Software Model Checking

Lecture 3

Aditya V. Nori
Microsoft Research India
So far..
So far..
Newton

- Given an error path $p$ in boolean program $B$
  - is $p$ a feasible path of the corresponding $C$ program?
    - Yes: found an error
    - No: find predicates that explain the infeasibility
Newton

- Execute path symbolically

- Check conditions for inconsistency using theorem prover (satisfiability)

- Obtain predicates after detecting inconsistency
do {
  KeAcquireSpinLock();

  if(*){
    KeReleaseSpinLock();
    KeReleaseSpinLock();

    } while (*);

  KeReleaseSpinLock();
}

Example

Model checking boolean program (bebop)
Example

Is error path feasible in C program? (newton)

```c
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock();
```
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets; b = true;
    if(request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++; b = b ? false : *;
    }
} while (nPackets != nPacketsOld) !b
KeReleaseSpinLock();
Symbolic simulation for C--

Domains

– variables: names in the program
– values: constants + symbols

State of the simulator has 2 components:

– store: map from variables to values
– conditions: predicates over symbols
Symbolic simulation algorithm

Input: path $p$

For each statement $s$ in $p$ do
  match $s$ with
  Assign($x,e$):
    let val = Eval($e$) in
    Store[$x$] := val
  Assume($e$):
    let val = Eval($e$) in
    Cond := Cond and val
    let result = CheckConsistency(Cond) in
    if (result == "inconsistent") then
        GenerateInconsistentPredicates()
  End

Say "Path $p$ is feasible"
int g;

main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

void cmp (int a , int b) {
    Goto L1, L2
    L1: assume(a==b);
        g = 0;
        return;
    L2: assume(a!=b);
        g = 1;
        return;
}
int g;

main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

Global:

main:
(1) x: X
(2) y: Y

Conditions:

void cmp (int a, int b) {
    Goto L1, L2

    L1: assume(a==b);
        g = 0;
        return;

    L2: assume(a!=b);
        g = 1;
        return;
}

Global:

main:
(1) x: X
(2) y: Y

Conditions:
void cmp (int a , int b) {
    Goto L1, L2
    L1: assume(a==b);
        g = 0;
        return;
    L2: assume(a!=b);
        g = 1;
        return;
}

int g;
main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

Global:

main:
(1) x: X
(2) y: Y

cmp:
(3) a: A
(4) b: B

Conditions:
Map:
X → A
Y → B
int g;
main(int x, int y){
  cmp(x, y);
  assume(!g);
  assume(x != y)
  assert(0);
}

void cmp (int a , int b) {
  Goto L1, L2
  L1: assume(a==b);
      g = 0;
      return;
  L2: assume(a!=b);
      g = 1;
      return;
}

Global:
(6) g: 0

main:
(1) x: X
(2) y: Y

cmp:
(3) a: A
(4) b: B

Conditions:
(5)(A == B) [3, 4]
void cmp (int a , int b) {
    Goto L1, L2
    L1: assume(a==b);
        g = 0;
        return;
    L2: assume(a!=b);
        g = 1;
        return;
}  

int g;
main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

Global:
(6)  g:   0
main:
(1)  x:   X
(2)  y:   Y
cmp:
(3)  a:   A
(4)  b:   B

Conditions:
(5)(A == B)  [3, 4]
(6)(X == Y)  [5]
int g;

main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

void cmp (int a , int b) {
    Goto L1, L2

    L1: assume(a==b);
        g = 0;
        return;

    L2: assume(a!=b);
        g = 1;
        return;
}

Global:
(6) g: 0

main:
(1) x: X
(2) y: Y
cmp:
(3) a: A
(4) b: B

Conditions:
(5) (A == B) [3, 4]
(6) (X == Y) [5]
(7) (X != Y) [1, 2]

Contradictory!
```c
int g;
main(int x, int y){
    cmp(x, y);
    assume(!g);
    assume(x != y)
    assert(0);
}

void cmp (int a , int b) {
    Goto L1, L2
    L1: assume(a==b);
        g = 0;
        return;
    L2: assume(a!=b);
        g = 1;
        return;
}

Global:
(6) g:  0
main:
(1) x: X
(2) y: Y
cmp:
(3) a: A
(4) b: B
Conditions:
(5)(A == B)  [3, 4]
(6)(X == Y)  [5]
(7) (X != Y) [1, 2]
Contradictory!
```
int g;

main(int x, int y) {
  cmp(x, y);
  assume(!g);
  assume(x != y)
  assert(0);
}

void cmp (int a , int b) {
  Goto L1, L2
  L1: assume(a==b);
      g = 0;
      return;
  L2: assume(a!=b);
      g = 1;
      return;
}

Predicates after simplification:
\{ x == y, a == b \}
Refinement Using Interpolants

Given formulas $\psi^-, \psi^+ \text{ s.t. } \psi^- \land \psi^+ \text{ is unsatisfiable}

An interpolant for $\psi^-, \psi^+$ is a formula $\Phi$ s.t.

