CS 489 / 698: Software and Systems Security

Module 2: Program Security (Defenses)
static and symbolic reasoning

Meng Xu *(University of Waterloo)*

Spring 2023

Outline

Why this topic?

A significant portion of software security research is related to program analysis:

- derive properties which hold for program P (i.e., inference)
- prove that some property holds for program P (i.e., verification)
- given a program P, generate a program P' which is
  - in most ways equivalent to P
  - behaves better than P w.r.t some criteria

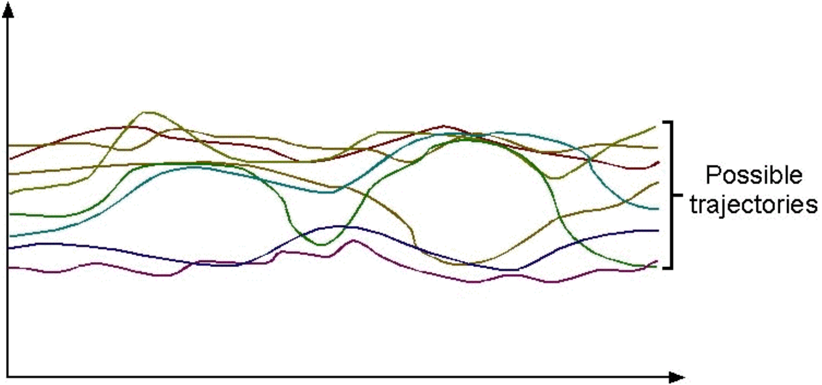
  (i.e., transformation)

Abstract interpretation provides a formal framework for developing program analysis tools.

Abstract interpretation in a nutshell

Acknowledgement: the illustrations in this section is borrowed from Prof. Patrick Cousot's webpage Abstract Interpretation in a Nutshell.

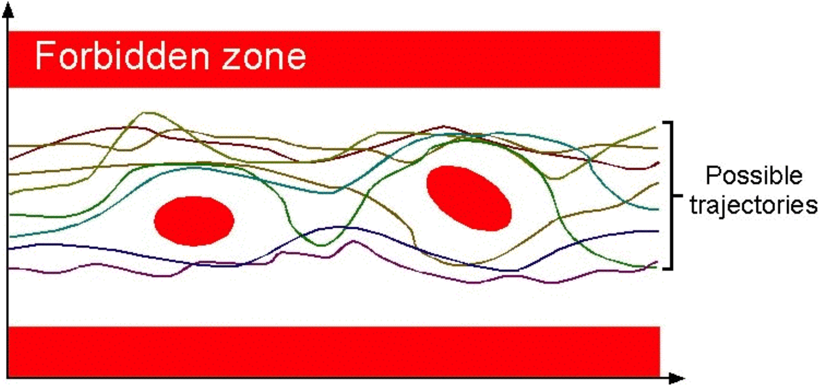
Program analysis: concrete semantics



The concrete semantics of a program is formalized by the set of all possible executions of this program under all possible inputs.

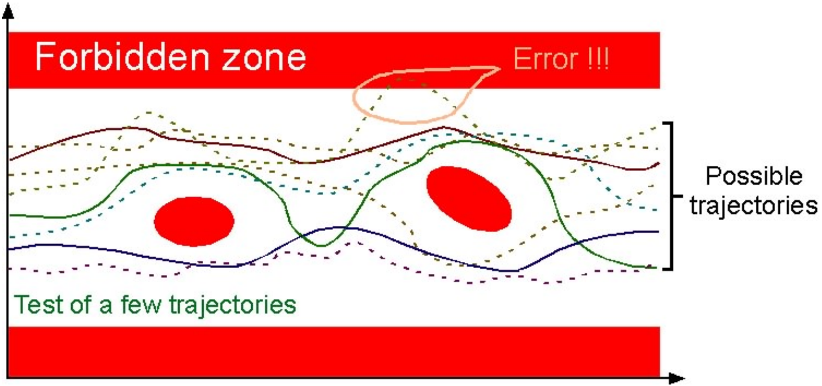
The concrete semantics of a program can be a *close to infinite* mathematical object / sequence which is impractical to enumerate.

