Assessing the Coverage of Formal Specifications

(Extended Abstract)

Dominic Steinhöfel

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
steinhoefel@cs.tu-darmstadt.de

Abstract. Deductive program verification is an intricate and time-consuming task, in spite of significant advances in state-of-the-art program provers. While proving the correctness of programs with respect to existing specifications can already be difficult, it can be even more demanding to come up with sensible specifications for methods and especially for loops. Another issue is related to programs heavily making use of software libraries: Their verification can be considered almost infeasible due to the lack of formal specifications of the libraries. We propose a method for assessing the coverage/strength of formal specifications based on "facts" extracted using heavyweight symbolic execution. We envision that this method can be employed for (1) assisting verification engineers in the incremental specification of programs, (2) comparing different specifications for the same program, and (3) obtaining information for specification generation tools. Our approach has been implemented as a prototype for Java which uses the heavyweight symbolic execution system KeY as a backend. We studied its practicability with several small examples and plan to conduct a more extensive case study in the near future.

1 Introduction and Preliminaries

In the past decades, *deductive software verification* techniques evolved from theoretical approaches reasoning about simple while languages to systems such as Spec#, Frama-C, OpenJML and KeY [1] which are capable of proving complex properties about programs in industrial programming languages such as C, C# and Java [2]. Still, one major challenges remains: The creation of suitable formal specifications for the software that should be proven correct. In [2], a sorting routine of the Java standard library, TimSort, is formally specified and the correctness of the routine w.r.t. the specification is shown using the KeY verifier. Although the specification does not even cover the main aspects of sortedness and permutation, a substantial effort was required for their construction. Current approaches to automatic specification generation are not yet capable of creating specifications that can be used for functional verification in the general case. We propose an approach using Symbolic Execution (SE) to extract "facts" about Java methods specified with the Java Modelling Language (JML) and to check the coverage of provided method and loop specifications w.r.t. those facts. One of the main applications of our method is aiding verification engineers in incrementally increasing the strength of their specifications; also, the facts might

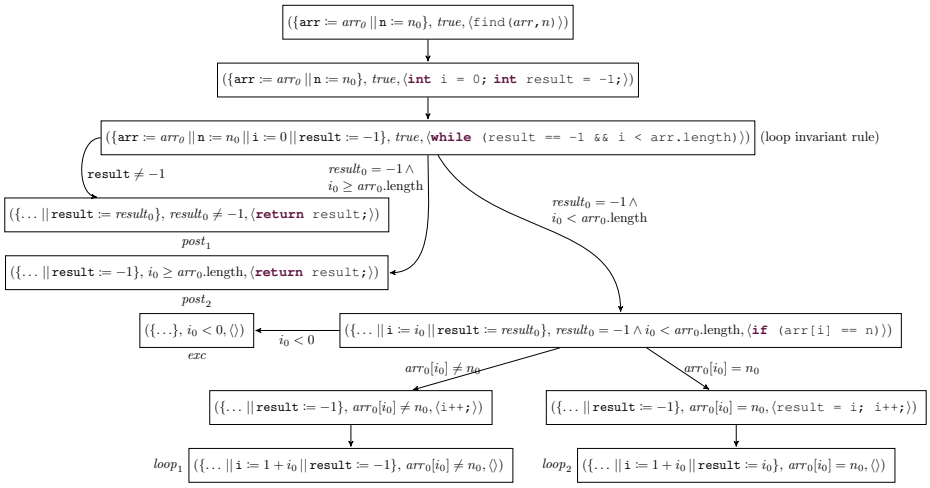

Fig. 1: Simplified SET for the `find` method in Listing 1

be used by specification generation tools to successively create stronger method contracts and loop invariants. One of our visions is that the approach could help comparing different specifications of software libraries submitted to a "collaborative specification" platform, which could be based on an imaginary rewards system in the fashion of StackOverflow.

