JavaSMT 3: Interacting with SMT Solvers in Java

Daniel Baier, Dirk Beyer, and Karlheinz Friedberger

LMU Munich, Munich, Germany

Abstract. Satisfiability Modulo Theories (SMT) is an enabling technology with many applications, especially in computer-aided verification. Due to advances in research and strong demand for solvers, there are many SMT solvers available. Since different implementations have different strengths, it is often desirable to be able to substitute one solver by another. Unfortunately, the solvers have vastly different APIs and it is not easy to switch to a different solver (lock-in effect). To tackle this problem, we developed JavaSMT, which is a solver-independent framework that unifies the API for using a set of SMT solvers. This paper describes version 3 of JavaSMT, which now supports eight SMT solvers and offers a simpler build and update process. Our feature comparisons and experiments show that different SMT solvers significantly differ in terms of feature support and performance characteristics. A unifying Java API for SMT solvers is important to make the SMT technology accessible for software developers. Similar APIs exist for other programming languages.

Keywords: Satisfiability Modulo Theories · SMT Solver · Java · API

1 Introduction

SMT solvers [6, 21] are used in a multitude of applications, e.g., in formal software analysis, where automated test-case generation [7, 16, 29, 30], SMT-based algorithms for software verification [10, 34], and interactive theorem proving [27, 44] are used. Applications and users rely on efficiency and expressiveness (supported SMT theories) to compute reasonable results in time. For application developers, the usability and API of the solver are also important aspects, and some features needed in applications, such as interpolation or optimization, are not available in some solvers.

Using the solver's own API directly makes it difficult to switch to another solver without rewriting extensive parts of the application, as there is no standardized binary API for SMT solvers. The SMT-LIB2 standard [4] improves this issue by defining a common language to interact with SMT solvers. However, this communication channel does not define a solver interface for special features like optimization or interpolation.[1] Additionally, the application has to parse the data provided by the SMT solver on its own, and this of course slightly changes from solver to solver.

[1] A proposal for adding interpolation queries exists since 2012, see https://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf .

JavaSMT [37] provides a common API layer across multiple back-end solvers to address these problems. Our Java-based approach creates only minimal overhead, while giving access to most solver features. JavaSMT is available under the Apache 2.0 License on GitHub.[2]

Contribution. Our contribution consists of three parts:

- We integrated more SMT solvers into the API framework JavaSMT (new: Boolector [43], CVC4 [5], and Yices2 [25]).
- We simplified the steps to get started using JavaSMT, by including support for more operating systems (new: MacOS and Windows) and more build techniques (new: Ant and Maven).
- We evaluated the performance of several algorithms for software verification to show that different SMT solvers have different strengths.

Outline. This paper first provides a brief overview of JavaSMT in Sect. 2, explaining the inner structure and features. Sect. 3 discusses the development since the previous publication [37]: more integrated SMT solvers and extended support for operating systems and build processes. Sect. 4 describes a case study, based on SMT-based algorithms [10] in a common verification framework.

Related Work. SMT-LIB2 [4] is the established standard format for exchanging SMT queries. It provides simple usage, is easy to debug, and widely known in the community. However, it requires extra effort to parse and transform formulas in the user application. Features like optimization, interpolation, and receiving nested parts of formulas are not defined by the standard, such that some SMT solvers provide their own individual solution for that. Alternatively, several SMT solvers already come with their special bindings for some programming languages. Most SMT solvers are written in C/C++, so interacting with them in these low-level languages is the easiest way. However, the support for higher-level languages is sparse. The most prominent language binding for several SMT solvers is Python, as it directly allows the access to C code and avoids automated memory management operations like asynchronous garbage collection. Bindings for Java are available for some SMT solvers, such as MathSAT5 and Z3, but missing, unsupported, or unmaintained for others, such as Boolector and CVC4.

