A Bounded Verification Tool for Java

Stefan Koppier

March 16, 2019

Introduction

Contents

Chapter 1

Features

The tool supports

1.1 Unsupported language features

We provide a comprehensive list describing each language feature that is not supporte by the tool. Note that if the reader is reading this document in a pdf viewer, the names of the lists provide a link to sources for more information about the missing part.

1.1.1 Object-oriented paradigm

No support for object-oriented concepts, except for the declaration of classes. The following three concepts are the main ones that are not supported. Anything that is built upon these conceps, e.g. calling a base class constructor, is also not supported.

inheritance	Inheritance of a class or interface.
interfaces	An interface declaration.
abstract classes	A class or method without an implementation.

1.1.2 Statements

do while loops	`do { } while(guard)`
for iterator loops	`for (int item : numbers) { }`
synchronized statements	`synchronized(x) { }`

1.1.3 Expressions

lambda expressions	`(x) -> x*x`
casting	`(int)3.0`
instance of	`x instanceof Integer`
method reference	`Foo::bar`

1.1.4 Others

static initializer blocks	Static block execution, which should be executed when the class is loaded.
static field initializer	Static class field initialization, which should be assigned when the class is loaded.
nested class definitions	Class declarations inside classes or inside methods.

generics	Generic types, of any kind.
enums	Enum declarations.

Chapter 2

On the semantical differences between Java and C

2.1 Types

The supported Java subset consists of three kind of types: primitives, (multi-dimensional) arrays, and classes.

In Java, primitives are stack-allocated, and arrays and classes are heap-allocated. This requires primitives to be treated as plain types, and arrays and objects to be treated as pointers.

2.1.1 Primitive data types

Java defines eight primitive data types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. All integral values are signed.

C defines one boolean type: `_Bool`, five standard signed integer types: `signed char`, `short int`, `int`, `long int`, `long long int`, and three standard floating point types: `float`, `double`, and `long double` [1, p. 40]. The downside of the integral primitives is that their size is not exactly specified, and is thus implementation specific. For example, the `int` primtive can be 32- or 64-bit, depending on the implementation. Luckily, CBMC defines four exact sized integral values: `__int8`, `__int16`, `__int32`, and `__int64` [2, p. 39].

We can define an exact mapping of the primitve data types of Java, to those of C and CBMC. This mapping can be found in 2.1.

Table 2.1: Mapping of the primitive data types.

Type	Description	C equivalent
`boolean`	true or false	`_Bool`
`char`	16-bit Unicode value	
`byte`	8-bit signed integral value	`__int8`
`short`	16-bit signed integral value	`__int16`
`int`	32-bit signed integral value	`__int32`
`long`	64-bit signed integral value	`__int64`
`float`	IEEE 754 32-bit floating point value	`float`
`double`	IEEE 754 64-bit floating point value	`double`

2.1.2 Arrays

Both Java and C have a concept of an array, but they are vastly different. In Java an array is an heap-allocated structure, in C an array is a stack-allocated structure, unless it explicitly allocated on the heap. The length of an array is contained in the Java array structure, this needs to be maintained explicitly in C.

To ease both differences, the choice is made to map the Java type of an array to a C struct, containing the elements in the array and the length of the array.

For example, the type `int[]` will be mapped to:

```
struct Int_Array {
    __int32 *elements;
    __int32 length;
};
```

Language: C

Multi-dimensional arrays are mapped in the same way. For example, the type `int[][]` will be mapped to:

```
struct Int_Array {
    __int32 *elements;
    __int32 length;
};

struct Int_Array_Array {
    struct Int_Array **elements;
    __int32 length;
};
```

Language: C

2.1.3 Classes

There is no direct translation of a Java class to a C class, as C does not have the concept of a class. A Java class consists of three components:

Fields — Fields can be divided in two sets, non-static fields, which are part of an instantiation of the class, and static fields, which are not part of an instantiation.

Methods — Methods can also be divided in two sets, non-static methods, which have an hidden `this` parameter to the object the method is called on, and static methods, which are not called on a specific object.

Constructors — The constructors are always static, and return an object of the type the constructor is defined in.

The class will be mapped to a struct declaration, containing all non-static fields of that class. All static fields will be mapped to a global variable declaration. In the mapping to C, methods and constructors are no longer part of the type itself, but a method will be declared for each invocation. The mapping of methods and constructors is be described in .

reference the mapping of methods.

For example, given the following class `Foo`, without its method and constructor declarations:

```
class Foo {
    static bool b;
    int x;
}
```

Language: Java

will be mapped to the following declarations in C

```
_Bool Foo_b;

struct Foo {
    __int32 x;
};
```

Language: C

2.2 Expressions

2.2.1 Literals

2.2.2 Operators

2.2.3 Assignment

2.2.4 Method invocation

2.2.5 Array creation

2.3 Statements

Chapter 3

Phases of the Tool

3.1 Lexing and Parsing

3.2 Analysis

3.2.1 Control Flow Analysis

We Control Flow Analysis (CFA) is the first step in the analysis process. We take the abstract syntax tree of the input program and transform it into the control flow graph (CFG).

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

- *V is the set of nodes. Each node is one of either: the entry point of a method, the exit point of a method, a method call, or a basic block.*
- *E is the set of edges. Each edge is one of either: an interprocedural edge, an intraprocedural edge, or a conditional edge.*

Intraprocedural Control Flow Analysis

The intraprocedural Control Flow Analysis, that is, the Control Flow within a single method, is constructed as follows:

For every statement, a basic block is inserted, containing that statement.

For the compound statements, we insert edges using the following approach:

3.2.2 Reachability Analysis

Given a Control Flow Graph, a depth-first search is performed on the node $v \in V$ that corresponds to the entry point of the method to be verified. The subgraph containing only the nodes that are reached from v is constructed.

3.3 Compilation

Chapter 4

CProver

4.1 Properties

array bounds	`while` `switch`
pointer	test
division by zero	test
arithmetic over- and underflow	test
shift greater than bit-width	test
floating-point for +/-Inf	test
floating-point for NaN	test
user assertions	test

Bibliography

[1] International standard iso/iec iso/iec 9899:201x programming languages c. Standard, International Organization for Standardization, Geneva, CH, 2011.

[2] Daniel Kroening. *The CProver User Manual*.