Reachability Analysis for Annotated Code

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

[1]Systems Research Group,
University College Dublin, Ireland

[2]Institute of Computer Science
University of Wrocław, Poland

SAVCBS '07

Mobius lero
IST-15905

Why Annotated Code?

Static Checking Example

```
//@ ensures \result >= a;
//@ ensures \result >= b;
int max(int a, int b) {
  if (b > a)
    return b;
  else
    return b;
}
```

Why Annotated Code?

Static Checking Example

```
//@ ensures \result >= a;
//@ ensures \result >= b;
int max(int a, int b) {
  if (b > a)
    return b;
  else
Bug ⇝ return b;
}
```

Is It Possible that Some Things Are not Checked?

Code-Spec Inconsistency

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

Is It Possible that Some Things Are not Checked?

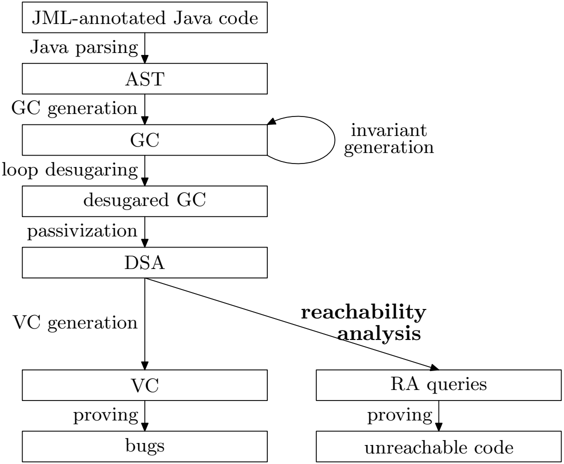
Code-Spec Inconsistency

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

Inconsistent Spec

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

int useLibraryFunc() {
  int r = libraryFunc(11);
  return 1/0; //not checked
}
```

ESC/Java2 Architecture



Input Language

Dynamic Single Assignment (DSA)

$$cmd := \textbf{assume}\ f \mid \textbf{assert}\ f \mid cmd \mathbin{[\!]} cmd \mid cmd\ ;\ cmd$$

where f is a first-order logic predicate on the program variables

Inconsistent Spec

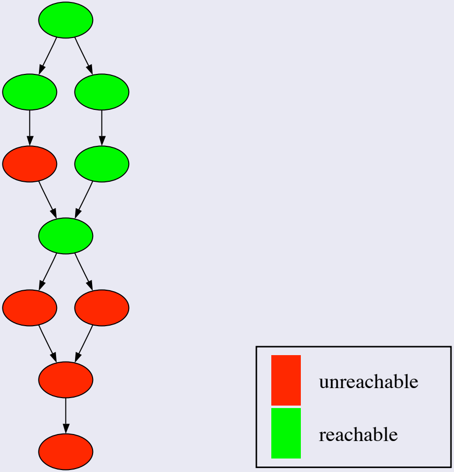
```
/*@ requires i >= 10;
  @ ensures \result == i;
  @ ensures \result < 10;*/
int libraryFunc(int i);

