IA159 Formal Methods for Software Analysis

Bounded Model Checking, *k*-Induction

Jan Strejček

Faculty of Informatics
Masaryk University

Focus and sources

focus

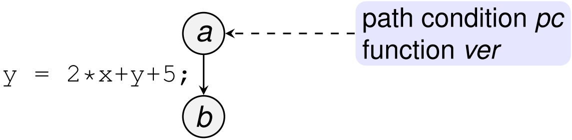
- memoryless version of symbolic execution
- bounded model checking (BMC)
- *k*-induction

source

- A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer: Software Verification Using *k*-Induction, SAS 2011.

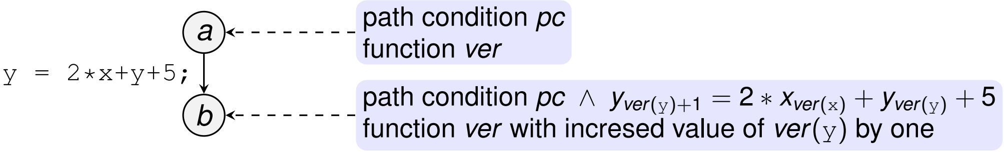
Memoryless version of symbolic execution

- does not use symbolic memory, the assignments are stored to path condition
- to do that, we need to consider another instance of each variable after each assignment to it and remeber its current instance
- let *ver* : $Vars \to \mathbb{N}$ be the function keeping the current instances
- initially, $ver(\mathtt{x}) = 1$ for each $x \in Vars$



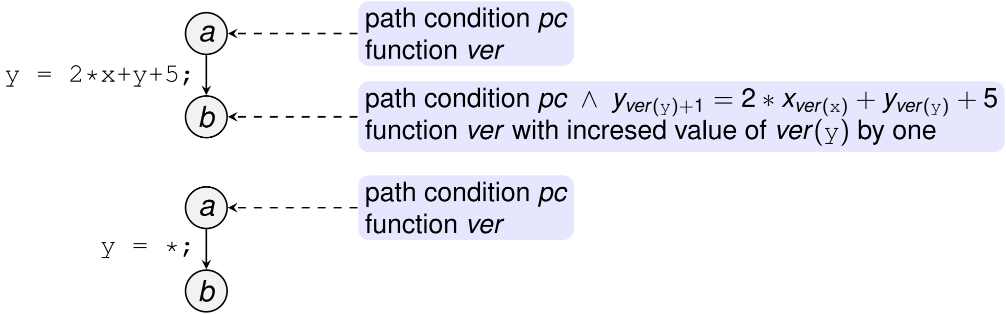
Memoryless version of symbolic execution

- does not use symbolic memory, the assignments are stored to path condition
- to do that, we need to consider another instance of each variable after each assignment to it and remeber its current instance
- let *ver* : $\mathit{Vars} \to \mathbb{N}$ be the function keeping the current instances
- initially, $\mathit{ver}(\mathtt{x}) = 1$ for each $x \in \mathit{Vars}$



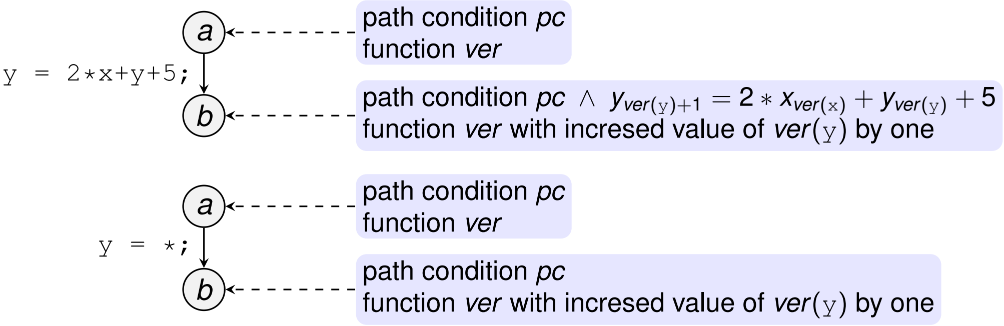
Memoryless version of symbolic execution

