JBMC: Bounded Model Checking for Java Bytecode
(Competition Contribution)

Lucas Cordeiro[1] [0000−0002−6235−4272],
Daniel Kroening[2,3] [0000−0002−6681−5283], and
Peter Schrammel[2,4] [0000−0002−5713−1381]

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

Abstract. JBMC is a bounded model checking tool for verifying Java bytecode. It is built on top of the CPROVER framework. JBMC processes Java bytecode together with a model of the standard Java libraries. It checks a set of desired properties, such as assertions and absence of uncaught exceptions, under given bounds on loops, recursion and data structures. Internally, it uses the same bounded model checking engine as its sibling tool CBMC and discharges the generated verification conditions with the help of MiniSAT 2.2.1.

1 Overview

JBMC is a bounded model checker based on Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT), which allows the verification of Java programs [3]. JBMC inherits memory model, symbolic execution engine and SAT/SMT backends of its sibling tool CBMC [2]. In particular, 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. Thus, JBMC supports Java bytecode and can verify programs that make use of classes, inheritance, polymorphism, arrays, bit-level operations and floating-point arithmetic using CBMC's verification engine.

JBMC can reason about array bound violations, unintended arithmetic overflows, and other kinds of functional and runtime errors. However, as with other bounded model checkers, JBMC is in general incomplete, i.e., can only be used to find property violations up to a given bound k but not to prove properties, unless we know an upper bound on the depth of the state space by checking whether all loops have been fully unrolled; this is accomplished by inserting a so-called *unwinding assertion* at the end of each loop and recursion to check for termination.

JBMC natively supports MiniSAT as its main solver to discharge verification conditions (VCs) and check for their satisfiability, but can also be used with other

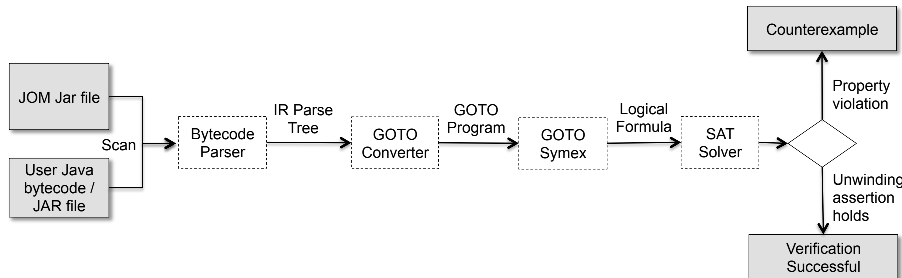

Fig. 1. JBMC Architecture. Grey rectangles represent input and output while white rectangles represent the JBMC main verification steps.

incremental SAT solvers such as Glucose. For SV-COMP 2019, however, JBMC does not use incremental bounded model checking to verify Java programs with (multiple) loops, i.e., it does not check the VCs in iteration $k+1$ by building upon the work done for iteration k [5].

2 Architecture

JBMC's architecture is illustrated in Figure 1. JBMC accepts Java bytecode class files or JAR files as input together with the JOM to parse the Java bytecode and translate it into the CPROVER control-flow graph representation, which is called a GOTO program; this transformation simplifies the Java bytecode representation (e.g., replacement of *switch* and *while* by *if* and *goto* statements) as well as lowering of exceptional control flow.

The GOTO Symex component performs a symbolic execution of the program, which thus handles dynamic memory allocation, encoding of virtual method dispatch, unrolling of the loops and unfolding of recursive method calls. In particular, JBMC uses two functions that compute the constraints C (i.e., assumptions and variable assignments) and properties P (i.e., built-in and user-defined assertions); it automatically generates safety conditions that check for null dereference, array bounds errors, type cast errors and other kinds of functional and runtime errors. Both functions accumulate the control-flow predicates at each program point and use these predicates to guard both the constraints and the properties, so that they properly reflect the Java bytecode's semantics. JBMC's VC generator then derives the VCs from these; the resulting bit-vector formula (i.e., $C \wedge \neg P$) is then passed on to the configured SAT solver to check for satisfiability. If this formula is *satisfiable*, then JBMC produces a counterexample; otherwise, if the formula is *unsatisfiable*, then a successful verification result is reported.

3 Features

JBMC uses an abstract representation of the standard Java libraries, called the Java operational model (JOM), which consists of simplified models of the most common classes from *java.lang* and a few from *java.util*; these models remove verification-irrelevant performance optimizations (e.g., in the implementation of container classes), exploit declarative specifications (using `assume` statement) and functions that are built into the CPROVER framework (e.g., for array and string manipulation).

