A Bounded Verification Tool for Java source code

Stefan Koppier,
6002978

April 23, 2019

Abstract

We present a bounded model checking tool for a subset of the Java language. The tool allows the verification of user-defined assertions, and that runtime exceptions never occur. We have built our tool on top of JBMC, a bounded model checking tool for Java bytecode. In this document we give a short description of JBMC, then describe the architecture of the tool and finally conclude with results and possible future work. The first results show that this tool successfully verifies the absence and presence of bugs in multiple programs, of which merge sort is one example.

Introduction

We present a tool that allows bounded model checking of Java source code. The tool works as a layer over JBMC, a bounded verification tool for Java bytecode, developed by Lucas Cordeiro et al. [2]. JBMC is developed using the CPROVER framework, which also drives the industrial strength bounded model checking tool CBMC [1].

The development of this tool was inspired by CRUST. [9]. CRUST is a bounded model checking tool to find memory problems in Rust source code. The goal of this project was to provide similar infrastructure as CRUST. That is: develop a basis to do further bounded model checking research of Java source code.

The advantage of using this tool, over using JBMC itself is that we provide an interface for Java source code, instead of Java bytecode. In terms, this allows for higher-level reasoning about the program path generation and verification process.

The tool is hosted on GitHub[1].

Structure of this document

The document contains four chapters and one appendix. In chapter 1, we will give an introduction of JBMC and give an overview of similar tools. Chapter 2 will explain the architecture of the tool, and explain each individual phase and what it tries to achieve. Chapter 3 will give insight on how to use the tool. We will conclude in chapter 4, where we also discuss possible future work. Appendix A show some examples and their results.

A change of course*

Out initial approach was to compile the Java programs to program paths in C. This turned out to be too much work, so we changed our approach to generate program paths in Java.

A first approach: compiling to C

We first envisioned to compile all Java program paths to program paths in C. We could then perform bounded model checking using CBMC [1].

The advantages we foresaw compiling to C were that CBMC is a more mature tool than JBMC. But after a while, the work required to translate the Java semantics to equivalent C semantics turned out to be overwhelming for the time we had available. More specifically, translating the exception handling system of Java required us to define a semantically equivalent exception handling system in C, as the language has no native support for exceptions. Besides the exception handling system, compiling to C had more disadvantages. It required array initialization to contain loop structures in the compiled code, something we wished to avoid. For example, see the Java input program and compiled C program in listings 1 and 2. The array is initialized using constants, but suppose it

[1]`https://github.com/StefanKoppier/BVJ`

is initialized with some computationally heavy function, and the initialization is inside some loop. This will generate many program paths which are hard to verify.

Listing 1: Java program containing array initialization.

```java
class Main {
    public static void main() {
        int[] array = new int[] { 1, 2 };
    }
}
```

Language: Java

Listing 2: Compiled C program containing the array initialization.

```c
#include <stdlib.h>
struct Int_Array {
    __int32 * elements; __int32 length;
};
struct Int_Array * new_Array$0()
{
    __int32 initial[2] = { 1, 2 };
    struct Int_Array * this = malloc(sizeof(struct Int_Array));
    {
        this->length = 2;
        this->elements = calloc(2, sizeof(__int32));
        for (__int32 i0 = 0; i0 < 2; ++i0)
            this->elements[i0] = initial[i0];
    }
    return this;
}
void main()
{
    struct Int_Array * array = new_Array$0();
}
```

Language: C

We finalized the C version of the tool as version v0.1, which can be found on GitHub[2]. The final C version has support for a similar Java subset as the Java version of the tool, excluding the exception handling system.

A second approach: compiling to Java

The second approach was to compile our Java program to Java program paths, and verify these using JBMC [2].

The main advantage of this approach is that we are semantically much closer to our input program, only a few modifications to the compiled program paths are needed. The main initial advantage is that the exception handling system is supported, which doesn't require us to define our own. Other advantages are that this approach is more extensible; it is easier to add new features, for it not requires a equivalent semantical C definition of the new features.

A disadvantage of the Java version of the tool is that JBMC is less mature than CBMC. But the upside is that JBMC received funding and thus is in active development.

The rest of this document describes the Java version of the tool.

