arXiv:2302.02381v1 [cs.SE] 5 Feb 2023

JBMC
Java 字节码有界模型检测工具*

Romain Brenguier、Lucas Cordeiro、Daniel Kroening 和 Peter Schrammel

摘要。 JBMC 是一款基于可满足性和可满足性模理论的开源有界模型检测工具，用于验证 Java 字节码。JBMC 依赖于 Java库的操作模型，该模型对其语义进行保守近似，以验证 Java 字节码中的断言违规、数组越界、非预期算术溢出以及其他类型的功能错误和运行时错误。如果已知状态空间深度的上界，JBMC 可用于证伪性质或证明程序正确性。JBMC 的实际应用包括但不限于缺陷查找、性质检查、测试输入生成、安全漏洞检测和程序合成。本文详细介绍 JBMC 的架构及其功能，包括深入探讨其背景理论和底层技术，其中包括一种先进的字符串求解器，用于确保 Java 字节码的安全性和可靠性。

Romain Brenguier，Diffblue Ltd，英国，电子邮箱：romain.brenguier@diffblue.comLucas C. Cordeiro，曼彻斯特大学，英国，电子邮箱：lucas.cordeiro@manchester.ac.ukDaniel Kroening，牛津大学，英国，以及 Diffblue Ltd，英国，电子邮箱：kroening@cs. ox.ac.ukPeter Schrammel，萨塞克斯大学，布莱顿，英国，以及 Diffblue Ltd，英国，电子邮箱：p. schrammel@sussex.ac.uk

* 本文的研究部分由英国工程与自然科学研究委员会资助，项目编号为 EP/T026995/1、EP/V000497/1，以及欧盟H2020 ELEGANT 项目（957286）。

1 引言

Java 编程语言是一种通用、并发、强类型、面向对象的语言 [26]。使用 Java 编写的应用程序会被编译为 Java 虚拟机（JVM）规格中定义的字节码指令集和二进制格式。[2] 这些编译后的 Java 字节码无需重新编译，即可在所有平台上的 JVM 上运行。然而，Java 程序可能存在缺陷，从而导致数组越界、非预期算术溢出以及其他类型的功能错误和运行时错误。此外，Java 支持多线程，因此可能出现竞态条件和死锁等问题。

文献中已有多种适用于 Java 程序（或其字节码指令表示）的验证器，它们采用显式状态、符号模型检测或静态检查技术 [3, 18, 20]。本文介绍一种名为 JBMC 的 Java 字节码验证有界模型检测工具。JBMC 是一款先进的验证工具，用于验证编译为 Java 字节码的程序。该工具于 2015 年在 CProver 框架 [11]之上开始开发。JBMC 将有界模型检测（BMC） [8]与布尔可满足性（SAT）和理论模可满足性（SMT） [4] 相结合，并进行完整的符号状态空间探索，使开发人员能够对 Java 程序进行位精确验证。它主要旨在帮助 Java 开发人员发现代码中的细微错误。具体而言，它可以验证用户指定的断言、运行时异常（例如，数组越界异常），以及从入口点逸出并因此导致 JVM 中止的异常。需要注意的是，JBMC 无需在 Java 代码中添加任何专门注释即可查找这类错误。不过，用户可以指定自己的断言，以验证自定义性质。

JBMC 可用于解决各种常见的验证问题，例如缺陷查找、性质检查、测试输入生成、安全漏洞检测和程序合成。这些验证问题可能出现在不同领域，例如嵌入式系统、Web 和企业应用程序。JBMC 是开源软件（采用 BSD 风格许可证），其源代码可在https://github.com/diffblue/cbmc/tree/develop/jbmc获取。

2 验证方法

JBMC 是 C 有界模型检查器（CBMC）的扩展， [12], 名为 JBMC，用于验证 Java 字节码。JBMC 由一个用于解析 Java 字节码的前端和一个 Java 操作模型（JOM）组成。JOM 是标准 Java 库的精确且便于验证的模型。在 JBMC 中，要分析的 Java 字节码被建模为一个状态转换系统，该系统从控制流图（CFG） [24] 中提取。该图是在将 Java 字节码转换为静态单赋值（SSA）形式的过程中构建的。CFG 中的一个节点表示

[2] https://docs.oracle.com/javase/specs/index.html

一个（非）确定性赋值或条件语句，而 CFG 中的一条边表示字节码控制位置可能发生的变化。

给定迁移系统 M、性质 ϕ 和界限 k，JBMC 将该系统展开 k 次，并将其转换为一个验证条件；当且仅当 ϕ 存在长度不超过 k 的反例时，该验证条件才可满足。验证条件是一个属于一阶逻辑中可判定子集的量词自由公式，随后由 SAT 求解器检查其可满足性。[3] 具体而言，JBMC 的后端会生成使用 QF_ABVFP+Strings 逻辑的公式，其中 QF 表示量词自由公式，A 表示数组理论，BV 表示定长位向量理论，FP 表示浮点数理论 [5]。

