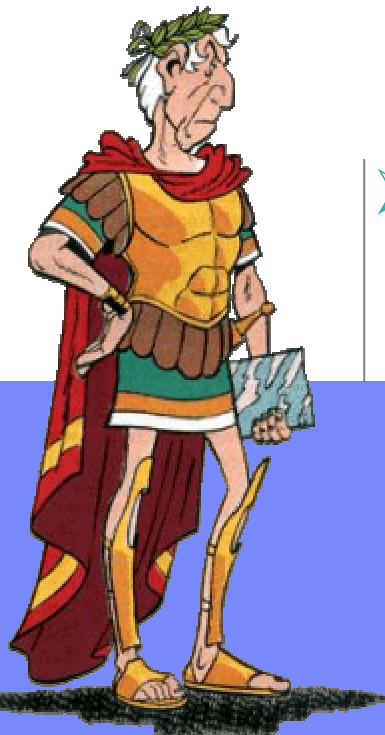




IBM

## Choosing among Alternative Pasts



➤ Marina Biberstein  
Eitan Farchi  
Shmuel Ur

IBM Labs in Haifa



## Table of contents

### **What's the problem?**

Choosing among the pasts

Soundness and value visibility

Conclusions



## Some problems in testing multithreaded programs

- ❖ Only few of the possible interleavings are usually generated for a given environment



## Some problems in testing multithreaded programs

- ◆ Only few of the possible interleavings are usually generated for a given environment

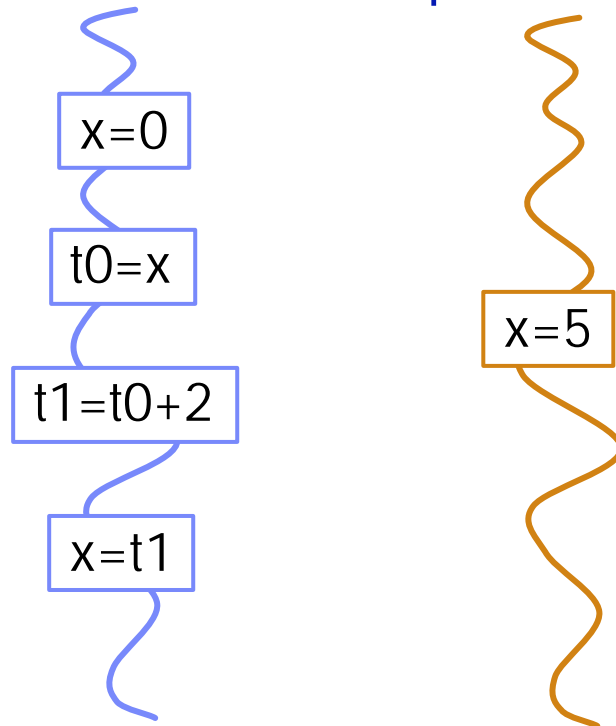
```
public class Print extends Thread{
    private String message;
    public Print(String _message){message = _message;}
    public void run(){System.out.print(message);}
}
public class Main{
    public static void main(String[] arguments){
        Print p1 = new Print("Hello, ");
        Print p2 = new Print("world!\n");
        p1.start();
        p2.start();
    }
}
```

◆ Almost always the output will be “Hello, world!”



## Some problems in testing multithreaded programs

- ◇ Only few of the possible interleavings are usually generated for a given environment
- ◇ There are a lot of possible interleavings



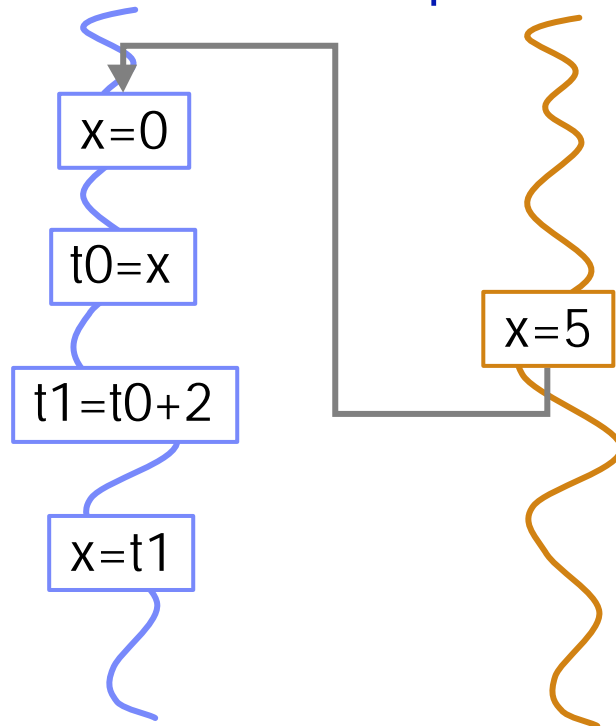
Here and later:

