JavaSMT: A Unified Interface for SMT Solvers in Java

Egor George Karpenkov[1,2], Karlheinz Friedberger[3], and Dirk Beyer[3]

[1] Univ. Grenoble Alpes, VERIMAG, 38000 Grenoble, France
[2] CNRS, VERIMAG, 38000 Grenoble, France
[3] University of Passau, 94032 Passau, Germany

Abstract. Satisfiability Modulo Theory (SMT) solvers received a lot of attention in the research community in the last decade, and consequently their expressiveness and performance have significantly improved. In the areas of program analysis and model checking, many of the newly developed tools rely on SMT solving. The SMT-LIB initiative defines a common format for communication with an SMT solver. However, tool developers often prefer to use the solver API instead, because many features offered by SMT solvers such as interpolation, optimization, and formula introspection are not supported by SMT-LIB directly. Additionally, using SMT-LIB for communication incurs a performance overhead, because all the queries to the solver have to be serialized to strings. Yet using the API directly creates the problem of a solver lock-in, which makes evaluating a tool with different solvers very difficult. We present JavaSMT, a library that exposes a solver-independent API layer for SMT solving. Our library aims to close the gap between API-based and SMT-LIB-based communication, by offering a large set of features with minimal performance overhead. JavaSMT has been used internally in CPAchecker since inception, and has been heavily tested in different verification algorithms. The library is available from its Github website https://github.com/sosy-lab/java-smt.

1 Introduction

During the last decade, SMT solvers have demonstrated an impressive increase in expressiveness (many supported theories) and efficiency (much larger scale of queries that can be answered within a small time-frame). As a consequence, many tools for software verification rely on an SMT solver as a back-end.

The SMT-LIB [3] initiative defines a common interface language for SMT solvers, much like SQL standardizes the interface to a relational database. However, from the perspective of a tool developer, using the textual SMT-LIB communication channel is often suboptimal. Firstly, it does not expose all the

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR" and from the Free State of Bavaria.

S. Blazy and M. Chechik (Eds.): VSTTE 2016, LNCS 9971, pp. 139–148, 2016.
DOI: 10.1007/978-3-319-48869-1_11

features that modern solvers offer: interpolation[1] multiple independent solvers, formula introspection, and optimization modulo theories are not included in SMT-LIB 2.0. It is also not possible to conditionally *store* formulas for future reuse and remove them when they are no longer needed. Secondly, such a textual communication can be very inefficient, because all queries to the solver have to be serialized to strings, and all of the solver output has to be parsed. For a tool that poses a large number of simple queries (such as in PDR [2]), parsing and serialization can become a performance bottleneck.

However, when using a solver API directly, users face the problem of "solver lock-in", which makes it difficult to evaluate different SMT solvers or to switch to a different SMT solver without rewriting a large chunk of the application.

We propose JavaSMT, a library that exposes a common API layer across several back-end solvers. It is written in Java and is available under the Apache 2.0 License on GitHub (https://github.com/sosy-lab/java-smt). JavaSMT communicates with solvers using their API, and imposes only a minimal amount of overhead. For the solvers that are implemented in Java the exposed API is used directly, and for the solvers in other languages we integrated JNI bindings.

Outlook. This paper refers to JavaSMT v1.01 [2]. The contributions of this paper are structured as follows: First, we describe the features that JavaSMT exposes in Sect. 2. Second, we present the project structure and the requirements for adding a new solver into JavaSMT in Sect. 3. Third, Sect. 4 discusses the strategies for managing memory of the JNI bindings, and the associated performance problem. Finally, we present a case study based on the Houdini algorithm [4] in Sect. 5, and conclude by comparing JavaSMT to related projects and discussing possible future work in Sect. 6.

2 Features

JavaSMT currently provides access to five different SMT solvers: MathSAT [1], OptiMathSAT [17], Z3 [14], SMTInterpol [12], and Princess [16]. Table 1 lists the theories and features that are supported by these solvers.

Formula Representation. To keep the memory overhead low, JavaSMT does not store its own internal representation of the formulas, but keeps only one single pointer to each formula in the solver's memory, possibly with an additional pointer to the current solver context. Consequently, the memory footprint of JavaSMT is proportional to a small constant multiplied by the number of formulas that the client application needs a reference to, *regardless* of the size of the constructed formulas. This choice ensures high performance, but obstructs transferring formulas between different contexts for different operations, such as checking satisfiability with Z3 and performing interpolation with SMTInterpol. For such inter-solver translations we use SMT-LIB serialization.