In the following, we discuss libraries, similar to JavaSMT, that provide access to several underlying SMT solvers via a common user interface in different popular languages, and their binding mechanism, i.e., whether the solver interaction is based on a native interface or text-based on SMT-LIB2. With SMT-LIB2, an arbitrary SMT solver can be queried, but the interaction happens through communicating processes and the solver is mostly limited to features defined in the standard. Accessing a native interface directly allows to support more features of the underlying solver, e.g., using callbacks, simplifying formulas, or eliminating quantifiers.

Table 1 provides an overview of the libraries for interacting with SMT solvers. We enumerate several special features that are not available in some libraries,

[2] https://github.com/sosy-lab/java-smt

Table 1: Comparison of different interface libraries for SMT solvers

	Reference	Language	Native API	SMT-LIB2	Unsat Cores	Interpolation	Optimization	Formula Decomposition	Project - Forks	Project - Stars	Project - Year Latest Commit
JavaSMT	[37]	Java	✓	✗	✓	✓	✓	✓	22	90	2021
PySMT	[28]	Python	✓	✓	✓	✓	✓	✗	99	363	2021
SMT Kit		C/C++	✓	✗	✓	✗	✗	✗	4	36	2014
Smt-Switch	[38]	C/C++	✓	✗	✓	✓	✗	✓	15	40	2021
jSMTLIB	[20]	Java	✗	✓	✓	✗	✗	✗	15	21	2020
metaSMT	[45]	C/C++	✗	✓	✗	✗	✓	✓	19	43	2016
rsmt2		Rust	✗	✓	✓	✗	✗	✗	10	24	2021
SBV		Haskell	✗	✓	✓	✗	✓	✗	17	134	2021
Scala SMT-LIB		Scala	✗	✓	✓	✗	✗	✓	18	44	2021
ScalaSMT	[17]	Scala	✗	✓	✗	✗	✗	✓	1	4	2019
what4		Haskell	✗	✓	✓	✗	✗	✗	5	97	2021

such as unsat cores, interpolation, or optimization queries. Those features depend on the support by the underlying SMT solver, but can be provided in general by an API on top of them. Most libraries use their own formula representation and not just wrap the objects provided by the SMT solver. This potentially allows for easier formula decomposition and inspection, e.g., by using the visitor pattern. JavaSMT directly provides formula decomposition if available in the SMT solver. The provided numbers of forks and stars of the project repositories on GitHub or Bitbucket can be seen as a measurement of popularity.

PySMT [28] is a Python-based project and aims at rapid prototyping of algorithms using the native API of the installed SMT solvers. It has the ability to perform formula manipulation without a back-end SMT solver and additionally supports the conversion of boolean formulas to plain SAT problems and then apply a SAT solver or a BDD library. This approach comes with the drawback of a noticeable memory overhead and performance of an interpreted language. metaSMT [45], SMT Kit, and Smt-Switch [38] provide solver-agnostic APIs for interacting with various SMT solvers in C/C++ to focus on the application instead of the solver integration. jSMTLIB [20], Scala SMT-LIB, and ScalaSMT [17] are solver-independent libraries written in Java or Scala and interact via SMT-LIB2 with SMT solvers. Scala SMT-LIB and ScalaSMT allow to use an additional domain-specific language to interact with SMT solvers and rewrite Scala syntax into valid SMT-LIB2 and back. Both partially extend the SMT-LIB2 standard, e.g., by offering the ability to overload operators or receive interpolants. SBV and what4 are generic Haskell libraries based on process interaction via SMT-LIB2 and support several SAT and SMT solvers. rsmt2 offers a generic Rust library that currently supports three SMT solvers.

2 JavaSMT's Architecture and Solver Integration

In the following, we describe the architecture of JavaSMT and its main concepts. Afterwards, we give an overview of the integrated SMT solvers and their features. The architecture did not significantly change, but we added a few new SMT solvers, as shown in Fig. 1.