更形式化地说，JBMC 中相关的模型检测问题可由以下逻辑公式表示：

$$\forall s_0, \ldots, s_k : I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \Longrightarrow \bigwedge_{i=0}^{k} \phi(s_i) \tag{1}$$

其中，ϕ 是安全性质，I 是 M 的初始状态集合，而 $T(s_i, s_{i+1})$ 是 M 在时间步 i 和 $i+1$ 之间的迁移关系。因此，$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ 表示 M 长度为 k 的执行。该公式有效，当且仅当 ϕ_i 对所有不超过 k 的步骤均成立。

我们不检查公式的有效性，而是对其取反并检查其可满足性：

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\phi(s_i) \tag{2}$$

当且仅当*iff* 某些 $i \leq k$ 情况下，存在一个在时间步 i 处违反了 ϕ 的可达状态时，方程 (2) 才可满足。如果方程 (2) 可满足，那么 ϕ 就被违反了，并且 SAT 求解器会提供一个满足赋值；根据该赋值，我们可以提取程序变量的值，从而构造一个反例。对于性质 ϕ，反例是一系列状态 $s_0, s_1, \ldots, s_k$，其中 $s_0 \in S_0$、$s_k \in S$和$T(s_i, s_{i+1})$ 对于 $0 \leq i < k$成立。如果方程 (2) 不可满足，我们就可以得出结论：在 k 步或更少的步数内，不存在可达的错误状态。

与其他方法相比，JBMC 验证方法的一个独特之处 [3, 18, 20], 在于结合使用有界模型检测（BMC） [8] 、布尔可满足性和理论模可满足性（SMT） [4]，并进行完整的符号状态空间探索，从而能够基于 QF_ABVFP+Strings 逻辑，对 Java 程序进行位精确验证。

[3] JBMC 原生支持 MiniSAT 作为其主要求解器，用于处理验证条件并检查其可满足性；同时也可以使用其他增量 SAT 求解器，例如 Glucose。

编程语言	Java 字节码
性质	断言、运行时异常、未捕获异常
运行环境	Linux ，Mac OS ，Windows
所用技术	符号执行、有界模型检测 可满足性和 SMT 求解
当前优势	字符串、浮点数
当前劣势	多线程、Java 库支持

表 1 JBMC 功能

3 软件项目

JBMC 基于 CPROVER 框架。[4] JBMC 使用 C语言++实现，目前约有 20 千行代码（不计 CPROVER 本身）。源代码可在 https://www.github.com/diffblue/cbmc 的 jbmc 目录中获取，并采用 BSD 许可证。JBMC 易于从源代码构建，可在包括 Linux、Mac OS 和 Windows 在内的多个平台上构建。

一个名为 CPROVER Java API 的 Java 库（cprover-api.jar，同样采用 BSD 许可证）提供了 JBMC 与 Java 程序之间的接口。[5] CPROVER Java API 也可在 Maven 中央仓库中获取。[6]

Java 操作模型（JOM）使用 CPROVER Java API 访问 JBMC 的功能，例如非确定性赋值或 假设。[7] JOM（core-models.jar）基于 OpenJDK 8，因此单独以带类路径例外的 GPL2 许可证提供。

表 1 概述了 JBMC 的功能。

4 用例

JBMC 可用于验证 Java 字节码中的数组边界违规、意外算术溢出以及其他类型的功能错误和运行时错误。Java 中的运行时错误（例如非法内存访问）由 JVM 检测，并会抛出相应的异常（例如空指针异常、ArrayIndexOutOfBoundsException）。当程序员使用 assert 关键字指定的条件被违反时，会抛出断言错误。JBMC 会分析程序并验证是否会发生此类错误。

[4] https://www.cprover.org[5] https://github.com/diffblue/java-cprover-api[6] https://search.maven.org/artifact/org.cprover.util/cprover-api[7] https://github.com/diffblue/java-models-library

JBMC 的命令行选项与 Java 可执行程序的类似。JBMC 可用于分析类文件：[8]

```
jbmc MyClass --unwind k
```

或 Java 归档文件（.jar）：

```
jbmc -jar file.jar --main-class MyClass --unwind k
```

在这两种情况下，程序分析的入口点都是指定主类的 public static void main 方法。如果给定的 Java 归档文件的清单文件中没有主类属性，则可以使用 --main-class 选项指定主类。与使用 Java 可执行程序运行程序时的惯例相同，也可以使用 -cp 选项指定类路径。

k 是一个正整数，用于限制循环展开和递归展开的次数。如果在深度不超过 *k* 的展开范围内未发现错误，JBMC 将报告 VERIFICATION SUCCESSFUL；否则，它会报告 VERIFICATION FAILED，并附带一个以执行轨迹（--trace）形式呈现的反例，其中包含每个程序状态下完整的变量赋值，以及文件、方法和行号信息。

请注意，如果 Java 字节码是通过带有调试信息的方式编译的（javac -g 选项），那么 JBMC 还可以在反例中提供原始程序变量名，而不只是字节码变量槽。可以通过 jbmc --help 查看 JBMC 的其他选项。

以下各节提到的示例可在此处找到：https://github.com/diffblue/cbmc/tree/develop/jbmc/regression/book-examples。