Program analysis: safety properties



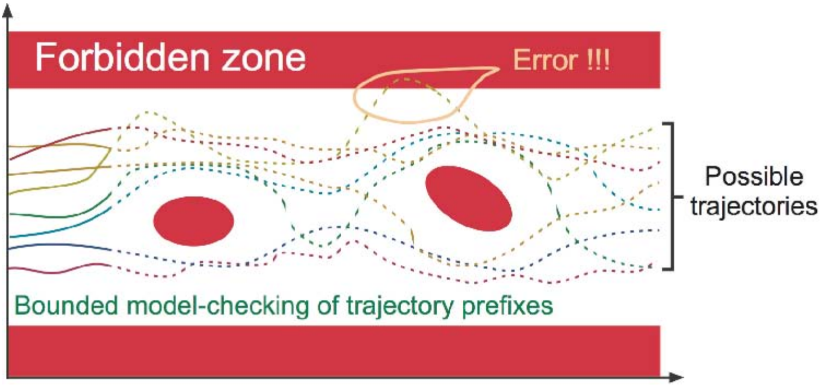
Safety properties of a program express that no possible execution of the program, when considering all possible execution environments, can reach an erroneous state.

Program analysis: testing



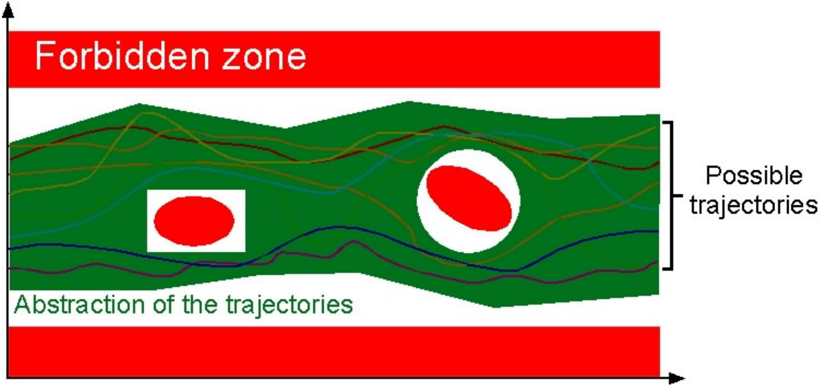
Testing consists in considering a subset of the possible executions.

Program analysis: bounded model checking



Bounded model checking consists in exploring the prefixes of the possible executions.

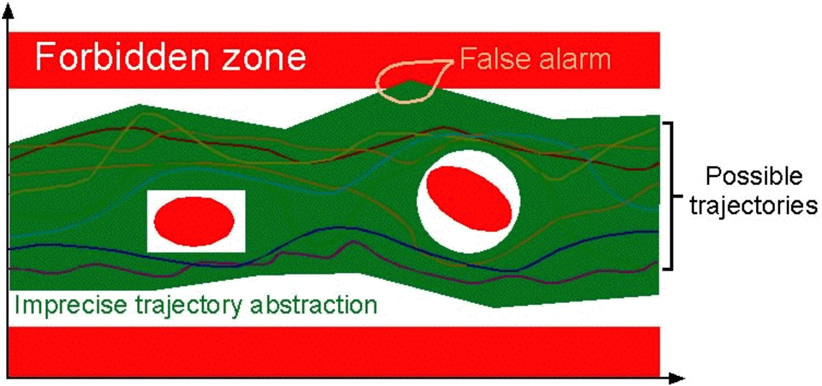
Program analysis: abstract interpretation



Abstract interpretation consists in considering an abstract semantics, that is a superset of the concrete program semantics.

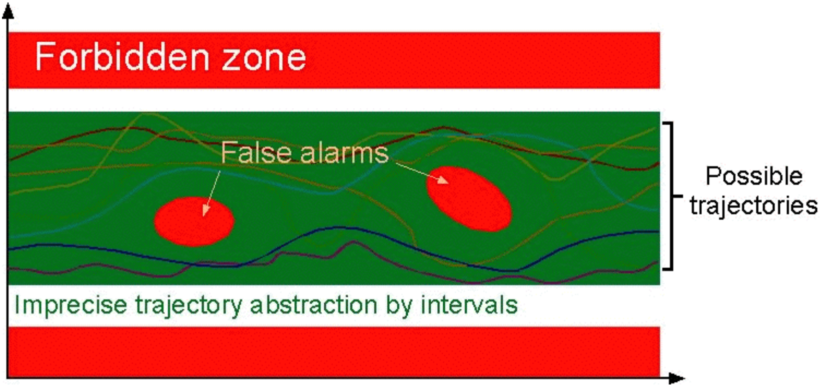
The abstract semantics covers all possible cases
$\Longrightarrow$ if the abstract semantics is safe (i.e. does not intersect the forbidden zone) then so is the concrete semantics.