[2]https://github.com/StefanKoppier/BVJ/releases/tag/v0.1

Contents

1 JBMC: a bounded model checking tool for Java bytecode **7**
1.1 Future work of JBMC . . . 7
1.2 A small survey of formal verification tools for Java . . . 7

2 The phases of the tool **9**
2.1 Lexing and Parsing . . . 10
2.1.1 The supported subset of Java . . . 10
2.2 Analysis . . . 11
2.2.1 The construction of the control flow graph . . . 11
2.2.2 Reachability analysis . . . 14
2.3 Linearization . . . 15
2.3.1 The construction of the program paths . . . 15
2.4 Compilation . . . 15
2.4.1 External filtering and modification of program paths . . . 16
2.5 Verification . . . 16

3 How to use the tool **17**
3.1 Prerequisites . . . 17

4 Conclusion **19**
4.1 Possible future work . . . 19

A Examples **25**
A.1 The Fibonacci sequence . . . 25
A.2 Merge sort . . . 27

Chapter 1

JBMC: a bounded model checking tool for Java bytecode

The tool is built upon JBMC, an actively developed bounded model checker for verifying Java bytecode [2]. JBMC is built upon the CProver framework, which is the basis of CBMC: an industrial strength bounded model checker for C and C++ programs. JBMC works by translating Java bytecode into what they call a GOTO program; a control flow graph representation of the program. This GOTO program is symbolically executed to generate a bit-vector formula. This bit-vector formula is verified by a SAT or SMT solver, which by default is MiniSat [4].

JBMC has support for checking that runtime exceptions do not occur and that user-defined assertions are valid. Assertions are natively supported in Java and can be written using the `assert e;` and `assert e : "foo";` statements. Assumptions can be written using a method call: `CProver.assume(e);`.

The team of JBMC actively maintains a model of the Java Class Library (JCL), the standard library included with the Java language. Currently, the main support of the model is focussed on including the Exception and String types. The model is maintained on GitHub[1].

1.1 Future work of JBMC

The team working on JBMC claim in their tool paper [2] that there currently is no support for the Java Native Interface, reflection, generics, lambda expressions, or multi-threaded programs. Their current plan is to extend JBMC to support the latter three.

1.2 A small survey of formal verification tools for Java

Besides JBMC, there exist several tools that aim to achieve a similar result.

JayHorn is a framework for verifying Java bytecode [7]. It works by transforming the input to Horn clauses and solving these clauses using a theorem prover. JayHorn claims to have soundness as its main focus. Although they claim soundness as their main focus, the development has not yet reached a point that this is the case. They do not have a stable release thus far that is sound. The project has recent activity in their GitHub codebase, so it seems to be under active development. On their GitHub page[2], they state that the following features are not fully sound:

- JNI, implicit method invocations (finalizers, class initializers, Thread.¡init¿, etc.)
- integer overflow

[1]https://github.com/diffblue/java-models-library
[2]https://github.com/jayhorn/jayhorn

- exceptions and flow related to that
- reflection API (e.g., Method.invoke(), Class.newInstance)
- invokedynamic
- code generation at runtime, dynamic loading
- different class loaders
- key native methods (Object.run, Object.doPrivileged)

Java Pathfinder (jpf) is a system to fomrally verify Java bytecode [6]. JPF focusses on verifying the absence of deadlocks and user-defined assertions. JPF works by constructing a Promela program, which can be fed into SPIN to be verified for correctness. JPF was developed at the NASA Ames Research Center and was made open source in 2015 under the Apache 2.0 license. The project is hosted on GitHub[3].

Clausio Giovanni Demartini et al. [3] describe a similar approach as jpf, the Java2Spin translator. Is also verifies the absence of deadline, by translating a Java programs into a Promela program. Although successful experimental results, I couldn't find a tool based on this paper under active development.

