Reachability Analysis for Annotated Code

Mikoláš Janota[1], Radu Grigore[1], and Michał Moskal[2]

[1] School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
mikolas.janota@ucd.ie

[2] Institute of Computer Science
University of Wrocław
ul. Joliot-Curie 15
50-383 Wrocław, Poland
michal.moskal@ii.uni.wroc.pl

Abstract. We devised a reachability analysis that exploits code annotations and implemented it as a component of the extended static checker ESC/Java2. The component reports unchecked code and a class of errors previously undetected. We applied the analysis to existing annotated code and uncovered errors that were unknown to the developers. We present the algorithm performing the analysis and discuss errors that it detects.

1 Introduction

Program annotations are logic specifications embedded in the actual program code [5]. They help programmers to implement the intended functionality. Variants of weakest precondition or strongest postcondition calculi are used to statically determine whether a program code conforms to its annotations. The extended static checker ESC/Java2 [6] is a tool that attempts to verify annotated Java programs following this approach.

Writing specifications, however, is a difficult task, and empirical evidence shows that automated sanity checking of annotations is desirable for successful application of the approach [1]. We focus here on a particular sanity check: code reachability. Code is unreachable if it is not executed for any possible input; unreachable code is often a bug. In Java, certain cases of unreachable code, such as commands following a **return** statement, are disallowed by the compiler. Traditionally, unreachable code is detected by data flow analysis [8].

In the presence of annotations, the notion of code unreachability needs to be extended as the annotations restrict the possible input. Consider the examples in the Fig. 1. Both examples contain code that causes a runtime exception when executed and we would like the static analysis to warn the user about it. In both cases, however, ESC/Java2 does not warn about the problem. In the first example, it is because of the discrepancy between the method's precondition and

```
//@ requires x > 10;
static int withPre(int x) {
  if (x < 10) {
    return 1/0; // not checked
  }
  return 1;
}
```

```
static int loopUnrolling () {
  int i = 0;
  while (i < 10) i++;
  return 1/0; // not checked
}
```

Fig. 1. Examples of problems addressed by the analysis.

the condition of the **if** statement. In the second example, it is not obvious why the checker should miss the error. By default, ESC/Java2 does not reason about any code following such loop, which is due to the imprecise modeling of the code (see Sect. 4.1 for details).

In both cases our analysis reports a warning, identifying the commands that are not checked.

The contributions of the article are: (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, available online[3].

2 Background

The Java Modeling Language [7] (JML) is an annotation language for Java programs embedded in program code as a special form of comments. ESC/Java2 is an extended static checker for JML-annotated Java code that attempts to verify that a given program conforms to its annotations. For a given program, ESC/-Java2 generates a formula, called *verification condition* (VC), using a strongest postcondition calculus. The checker tries to prove the verification condition by querying an automated theorem prover. If the VC is not proven valid, warnings are provided to the user. These warnings describe in what way the specified program might cause run-time exceptions (such as **null**-pointer dereferencing) or how the program may violate its JML specification.

The checker translates JML-annotated Java code to a VC in several stages. One these stages uses an intermediate representation called *dynamic single assignment* (DSA). A DSA program is used to express the semantics of both the Java code and the JML specification. In DSA, each variable is assigned-to at most once, which enables replacing assignments by assumptions (see [4] for details).

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

C	$\mathrm{N}(P, C)$	$\mathrm{W}(P, C)$
skip	P	false
assume f	$f \wedge P$	false
assert f	$f \wedge P$	$P \wedge \neg f$
$C_1 \| C_2$	$\mathrm{N}(P, C_1) \vee \mathrm{N}(P, C_2)$	$\mathrm{W}(P, C_1) \vee \mathrm{W}(P, C_2)$
$C_1; C_2$	$\mathrm{N}(\mathrm{N}(P, C_1), C_2)$	$\mathrm{W}(P, C_1) \vee \mathrm{W}(\mathrm{N}(P, C_1), C_2)$

Fig. 2. Strongest postcondition transformers.

In the rest of the paper we will assume a first-order logic language for formulas and a theory T for the context of validity, we will write $T \models f$ to denote that f is valid in the context of the theory T. The theory T expresses the *background predicate*, an axiomatization of the execution semantics of Java programs.

