme for PrimOp's based on the corresponding SMT-LIB names.

AbsTerm. The AbsTerm abstract class represents logical terms, and a Term is a pointer to an AbsTerm. A Term can be a symbol (uninterpreted constant or function), a parameter (variable to be bound by a quantifier), a value (term corresponding to a model value such as 0 or 1), or an operator applied to one or more terms. Parameters are bound using the Forall or Exists operators. Terms can be queried to obtain their Sort, Op, and children (if a Term is not a function application, its Op is null). Note that, unlike the current SMT-LIB standard, we consider uninterpreted functions themselves to be terms. To create an uninterpreted function application, the Apply Op is used, where the first argument is the function to be applied and the rest of the arguments are

Terms representing the arguments to the function. This simplifies the interface and also makes it possible to support higher-order constructs if an underlying solver supports it. It also facilitates an invariant maintained by Smt-Switch: any Term with a non-null operator is equal to the result of querying its Op and children and then creating a new Term with the obtained Op and children.

AbsSmtSolver. The AbsSmtSolver class provides the main interface that a user interacts with. It has methods for declaring Sorts, building Terms, asserting formulas, and checking for satisfiability. The method names mirror the commands of SMT-LIB, replacing "-" with "_." One exception is assert, which is assert_formula in Smt-Switch to avoid clashing with the C assertion macro. SmtSolver is a pointer to an AbsSmtSolver.

Solver Factories. A solver factory defines a single static method: create. Each backend solver implementation defines a corresponding factory in a dedicated header file. The create function produces an SmtSolver for its corresponding backend. It takes a single Boolean parameter called logging, which specifies whether to add a layer to keep track of the structure of terms being created. This is useful if the underlying solver does not preserve term structure (e.g., if it performs on-the-fly rewriting of created terms) and the user needs the invariant that if you create a term with a given Op and children and then query the Op and children of the new term, you get back the Op and children you started with (note that this is the inverse of the invariant mentioned in the AbsTerm section above). Smt-Switch currently has backends for Boolector [18], Bitwuzla [17], CVC4 [3], MathSAT [5], Yices2 [6], and Z3 [16]. It also provides two more special implementations of AbsSmtSolver: i) PrintingSolver – a wrapper around a backend that logs all API calls and dumps them as an SMT-LIB script to an output stream – especially useful for debugging as it provides a way to reproduce a behavior seen in Smt-Switch using just the underlying solver with an SMT-LIB file; and ii) GenericSolver – communicates interactively with an arbitrary SMT-LIB-compliant solver binary through pipes.

2.2 Additional Features

Analysis. Smt-Switch provides utility functions for i) gathering all sub-terms matching some given criteria in a term; ii) reducing an unsatisfiable core; iii) returning a flat list of all the arguments of a commutative and associative operator (e.g., Boolean and or or); iv) manipulating disjoint sets (union-find data structures) of Smt-Switch Terms; and v) traversing and rewriting Smt-Switch Terms – by inheriting from the IdentityWalker or TreeWalker classes. In the former, each sub-term is visited once, regardless of how many times it occurs in the formula. In the latter, every occurrence of every sub-term is visited.

Term Translation. The TermTranslator class can be used to copy terms from one backend solver to another. The only requirement for this to work is that the source solver must implement the term traversal interface methods. This functionality makes it easy to communicate information among several solvers.

Portfolio Solving. Smt-Switch provides infrastructure for using a portfolio of backend solvers to solve a single problem in parallel (i.e., the first to finish reports the answer).

Additional Frontends. In addition to its C++ library, Smt-Switch provides a Flex [20] and Bison [8] parser for the SMT-LIB language, and a Python module for translating between PySMT [9] terms and Smt-Switch terms.

3 Example

In this section, we demonstrate the Smt-Switch API with a simple example. Figure 2 (left) uses Smt-Switch with the CVC4 backend to solve simple queries over bitvectors and uninterpreted functions. It starts by including C++ and Smt-Switch headers and invoking the relevant using declarations. The main function then begins by creating a backend SmtSolver using CVC4 without logging. Note that changing the backend solver can easily be done by only changing this line and the factory being included. The logic is set to quantifier-free formulas over bitvectors and uninterpreted functions (QF_UFBV), and solver options are used to enable incremental solving, models, and the production of "unsat assumptions" (an SMT-LIB variant of unsatisfiable core functionality, in which the core is taken from a specified set of assumptions). This is followed by creating two sorts: a bitvector sort of width 32, and a function sort with that sort as both domain and codomain. The next three lines create two bitvector symbolic constants and an uninterpreted function. Next, the terms x0 and y0 are created, corresponding to the least significant half of the bitvectors x and y, respectively, by applying the bitvector extract operator with upper index 15 and lower index 0.

