JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode*

Lucas Cordeiro[1,2][0000−0002−6235−4272],
Pascal Kesseli[1],
Daniel Kroening[1,2][0000−0002−6681−5283],
Peter Schrammel[1,3][0000−0002−5713−1381], and
Marek Trtik[1]

[1] Diffblue Ltd, Oxford, United Kingdom
[2] University of Oxford, Oxford, United Kingdom
[3] University of Sussex, Brighton, United Kingdom

Abstract. We present a bounded model checking tool for verifying Java bytecode, which is built on top of the CPROVER framework, named *Java Bounded Model Checker* (JBMC). JBMC processes Java bytecode together with a model of the standard Java libraries and checks a set of desired properties. Experimental results show that JBMC can correctly verify a set of Java benchmarks from the literature and that it is competitive with two state-of-the-art Java verifiers.

1 Introduction

The Java Programming Language is a general-purpose, concurrent, strongly typed, object-oriented language [13]. Applications written in Java are compiled to the bytecode instruction set and binary format as defined in the Java Virtual Machine (JVM) specification. This compiled Java bytecode can run on all platforms on top of a JVM without the need for recompilation. However, Java programs may have bugs, which may result in array bound violations, unintended arithmetic overflows, and other kinds of functional and runtime errors. In addition, Java allows multi-threading, and thus, problems such as race conditions and deadlocks can occur.

To detect such issues, we developed an extension to the C Bounded Model Checker (CBMC) [6], named JBMC,[4] that verifies Java bytecode. JBMC consists of a frontend for parsing Java bytecode and a Java operational model (JOM), which is an exact but verification-friendly model of the standard Java libraries. A distinct feature of JBMC, when compared with other approaches [2,7,9], is the use of Bounded Model Checking (BMC) [4] in combination with Boolean Satisfiability and Satisfiability Modulo Theories (SMT) [3] and full symbolic state-space exploration, which allows us to perform a bit-accurate verification of Java programs. Apart from JBMC, there are other Java verifiers, which use different verification approaches.

* Support by ERC project 280053 CPROVER and the H2020 FET OPEN 712689 SC^2.

[4] Available at `https://www.cprover.org/jbmc/`

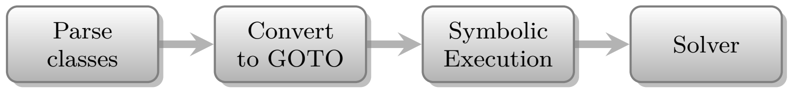


Fig. 1: JBMC verification process

Existing Java Verifiers. *JayHorn* is a verifier for Java bytecode [9] that uses the Java optimization framework Soot [14] as a front-end and then produces a set of constrained Horn clauses to encode the verification condition (VC). *Java Path Finder* (JPF) is an explicit-state and symbolic software model checker for Java bytecode [2]. JPF is used to find and explain defects, collect runtime information as coverage metrics, deduce test vectors, and create corresponding test drivers for Java programs. JPF checks for property violations such as deadlocks or unhandled exceptions along all potential execution paths as well as user-specified assertions. *ESC/Java* is a compile-time extended static checker, which detects common programming errors (e.g., null dereference, array bounds errors, and type cast errors) [7]. It uses an automatic theorem prover to catch bugs that go beyond the abilities of the Java type checker, including runtime errors and synchronization errors in concurrent programs.

2 JBMC: A Bounded Model Checker for Java Bytecode

2.1 Architecture and Implementation

Our front-end integrates a class loader, which accepts Java bytecode *class* files and *jar* archives. The parse trees for the classes are translated into the CPROVER CFG representation, which is called a *GOTO* program [6].

To handle polymorphism, JBMC encodes virtual method dispatch into a *switch* over the runtime type information attached to the object in order to select the correct method to be called. Similarly, the complex control flow arising from exceptions is encoded into conditional branches. We record the exception thrown in a global variable, which is then used to propagate the exception up the call stack until a matching *catch* statement (if any) to handle the error is reached. JBMC can detect when the JVM would abort due to an exception that is not caught within the program.

The resulting *GOTO* program is then passed to the bounded model checking algorithm for finding bugs. The BMC algorithm symbolically executes the program, unwinding loops and unfolding recursive function calls up to a given bound. The resulting bit-vector formula is then passed on to the configured SAT or SMT solver [6].

