IA159 Formal Methods for Software Analysis

Symbolic Execution and Applications

Jan Strejček

Faculty of Informatics
Masaryk University

Focus and sources

focus

- symbolic execution
- automated whitebox fuzz testing
- bounded model checking

sources

- J. C. King: *Symbolic Execution and Program Testing*, Communications of ACM, 1976.
- P. Godefroid, M. Y. Levin, and D. Molnar: *Automated whitebox fuzz testing*, NDSS 2008.

Special thanks to Marek Trtík for providing me his slides.

Motivation

```
procedure $sum(a, b, c)$ {
    $x = a + b;$
    $y = b + c;$
    $z = x + y - b;$
    return $z;$
}
```

Testing checks that the program behaves correctly on selected inputs

- $sum(1, 1, 1) =$
- $sum(1, 2, 3) =$
- ...

Motivation

```
procedure $sum(a, b, c)$ {
    $x = a + b$;
    $y = b + c$;
    $z = x + y - b$;
    return $z$;
}
```

Testing checks that the program behaves correctly on selected inputs

- $sum(1, 1, 1) = 3$
- $sum(1, 2, 3) = 6$
- ...

Motivation

```
procedure $sum(a, b, c)$ {
    $x = a + b$;
    $y = b + c$;
    $z = x + y - b$;
    return $z$;
}
```

We can execute the program with symbols $\alpha_1, \alpha_2, \alpha_3$ representing arbitrary input values

- $sum(\alpha_1, \alpha_2, \alpha_3) =$

Motivation

```
procedure $sum(a, b, c)$ {
    $x = a + b$;
    $y = b + c$;
    $z = x + y - b$;
    return $z$;
}
```

We can execute the program with symbols $\alpha_1, \alpha_2, \alpha_3$ representing arbitrary input values

- $sum(\alpha_1, \alpha_2, \alpha_3) = \alpha_1 + \alpha_2 + \alpha_3$

$\rightarrow$ symbolic execution

Symbolic execution semantics in general

Each programming language has an execution semantics describing

- the data objects which program variables may represent
- how statements manipulate data objects
- how control flows through the statements of a program

In symbolic execution semantics

- real data objects can be represented by symbols
- basic operators of the language are extended to accept symbolic input and produce symbolic output

Simple programming language

Consider the following programming language

- all program variables are of type unbounded signed integer
- input can be obtained by procedure parameters, global variables, or read operations
- arithmetic expressions may contain only operators $+, -, *$
- commands:
  - assignment `<var> := <expr>`
  - `goto <label>`
  - `if-then-else` with condition `<expr>` ≥ 0

Semantics of the language

Standard execution semantics

- data objects = signed integers
- ...

Symbolic execution semantics

- besides integers, we can use symbols from the list $\alpha_1, \alpha_2, \alpha_3, \ldots$ to represent some data objects
- the only opportunity to introduce symbolic data objects is as inputs to the program
- the evaluation rules for arithmetic expressions used in assignments and `if` statements must be extended to handle symbolic values
- `goto` works exactly as in normal executions

Extending rules for expressions and `if` statement

Values of expressions and variables are integer polynomials over the symbols $\alpha_1, \alpha_2, \ldots$.

Extending rules for expressions and `if` statement

Values of expressions and variables are integer polynomials over the symbols $\alpha_1, \alpha_2, \ldots$.

Although `if` statement does not change the state of program variables, it plays a key role in the definition of symbolic semantics.

We extend the system state by path condition *pc*, which is a conjunction of inequalities of the form $R \geq 0$ or $\neg(R \geq 0)$, where R is a polynomial over $\alpha_1, \alpha_2, \ldots$.

- *pc* is initially set to *true*
- *pc* can only be modified when executing `if` statements

Intuitively, *pc* accumulates conditions navigating the execution along the current path.

Extending path condition

Let q be an inequality resulting from substituting values of variables into condition of an `if` statement.

Assuming $pc \neq \mathit{false}$, at most one of the following implications can be valid:

(a) $pc \implies q$
(b) $pc \implies \neg q$

A formula φ is valid if and only if $\neg\varphi$ is unsatisfiable. Hence, SMT solvers are used to check validity.

Extending path condition

Let q be an inequality resulting from substituting values of variables into condition of an `if` statement.

