Sireum/Kiasan

an extensible symbolic execution framework

An Introduction

Funding

NSF

AFOSR AIR FORCE OFFICE OF SCIENTIFIC RESEARCH

Rockwell Collins

CIS @ K-STATE COMPUTING AND INFORMATION SCIENCES

SANTOS LAB

Symbolic Execution

- a precise and intuitive analysis technique
  - similar to concrete execution
  - but, can reason about unknown data
- has received significant renewed interests
  - program testing (bug finding), automatic test-case generation
  - in our work: verification

Symbolic Execution [King:ACM76]

```
void foo(int x,
   int y,int z)
{
  z = x + y;
  if (z > 0){
    z++;
  }
}
```

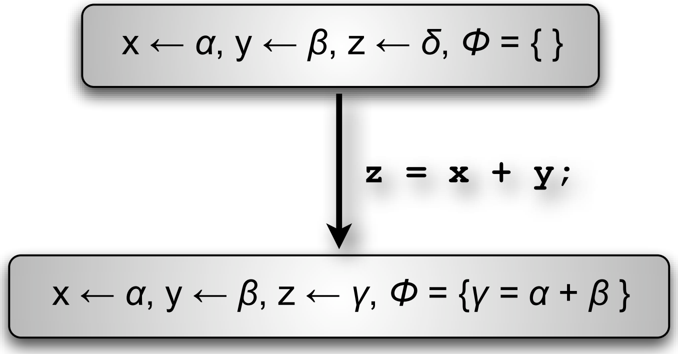
Symbolic Execution [King:ACM76]

```
void foo(int x,
   int y,int z)
{
  z = x + y;
  if (z > 0){
    z++;
  }
}
```

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \delta,\ \Phi = \{\ \}$

Symbolic Execution [King:ACM76]

```
void foo(int x,
   int y,int z)
{
  z = x + y;
  if (z > 0){
    z++;
  }
}
```



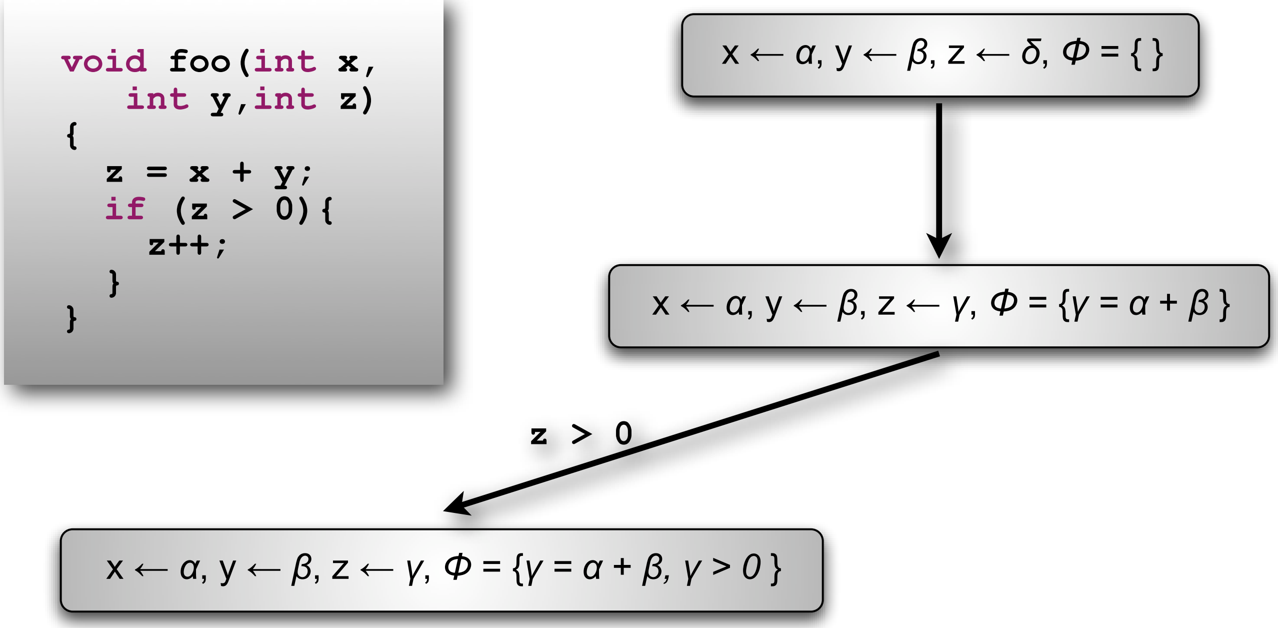
$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{ \}$