4.1 功能属性

可以借助断言语句在 Java 程序中指定功能属性。以图 1 所示的 StringUtil.java 为例。getLastToken 方法在第 17 行包含这样一条 assert 语句。预期在使用分隔符 delimiter 将给定的字符串 toSplit 拆分为最多 limit 个词元时，如果尚未达到上限，lastToken 中就不包含分隔符。我们尝试通过执行以下命令来验证此性质：[9]

```
jbmc StringUtil.getLastToken --classpath
cprover-api.jar:core-models.jar:. --
max-nondet-string-length 100 --unwind 2
```

[8] 如果类 MyClass 位于 my.package 包中，则将其编译到 *some-dir*/my/package/MyClass.class，然后在 *some-dir* 中执行 *jbmc-installation-dir*/jbmc my.package.MyClass --unwind *k*.[9] 在 Linux 和 Mac OS 系统上。在 Windows 上，类路径元素需要用 ; 分隔，而不是用 :。

```
1 public class StringUtil {2  public static String getLastToken(3 String toSplit, char 分隔符,
int limit) {4  if(toSplit  ==  null) {5  return null;6  }7  int numberOfTokens  =  0;
8  int tokenStart  =  0;9  int tokenEnd  =  toSplit.indexOf(分隔符, tokenStart);10  while
(tokenEnd !=  -1 && numberOfTokens  <  limit - 1) {11  ++numberOfTokens;12
tokenStart  =  tokenEnd  +  1;13  tokenEnd  =  toSplit.indexOf(分隔符, tokenStart);14  }
15  ++numberOfTokens;16  String lastToken  =  toSplit.substring(tokenStart,
toSplit.length());17  断言(lastToken.indexOf(分隔符)  <  0  ||  numberOfTokens  ==  li
mit);18  return lastToken;19  }20  }
```

图 1 函数性质示例

```
1  ...2  结果：3  ...4  [StringUtil.getLastToken.断言.1]5  Stri
ngUtil.java 文件第 17 行的断言6  函数 StringUtil.getLastToken 字节码
索引 52：7  失败8  ...9  验证失败
```

图 2 图 1 中函数性质示例的结果。

其中， -classpath 会将 CPROVER API（cprover-api.jar）和 JOM（ core-models.jar）添加到类路径；--max-nondet-string-length 限制非确定性字符串的最大长度。请注意，此处指定的入口点不是主方法，而是 getLastToken 方法。

对于这个特定示例，JBMC 报告的结果如图 2 所示。这意味着该断言不成立。

我们可以添加 --trace 选项来了解原因。随后，JBMC 会显示执行轨迹，如图 3 所示。这表明，当 limit 等于 0 时，我们的实现不符合规格。这很可能是因为我们甚至没有考虑过非正数 limit 应该如何处理。因此，我们必须明确规定这种情况，并据此实现。

```
1 ...2 StringUtil.getLastToken.断言的执行轨迹：3 ...4 String
Util.java 第 4 行，函数 __CPROVER__start5 分隔符='\ua775'67
StringUtil.java 第 4 行，函数 __CPROVER__start8 limit=09 ...10
StringUtil.java 第 14 行，函数 StringUtil.getLastToken11
numberOfTokens=112 ...
```

图 3 图 1 中函数性质示例的执行轨迹。

```
1 public class BinarySearch {2 public static int binarySearch(int[] 数组, i
nt value) {3 int lowerBound = 0;4 int upperBound =数组.length;5 int i = 数组.
length / 2;6 while(数组[i] != value && upperBound > lowerBound + 1) {7
if(数组[i] > value) upperBound =i; else lowerBound =i;8 i =(upperBound +
lowerBound) / 2;9 }10 return i;11 }12 }
```

图 4 检测未捕获的运行时异常示例

```
1 ...2 结果：3 ...4 [BinarySearch.binarySearch:([II)I.1] 无未捕获异常：失败5 ...6 验
证失败
```

图 5 图 4 中运行时异常示例的结果。

4.2 运行时异常

Java 中的运行时异常派生自*RuntimeException* 类，在 JVM 的正常运行过程中可能会被抛出 [26]。运行时异常不会在方法的 throws 子句中声明。图 4 展示了 BinarySearch.java，它实现了二分查找算法，用于查找数组中某个整数的索引。我们可以运行 JBMC，自动检查此程序中是否存在未捕获的运行时异常，方法是执行以下命令：

1 ...2 BinarySearch.binarySearch 的执行轨迹：([II)I.1:3 ...4 文件 BinarySearch.java，第 3 行，函数 __CPROVER__start5 数组=null6 ...7 文件 BinarySearch.java，第 4 行，函数 BinarySearch.binarySearch8 dynamic_object4.@class_identifier="java::java.lang.NullPointerException"9 ...