2.2 Java Operational Model

The Java language relies on compiler-generated functions and classes as well as a large standard library. In order to correctly support Java functionality, we developed an abstract representation of the standard Java libraries, called the operational model (OM). The use of OMs is commonplace in analysers for

Java; for instance, a similar approach was previously proposed for the formal verification of Android applications [12]. Currently, our OM consists of models of the most common classes from *java.lang* and a few from *java.util.* Our Java OM simplifies the implementation of the standard Java library by removing verification-irrelevant performance optimizations (e.g., in the implementation of container classes), exploiting declarative specifications (using *assume*) and functions that are built into the CPROVER framework (e.g., for array and string manipulation). We are continuously extending our OM to speed up verification by replacing the original standard Java library classes by our models.

Java has an `assert(`c`)` statement for specifying safety properties. In addition, we provide API classes that allow users to define non-deterministic verification harnesses and stub functions. The API contains such methods for primitive types (e.g., `int nondetInt()`) and *generic* methods (i.e., parametrised by a type `T`) as `<T> T nondetWithNull()` and `<T> T nondetWithoutNull()` to non-deterministically initialize object references that may or may not be `null`. The API also provides an `assume(`c`)` method, which advises JBMC to ignore paths that do not satisfy a user-specified condition c.

Currently, JBMC handles neither the Java Native Interface, which allows Java code to interface native libraries, nor reflection, which allows the program to inspect and manipulate itself at runtime. We are currently extending JBMC to support generics and lambdas; and to verify multi-threaded Java programs (that use *java.lang.Thread*), exploiting the partial order encoding technique of [1].

2.3 String Solver

One of the biggest challenges in verifying Java programs is the widespread use of character strings, which makes verification problems resulting from Java programs highly complex. Solving such constraints is an active area of research [5,8,11]. JBMC implements a solver for strings to determine the satisfiability of a set of constraints involving string operations. Our string solver supports the most common basic accesses (e.g., obtain the length of a string and a character at a given position); comparisons (e.g., lexicographic comparison and equality); transformations (e.g., insertion, concatenation, replacement, and removal); and conversions (e.g., conversion of the primitive data types into a string and parsing them from a string). The axioms for these operations use quantified constraints. For instance, a Java expression `s.substring(5)` is translated into a predicate $\mathit{substring}(\mathit{res}, s, 5)$, where res, s are pairs $(\mathit{length}, \mathit{charArray})$, representing the resulting and the input string `s`, respectively; and *substring* is axiomatized by the formula $\forall i.(0 \leq i \wedge i < s.\mathit{length} - 5) \rightarrow (\mathit{res}.\mathit{length} = s.\mathit{length} - 5) \wedge (\mathit{res}.\mathit{charArray}[i] = s.\mathit{charArray}[i + 5])$. The universal quantifiers are handled using quantifier elimination [10].

2.4 JBMC Usage

Runtime errors in Java (e.g., illegal memory access) are detected by the JVM and an appropriate exception is thrown (e.g., `NullPointerException`, `ArrayIndex-OutOfBoundsException`). An `AssertionError` is thrown on violation of a condition specified by the programmer using the `assert` keyword. JBMC analyzes the program and verifies whether such error conditions occur.

JBMC can be used to analyze a single class file:[5] `jbmc C.class --unwind` k or a Java archive (jar) file: `jbmc file.jar --main-class class --unwind` k. In both cases the entry point for the analysis of the program is the `static void main` method of the specified main class. k is a positive integer limiting the number of times loops are unwound and recursions are unfolded. If no bug is found, up to a k-depth unwinding, then JBMC reports `VERIFICATION SUCCESSFUL`; otherwise, it reports `VERIFICATION FAILED` along with a counterexample in the form of an execution trace (`--trace`), which contains the full variable assignment in each program state with file, method, and line information. Note that if the Java bytecode is compiled with debug information, then JBMC can also provide the original program variable names in the counterexample, rather than just bytecode variable slots. Further JBMC options can be retrieved via `jbmc --help`.