1. $\psi^-$ implies $\Phi$
2. $\Phi$ has symbols common to $\psi^-, \psi^+$
3. $\Phi \land \psi^+$ is unsatisfiable

$\Phi$ computable from proof of unsat. of $\psi^- \land \psi^+$

[Henzinger-Jhala-Majumdar-McMillan POPL 04] give a way to generate predicates for refinement from infeasible traces using interpolants
Interpolants: Example (1)

Trace

\( pc_1: \ x = \text{ctr} \)
\( pc_2: \ \text{ctr} = \text{ctr} + 1 \)
\( pc_3: \ y = \text{ctr} \)
\( pc_4: \ \text{assume}(x = i - 1) \)
\( pc_5: \ \text{assume}(y \neq i) \)

Trace Formula

\( x_1 = \text{ctr}_0 \)
\( \land \ \text{ctr}_1 = \text{ctr}_0 + 1 \)
\( \land \ y_1 = \text{ctr}_1 \)
\( \land \ x_1 = i_0 - 1 \)
\( \land \ y_1 \neq i_0 \)

Predicate Map

\( pc_2: \ x = \text{ctr} \)

Cut + Interpolate at each point

Pred. Map: \( pc_i \mapsto \text{Interpolant from cut i} \)
### Interpolants: Example (2)

<table>
<thead>
<tr>
<th>Trace</th>
<th>Trace Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pc_1): (x = \text{ctr})</td>
<td>(x_1 = \text{ctr}_0)</td>
</tr>
<tr>
<td>(pc_2): (\text{ctr} = \text{ctr} + 1)</td>
<td>(\land \text{ctr}_1 = \text{ctr}_0 + 1)</td>
</tr>
<tr>
<td>(pc_3): (y = \text{ctr})</td>
<td>(\land y_1 = \text{ctr}_1)</td>
</tr>
<tr>
<td>(pc_4): assume((x = i - 1))</td>
<td>(\land x_1 = i_0 - 1)</td>
</tr>
<tr>
<td>(pc_5): assume((y \neq i))</td>
<td>(\land y_1 \neq i_0)</td>
</tr>
</tbody>
</table>

- Cut + Interpolate at each point
- Pred. Map: \(pc_i \mapsto \text{Interpolant from cut } i\)

**Predicate Map**
- \(pc_2\): \(x = \text{ctr}\)
- \(pc_3\): \(x = \text{ctr} - 1\)
Interpolants: Example (3)

- **Trace**
  - $pc_1$: $x = \text{ctr}$
  - $pc_2$: $\text{ctr} = \text{ctr} + 1$
  - $pc_3$: $y = \text{ctr}$
  - $pc_4$: assume($x = i-1$)
  - $pc_5$: assume($y \neq i$)

- **Trace Formula**
  - $x_1 = \text{ctr}_0$
  - $\land \text{ctr}_1 = \text{ctr}_0 + 1$
  - $\land y_1 = \text{ctr}_1$
  - $\land x_1 = i_0 - 1$
  - $\land y_1 \neq i_0$

- **Predicate Map**
  - $pc_2$: $x = \text{ctr}$
  - $pc_3$: $x = \text{ctr}-1$
  - $pc_3$: $y = x+1$

- **Cut + Interpolate at each point**

- **Pred. Map:** $pc_i \mapsto$ Interpolant from cut i

- **Interpolate**
  - $\psi^-$
  - $\psi^+$

- $y_1 = x_1 + 1$
Interpolants: Example (4)