图 6 图 4 中运行时异常示例的执行轨迹。

```
jbmc BinarySearch.binarySearch --
throw-runtime-exceptions --unwind 2
```

其中，--throw-runtime-exceptions 会将隐式运行时异常显式化，并在这些异常逃出入口方法时检测它们。如果不使用 --throw-runtime-exceptions，即使异常被外围的 try-catch 捕获，JBMC 也会在抛出异常时报告属性验证失败。

运行上述命令后，JBMC 会报告图 5 所示的结果。这表明“没有未捕获异常”这一属性验证失败，也就是说，存在未捕获异常。使用 --trace 选项运行，即可查明逃逸的异常属于哪种类型。

JBMC 在图 6 中显示了反例轨迹。如果数组参数为 null，第 4 行会抛出空指针异常。如果我们尝试修复这个问题并再次检查，就会发现该方法还可能抛出另一种运行时异常：当使用空数组调用该方法时，第 6 行会抛出数组索引越界异常。

4.3 测试输入

对于图 7 中的类，我们希望编写测试，以覆盖源代码中的特定分支。为此，我们需要找到能够到达第 8 行和第 14 行等位置的输入。可以通过在这些指令前添加 assert(false); 指令，并使用 --trace 选项运行 JBMC 来实现：

```
jbmc LocatorHandler.autoLocator --classpath
cprover-api.jar:core-models.jar:. --max-nondet-string-length 10 --
unwind 10 --trace
```

从图 8 所示的输出中可以看到，两个程序位置都已到达，也就是说，相应的断言均已失败。

接下来，我们可以检查执行轨迹，了解输入是如何初始化的。例如，对于第 7 行的断言，我们可以看到图 9 所示的值。这意味着调用

```
1 公共类 LocatorHandler {2 公共枚举 LocatorType {XPATH, ID; }3 公共静态 Locator
autoLocator(Locator locator , LocatorType 类型) {4 切换 (类型) {5 情况 XPATH:6 如果
(locator.startsWith("xpath=")) {7 //assert(false);8 locator =
locator.substring(locator.indexOf("=") + 1);9 }10 返回 new
XPathLocator(locator);11 情况 ID:12 如果 (locator.startsWith("id=")) {13 //
assert(false);14 locator = locator.substring(locator.indexOf("=") + 1);15 }16 返
回 new IdLocator(locator);17 }18 返回 null;19 }20 }
```

图 7 创建测试输入的示例

```
1 结果：2 ...3 [LocatorHandler.autoLocator.断言.1]4
LocatorHandler.java 文件第 7 行的断言5 函数
LocatorHandler.autoLocator 字节码索引 14：6 失败78 [
LocatorHandler.autoLocator.断言.2]9 LocatorHandler.java 文件第 13 行
的断言10 函数 LocatorHandler.autoLocator 字节码索引 37：11 失败12 ...
```

图 8 图 7 中测试输入示例的结果。

带有定位器="xpath=a"且类型为=XPATH的方法将转换第 8 行的语句。同样，我们可以找到到达第 14 行所需的输入。

4.4 安全漏洞

JBMC 可用于查找潜在的安全缺陷，例如，识别未经信任的字符串在未经过净化的情况下被传递给敏感函数处理。在图 10 的示例中，如果 xml 字符串来自不受信任的来源，它就可能

```
1 LocatorHandler.autoLocator.断言的执行轨迹：2 ...3 LocatorHandler.java 第 4 行，函数
__CPROVER__start4 dynamic_object2={ 'x', 'p', 'a', 't', 'h', '=', 'a' }5 ...6 LocatorHandler.java 第 4
行，函数 __CPROVER__start7 dynamic_object1.data=dynamic_object28 ...9
LocatorHandler.java 第 4 行，函数 __CPROVER__start10 输入定位器：&dynamic_object111 ...12
java/lang/Enum.java 第 118 行，函数 java.lang.Enum(java.lang.String, int)13
dynamic_object3.@java.lang.Enum.name=&XPATH14 ...15 LocatorHandler.java 第 4 行，线程 0,
函数 __CPROVER__start16 输入类型：&dynamic_object317 ...
```

图9 图 7 中测试输入示例的执行轨迹。

```
1 import java.io.StringReader;2 import javax.xml.bind.JAXBContext;3 import
javax.xml.bind.JAXBException;4 import javax.xml.bind.Unmarshaller;5 import
javax.xml.stream.XMLInputFactory;6 import javax.xml.stream.XMLStreamException;7 i
mport javax.xml.stream.XMLStreamReader;8 import org.w3c.dom.Comment;910 public class
SimpleXXE {11 protected Comment parseXml(String xml)12 throws JAXBException ,
XMLStreamException {13 JAXBContext jc = JAXBContext.newInstance(Comment.class);14
XMLInputFactory xif = XMLInputFactory.newFactory();15
xif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES , true);16
xif.setProperty(XMLInputFactory.IS_VALIDATING , false);17
xif.setProperty(XMLInputFactory.SUPPORT_DTD, true);18 XMLStreamReader xsr =
xif.createXMLStreamReader(new StringReader(xml));19 Unmarshaller unmarshaller =
jc.createUnmarshaller();20 return (Comment) unmarshaller.unmarshal(xsr);21 }22 }
```

