Plan: Bounded Verification of Java

S. Koppier, 6002978

January 14, 2019

1 Java subset

We consider the following language constructs described below. Note that the object oriented concepts, the do while loop, the switch, exceptional flow, and synchronized behaviour are not supported.

IfThen The basic if loop structure.

IfThenElse The basic if then else structure.

While The basic while loop structure.

For The for statement, which can be transformed in a while statement.

Break The break statement.

Continue The continue statement.

Return The return statement.

Empty The empty statement.

Assert The assert statement, to be translated to cbmc assertions.

Expressions The complete set of expressions should be supported.

Annotations These annotations might be useful to define the verification DSL.

2 Tasks

2.1 Library

2.1.1 Lexing and Parsing

We need to determine what subset of Java that will be implemented. There exists a hackage package, language-java, which can likely take care of the parsing subproblem.

The concrete tasks that are associated with this part are:

1. Determine the subset of Java that will be supported.
2. Create a phase which allows us to parse Java files into an AST.

2.1.2 Control Flow Analysis

We need to transform the resulting AST of the parsing phase into the CFG. This can be done in either Haskell or using an attribute grammar, e.g. UUAGC.

The attribute grammar system UUAGC does not support extensions of existing data types, thus we have the option of branching the existing Java parser, and creating a copy of the AST in UUAGC code. A good starting point for this task is described by Nilsson-Nyman et al. [1].

Another option is to use the existing fold, and write the CFA directly in Haskell. When Haskell is used directly, we can use the existing fold. But the downside of this is that it will likely result in hard to write and unmaintainable code.

The concrete tasks that are associated with this part are:

1. Design and create the data type of the CFG.
2. Create a phase which allows us to transform an AST into the CFG.
3. Implement the Control Flow Analysis for Java; preferably using an attribute grammar system.

2.1.3 Program paths

We need to transform the CFG into a set of program paths, each of length at most n. Each branch in the CFG will result in an extra program path, as was taught during the course Program Semantics and Verification.

The concrete tasks that are associated with this part are:

1. Unwind loops: `while`, `for`, `do while`.
2. Remove branching: `if else`, `switch`.
3. Find out how to deal with exceptions: `throw`, `catch`.
4. Unwind recursion, which can be done in a way similar to loop unwinding. This needs to be looked into.
5. Create a phase which allows us to transform a CFG into all program paths of length at most n.

2.1.4 Conversion to C

We need to convert each program path into it's C representation. This includes all data types, and called methods.

The concrete tasks that are associated with this part are:

1. Define a mapping between the standard data types: byte, short, int, long, float, double, bool, and if the time allows it char and String.
2. Define a mapping between the language flow constructs: `if`, `goto`.
3. Define a mapping of the memory management: allocation using the `new` keyword and the deallocation.
4. Define a mapping of a class:
   (a) The data type itself;
   (b) the constructors;
   (c) the methods;
   (d) the variables;
   (e) the static methods and variables;
5. Define a mapping of inheritance.
6. Define a mapping of interface implementation.
7. Create a phase which allows us to transform a program path into the C representation.

2.1.5 Verification

We will pass each Java program path converted to it's C program path counterpart into cbmc to do the verification.

The verification options are:

1. **Array bounds checks**
2. **Pointer checks**
3. Memory leaks

4. **Division by zero checks**
5. **Signed arithmetic over- and underflow checks**
6. Pointer arithmetic over- and underflow checks
7. Value representation after type cast
8. **Shift greater than bit-width**
9. **Floating-point for +/ Inf**
10. **Floating-point for NaN**
11. **User assertions**
12. **User assumptions**

The concrete tasks that are associated with this part are:

1. Determine the arguments of cbmc that we should be able to tweak/set.
2. Create a Haskell interface which allow us to use cbmc.
3. Create a phase which allows us to verify a program path in its C representation.

2.2 Application

To showcase and verify the functionality of the library, as described in section 2.1, we need to develop an application using its functionality.

The concrete tasks that are associated with this part are:

1. Develop an application that uses all phases of the library, actually performing the bounded verification.

2.3 Documentation

2.3.1 Project report

Write a document that sums up the activities, functionality, and results of the project.

2.3.2 Manual

Documentation of the written code, i.e. pandoc documentation.

2.3.3 Presentation

Might not be an actual part of the project - give a colloqium talk about the project.

3 Extensions

3.1 Verification DSL

Design and implement a verification DSL which allows to write pre- and postconditions of functions. Something in the form of method annotations could work, but needs more research. For example:

```
@Verify(pre  = ((x, y) -> y != 0)
       ,post = ((x, y, result) -> result == (x / y)))
public static int divide(int x, int y) {
    int result = x / y;
    return result;
}
```

4 Schedule

The project starts on the 4th of February 2019 (week 6) and lasts until the 12th of April 2019 (week 15), which is 10 weeks, 50 days, or 400 hours.

When we follow the schedule described below, we need approximately 270 hours, or 34 days. Thus we have 16 days left, which we can use to research the possibilities for the extension described in section 3.1.

Task	**Hours**
Lexing and Parsing	**8**
Determine the subset of Java that will be supported	4
Create a phase which allows us to parse Java files into an AST	4
Control Flow Analysis	**52**
Design and create the data type of the CFG	4
Create a phase which allows us to transform an AST into the CFG	8
Implement the Control Flow Analysis for Java	40
Program paths	**48**
Unwind loops	4
Remove branching	4
Find out how to deal with exceptions	16
Unwind recursion	16
Create a phase which allows us to transform a CFG into all program paths	8
Conversion to C	**68**
Define a mapping between the standard data types	8
Define a mapping between the language flow constructs	4
Define a mapping of the memory	16
Define a mapping of: data type	4
Define a mapping of: constructors	8
Define a mapping of: methods	8
Define a mapping of: variables	4
Define a mapping of: static methods and variables	8
Create a phase which allows us to transform a program path into the C representation	8
Verification	**28**
Determine the arguments of cbmc that we should be able to tweak/set	4
Create a Haskell interface which allow us to use cbmc	16
Create a phase which allows us to verify a program path in its C representation	8
Application	**48**
Develop an application that uses all phases of the library, actually performing the bounded verification	48
Documentation	**18**
Project report	16
Manual	2
	270 (34 days)

References

[1] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. Declarative intraprocedural flow analysis of java source code. *Electronic Notes in Theoretical Computer Science*, 238(5):155–171, 2009.