[1] A proposal draft [11] exists since 2012.
[2] https://github.com/sosy-lab/java-smt/releases/tag/1.0.1.

Table 1. Theories and features supported by different solvers

	MathSAT	OptiMathSAT	Z3	SMTInterpol	Princess
Integer	+	+	+	+	+
Rational	+	+	+	+	-
Array	+	+	+	+	+
Bitvector	+	+	+	-	-
Float	+	+	-	-	-
Unsat Core	+	+	+	+	-
Partial Models	-	-	+	-	+
Assumptions	+	+	+	+	+
Quantifiers	-	-	+	-	+
Interpolation (Tree/Sequential)	+	+	+	+	+
Optimization	-	+	+	-	-
Incremental Solving	+	+	+	+	+
SMT-LIB2	+	+	+	+	+

Type Safety. Using and enforcing types is beneficial for a software library, because it guarantees the absence of errors that are caused by incorrect type usage *at compile time* and can increase the level of trust in the software. Improving such confidence is particularly important for tools for software verification, because the verdict of such tools is only reliable if all components operate correctly ("who verifies the software verifier").

JavaSMT uses the Java type system to differentiate between the different sorts of formulas (e.g., `BooleanFormula` and `IntegerFormula`) and guarantees that all operations respect the formula type. The typed interface avoids incorrect operations (such as adding integers to Booleans), which would not pass the compiler. Type safety also extends to model evaluation: for example, evaluating an `IntegerFormula` is guaranteed at compile time to return a `BigInteger`.

Formula Introspection. In many applications, formula introspection is a required feature. For instance, an analysis might wish to re-encode expensive non-linear operations as uninterpreted functions, or to find and rename all variables used in the formula.

In our experience with formula introspection and transformation code in CPAchecker [6], we have discovered that writing *correct and robust* formula-traversing code can be very challenging, due to:

- cases missed by the client, e.g., an unexpected `XOR`,
- incorrect assumptions by the client, such as assuming that the input formula has no quantifiers,
- not performing memoization for recursive traversals, resulting in exponential blow-up on formulas represented as directed acyclic graphs, or
- performing recursive traversal using recursion, since it can result in stack-overflow exceptions on large formulas.

In order to decrease the likelihood of such bugs, we use the Visitor design pattern (cf. [9], Chap. 5) for formula traversal and transformation. Two visitor interfaces are exposed: `BooleanFormulaVisitor` and `FormulaVisitor`. The Boolean visitor requires implementations for Boolean primitives that can occur in the formulas (equality, implication, etc.) and matches all other formulas as atoms. It is useful for transformation of the Boolean structure of the formula, such as a conversion to negation normal form. The `FormulaVisitor` does not explicitly require matching each possible function, but provides an enumeration consisting of most common function declarations (addition, subtraction, comparison, etc.) and can be used to recursively traverse the entire formula, e.g., in order to find all used variables.

Our experience shows that such an approach leads to considerably safer code as compared to direct formula manipulation.

3 Project Architecture

The overall structure of the library is shown in Fig. 1. An interaction with the JavaSMT library starts with a `SolverContextFactory`, which is used to create a `SolverContext` object, encapsulating a context for a particular solver. All further interaction is performed through the `SolverContext` class, which exposes the features outlined in Sect. 2. Instances of `SolverContext` are *not* thread-safe, and should be accessed only from a single thread. However, separate contexts are independent from each other and can be safely used from different threads, provided that the underlying solver supports multithreading on different contexts.

An interface to every represented solver is implemented as a separate package with an entry class that implements the `SolverContext` API.

4 Memory Management

Different SMT solvers resort to different strategies for memory management. The solvers running in managed environments (e.g., SMTInterpol and Princess running on JVM) use the available garbage collector, while solvers exposing a C API have to expose the memory management API to a user. The underlying problem is that for a library that exposes its API through the native non-managed language, it is *impossible* to know whether a previously returned object is still referenced by the client application, or whether it can be deleted.