图 10 安全漏洞示例 Java 代码

这会对执行此代码的系统造成危害，因为攻击者可以向程序中注入任意 Java 代码。

要使用 JBMC 检测这些情况，我们可以通过声明一个 InstrumentedString 类来插桩代码，并在其中定义一个布尔标志 is_tainted，该标志设置为

1 public class SignalUtil { 2 public static long abs1(int signal){ 3 long result; 4 if(signal >= 0) 5 result = signal; 6 else 7 result = -1*signal; 8 return result; 9 } 10 public static long abs2(int signal){ 11 if(signal < 0) 12 return -signal; 13 else 14 return signal; 15 } 16 } (a) 1 public class EquivalenceCheck { 2 public void check(int signal){ 3 断言(SignalUtil.abs1(signal) == SignalUtil.abs2(signal)); 4 } 5 } (b)

图 11 (a) 原始函数 abs1 和优化后的版本 abs2。(b) 用于等价性检查的测试框架。

对于直接来自程序的字符串，标志为 false；而对于来自用户的字符串，标志为 true。随后，在敏感函数中添加 assert(!arg.is_tainted); 断言。也可以使用净化函数将该标志设为 false。要自动执行此类插桩，可以使用污点分析。CPROVER 框架中有一个名为 JANALYZER 的工具，它对污点分析提供了有限的实验性支持。

4.5 等价性检查

对于大型软件系统，从头开始重新验证整个软件的工作量可能过大；如果（如常见情况那样）改动很小，这些工作大多也会白费 [25]。持续软件验证通过利用以往验证运行的信息，确定给定软件系统中哪些部分需要重新验证 [10]，在这方面发挥着重要作用。JBMC 可用于证明给定方法的两个版本在计算上等价。

例如，请考虑图 11(a) 中展示的两个用于计算信号绝对值的函数版本。它们提取自该软件

```
1 结果：2 ...3 [EquivalenceCheck.check:(I)V.断言.1]4 位于文件
EquivalenceCheck.java 第 3 行的断言5 函数 EquivalenceCheck.check:(I)
V 字节码索引 11：6 成功7 ...
```

图 12 对图 11 中示例进行等价性检查的结果。

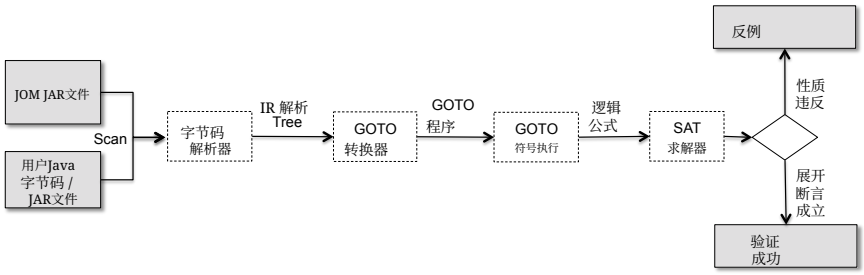

图 13 JBMC 架构 [14]。灰色矩形表示输入和输出，白色矩形表示 JBMC 的主要验证步骤。

医疗器械产品 [13] 的两个版本。为了证明这两个方法等价，我们比较它们的输入输出关系。

因此，我们按照图 11(b) 所示编写了一个等价性检查测试框架。该测试框架会检查：如果两个版本接收到相同的输入，并产生相同的输出，那么它们的行为是否等价。输入是指被比较程序读取的任何数据，输出则是程序修改的任何数据，包括返回值和它们抛出的异常。如果程序会对静态字段产生副作用或修改输入参数，编写测试框架时必须格外谨慎。具体来说，在这个简单示例中，如果输入相同，返回值也必须相同。接下来，我们可以通过运行

```
jbmc EquivalenceCheck.check
```

并获得图 12 所示的结果，从而证明这两个版本输入输出等价。

5 架构

JBMC 的架构（见图 13）是一条由多个处理阶段组成的流水线，将输入的 J
ava 字节码从一种格式转换为另一种格式，直至获得验证结果。

5.1 前端

我们的前端集成了一个类加载器，可接受 Java 字节码类文件和*jar* 归档文件。类的解析树会被转换为 CPROVER 控制流图表示，这种表示称为 *GOTO*程序 [12]。

5.2 GOTO

GOTO 程序组件会将 Java 字节码转换为 goto 程序，从而简化其表示（例如，将*switch* 和*while* 替换为*if* 和*goto* 语句）。GOTO 符号执行组件会对 Java 字节码进行符号模拟，从而处理循环展开和递归函数消除；并生成要在 SAT/SMT 后端中编码的验证条件。

5.3 降级

5.3.1 多态

多态允许变量实例绑定到由继承关系关联的不同类型的引用 [1]。JBMC 通过将虚方法分派编码为一个*switch* 语句来处理多态，该语句根据附加到对象上的运行时类型信息选择要调用的正确方法。具体而言，当 Java 类声明非静态方法时，这些方法默认是“虚方法”，JBMC 会创建一个虚函数表，其中包含指向该类中每个虚方法的指针。如果派生类没有重写某个虚方法，则这些指针会复制到派生类的虚函数表中。相反，如果派生类重写了某个虚方法，则派生类虚函数表中的指针会指向被重写方法的实现。每当调用虚方法时，JBMC 都会使用 SMT-lib 中的 ITE 运算符，通过 switch 语句执行虚函数表中指针所指向的方法 [5]。