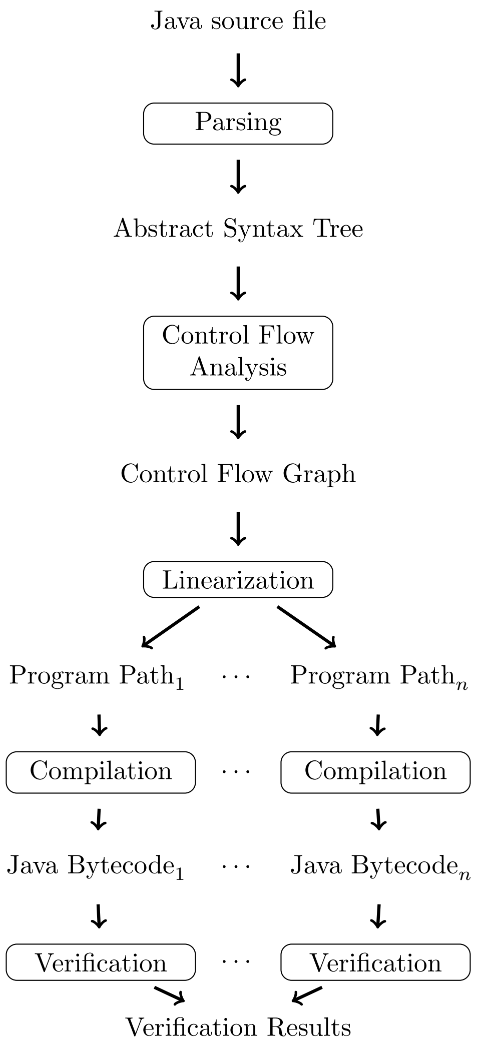
[3]https://github.com/javapathfinder/

Chapter 2

The phases of the tool

The tool consists of five main phases: parsing, control flow analysis, linearization, compilation and verification. A global overview of all these phases and their inputs and outputs can be found in figure 2.1.

Figure 2.1: An overview of the phases of the tool.



Java source file
Parsing
Abstract Syntax Tree
Control Flow Analysis
Control Flow Graph
Linearization
Program Path$_1$ ··· Program Path$_n$
Compilation ··· Compilation
Java Bytecode$_1$ ··· Java Bytecode$_n$
Verification ··· Verification
Verification Results

All phases are chained into one phase, the Complete phase. This phase is defined in the `Complete` module and is the main API of the tool.

2.1 Lexing and Parsing

Lexing and parsing is the first phase of the tool. It consists of two subphases: the lexer and parser, and the syntax transformation. The complete phase takes a `String` as input, and transforms it to a `CompilationUnit'` defined in the `Parsing.Syntax` module. The `CompilationUnit'` can be seen as the root node of the abstract syntax tree.

The lexing and parsing is done using an intermediate abstract syntax tree, defined in the library language-java. This library also contains the lexer and parser that is used within the tool.

The output of the parser of the language-java library is fed into a syntax transformation subphase. This phase transforms the abstract syntax tree of the language-java library into the abstract syntax tree defined in `Parsing.Syntax`. Our abstract syntax tree is almost an one-to-one correspondence of the abstract syntax tree defined in language-java.

The definitions in `Parsing.Syntax` are contained in an attribute grammar file, which can be built using the Attribute Grammar System of Utrecht University[1]. This system allows easy information retrieval of the abstract syntax tree, but at the cost that we cannot use the abstract syntax tree defined in the language-java library directly.

2.1.1 The supported subset of Java

Only a subset of the Java language is supported. There is support for a single Java source file containing one or more class declarations. Interfaces and enum declarations are not supported.

Classes may contain fields, methods and constructors. All modifiers on these levels are supported, including annotations. Classes may not contain generics, note that JBMC also does not have support for generics. Methods and constructors may not be overloaded, there cannot exist two methods with the same name in the same class.

Methods and constructors contain statements, of which we distinct two types. Compound statements, and basic statements. Compound statements are statements which itself contain statements. Basic statements are statements which do not contain statements. A constructor may not call its base constructor explicitly. The compound statements that are supported are:

- A block, introduced using curly braces;
- an `if else` statement;
- a `while` loop;
- a `try catch finally` statement;
- a `for` loop.

Note that `while`- and `for` loops can be labeled, e.g. `l: while(g)`.

The basic statements that are are supported are:

- A variable declaration;
- a skip statement;
- an expression statement;
- an `assert` statement;
- a `break` statement;
- a `continue` statement;
- a `return` statement;
- a `throw` statement.