We introduce some preliminaries, and refer the reader to [1,3] for more details. SE results in an Symbolic Execution Tree (SET) consisting of SE *states*. An SE state is a triple of a *symbolic state* $\{\mathtt{x} := t \,\|\, \ldots\}$, a *path condition* (a closed first-order formula determining a path) and a *program counter*. The latter is a dynamic logic formula with a program, determining the following subtree; in the presentation in this paper, we abbreviate it to the next statement to execute.

2 Extracting Facts by Symbolic Execution

Listing 1 depicts a simple method for finding an integer in an array. Fig. 1 shows the corresponding SET. The tree contains five (symbolically) feasible final states: The states $post_1$ and $post_2$ represent return states after loop termination, while $loop_1$ and $loop_2$ arise from the execution of the loop body. The exceptional state *exc*, in which `i` is negative, is practically infeasible, but still contained in the SET if the loop invariant does not exclude that case. From those states, we extract the following facts: (1) Two *post condition facts* $\mathtt{result} \doteq result_0$ (where $result_0 \neq -1$) and $\mathtt{result} \doteq -1$ (if n was not found) from the states $post_1$ and $post_2$, (2) two *loop body facts* $\mathtt{i} \doteq 1+i_0$ and $\mathtt{result} \doteq -1$ from $loop_1$, (3) another two loop body facts $\mathtt{i} \doteq 1 + i_0$ and $\mathtt{result} \doteq i_0$ from $loop_2$. In addition, we can

```
public int find(int[] arr, int n) {
    int i = 0, result = -1;
    while (result == -1 &&
           i < arr.length) {
        if (arr[i] == n)
            result = i;
        i++;
    }
    return result;
}
```

Listing 1: `find` method

obtain two special fact categories depending on the method post condition and loop invariant: (4) A set of *use case facts* from all *post* nodes, containing one fact for each conjunctive element in the specified post condition of the method. An unsatisfied use case fact means that either the post condition cannot be satisfied by the method, or that the loop invariant is too weak. (5) A set of *exception facts* if the method precondition or loop invariant are not excluding potential runtime exceptions to be thrown. Uncovered exception facts are reported by our tool.

For each fact, we distinguish three cases: Whether the fact is *covered*, i.e., implied by the results of SE and the specification elements; whether it is *"trivially" covered*, i.e., without the specification elements and can thus be ignored in strength estimation, or whether it is *overapproximated*, that is, the fact (e.g., $n \doteq 3$) implies the specification (e.g., $n \geq 0$).

3 Applications

For effectively using facts analysis in incremental specification development, we suggest to begin with an initial, intuitive method post condition. Afterward, the loop invariant (and method precondition) should be strengthened until no more exception or use case facts are uncovered. Finally, post condition and loop body facts can successively be taken into account until the specification reaches a satisfactory state. One result of our analysis is a percentage number of covered facts (the *strength*), where overapproximated facts are weighted by a factor smaller than 1. These numbers can be employed to compare different existing specifications, or to guide the search in an automatic approach for specification generation. In the latter case, also the feedback about facts of the program can be helpful. At our companion website `key-project.org/papers/coverage-formspec`, we provide a Java executable of our tool and an evaluation for the `find` method.

4 Future Work and Conclusion

We presented an approach for estimating the strength of formal specifications written in JML for Java programs. To the best of our knowledge, there is no other approach analyzing the strength, or comparing different versions, of formal specifications. We envision that our technique can be useful for both manually and automatically creating method and loop specifications. In the future, we plan to perform an extensive case study to assess the utility of the approach.

References

1. Ahrendt, W., Beckert, B., et al. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer International Publishing (2016)
2. Gouw, S.d., Rot, J., et al.: OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Pasareanu, C.S. (eds.) Proc. of the 27th Intl. Conf. on Computer Aided Verification. Springer (2015)
3. Scheurer, D., Hähnle, R., et al.: A General Lattice Model for Merging Symbolic Execution Branches. In: Ogata, K., Lawford, M., et al. (eds.) ICFEM 2016, Proceedings. pp. 57–73. Springer International Publishing (2016)