We use e and f to denote logic formulas, possibly with free variables. In the following, by DSA we will understand the language defined by the following grammar:

$$cmd := \textbf{skip} \mid \textbf{assume}\, f \mid \textbf{assert}\, f \mid cmd \| cmd \mid cmd; cmd$$

Additionally, we will use the following shorthands:

$$\textbf{if}\, C\, \textbf{then}\, B_1\, \textbf{else}\, B_2 \equiv ((\textbf{assume}\, C; B_1) \| (\textbf{assume}\, \neg C; B_2))$$
$$\textbf{if}\, C\, \textbf{then}\, B \equiv \textbf{if}\, C\, \textbf{then}\, B\, \textbf{else skip}$$

Informally, the purpose of the **assume** f command is that once the execution reaches this command f can be assumed; if the execution trace reaches this command and f does not hold, that execution trace blocks. The purpose of the **assert** f command is that once the execution reaches this command f must be checked; if it is invalid then an error occurs. The command $C_1 \| C_2$ represents nondeterministic choice between the two commands and the command $C_1; C_2$ represents a sequence.

To formally define the semantics of DSA, we introduce two[4] strongest postcondition predicate transformers N and W, where N propagates the *normal behavior* and W the *wrong behavior*, whose semantics are captured by the following definition.

Definition 1 *1. For predicate transformers N and W defined as in Fig. 2 and for a precondition P and a command C, we say that C terminates* normally *for P if and only if the predicate $\mathrm{N}(P, C)$ holds and it terminates* wrongly *for P (or* goes wrong *for P) if and only if $\mathrm{W}(P, C)$ holds.*
2. The verification condition *for a program C is $\neg \mathrm{W}(true, C)$.*

Therefore the verification condition expresses that the program does not go wrong for any possible input.

[4] The implementation additionally contains a third predicate for *exceptional behavior*.

An important property of this calculus is that a command with an unsatisfiable precondition does not go wrong. If a precondition is unsatisfiable, an analysis relying on a strongest postcondition calculus does not provide any useful information to the user. Moreover, such a scenario is most likely unintentional.

Observation 2 *If* $T \models \neg P$ *then* $T \models \neg W(P, C)$*, for all formulas* P *and all commands* C*.*

3 Definition of Unreachability

Informally, a command is unreachable if none of the execution traces leading to it have satisfiable normal behavior. An unreachable assertion is never checked.

To express this idea formally, this section defines the notion of unreachability in the context of the normal behavior predicate transformer N and an acyclic control flow graph. Let $\mathcal{C}$ be a set of commands.

Definition 3 *A* control-flow graph *is a tuple* $\langle V, E, I, O, \mathcal{L}\rangle$*, where* V *is the set of nodes,* $E \subseteq V \times V$ *is the set of edges,* $I \subseteq V$ *the set of* entry nodes *and* $O \subseteq V$ *is the set of* exit nodes*. Nodes are mapped to commands by* $\mathcal{L} : V \to C$*. Additionally, we require that entry nodes do not have parents, exit nodes do not have children, the graph is acyclic, and the set of nodes is finite.*

Definition 4 *For a control flow graph* $G \equiv \langle V, E, I, O, \mathcal{L}\rangle$*, we define parents and precondition of a node:*

$$\text{parents}_G(n) \equiv \{p \in V \bullet \langle p,\, n\rangle \in E\}$$

$$\text{pre}_G(n) \equiv \begin{cases} \text{true} & \text{if } n \in I \\ \bigvee_{p \in \text{parents}_G(n)} N(\text{pre}_G(p), \mathcal{L}(p)) & \text{otherwise} \end{cases}$$

Definition 5 *Node* n *is* semantically unreachable *in a control flow graph* G *if and only if* $T \models \neg\, \text{pre}_G(n)$*.*

Whenever we use the term 'unreachable' (and 'reachable') we refer to semantic unreachability as defined above, not to the graph-theoretic notion.

The DSA maps to a subclass of control-flow graphs, called serial parallel graphs.