- does not use symbolic memory, the assignments are stored to path condition
- to do that, we need to consider another instance of each variable after each assignment to it and remeber its current instance
- let *ver* : $Vars \to \mathbb{N}$ be the function keeping the current instances
- initially, $ver(\mathtt{x}) = 1$ for each $x \in Vars$



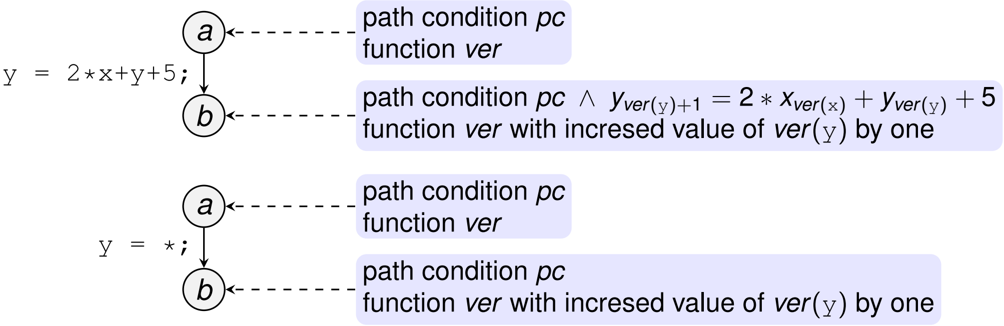
Memoryless version of symbolic execution

- does not use symbolic memory, the assignments are stored to path condition
- to do that, we need to consider another instance of each variable after each assignment to it and remeber its current instance
- let *ver* : $Vars \to \mathbb{N}$ be the function keeping the current instances
- initially, $ver(\mathtt{x}) = 1$ for each $x \in Vars$



Memoryless version of symbolic execution

- does not use symbolic memory, the assignments are stored to path condition
- to do that, we need to consider another instance of each variable after each assignment to it and remeber its current instance
- let *ver* : *Vars* $\to \mathbb{N}$ be the function keeping the current instances
- initially, $ver(\mathtt{x}) = 1$ for each $x \in Vars$



- symbolic execution of branching statements is modified similarly

Bounded model checking (BMC)

Bounded model checking (BMC)

- a technique for finding bugs
- proves correctness only very rarely
- similar to memoryless symbolic execution, but creates one SMT query

Bounded model checking (BMC)

- a technique for finding bugs
- proves correctness only very rarely
- similar to memoryless symbolic execution, but creates one SMT query
- workflow

```
unwind all loops and recursion k-times for a given bound k
compute the error reaching formula φ from the unwound program
if φ is satisfiable then
    return bug found
else
    compute the bound reaching formula ψ from the unwound program
    if the ψ is unsatisfiable then
        return the program is correct
    else
        return unknown (increase bound and start again)
```

Example

original program

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned char s = 0;
unsigned int i = 0;
while (i < n) {
  v = input();
  s += v;
  ++i;
}
assert(s >= v);
```

Example

original program

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned char s = 0;
unsigned int i = 0;
while (i < n) {
  v = input();
  s += v;
  ++i;
}
assert(s >= v);
```

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned char s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

Example

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned char s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

Example

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned char s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

error reaching formula φ

$$
\begin{aligned}
& n_1 > 0 \;\wedge\; v_1 = 0 \;\wedge\; s_1 = 0 \;\wedge\; i_1 = 0 \;\wedge \\
& \wedge\; ((i_1 \geq n_1 \;\wedge\; s_1 < v_1) \;\vee \\
& \quad \vee\; (i_1 < n_1 \;\wedge\; s_2 = s_1 + v_2 \;\wedge\; i_2 = i_1 + 1 \;\wedge \\
& \quad\quad \wedge\; ((i_2 \geq n_1 \;\wedge\; s_2 < v_2) \;\vee \\
& \quad\quad \vee\; (i_2 < n_1 \;\wedge\; s_3 = s_2 + v_3 \;\wedge\; i_3 = i_2 + 1 \;\wedge \\
& \quad\quad\quad \wedge\; ((i_3 \geq n_1 \;\wedge\; s_3 < v_3) \;\vee \\
& \quad\quad\quad \vee\; (i_3 < n_1 \;\wedge\; s_4 = s_3 + v_4 \;\wedge\; i_4 = i_3 + 1 \;\wedge \\
& \quad\quad\quad\quad \wedge\; i_4 \geq n_1 \;\wedge\; s_4 < v_4))))))
\end{aligned}
$$

