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.

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.

```
/*@ 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 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.

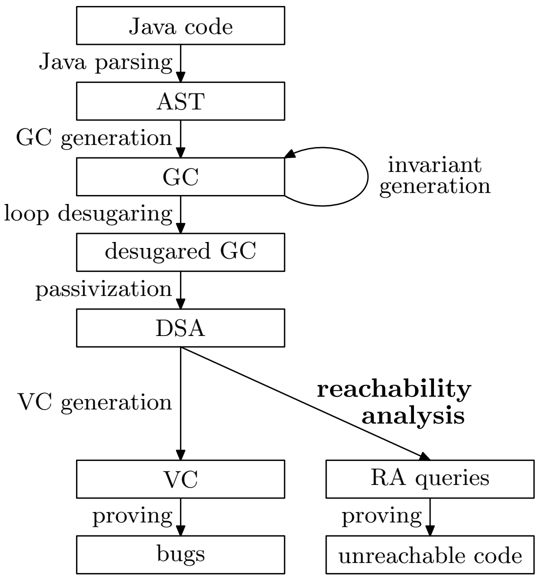

Figure 2: ESC/Java2 architecture

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

[1]`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	$f \wedge P$	$P \wedge \neg f$
$C_1 [] C_2$	$\mathrm{N}(P, C_1) \vee \mathrm{N}(P, C_2)$	$\mathrm{W}(P, C_1) \vee \mathrm{W}(P, C_2)$
$C_1; C_2$	$\mathrm{N}(\mathrm{N}(P, C_1), C_2)$	$\mathrm{W}(P, C_1) \vee \mathrm{W}(\mathrm{N}(P, C_1), C_2)$

Figure 3: Strongest postcondition transformers.

checker, it is disabled by default and can be enabled by the switch `-era`.

2.2 VC Generation from DSA

As we explained in the previous section, DSA is the input of the reachability analysis and thus it deserves special attention. Hence, in this section we formally define the DSA language and how a VC is obtained from a DSA program.

Before we proceed, we make several assumptions. In the rest of the paper we assume a first-order logic language for formulas and a theory T for the context of validity. We write $T \models f$ to denote that f is valid in the context of the theory T. The theory T expresses the *background predicate*, a (partial) axiomatization of the Java semantics.

We use f to denote a predicate represented as a logic formula where the free variables correspond to the predicate's arguments. In the following, by DSA we understand the language defined by the following grammar:

$$\mathit{cmd} := \textbf{skip} \mid \textbf{assume}\, f \mid \textbf{assert}\, f \mid \mathit{cmd} [] \mathit{cmd} \mid \mathit{cmd}; \mathit{cmd}$$

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 an 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 is checked and if it is invalid, an error occurs. The command $C_1 [] C_2$ represents a 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 strongest postcondition predicate transformers — N and W. The predicate N propagates the *normal behavior* and the predicate W propagates the *wrong behavior*. Their semantics are captured by the following definition.

Definition 1. For the predicate transformers N and W defined as in Figure 3, we define the following:

1. For a precondition P and a command C, we say that C *goes wrong* if and only if $\mathrm{W}(P, C)$ is satisfiable, i.e.,
$$T \not\models \neg \mathrm{W}(P, C)$$
2. The *verification condition* for a program C is the following formula:
$$\neg \mathrm{W}(\mathit{true}, C)$$
3. The program C *conforms to its specification* if and only if its verification condition is valid:
$$T \models \neg \mathrm{W}(\mathit{true}, C)$$

Intuitively, the verification conditions expresses that no possible execution breaks any of the assertions.

An important property of this semantics is that a command with an unsatisfiable precondition does not go wrong.

Observation 1. *If $T \models \neg P$, then $T \models \neg W(P, C)$, for all predicates P and all commands C.*

This observation is not surprising since an unsatisfiable precondition states that the command in question should never be run according to its specification. What is less obvious is that this fact also comes into effect for a subcommand in a sequence of commands. For example, consider the sequence $C_1; (C_2; C_3)$. We can say that the postcondition of C_1 is a precondition of $C_2; C_3$. In particular, if $\mathrm{N}(\mathit{true}, C_1)$ is unsatisfiable, the whole sequence cannot go wrong because of C_2 or C_3. In other words, C_2 and C_3 are not checked. In such situations, an analysis relying on a strongest postcondition calculus does not provide any useful information about these subcommands. Moreover, such a scenario is most likely unintentional.

3. DEFINITION OF UNREACHABILITY

Informally, a command is unreachable if all the execution traces leading to it have an unsatisfiable normal behavior. To express this idea formally, this section defines the notion of unreachability in the context of the normal behavior (the predicate transformer N) and an acyclic control flow graph. Let $\mathcal{C}$ denote the subset of the DSA language consisting of the commands **skip**, **assume** f, and **assert** f.

Definition 2. 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, $I \subseteq V$ is the set of *entry nodes* and $O \subseteq V$ is the set of *exit nodes*. Nodes are labeled with commands by the function $\mathcal{L} : V \to \mathcal{C}$. 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.

The DSA maps to a subclass of control flow graphs, called *series–parallel* graphs [27], constructed as follows.

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
   $$\langle V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2, \mathcal{L}_1 \cup \mathcal{L}_2\rangle$$

Once we have the control flow graph, the definition of unreachability is straight forward.

Definition 3. We define the *parents* and the *precondition* of a node in a control flow graph $G \equiv \langle V, E, I, O, \mathcal{L} \rangle$ as follows:

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

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

Definition 4. 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.

3.1 How Unreachability Corresponds to DSA