Program analysis: abstract interpretation false alarm 1



False alarms caused by widening during execution.

Program analysis: abstract interpretation false alarm 2



False alarms caused by abstract domains.

Outline

What is abstract interpretation?

Consider detecting that one branch will not be taken in:

$\textbf{int}\ x, y, z; \quad y := read(file); \quad x := y * y;$

$\textbf{if}\ x \ \geq\ 0\ \textbf{then}\ z := 1\ \textbf{else}\ z := 0$

- Exhaustive analysis in the standard domain: non-termination
- Human reasoning about programs – uses abstractions: signs, order of magnitude, odd/even, ...

Basic idea: use approximate (generally finite) representations of computational objects to make the problem of program dataflow analysis tractable.

What is abstract interpretation?

Abstract interpretation is a formalization of the above procedure:

- define a non-standard semantics which can approximate the meaning (or behaviour) of the program in a finite way
- expressions are computed over an approximate (abstract) domain rather than the concrete domain (i.e., meaning of operators has to be reconsidered w.r.t. this new domain)

Example: integer sign arithmetic

Consider the domain $D = Z$ (integers)
and the multiplication operator: $* : Z^2 \rightarrow Z$

We define an "abstract domain:" $D_\alpha = \{[-],[+]\}$
and abstract multiplication: $*_\alpha : D_\alpha^2 \rightarrow D_\alpha$ defined by:

$*_\alpha$	$[-]$	$[+]$
$[-]$	$[+]$	$[-]$
$[+]$	$[-]$	$[+]$

This allows us to conclude, for example, that $y = x^2 = x * x$ is never negative.

Some observations

- The basis is that whenever we have $z = x * y$ then:
  if $x, y \in Z$ are approximated by $x_\alpha, y_\alpha \in D_\alpha$
  then $z \in Z$ is approximated by $z_\alpha = x_\alpha *_\alpha y_\alpha$
  - Essentially, we map from an unbounded domain to a finite domain.
- It is important to formalize this notion of approximation, in order to be able to reason/prove that the analysis is correct.
- Approximate computation is generally less precise but faster (hence the tradeoff).

Example: integer sign arithmetic (refined)

Again, $D = Z$ (integers)
and: $* : Z^2 \to Z$

We can define a more refined "abstract domain"
$D'_\alpha = \{[-], [0], [+]\}$

and the corresponding abstract multiplication: $*_\alpha : {D'}_\alpha^2 \to D'_\alpha$

$*_\alpha$	$[-]$	$[0]$	$[+]$
$[-]$	$[+]$	$[0]$	$[-]$
$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$

This allows us to conclude, for example, that $z = y * (0 * x)$ is zero.

More observations

- There is a degree of freedom in defining different abstract operators and domains.
- The minimal requirement is that they be “safe” or “correct”.
- Different “safe” definitions result in different kinds of analysis.

Example: integer sign arithmetic (with addition)

Again, $D = Z$ (integers)
and now we want to define the *addition* operator $+ : Z^2 \rightarrow Z$

We cannot use $D'_\alpha = \{[-], [0], [+]\}$ because we wouldn't know how to represent the result of $[+] +_\alpha [-]$, (i.e., the abstract addition would not be closed).

Solution: introduce a new element "$\top$" in the abstract domain as an approximation of any integer.

Example: integer sign arithmetic (with addition)

New "abstract domain": $D'_\alpha = \{[-], [0], [+], \top\}$

Abstract $+_\alpha : D'^2_\alpha \to D'_\alpha$

$+_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[-]$	$[-]$	$\top$	$\top$
$[0]$	$[-]$	$[0]$	$[+]$	$\top$
$[+]$	$\top$	$[+]$	$[+]$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Abstract $*_\alpha : D'^2_\alpha \to D'_\alpha$

$*_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[+]$	$[0]$	$[-]$	$\top$
$[0]$	$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$	$\top$
$\top$	$\top$	$[0]$	$\top$	$\top$

We can now reason that $z = x^2 + y^2$ is never negative

More observations

- In addition to the imprecision due to the coarseness of D_α, the abstract versions of the operations (dependent on D_α) may introduce further imprecision
- Thus, the choice of *abstract domain* and the definition of the *abstract operators* are crucial.