Architecture. JavaSMT provides a common API for various SMT solvers. The architecture, shown in Fig. 1, consists of several components: As common context, we use a `SolverContext` that loads the underlying SMT solver and defines the scope and lifetime of all created objects. As long as the context is available, we track memory regions of native SMT-solver libraries. When the context is closed, the corresponding memory is freed and garbage collection wipes all unused objects. Within a given context, JavaSMT provides `FormulaManager`s for creating formulas in various theories and `ProverEnvironment`s for solving SMT queries.

A `FormulaManager` allows to create symbols and formulas in the corresponding theories and provides a type-safe way to combine symbols and formulas in order to encode a more complex SMT query. We support the structural analysis (like splitting a formula into its components or counting all function applications in a formula) and transformations (like substituting symbols or applying equisatisfiable simplifications) of formulas.

Each `ProverEnvironment` represents a solver stack and allows to push/pop boolean formulas and check them for satisfiability (the hard part). This follows the idea of incremental solving (if the underlying SMT solver supports it). After a satisfiability check, the `ProverEnvironment` provides methods to receive a model, interpolants, or an unsatisfiable core for the given formula.

JavaSMT guarantees that formulas built with a single `FormulaManager` can be used in several `ProverEnvironment`s, e.g., the same formula can be pushed onto and solved within several distinct `ProverEnvironment`s. The interaction with independent `ProverEnvironment`s works from multiple threads. However, some SMT solvers require synchronization (e.g., locking for an interleaved usage) and other solvers do not require external synchronization (this allows concurrent usage).

SMT-Solver Integration and Bindings. Of the eight SMT solvers that are available in JavaSMT, only Princess [46] and SMTInterpol [18] were 'easy' to integrate, as they are written in Scala and Java, respectively. Those solvers also use the available memory management and garbage collection of the Java Virtual Machine (JVM). All other solvers are written in C/C++ and need a Java Native Interface (JNI) wrapper to interface with JavaSMT. Z3 [40] and CVC4 [5] provide their own Java wrappers, while the bindings used for MathSAT5 [19], Boolector [42], and Yices2 [25] are maintained by us. Those bindings are self-written or partially based on a version of the solver developers, extended with exception handling, and usable for debugging in JavaSMT. By providing language bindings for solvers in our library, we relieve the solver developers from this burden, and the implementation of exception handling and memory management is done in an efficient and common manner across several solvers.

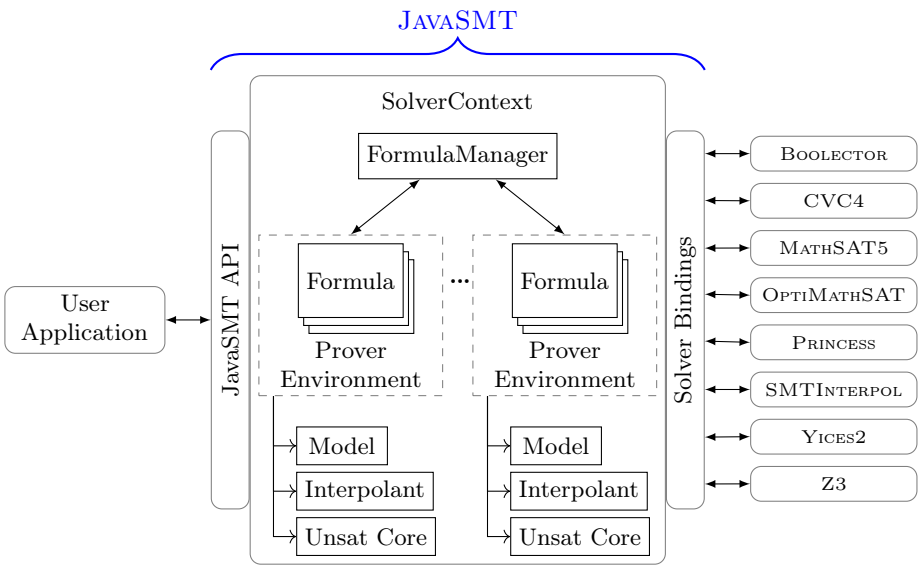

Fig. 1: Overview of JavaSMT

Table 2: Size (LOC) of the Java-based solver wrappers and native solver bindings

	Boolector	CVC4	MathSAT5	OptiMathSAT	Princess	SMTInterpol	Yices2	Z3
Java-based Wrapper	1644	1918	3229	3229	2042	2117	2728	2674
JNI Bindings	3136		1388	1508			1598	