Assuming $pc \neq false$, at most one of the following implications can be valid:

(a) $pc \implies q$
(b) $pc \implies \neg q$

A formula φ is valid if and only if $\neg\varphi$ is unsatisfiable. Hence, SMT solvers are used to check validity.

If one implication is valid, then we speak about non-forking execution and pc is not changed.

- If (a) is valid, the execution continues by `then` branch.
- If (b) is valid, the execution continues by `else` branch.

Extending path condition

Let q be an inequality resulting from substituting values of variables into condition of an `if` statement.

Assuming $pc \neq false$, at most one of the following implications can be valid:

(a) $pc \implies q$
(b) $pc \implies \neg q$

A formula φ is valid if and only if $\neg\varphi$ is unsatisfiable. Hence, SMT solvers are used to check validity.

When neither (a) nor (b) is valid, then we speak about forking execution. The current execution forks into two independent ones, since both branches are possible. Path conditions of resulting executions are updated as follows:

- $pc \leftarrow pc \wedge q$ for `then` branch
- $pc \leftarrow pc \wedge \neg q$ for `else` branch

Example

```
procedure $power(x, y)$ {
        $z := 1$;
        $j := 1$;
lab:    if $(y - j \geq 0)$ then {
            $z := z * x$;
            $j := j + 1$;
            goto lab;
        }
        return $z$;
}
```

(draw symbolic execution on the whiteboard)

Path condition is always satisfiable

Clearly, every path condition corresponds exactly to one execution path and vice versa.

Theorem

At each point of every symbolic execution $pc \not\equiv false$.

Proof: Initially, pc is set to *true*. Further, pc is modified only at forking executions, using assignments of the form $pc \leftarrow pc \wedge q$ and $pc \leftarrow pc \wedge \neg q$.

Forking execution implies that $pc \implies \neg q$ is not valid. Hence, $\neg(pc \implies \neg q)$ is satisfiable. As $pc \wedge q \equiv \neg(pc \implies \neg q)$, $pc \wedge q$ is also satisfiable.

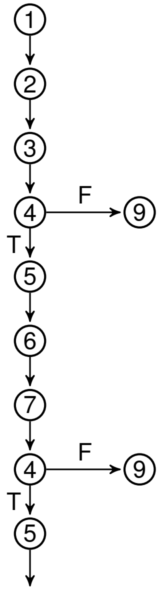
The case $pc \wedge \neg q$ is similar. $\square$

Symbolic execution tree

The execution paths followed during the symbolic execution of a procedure can be expressed by symbolic execution tree.

- executed statement = a node labeled with the statement number
- transition between executed statements = a directed arc connecting the corresponding nodes
- for each forking `if` statement execution there are two outgoing arcs labeled with `T` and `F` for `then` and `else` branch, respectively

Symbolic execution tree for *power*(α_1, α_2)


$$\begin{tikzcd}
1 \arrow[d] & \\
2 \arrow[d] & \\
3 \arrow[d] & \\
4 \arrow[r, "F"] \arrow[d, "T"'] & 9 \\
5 \arrow[d] & \\
6 \arrow[d] & \\
7 \arrow[d] & \\
4 \arrow[r, "F"] \arrow[d, "T"'] & 9 \\
5 \arrow[d] & \\
{} &
\end{tikzcd}$$

Symbolic execution tree

Lemma

For each terminal leaf in the tree there exists particular non-symbolic input, which will trace the same path.

Proof: Every input satisfying the corresponding *pc* trace the same path. As *pc* is always satisfiable, there exists such an input. □

Symbolic execution tree

Lemma

For each terminal leaf in the tree there exists particular non-symbolic input, which will trace the same path.

Proof: Every input satisfying the corresponding *pc* trace the same path. As *pc* is always satisfiable, there exists such an input. $\square$

Lemma

Path conditions associated with any two terminal leaves are distinct, i.e. $pc_1 \land pc_2 \equiv \mathit{false}$.