Concerns in abstract interpretation

- Required:
  - Correctness – safe approximations: the analysis should be "conservative" and errs on the "safe side"
  - Termination – compilation should definitely terminate

  (note: not always the case in everyday program analysis tools!)

- Desirable – "practicality":
  - Efficiency – in practice finite analysis time is not enough: finite *and* small is the requirement.
  - Accuracy – too many false alarms is harmful to the adoption of the analysis tool ("the boy who cried wolf").
  - Usefulness – determines which information is worth collecting.

Outline

Abstract domain example: intervals

Consider the following abstract domain for $x \in Z$ (integers):
$x = [a, b]$ where

- a can be either a constant or $-\infty$ and
- b can be either a constant or ∞.

Example:

$\{x^{\#} = [0, 9],\ y^{\#} = [-1, 1]\}$

```
z = x + 2 * y
```

$\{z^{\#} = [0, 9] +^{\#} 2 \times^{\#} [-1, 1] = [-2, 11]\}$

Q: Why $z^{\#}$ is an abstraction of z?

Join operator

The join operator $\sqcup$ merges two or more abstract states into one abstract state.

Joining operator example

$\{x^{\#} = [0, 10]\}$

```
if (x < 0) then
```

$\{x^{\#} = \emptyset\}$

```
   s := -1
```

$\{x^{\#} = \emptyset,\ s^{\#} = \emptyset\}$

```
else if (x > 0) then
```

$\{x^{\#} = [1, 10]\}$

```
   s := 1
```

$\{x^{\#} = [1, 10],\ s^{\#} = [1, 1]\}$

```
else
```

$\{x^{\#} = [0, 0]\}$

```
   s := 0
```

$\{x^{\#} = [0, 0],\ s^{\#} = [0, 0]\}$

$\{x^{\#} = \emptyset \sqcup [1, 10] \sqcup [0, 0] = [0, 10],\ s^{\#} = \emptyset \sqcup [1, 1] \sqcup [0, 0] = [0, 1]\}$

What about loops?

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = \langle even \rangle\}$
```
while (x < 100) {
```
$\{x^{\#} = \langle even \rangle\}_1$ $\{x^{\#} = \langle even \rangle \sqcup \langle even \rangle = \langle even \rangle\}_2$
```
  x := x + 2
```
$\{x^{\#} = \langle even \rangle\}_1$
```
}
```
$\{x^{\#} = \langle even \rangle\}$

Two iterations to reach fixedpoint (i.e., none of the abstract states changes).

Collecting semantics

$\{x^\# = \emptyset\}$

```
x := 0
```

$\{x^\# = [0,0]\}$

```
while (x < 100) {
```

$\{x^\# = [0,0]\}_1$ $\{x^\# = [0,0] \sqcup [2,2] = [0,2]\}_2$
$\{x^\# = [0,2] \sqcup [2,4] = [0,4]\}_3$ $\{\cdots\}_4, \{\cdots\}_5, \cdots$
$\{x^\# = [0,96] \sqcup [2,98] = [0,98]\}_{50}$

```
  x := x + 2
```

$\{x^\# = [2,2]\}_1$ $\{x^\# = [2,2] \sqcup [2,4] = [2,4]\}_2$
$\{x^\# = [2,4] \sqcup [2,6] = [2,6]\}_3$ $\{\cdots\}_4, \{\cdots\}_5, \cdots$
$\{x^\# = [2,98] \sqcup [2,100] = [2,100]\}_{50}$

```
}
```

$\{x^\# = [100,100]\}$

50 iterations to reach fixedpoint (i.e., none of the abstract states

Widening operator

We compute the limit of the following sequence:

$$X_0 = \bot$$
$$X_{i+1} = X_i \triangledown F^{\#}(X_i)$$

where $\triangledown$ denotes the widening operator.