Table 2 lists the size (lines of code) of the wrappers to integrate each solver in JavaSMT, in order to get a rough impression of the required effort to get a solver and its bindings usable in JavaSMT. The size information consists of two parts, namely the JNI bindings that are written in C/C++ and the Java code that implements the necessary interfaces of JavaSMT. An expressive solver API (like MathSAT5 or OptiMathSAT [47]) needs more code for their binding, with only a small increment in complexity compared to other solver bindings.

Note that the evolution of JavaSMT depends on the evolution of the underlying SMT solvers. Z3 is well-known, has a large user group, and an active development team. Yet, interpolation support for Z3 was dropped with release 4.8.1.[3] Bitwuzla [41] is the successor of the SMT solver Boolector, for which the developers still provide small fixes. Bitwuzla can be supported in JavaSMT in the future. CVC4 has been developed further to CVC5. However, the maintainers

[3] https://github.com/Z3Prover/z3/releases/tag/z3-4.8.1

dropped the existing Java API, partially because of issues with the Java garbage collection, and plan to replace it.[4] Yices2 is also actively maintained and adds new features regularly. For example, the developers added support for third-party SAT solvers such as CaDiCaL and CryptoMiniSat [48].

3 New Contributions in JavaSMT 3

This section describes the improvements over the JavaSMT version from five years ago [37], split into two parts. First, we describe newly integrated solvers and theory features. Second, we provide information about the build process.

Support for Additional SMT Solvers. JavaSMT 3 provides access to eight SMT solvers. Besides the solvers that were already integrated before, MathSAT5, OptiMathSAT, Z3, Princess, and SMTInterpol, the user can now additionaly use Boolector, CVC4, and Yices2. Table 3 lists available theories and important features supported by each individual solver. Boolector is specialized in Bitvector-based theories, but does not support the Integer theory. It is shipped with several back-end SAT solvers, from which the user can choose a favorite: CaDiCaL, CryptoMiniSat [48], Lingeling, MiniSat [26], and PicoSAT [13]. All solvers support the input of plain SMT-LIB2 formulas. However, the feature most requested by JavaSMT users is the input and output of SMT queries via the API, i.e., parsing and printing boolean formulas for a given context. This feature is required for (de-)serializing formulas to disk, for network transfer, and to translate formulas from one solver to another one. This feature is unfortunately missing for the newly integrated solvers, even though each solver internally already contains code for parsing and printing SMT-LIB2 formulas.

For formula manipulation, JavaSMT accesses the components of a formula, e.g., operators and operands. We do not require full access to the internal data structures of the SMT solvers, but only limited access to the most basic parts. Only Boolector does not provide the necessary API.

Build Simplification. JavaSMT 3 also supports more operating systems than before. Besides the existing support for Linux, we started to provide pre-compiled binaries for MacOS and Windows for more than half of the available solvers. This simplifies the initial steps for new users, which previously were required to compile and link the solvers on their own. This was an involving task, because of the diversity of build systems and dependencies of each solver.

In addition to this, we now offer direct support for two popular build systems for Java applications, namely Ant and Maven. JavaSMT comes with several examples and documentation, such that the mentioned build systems can be used to set up JavaSMT in a ready-to-go state on most systems. This eliminates the need for complex manual set up of dependencies and eases the use of JavaSMT and the SMT solvers.

