A Bounded Verification Tool for Java

Stefan Koppier

February 11, 2019

Introduction

Contents

Chapter 1

On the semantical differences between Java and C

1.1 Types

1.1.1 Primitive data types

Java defines eight primitive data types: **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**, **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 size 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 1.1.

Table 1.1: 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**

1.1.2 Arrays

1.1.3 Strings

1.1.4 Classes

1.1.5 Interfaces

1.1.6 Enum

1.2 Expressions

1.2.1 Literals

1.2.2 Operators

1.2.3 Assignment

1.3 Namespaces

Chapter 2

Phases of the Tool

2.1 Lexing and Parsing

2.2 Analysis

2.2.1 Syntax Transformation

2.2.2 Control Flow Analysis

Definition 1

Define a basic block.

Definition 2 *The Control Flow Graph (CFG) of a method M is a directed graph $G_M = (V, E, s)$ where*

- *V is the set of nodes. Each vertex v in V is a pair consisting of an integral label, and a basic block. There exists a vertex v for every basic block in the method M.*

- *E is the set of edges. There exists an edge (u, v) if and only if the basic block of v follows the basic block of v in the method M.*

- *s is the vertex in V that is the initial basic block in the method M.*

Definition 3 *The Extended Control Flow Graph (ECFG) of a program P is a directed graph $G_P = (V, E, s)$ where*

- *V is the set of nodes. Each vertex v in V is a pair consisting of an integral label, and a CFG. There exists a vertex v for every method M in P.*

- *E is the set of edges. There exists an edge (u, v) if and only if the method of u invokes the method of v.*

- *s is the initial method of the program P.*

We perform the Control Flow Analysis (CFA) in a way similar to that described by Nielson et al. [3].

We define the CFA as:

$$
\begin{aligned}
init([x = e]^l) &= l \\
init([S_1];\ [S_2]) &= init(S_1) \\
init([\textbf{assert}\ e\ g]^l) &= l \\
init([\textbf{assume}\ e\ g]^l) &= l \\
init([\textbf{break}\ x]^l) &= l \\
init([\textbf{continue}\ x]^l) &= l \\
init(\textbf{if}\ ([e]^l)\ \{S\}) &= l \\
init(\textbf{if}\ ([e]^l)\ \{S_1\}\ \textbf{else}\ \{S_2\}) &= l \\
init(\textbf{while}\ ([e]^l)\ \{S\}) &= l \\
init(\textbf{for}\ (;\ [e]^l;\ \dots)\ \{S\}) &= l \\
init(\textbf{for}\ ([i]^{l'};\ [e]^l;\ \dots)\ \{S\}) &= l'
\end{aligned}
$$

$$
\begin{aligned}
final([x = e]^l) &= \{l\} \\
final([S_1];\ [S_2]) &= final(S_2) \\
final([\textbf{assert}\ e\ g]^l) &= \{l\} \\
final([\textbf{assume}\ e\ g]^l) &= \{l\} \\
final([\textbf{break}\ x]^l) &= \{l\} \\
final([\textbf{continue}\ x]^l) &= \{l\} \\
final(\textbf{if}\ ([e]^l)\ \{S\}) &= final(S) \\
final(\textbf{if}\ ([e]^l)\ \{S_1\}\ \textbf{else}\ \{S_2\}) &= final(S_1) \cup final(S_2) \\
final(\textbf{while}\ ([e]^l)\ \{S\}) &= \{l\} \cup breaks_0(S) \\
final(\textbf{for}\ (\dots;\ [e]^l;\ \dots)\ \{S\}) &= \{l\} \cup breaks_0(S) \\
final(x:\ \textbf{while}\ ([e]^l)\ \{S\}) &= \{l\} \cup breaks_x(S) \\
final(x:\ \textbf{for}\ (\dots;\ [e]^l;\ \dots)\ \{S\}) &= \{l\} \cup breaks_x(S)
\end{aligned}
$$

$$
\begin{aligned}
flow([x = e]^l) &= \emptyset \\
flow([S];\ [\textbf{break}\ x]^l) &= flow(S) \\
flow([S];\ [\textbf{continue}\ x]^l) &= flow(S) \\
flow([S_1];\ [S_2]) &= flow(S_1) \cup flow(S_2) \cup \{(l, init(S_2) \mid \in final(S_1))\} \\
flow([\textbf{assert}\ e\ g]^l) &= \emptyset \\
flow([\textbf{assume}\ e\ g]^l) &= \emptyset \\
flow([\textbf{break}\ x]^l) &= \emptyset \\
flow([\textbf{continue}\ x]^l) &= \emptyset \\
flow(\textbf{if}\ ([e]^l)\ \{S\}) &= flow(S) \cup (l, init(S)) \\
flow(\textbf{if}\ ([e]^l)\ \{S_1\}\ \textbf{else}\ \{S_2\}) &= flow(S_1) \cup flow(S_2) \cup (l, init(S_1)) \cup (l, init(S_2)) \\
flow(\textbf{while}\ ([e]^l)\ \{S\}) &= todo \\
flow(\textbf{for}\ (;\ [e]^l;\ \dots)\ \{S\}) &= todo \\
flow(\textbf{for}\ ([i]^{l'};\ [e]^l;\ \dots)\ \{S\}) &= todo
\end{aligned}
$$

2.2.3 Reachability Analysis

Chapter 3

CProver

3.1 Properties

array bounds	test
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*.

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