MathSAT exposes a “manual” garbage-collection interface, which removes all formulas except those that are specifically requested to be kept. This requires an application to keep track of all created objects that can still be referenced.

Z3 uses a reference-counting approach, where an object is considered unreachable whenever its reference count reaches zero. While this interface can be effectively used from C++ to offer automatic memory management using RAII (incrementing references in constructors, and decrementing in destructors), using it in an *efficient* and correct way is surprisingly difficult from Java.

The official Z3 Java API is using Java finalizers [3] to decrement the references, explicitly performing locking on the queue of references that need to be decremented. Unfortunately, finalizers are known to have a very severe memory and performance penalty (cf. [10], Chap. 2.7). Thus we have developed our own Z3 JNI bindings with a memory strategy based on using `PhantomReference` and `ReferenceQueue`, provided by the JDK to get a more fine-grained control over the garbage collection.

We present the performance evaluation of three different memory managing strategies for Z3: (1) using the official Z3 API, which relies on finalizers, (2) using our phantom reference-based implementation, and (3) not closing resources at all. We have chosen a benchmark setup that runs a program analysis with local policy iteration [8] on the SV-COMP [5] data set. Obtained results are shown in Fig. 2. Unsurprisingly, the approach using finalizers has the worst performance by far, with performance penalty often eclipsing the analysis time, and a very large memory consumption. The no-GC approach minimizes both memory and time consumption. We attribute the high performance of the no-GC approach to the hash-consing used in Z3, which results in no additional memory consumption for ASTs that were previously already constructed.

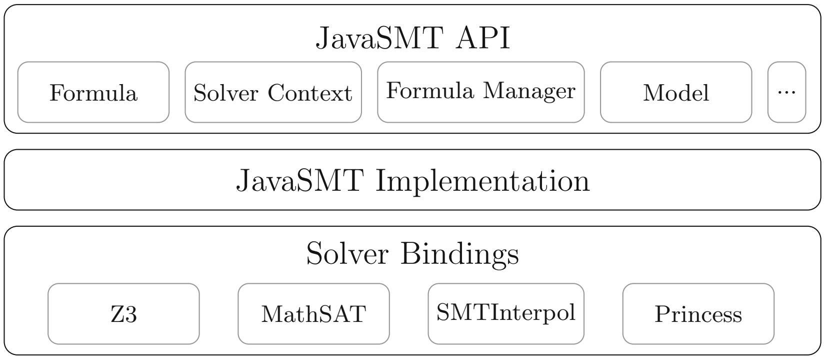

Fig. 1. JavaSMT Architecture

[3] Since the publication of this paper, Z3 bindings were updated by one of the authors of this paper to use a more efficient memory management strategy.

5 Case Study: Inductive Formula Weakening

To give a tour of the library, we present a usable implementation of the inductive-invariant synthesis algorithm HOUDINI [4]. In order to provide the context, we include a brief background that explains the algorithm and its motivation.

Background. We consider a program that manipulates a set X of variables. The program is defined by the initial condition $I(X)$ and the transition relation $\tau(X, X')$. Both I and τ are quantifier-free first-order formulas.

A lemma F is called inductive with respect to τ if it implies itself over the primed variables after the transition:

$$\forall X, X' : F(X) \wedge \tau(X, X') \implies F(X') \tag{1}$$

Inductiveness can be checked with a single query to an SMT solver. The lemma F is inductive with respect to τ iff the following formula (2) is unsatisfiable:

$$F(X) \wedge \tau(X, X') \wedge \neg F(X') \tag{2}$$

The HOUDINI algorithm finds a maximal inductive subset of a given set L of *candidate* lemmas which satisfies the initial condition $I(X)$. Firstly, it filters out all lemmas from L which are not implied by I. Then, it repeatedly checks $\bigwedge L$ for inductiveness using (2), and updates L to exclude the lemmas that give rise to counterexamples-to-induction. At the end the algorithm terminates with an inductive subset $L_I \subseteq L$.

Counterexamples-to-induction are derived from a *model* returned by an SMT solver in response to a query in (2) (such a model exists iff the conjunction of lemmas is not inductive). Given a model $\mathcal{M}$, the HOUDINI algorithm filters out all lemmas $l \in L$ for which $\mathcal{M} \models \neg l(X')$ holds. After such filtering is applied in a fixed-point manner, a (possibly empty) inductive subset remains.