5.3.2 异常处理

异常是在程序执行过程中出现的意外情况，例如运行时错误 [17]。在 Java 中，异常处理分为三个（基本）要素：*try* 块，可将抛出的异常导向*catch*语句；一组*catch* 语句，用于处理抛出的异常；以及用于引发异常的*throw* 语句。与多态类似，JBMC 将异常引发的复杂控制流编码为条件分支。

我们将抛出的异常记录在一个全局变量中，然后利用该变量沿调用栈向上传播异常，直到遇到能够处理该错误的匹配*catch* 语句（如果存在）。为了在 JBMC 中支持异常处理，我们扩展了 GOTO 转换代码和符号引擎。在前者中，我们定义了新的指令，并将 throw 表达式建模为跳转；在后者中，我们实现了抛出和捕获异常的规则。JBMC 可以检测 JVM 是否会因程序中存在未捕获的异常而中止。

5.4 符号执行

随后，将生成的 *GOTO* 程序交由有界模型检测算法来查找错误。有界模型检测算法会对程序进行符号执行，展开循环，并将递归函数调用展开到给定界限。然后，将生成的位向量公式传递给配置好的可满足性或可满足性模理论求解器 [12]。

5.5 Java 操作模型

Java 语言依赖编译器生成的函数和类，以及庞大的标准库。为了正确支持 Java 功能，我们开发了标准 Java 库的一种抽象表示，称为操作模型（ OM）。在 Java 分析器中，使用 OM 十分常见；例如，此前有人提出过一种类似的方法，用于对 Android 应用进行形式化验证 [23]。目前，我们的 OM 包含 java.lang 中最常见的一些类，以及 java.util 中的少数几个类。[10] 我们的 Java OM 通过移除与验证无关的性能优化（例如容器类的实现中的优化），利用声明式规格（使用 假设）以及 CPROVER 框架内置的函数（例如用于数组和字符串操作的函数），简化了标准 Java 库的实现。这些内置功能通过 CPROVER API 库提供。我们持续扩展 OM，以便通过用我们的模型替换原始标准 Java 库类来加快验证。

Java 提供了 assert(c) 语句，用于指定安全属性。此外，CPROVER API 在 org.cprover.CProver 类中提供了相关方法，使用户能够定义非确定性验证框架和桩函数。该 API 为基本类型提供了此类方法（例如 int nondetInt()），并提供了用于泛型方法（即由类型 T 参数化）的方法，例如 <T> T nondetWithNull() 和 <T>T nondetWithoutNull()，以非确定性地初始化可能为 null 或非 null 的对象引用。该 API 还提供了 assume(c) 方法，用于告知 JBMC 忽略不满足用户指定条件 c 的路径。

[10] 请参阅 https://github.com/diffblue/java-models-library，了解可用的类、方法及其限制。

如果 JBMC 遇到未建模的库调用，它将输出警告，并将返回值视为非确定性的。[11]

CPROVER API 和 Java 库模型以 JAR 文件的形式提供，需要将其添加到被分析项目的类路径中，参见第 3 节。

目前，JBMC 尚不支持 Java 本地接口（它允许 Java代码与本机库交互），也不支持反射（它允许程序在运行时检查并操纵自身）。JBMC 部分支持泛型和 Lambda 表达式，也支持多线程 Java 程序（使用 java.lang.Thread），并采用了 [2] 的偏序编码技术。

5.6 字符串约束求解器

Java 程序验证面临的最大挑战之一是字符串的广泛使用，这使得由 Java 程序引发的验证问题变得非常复杂。求解此类约束是一个活跃的研究领域 [9, 19, 22]。

JBMC 实现了一个字符串求解器，用于确定一组涉及字符串操作的约束是否可满足。我们的字符串求解器支持最常见的基本访问操作（例如，获取字符串长度和指定位置的字符）；比较操作（例如，字典序比较和相等性比较）；变换操作（例如，插入、连接、替换和删除）；以及转换操作（例如，将基本数据类型转换为字符串，以及从字符串中解析基本数据类型）。

这些操作的公理涉及量化约束。例如，Java 表达式 s.substring(5) 会被转换为谓词 $substring(res, s, 5)$，其中 res 和 s 是一对（$length$, $charArray$），分别表示结果字符串和输入字符串 s；而 $substring$ 由公式 $\forall i。(0 \leq i \wedge i < s。length - 5) \rightarrow (res。length = s。length - 5) \wedge (res。charArray[i] = s。charArray[i+5])$ 公理化。全称量词通过量词消去方案 [21]处理。字符串求解器采用细化循环，依靠一组公理检查量化约束集的可满足性，并使用实例化函数生成由该约束集蕴含的命题公式，随后由可满足性求解器求解。