Widening operator example

$\{x^{\#} = \emptyset\}$

```
x := 0
```
$\{x^{\#} = [0, 0]\}$
```
while (x < 100) {
```
$\{x^{\#} = [0, 0]\}_1$ $\{x^{\#} = [0, 0] \triangledown [2, 2] = [0, +\infty]\}_2$
$\{x^{\#} = [0, +\infty] \triangledown [2, +\infty] = [0, +\infty]\}_3$
```
  x := x + 2
```
$\{x^{\#} = [2, 2]\}_1$ $\{x^{\#} = [2, +\infty]\}_2$ $\{x^{\#} = [2, +\infty]\}_3$
```
}
```
$\{x^{\#} = [100, +\infty]\}$

3 iterations to reach fixedpoint (i.e., none of the abstract states changes).

Narrowing operator

We compute the limit of the following sequence:

$$X_0 = \bot$$

$$X_{i+1} = X_i \triangle F^{\#}(X_i)$$

where $\triangle$ denotes the narrowing operator.

Narrowing operator example

$\{x^{\#} = \emptyset\}$

```
x := 0
```

$\{x^{\#} = [0, 0]\}$

```
while (x < 100) {
```

$\{x^{\#} = [0, +\infty]\}$ $\{x^{\#} = [0, +\infty] \triangle [0, 99] = [0, 99]\}_1$

$\{x^{\#} = [2, 101] \triangle [0, 99] = [0, 99]\}_2$

```
  x := x + 2
```

$\{x^{\#} = [2, +\infty]\}$ $\{x^{\#} = [2, 101]\}_1$ $\{x^{\#} = [2, 101]\}_2$

```
}
```

$\{x^{\#} = [100, 101]\}$

2 iterations to reach fixedpoint (i.e., none of the abstract states changes).

Outline

Motivation

Q: Why research on symbolic execution when we have unit testing or even fuzzing?

A: A more complete exploration of program states.

Illustration

```
fn foo(x: u64): u64 {
    if (x * 3 == 42) {
        some_hidden_bug();
    }
    if (x * 5 == 42) {
        some_hidden_bug();
    }
    return 2 * x;
}
```

Unit Test

`foo(0);`

`foo(1);`

Fuzzing

`foo(0);`

`foo(1);`

`foo(12);`

`foo(78);`

......

`foo(9,223,372,036,854,775,808);`

Symbolic execution

foo(x)

aborts when $x = 14$

returns $2x$ otherwise

Satisfiability Modulo Theories (SMT)

Definition: A procedure that decides whether a mathematical formula is satisfiable.

Example:

- $3x = 42 \longrightarrow$ satisfiable with $x = 14$
- $2x \geq 2^{64} \longrightarrow$ satisfiable with $x \geq 2^{63}$
- $5x = 42 \longrightarrow$ unsatisfiable, cannot find an x

Ask two question whenever you see a symbolic execution work:

- How does it convert code into mathematical formula?
- What does it try to solve for?