[4] https://github.com/cvc5/cvc5/issues/5018

Table 3: SMT theories and features supported by SMT solvers in JavaSMT 3

		Boolector	CVC4	MathSAT5	OptiMathSAT	Princess	SMTInterpol	Yices2	Z3
SMT Theories	Integer	✗	✓	✓	✓	✓	✓	✓	✓
	Rational	✗	✓	✓	✓	✓	✓	✓	✓
	Array	✓	✓	✓	✓	✓	✓	✗	✓
	Bitvector	✓	✓	✓	✓	✓	✗	✓	✓
	Float	✗	✓	✓	✓	✗	✗	✗	✓
	UF	✓	✓	✓	✓	✓	✓	✓	✓
	Quantifier	✓	✓	✗	✗	✓	✗	✓	✓
Features	Incremental Solving	✓	✓	✓	✓	✓	✓	✓	✓
	Model	✓	✓	✓	✓	✓	✓	✓	✓
	Assumption Solving	✓	✗	✓	✓	✗	✗	✓	✓
	Interpolation	✗	✗	✓	✓	✓	✓	✗	✗
	Optimization	✗	✗	✗	✓	✗	✗	✗	✓
	UnsatCore	✗	✓	✓	✓	✓	✓	✓	✓
	UnsatCore with Assumptions	✗	✗	✓	✓	✗	✓	✓	✓
	SMT-LIB2 (plain text input)	✓	✓	✓	✓	✓	✓	✓	✓
	SMT-LIB2 (via API)	✗	✗	✓	✓	✓	✓	✗	✓
	Quantifier Elimination	✗	✓	✗	✗	✓	✗	✗	✓
	Formula Decomposition	✗	✓	✓	✓	✓	✓	✓	✓

4 Evaluation

Frameworks that provide a unified API to SMT solvers (such as JavaSMT, PySMT, and ScalaSMT) are necessary because the characteristics of the SMT solvers vary a lot. In the evaluation we provide support for this argument.

We inlined a discussion of the features already in the previous section. Table 3 provides the overview of supported theories and shows that certain theories are available only for a subset of SMT solvers. The table also shows that there are several features that restrict the choice of SMT solvers for certain applications.

In terms of performance, we evaluate JavaSMT 3 as a component of CPAchecker [11], which is an open-source software-verification framework [5] that provides a range of different SMT-based algorithms for program analysis [10] and encoding techniques for program control flow [8, 12]. We compare three well-known and successful SMT-based algorithms for software model checking and show that — when using the same algorithm and identical problem encoding — the performance result of an analysis depends on the used SMT solver. Some

[5] https://cpachecker.sosy-lab.org

algorithms depend on special features of the SMT solver, e.g., to provide a certain type of formula (such as interpolants) and operation on a formula (such as access to subformulas). There are SMT solvers that can not be used for some algorithms.

We aim to show that depending on the feature set of the SMT solvers, it is important to support a common API, and additionally, that using the text-based interaction via SMT-LIB2 is not an efficient solution, when it comes to formula analysis like adding additional information into a formula.

Benchmark Programs. We evaluate the usage of JavaSMT on a large subset of the SV-benchmark suite [6] containing over 1 000 verification tasks. To have a broad variation of benchmark tasks, we include reachability problems from the categories *BitVectors*, *ControlFlow*, *Heap*, and *Loops*.

BitVectors depends on bit-precise reasoning and thus, the SMT solver needs to support Bitvector logic. *Heap* depends on modeling heap memory access, e.g., which is either encoded in the theory of Arrays or as Uninterpreted Functions. The category *Loops* contains tasks where the state space is potentially quite large.

Experimental Setup. We run all our experiments on computers with Intel Xeon E3-1230 v5 CPUs with 3.40 GHz, and limit the CPU time to 15 min and the memory to 15 GB. We use CPAchecker revision r36714, which internally uses JavaSMT 3.7.0-73. The time needed for transforming the input program into SMT queries is rather small compared to the analysis time. Additionally, the progress of an algorithm depends on the result (e.g., model values or interpolants) returned from an SMT solver, thus we do not explicitly extract the run time required by the SMT solver itself for answering the satisfiability problem, but we measure the complete CPU time of CPAchecker for the verification run.