字符串求解器由 --max-nondet-string-length 命令行选项指定的最大字符串长度进行参数化。它会判定一个涉及字符串的公式对于长度不超过该指定值的任意字符串是可满足还是不可满足。

5.6.1 量化约束的形式

每个字符串都关联一个字符数组和一个长度，内置字符串函数会被转换为数组上的存在公式和全称公式。这些

[11] 在 SV-COMP 中，遇到此类情况时，分析必须返回 UNKNOWN。

公式具有特定形式：$\forall i \in \mathbb{N}$。index_guard(i) $\rightarrow$ value_constraint(i)，其中：index_guard 必须定义一个整数区间，可以包含符号，但不能引用任何字符串/数组；value_constraint 可以访问索引为 i 线性函数的字符串（这样才能还原）。例如，插入(r，s，t，偏移量) 可编码为以下公式：

$$r.length = s.length + t.length \forall i, 0 \leq i < offset \rightarrow r.charArray[i] = s.charArray[i] \forall i, offset \leq i < offset + t.length \rightarrow r.charArray[i] = t.charArray[i - offset] \forall i, offset + t.length \leq i < r.length \rightarrow r.charArray[i] = s.charArray[i - t.length]$$

5.6.2 实例化

方程组中的每个字符串都关联着一个索引集，该索引集会在细化过程的每一步中按需逐步扩展。随后，使用这些索引实例化全称公式，从而得到与约束的一个更宽松版本相对应的存在公式。例如，如果 r 的索引集包含 r.长度 -1，则公式

$$\forall i, offset \leq i < offset + t.length \rightarrow r.charArray[i] = t.charArray[i - offset]$$

将被实例化为

$$offset < r.length - 1 < offset + t.length \rightarrow r.charArray[r.length - 1] = t.charArray[r.length - offset].$$

存在公式会被添加到传递给底层 SAT 求解器的方程组中。此时，如果底层求解器回答不可满足，我们便可以断定原始方程组不可满足，因为我们提供给它的方程组约束不足。另一种情况下，我们尚不能得出结论，但可以检查求解器给出的模型是否确实正确。

5.6.3 检查模型

假设求解器对这个约束不足的方程集合给出可满足 的回答，并且能够为我们提供字符串的模型，我们就可以检查该模型是否满足全称约束。具体做法是对约束取否定，这样就得到一个存在量化约束，然后将符号替换为模型中找到的值。由此得到的公式中，原来的全称量化变量 i 是唯一未知量，因此可以调用 SAT 求解器来检查其有效性。如果该公式有效，那么找到的模型确实是原方程组的模型。否则，就需要细化索引集。

请注意，字符串的模型可能不完整，因为实例化公式只在某些特定索引处引用这些字符串。为了使其完整，我们将每个索引处的值解释为模型中已知的下一个索引处的值。实际上，如果求解器只给出 s.字符数组[3] = 'a' 和 s.字符数组[5] = 'b'，我们就会在公式中将 s.字符数组[i] 替换为 $i \leq 3$?'a' : 'b'。

5.6.4 细化索引集

如果无法通过上述两个步骤得出结论，我们就通过将字符串中的索引传播到出现在同一方程中的其他字符串，来扩充索引集。例如，对于公式 $\forall i$，偏移量 $\leq i <$ 偏移量 $+ t$。长度 $\rightarrow r$。字符数组[i] = t。字符数组[i –偏移量], , 对于属于索引集的索引 j，我们将相应的索引 j –偏移量 加入 t 的索引集。反过来，对于属于 t 的索引集的索引 k，我们将 k +偏移量 加入 r 的索引集。然后从实例化步骤重新开始。要判断可满足性，可能需要连续进行多次细化；但当索引集达到不动点时，上述两个步骤之一就会成功。

6 评估

JBMC 参加了 [16] SV-COMP'19 [6] 和 SV-COMP'20 [7], 的 Java赛道 [15] ，分别获得金牌和银牌。在这几届 SV-COMP 中，我们观察到了一些局限性。

JBMC 只能对有界程序执行有界模型检测或完整模型检查。对于需要大量展开才能验证的程序，它会遇到问题，因为我们的 BMC引擎会生成一个庞大的公式，供底层 SAT/SMT求解器检查。

我们的 Java 操作模型还相当不完整。许多实际应用中会用到的类目前尚未得到支持。具体来说，这些类的方法缺少前置条件和后置条件，以及模拟特性（例如元素的操作方式）。因此，JBMC 对程序原始语义进行了过度近似。

JBMC 对多线程的支持效率不高。尽管 JBMC 使用偏序而非交错执行来建模程序执行，但自动分析多线程 Java代码仍然非常困难。

目前，我们的字符串求解器尚不支持正则表达式，也不支持反射或本地方法。

7 结论与未来方向

JBMC 是一款针对 Java 字节码的有界模型检测器。它基于 CPROVER 框架构建，并与 CBMC 共享大部分后端功能。尽管 JBMC 是一款相对较新的工具，但如果考虑 SV-COMP 最近几届比赛的表现 [6, 7]，它目前是最优秀的 Java 模型检测器之一。不过，它在支持的 Java 语言特性方面仍有许多局限。