An assume statement can be written as `CProver.assume(e)`. The package that contains the `CProver` class does not have to be included as this is done by the compiler.

There is one contextual issue: having a `break` or `continue` statement transferring the control flow from a `try`, `catch` or `finally` statement. For example, see listing 2.1. This code is semantically valid, but fails to compile correctly.

[1] `hackage.haskell.org/package/uuagc`

Listing 2.1: Breaking from a catch, which is problematic.

```java
void foo() {
    int x = 0;
    while (true) {
        try {
            x = x / x;
        } catch (ArithmeticException e) {
            break;
        }
    }
}
```

Language: Java

All expression constructs are supported, except for three. There is no support for casting, the `instanceof` operator, and lambda expressions. Lambda expressions are unsupported by JBMC.

2.2 Analysis

The second phase of the tool is the Control Flow Analysis. It takes a `CompilationUnit'` as input and transforms it into a `CFG`, defined in the `Analysis.CFG` module.

The control flow graph is implemented using the inductive graph representation of the fgl library. The inductive graph is a graph suitable for functional style programming. They are developed by Martin Erwig [5].

Definition 1 *The Control Flow Graph (CFG) of a Java program P is a directed graph $G_P = (V, E)$ where*

- *V is the set of nodes. Each node is one of either: 1. a method entry point; 2. a method exit point; 3. an invocation of an method; 4. a statement, for initializer, or a for update; 5. a catch block; 6. a finally block.*
- *E is the set of edges. Each edge is one of either: 1. an intraprocedural edge; 2. an interprocedural edge; 3. an intraprocedural block entry edge; 4. an intraprodudural block exit edge; 5. an intraprocedural block entry and exit edge.*

A block is a point where the control flow enters a new scope. We define five types of blocks:

A basic block	induced by a pair of brackets not following a method, conditional block, or try catch finally blocks.
A conditional block	induced by a conditional block, i.e. an if then else statement or a loop structure.
The try, catch and finally blocks	induced by a try catch finally statement.

2.2.1 The construction of the control flow graph

The construction of the control flow graph is based on the control flow analysis described in Principles of Program Analysis by Nielson et al. [8]. It labels each statement in the program with an unique number, and each statement has an initial label and final labels. We will describe the initial, the final labels, and the set of vertices and set of edges for each kind of compound statement below.

We denote the initial node of a statement by $init : Statement \rightarrow Node$, the final nodes of a statement by $final : Statement \rightarrow \{Node\}$. We use $V(x)$ and $E(x)$ to denote the set of vertices and edges generated by object x. We use $node(x)$, $edge(x)$, and $edges(x, y)$ to denote a new node, or new edge(s) for objects x and y.

A constructor or method

Given a constructor or method M with body S, we will have:

$$V = \{node(M_{entry}), node(M_{exit})\} \cup V(S)$$

$$E = \begin{cases} edge(node(M_{entry}), node(M_{exit})) & \text{if } V(S) = \emptyset \\ edge(node(M_{entry}), init(S)) \cup edges(final(S), node(M_{exit})) & \text{otherwise} \end{cases}$$

Or less formally, the vertices are the method entry point the method exit point, and the vertices created by the body of the method. The edges are an edge from the method entry point to the initial of the body, and the edges from the final of the body to the method exit point.

A sequence of statements

Given a sequence S of statements S_1 and S_2. We have that:

$$init(S) = init(S_1)$$

$$final(S) = final(S_2)$$

$$V = V(S_1) \cup V(S_2)$$

$$E = \begin{cases} E(S_1) \\ \quad \text{if } S_1 \text{ is a } \texttt{break} \text{ or } \texttt{continue} \\ \\ edges(final(S_1), init(S_2)) \cup edges(breaks(S_1), init(S_2)) \cup E(S_1) \cup E(S_2) \\ \quad \text{if } S_1 \text{ is a loop} \\ \\ edges(final(S_1), init(S_2)) \cup E(S_1) \cup E(S_2) \\ \quad \text{otherwise} \end{cases}$$

Or less formally, the vertices are the vertices of the statements. The edges are the edges of the statements, and we have three cases:

- We have a `break` or `continue` statement: we end the sequence at the first statement;
- we have a loop statement: we add an edge from the final of the first statement and the break statements of the loop to the initial of second statement;
- we have any other statement: we add an edge from the final of the first statement to the initial of the next statement.

