Detecting Erroneous Assumptions when verifying software using SMT solvers

David R. Cok
Eastman Kodak Company Research Laboratories
14 July 2008
AFM '08

(Note: E-version distributed at CAV is the preliminary, not final version)

Context

- Industrial software verification
- Extended static checking
  - software verification via
    - user supplied or implicit specifications
    - creating a verification condition from the code and specifications, and then
    - validating it (preferably automatically) using a theorem prover
  - e.g. ESC/Java(2), Key for Java, Spec# for C#, also Mobius project, COQ system, ...
  - e.g. provers: SIMPLIFY, Yices, CVC3, Z3, PVS, ...

Erroneous assumptions are insidious

- User written material is subject to error
  - Explicit assumptions
  - Method specifications

- False assumptions are generally not what was intended

- Insidious: hide other errors

- If a verification system produces no errors
  - Everything OK?
  - Something not being checked?
  - False assumption hiding an invalid assertion?

- Lots of work on this in model checkers; some in automated runtime test analysis

Review: translation of programs to VCs

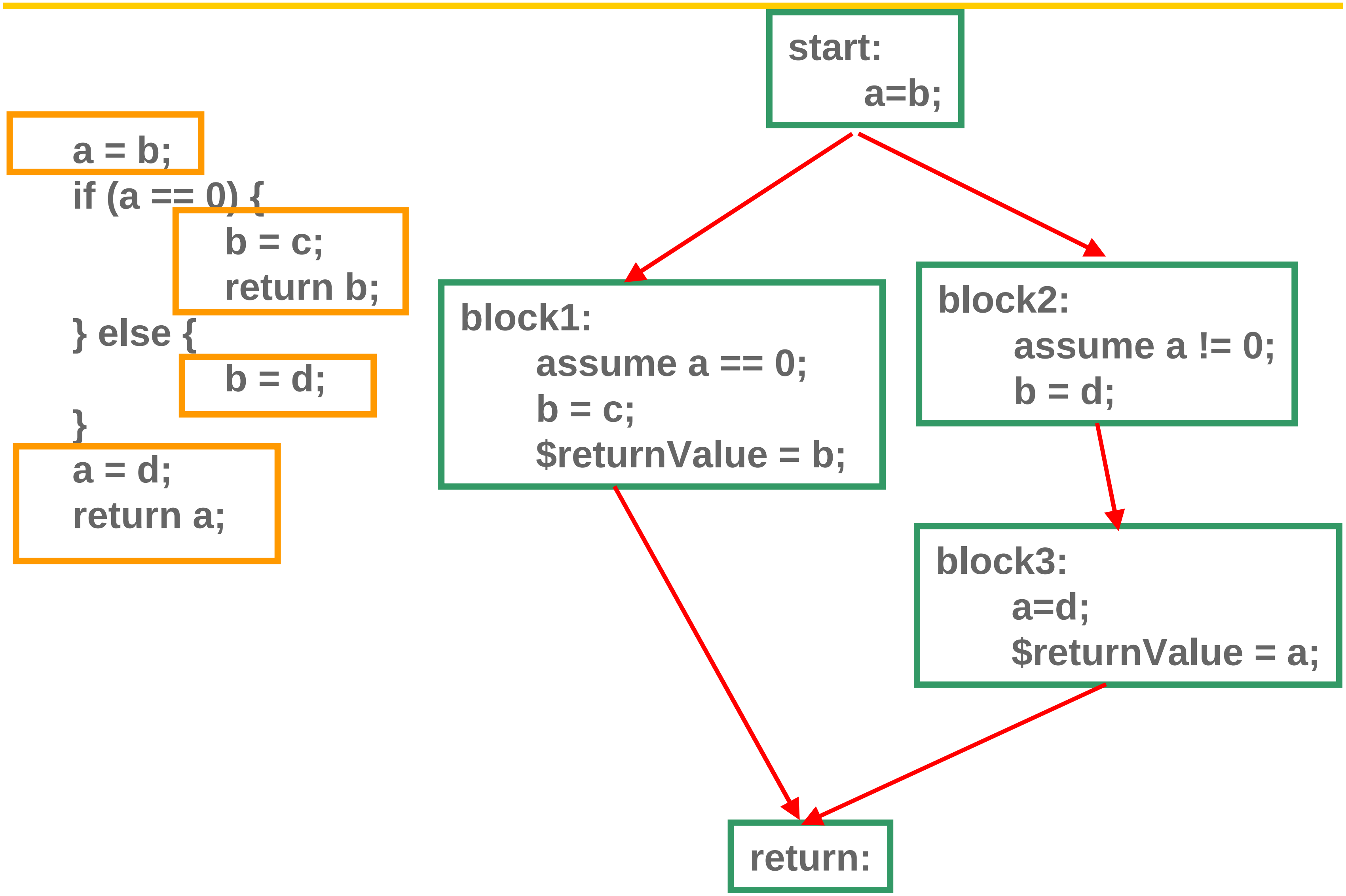
- Break up a program into basic blocks
  - Each block has no branches
  - Blocks are followed by other blocks