Example

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned char s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

error reaching formula φ

$$
\begin{aligned}
& n_1 > 0 \ \wedge \ v_1 = 0 \ \wedge \ s_1 = 0 \ \wedge \ i_1 = 0 \ \wedge \\
& \wedge \ ((i_1 \geq n_1 \ \wedge \ s_1 < v_1) \ \vee \\
& \quad \vee \ (i_1 < n_1 \ \wedge \ s_2 = s_1 + v_2 \ \wedge \ i_2 = i_1 + 1 \ \wedge \\
& \quad\quad \wedge \ ((i_2 \geq n_1 \ \wedge \ s_2 < v_2) \ \vee \\
& \quad\quad \vee \ (i_2 < n_1 \ \wedge \ s_3 = s_2 + v_3 \ \wedge \ i_3 = i_2 + 1 \ \wedge \\
& \quad\quad\quad \wedge \ ((i_3 \geq n_1 \ \wedge \ s_3 < v_3) \ \vee \\
& \quad\quad\quad \vee \ (i_3 < n_1 \ \wedge \ s_4 = s_3 + v_4 \ \wedge \ i_4 = i_3 + 1 \ \wedge \\
& \quad\quad\quad\quad \wedge \ i_4 \geq n_1 \ \wedge \ s_4 < v_4))))))
\end{aligned}
$$

satisfiable

variable types are considered

bitvector arithmetic is used

$n_1 = 2$

$v_1 = 0$	$s_1 = 0$	$i_1 = 0$
$v_2 = 224$	$s_2 = 224$	$i_2 = 1$
$v_3 = 63$	$s_3 = 31$	$i_3 = 2$

bug found!

Example 2

original program

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
while (i < n) {
  v = input();
  s += v;
  ++i;
}
assert(s >= v);
```

Example 2

original program

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
while (i < n) {
  v = input();
  s += v;
  ++i;
}
assert(s >= v);
```

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

Example 2

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

Example 2

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

error reaching formula φ

$$
\begin{aligned}
& n_1 > 0 \ \wedge \ v_1 = 0 \ \wedge \ s_1 = 0 \ \wedge \ i_1 = 0 \ \wedge \\
& \wedge \ ((i_1 \geq n_1 \ \wedge \ s_1 < v_1) \ \vee \\
& \quad \vee \ (i_1 < n_1 \ \wedge \ s_2 = s_1 + v_2 \ \wedge \ i_2 = i_1 + 1 \ \wedge \\
& \quad\quad \wedge \ ((i_2 \geq n_1 \ \wedge \ s_2 < v_2) \ \vee \\
& \quad\quad \vee \ (i_2 < n_1 \ \wedge \ s_3 = s_2 + v_3 \ \wedge \ i_3 = i_2 + 1 \ \wedge \\
& \quad\quad\quad \wedge \ ((i_3 \geq n_1 \ \wedge \ s_3 < v_3) \ \vee \\
& \quad\quad\quad \vee \ (i_3 < n_1 \ \wedge \ s_4 = s_3 + v_4 \ \wedge \ i_4 = i_3 + 1 \ \wedge \\
& \quad\quad\quad\quad \wedge \ i_4 \geq n_1 \ \wedge \ s_4 < v_4))))))
\end{aligned}
$$

Example 2

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

error reaching formula φ

$$
\begin{aligned}
& n_1 > 0 \ \wedge \ v_1 = 0 \ \wedge \ s_1 = 0 \ \wedge \ i_1 = 0 \ \wedge \\
& \wedge \ ((i_1 \geq n_1 \ \wedge \ s_1 < v_1) \ \vee \\
& \quad \vee \ (i_1 < n_1 \ \wedge \ s_2 = s_1 + v_2 \ \wedge \ i_2 = i_1 + 1 \ \wedge \\
& \quad\quad \wedge \ ((i_2 \geq n_1 \ \wedge \ s_2 < v_2) \ \vee \\
& \quad\quad \vee \ (i_2 < n_1 \ \wedge \ s_3 = s_2 + v_3 \ \wedge \ i_3 = i_2 + 1 \ \wedge \\
& \quad\quad\quad \wedge \ ((i_3 \geq n_1 \ \wedge \ s_3 < v_3) \ \vee \\
& \quad\quad\quad \vee \ (i_3 < n_1 \ \wedge \ s_4 = s_3 + v_4 \ \wedge \ i_4 = i_3 + 1 \ \wedge \\
& \quad\quad\quad\quad \wedge \ i_4 \geq n_1 \ \wedge \ s_4 < v_4))))))
\end{aligned}
$$