`z = x + y;`

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta \}$

Symbolic Execution [King:ACM76]

```
void foo(int x,
   int y,int z)
{
  z = x + y;
  if (z > 0){
    z++;
  }
}
```



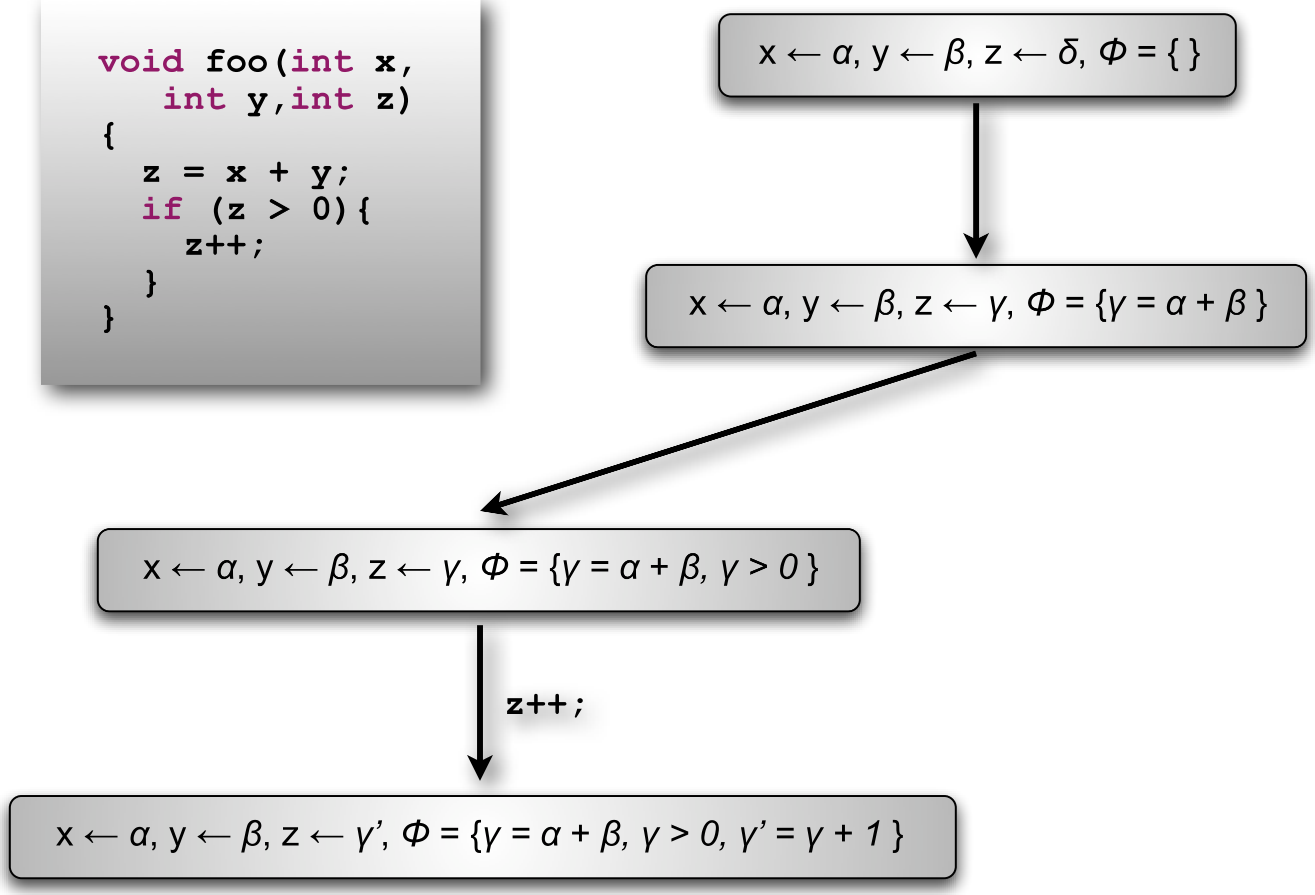
$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \delta,\ \Phi = \{\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0\}$

