Bounded Model Checking

CS560: Reasoning About Programs

Roopsha Samanta

Roadmap

Previously

- Invariant generation and abstract interpretation

Today

- Bounded model checking for programs

Model Checking

Clarke

Emerson

Sifakis

Clarke & Emerson, *Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, 1981*

Algorithmic framework for exhaustive exploration of finite-state transition systems to check temporal properties

Bounded Model Checking (BMC)

- Represent the transition system as a formula in propositional logic
- Instead of exploring all states exhaustively, unroll the transition relation up to a certain fixed bound and search for violations of the property within that bound
- Transform this search to a Boolean satisfiability problem, solve using a SAT solver
- If no *counterexamples* found using current bound, increment bound and repeat

Great for refutation of properties!

BMC for Programs using SMT Solvers

1. Unwind loops a fixed k times
   - Duplicate the loop body k times
   - Guard each copy using an if statement that checks the loop condition
   - At the end of k repetitions, add an *unwinding assertion* that asserts the negation of the loop condition
     - If unwinding assertion fails, then there exists a program execution that exceeds bound (loop executes > k times)
     - If unwinding assertion passes, then bound is sufficient for verification!

```
x := 0
y := 0
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    y := y + 1
assert x ≤ y
```

Unwind loop
k = 2 times

```
x := 0
y := 0
if (y <= n)
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    y := y + 1
    if (y <= n)
        if (z == 0) then
            x := x + 1
        else
            x := x + y
        y := y + 1
        assert y > n
assert x ≤ y
```

Unwinding assertion

BMC for Programs using SMT Solvers

2. Transform program to static single assignment (SSA) form
   - Rename variables so that each variable is assigned only once
   - For each join point after a conditional, add new variables with selectors

$x := 0$
$y := 0$
if $(y <= n)$
    if $(z == 0)$ then
        $x := x + 1$
    else
        $x := x + y$
    $y := y + 1$
    if $(y <= n)$
        if $(z == 0)$ then
            $x := x + 1$
        else
            $x := x + y$
        $y := y + 1$
        assert $y > n$
assert $x \leq y$

SSA conversion →

$x_1 := 0$
$y_1 := 0$
if $(y_1 <= n_0)$
    if $(z_0 == 0)$ then
        $x_2 := x_1 + 1$
    else
        $x_3 := x_1 + y_1$
    $x_4 := \phi(x_2, x_3)$
    $y_2 := y_1 + 1$
    if $(y_2 <= n_0)$
        if $(z_0 == 0)$ then
            $x_5 := x_4 + 1$
        else
            $x_6 := x_4 + y_2$
        $x_7 := \phi(x_5, x_6)$
        $y_3 := y_2 + 1$
        assert $y_3 > n_0$
assert $x_7 \leq y_3$

If $z_0 == 0$,
$\phi(x_2, x_3) = x_2$,
else
$\phi(x_2, x_3) = x_3$

BMC for Programs using SMT Solvers

3. Generate a SAT/SMT encoding
   - Generate forward propagation constraints R for program
   - Generate constraints A for (unwinding) assertions
   - Check if $R \wedge \neg A$ is satisfiable
   - If sat, then some (unwinding) assertion is violated
   - If unsat, program satisfies all assertions!

```
x1 := 0
y1 := 0
if (y1 <= n0)
    if (z0 == 0) then
        x2 := x1 + 1
    else
        x3 := x1 + y1
    x4 := φ(x2, x3)
    y2 := y1 + 1
    if (y2 <= n0)
        if (z0 == 0) then
            x5 := x4 + 1
        else
            x6 := x4 + y2
        x7 := φ(x5, x6)
        y3 := y2 + 1
        assert y3 > n0
assert x7 ≤ y3
```

Strongest postcondition computation without existential quantification!

$R \equiv$
$x_1 = 0 \ \wedge$
$y_1 = 0 \ \wedge$
$x_2 = x_1 + 1 \wedge$
$x_3 = x_1 + y_1 \wedge$
$y_2 = y_1 + 1 \wedge$
$x_5 = x_4 + 1 \wedge$
$x_6 = x_4 + y_2 \wedge$
$y_3 = y_2 + 1 \wedge$
$(y_1 \leq n_0 \wedge ((z_0 = 0 \wedge x_4 = x_2) \vee (z_0 \neq 0 \wedge x_4 = x_3)) \wedge$
$(y_2 \leq n_0 \wedge ((z_0 = 0 \wedge (x_7 = x_5) \vee (z_0 \neq 0 \wedge (x_7 = x_6))))$

$A \equiv y_3 > n_0 \wedge x_7 \leq y_3$

$x_1 := 0$
$y_1 := 0$
if $(y_1 <= n_0)$
    if $(z_0 == 0)$ **then**
        $x_2 := x_1 + 1$
    else
        $x_3 := x_1 + y_1$
    $x_4 := \phi(x_2, x_3)$
    $y_2 := y_1 + 1$
    if $(y_2 <= n_0)$
        if $(z_0 == 0)$ **then**
            $x_5 := x_4 + 1$
        else
            $x_6 := x_4 + y_2$
        $x_7 := \phi(x_5, x_6)$
        $y_3 := y_2 + 1$
        assert $y_3 > n_0$
assert $x_7 \leq y_3$

$R \equiv$
$x_1 = 0 \land$
$y_1 = 0 \land$
$x_2 = x_1 + 1 \land$

If $R \land \neg A$ is unsat, program correct!
Else if $R \land \neg A_2$ is sat, program buggy!
Else $R \land \neg A_1$ is sat, i.e., program correct upto bound k, but there exists some execution of length > k; increase bound k and repeat.

$A \equiv y_3 > n_0 \land x_7 \leq y_3$

$A_1 \equiv y_3 > n_0$

$A_2 \equiv x_7 \leq y_3$

Summary

Today

- Bounded model checking for programs
  - Unwinds loops a fixed, bounded number of times
  - SSA form makes forward propagation straightforward
  - An effective refutation technique
  - Unwinding assertions enable verification

Next

- Model Checking