1. if C is one of **skip**, **assume** f or **assert** f, then it maps to $\langle\{n\}, \{\}, \{n\}, \{n\}, [n \mapsto C]\rangle$, where n is a fresh node
2. if C_1 maps to $\langle V_1, E_1, I_1, O_1, \mathcal{L}_1\rangle$ and C_2 maps to $\langle V_2, E_2, I_2, O_2, \mathcal{L}_2\rangle$ then
   (a) $C_1; C_2$ maps to $\langle V_1 \cup V_2, E_1 \cup E_2 \cup (O_1 \times I_2), I_1, O_2, \mathcal{L}_1 \cup \mathcal{L}_2\rangle$
   (b) and $C_1 || C_2$ maps to $\langle V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2, \mathcal{L}_1 \cup \mathcal{L}_2\rangle$

To see how the DSA calculus defined in the Sect. 2 relates to semantic unreachability, consider a DSA command C, a graph G obtained from it, and a node n of G. Let C' be a command obtained by replacing the sub-command corresponding to n with **assert** false and all sub-commands **assert** f of C with **assume** f. A node n is semantically unreachable in G if and only if $T \models \neg \text{W}(true, C')$. This captures the intuitive meaning of an unchecked assertion.

4 Scenarios of Unreachable Code

4.1 Loop Unrolling

Loops are one of the toughest nuts to crack when reasoning about programs. In extended static checking, to be able to reason about programs that do not contain loop invariants we use a technique called *loop unrolling*. This technique is parameterized by a constant L and reasons only about the scenarios when a given loop terminates in $0, 1, \ldots, L$ iterations. By following this approach we can miss some errors.

The following schematically describes the result of an unrolling with $L = 2$:

```
while (C) {          ⇒  if C then B;
  B                     if C then B;
}                       if C then assume false
```

Execution traces that do not terminate in L iterations are modeled as blocking in the loop by the **assume** false command.

Loop unrolling contains a significant pitfall. If for all possible inputs the analyzed loop does not terminate within L iterations, the checker does not reason about the code following the loop.

Consider the following translation of a Java code to its DSA representation:

```
int i = 0;                 C1 : if 0 < 10 then assume i1 = 0 + 1;
while (i < 10)             C2 : if i1 < 10 then assume i2 = i1 + 1;
    i++;                ⇒  C3 : if i2 < 10 then assume false;
return 1/0;                C4 : assert 0 ≠ 0;
                           C5 : assume RES = 1/0
```

If we compute the normal behavior of the first three commands, we observe that $T \models \neg\mathrm{N}(C_1; C_2; C_3, true)$. From Lemma 2, $T \models \neg\mathrm{W}(\mathrm{N}(C_1; C_2; C_3, true), C_4; C_5)$. Therefore, the assertion C_5 cannot cause the program to go wrong since from the point of the checker that assertion is unreachable.

The analysis presented in this article detects that the code following the loop is not checked, hence, once the user is informed about it, he or she can adjust the appropriate parameter of the checker to unroll the problematic loop more times.

4.2 Mistakes of the User

The previous section discussed code unreachable due to the inaccurate modeling of Java by the checker. Unreachable code, however, can also directly stem from inconsistencies in user's code and specifications.

We present four kinds of unreachable code in the Fig. 3. The unreachableCode method contains unreachable code in the classic sense: the division by zero is not checked. It is most likely a bug in the user code. More subtle problems arise when we take into account annotations as well, as in the withPre method, from Fig. 1. An extreme example of an inconsistency in specifications is the method badSpec

```
int unreachableCode(int x) {
  if (x > 10)
    if (x < 5)
      return 1/0;
  return 0;
}

//@ modifies a, b;
void modAB() { ... }

//@ modifies a;
int modA() {
  modAB();
  return 1/0;
}
```

```
//@ requires x > 0;
//@ requires x < 0;
int badSpec(int x, int y) {
   return 1/0;
}

//@ requires i >= 10;
//@ ensures \result == i;
//@ ensures \result < 10;
void libraryFunc(int i);

void useLibraryFunc() {
   libraryFunc(11);
   int x = 1/0;
}
```

Fig. 3. Examples of different types of unreachable code.

which has unsatisfiable precondition. Such methods always pass the analysis by the checker.