Symbolic Execution [King:ACM76]

```
void foo(int x,
    int y,int z)
{
   z = x + y;
   if (z > 0){
      z++;
   }
}
```



$x \leftarrow \alpha$, $y \leftarrow \beta$, $z \leftarrow \delta$, $\Phi = \{\ \}$

$x \leftarrow \alpha$, $y \leftarrow \beta$, $z \leftarrow \gamma$, $\Phi = \{\gamma = \alpha + \beta\ \}$

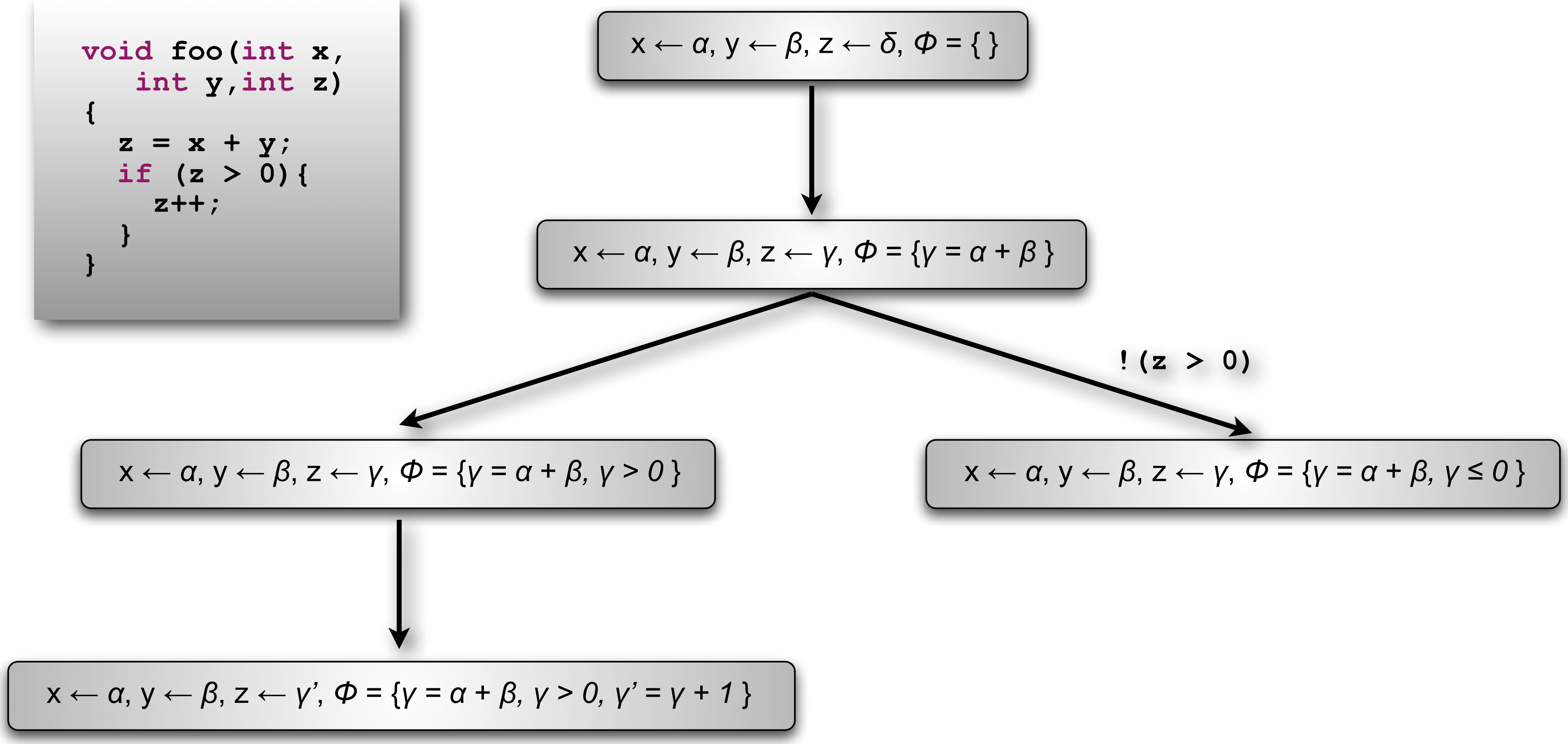
$x \leftarrow \alpha$, $y \leftarrow \beta$, $z \leftarrow \gamma$, $\Phi = \{\gamma = \alpha + \beta,\ \gamma > 0\ \}$