unsatisfiable

bound reaching formula ψ

$$
\begin{aligned}
& n_1 > 0 \ \wedge \ v_1 = 0 \ \wedge \ s_1 = 0 \ \wedge \ i_1 = 0 \ \wedge \\
& \wedge \ i_1 < n_1 \ \wedge \ s_2 = s_1 + v_2 \ \wedge \ i_2 = i_1 + 1 \ \wedge \\
& \wedge \ i_2 < n_1 \ \wedge \ s_3 = s_2 + v_3 \ \wedge \ i_3 = i_2 + 1 \ \wedge \\
& \wedge \ i_3 < n_1 \ \wedge \ s_4 = s_3 + v_4 \ \wedge \ i_4 = i_3 + 1 \ \wedge \\
& \wedge \ i_4 < n_1
\end{aligned}
$$

Example 2

unwound program for $k = 3$

```
unsigned char n = input();
if (n == 0) {return 0};
unsigned char v = 0;
unsigned int s = 0;
unsigned int i = 0;
if (i < n) {
  v = input();
  s += v;
  ++i;
  if (i < n) {
    v = input();
    s += v;
    ++i;
    if (i < n) {
      v = input();
      s += v;
      ++i;
      if (i < n) {
        bound_reached();}}}}
assert(s >= v);
```

error reaching formula φ

$$
\begin{aligned}
& n_1 > 0 \ \wedge \ v_1 = 0 \ \wedge \ s_1 = 0 \ \wedge \ i_1 = 0 \ \wedge \\
& \wedge \ ((i_1 \geq n_1 \ \wedge \ s_1 < v_1) \ \vee \\
& \quad \vee \ (i_1 < n_1 \ \wedge \ s_2 = s_1 + v_2 \ \wedge \ i_2 = i_1 + 1 \ \wedge \\
& \quad\quad \wedge \ ((i_2 \geq n_1 \ \wedge \ s_2 < v_2) \ \vee \\
& \quad\quad \vee \ (i_2 < n_1 \ \wedge \ s_3 = s_2 + v_3 \ \wedge \ i_3 = i_2 + 1 \ \wedge \\
& \quad\quad\quad \wedge \ ((i_3 \geq n_1 \ \wedge \ s_3 < v_3) \ \vee \\
& \quad\quad\quad \vee \ (i_3 < n_1 \ \wedge \ s_4 = s_3 + v_4 \ \wedge \ i_4 = i_3 + 1 \ \wedge \\
& \quad\quad\quad\quad \wedge \ i_4 \geq n_1 \ \wedge \ s_4 < v_4))))))
\end{aligned}
$$

unsatisfiable

bound reaching formula ψ

$$
\begin{aligned}
& n_1 > 0 \ \wedge \ v_1 = 0 \ \wedge \ s_1 = 0 \ \wedge \ i_1 = 0 \ \wedge \\
& \wedge \ i_1 < n_1 \ \wedge \ s_2 = s_1 + v_2 \ \wedge \ i_2 = i_1 + 1 \ \wedge \\
& \wedge \ i_2 < n_1 \ \wedge \ s_3 = s_2 + v_3 \ \wedge \ i_3 = i_2 + 1 \ \wedge \\
& \wedge \ i_3 < n_1 \ \wedge \ s_4 = s_3 + v_4 \ \wedge \ i_4 = i_3 + 1 \ \wedge \\
& \wedge \ i_4 < n_1
\end{aligned}
$$

satisfiable $\implies$ unknown (bound reachable)