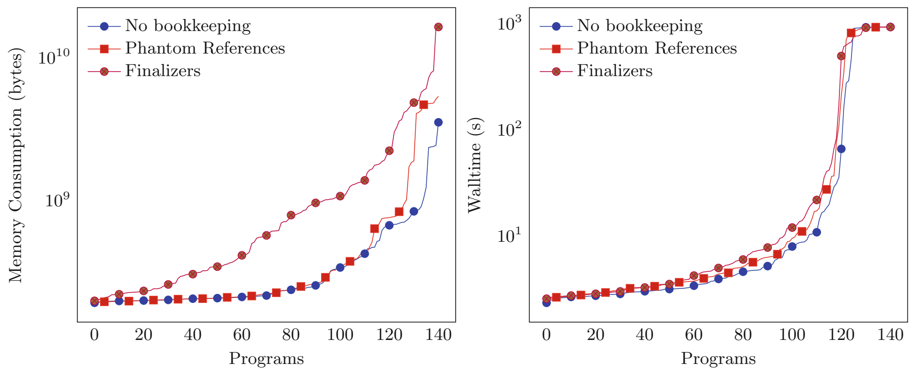

Fig. 2. Resource usage comparison across different memory management strategies for Z3

Implementation.
Initialization: To initialize JavaSMT, we pass the required classes using dependency injection, as shown in Listing 1. This code snippet generates a configuration from passed command-line arguments (configuration can choose a solver, and tweak any of its options), a logger instance, and initializes the solver context.

Formula Transformation: The Houdini algorithm gets a set of lemmas as an input. However, for checking inductiveness as shown in (2) we need *primed* versions of these lemmas, which we obtain by renaming all free variables using a transformation visitor as shown in Listing 2.

Instead of directly removing asserted lemmas from the solver, we use annotation with auxiliary *selector* variables. Each lemma l_i is converted to $l_i \vee s_i$, where s_i is a fresh Boolean variable. After such an annotation, the lemma l_i can be relaxed by asserting an assumption s_i. The code for input-lemma annotation is shown in Listing 3. Finally, the main Houdini loop, which performs lemma filtering until inductiveness, is shown in Listing 4.

```java
public class HoudiniApp {
  private final FormulaManager fmgr;
  private final BooleanFormulaManager bfmgr;
  private final SolverContext context;

  public HoudiniApp(String[] args) throws Exception {
    Configuration config = Configuration.fromCmdLineArguments(args);
    LogManager logger = BasicLogManager.create(config);
    ShutdownNotifier notifier = ShutdownManager.create().getNotifier();

    context = SolverContextFactory.createSolverContext(
        config, logger, notifier);
    fmgr = context.getFormulaManager();
    bfmgr = context.getFormulaManager().getBooleanFormulaManager();
  }
}
```

Listing 1: JavaSMT initialization

```java
private BooleanFormula prime(BooleanFormula input) {
  return fmgr.transformRecursively(input,
      new FormulaTransformationVisitor<Formula>() {
    @Override
    public Formula visitFreeVariable(Formula f, String name) {
      return fmgr.makeVariable(
          fmgr.getFormulaType(f), name + "'");
    }
  });
}
```

Listing 2: Transforming formulas with JavaSMT

```
public List<BooleanFormula> houdini(
    List<BooleanFormula> lemmas, BooleanFormula transition)
    throws SolverException, InterruptedException {
  List<BooleanFormula> annotated = new ArrayList<>();
  List<BooleanFormula> annotatedPrimes = new ArrayList<>();
  Map<Integer, BooleanFormula> indexed = new HashMap<>();

  for (int i = 0; i < lemmas.size(); i++) {
    BooleanFormula lemma = lemmas.get(i);
    BooleanFormula primed = prime(lemma);

    annotated.add(bfmgr.or(getSelectorVar(i), lemma));
    annotatedPrimes.add(bfmgr.or(getSelectorVar(i), primed));
    indexed.put(i, lemma);
  }

  // ... Continuted Later ...
}

private BooleanFormula getSelectorVar(int idx) {
  return bfmgr.makeVariable("SEL_" + idx);
}
```