`z++;`

$x \leftarrow \alpha$, $y \leftarrow \beta$, $z \leftarrow \gamma'$, $\Phi = \{\gamma = \alpha + \beta,\ \gamma > 0,\ \gamma' = \gamma + 1\ \}$

Symbolic Execution [King:ACM76]

```
void foo(int x,
    int y,int z)
{
   z = x + y;
   if (z > 0){
      z++;
   }
}
```



$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \delta,\ \Phi = \{\,\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta\,\}$

!(z > 0)

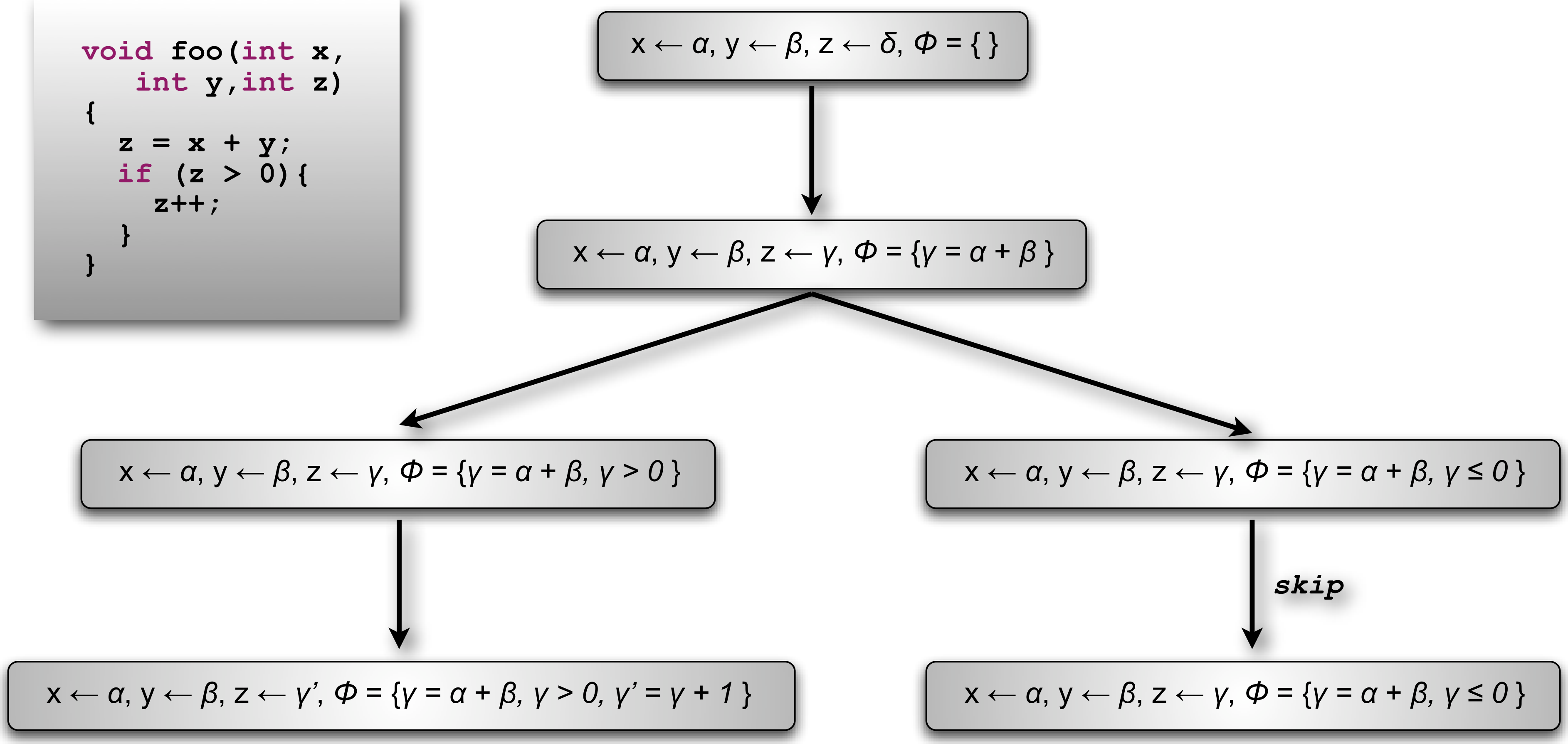
$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0\,\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\,\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma',\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0,\ \gamma' = \gamma + 1\,\}$