An if else statement

Given an if else statement I, with the body of the true branch $S_\top$ and the body of the false branch $S_\bot$, we will have:

$$
\begin{aligned}
init(I) &= node(I) \\
final(I) &= final(S_\top) \cup final(S_\bot) \\
V &= \{node(I)\} \cup V(S_\top) \cup V(S_\bot) \\
E &= edge(node(I), init(S_\top)) \cup edge(node(I), init(S_\bot)) \cup E(S_\top) \cup E(S_\bot)
\end{aligned}
$$

Or less formally, the vertices are the if else statement itself, and the nodes of the true branch and the false branch. The edges are, an edge from the if else statement itself to the initial of the body of the true branch and to the initial of the body of the false branch.

A while loop

Given a while loop L, with guard g and body S, we have:

$$
\begin{aligned}
init(L) &= \begin{cases} node(L) & \text{if } V(g) = \emptyset \\ init(g) & otherwise \end{cases} \\
final(L) &= \{node(L)\} \\
V &= \{node(L)\} \cup V(S) \cup V(g) \\
E &= edge(node(L), init(S)) \cup edges(final(S), node(L)) \\
&\quad \cup edges(continues(L), node(L)) \cup E(S) \cup E(g)
\end{aligned}
$$

where $continues(L)$ denotes the `continue` statements that belong to the loop L.

Or less formally, the vertices are the loop itself, and the nodes of the loop body. The edges are, an edge from the loop itself to the initial body, edges from the final of the body to the loop itself, edges from the continue statements of this loop to the loop itself and the edges generated by the body of this loop.

Note that a while loop is always followed by a `CProver.assume(!g)` statement.

A try catch finally statement

Given a try statement T with body S_T catch statements $[C_1, \ldots, C_n]$ with bodies $[S_1, \ldots, S_n]$ and possibly a finally statement F with body S_F, we will have:

$$init(T) = node(T)$$

$$final(T) = \begin{cases} final(S_F) & \text{if } T \text{ has a } \texttt{finally} \text{ block} \\ final(S_n) & \text{otherwise} \end{cases}$$

$$V = \{node(T), node(C_i), \ldots, node(C_n)\} \cup \{node(F)\} \cup V(S_T) \cup V(S_1) \cup \ldots \cup V(S_n) \cup S_F$$

$$\begin{aligned} E = {} & edge(node(T), init(S_T)) \cup edges(final(S_T), node(C_1)) \cup edge(node(C_1), init(S_1)) \\ & \cup edges(final(S_1), node(C_2)) \cup \ldots \cup edges(final(C_n), node(F)) \\ & \cup edge(node(F), init(S_F)) \cup E(S_T) \cup E(S_1) \cup \ldots \cup E(S_n) \cup E(S_F) \end{aligned}$$

Or less formally, the vertices are the try statement, catch statements and finally statement with all bodies of these blocks combined. The edges are an edge from the try statement to the init of the try statement itself, and we chain all finals of the following blocks to the nodes of the try statements that follow.

A for loop

Given a for loop L with body S and initialization expression i, guard g, and update expression u.

$$init(L) = \begin{cases} node(i) & \text{if } i \text{ exists} \\ node(L) & \text{otherwise} \end{cases}$$

$$final(L) = \{node(L)\}$$

$$V = \{node(i), node(L), node(u)\} \cup V(S)$$

$$\begin{aligned} E = {} & edge(node(i), node(L)) \cup edge(node(L), init(S)) \cup edges(final(S), node(u)) \\ & \cup edges(node(u), node(L)) \cup edges(continues(L), node(u)) \cup E(S) \end{aligned}$$

Or less formally, we have a node for the loop itself, the initialization, the update and the body of the loop. We have an edge from the update to the loop itself, an edge from the loop itself to the initial of the body. We have an edge from the final of the body to the update, or the loop itself, depending if the update exists. Finally, we have an edge from the update to the loop itself, if the update exists.

Note that a for loop is always followed by a `CProver.assume(!g)` statement.