To better understand the definition of semantic unreachability (Definition 4), we explore how the DSA command corresponds to its control flow graph. First observe that the DSA semantics (see Figure 3) has the following properties, where $\mathcal{B}$ is either N or W.

$$\mathcal{B}(P, (C_1 \,[\!]\, C_2); D) = \mathcal{B}(P, (C_1; D) \,[\!]\, (C_2; D)) \qquad (1)$$

$$\mathcal{B}(P, D; (C_1 \,[\!]\, C_2)) = \mathcal{B}(P, (D; C_1) \,[\!]\, (D; C_2)) \qquad (2)$$

$$\mathcal{B}(P, C_1 \,[\!]\, (C_2 \,[\!]\, C_3)) = \mathcal{B}(P, (C_1 \,[\!]\, C_2) \,[\!]\, C_3) \qquad (3)$$

$$\mathcal{B}(P, C_1; (C_2; C_3)) = \mathcal{B}(P, (C_1; C_2); C_3) \qquad (4)$$

By applying these equalities, any command can be rewritten into the choice between all its possible *execution traces*:

$$(C_1^1; C_2^1; \ldots; C_{l_1}^1) \,[\!]\, \ldots \,[\!](C_1^m; C_2^m; \ldots; C_{l_m}^m)$$

where each C_i^j is neither the choice nor the sequence, while preserving the behaviors' semantics. The paths in the graph obtained from the process described above correspond to these execution traces. For both the normal and the wrong behavior, each behavior is a disjunct of the behaviors of these traces (see Figure 3); in particular the whole command goes wrong if and only if at least one of its traces goes wrong.

Let us consider a node n labeled with the command C. Then each of the paths going through n correspond to an execution trace of the form $C_{\text{pre}}; C; C_{\text{post}}$, where C_{pre} is a *prefix* of the pertaining execution trace. Let Pre be the set of all these prefixes, then the function $\text{pre}_G(n)$ (see Definition 3) returns the disjunct of normal behaviors of the prefixes, i.e., $\bigvee_{C_\text{p} \in \text{Pre}} \mathrm{N}(\mathit{true}, C_\text{p})$.

Hence, Definition 4 captures our intuition that a command is unreachable if all the paths leading to it have an unsatisfiable normal behavior. In particular, if C is an assertion, the whole program cannot go wrong because of that assertion if it is unreachable as none of the traces leading to C can go wrong because of C (see Observation 1).

4. SCENARIOS OF UNREACHABLE CODE

In this section we discuss typical scenarios that result in unreachable code. In Section 6 we present how often these scenarios appear in practice. We start by showing several typical cases of discrepancies in the code or specifications. The last two subsections discuss unreachable code in the presence of loops.

4.1 Incoherence of Specification and Code

We present three kinds of unreachable code in Figure 4. 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 Figure 1. An extreme example of an inconsistency in specifications is the method badSpec which has an unsatisfiable precondition. Such methods always pass extended static checking without reachability analysis.

A common case of unreachable code is related to the use of JML's `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. ESC/Java2 models this by inserting an **assert** *false* before the call to modAB. 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. (Assertions that are merely invalid but satisfiable do not hide other checks.)

In the introduction we have already mentioned that specifications of methods for which an implementation is not available are a common source of inconsistencies. Consider again the methods libraryFunc and useLibraryFunc in Figure 1. The body of the method useLibraryFunc is translated to DSA as follows:

C_1: **assert** $11 \geq 10$;
C_2: **assume** $r_1 = 11 \wedge r_1 < 10$;
C_3: **assert** $0 \neq 0$;
C_4: **assume** $RES = 1/0$

Here, the command C_1 represents the check for the precondition of libraryFunc and C_2 represents its postcondition. This is a general approach of translating method calls, preconditions are translated as asserts and postconditions as assumes. If the called method can modify the program state, the variables whose values may change need to be reset. The technique for reseting values of variables is called havocking and we will briefly describe it in Section 4.2. Nevertheless, in this particular case the modifies \nothing; clause in the specification of libraryFunc guarantees that it does not modify anything. The command C_3 checks that the division by 0 will not occur and the command C_4 stores the result of the division in a special variable RES modeling the method's return value. Apparently, the normal behavior $(11 \geq 10) \wedge (r_1 = 11 \wedge r_1 < 10)$ of $C_1; C_2$ is unsatisfiable. Recalling Definition 4, the commands C_3 and C_4 are unreachable. Hence, the method uselibraryFunc cannot go wrong because of the assertion C_3, i.e., it is not checked.

Because ESC/Java2 is a modular checker, method calls are always checked with respect to the specification of the called method and its implementation is ignored. This means that the situation described above would occur even if we did have an implementation for the method libraryFunc. If we had the implementation, however, we would uncover that the postcondition is not satisfiable during the check of that implementation (as an unsatisfiable postcondition cannot be established). See [6] for a detailed discussion about the pitfalls of specifications without implementation.

4.2 Safe Loop Desugaring

As we have described in Section 2, ESC/Java2 supports two modes of loop desugaring. In this section we discuss