Trace

$pc_1$: $x = ctr$

$pc_2$: $ctr = ctr + 1$

$pc_3$: $y = ctr$

$pc_4$: $\text{assume}(x = i-1)$

$pc_5$: $\text{assume}(y \neq i)$

Trace Formula

$x_1 = ctr_0$

$\land ctr_1 = ctr_0 + 1$

$\land y_1 = ctr_1$

$\land x_1 = i_0 - 1$

$\land y_1 \neq i_0$

Predicate Map

$pc_2$: $x = ctr$

$pc_3$: $x = ctr-1$

$pc_4$: $y = x+1$

$pc_5$: $y = i$

• Cut + Interpolate at each point

• Pred. Map: $pc_i \mapsto \text{Interpolant from cut i}$
Interpolants: Example (5)

Trace

\[ pc_1: x = \text{ctr} \]
\[ pc_2: \text{ctr} = \text{ctr} + 1 \]
\[ pc_3: y = \text{ctr} \]
\[ pc_4: \text{assume}(x = i-1) \]
\[ pc_5: \text{assume}(y \neq i) \]

Trace Formula

\[ x_1 = \text{ctr}_0 \]
\[ \land \text{ctr}_1 = \text{ctr}_0 + 1 \]
\[ \land y_1 = \text{ctr}_1 \]
\[ \land x_1 = i_0 - 1 \]
\[ \land y_1 \neq i_0 \]

Predicate Map

\[ pc_2: x = \text{ctr} \]
\[ pc_3: x = \text{ctr}-1 \]
\[ pc_4: y = x+1 \]
\[ pc_5: y = i \]

Theorem: Predicate map makes trace abstractly infeasible
Generalization of refinement predicates

main()
{
    int x;
    x = 0;
    while(*) {
        x++;
    }
    assert(x >= 0);
}
Areas for future work

• Abstraction-refinement loop with richer models (than just booleans) for solving the progress problem with pointers.

• Combination of over-approximation and under-approximation based methods to avoid refining in irrelevant places in the program.
Synergy: A new algorithm for property checking, FSE '06

Bhargav Gulavani, Thomas Henzinger, Yamini Kannan, Aditya Nori, Sriram Rajamani
Problem statement

• Check if a program satisfies a given safety property:
  – API usage rules
  – Protocols on objects

• Interesting programs have infinite state spaces ranging over infinite domains
  – This problem in general is undecidable
Two approaches to property checking

- **Testing**: find inputs and executions that demonstrate effectively violations of a property
- **Verification**: find a proof that all executions of the program satisfy a property
Tests: presence of bugs

```c
void foo(int a)
{
  0: i = 0;
  1: c = 0;
  2: while (i < 1000) {
      3: c = c + i;
      4: i = i + 1;
  }
  5: assume (a <= 0);
  6: assert (false);
}
```

(a = -5)
Proofs: absence of bugs

```c
void foo(int y1, int y2)
{
0:   state = 1;
1:   if (y1) {
2:     x0 = x0 + 1;
   } 
else {
3:     x0 = x0 - 1;
   }
4:   if (y2) {
5:     x1 = x1 + 1;
   } 
else {
6:     x1 = x1 - 1;
   }
7:   assert (state == 1);
}
```

Exponential number of tests required

Linear proof exists!
Key insights

• **Testing works** when errors are easy to find and is inefficient for finding proofs

• **Verification works** when proofs are easy to find and is inefficient for finding errors
Questions

• Can we combine “systematically” testing with verification?

• How does one generate/direct test cases?
  – Can abstraction help?

• Given a spurious abstract error trace, how does one perform refinement?
  – Can testing help?
Solution: Synergy

• Combines under- and over-approximation reasoning (testing and verification) of programs.

• Unifies several disparate existing algorithms in the literature:
  a) Counterexample driven refinement approaches for verification (SLAM, BLAST)
  b) Directed testing approaches (DART)
  c) Partition refinement algorithms (Lee-Yannakakis, Paige-Tarjan)
The Synergy Algorithm

- Simultaneously perform testing and proving
- Use in-progress proof to guide testing
- Use in-progress tests to guide proving
Example(1)

void foo(int y)
{
  1:   do {
  2:     lock();
  3:     x = y;
  4:     if (*) {
  5:       unlock();
  6:       y = y + 1;
  7:     } while (x != y);
  8:     unlock();
}
Example(1)

\[ \tau = (0, 1, 2, 3, 4, 7, 8, 9) \]

\[ y = 1 \]

```cpp
void foo(int y)
{
    1:    do {
        2:        lock();
        3:        x = y;
        4:        if (*) {
            5:            unlock();
            6:            y = y + 1;
        }
    7:    } while (x != y);
    8:    unlock();
}
```
void foo(int y)
{
  do {
    lock();
    x = y;
    if (*) {
      unlock();
      y = y + 1;
    }
  } while (x != y);
  unlock();
}
void foo(int y)
{
  do {
    lock();
    x = y;
    if (*) {
      unlock();
      y = y + 1;
    }
  } while (x != y);
  unlock();
}