Symbolic Execution [King:ACM76]

```
void foo(int x,
   int y,int z)
{
  z = x + y;
  if (z > 0){
    z++;
  }
}
```



$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \delta,\ \Phi = \{\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0\ \}$

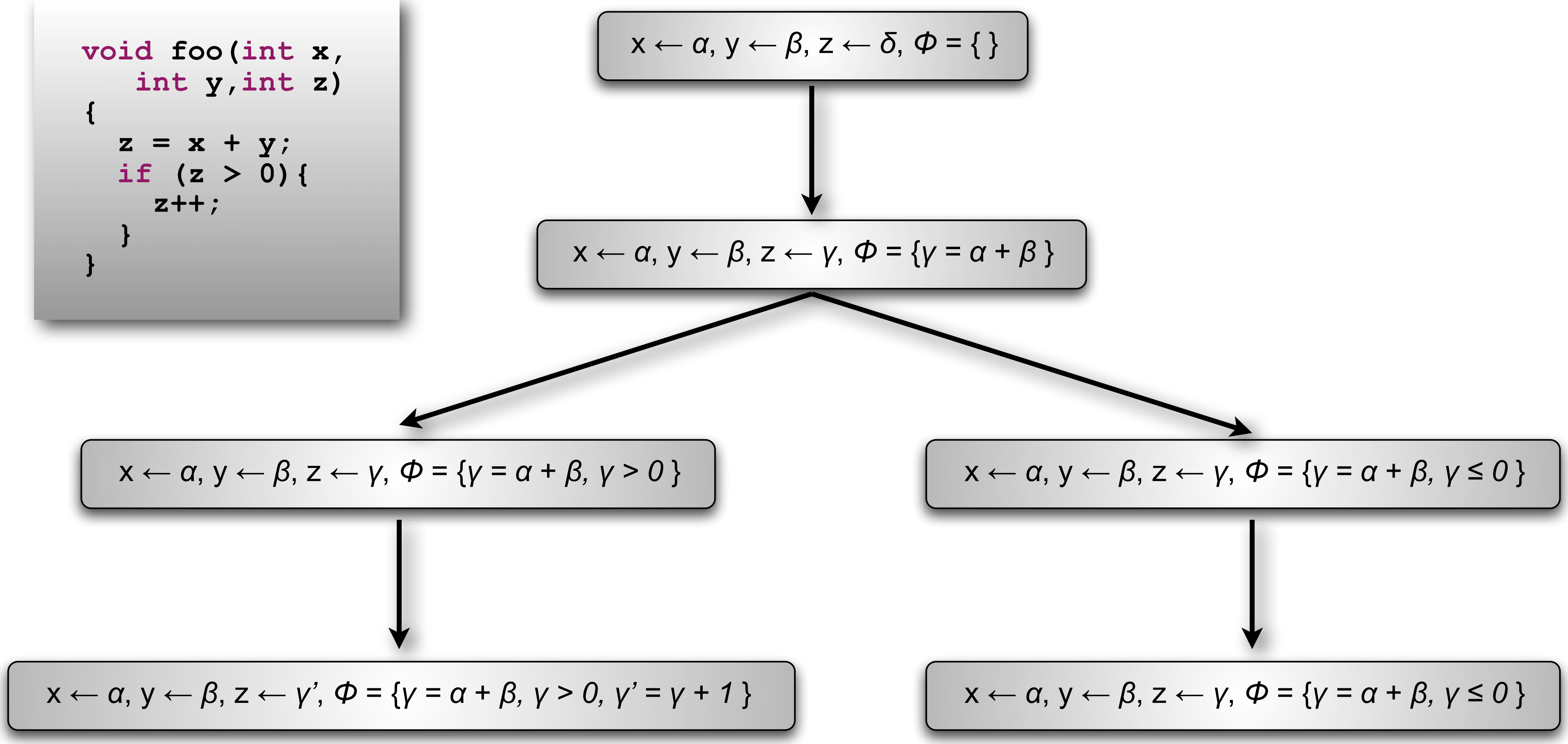
$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma',\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0,\ \gamma' = \gamma + 1\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\ \}$