Proof: The two paths leading from the root to two different terminal nodes have a unique forking node where the paths diverge. At that forking node some q was added to one while $\neg q$ to the other. Since $q \land \neg q \equiv \mathit{false}$, the lemma holds. $\square$

Commutativity

If one normally executes a program with a specific set of integers $\{j_i\}$, the result will be the same as executing it symbolically (using a set of $\{\alpha_i\}$) and then instantiating the symbolic results, i.e., assigning $\{j_i\}$ to $\{\alpha_i\}$.

$$\begin{tikzcd}
P,\{\alpha_i\},\{j_i\} \arrow[r, "\text{symbolic execution}"] \arrow[d, "\text{set parameters to integer values}"] & P(\{\alpha_i\}),\{j_i\} \arrow[d, "\text{substitute into symbolic result}"] \\
P,\{\alpha_i\} \leftarrow \{j_i\} \arrow[r, "\text{standard execution}"] & P(\{j_i\})
\end{tikzcd}$$

Memoryless version of symbolic execution

Applications in verification

Programs can be enriched with `assume(`φ`)` and `assert(`φ`)` statements. When symbolic execution passes through

- `assume(`φ`)`, it executes $pc \leftarrow pc \wedge \varphi$.
- `assert(`φ`)` and $pc \implies \varphi$ is not valid, it reports an error.

With these constructs, symbolic execution can be used with a modification of Floyd's proof method to prove program correctness.

This application is straightforward for any program whose symbolic execution tree is finite.

Practical issues

- deciding validity or satisfiability of formulas can be expensive or even impossible (e.g. for our simple language with unbounded data types)
- in practice, symbolic execution uses expressions and formulas over bitvector theory (operations and relations correspond to CPU instructions, e.g. artihmetic operations with overflows, bitwise operations, etc.), where validity and satisfiability are decidable (but expensive)

Practical issues

variable storage referencing problem

- when i is dependent on input, then $A[i]$ can point to various locations in memory
- unsatisfactory solution:
  handle $A[i]$ as $ITE(i = 1, A[1], ITE(i = 2, A[2], \ldots))$

Practical issues

variable storage referencing problem

- when i is dependent on input, then $A[i]$ can point to various locations in memory
- unsatisfactory solution:
  handle $A[i]$ as $ITE(i = 1, A[1], ITE(i = 2, A[2], \ldots))$

other memory related problems

- reading/writing via pointers
- comparison of addresses (inner program nondeterminism)
- allocation of memory blocks of symbolic size

Practical issues

variable storage referencing problem

- when i is dependent on input, then $A[i]$ can point to various locations in memory
- unsatisfactory solution:
  handle $A[i]$ as $ITE(i = 1, A[1], ITE(i = 2, A[2], \ldots))$

other memory related problems

- reading/writing via pointers
- comparison of addresses (inner program nondeterminism)
- allocation of memory blocks of symbolic size

solution: fully symbolic memory model

- performance issues

Practical issues

- path explosion problem
  - the number of branches in the symbolic execution tree can be extremely high or even infinite
  - typical for program cycles with the number of iterations depending on the input (symbolic execution forks again and again)
  - construction of full symbolic execution tree is often infeasible
- issues with complex arithmetic operations (e.g. in hashing, encryption or decryption), calls to the operating system and libraries
- practical solutions
  - concretization
  - concolic execution

Concolic execution

- concolic = concrete + symbolic
- program is executed on a real input and on symbolic input simultaneously
- symbolic execution does not fork, it always follows the concrete execution and computes *pc*
- if a symbolic value is not available, we can switch to a concrete one

Real applications

- typical applications
  - bug finding
  - test generation
  - analysis of abstract error traces
- often combined with other techniques
- used in many tools including Klee, PEX, SAGE, SLAM, Ultimate Automizer, Symbiotic

Automated whitebox fuzz testing

Automated whitebox fuzz testing

- an example of modern and sophisticated testing method
- implemented in SAGE (Scalable, Automated, Guided Execution)
- discovered 30+ new bugs in large-shipped (and thus intensively tested) file-reading Windows applications including image processors, media players, file decoders
- combines fuzz testing and symbolic execution

Key ideas

- symbolic execution is expensive compared to running tests
- thus we want to generate as many new inputs from one symbolic execution as possible
- input for the next symbolic execution is selected by some scoring function applied to all generated inputs
- in particular, the input that explored the most (so-far uncovered) pieces of code is chosen for the next symbolic execution