参考文献

1. Alexander, R.T., Offutt, J., Bieman, J.M.：基于耦合的面向对象测试的故障检测能力。载于：软件可靠性工程，第 207–2002 页（2002） 2. Alglave, J., Kroening, D., Tautschnig, M.：用于高效有界模型检测并发软件的偏序。载于：CAV，*LNCS*，第 8044 卷，第 141–157 页（2013） 3. Anand, S., Pasareanu, C.S., Visser, W.：JPF-SE：Java PathFinder 的符号执行扩展。载于：TACAS，*LNCS*，第 4424 卷，第 134–138 页（2007） 4. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.：模理论可满足性，人工智能与应用前沿，第 185 卷，第 26 章，第 825–885 页。IOS Press（2009） 5. Barrett, C., Stump, A., Tinelli, C., Boehme, S., Cok, D., Deharbe, D., Dutertre, B., Fontaine, P., Ganesh, V., Griggio, A., Grundy, J., Jackson, P., Oliveras, A., Krstić, S., Moskal, M., De Moura, L., Sebastiani, R., Cok, T.D., Hoenicke, J.：SMT-LIB标准：2.0版。技术报告（2010） 6. Beyer, D.：C语言和 Java 程序的自动验证：SV-COMP 2019。载于：系统构造与分析工具和算法，计算机科学讲义，第 11429 卷，第 9 7 133 155 2019 10 1007 978 3 030 17502 3 页。–（）。DOI . / - - - - \。Beyer, D.：自动软件验证进展：SV-COMP 2020。载于：TACAS，*LNCS*，第 12079 卷，第 21 8 347 367 2020 10 1007 978 3 030 45237 7 页。–（）。DOI . / - - - - \。Biere, A., Heule, M., van Maaren, H., Walsh, T.（编）：可满足性手册，人工智能与应用前沿，第 185 卷。IOS Press（2009） 9. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.：带有 ReplaceAll函数的字符串约束有哪些可判定问题。PACMPL **2**（POPL），3:1–3:29（2018） 10. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.：Amazon s2n 的连续形式化验证。载于：CAV，*LNCS*，第 10982 卷，第 430–446 页。Springer（2018） 11. Clarke, E., Kroening, D., Lerda, F.：用于检查 ANSI-C程序的工具。载于：系统构造与分析工具和算法，计算机科学讲义，第 2988 卷，第 168–176 页。Springer（2004） 12. Clarke, E.M., Kroening, D., Lerda, F.：用于检查 ANSI-C程序的工具。载于：TACAS，*LNCS*，第 2988 卷，第 168–176 页（2004） 13. Cordeiro, L.C., Fischer, B., Chen, H., Marques-Silva, J.：考虑严格硬件约束的医疗设备嵌入式软件半形式化验证。载于：ICESS，第 396–403 页。IEEE计算机学会（2009） 14. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.：JBMC：用于验证 Java 字节码的有界模型检测工具。载于：计算机辅助验证，CAV，*LNCS*，第 10981 卷，第 183–190 页。Springer（2018） 15. Cordeiro, L.C., Kroening, D., Schrammel, P.：软件验证竞赛（SV-COMP）中的 Java验证工具基准测试。CoRR **abs/1809.03739**（2018）。URLhttp://arxiv.org/abs/1809.03739

16. Cordeiro, L.C., Kroening, D., Schrammel, P.：JBMC：Java 字节码的有界模型检测——（竞赛贡献）。载于：系统构造与分析工具和算法，计算机科学讲义，第 11429 卷，第 219–223 页（2019）。DOI 103 17 17 1007 978 3 030 17502 3 171010079783030175 02. / - - - - \。URL https://doi.org/./----。D̄eitel, H.M., Deitel, P.J.：Java 编程入门：后期̄对象版，全球版。Prentice Hall 出版社（2019） 18. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.：PLDI 2002：Java 扩展静态检查。SIGPLAN 通讯 **48**(4S)，22–33（2013） 19. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.：高效求解带有连接和转换器的字符串约束。PACMPL **2**（POPL），4:1–4:32（2018） 20. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.：JayHorn：用于验证 Java 程序的框架。载于：CAV，*LNCS*，第 9779 卷（2016） 21. Li, G., Ghosh, I.：PASS：使用参数化数组和区间自动机进行字符串求解。载于：HVC，*LNCS*，第 8244 卷，第 15–31 页（2013） 22. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.：用于字符串约束的高效 SMT 求解器。系统设计中的形式化方法 **48**(3)，206–234（2016） 23. van der Merwe, H., Tkachuk, O., van der Merwe, B., Visser, W.：为验证 Android 应用程序生成库模型。ACM SIGSOFT 软件工程通讯 **40(1)**，1–5（2015） 24. Muchnick, S.S.：高级编译器设计与实现。Morgan Kaufmann 出版社公司（1997） 25. O'Hearn, P.W.：连续推理：扩大形式化方法的影响。载于：LICS，第 13–25 页。ACM（2018） 26. Oracle：Java^{TM}编程语言。https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html（2017）。访问日期：2018年1月31日