Symbolic Execution [King:ACM76]

```
void foo(int x,
    int y,int z)
{
   z = x + y;
   if (z > 0){
      z++;
   }
}
```



$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \delta,\ \Phi = \{\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\ \}$

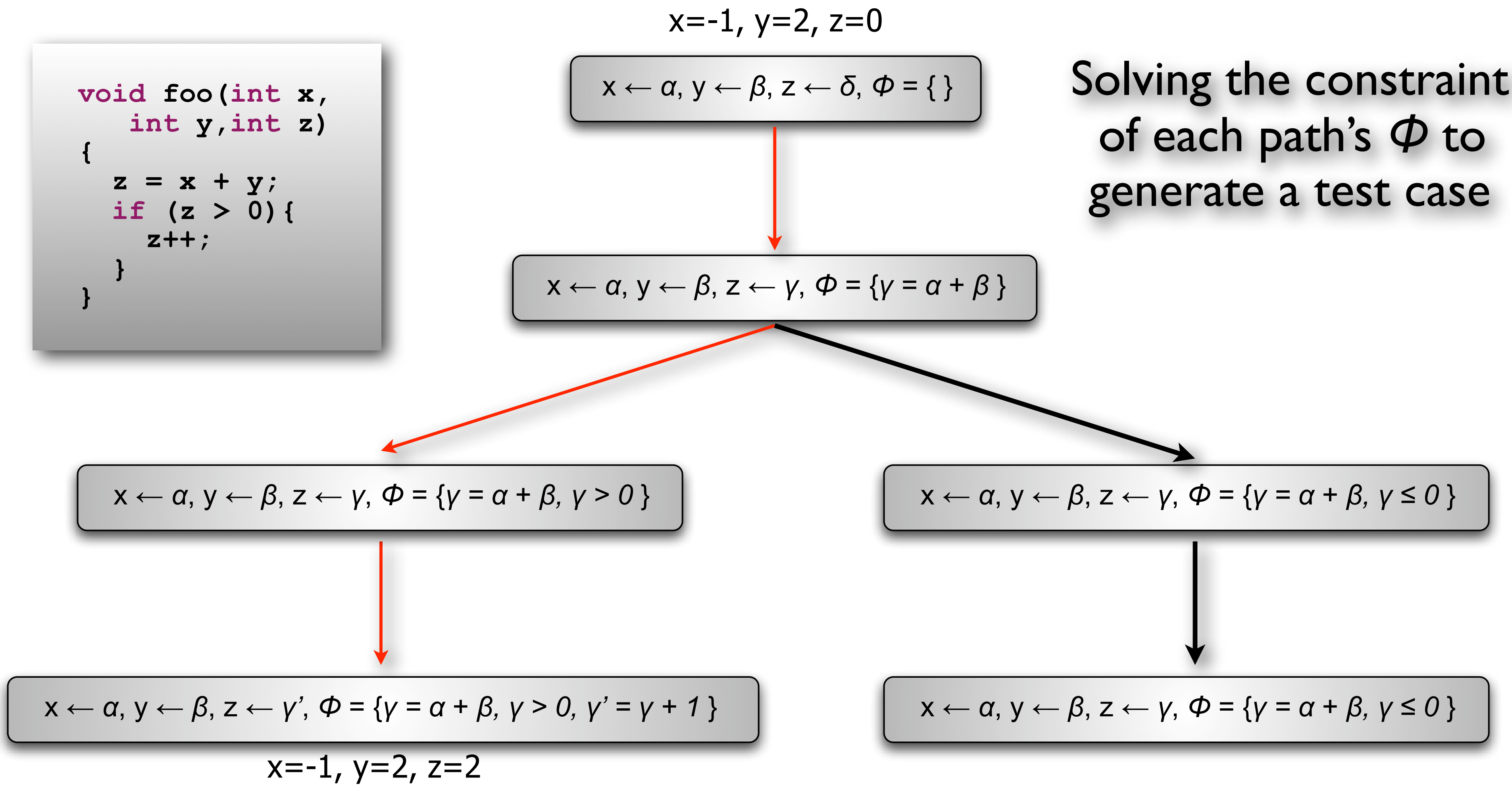
$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma',\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0,\ \gamma' = \gamma + 1\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\ \}$