A common case of unreachable code is related to the `modifies` clause. Consider the methods modA, which promises to modify only a, and modAB, which can also modify b. Therefore, modAB should not be called from modA. This causes the method modA to go wrong and therefore ESC/Java2 warns about it. Often, however, this causes the rest of the assertions to be unreachable. This scenario is a specific instance of a general issue where an unsatisfiable asserted expression generates one warning and hides other warnings.

Another common source of inconsistencies are specifications of methods for whom the implementation is not available. This is illustrated by the methods libraryFunc and useLibraryFunc in Figure 3. As ESC/Java2 checks the code with respect to the specification and there is no code for the method libraryFunc, the checker does not detect that its postcondition is unsatisfiable. Once a method with unsatisfiable postcondition is used in code, everything following that method is not checked. The explanation for this behavior is that a call to a method is translated into an assertion checking the method's preconditions and an assumption establishing the method's postcondition. See [1] for a detailed discussion on such specifications.

5 The Algorithm

We are given a directed acyclic flow graph in which we want to detect semantically unreachable nodes. An efficient algorithm is needed to make the analysis usable in practice. For that we need to (1) compute small prover queries, and

(2) call the prover only a few times. Experimental data shows that the response time of the automated theorem prover (Simplify) sharply increases when the size of the query exceeds a certain limit; this motivates (1). A prover call is on average hundreds times slower than any reasonable manipulation of the flow graph; this motivates (2).

The precondition definition directly maps to a recursive (memoized) algorithm that creates a directed acyclic graph (DAG) with $n-1$ nodes for $\vee$ and m nodes for $\wedge$, where n is the number of nodes and m is the number of edges in the flow graph. Unfolding naively the DAG into a tree to send it to a prover can yield queries with exponential size. A simple way to obtain preconditions that produce queries with linear size is to introduce an auxiliary variable for each precondition, and then use it to express subsequent preconditions. But introducing auxiliaries comes at a cost. We can minimize the size of the formula by introducing auxiliaries only for subformulas of size S when they appear in P places and $PS - P - S \geq 2$. This transformation reduces the size of the queries dramatically: On our benchmarks it reduced by 90% the number of queries that are too big for the prover to process. Eliminating sharing exploits the series–parallel structure of the flow graph and the queries are roughly the same size as the normal behavior computed directly on the DSA as in [4].

The auxiliary variables can be defined using equivalence.

$$\big(a \Leftrightarrow f(b)\big) \wedge g(a,b) \tag{1}$$

Here b is a set of variables, a is the auxiliary variable, $f(b)$ is its definition, and $g(f(b),b)$ is the original formula. Now consider the alternative:

$$\big(a \implies f(b)\big) \wedge g(a,b) \tag{2}$$

It can be shown that (1) is satisfiable if and only if (2) is satisfiable, provided that g is monotonic in a, that is $g(\text{false},b) \implies g(\text{true},b)$. We can make sure that that is the case by eliminating sharing only below the operators $\wedge$ and $\vee$. In practice, replacing (1) by (2) reduces the proving time to two thirds.

We say that the nodes of the flow graph that can be tracked back to Java code are *interesting*. There are very few of them, less than 20 in most cases. Processing them takes negligible time, which is why later we shall concentrate on minimizing the number of prover queries. We contract the graph by keeping only the interesting nodes; we have an edge (u,v) in the contracted graph if in the original one there was a path from u to v with no other interesting node. This can be done in $O(mn)$ time with a slight modification of a DFS-based solution to the transitive closure problem. The contracted graph has a unique inital node denoted by i.

The key observation that allows us to have fewer prover calls than interesting nodes is that the information about node reachability can be propagated in the flow graph according to these rules: (1) we can infer that u is unreachable if all paths from i to u contain an unreachable node, and (2) we can infer that u is

reachable if it dominates a reachable node v, that is, if all paths from i to v that do not contain unreachable nodes go through u. These rules are expressed in terms of paths, implying that we can use the propagation algorithm (Fig. 4) on the original graph as well as on the contracted graph.

```
PROPAGATE-UNREACHABLE(u)
    label u as unreachable
    for each child v of u such that v has only unreachable parents
        do PROPAGATE-UNREACHABLE(v)

PROPAGATE-REACHABLE(u)
    label u as reachable
    if u has an immediate dominator d call PROPAGATE-REACHABLE(d)
```

Fig. 4. Propagation of the reachability information.