We then assert that applying the function to x and y results in different values and push a new context, in which we assert that the bottom halves of x and y are equal. This is followed by a successful satisfiability check, after which we print the value assigned to x and pop to the top level context. The query is satisfiable because x and y can have different most significant bits, and thus the function applications could return different values.

We then create a term that represents the bit-wise and of x and y and three Boolean terms built in various ways from x and y. The final satisfiability check is done with these terms as assumptions and is unsatisfiable, because the assumptions entail that x equals y, contradicting the top-level assertion that f applied to x is different from f applied to y. Finally, we extract a subset of the assumptions that (together with existing assertions) is sufficient for unsatisfiability. The output of the program is shown at the top right-hand side of the figure. Looking at the output, we can see that only the last two assumptions are needed for unsatisfiability.

The remainder of the right-hand side of Fig. 2 shows the corresponding SMT-LIB commands for the C++ code. An artifact containing the example in both

```cpp
#include <iostream>
#include "smt-switch/cvc4_factory.h"
#include "smt-switch/smt.h"
using namespace smt;
using namespace std;
int main()
{
 SmtSolver s =
   CVC4SolverFactory::create(false);

 s->set_logic("QF_UFBV");
 s->set_opt("incremental", "true");
 s->set_opt("produce-models", "true");
 s->set_opt("produce-unsat-assumptions",
   "true");
 Sort bvs = s->make_sort(BV, 32);
 Sort funs =
   s->make_sort(FUNCTION, {bvs, bvs});

 Term x = s->make_symbol("x", bvs);
 Term y = s->make_symbol("y", bvs);
 Term f = s->make_symbol("f", funs);

 Op ext = Op(Extract, 15, 0);
 Term x0 = s->make_term(ext, x);
 Term y0 = s->make_term(ext, y);

 Term fx = s->make_term(Apply, f, x);
 Term fy = s->make_term(Apply, f, y);
 s->assert_formula(
   s->make_term(Distinct, fx, fy));

 s->push(1);
 s->assert_formula(
   s->make_term(Equal, x0, y0));
 cout <<  s->check_sat() << endl;

 cout << s->get_value(x) << endl;
 s->pop(1);

 Term xy = s->make_term(BVAnd, x, y);
 Term a1 = s->make_term(BVUge, x0, y0);
 Term a2 = s->make_term(BVUge, xy, x);
 Term a3 = s->make_term(BVUge, xy, y);
 cout <<
   s->check_sat_assuming({a1, a2, a3})
   << endl;
 UnorderedTermSet ua;
 s->get_unsat_assumptions(ua);
 for (Term t : ua) { cout << t << endl; }
}
```

```
Output:
-------
sat
(_ bv0 32)
unsat
(bvuge (bvand x y) y)
(bvuge (bvand x y) x)

SMT-LIB:
--------
(set-logic QF_UFBV)
(set-option :incremental true)
(set-option :produce-models true)
(set-option :produce-unsat-assumptions
   true)
(define-sort bvs () (_ BitVec 32))


(declare-const x bvs)
(declare-const y bvs)
(declare-fun f (bvs) bvs)

(define-fun x0 () (_ BitVec 16)
  ((_ extract 15 0) x))
(define-fun y0 () (_ BitVec 16)
  ((_ extract 15 0) y))


(assert (distinct (f x) (f y)))


(push 1)
(assert (= x0 y0))

(check-sat)

(get-value (x))
(pop 1)

(define-fun xy () bvs (bvand x y))
(define-fun a1 () Bool (bvuge x0 y0))
(define-fun a2 () Bool (bvuge xy x))
(define-fun a3 () Bool (bvuge xy y))

(check-sat-assuming (a1 a2 a3))


(get-unsat-assumptions)
```

Fig. 2. Left: C++ API Example. Right: the output from running the program, as well as the SMT-LIB script that corresponds to the C++ example.

C++ and Python (as well as scripts for reproducing the results in Sect. 5) is available at https://doi.org/10.6084/m9.figshare.14566449.v1.