τ = (0, 1, 2, 3, 4, 7, 8, 9)

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>y₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock.state</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>⊥</td>
<td></td>
</tr>
</tbody>
</table>

symbol table
Symbolic execution

\[ \tau = (0, 1, 2, 3, 4, 7, 8, 9) \]

void foo(int y)
{
  do {
    lock();
    x = y;
    if (*) {
      unlock();
      y = y + 1;
    }
  } while (x != y);
  unlock();
}
Symbolic execution

```c
void foo(int y)
{
    do {
        lock();
        x = y;
        if (*) {
            unlock();
            y = y + 1;
        }
    } while (x != y);
    unlock();
}

\[ \tau = (0, 1, 2, 3, 4, 7, 8, 9) \]
```

symbol table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>( y_0 )</td>
</tr>
<tr>
<td>lock.state</td>
<td>L</td>
</tr>
<tr>
<td>( x )</td>
<td>( y_0 )</td>
</tr>
</tbody>
</table>
void foo(int y)
{
    do {
        lock();
        x = y;
        if (*) {
            unlock();
            y = y + 1;
        }
    } while (x != y);
    unlock();
}

\[ \tau = (0, 1, 2, 3, 4, 7, 8, 9) \]

<table>
<thead>
<tr>
<th>symbol table</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
</tr>
<tr>
<td>lock.state</td>
</tr>
<tr>
<td>( x )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x = y) = (y_0 = y_0) = T)</td>
</tr>
</tbody>
</table>
Symbolic execution

```c
void foo(int y)
{
  do {
    lock();
    x = y;
    if (*) {
      unlock();
      y = y + 1;
    }
  } while (x != y);
  unlock();
}
```

symbol table

<table>
<thead>
<tr>
<th></th>
<th>y0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>y0</td>
</tr>
<tr>
<td>lock.state</td>
<td>L</td>
</tr>
<tr>
<td>x</td>
<td>y0</td>
</tr>
</tbody>
</table>

constraints

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = y) = (y0 = y0 ) = T</td>
</tr>
<tr>
<td>(lock.state != L) = (L != L) = F</td>
</tr>
</tbody>
</table>

\[ \tau = (0,1,2,3,4,7,8,9) \]
Refine proof?

- Split region using preimage operation:
  \[ \text{Pre}(S_k) = \{ s \in \Sigma \mid \exists s' \in S_k . s \rightarrow s' \} \]

- For example,
  \[ \text{Pre}(x := x+1, x<5) = (x+1 < 5) = (x<4) \]

- For this example,
  \[ \text{Pre}(\text{assume(lock.state} \neq L), \text{true}) = (\text{lock.state} \neq L) \]
Example(1)

void foo(int y)
{
  1:   do {
  2:     lock();
  3:     x = y;
  4:     if (*) {
  5:       unlock();
  6:       y = y + 1;
  7:     } while (x != y);
  8:   } unlock();
}

split <pc=8> into two regions wrt p= (lock.state != L)
Can extend test beyond frontier?

Input: Program P
Property ψ

Construct random tests
Construct initial proof

Test succeeded?

Proof succeeded?

τ = error path in failed proof
f = frontier of error path

Can extend test beyond frontier?

Refine proof

void foo(int y)
{
1:   do {
2:     lock();
3:     x = y;
4:     if (*) {
5:       unlock();
6:       y = y + 1;
7:     } while (x != y);
8:   unlock();
}
Correct, the program is

```c
void foo(int y)
{
  1:   do {
  2:     lock();
  3:     x = y;
  4:     if (*) {
  5:       unlock();
  6:       y = y + 1;
  7:     } while (x != y);
  8:   unlock();
}
```
Input: Program P Property ψ

Construct random tests
Construct initial proof

Test succeeded?
yes
Bug!

no
Proof succeeded?
yes
Proof!

no
τ = error path in failed proof
f = frontier of error path

Can extend test beyond frontier?
yes

no
Refine proof

void foo(int a)
{
0: i = 0;
1: c = 0;
2: while (i < 1000)
{
3: c = c + i;
4: i = i + 1;
}
5: if(a <= 0)
6: error();
}
Example(2)