```
//@ requires x > 0;                    //@ modifies a, b;
//@ requires x < 0;                    void modAB() { ... }
int badSpec(int x, int y) {
   return 1/0;
}

int unreachableCode(int x) {           //@ modifies a;
  if (x > 10)                          int modA() {
    if (x < 5)                           modAB();
      return 1/0;                        return 1/0;
  return 0;                            }
}
```

Figure 4: Examples of different types of unreachable code.

what information the reachability analysis provides in the safe desugaring mode. Please recall that the loop safe mechanism relies on loop invariants.

Since the reachability analysis does not depend on the way loops are desugared, we do not provide full account of this process. We illustrate the behavior on examples instead. Consider the following method where the user provided the invariant $i \geq 5$:

```
//@ requires i >= 5;
void infiniteLoop (int i) {
  //@ loop_invariant i >= 5;
  while (i >= 0) i = 5;
  //@ assert false;
}
```

In the loop safe mode this method is desugared as follows:

```
$C_1$:  assume $i_0 \geq 5$;
$C_2$:  assert $i_0 \geq 5$;
$C_3$:  assume $i_1 = i'$;
$C_4$:  assume $i_1 \geq 5$;
$C_5$:  ((assume $i_1 \geq 0$; assume $i_2 = 5$;
        assert $i_2 \geq 5$; assume $false$)
        $[\!]$(assume $\neg(i_1 \geq 0)$));
$C_6$:  assert $false$
```

The command C_1 represents the precondition and C_2 checks for the validity of the loop invariant before the loop. The command C_3 resets the value of i to a fresh value. This is called *havocking* and it models the fact that for an arbitrary iteration we do not know anything about the variables modified in the loop body except for what is in the loop invariant. In other words, havocking discards the information about these variables that was available before the loop. The command C_4 corresponds to the fact that the loop invariant holds before any iteration (knowing that it was established before the loop). The command C_5 represents a choice between termination of the loop and the loop body. More precisely, the left branch of the choice command models an arbitrary iteration of the loop, checks the loop invariant after the iteration, and blocks. The right branch of the choice command conditions further execution by the negation of the loop's guard.

Now we observe that the conjunct of the invariant and the negation of the guard $(i \geq 5) \wedge \neg(i \geq 0)$ is unsatisfiable. Therefore, the reachability analysis detects that the assertion at the end is unreachable.

The example above illustrates that the reachability analysis discovers that a loop does not terminate but only if the loop invariant is strong enough. Thus, it would be beneficial to combine the reachability analysis with techniques for loop invariant generation [10, 17, 23]. For example, consider the following excerpt of code:

```
int sum = 0;
for (int i = 0; i < 10; j++) sum += i;
//@ assert false;
```

The loop above does not terminate. If a technique for loop invariant inference is used, the user is likely to expect that the invariant $0 \leq i \wedge i \leq 10$ will be automatically inferred. Instead, however, the invariant $i = 0$ is inferred and the rest of the method is unchecked. Hence, the reachability analysis provides a warning about this bug.

4.3 Loop Unrolling

Apart from the safe desugaring discussed in the previous section, ESC/Java2 supports an unsound handling of loops 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, ESC/Java does not detect errors that may only happen when a loop is executed more than L times. The following schematically describes the result of an unrolling for $L = 2$:

```
while (C) {              if $C$ then $B$;
  B            $\Rightarrow$          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 command **assume** *false*.