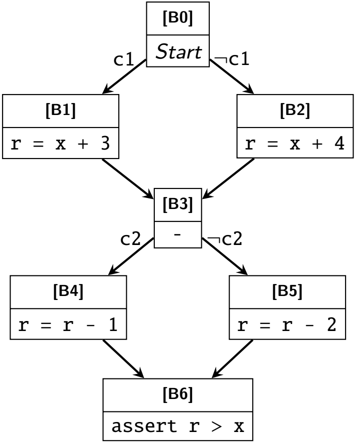
Program modeling desiderata

- Control-flow graph exploration
- Loop handling
- Memory modeling
- Concurrency

Outline

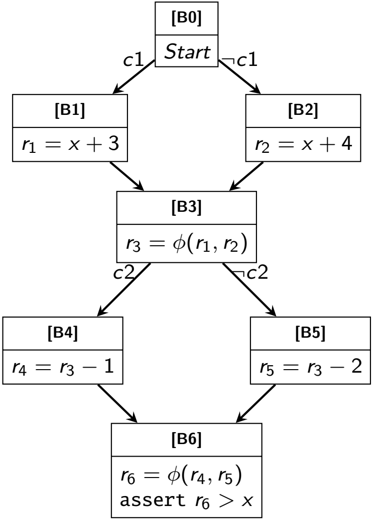
An example of a pure function

```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



The example in SSA form

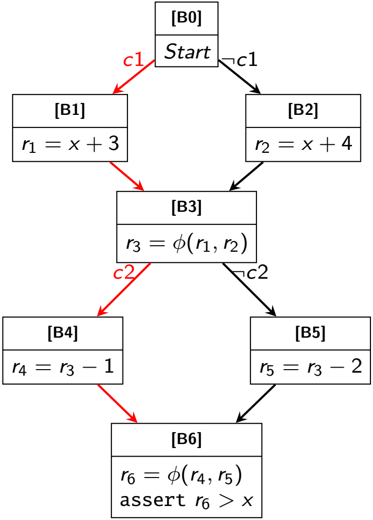
```
fn foo(
    c1: bool, c2: bool,
    x: u64
) -> u64 {
    let r = if (c1) {
        x + 3
    } else {
        x + 4
    };

    let r = if (c2) {
        r - 1
    } else {
        r - 2
    };

    r
}
spec foo {
    ensures r > x;
}
```



Path-based exploration

Vars: $c1,\ c2,\ x,\ r_{1-6}$

B0	Sym. repr. Path cond.	$\emptyset$ True
B1	Sym. repr. Path cond.	$r_1 = x + 3$ $c1$
B3	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $c1$
B4	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $c_1 \wedge c_2$
B6	Sym. repr. Path cond.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $r_6 = r_4$ $c_1 \wedge c_2$



Proving procedure (per path)

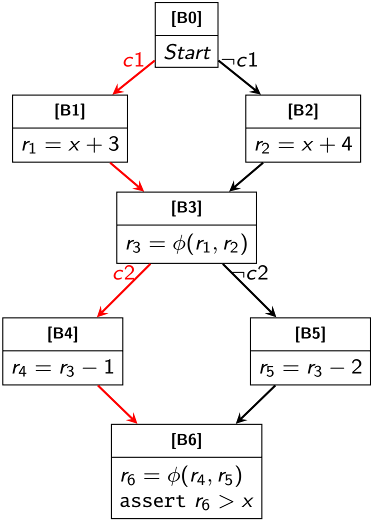
Vars: $c1$, $c2$, x, r_{1-6}

B6	Sym. repr.	$r_1 = x + 3$ $r_3 = r_1$ $r_4 = r_3 - 1$ $r_6 = r_4$
	Path cond.	$c_1 \wedge c_2$

$\rightsquigarrow$

Prove that $\forall\ c1, c2, x, r_{1-6}$:

$$((c1 \wedge c2) \wedge (\\ (r_1 = x + 3) \\ (r_3 = r_1) \\ (r_4 = r_3 - 1) \\ (r_6 = r_4) \\ )) \Rightarrow (r_6 > x)$$

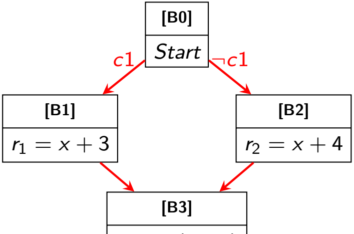


Proving procedure (all paths)

Prove that
$\forall\ c1, c2, x, r_{1-6}$:

$((c1 \wedge c2) \wedge ($
$(r_1 = x + 3)$
$(r_3 = r_1)$
$(r_4 = r_3 - 1)$
$(r_6 = r_4)$
$)) \Rightarrow (r_6 > x)$

$((c1 \wedge \neg c2) \wedge ($
$(r_1 = x + 3)$
$(r_3 = r_1)$
$(r_5 = r_3 - 2)$
$(r_6 = r_5)$
$)) \Rightarrow (r_6 > x)$



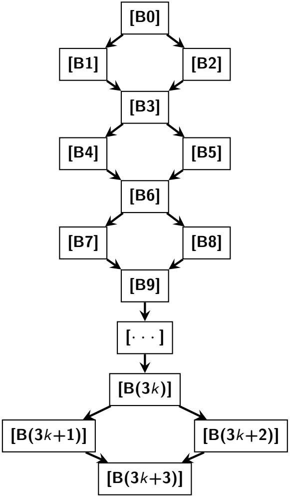
Path explosion

2^{2} paths

2^{3} paths

$\cdots$

2^{k} paths



$\langle$ **End** $\rangle$