Input: Program P
Property $\psi$

Construct random tests
Construct initial proof

Test succeeded?
yes Bug!
no

Proof succeeded?
yes Proof!
no

$\tau = \text{error path in failed proof}$
$f = \text{frontier of error path}$

Can extend test beyond frontier?
yes
no Refine proof

void foo(int $a$)
{
  0: $i = 0$;
  1: $c = 0$;
  2: while ($i < 1000$) {
    3: $c = c + i$;
    4: $i = i + 1$;
  }
  5: if($a <= 0$)
    6: error();
}
Symbolic execution

```c
void foo(int a)
{
    0:    i = 0;
    1:    c = 0;
    2:    while (i < 1000)
    {  
    3:        c = c + i;
    4:        i = i + 1;
    }
    5:    if(a <= 0)
    6:        error();
}
```

Symbol table:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>⊥</td>
</tr>
<tr>
<td>i</td>
<td>⊥</td>
</tr>
<tr>
<td>c</td>
<td>⊥</td>
</tr>
</tbody>
</table>
void foo(int a)
{
0:  i = 0;
1:  c = 0;
2:  while (i < 1000)
   {
3:    c = c + i;
4:    i = i + 1;
   }
5:  if(a <= 0)
6:    error();
}

τ=(0,1,2,(3,4,2)^1000,5,6)

symbol table

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic execution

void foo(int a)
{
  0: i = 0;
  1: c = 0;
  2: while (i < 1000)
   {
    3: c = c + i;
    4: i = i + 1;
   }
  5: if(a <= 0)
  6:     error();
}

\[ \tau = (0, 1, 2, (3, 4, 2)^{1000}, 5, 6) \]

<table>
<thead>
<tr>
<th>symbol table</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i &lt; 1000) = (0 &lt; 1000) = \text{T}</td>
</tr>
</tbody>
</table>
void foo(int a) {
    0:  i = 0;
    1:  c = 0;
    2:  while (i < 1000) {
        3:  c = c + i;
        4:  i = i + 1;
    } 
    5:  if(a <= 0) 
    6:      error();
}
void foo(int a)
{
  0:  i = 0;
  1:  c = 0;
  2:  while (i < 1000)
      {
  3:    c = c + i;
  4:    i = i + 1;
      }
  5:  if(a <= 0)
  6:    error();
}

\( \tau = (0,1,2,(3,4,2)^{1000},5,6) \)
Example(2)

Input:
Program P
Property $\psi$

Construct random tests
Construct initial proof

Test succeeded?
yes

Bug!

Proof succeeded?
yes

Proof!

no

$\tau =$ error path in failed proof
$f =$ frontier of error path

Can extend test beyond frontier?
yes

no

Refine proof

Input:
Program P
Property $\psi$

Construct random tests
Construct initial proof

Test succeeded?
yes

Bug!

Proof succeeded?
yes

Proof!

no

$\tau =$ error path in failed proof
$f =$ frontier of error path

Can extend test beyond frontier?
yes

no

Refine proof

void foo(int $a$)
{
  0: $i = 0$;
  1: $c = 0$;
  2: while ($i < 1000$) {
  3:     $c = c + i$;
  4:     $i = i + 1$;
  
  5:      if($a <= 0$)  
  6:         error();
}
Soundness and Termination

• **Theorem:** Suppose we run Synergy on any program $P = \langle \Sigma, \sigma^I, \rightarrow \rangle$ and property $\psi$
  – If Synergy returns (“pass”, $\Sigma_{\approx}$), then the abstract program $P_{\approx} = \langle \Sigma_{\approx}, \sigma^I_{\approx}, \rightarrow_{\approx} \rangle$ simulates $P$, and thus is a proof that $P$ does not reach $\psi$
  – If Synergy returns (“fail”, $t$), then $t$ is an error trace

• **Theorem:** If $P = \langle \Sigma, \sigma^I, \rightarrow \rangle$ has a finite bisimulation quotient, then Synergy terminates

• **Theorem:** Synergy terminates in strictly more cases than the Lee-Yannakakis algorithm
The Future

• Only integer domains with linear arithmetic have been considered
  – Incorporate more expressive domains
  – New techniques for discovering predicates
  – Interprocedural analysis

• Check properties of device drivers

• A more comprehensive analysis of combining over- and under-approximations of programs