- Transform variables into (dynamic) single assignment form

- Passify the program by converting all assignments to assumptions

(Barnett & Leino, 2005)

Basic blocks



```
a = b;
if (a == 0) {
        b = c;
        return b;
} else {
        b = d;
}
a = d;
return a;
```

```
start:
        a=b;
```

```
block1:
        assume a == 0;
        b = c;
        $returnValue = b;
```

```
block2:
        assume a != 0;
        b = d;
```

```
block3:
        a=d;
        $returnValue = a;
```

```
return:
```

Dynamic Single Assignment

```
int a = 0;
int b = 1;
b = a + b;
a = b + a;
```

```
a$0 = 0;
b$0 = 1;
b$1 = a$0 + b$0;
a$1 = b$1 + a$0;
```

Tricky points

- arrays and object field assignments
- blocks with multiple parents

The a$0 etc. are *logical* variables (quantified over the appropriate domain of values)

Passification

```
a = 0;

b = a;

b = a + b;
```

```
a$0 = 0;

b$0 = a$0;

b$1 = a$0 + b$0;
```

```
assume a$0 == 0;

assume b$0 == a$0;

assume b$1 == a$0 + b$0;
```

Convert basic block to block equations:

```
blockA:
  assume P;
  assume Q;
  assert R;
  assume S;
  goto blockB,
          blockC;
```

Assumptions come from

- assignments
- branch conditions
- loop conditions
- preconditions
- postconditions of called methods
- explicit user assumptions

Convert basic block to block equations:

```
blockA:
  assume P;
  assume Q;
  assert R;
  assume S;
  goto blockB,
          blockC;
```

Assertions come from

implicit checks (e.g. array index)
loop specifications
postconditions
preconditions of called methods
explicit user assertions

Convert basic block to block equations:

```
blockA:
  assume P;
  assume Q;
  assert R;
  assume S;
  goto blockB, blockC;
```

```
blockA ≡
   P →
     ( Q →
          ( R & ( S →
                  (blockB & blockC) ) ) )

blockB ≡ ...

blockC ≡ ...
```

Each block has a (logical) block variable

- if true, execution encounters no false assertions
- may block at a false assumption

... and block equations to a Verification Condition

((blockA ≡ ...)

& (blockB ≡ ...)

& ...) => blockA

The variable of the starting block

This says: for any assignment of values to variables,
if the block equations are satisfied,
then the program has a valid execution

A valid execution allows false assumptions

Parallel path form of the VC

$$
\begin{array}{ll}
 & (P \;\&\; Q \;\&\; R \;\&\; \ldots) \Rightarrow T_1 \\
\& & (P \;\&\; Q \;\&\; S \;\&\; \ldots) \Rightarrow T_2 \\
\& & (X \;\&\; Q \;\&\; \ldots \qquad\quad) \Rightarrow T_3 \\
\& & (Z \;\&\ldots \qquad\qquad\quad) \Rightarrow T_4 \\
\& & \ldots
\end{array}
$$

Each conjunct is an execution path:

a sequence of assumptions ending in an assertion

Lots of common subformulas

Parallel path form of the VC

$$
\begin{aligned}
& (P \mathbin{\&} Q \mathbin{\&} R \mathbin{\&} \ldots) \Rightarrow T_1 \\
\mathbin{\&}\quad & (P \mathbin{\&} Q \mathbin{\&} S \mathbin{\&} \ldots) \Rightarrow T_2 \\
\mathbin{\&}\quad & (X \mathbin{\&} Q \mathbin{\&} \ldots \qquad\quad) \Rightarrow T_3 \\
\mathbin{\&}\quad & (Z \mathbin{\&} \ldots \qquad\qquad\quad) \Rightarrow T_4 \\
\mathbin{\&}\quad & \ldots
\end{aligned}
$$