…symbolic execution characterizes (theoretically) infinite number of real executions!

Test Case Generation

```
void foo(int x,
   int y,int z)
{
  z = x + y;
  if (z > 0){
    z++;
  }
}
```



x=-1, y=2, z=0

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \delta,\ \Phi = \{\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma',\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0,\ \gamma' = \gamma + 1\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\}$

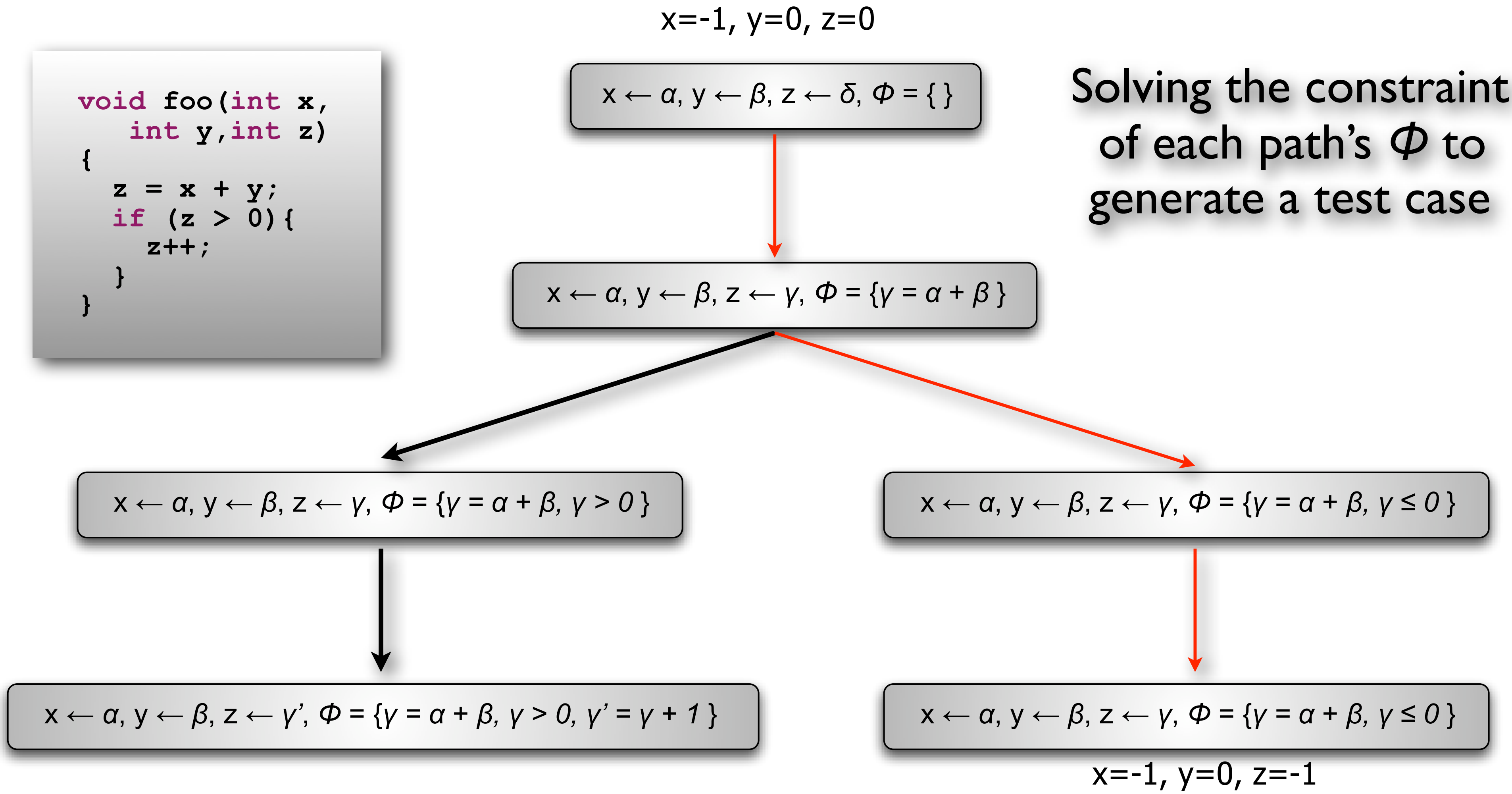
x=-1, y=2, z=2

Solving the constraint of each path's Φ to generate a test case

... the explored computation tree can be directly leveraged for test case generation!

Test Case Generation

```
void foo(int x,
    int y,int z)
{
   z = x + y;
   if (z > 0){
      z++;
   }
}
```



x=-1, y=0, z=0

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \delta,\ \Phi = \{\ \}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma',\ \Phi = \{\gamma = \alpha + \beta,\ \gamma > 0,\ \gamma' = \gamma + 1\}$

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \gamma,\ \Phi = \{\gamma = \alpha + \beta,\ \gamma \leq 0\}$

x=-1, y=0, z=-1

Solving the constraint of each path's Φ to generate a test case

... the explored computation tree can be directly leveraged for test case generation!

Observations

- SymExe mine path conditions
  - ... very precise
- Issue: path explosion
  - prune infeasible paths (constraint solvers)
  - fork at branch points (parallelize/distribute)
- Issue: termination
  - bounding (lazy)
  - require loop invariants

Kiasan Motivations

- At SAnToS Lab, we use SymExe for
  - contract checking
