Reachability Analysis for Annotated Code

Mikoláš Janota
School of Computer Science
and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie

Radu Grigore
School of Computer Science
and Informatics
University College Dublin
Belfield, Dublin 4, Ireland

Michał Moskal
Institute of Computer Science
University of Wrocław
ul. Joliot-Curie 15
50-383 Wrocław, Poland
mjm@ii.uni.wroc.pl

ABSTRACT

Well-specified programs enable code reuse and therefore techniques that help programmers to annotate code correctly are valuable. We devised an automated analysis that detects unreachable code in the presence of code annotations. We implemented it as an enhancement of the extended static checker ESC/Java2 where it serves as a check of coherency of specifications and code. In this article we define the notion of semantic unreachability, describe an algorithm for checking it and demonstrate on a case study that it detects a class of errors previously undetected, as well as describe different scenarios of these errors.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Tools; D.2.4 [**Software/Program Verification**]: Formal Methods

General Terms

Verification

Keywords

JML, ESC/Java2

1. INTRODUCTION

Program annotations are logic specifications embedded in the actual program code [16]. They enable programmers to express the intended functionality. Variants of a weakest precondition or a strongest postcondition calculus are used to statically determine whether a program code conforms to its annotations. The extended static checker ESC/Java2 [18] is a tool that attempts to verify annotated Java programs following this approach (Section 2.1).

Empirical evidence shows that automated sanity checking of annotations is desirable [6]. In particular, Leavens et al. [22] propose as one of the challenges for software verification the following: "Provide assistance in specifying libraries of classes." In this article we focus on a particular sanity check — code reachability. Code is unreachable if it is not executed for any possible input. Unreachable code, also known as *dead code*, is often a bug. For example, the Java compiler tries to prevent bugs by disallowing code following a **return** statement.

```
/*@ requires x > 10;
  @ ensures
  @   \result == 1;*/
int withPre(int x) {
 if (x < 10) {
   // not checked
   return 2;
 }
 return 1;
}
```

```
/*@ requires i >= 10;
  @ ensures
  @   \result == i;
  @ ensures
  @   \result < 10;
  @ modifies
  @    \nothing;*/
int libraryFunc (int i);

int useLibraryFunc() {
 int r = libraryFunc(11);
 return 1/0;
}
```

Figure 1: Examples of code that is unreachable once the annotations are taken into account.

Annotations provide extra information about the program, so the notion of code unreachability needs to be extended. Consider the examples in Figure 1. The precondition of the method withPre, expressed by the requires clause, restricts the value of the parameter x to be greater than 10 and the postcondition, expressed by the ensures clause, restricts the return value to be always 1. The **return** statement in the "then" branch of the **if** statement appears to be violating the postcondition. Nevertheless, because of the precondition, this code conforms to its annotation and a static checker like ESC/Java2 will not produce a warning. The fact that a method contains code that is unreachable from the point of the specification is likely to be a bug, either in the specification or in the program code.

The method libraryFunc illustrates a method for which we do not have an implementation (for example because the implementation is proprietary) and we need to rely on its specification. In ESC/Java2 all the methods in the standard Java API are treated in this way. Unfortunately, the specification is inconsistent as it requires the return value to be at least 10 and at the same time to be less than 10. The repercussions of this inconsistency are demonstrated by the useLibraryFunc. The **return** statement in this method seems wrong and yet the extended static checker does not give a warning. The reason for this behavior of the checker

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM ISBN 978-1-59593-721-6/07/0009 ...$5.00.

is less obvious than in the previous example and we will explain it in more detail later. Intuitively, as the specification of libraryFunc is inconsistent, from the point of view of the checker the call to that function never terminates and therefore the checker 'believes' that the **return** statement is never executed.

Hence, the problem we address in this article is how to detect unreachable code in the presence of annotations and how we can benefit from such analysis in extended static checking. More specifically, the contributions of the article are as follows: (1) we introduce the notion unreachability for annotated code, (2) we identify several types of unreachable code categorized by their root cause, (3) we present an efficient algorithm for detecting unreachable code, (4) we present an evaluation of the analysis on an existing code base, and (5) an implementation, which is part of ESC/Java2[1].