Listing 3: Annotating formulas with JavaSMT

```
try (ProverEnvironment prover =
    context.newProverEnvironment(ProverOptions.GENERATE_MODELS)) {
  prover.addConstraint(transition);
  prover.addConstraint(bfmgr.and(annotated));
  prover.addConstraint(bfmgr.not(bfmgr.and(annotatedPrimes)));

  while (!prover.isUnsat()) {
    try (Model m = prover.getModel()) {
      for (int i = 0; i < annotatedPrimes.size(); i++) {
        BooleanFormula annotatedPrime = annotatedPrimes.get(i);
        if (!m.evaluate(annotatedPrime)) {
          prover.addConstraint(getSelectorVar(i));
          indexed.remove(i);
        }
      }
    }
  }
}
return new ArrayList<>(indexed.values());
```

Listing 4: Houdini main loop with JavaSMT

6 Related Work

jSMTLIB [7] is a solver-agnostic library for Java which uses SMT-LIB for communication with the solvers, and thus has the associated restrictions outlined in Sect. 1, including costly serialization overhead and a limitation to the features offered by SMT-LIB. In contrast, our work presents a solver-independent library for Java which connects directly to the solvers API.

The newly published jDart [13] tool bundles a jConstraints library that offers a functionality similar to JavaSMT. However, JavaSMT has more features, communicates with solvers using their API, and provides an efficient memory-management strategy (jConstraints uses the official Z3 Java API, which relies on finalizers). Additionally, our library provides several solvers that can be installed automatically and one simple configuration option to switch between them. For jConstraints, the user has to manually include and configure all the solver's bindings and binaries. We have learned that these steps are complicated and error-prone, as the library might be used as part of a bigger software system. Thus, our solvers and their bindings do not require to setup any special environment.

The problem of creating such a library has also been tackled for Python in PySMT [15]. In contrast to our work, PySMT keeps the formula structure itself, while delegating the queries to the solvers. While this allows for creating formulas without any solvers installed, and for easier transfer of formulas between different contexts, it incurs a large memory overhead.

7 Conclusion

We have presented JavaSMT, a new library for efficient and safe communication with SMT solvers. The advantages of using such a library over communicating using SMT-LIB include performance, access to new features, and the ability to control which formulas remain in scope and which should be discarded. Some disadvantages exist as well — using JavaSMT means restricting to the supported solvers, and relying on JavaSMT developers to update the solvers in time. Our experience with using SMT solvers is that for applications that pose a few large, monolithic queries and need only standard features, the communication using SMT-LIB is optimal, while for tools that post many cheap, incremental queries, using the API via JavaSMT is the better solution. New editions of SMT-LIB could make missing features like interpolation available (proposed draft already exists [11]), but giving the user control over memory management for formulas (Sect. 4), or allowing efficient communication without string serialization and parsing may be far outside of the scope of SMT-LIB initiative. So for users requiring such features, an intermediate-layer library is always beneficial.

Acknowledgements. The authors thank P. Wendler for valuable discussions on design decisions and principles behind JavaSMT, and all JavaSMT contributors for their programming efforts.

References

1. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
2. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Technical report. Department of Computer Science, University of Iowa (2015). www.SMT-LIB.org
4. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
5. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 887–904. Springer, Heidelberg (2016)
6. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
7. Cok, D.R.: The jSMTLIB User Guide (2013). http://smtlib.github.io/jSMTLIB/jSMTLIBUserGuide.pdf. Accessed 10 Feb 2016
8. Karpenkov, E.G., Monniaux, D., Wendler, P.: Program analysis with local policy iteration. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 127–146. Springer, Heidelberg (2016)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
10. Bloch, J.: Effective Java (The Java Series), 2nd edn. Prentice Hall, Upper Saddle River (2008)
11. Christ, J., Hoenicke, J.: Interpolation in SMTLIB 2.0 (2012). https://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf. Accessed 10 Feb 2016
12. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
13. Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: jDart: A dynamic symbolic analysis framework. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 442–459. Springer, Heidelberg (2016)
14. 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)
15. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT 2015 (2015)
16. Rümmer, P.: E-matching with free variables. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 359–374. Springer, Heidelberg (2012)
17. Sebastiani, R., Trentin, P.: OptiMathSAT: A tool for optimization modulo theories. In: Kröening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 447–454. Springer, Heidelberg (2015)