Models of Software Systems 2022/23

Raul Barbosa <rbarbosa@dei.uc.pt>

Assignment 1 – Specifying and verifying Java programs with OpenJML

Abstract

In this assignment you are given two software artefacts with the goal of specifying their behaviour in the Java Modeling Language (JML) and formally verifying their correctness with the OpenJML tool. The first program indicates whether an object of a class is "equal to" some other one. The second program sorts an integer array. The assignment shall be carried out in groups of two students (groups of three students are accepted in exceptional circumstances).

1 Object equality

Consider, in Figure 1, the implementation of a straightforward class, which holds a long value and provides methods to encapsulate its access. At first sight, the `equals()` method may appear to be correct. It calls the `equals()` method of class `Long` to compare "this" object's value with the argument's value.

```
public class Value {
    private Long value;

    public Value(Long v) {
        this.value = v;
    }

    public Long getValue() {
        return value;
    }

    public void setValue(Long v) {
        this.value = v;
    }

    public boolean isStrictlyPositive() {
        return value > 0;
    }

    public boolean isNegative() {
        return value < 0;
    }

    @Override
    public boolean equals(Object o) {
        Value other = (Value) o;
        if(!this.value.equals(other.value))
            return false;
        return true;
    }
}
```

Figure 1: Plain old Java object (POJO) that overrides method `equals()`

Although this implementation of the `equals()` method works correctly in most circumstances, it is inconsistent with the standard specification of the `equals()` method of class `Object`[1]. The standard specification is available through the link at the footer of this page.

The standard specification mentions the behaviour of `x.equals(null)` and, accordingly, we should add the `nullable` modifier to the parameter in `public boolean equals(/*@ nullable @*/ Object o)` in order to allow it to be null. Otherwise, by default, in JML references are never null. By running the static checker using `openjml -esc Value.java` verification fails because there's a bug (you may fix it).

Part 1. Write the JML contract for method `equals()` by specifying its postcondition (and precondition if needed). The contract should be consistent with the standard specification of `Object.equals()`, linked at the footer of this page.

Part 2. Explain the verification failures reported by OpenJML on the original code in Figure 1. This should be done by enabling static checking (ESC) in OpenJML. Correct the code so that the prover is able to formally verify the contract of the `equals()` method.

2 Array sorting

More than verifying a piece of code, the aim of this problem is also to prove the correctness of an algorithm. Code that contains algorithms is often the hardest to understand and to analyse. Figure 2 shows a Java implementation of an algorithm that takes as input an array of integers and sorts it.

```
public class ICantBelieveItCanSort {
    public static void sort(int[] a) {
        for(int i = 0; i < a.length; i++) {
            for(int j = 0; j < a.length; j++) {
                if(a[i] < a[j]) {
                    int tmp = a[i];
                    a[i] = a[j];
                    a[j] = tmp;
                }
            }
        }
    }
}
```

Figure 2: Java implementation of the ICan'tBelieveItCanSort algorithm

Quite simply, the algorithm loops over all pairs of (i, j) values, compares and swaps them. However, on first inspection, it may appear to be incorrect, because j is not constrained relatively to i and the direction of the comparison is intuitively reversed. We need proof.

Part 3. Write the JML contract for method `sort()`, establishing that the array is sorted on method exit. It should also specify that the resulting array is a permutation of the input array.

Part 4. Formally verify that the specification of the `sort()` method holds, using OpenJML's static checker. Annotate loops with the necessary loop invariants (and any other lemmas).

To verify the `sort()` method using OpenJML's static checker (ESC), loops must be annotated with their invariants. It is most often necessary to specify invariants for: constraining the values of the loop indices; inductively stating what's been done so far; describing modified memory locations; and a termination condition. The original paper (`https://arxiv.org/abs/2110.01111`) provides valuable information on the inductive predicates that are needed to obtain a proof.

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)`

3 Recommended approach and reporting

The final report shall consist of a single text file (plain `.txt` or, at most, `.md` or `.tex`) and the `.java` files with the JML specifications written to solve the assignment. The written report shall clearly describe the modeling and verification steps taken to address all parts of the assignment. It should also specify the necessary switches passed to all components: Java compiler and OpenJML.

Informally describe how the correctness properties are formalized from the natural language specification. Arguments must be rigorous even if informal when claiming that the JML specification is coherent and complete. Answer all parts of the assignment and justify any simplifying assumptions you make. The suggested maximum size for the written report is 400 words.