2.2.2 Reachability analysis

The tool has a subphase in place which allows for reachability analysis. This reachability analysis is a depth-first search starting at node $v \in V$ which corresponds to the method entry point of the method to be verified. This subphase is currently disabled as it may construct correct, but hard to read graphs, when verifying a specific method that is not `main`.

2.3 Linearization

The third phase is the linearization phase; the phase that takes a control flow graph and generates all program paths from the starting point of the method to be verified until the last point of method to be verified, up to the length k.

The verification starts at the method entry point node of the method to be verified and ends at the method exit point of the method to verified if there are no more stack frames on the call stack. This is due to recursive method calls.

The length of a program paths is defined by the number of basic statements in the program path, not counting any empty statement, i.e. `;`. This choice is made as most realistic programs do not contain empty statements, but the tool inserts skip statements at multiple places, keeping k similar to the original program.

A program path is defined as a list of `PathStmt`, a 2-tuple containing a `PathType` and a `PathMetaInfo`. The `PathType` can be either a basic statement, an block entry or an block exit. The `PathMetaInfo` contains information about the package, class and method, and the call name of the method this `PathType` is enclosed in.

2.3.1 The construction of the program paths

The program path construction algorithm works as follows. We traverse edges of the control flow graph from the method entry node of the method to be verified until we either reach the final node, or the generated program path exceeds the maximum length k.

During the traversal, we keep track of an accumulator `PathAccumulator` which is a 5-tuple containing:

- The `CallHistory`: the number of times each method is called;
- the `StmtManipulations`: the method renaming that has to be performed at each node;
- the `CallStack`: the stack of method calls;
- the `ProgramPaths`: the program paths generated thus far;
- and the `Int`: the length we can append to the current paths, i.e. $k-$current length of the program paths generated thus far.

When we reach a node that contains a statement, we append it to all program paths and check if this node contains an method calls. If this is the case, we traverse the expression in the statement and rename the method calls, this is done in the `Linearization.Renaming` module.

When we traverse an edge, we check if the edge contains a block entry or exit. If this is the case, we add this block entry or exit to all paths and continue the traversal. The complete algorithm can be found in the `Linearization.Path` module.

2.4 Compilation

The fourth phase is the compilation phase. In this file, the program paths are reconstructed into valid Java programs. These Java programs are then compiled into Java bytecode using `javac` and packed into jar files using `jar`. This phase runs in parallel, speeding up the compilation. The number of worker threads can be changed by tweaking a verification argument.

The compilation phase works by reconstructing each program path back into a `CompilationUnit'`, which is then compiled and packed. This reconstruction works by grouping the program path by its classes, then by its methods and constructors, then by its blocks, and finally by its statements.

The compiler does not make many modifications to the program paths: the main modification is that constructors, and constructor calls, are changed into static methods and regular method invocations. This modification is necessary as constructors cannot have names that are not equal to the class name. We need unique constructor names as same the same constructor can be called multiple times, but each having an unique program sub-path associated with that constructor call.

2.4.1 External filtering and modification of program paths

The `Arguments` value given to the verification tool contains a record value, shown in listing 2.2. This function allows for the adjustment and filtering of specific abstract syntax trees that are created from the generated program paths.

Listing 2.2: Filtering record value of `Arguments`.

```
pathFilter :: CompilationUnit' -> Maybe CompilationUnit'
```

When this function results in `Nothing`, the abstract syntax tree will be filtered. When this function results in `Just p`, the abstract syntax tree `p` will be verified.

2.5 Verification

The fifth and final phase is the verification phase. This phase runs JBMC on each compiled Java program and parses the XML output of JBMC. Similar to the compilation phase, this phase runs in parallel.

Chapter 3

How to use the tool

The tool can be used in two ways: using GHCi and by running the excutable. The tool can be build using the command `make repl`, which also starts GHCi.

When GHCi is used, there are 3 functions which provide an easy to use interface to verify: `verify`, `verifyWithMaximumDepth`, and `verifyFunctionWithMaximumDepth`. There is an `arguments` constant defined which allow for the tweaking of the arguments for the verification process.

When running the executable, the arguments in listing 3.1 can be used to tweak the argument for the verification process. For example, the command `bvj "examples/Test.java" -c -k10 -t4` verifies the file `examples/Test.java` printing minimal information, with a maximum program path length of 10 and uses 4 threads.