  - various application domains: Java, Spark, model
- Need: a "rapid" way to develop SymExe engine
  - experimentations
  - targeting various application domains
  - at different level abstraction levels

kiasan = symbolic (Indonesian)

Earlier Work on Java

- Adapted JPF Lazy Initialization [TACAS03] for contract checking [ASE06]
    - TACAS03 did not handle inheritance properly (unsound)
        - made it sound
    - added an object abstraction
        - ... relatively sound and complete
        - ... an order of magnitude improvement
- Introduced another level of object abstraction [SEFM07]
    - ... relatively sound and complete
    - ... an order of magnitude faster
- Introduced a linear semi-decision procedure [FSE09]
    - ... an order of magnitude faster

ASE Journal 2012

Recent Work on Spark

- Adaptation to Spark
  - Bakar Kiasan [NASA FM11]
  - Extended case study [SigAda11]
- Tailored SymExe for Spark [NASA FM12]
  - added first-class support for “value” structures
  - demonstrated that it is faster than record/array theory support in SMT solvers (e.g., Z3) for SymExe
- Explicating SymExe (xSymExe) [TR]

Sireum/Kiasan: Design Goals

- An extensible SymExe framework
  - easy to customize semantics
- ... designed to be highly parallel
  - leverage (massively) multi-core machines
- ... designed to be distributable
  - leverage clusters of machines

Today’s Roadmap

- ... - 10:30 Intro to Bakar Kiasan, Spark,
  and Design-by-Contract
  Tool Setup
- 11:00 - 12:30 Tool User: Hands-on with Bakar Kiasan
  Sireum/Kiasan Design & Arch.
- 14:00 - 15:30 Tool Dev: Hands-on with Sireum/Kiasan
- 16:00 - 16:30 xSymExe & Wrap-up