4 Related Work

The most closely related tools are smt-kit [11] and *metaSMT* [19], other C++ APIs for SMT solving. Both utilize templates to be solver agnostic and have term representations that are separate from the underlying solver, as opposed to Smt-Switch which provides an abstract interface and only a light wrapper around the term representations of the underlying solvers. This design choice reduces overhead and keeps maintenance simple. *metaSMT* makes clever use of C++ template meta-programming to help reduce its overhead. Furthermore, it provides several features including bit-blasting and infrastructure for portfolio solving. However, *metaSMT* only supports bitvectors, arrays, and uninterpreted functions. Adding new theories to either smt-kit or *metaSMT* would likely be a bigger undertaking than in the comparatively simple Smt-Switch. Neither smt-kit nor *metaSMT* appear to be under active development since 2014 and 2016, respectively.

Two other related tools are PySMT [9] and sbv [7]. PySMT is a solver-agnostic SMT solving API for Python. PySMT has its own term representation and translates formulas to the underlying solvers dynamically once they are asserted. It also uses a class hierarchy to support different solvers. sbv is a solver-agnostic SMT-based verification tool for Haskell. It provides its own datatypes for representing various SMT queries and communicates with solvers through SMT-LIB with pipes. A similar related tool in the context of SAT-solving is PySAT [12], which provides a solver-agnostic Python interface to SAT-solvers.

5 Evaluation

We evaluate Smt-Switch by comparing several state-of-the-art SMT solvers with Smt-Switch (using backends for those same solvers). We use default options for the (underlying) solvers in both cases.[1] We compare on SMT-LIB [2] divisions with bitvectors and arrays, because all solvers support these theories (we use the SMT-LIB frontend for Smt-Switch). We ran on all combinations of incremental vs. non-incremental, and quantified vs. quantifier-free for those theories.[2] We sampled benchmarks from other divisions and obtained comparable results. All experiments were run on a 3.5 GHz Intel Xeon E5-2637 v4 CPU with a timeout of 20 min and a memory limit of 8 Gb.

Our results are shown for non-incremental and incremental benchmarks in Figs. 3 and 4, respectively. The total number of benchmarks in the division is shown next to the logic in the top row. The tables display the number solved

[1] GitHub Commit or Version

Smt-Switch:	17c57ac0f0574cf76125ead56a598fce15c56004
Boolector:	95859db82fe5b08d063a16d6a7ffe4a941cb0f7d
CVC4:	3dda54ba7e6952060766775c56969ab920430a8a
MathSAT:	5.6.4 (697e45d7ef56)
Yices2:	98fa2d882d83d32a07d3b8b2c562819e0e0babd0
Z3:	6cc52e04c3ea7e2534644a285d231bdaaafd8714

.

[2] Note that ABV – incremental, quantified arrays and bitvectors does not have any benchmarks in SMT-LIB.

solver	QF_BV (41713)	QF_ABV (15084)	BV (5846)	ABV (169)
btor	41313 (6.1s)	15045 (1.2s)	5544 (3.8s)	-
ss-btor	41298 (6.0s) -1.8%	15045 (1.2s) 3.5%	5543 (3.7s) -4.0%	-
bitwuzla	41366 (5.8s)	15046 (1.0s)	5557 (4.5s)	-
ss-bitwuzla	41357 (6.2s) 5.8%	15046 (1.0s) 2.2%	5559 (4.5s) 0.6%	-
cvc4	38424 (14.2s)	14480 (8.5s)	5465 (1.1s)	17 (0.1s)
ss-cvc4	38425 (15.2s) 6.9%	14618 (4.4s) -48.1%	5472 (1.0s) -13.6%	17 (0.1s) 6.2%
msat	39609 (17.3s)	14940 (2.7s)	-	-
ss-msat	39598 (18.5s) 6.7%	14937 (2.9s) 6.1%	-	-
yices2	40707 (5.8s)	15015 (2.1s)	-	-
ss-yices2	40695 (6.1s) 4.7%	15007 (2.2s) 9.1%	-	-
z3	40261 (15.6s)	14916 (3.0s)	5522 (1.5s)	44 (2.4s)
ss-z3	40092 (15.9s) 1.7%	14915 (2.8s) -5.3%	5523 (1.8s) 22.5%	44 (1.5s) -40.3%

Fig. 3. Results on non-incremental SMT-LIB benchmarks.

and average runtime on commonly solved instances. The number solved for incremental benchmarks is the sum of all completed satisfiability checking calls. An incremental benchmark is counted as commonly solved for the average runtime calculation if both solvers completed all queries. The `Smt-Switch` rows also show the percent increase in runtime when using `Smt-Switch`.