3 Experimental Evaluation

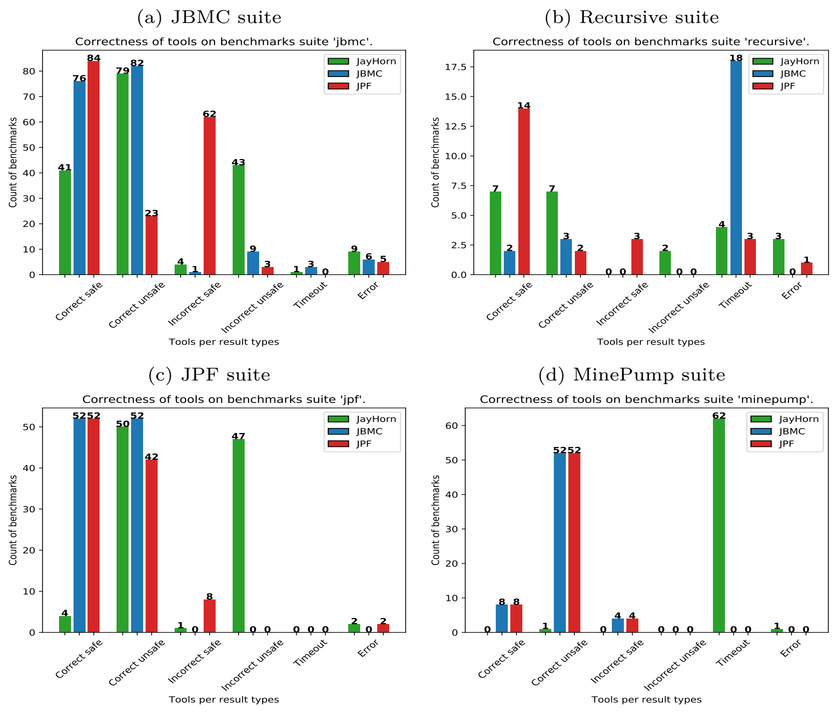
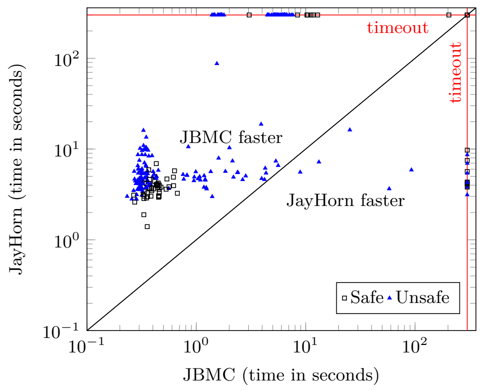


Fig. 2: Verification results for JayHorn, JBMC and JPF

[5] If a class `C` is in a package `x.y`, then compile it to *some-dir*`/x/y/C.class`, and in *some-dir* execute *jbmc-installation-dir*`/jbmc x/y/C.class --unwind` k.



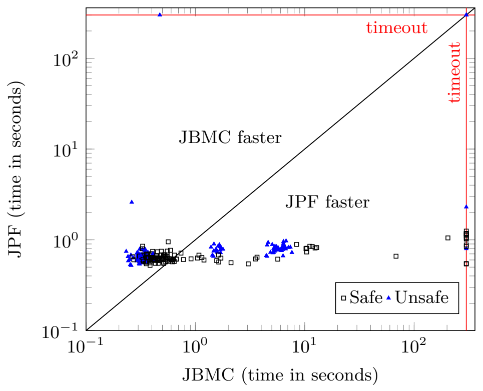


Fig. 3: Runtime comparison of JBMC to JayHorn and JPF

There is no standard benchmark suite for Java verification. Therefore, we took our entire regression test suite consisting of 177 benchmarks (including known bugs and hard benchmarks that JBMC cannot yet handle); these benchmarks (denoted as “jbmc”) test common Java features (e.g., polymorphism, exceptions, arrays, and strings). We also used 23 recursive benchmarks (denoted as “recursive”) taken from the JayHorn repository [9], and 64 minepump benchmarks (denoted as “minepump”) from the SV-COMP repository. Additionally, we have extracted 104 benchmarks from the JPF regression test suite [2]. The following table summarizes the characteristics of the benchmark sets:[6]

benchmark set	total	safe	unsafe	avg. LOC
jbmc	177	89	88	25
jpf	104	52	52	52
recursive	23	14	9	35
minepump	64	8	56	62
total	368	163	205	40

3.1 Objectives and Setup

Our experiments aim at answering two research questions: [RQ1] **(correctness)** How accurate is JBMC when verifying the chosen benchmarks? [RQ2] **(performance)** How does JBMC performance compare to other existing verifiers? To answer both questions, we analyze all benchmarks with three Java verifiers (JBMC v5.8-cav18, JayHorn v0.5.1, and JPF v32) on an Intel Core i7-6700 CPU 8×3.40 GHz, with 32 GB of RAM, running Ubuntu 16.04 LTS. We restrict CPU time and memory to 300 s and 15 GB, respectively. JBMC uses a stepwise approach to unwinding loops (to prove unbounded safety) and runs with MiniSat2 as its SAT backend.