Notes on BMC

- very efficient in finding bugs
- uses a sort of SSA when constructing the formula
- constant propagation can simplify the program and the formula and it can reveal that the bound is unreachable
- implemented for example in CBMC
  - tool for bounded model checking of C and C++ programs
  - supports C89, C99, most of C11 and most extensions of gcc and Visual Studio
  - the winner of SV-COMP 2014
  - `https://www.cprover.org/cbmc/`
  - a version for Java programs called JBMC

k-induction

k-induction

- extension of BMC that can prove correctness more often
- very successful on symbolic transition systems
- to prove program correctness, we show
  1. base case:
     all feasible paths starting in an initial state of length at most k are correct
  2. induction step:
     each feasible path of length $k+1$ that has a correct prefix of length k is also correct

  - if the base case fails, we found a bug
  - if the induction step fails, we can increase k and try again

k-induction

- extension of BMC that can prove correctness more often
- very successful on symbolic transition systems
- to prove program correctness, we show
  1. base case:
     all feasible paths starting in an initial state of length at most k are correct
  2. induction step:
     each feasible path of length $k+1$ that has a correct prefix of length k is also correct

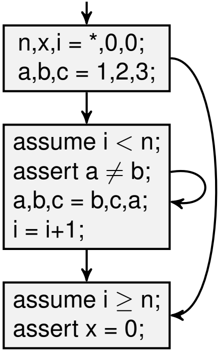
  - if the base case fails, we found a bug
  - if the induction step fails, we can increase k and try again
- the idea can be applied to programs in different ways
  - k-induction on single-loop programs
- k-induction does semantically the same as backward symbolic execution
  - M. Chalupa and J. Strejček: Backward Symbolic Execution with Loop Folding, SAS 2021.

k-induction on single-loop programs

```
n,x,i = *,0,0;
a,b,c = 1,2,3;
while (i < n) {
  assert(a != b);
  a,b,c = b,c,a;
  i++;
}
assert(x = 0);
```

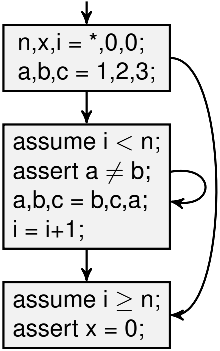
k-induction on single-loop programs

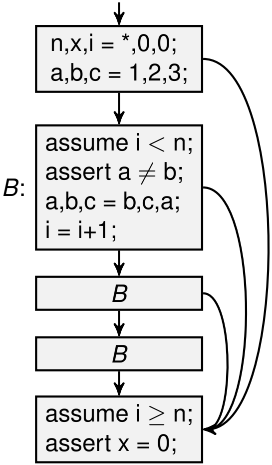
```
n,x,i = *,0,0;
a,b,c = 1,2,3;
while (i < n) {
  assert(a != b);
  a,b,c = b,c,a;
  i++;
}
assert(x = 0);
```



k-induction on single-loop programs

```
n,x,i = *,0,0;
a,b,c = 1,2,3;
while (i < n) {
  assert(a != b);
  a,b,c = b,c,a;
  i++;
}
assert(x = 0);
```

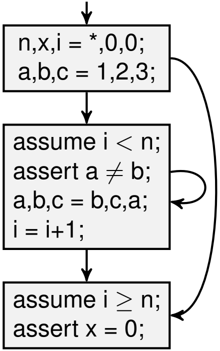


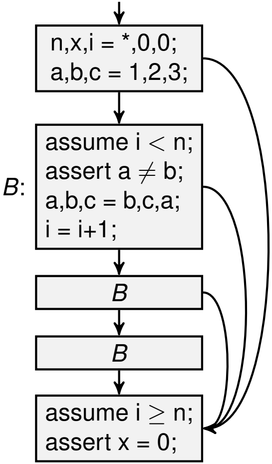


base case (= BMC)
for $k = 3$

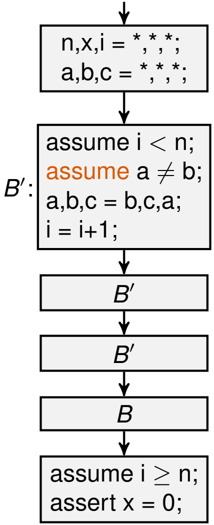
k-induction on single-loop programs

```
n,x,i = *,0,0;
a,b,c = 1,2,3;
while (i < n) {
  assert(a != b);
  a,b,c = b,c,a;
  i++;
}
assert(x = 0);
```





base case (= BMC)
for $k = 3$



induction step for $k = 3$

The End