The data are a rough approximation of the overhead incurred when using `Smt-Switch`. It is rough because our experiment measures parsing time as well as expression construction and solving time. CVC4, for example, uses an ANTLR-based parser, which is a bit slower than other parsers. There is also some noise due to differences in how a solver's API performs relative to its standalone binary. For example, one outlier in the incremental MathSAT QF_BV results skews the overhead significantly for that dataset, though this is due to some difference in the search rather than parsing or expression-building overhead. When `Smt-Switch` solves fewer benchmarks, it is often due to benchmarks that were already close to the timeout with the standalone solver. Overhead is most pronounced on large files, where both parsing and expression-building are exercised more often. Still, the data over many benchmarks and solvers does suggest that the overhead of using `Smt-Switch` is low, generally less than 10% (and some of this is due to parsing differences). Given the flexibility provided by `Smt-Switch`, this level of overhead should be acceptable for many applications.

solver	QF_BV (2589)	QF_ABV (1272)	BV (18)
btor	52375 (6.5s)	3243 (21.5s)	-
ss-btor	52395 (6.9s) 5.8%	3247 (21.1s) -1.8%	-
bitwuzla	52366 (7.1s)	3247 (22.4s)	12445 (1.6s)
ss-bitwuzla	52366 (7.3s) 2.8%	3246 (22.9s) 2.2%	12445 (0.8s) -49.8%
cvc4	51262 (17.5s)	2504 (22.1s)	36097 (54.4s)
ss-cvc4	51252 (15.4s) -12.2%	2740 (2.3s) -89.6%	35852 (26.0s) -52.3%
msat	52333 (8.4s)	3121 (5.7s)	-
ss-msat	52255 (10.0s) 19.4%	3119 (6.3s) 10.5%	-
yices2	52538 (6.2s)	3242 (6.3s)	-
ss-yices2	52490 (6.7s) 7.0%	3243 (6.7s) 6.0%	-
z3	52347 (18.1s)	2911 (31.3s)	37433 (33.6s)
ss-z3	52238 (18.2s) 0.6%	2871 (30.0s) -4.2%	37231 (25.0s) -25.6%

Fig. 4. Results on incremental SMT-LIB benchmarks.

6 Conclusion

We presented `Smt-Switch`, a solver-agnostic C++ API for SMT solving. This system is open-source, supports a variety of solvers and theories, and has already been used in several projects [15,21]. Future work includes i) further reducing the overhead with additional performance tuning; ii) support for more theories (e.g., the floating point and string theories); and iii) providing additional utility functions and classes.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical Report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2021). www.SMT-LIB.org
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: The best of both worlds. Comput. Sci. Eng. **13**(2), 31–39 (2011)
5. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
6. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
7. Erkok, L.: SBV: SMT Based Verification in Haskell (2019). http://leventerkok.github.io/sbv/
8. Free Software Foundation: bison (2021). https://www.gnu.org/software/bison/
9. Gario, M., Micheli, A.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT), pp. 373–384 (2015)
10. Google: GoogleTest. https://github.com/google/googletest
11. Horn, A.: Smt-kit: C++11 library for many sorted logics. http://ahorn.github.io/smt-kit/
12. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: a python toolkit for prototyping with sat oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 428–437. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_26
13. KitWare: CMake. https://cmake.org
14. Krekel, H., Oliveira, B., Pfannschmidt, R., Bruynooghe, F., Laugher, B., Bruhin, F.: pytest 5.4.2 (2004). https://github.com/pytest-dev/pytest
15. Mann, M., et al.: Pono: a flexible and extensible SMT-based model checker. In: CAV. Lecture Notes in Computer Science, Springer (2021)
16. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

17. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR abs/2006. 01621 (2020). https://arxiv.org/abs/2006.01621
18. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 587–595. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_32
19. Riener, H., et al.: metaSMT: focus on your application and not on solver integration. Int. J. Softw. Tools Technol. Transf. **19**(5), 605–621 (2017)
20. Vern Paxson: flex (2021). https://github.com/westes/flex
21. Zohar, Y., Irfan, A., Mann, M., Nötzli, A., Reynolds, A., Barrett, C.: lazybv2int at the SMT Competition 2020. https://github.com/yoni206/lazybv2int (2020)