The VC is true iff each path (trace) either

- has a false assumption
- has a true assertion

Assumptions

- assignments
- loop invariants
- branch/loop conditions
- preconditions
- called method postconditions
- explicit assumptions

System generated:
No problems

Bad invariants create
unprovable assertions
as well as bad assumptions

Assumptions

- assignments
- loop invariants
- branch/loop conditions
- preconditions
- called method postconditions
- explicit assumptions

If a branch condition is always false:
dead code

Loop condition is always false:
not executed or
never terminated loop

Assumptions

- assignments
- loop invariants
- branch/loop conditions
- preconditions → Contradictory preconditions: any assertion succeeds
- called method postconditions
- explicit assumptions

Assumptions

- assignments
- loop invariants
- branch/loop conditions
- preconditions
- called method postconditions
- explicit assumptions

Contradictory postconditions:
any subsequent
assertion succeeds

Should be caught when the called method is verified

Assumptions

- assignments
- loop invariants
- branch/loop conditions
- preconditions
- called method postcond
- explicit assumptions

False user assumption:
any subsequent
assertion succeeds

(Might be false just on one path)

Assumptions

Need to check for assumptions that are false
(given previous assumptions):

- false on all paths:
  preconditions,
  branch conditions (dead code)

- false on some path:
  user assumptions,
  called method postconditions

Specific path check

In a path

(P1 & P2 & P3 & P4 & ...) => T

assumption Pk is OK if

Need to check each assumption on each path ???

(P1 & ... & Pk) is satisfiable

Equivalently

(P1 & ... & Pk) => false is invalid

Better: check all assumptions in a given path

In a path

(P1 & P2 & P3 & P4 & ... & Pn) => T

all assumptions are OK if

(P1 & ... & Pn) is satisfiable

Equivalently

(P1 & ... & Pn) => false is invalid

One check per path.
Still, there may be many paths.

Also, some paths are infeasible because of contradictory branch conditions

Checking within the block equations

```
block:
  assume P;
  assume Q;
  assert false;
  assume R;
  ...
```

Insert an extra assertion:

If VC is still valid, then something is wrong prior to the assertion.

[If the assertion provokes a warning then all is well.]

Might as well do the check at the end of the block.

Checks that the assumptions are valid on SOME path (not necessarily all paths)

Previous work: Janota et al., 2007

- Putting in ‘assert false;’ is a standard manual idiom for checking feasibility of assumptions
- Janota et al. automated this in ESC/Java2, along with a search algorithm
  - optimized for short VCs and few prover invocations
- Improvements:
  - Use incremental satisfiability checks
  - How to do path specific checks
  - Use unsatisfiable cores

Incremental satisfiability checking

- Minimal changes to the VC
- Uses the SMT solver's ability to
  - push/pop program state
  - or to retract assertions

Incremental satisfiability checking

- Put in all the ‘assert’ statements to check assumptions at once. But

instead of

```
block:
  assume P;
  assume Q;
  assert false;
  assume R;
  ...
```

write (e.g. for check # 17)

```
block:
  assume P;
  assume Q;
  assert $$count != 17;
  assume R;
  ...
```

Incremental satisfiability checking

Then, for the usual SAT check of the VC, check

VC & ($$count == 0)

And then check each assumption N by testing

VC & ($$count == N)

(retract ‘$$count==0’ and assert ‘$$count == N’)

Performance question

Which is faster:

reformulating the VC and restarting the prover

or

saving/restoring program state, followed by an incremental SAT check

[or

using retract/reassert]?

The prover needs to do this internally to facilitate backtracking

In Yices, enabling this mode is overall less efficient.

Path specific checks

- Use a conditional assertion:

instead of

```
block:
  assume P;
  assume Q;
  assert false;
  assume R;
  ...
```

write

```
block:
  assume P;
  assume Q;
  assert !Z;
  assume R;
  ...
```

where Z is true only for the path being checked
(it is a conjunction of all the branch conditions for the path)

Performance question

Which is faster:

reformulating the VC and restarting the prover with just the small VC for a specific path

or

using incremental checking with the full VC?

Even better: avoid path-specific checking

@NonNull int[ ] a;

...

sort(a);

Postcondition:
forall int i: ((0<i && i<a.length) =>
a[i-1] <= a[i])

...