2. BACKGROUND

Programmers reduce development time dramatically by reusing components that are well documented [20]. In the Java world this is achieved by using `javadoc`, which supports a form of structured documentation [15, 19]. The Java Modeling Language [21] (JML) was designed to allow more formal documentation. Tools can statically check if code and JML-annotations agree. When static checking fails (for example because the code is too complex), the annotations can be compiled into runtime checks. Moreover, unit tests can be generated automatically [5].

The leading static checker for JML-annotated Java is ESC/Java2. Spec$^{\#}$ has a similar architecture and works for annotated C$^{\#}$ programs [2].

2.1 ESC/Java2 Architecture

JML annotations are embedded in Java code as a special form of comments. They are used to specify the behavior of classes and methods in terms of preconditions, postconditions, invariants, and other higher-level constructs. ESC/Java2 checks if code and annotations agree and if there are no runtime exceptions. Methods are checked one at a time, ignoring other methods' implementation and relying on their specification.

For a given JML-annotated method, ESC/Java2 generates a formula, called a *verification condition* (VC), using a strongest postcondition calculus. Further, it tries to prove the verification condition by using an automated theorem prover. If the VC is not proven valid, the checker produces warnings derived from the counterexamples provided by the prover. These warnings describe how the program may violate its JML specification, or in what way the specified program might cause runtime exceptions (such as NullPointerException).

ESC/Java2 performs the translation of JML-annotated Java code to a VC in several stages. This process is schematically depicted in Figure 2.

Given a JML-annotated Java program, the front-end produces an *abstract syntax tree* (AST), which is translated into an intermediate representation called *guarded commands* (GC) [25]; this representation captures both the Java code and its JML annotation. The components that infer invariants [17, 13] work on this representation. Subsequently, loops are translated into structurally simpler commands by a process called *loop desugaring*. ESC/Java2 supports two modes of loop desugaring: One mode is called *loop unrolling* and does not require loop invariants, but it is unsound (see Section 4.3 for more details); the other mode is called *safe desugaring* and treats loops in accord with Hoare logic [16], but requires loop invariants.

[1] `http://kindsoftware.com/products/opensource/ESCJava2/cvs.html`

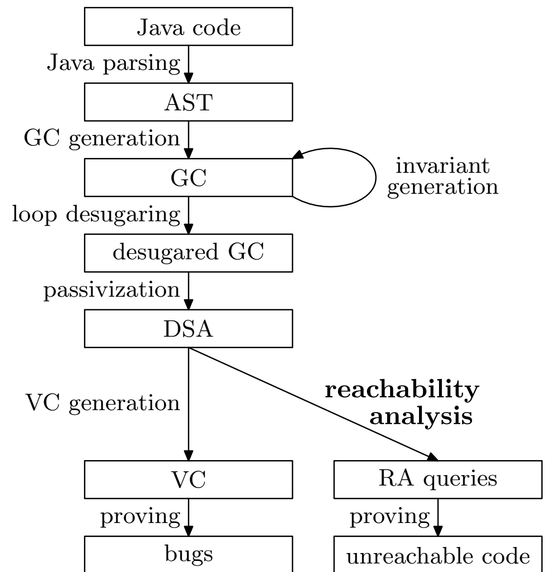

Figure 2: ESC/Java2 architecture

There is an obvious tradeoff between loop unrolling and safe desugaring. The loop unrolling mode may miss some errors as it does not reason about all possible execution traces of the program. The safe loop desugaring does not suffer from this deficiency but it leads to spurious warnings if a strong-enough loop invariant is not provided. Loop invariant generation techniques are used to infer invariants automatically and hence alleviate the annotation burden imposed on the user [10, 13, 23, 17]. Nevertheless, these techniques are computationally expensive and they do not always succeed in finding the proper invariant. In ESC/Java2 loop unrolling is the default loop desugaring mode, since the alternative is not yet practical.

After loop desugaring, the desugared GC is translated into an assignment-free form, or *passive form*, called *dynamic single assignment* (DSA). This is done by ensuring that each variable is assigned-to at most once, which often requires additional variables, and by replacing assignments with assumptions. The main purpose of this particular transformation is to avoid the exponential explosion in the size of the generated VC (see [14] for details).

After the DSA form is generated, the VC is generated from it and sent to a theorem prover. Finally, the output of the prover is processed to provide feedback to the user.

The process described above represents the skeleton of ESC/Java2 and additional analyses can be 'hooked' in this architecture to facilitate the application of the tool. This is the case of the reachability analysis presented in this paper, which is applied on the DSA representation and uses a theorem prover. Since this particular analysis slows down the