[6] Benchmarks and detailed results are available at `https://www.cprover.org/jbmc`.

3.2 Results

Figure 2 gives an overview of the experimental results for the four benchmark suites. *Correct safe* means that the program was analyzed to be free of errors, *correct unsafe* means that the error in the program was found, *incorrect safe* means that the program had an error but the verifier did not find it, *incorrect unsafe* means that an error is reported for a program that fulfills the specification, *timeout* indicates that the verifier has exceeded the time limit, and *error* represents an internal failure in the verifier or exhaustion of available memory. The following table summarizes the overall results:

	correct			incorrect				
	total	safe	unsafe	total	safe	unsafe	timeout	error
JayHorn	189	52	137	97	**5**	92	67	15
JBMC	**327**	138	**189**	**14**	**5**	9	21	**6**
JPF	277	**158**	119	80	77	**3**	**3**	8

The experimental results show that JBMC reached a successful verification rate of approximately 89% while JayHorn reported 51% and JPF 75%, which positively answers RQ1. JayHorn and JPF currently produce 6 times more *incorrect* results (i.e., bugs in the tool) than JBMC. To answer RQ2, Figure 3 compares the analysis times for the benchmarks where the tools return correct results. None of the three tools is consistently better than the other two. JBMC is faster than JPF on 176 benchmarks, JPF is faster than JBMC on 93. JBMC is faster than JayHorn on 222 benchmarks, whereas JayHorn is faster than JBMC on 25. In comparison to JayHorn, JBMC deals poorly with recursion, as its analysis led to timeout for 69% of the recursive benchmarks, whereas JayHorn could only solve a single benchmark from the minepump benchmark suite. In summary, we observed that JBMC's scalability depends mainly on the complexity of string operations, loops, recursion and (floating-point) arithmetic.

4 Conclusions and Future Work

Despite more than 15 years of research in BMC and Java verification, JBMC is the first BMC-based Java verifier. To achieve this, we based our implementation on an industrial-strength verification framework, and developed a Java OM, removing verification-irrelevant optimizations and exploiting declarative specifications and built-in functions. Because of the prevalent use of character strings in Java programs, we have also developed a string solver using an efficient quantifier elimination scheme. We compare JBMC to JayHorn and JPF, which are state-of-the-art verifiers for Java bytecode based on constrained Horn clauses and path-based symbolic execution, respectively. Experimental results show that JBMC achieves a successful verification rate of 89% compared to 51% of JayHorn and 75% of JPF. For future work, the Java OM will be extended to support more Java classes, with the goal of speeding up verification of larger Java applications. In addition, we are currently extending JBMC to verify multi-threaded programs.

Acknowledgments. We thank P. Rümmer and W. Visser for helpful discussions about JayHorn and JPF, respectively.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV. LNCS, vol. 8044, pp. 141–157 (2013). https://doi.org/10.1007/978-3-642-39799-8_9
2. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: TACAS. LNCS, vol. 4424, pp. 134–138 (2007). https://doi.org/10.1007/978-3-540-71209-1_12
3. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 26, pp. 825–885. IOS Press (Feb 2009)
4. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
5. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the ReplaceAll function. PACMPL **2**(POPL), 3:1–3:29 (2018). https://doi.org/10.1145/3158091
6. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176 (2004). https://doi.org/10.1007/978-3-540-24730-2_15
7. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: PLDI 2002: Extended static checking for Java. SIGPLAN Notices **48**(4S), 22–33 (2013). https://doi.org/10.1145/2502508.2502520
8. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4:1–4:32 (2018). https://doi.org/10.1145/3158092
9. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: CAV. LNCS, vol. 9779 (2016). https://doi.org/10.1007/978-3-319-41528-4_19
10. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: HVC. LNCS, vol. 8244, pp. 15–31 (2013)
11. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Formal Methods in System Design **48**(3), 206–234 (2016). https://doi.org/10.1007/s10703-016-0247-6
12. van der Merwe, H., Tkachuk, O., van der Merwe, B., Visser, W.: Generation of library models for verification of Android applications. ACM SIGSOFT Software Engineering Notes **40**(1), 1–5 (2015). https://doi.org/10.1145/2693208.2693247
13. Oracle: Java^{TM} programming language. https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html (2017), accessed: 31-01-2018
14. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot – a Java bytecode optimization framework. In: CASCON. p. 13. IBM Press (1999)