Analysis Configuration. We use three different SMT-based algorithms for software verification [10]. The first approach is bounded model checking (BMC) [14, 15], which is applied in software and hardware model checking since many years. In this approach, a verification problem is encoded as single large SMT query and given to the SMT solver. No further interaction with the SMT solver is required. In our evaluation, we use a loop bound $k = 10$, which limits the size of the SMT query.

The second approach is k-induction [9, 24], which extends BMC, and which uses auxiliary invariants to strengthen the induction hypothesis. In this approach, the algorithm generates several SMT queries (base case, inductive-step case, each with increasing loop bound) and uses an invariant generator that provides the auxiliary invariants. We use an interval-based invariant generator that provides not only the invariants, but also information about pointers and aliases, which must be inserted into the SMT formula using the formula visitor.

The third approach is predicate abstraction [3, 12, 31, 35], which uses Craig interpolation [22, 32, 39] to compute predicate abstractions of the program. This approach does not only query the SMT solver multiple times, but also uses (sequential) interpolation, which is currently supported only by MathSAT5, Princess, and SMTInterpol.

[6] https://github.com/sosy-lab/sv-benchmarks

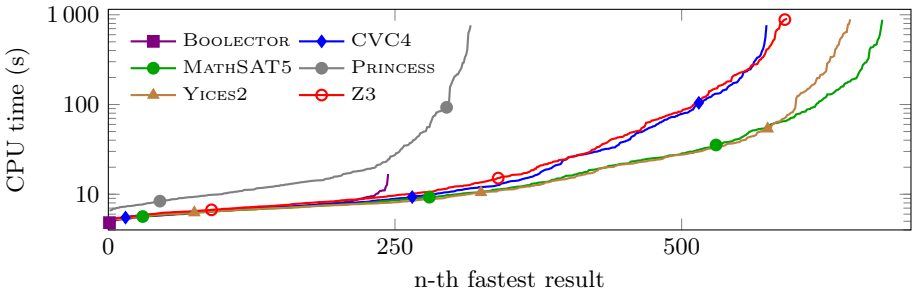

Fig. 2: Quantile plot for the runtime of k-induction with several SMT solvers

All approaches are executed in two configurations, depending on the used encoding of program statements: First, we apply a bitvector-based encoding that precisely models bit-precise arithmetics and overflows of the program. Second, an encoding based on linear integer arithmetic is used, which approximates the concrete program execution and is sufficient for some programs.

Solver Configuration. Overall, we aim to show that each solver provides a unique fingerprint of features and results. We aim for a precise program analysis and thus configure the SMT solvers to be as precise as possible, but with a reasonable configuration for each solver (i.e., without using a feature combination that is unsupported by the SMT solver).

SMTInterpol does not support efficient solving of SMT queries in Bitvector logic, thus, it is configured to use only Integer logic. Boolector misses Integer logic, thus, it is applied only to the bit-precise configurations. Additionally, this SMT solver does not support formula inspection and decomposition, which is required by several components in k-induction, e.g., to encode proper pointer aliasing for the program analysis. While the code for formula inspection is called quite often, its influence on the results for the selected benchmark tasks is small. In order to be comparable as far as possible, we deactivate pointer aliasing when using Boolector. Yices2 misses proper support for Array logic, thus, we use a UF-based encoding of heap memory as alternative for this solver, which results in a slightly unsound analysis, but a comparable formula size and run time.