Listing 3.1: Argument for the executable

```
Parameters:
File    Path to the file to be verified.

Flags:
-c           --compact                          Display less information.
-r           --remove                           Remove the output files.
-f[FUNCTION] --function[=FUNCTION]              Verify the function.
-k[DEPTH]    --depth[=DEPTH]                    Maximum program path generation depth.
-t[THREADS]  --threads[=THREADS]                Number of threads.
-u[UNWIND]   --unwind[=UNWIND]                  Maximum loop unwinding in JBMC.
             --verification-depth[=VER-DEPTH]  Maximum depth in JBMC.
```

3.1 Prerequisites

Using the tool requires the following:

1. The Java development kit installed, ensure that `javac` and `jar` can be found in the PATH variable;
2. the JBMC tool, ensure that that `jbmc` can be found in the PATH variable;
3. the following Haskell libraries:

   - `fgl 5.7.0.0`
   - `directory 1.3.0.2`
   - `terminal-progress-bar`
   - `command`
   - `filepath`
   - `dates`
   - `pretty`
   - `xeno`
   - `utf8-string`
   - `language-java`
   - `transformers`
   - `containers`
   - `parallel-io`
   - `split`
   - `simple-get-opt`

Chapter 4

Conclusion

Concluding, we presented a bounded model checking tool which covers a subset of the Java language. The examples in the appendix show that the tool can correctly detect errors in algorithms, like merge sort. Making this a feasible approach to formally verify Java source code, instead of Java bytecode, which other bounded model checking tools focus on.

4.1 Possible future work

Extend the subset of the Java language

Currently, the tool only allows for the verification of a subset of the Java language. This can be extended to allow for more Java programs to be verified.

One specific example is allowing `break` and `continue` statements inside a `try`, `catch`, or `finally` block inside a loop.

Add support for multi-threaded programs

The developers of JBMC claim to have multi-threading support as one of their top priorities. When JBMC supports multi-threading, the tool can be extended to support multi-threading as well.

Bibliography

[1] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[2] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. volume 10981 of *LNCS*, pages 183–190. Springer, 2018.

[3] Claudio Giovanni Demartini, Radu Iosif, and Riccardo Sisto. Modeling and validation of java multithreading applications using spin. 1998.

[4] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[5] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[6] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[7] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. Jayhorn: A framework for verifying java programs. In *International Conference on Computer Aided Verification*, pages 352–358. Springer, 2016.

[8] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[9] John Toman, Stuart Pernsteiner, and Emina Torlak. Crust: A bounded verifier for rust (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 75–80. IEEE, 2015.

Appendices

Appendix A

Examples

A.1 The Fibonacci sequence

Let us have a look at the program in listing A.1. This program computes the second fibonacci number, and asserts if it is equal to 1. This program generates the control flow graph shown in figure A.1.

Listing A.1: Program verifying the fib method.

```
class Main {
    public static void main(String[] argv) {
        assert fib(2) == 1 : "fib(2) != 1";
    }

    public static int fib(int x) {
        if (x == 0) {
            return 0;
        } else if (x == 1) {
            return 1;
        } else {
            return fib(x - 1) + fib(x - 2);
        }
    }
}
```

