Interprocedural BMC

With Function Summaries

Manasij Mukherjee
Mandayam Srivas
Chennai Mathematical Institute

What is BMC?

- Bounded Model Checking
  - *Can we find a counterexample in k steps?*
    - Unwind loops up to k iterations
    - Inline all functions
- Advantage
  - Simple implementation
  - Fast in general
- Disadvantage
  - Large verification conditions, might not scale

Function Summaries

- Compact representation
  - Input-Output relation
  - Possibly an abstraction
- Summarize from different perspectives
  - Transition relation
  - Input assumption directed
  - Error property (assertion) directed

Function:

```
int foo(int x) {
  return x + 1;
}
```

Summary:

```
foo_ret == x + 1
```

Function Summaries - Example

Function:

```
int foo(int x) {
  int z = 10;
  if (x < z) {
    return x + z;
  } else {
    return x - z;
  }
}
```

Summary :

```
foo_ret == ite(x < 10, x + 10, x - 10)
```

Detour - SSA Representation in 2ls

```
int foo(int x) {
  int z = 10;
  if (x < z) {
    return x + z;
  } else {
    return x - z;
  }
}
```

```
(E) $guard#0 == TRUE

(E) z#1 == 10

(E) $cond#2 == x >= 10

(E) foo#return_value#3 == 10 + x
(E) $guard#3 == (!$cond#2 && $guard#0)

(E) $cond#5 == TRUE

(E) $cond#6 == TRUE
(E) $guard#6 == FALSE

(E) foo#return_value#7 == -10 + x
(E) $guard#7 == ($cond#2 && $guard#0)
```

```
(E) $cond#9 == TRUE

(E) $guard#10 == ($cond#6 &&
$guard#6)

(E) foo#return_value#11 ==
nondet_symbol(ssa::nondet11.1)

(E) foo#return_value#phi12 ==
($guard#10 ?
foo#return_value#11 : ($cond#9
&& $guard#7 ?
foo#return_value#7 :
foo#return_value#3))
(E) $guard#12 == ($cond#5 &&
$guard#3 || $cond#9 &&
$guard#7 || $guard#10)
```

Function Summaries - Simplification

Function:

```
int foo(int x) {
  if (x < 10) {
    x++;
  }
  return x;
}
```

Summary :

```
foo_ret == ite(x < 10, x + 1, x)
```

Example:

```
int main() {
  for (int i = 0; i < 10; ++i) {
    print(foo(i));
  }
}
```

Summary

```
foo_ret == x + 1
```

Comparison with inlining - I

Example:

```
int main() {
  for (int i = 0; i < 10; ++i) {
    print(foo(i));
  }
}
```

Function Body for foo:

```
int foo(int x) {
  if (x < 10) {
    x++;
  }
  return x;
}
```

Unwinding

```
int main(){
  if (i < 10) {
    print(foo(i));
    ++i;
    if (i < 10) {
      print(foo(i));
      ++i;
      if (i < 10) {
        print(foo(i));
        ++i;
        <up to K times>
      }
    }
  }
}
```

Comparison with inlining - II

```
int main(){
  if (i < 10) {
    int ret;
    int x = i;
    if (i < 10) {
      ret = i + 1;
    } else {
      ret = i;
    }
    print(ret);
    ++i;
    if (i < 10) {
      print(foo(i));
      ++i;
      if (i < 10) {
        print(foo(i));
        ++i;
        <up to K times>
      }
    }
  }
}
```

Inlining

Function:

```
int foo(int x) {
  if (x < 10) {
    return x + 1;
  } else {
    return x;
  }
}
```

Summary

```
int main(){
  if (i < 10) {
    int ret = i + 1;
    print(ret);
    ++i;
    if (i < 10) {
      print(foo(i));
      ++i;
      if (i < 10) {
        print(foo(i));
        ++i;
        <up to K times>
      }
    }
  }
}
```

Backward Summaries

- Propagate backwards from assertions
  - Weakest Precondition
  - Can simplify to TRUE/FALSE
    - Depending on context

Function:

```
void foo(int x) {
  int twice = x + x;
  assert (twice > x);
}
```

WP:

```
~(x + x) > x
```

- Simulate effect on error conditions
  - Property directed
  - Smaller verification conditions

Backward Summaries -- Simplification I

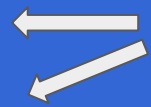
Function

```
void foo(int x) {
    int twice = x + x;
    assert (twice > x);
}
```

WP := `~ (x + x) > x`

Example

```
int main() {
  if (x == 1) {
    foo(x - 1);
  } else if (x < 0) {
      foo(x);
  }
}
```



Context

Context Sensitive WP := **TRUE**

Backward Summaries -- Simplification II

Function:

```
void foo(int x) {
  if (x > 0) {
    int twice = x + x;
    assert(twice > x);
  } else {
     for (int i = 0; i < x; ++i) {
       print(i);
     }
  }
}
```

No assert here

Example:

```
int main() {
  int x = nondet_int();
  foo(x);
}
```

Pruned WP:

```
~ ((x > 0) -> (x + x > x))
```

High level view

- Original algorithm from
  - Compositional Safety Refutation Techniques. ATVA 2017
    - Kumar Madhukar, Peter Schrammel, Mandayam K. Srivas
  - Implemented in the 2ls tool in the CBMC/CProver framework.
- Summary composition
  - Replace function call sites with placeholder predicates
  - Backward OR Forward traversal of CFG in SSA
  - Generate a new summary or use a cached one for each placeholder.
  - Simplify when it makes sense

Under-approximation of Backward Summaries

- Leave out terms
  - From ite
  - From or
- Refinement loop
  - When UNSAT

Function:

```
int foo(int x) {
  if (x < 10) {
    return x + 1;
  } else {
    return x;
  }
}
```

Summary:

```
foo_ret == ite(x < 10, x + 1, x)
```

```
Under-approximation:
x < 10 AND foo_ret == x + 1
```

Caching Summaries

- Multiple contexts
  - C1 → S1 AND C2 → S2 AND ...
  - Example: $i_1 < 10 \rightarrow (ret_1 == x)$ AND $(i_1 < 11) \rightarrow (ret_1 == x)$
- Subsumption
  - Can skip summary computation
    - Iff C2 → C1
    - Tradeoff : Semantic vs Syntactic subsumption check
  - $i_1 < 10 \rightarrow (ret_1 == x)$

Hybrid Approach

- Compute forward summaries
- Cache for similar contexts
  - Example: Calls within a for loop
- Weakest Precondition
  - Use cached forward summaries

```
for (int i = 0; i < n; ++i) {
  result = foo(i);
}
```

Hybrid Approach - Example

Example:

```
int main() {
  int result;
  for (int i = 0; i < 10; ++i) {
   result = foo(i);
  }
  assert (result == 10);
}
```

Function Body for foo:

```
int foo(int x) {
  if (x < 10) {
    x++;
  }
  return x;
}
```

```
int main(){
  Int result;
  if (i < 10) {
    int ret = i + 1;
    result = ret;
    ++i;
    if (i < 10) {
      int ret = i + 1;
      result = ret;
      ++i;
      if (i < 10) {
        int ret = i + 1;
        result = ret;
        ++i;
        <up to K times>
      }
    }
  }
  assert (result == 10);
}
```

Propagate

Thank You