Results and Discussion. Figure 2 provides the quantile plot for the results of k-induction configurations with bit-precise encoding using several SMT solvers. The plot shows the CPU time for valid analysis results, i.e., proofs or counterexamples found, for both expected results true and false. We aim for providing all result that are useful for a user and do not show results where the tool (or SMT solver) crashes or runs out of resources. We do not subtract the run time required for the framework CPAchecker itself (which starts a Java virtual machine), as we assume it to be comparable per program task; we are only interested in the asymptotics in this evaluation. The overall performance of SMT solvers is similar for simple verification tasks, i.e., those with a small run time in the analysis. For difficult tasks with harder SMT queries, the differences of the SMT solvers emerge. When applying k-induction, the analysis inserts additional constraints into the

Table 4: Run time for using different SMT solvers for bounded model checking ('BMC'), k-induction ('KI'), and predicate abstraction ('PA') with the theories of Bitvectors ('BV') and Integers ('Int'); CPU time given in seconds with two significant digits, ' TO' indicates timeouts (900 s), ' ERR' indicates errors, and empty cells indicate that the theory or interpolation was not supported

Verification Task	s3_srvr.blast.07.i.cil-2	byte_add_1-1	ps6-ll_valuebound100	s3	diamond_1-1	modulus-2	jain_5-2	s3_clnt_1.cil-2	diskperf_simpl1.cil	rule57_ebda_blast
Algorithm	BMC	BMC	KI	KI	KI	KI	PA	PA	PA	PA
Encoding	Int	BV	Int	Int	BV	BV	Int	Int	BV	BV
Boolector		**5.8**			ERR	ERR				
CVC4	340	6.4	TO	TO	110	TO				
MathSAT5	17	7.8	200	53	60	54	TO	**11**	**16**	**7.1**
Princess	TO	TO	530	TO	260	TO	**38**	160	TO	ERR
SMTInterpol	50		TO	140			TO	13		
Yices2	**14**	7.7	340	**23**	**34**	28				
Z3	15	6.7	**130**	66	43	**21**				

SMT formula and requires the SMT solver to allow access to components of existing formulas. As Boolector misses this specific feature, k-induction cannot be very effective here. Other SMT solvers are the preferred choice.

Table 4 contains some example tasks from all used algorithms and encodings, where the difference between distinct SMT solvers is noteworthy. Choosing the optimal SMT solvers for an arbitrary problem task is not obvious.

5 Conclusion

We contribute JavaSMT 3, the third generation of the unifying Java API for SMT solvers. The package now contains more SMT solvers, an improved build process, and support for MacOS and Windows. The project has over 20 contributors, 2 500 commits, and overall about 41 000 lines of code.[7] JavaSMT is used in Java applications (e.g., [23, 33, 36]) as a solution to combine convenience and performance for the interaction with SMT solvers, or to switch between different solvers and compare them [11, 49]. The most prominent application using JavaSMT is the verification framework CPAchecker (a widely-used software

[7] https://www.openhub.net/p/java-smt

project [8] with 73 forks on GitHub alone), for which JavaSMT was originally developed. In the future, we plan to support more SMT solvers, operating systems, and hardware architectures, while keeping the user interface stable. We hope that even more researchers and developers of Java applications can benefit from SMT solving via a convenient and powerful API.

Data Availability Statement. All benchmark tasks for evaluation, configuration files, a ready-to-run version of our implementation, and tables with detailed results are available in our reproduction package on Zenodo as virtual machine [1] and as ZIP archive [2]. The source code of the open-source library JavaSMT [37] is available in the project repository; see `https://github.com/sosy-lab/java-smt`.

References

1. Baier, D., Beyer, D., Friedberger, K.: Reproduction package (VM) for article 'JavaSMT 3: Interacting with SMT solvers in Java'. Zenodo (2021). https://doi.org/10.5281/zenodo.4708050
2. Baier, D., Beyer, D., Friedberger, K.: Reproduction package (ZIP) for article 'JavaSMT 3: Interacting with SMT solvers in Java'. Zenodo (2021). https://doi.org/10.5281/zenodo.4865175
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Proc. TACAS. pp. 268–283. LNCS 2031, Springer (2001). https://doi.org/10.1007/3-540-45319-9_19
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Proc. SMT (2010)
5. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV. pp. 171–177. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
6. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
7. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
8. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
9. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
10. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
11. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