(needs to know: j < k => a[j] <= a[k])

[Prover does not do induction]

Even better: avoid path-specific checking

Could write:

Postcondition:
forall int i: ((0<i && i<a.length) =>
a[i-1] <= a[i])

```
@NonNull int[ ] a;

...

sort(a);

/*@ assume (\forall int j,k; 0<=j && j<=k && k<a.length;
                                a[j] <= a[k]); */

...
```

(needs to know: j < k => a[j] <= a[k])

Even better: avoid path-specific checking

Better:

```
@NonNull
...
sort(a);
/*@ assume (\forall int i; 0<i && i<a.length;
                          a[i-1] <= a[i])    =>

                (\forall int j,k; 0<=j && j<=k && k<a.length;
                          a[j] <= a[k]); */
...
```

(needs to know: j < k => a[j] <= a[k])

Postcondition:

Presumes the prover can handle this syntax.

Presumes the prover will instantiate the quantifications when needed.

Using unsatisfiable cores

- The usual check of a program’s VC tells if the VC is unsatisfiable (== the program is valid)

- Some provers can also provide an unsatisfiable core: a subset of assertions that by themselves are unsatisfiable.

- This can be used to check for bad assumptions (and in general for irrelevant code/specs)

Using unsatisfiable cores

Instead of a monolithic VC:

```
(  ( blockA ≡ ... )
& ( blockB ≡ ... )
& ...                           ) => blockA
```

use individual assertions (depending on the prover):

```
assert blockA ≡ ... ;
assert blockB ≡ ... ;
...
assert !blockA;
```

Using unsatisfiable cores

AND, for a given check, insert an extra assert statement and a top-level assertion that the predicate is true

```
block:
  assume P;
  assume Q;
  assume R;
  assert Zk;
  ...
```

```
assert blockA ≡ ... ;
assert blockB ≡ ... ;
assert ...
assert !blockA;
assert Zk;
```

However, if ‘assert Zk’ is NOT part of the UNSAT core, then it does not matter if Zk is true => SOMETHING AMISS

Using unsatisfiable cores

- Insert an extra (but different) ‘assert Zk’ wherever checks are needed (can also use path dependent predicates)
- Test whether the associated formula is part of the unsatisfiable core (one check if the core is minimal)
- If yes => preceeding assumptions are feasible
- If no => something is infeasible prior to the assert

Using unsatisfiable cores

Issue:

- tools do not guarantee *minimal* unsatisfiable cores
- may need to individually test the some of the assertions in the provided UNSAT core to see if they are in the minimal core
- no fast algorithm known

Performance question:

- Is using UNSAT cores a performance improvement over individual SAT checks?

Implementation

Techniques tested using

- a nascent version of JML for Java 1.6/1.7
- built on the OpenJDK source code base
  - provides the Java 1.6->1.7 functionality
- using Yices as the backend prover
  - allows incremental SAT checking
  - provides UNSAT cores

Tested by hand using C#/Spec#

(no incremental or UNSAT core functionality)

Industrial scale performance comparisons in progress...

For the future: Relevance

- Vacuity is a subset of Relevance
- UNSAT cores can be used to assess relevance
- A subterm or set of terms is not relevant if it is not needed to prove the result

Test for relevance

Change the VC

... <expr> ...

to

... Z ...
& Z == <expr>

and check for unsatisfiability (VC is equivalent)

If ‘Z == <expr>’ is NOT part of the UNSAT core, then it is not needed to prove the specifications:

it is irrelevant

Implications of irrelevance

- Problem with the code: some computations might actually be irrelevant
  - Unused assignments
  - Incorrect logic

- Problem with the specs:
  - Specs have inadequate *coverage* (not all of the code is needed to establish the specs)
  - Analogous to coverage checking for runtime tests

Concluding Observations

- As noted by many: checking for infeasible (vacuous) assumptions is important

- Such checks can be simplified and the performance improved (we anticipate) by using
  - incremental satisfiability checks
  - unsatisfiable cores

- It can be helpful to reformulate the VC using new variables that substitute for subformulae under scrutiny (appropriate names can help in understanding counterexamples)

- User-supplied assumptions are best formulated as quantified tautologies without free variables

Performance questions in progress

- SAT checking vs. UNSAT cores
    - (there is a penalty to assert formulae such that cores can be produced and to allow retractions)
- Using incremental checks vs. from scratch checks (with usual satisfiability checking) to check assumptions
- Use of definitions vs. formulating multiple smaller VCs (for path-specific SAT checking)
- Are these comparisons significantly different across different provers

Kodak