JBMC also implements a solver for strings to determine the satisfiability of a set of constraints involving strings [4]. Specifically, our string solver implements a decision procedure for string operations that are typically used by Java programs, such as concatenation, search, extract and conversions to other data types. This decision procedure uses incremental SAT solving to lazily instantiate quantifiers.

JBMC also provides API classes that allow users to define non-deterministic verification harnesses and stub functions as used in the SV-COMP benchmarks. The API[5] contains such methods for primitive data-types (e.g. `nondetDouble()`) and *strings* (e.g. `nondetString()`). The API also provides an `assume(`*condition*`)` method, which advises JBMC to ignore paths that do not satisfy a user-specified condition. JBMC is able to check for array bounds, division by zero, unintended arithmetic overflows, runtime errors in Java (e.g. illegal memory access) and user-specified assertions.

Current development efforts include improving support for regular expressions, multi-threaded programs and enabling output of VCs using the SMT-LIB format to be checked by SMT solvers such as Z3, CVC4, Boolector, MathSAT and Yices.

4 Strengths and Weaknesses

JBMC does not produce any incorrect result for any of the Java verification tasks available in SV-COMP 2019 [1]; it correctly claims 139 benchmarks correct and finds existing errors in 192. However, JBMC crashes (and returns `unknown`) in 37 benchmarks due to time or memory exhaustion, or due to missing models of the Java standard library. JBMC can handle most Java basic features (e.g., inheritance, polymorphism and exceptions) and strings manipulations (but *regexes* are not fully supported yet). However, JBMC's concurrency support is still limited and there is no support for Java 8 lambdas, reflection and Java Native Interface (JNI). As its sibling CBMC, JBMC can only prove bounded programs unless an upper bound is known on the depth of the state space, which is not generally the case. Lastly, our JOM does not cover the entire Java standard library.

[5] https://github.com/diffblue/java-models-library/blob/master/src/main/java/org/sosy_lab/sv_benchmarks/Verifier.java

```java
1 import org.sosy_lab.sv_benchmarks.Verifier;
2 public class Main {
3   public static void main(String[] args) {
4     String arg = Verifier.nondetString();
5     float floatValue = Float.parseFloat(arg);
6     String tmp = String.valueOf(floatValue);
7     assert tmp.equals("2.50");
8   }
9 }
```

Fig. 2. Illustrative Java code extracted from SV-COMP 2019 (StringValueOf08).

5 Tool Setup

The competition submission is based on JBMC version 5.10.[6] For the competition, JBMC is called from a wrapper script.[7] The wrapper script compiles the .java source files in the given benchmark directories and then invokes the jbmc binary repeatedly with increasing values for the unwinding bound until the property has been refuted (answering false) or the program has been fully unwound without finding a property violation (answering true). See the wrapper script for the relevant command line options given to JBMC. As an example, we can run the JBMC wrapper script to check for a reachability property in the program shown in Figure 2 by executing the following command:

```
./jbmc --propertyfile <path-to-sv-benchmarks>/properties/assert.prp
    <path-to-sv-benchmarks>/java/jbmc-regression/StringValueOf08
```

where assert.prp indicates the specification to be verified for StringValueOf08. Note that this program invokes in line 4 a non-deterministic method (*Verifier.nondetString();*) to produce an arbitrary string value; this method is provided by SV-COMP in org.sosy_lab.sv_benchmarks.Verifier. The JOM (core-models.jar) is also part of the submission archive. If a verification task uses a Java library method that is not part of the JOM then the wrapper script returns unknown. The Benchexec tool info module is called jbmc.py and the benchmark definition file jbmc.xml. The competition submission of JBMC uses MiniSAT 2.2.1 as SAT backend. JBMC competes in the Java category.

6 Software Project

JBMC is maintained by Peter Schrammel together with numerous contributors[8] from the community. It is publicly available under a BSD-style license. The source code is available at http://www.github.com/diffblue/cbmc in the jbmc directory. Instructions for building JBMC from source are given in the file COMPILING.md.

---

[6] Executable available at https://gitlab.com/sosy-lab/sv-comp/archives/tags/svcomp19
[7] Can be built from https://github.com/diffblue/cprover-sv-comp/tree/svcomp19
[8] https://github.com/diffblue/cbmc/graphs/contributors

References

1. Beyer, D.: Automatic Verification of C and Java Programs: SV-COMP 2019. In: Proc. TACAS, part 3. LNCS 11429, Springer (2019)
2. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
3. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: Computer Aided Verification, CAV. LNCS, vol. 10981, pp. 183–190. Springer (2018)
4. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: HVC. LNCS, vol. 8244, pp. 15–31 (2013)
5. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Incremental bounded model checking for embedded software. Formal Asp. Comput. 29(5), 911–931 (2017)