The main algorithm

```
procedure GenerateInputs(inputSeed)
    inputSeed.bound $\leftarrow 0$
    workList $\leftarrow \{$inputSeed$\}$
    Run&Check(program, inputSeed)
    while workList $\neq \emptyset$ do
        input $\leftarrow$ PickFirstItem(workList)
        childInputs $\leftarrow$ ExpandExecution(input)
        foreach newInput $\in$ childInputs do
            Run&Check(program, newInput)
            Score(newInput)
            workList $\leftarrow$ workList $\cup \{$newInput$\}$
```

- `Score`(newInput) counts the newly covered blocks
- workList is ordered by the score of inputs
- `PickFirstItem`(workList) returns the input with the highest score

Application of symbolic execution

```
procedure ExpandExecution(input)
    childInputs ← ∅
    PC ← SymbolicExecution(program, input)
    for $j$ ← input.bound to $|PC| - 1$ do
        if $\bigwedge_{i=0}^{j-1} PC[i] \;\wedge\; \neg PC[j]$ has solution $M$ then
            newInput ← Combine(input, $M$)
            newInput.bound ← $j$
            childInputs ← childInputs ∪ {newInput}
    return childInputs
```

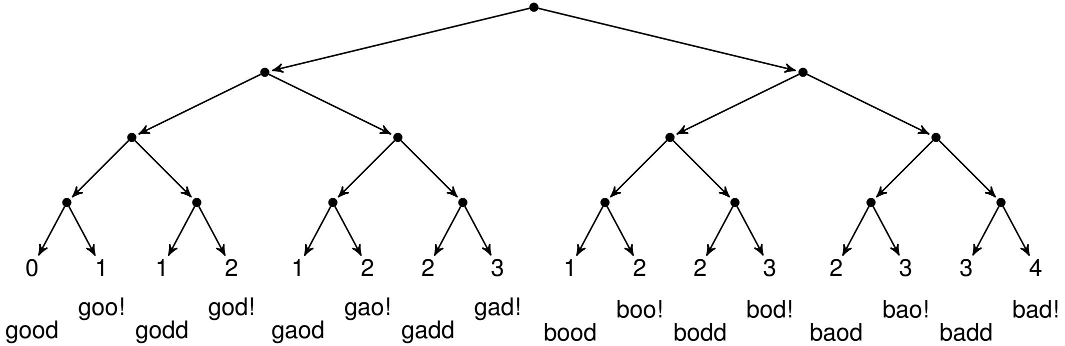
- `Combine`(input, M)
  - creates a new input from the original input and M
  - `Combine`("abcde", input[3] $\mapsto$ "F") returns "abcFe"
- path conditions are represented as arrays PC of conjuncts

Example

```c
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

Example

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```



Notes

- the algorithm can be parallelized: only workList and the overall block coverage need to be shared
- SAGE recovers easily from divergencies (situations when an execution deviates from the assumed execution path) induced e.g. by inner program nondeterminism
- SAGE runs 24/7 on large clusters, available for Microsoft developers

Bounded model checking

Bounded model checking (BMC)

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

Bounded model checking (BMC)

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

workflow

```
1 unwind all loops and recursion to a given bound $k$
2 compute the error reaching formula in unwound program
3 check satisfiability of the formula
4 if satisfiable
5     then bug found
6     else if the bound is not reachable
7             then the program is correct
8             else unknown (increase bound and goto 1)
```

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);
```

unwound program for $k = 3$

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

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);
```

unwound program for $k = 3$

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

satisfiable

variable types are considered

bitvector arithmetic is used

$$
\begin{array}{lll}
n_1 = 2 & & \\
v_1 = 0 & s_1 = 0 & i_1 = 0 \\
v_2 = 224 & s_2 = 224 & i_2 = 1 \\
v_3 = 63 & s_3 = 31 & i_3 = 2
\end{array}
$$

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);
```

unwound program for $k = 3$

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

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);
```

unwound program for $k = 3$

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

unsatisfiable

is the bound reachable?

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

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);
```

unwound program for $k = 3$

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

unsatisfiable

is the bound reachable?

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

satisfiable $\implies$ bound reachable

Notes on BMC

- very efficient in finding bugs
- constant propagation can simplify the program and the formula
- 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