int useLibraryFunc() {
  int r = libraryFunc(11);
  return 1/0;  // not checked
}
```

useLibraryFunc as DSA

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

Reachability Propagation in Control Flow Graph

Code is unreachable if all paths leading to it block:



Computing Unreachable Code

Construct a *control flow graph* from DSA

- directed acyclic (DAG)
- nodes are labeled with commands:

$$\mathcal{L} : \mathrm{Nodes} \to \{\textbf{assume}\, f, \textbf{assert}\, f\}$$

Computing Unreachable Code

Construct a *control flow graph* from DSA

- directed acyclic (DAG)
- nodes are labeled with commands:

$$\mathcal{L} : \mathrm{Nodes} \to \{\textbf{assume}\ f, \textbf{assert}\ f\}$$

Compute *preconditions* and *postconditions* for nodes

$$\mathrm{post}(n) \equiv \mathrm{SP}(\mathrm{pre}(n), \mathcal{L}(n)) = \mathrm{pre}(n) \wedge f$$

$$\mathrm{pre}(n) \equiv \begin{cases} \mathit{true} & \text{if } n \text{ is an entry node} \\ \bigvee_{p \in \mathrm{parents}(n)} \mathrm{post}(p) & \text{otherwise} \end{cases}$$

Computing Unreachable Code

Construct a *control flow graph* from DSA

- directed acyclic (DAG)
- nodes are labeled with commands:

$$\mathcal{L} : \text{Nodes} \to \{\textbf{assume}\ f, \textbf{assert}\ f\}$$

Compute *preconditions* and *postconditions* for nodes

$$\text{post}(n) \equiv \text{SP}(\text{pre}(n), \mathcal{L}(n)) = \text{pre}(n) \wedge f$$

$$\text{pre}(n) \equiv \begin{cases} \textit{true} & \text{if } n \text{ is an entry node} \\ \bigvee_{p \in \text{parents}(n)} \text{post}(p) & \text{otherwise} \end{cases}$$

Call the Theorem Prover

for each node n,
ask the theorem prover if $\text{pre}(n)$ is *unsatisfiable*

Can We Do Better?

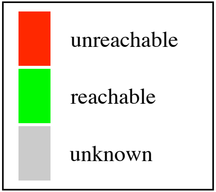
Observations

1. reachability information can be propagated
2. most nodes are reachable
3. most nodes dominate some other node

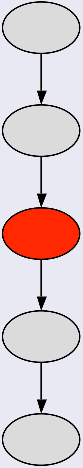
Can We Do Better?

Observations

1. reachability information can be propagated
2. most nodes are reachable
3. most nodes dominate some other node



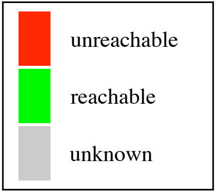
Example of Propagation



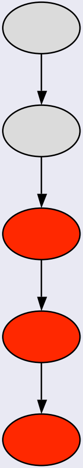
Can We Do Better?

Observations

1. reachability information can be propagated
2. most nodes are reachable
3. most nodes dominate some other node



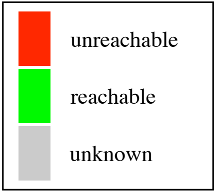
Example of Propagation



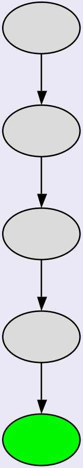
Can We Do Better?

Observations

1. reachability information can be propagated
2. most nodes are reachable
3. most nodes dominate some other node



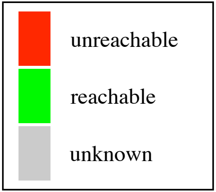
Example of Propagation



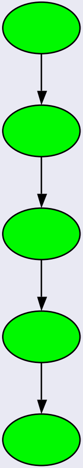
Can We Do Better?

Observations

1. reachability information can be propagated
2. most nodes are reachable
3. most nodes dominate some other node



Example of Propagation



Algorithm — Greedy Heuristic

1. Compute:
   i. T — the immediate dominator tree of the nodes not known to be unreachable.
   ii. r — the root of T.
2. Choose an unlabeled node x in T with a maximal number of unlabeled dominators (greedy choice).
   i. Query the prover on x.
   ii. Label x *reachable*/*unreachable* accordingly and propagate.
   iii. If x is reachable then *go to* step 1.
3. By using binary search find the unreachable node on the path from r to x that is closest to r (the 'broken link' in chains). Label and propagate accordingly.
4. Repeat from step 1 while there are unlabeled nodes.

Case Study

Where

- ESC/Java2's front-end (`javafe`)
- 1890 methods
- running time 9 hours where reachability analysis took 34.8%

The Most Interesting Problems

- uncovered 5 inconsistencies in the JDK specifications
    - including a problem in treating of the *informal comment* `ensures \result <=> (* is upper-case *)`
- deficiencies of the checker (e.g., in *loop unrolling*)
- catching an undeclared exception
- most common: an error hiding subsequent code
- in some cases we don't know why the code is unreachable

Conclusions and Future Work

- unreachable code is a problem in practice, nevertheless,
- finding the exact source of unreachability is difficult, thus,
- in our future work we want to explore how we can provide more helpful feedback to the user

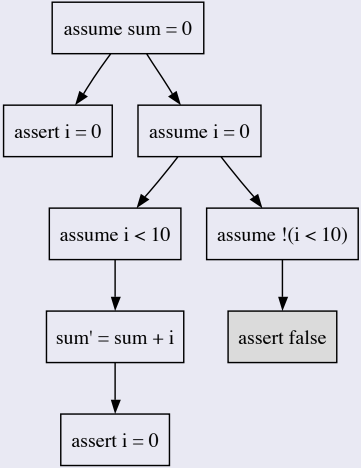
The implementation is in the ESC/Java2's `cvs` *head and can be enabled by the switch* `-era`.

Example with a Loop

Infinite Loop

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

DSA Control Flow Graph



Loop Unrolling

Loop Unrolled Twice

if C **then** B;
if C **then** B;
if C **then assume** *false*;