Language: Java

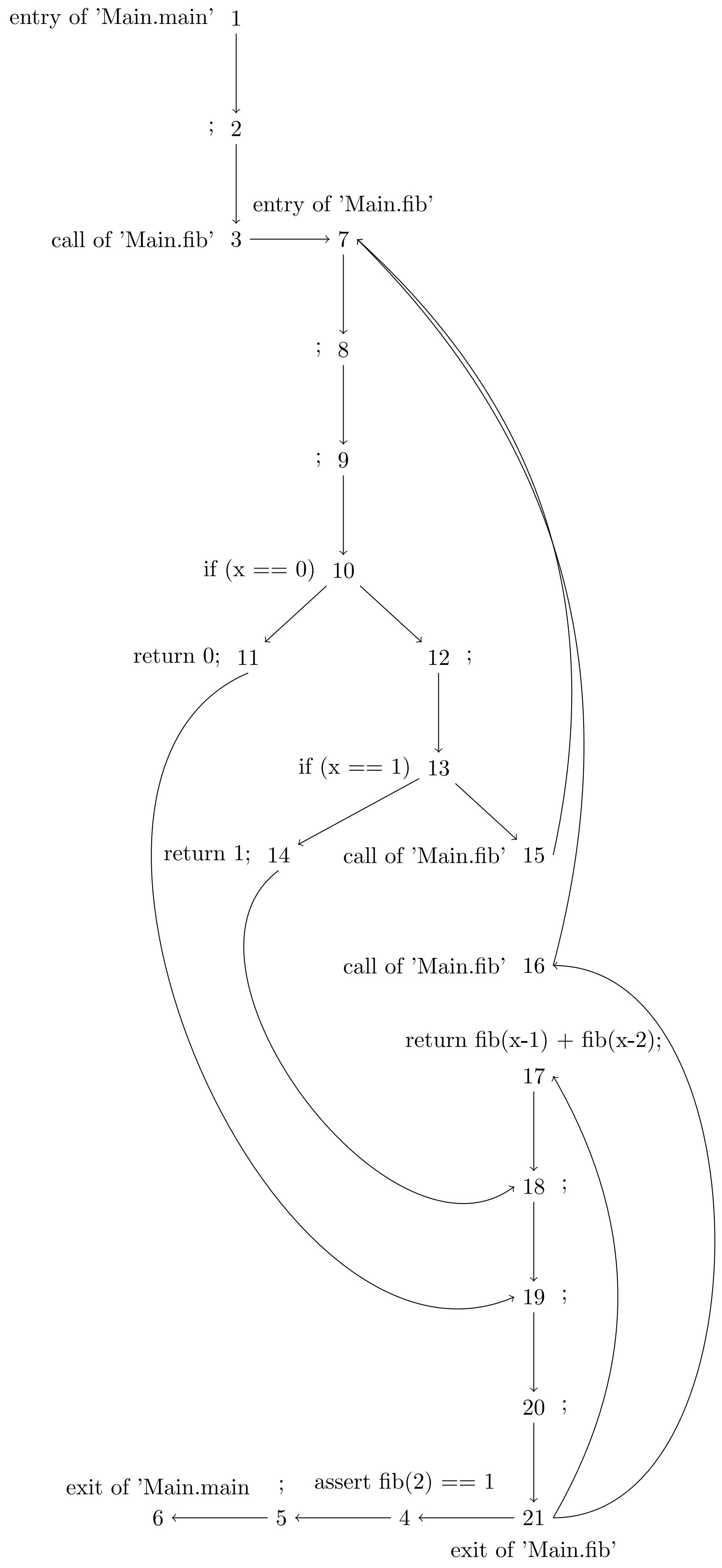
When we verify this program using $k = 10$, we get the following five program paths, of which the shortest is:

```
[CProver.assume(x==0); { return 0; } assert Main_fib0(2)==1 : "fib(2) != 1";]
```

The tool shows that all five program paths are correct.

Modifying the `assert fib(2) == 1 : "fib(2) != 1";` to be `assert fib(2) == 100 : "fib(2) != 100";` and running the tool again shows that the assertion does not hold.

Figure A.1: Control flow graph of listing A.1.



A.2 Merge sort

Let us have a look at the program in listing A.2. This program defined an array `elems` which is passed to the `sort` method, which is an implementation of merge sort.

The tool correctly verifies the assertion that the array is sorted.

Listing A.2: Program verifying the sort method.

```
class Main {
    public static void main(String[] argv) {
        int[] elems = new int[] { 2, 1 };
        sort(elems, 0, elems.length - 1);
        assert (elems[0] == 1) && (elems[1] == 2);
    }

    public static int partition(int[] array, int low, int high)
    {
        int pivot = array[high];
        int i = low - 1;
        int temp;

        for (int j = low; j < high; j++) {
            if (array[j] <= pivot)
            {
                i++;
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        temp = array[i+1];
        array[i+1] = array[high];
        array[high] = temp;

        return i + 1;
    }

    public static void sort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);
            sort(array, low, pi - 1);
            sort(array, pi + 1, high);
        }
    }
}
```

Language: Java