[8] `https://github.com/sosy-lab/cpachecker`

12. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)
13. Biere, A.: PicoSAT Essentials. JSAT **4**(2-4), 75–97 (2008). https://doi.org/10.3233/SAT190039
14. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
15. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
16. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
17. Cassez, F., Sloane, A.M.: ScalaSMT: Satisfiability modulo theory in Scala (tool paper). In: Proc. SCALA. pp. 51–55. ACM (2017). https://doi.org/10.1145/3136000.3136004
18. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Proc. SPIN. pp. 248–254. LNCS 7385, Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19
19. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
20. Cok, D.R.: jSMTLIB: Tutorial, validation, and adapter tools for SMT-LIBv2. In: Proc. NFM. pp. 480–486. LNCS 6617, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_36
21. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. JSAT **9**, 207–242 (2016)
22. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
23. Demarchi, S., Menapace, M., Tacchella, A.: Automating elevator design with satisfiability modulo theories. In: Proc. ICTAI. pp. 26–33. IEEE (2019). https://doi.org/10.1109/ICTAI.2019.00013
24. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS. pp. 351–368. LNCS 6887, Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26
25. Dutertre, B.: Yices 2.2. In: Proc. CAV. pp. 737–744. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
26. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT. pp. 502–518. LNCS 2919, Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
27. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: VerifyThis: Verification competition with a human factor. In: Proc. TACAS. pp. 176–195. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_12
28. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proc. SMT (2015)
29. Godefroid, P., Sen, K.: Combining model checking and testing. In: Handbook of Model Checking, pp. 613–649. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_19
30. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS. The Internet Society (2008)
31. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10

32. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
33. Ibrhim, H., Khattab, S., Elsayed, K., Badr, A., Nabil, E.: A formal methods-based rule verification framework for end-user programming in campus building automation systems. Building and Environment **181**, 106983 (2020). https://doi.org/10.1016/j.buildenv.2020.106983
34. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
35. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15
36. Joshaghani, R., Black, S., Sherman, E., Mehrpouyan, H.: Formal specification and verification of user-centric privacy policies for ubiquitous systems. In: Proc. IDEAS. pp. 31:1–31:10. ACM (2019). https://doi.org/10.1145/3331076.3331105
37. Karpenkov, E.G., Friedberger, K., Beyer, D.: JavaSMT: A unified interface for SMT solvers in Java. In: Proc. VSTTE. pp. 139–148. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_11
38. Mann, M., Wilson, A., Tinelli, C., Barrett, C.W.: SMT-Switch: A solver-agnostic C++ API for SMT solving. arXiv/CoRR (2007.01374) (2020), https://arxiv.org/abs/2007.01374
39. McMillan, K.L.: Interpolation and model checking. In: Handbook of Model Checking, pp. 421–446. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_14
40. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
41. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. arXiv/CoRR (2006.01621) (2020), https://arxiv.org/abs/2006.01621
42. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/sat190101
43. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
44. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer (2002). https://doi.org/10.1007/3-540-45949-9
45. Riener, H., Haedicke, F., Frehse, S., Soeken, M., Große, D., Drechsler, R., Fey, G.: metaSMT: Focus on your application and not on solver integration. Int. J. Softw. Tools Technol. Transf. **19**(5), 605–621 (2017). https://doi.org/10.1007/s10009-016-0426-1
46. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proc. LPAR. pp. 274–289. LNCS 5330, Springer (2008). https://doi.org/10.1007/978-3-540-89439-1_20
47. Sebastiani, R., Trentin, P.: OptiMathSAT: A tool for optimization modulo theories. In: Proc. CAV. pp. 447–454. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_27
48. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Proc. SAT. pp. 244–257. LNCS 5584, Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_24
49. Sprey, J., Sundermann, C., Krieter, S., Nieke, M., Mauro, J., Thüm, T., Schaefer, I.: SMT-based variability analyses in FeatureIDE. In: Proc. VaMoS. pp. 6:1–6:9. ACM (2020). https://doi.org/10.1145/3377024.3377036