We compute dominators ignoring nodes already marked as unreachable using the simple algorithm of Cooper [2], which works in $O(mn)$ time for DAGs. The critical part that makes our implementation fast in practice is the heuristic used to decide for which node we query the prover.

```
REACHABILITY-ANALYSIS()
    while there are unlabeled nodes
        do choose an unlabeled node u that has
              a maximal number of unlabeled dominators
           if the prover says that the precondition of u is reachable
              then PROPAGATE-REACHABLE(u)
              else use binary search with prover queries to identify
                      the farthest unreachable dominator d of u
                   PROPAGATE-UNREACHABLE(d)
                   RECOMPUTE-DOMINATORS
                   if d has an immediate dominator d'
                      then PROPAGATE-REACHABLE(d')
```

Fig. 5. The algorithm used to implement the analysis.

In the case that all nodes are reachable and interesting the greedy algorithm (Fig. 5) is optimal, because the prover must be called for all the leafs of the (immediate) dominator tree. In practice the performance is good: The time nec-

essary to verify one package of the ESC/Java2 frontend on a computer with 3GHz Pentium CPU is 934 seconds, out of which 47% is spent in the reachability analysis, out of which 99.5% is spent in the prover. Therefore ESC/Java2 does not become significantly slower because of the reachability analysis. The total number of leafs in the dominator trees is 179 and the number of prover calls is 191. The number of methods was 111. The average number of nodes in the flow graph is 270 and in the contracted flow graph it is 6.

6 Case Study

We have tested the analysis on the ESC/Java2 frontend, the `javafe` package. The package contains 217 classes.

We have found 5 inconsistencies in the specifications of the JDK, which are not reported without the reachability analysis. The ESC/Java2 repository contains handcrafted tests to detect this type of problems. These tests, however, did not uncover these problems because they were not exhaustive. It should be noted that fixing these problems involved a tedious process of narrowing down the set of inconsistent annotations. This effort, however, was justified by the wide usage of these specifications.

An incorrect use of the `modifies` clause (as in Fig. 3) hiding the rest of the potential warnings appeared 9 times. Other warning hiding appeared 7 times. The case of unreachable code resulting from loop unrolling, as discussed in the Sect. 4.1 appeared 4 times. In 10 cases the informal comments indicated that the author was aware that the code is unreachable. Code of this type can be marked with the `unreachable` pragma and then the analysis does not warn about it. We detected only one case of unreachable code in the classical sense.

In several cases the unreachable was due to the unsound modeling of the `modifies \everything;` pragma. This pragma is the default annotation if no `modifies` clause is provided. Whenever a method with the `modifies \everything;` annotation is called, ESC/Java2 does not consider the potential state change. Therefore, the code that we have found is actually executed. Nevertheless, ESC/Java2 does not check that code, thus the warnings provided by the analysis were not spurious.

In the remaining 12 cases we were not able precisely identify the source of the problem. Nevertheless, we suspect that the source lies in inconsistent specifications of classes inside the `javafe` package. Such inconsistencies are very hard to pinpoint as they involve object invariants in a class hierarchy.

7 Conclusion and Future Work

We devised the theoretical underpinnings of reachability analysis for annotated code, implemented it efficiently, and classified the bugs that it helps to find. We intend to adapt it for BoogiePL [3], whose flow graphs are not necessarily series–parallel.

We pose two open problems related to this analysis.

Provide better warnings. As the case study shows, although our analysis uncovers real bugs, they are often hard to track down. The warning message should also pinpoint the likely locations causing code to be unreachable, not only to the location of the unreachable code. Even better, the warning should also classify the problem, for example by saying that it is a 'loop unrolling' problem if that is the case.

Optimize VCs and prover queries. The reachability analysis suggests that one VC per method might not be optimal, for example because it includes all the unreachable code. In general, what is an optimal strategy for querying the prover for the correctness of a method, given its flow graph?

References

1. Patrice Chalin. Early detection of JML specification errors using ESC/Java2. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*. ACM Press, November 2006.
2. Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm, 2001. Available online at `www.cs.rice.edu/~keith/EMBED/dom.pdf`.
3. Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.
4. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.
5. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
6. Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, January 2005.
7. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.
8. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer–Verlag, 1999.