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 (for $L = 2$):

```
                           $C_1$:  if $0 < 10$ then
                                     assume $i_1 = 0 + 1$;
int i = 0;                 $C_2$:  if $i_1 < 10$ then
while (i < 10)                       assume $i_2 = i_1 + 1$;
    i++;           $\Rightarrow$      $C_3$:  if $i_2 < 10$ then
return 1/0;                          assume $false$;
                           $C_4$:  assert $0 \neq 0$;
                           $C_5$:  assume $RES = 1/0$
```

We note that $T \models \neg\mathrm{N}(\mathit{true}, C_1; C_2; C_3)$. From Observation 1, it follows that $T \models \neg\mathrm{W}(\mathrm{N}(\mathit{true}, C_1; C_2; C_3), C_4; C_5)$. Therefore, the assertion C_4 cannot cause the program to go wrong since from the point of view of the checker that assertion is unreachable.

The analysis presented in this article detects that the code following the loop is not checked. Once the user is informed about it, he or she may either instruct ESC/Java2 to unroll the loop more times or may provide appropriate loop invariants and instruct ESC/Java2 to use safe desugaring.

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 used in these experiments (Simplify [12]) sharply increases when the size of the query exceeds a certain limit, which motivates (1). A prover call is on average hundreds times slower than any reasonable manipulation of the flow graph, which motivates (2).

The precondition of each node can be computed from Definition 3. If the implementation is memoized then the precondition will be represented as 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. (Note that $\mathrm{N}(\mathrm{pre}_G(p), \mathcal{L}(p))$ may introduce at most one $\wedge$ operator, according to Figure 3.) Unfolding the DAG naively into a tree to send it to a prover often yields 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 auxiliaries increase the query size. 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. This transformation exploits the series–parallel structure of the flow graph. Hence, the queries are roughly the same size as the normal behavior computed directly on the DSA as in [14].

The auxiliary variables can be defined using equivalence.

$$\big(a \Leftrightarrow f(b)\big) \wedge g(a, b) \qquad (5)$$

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:

$$\big(a \Rightarrow f(b)\big) \wedge g(a, b) \qquad (6)$$

It can be shown that (5) is satisfiable if and only if (6) is satisfiable, provided that g is monotonic in a, that is, $g(\mathit{false}, b) \Rightarrow g(\mathit{true}, b)$. We can make sure that that is the case by eliminating sharing only below the operators $\wedge$ and $\vee$. (Note that $\wedge$ and $\vee$ are the only operators introduced by the N predicate.) In practice, replacing (5) by (6) 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*. The details of how to keep track from where in the Java code a DSA command comes from are outside the scope of this paper and can be found elsewhere [24]. For typical Java code there are less than 20 interesting nodes 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 (Figure 5) 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 $d$
      then PROPAGATE-REACHABLE($d$)
```

Figure 5: Reachability propagation.

```
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 satisfiable
            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($d'$)
```

Figure 6: The algorithm implementing the analysis.

We compute dominators ignoring nodes already marked as unreachable using the simple algorithm of Cooper [8], 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.

In the case that all nodes are reachable and interesting the greedy algorithm (Figure 6) is optimal, because the prover must be called for all the leafs of the (immediate) dominator tree. In practice the performance is good. We have run ESC/Java2 on its front-end (`javafe`) which contains 1890 methods and is one of the largest coherent set of JML-annotated code available. The total running time is 31589 seconds (almost 9 hours), out of which 34.8% is spent in the reachability analysis, out of which 99.8% is spent in the prover. The total number of leafs in the dominator trees is 3256 and the number of prover calls is 3351. The average number of nodes in the flow graph is a few hundred and in the contracted flow graph it is 10. For this benchmark we used the default loop desugaring in ESC/Java2, which is unrolling once.

6. CASE STUDY

As described in the previous section, we have tested the analysis on the ESC/Java2 front-end, the `javafe` package. The package contains 217 classes.

We have found 5 inconsistencies in the specifications of the JDK that are not reported without the reachability analysis. More details can be found in the ESC/Java2 bug-tracker[2] under the bugs #595, #550, #568, #549, #545. We found one more inconsistency in the JDK specification which was due to the incorrect handling of a JML feature *informal comment* by ESC/Java2. ESC/Java2 treats an informal comment as *true*, this is harmless in most cases (such as `requires (* is upper-case *)`) but for example, `ensures \result <=> (* is upper-case *)` likely results in an unintended specification (see bug #547).

ESC/Java2's repository contains handcrafted tests to detect inconsistencies in the JDK specifications. These tests did not detect the problems uncovered by the reachability analysis because they are not exhaustive. We should note 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.

In 1 case a **catch**-block was unreachable because it was catching an exception that was not declared in any of the specifications of the methods called in the **try**-block (see documentation for the `signals_only` pragma).

An incorrect use of the `modifies` clause (as in Figure 4) hiding the rest of the potential warnings appeared 9 times. Warnings hiding subsequent code appeared 6 times. The case of unreachable code resulting from loop unrolling, as discussed in Section 4.3, appeared 4 times. In 9 cases the informal comments indicated that the author was aware that the code is unreachable. The user can mark such code 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 unreachability 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 are not spurious.

[2]`http://sort.ucd.ie/tracker/?group_id=97`

In the remaining 12 cases we were not able to 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. RELATED WORK

Traditionally, unreachable code is detected by techniques based on data flow analysis or abstract interpretation [9]. These techniques are generally known under the term *dead code elimination* [26] and are used for code optimization. To our knowledge, automated theorem proving is not used in mainstream compilers. Interactive theorem proving, however, is used to show properties of code optimizations. For example, Blech et al. [3] applied the higher-order theorem prover Isabelle/HOL to mechanically prove that a code optimization based on dead code elimination is semantics preserving.

Another stream of research related to our work is focused on reasoning about specifications for which there is no implementation available. Chalin [6] describes an enhancement of ESC/Java2 that checks 'definedness' of specifications. An example of a partial specification is `requires a.x == 0`; since it does enforce `a` to be non-**null**. As in the case of our work, this technique is fully automated.

Bouquet et al. [4] use a constraint solver to *animate* specifications. Basically, specification animation provides a way to debug specifications without implementation. The animating system maintains an abstract state and the user can ask the system what happens to that state if a certain method is called. Using this technique, it is possible to uncover that a sequence of method calls necessarily lead to an inconsistent state.

The term reachability analysis is used in related areas in a slightly different sense. In model checking it denotes the analysis that searches for reachable states of the given state space [1]. In heap analysis the reachability analysis is done on the reference graph [7].

8. SUMMARY 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 [11], 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 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?

9. ACKNOWLEDGMENTS

This work is funded by Science Foundation Ireland under grant number 03/CE2/I303-1, "LERO: the Irish Software Engineering Research Centre" and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The article contains only the authors' views and the Community is not liable for any use that may be made of the information therein.

10. REFERENCES

[1] P. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on SAT-solvers. *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 2000.

[2] M. Barnett, K. Leino, and W. Schulte. The Spec$^{\#}$ programming system: An overview. In *Proceeding of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, 2004.

[3] J. O. Blech, S. Glesner, and J. Leitner. Formal verification of dead code elimination in Isabelle/HOL. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, 2005.

[4] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In *Proceedings of Formal Methods, International Symposium of Formal Methods (FM 2005)*, Lecture Notes in Computer Science. Springer–Verlag, 2005.

[5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, Feb. 2005.

[6] P. 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, Nov. 2006.

[7] S. Chatterjee, S. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[8] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm, 2001. Available online at `www.cs.rice.edu/~keith/EMBED/dom.pdf`.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1978.

[11] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[12] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery*, 52(3):365–473, 2005.

[13] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2002.

[14] C. Flanagan and J. 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, Jan. 2001.

[15] L. Friendly. The design of distributed hyperlinked programming documentation. *IWHD*, 95:151–173, 1995.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[17] M. Janota. Assertion-based loop invariant generation. In *Proceedings of the 1st International Workshop on Invariant Generation (WING '07)*, June 2007.

[18] J. R. Kiniry and D. 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, Jan. 2005.

[19] D. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[20] C. Krueger. Software reuse. *ACM Computing Surveys*, 24(2), 1992.

[21] G. T. Leavens, A. L. Baker, and C. Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.

[22] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19, 2007.

[23] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005.

[24] K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1-3):209–226, 2005.

[25] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May 1999.

[26] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer–Verlag, 1999.

[27] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of Series–Parallel digraphs. *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12, 1979.