- x, y, z are shared variables
- t0, t1, t2 are locals



## Some problems in testing multithreaded programs

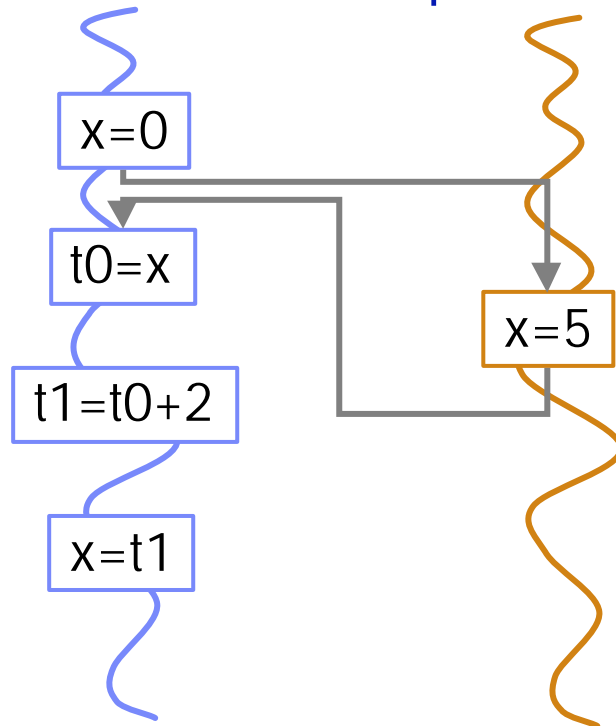
- ◇ Only few of the possible interleavings are usually generated for a given environment
- ◇ There are a lot of possible interleavings





## Some problems in testing multithreaded programs

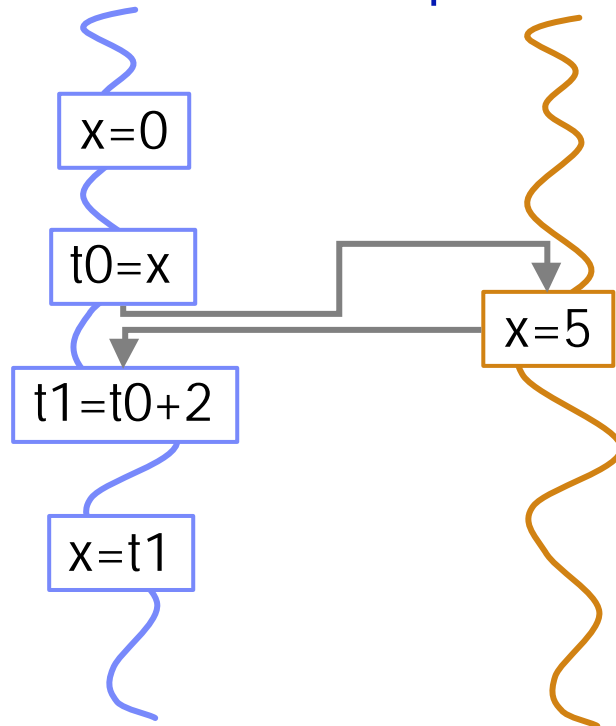
- ◇ Only few of the possible interleavings are usually generated for a given environment
- ◇ There are a lot of possible interleavings





## Some problems in testing multithreaded programs

- ◇ Only few of the possible interleavings are usually generated for a given environment
- ◇ There are a lot of possible interleavings

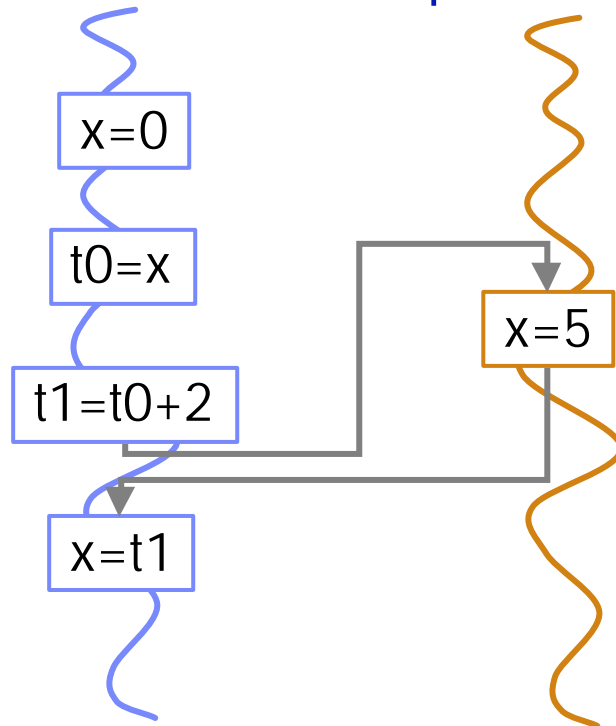






## Some problems in testing multithreaded programs

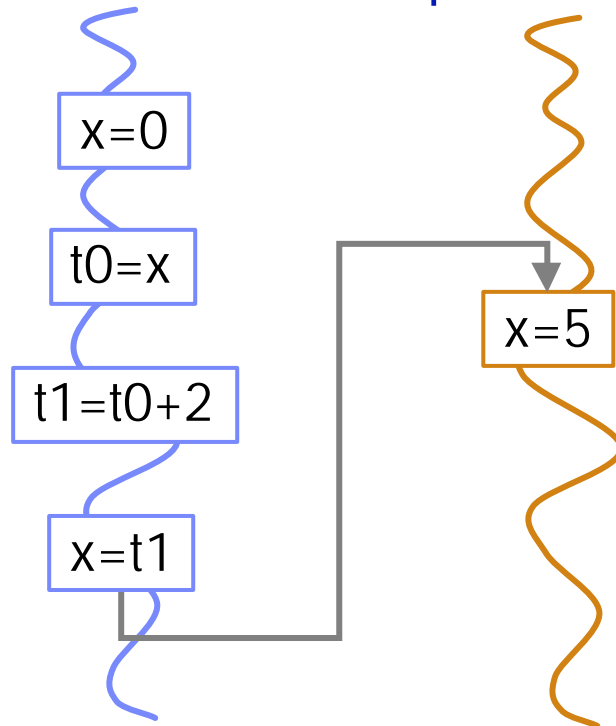
- ◇ Only few of the possible interleavings are usually generated for a given environment
- ◇ There are a lot of possible interleavings





## Some problems in testing multithreaded programs

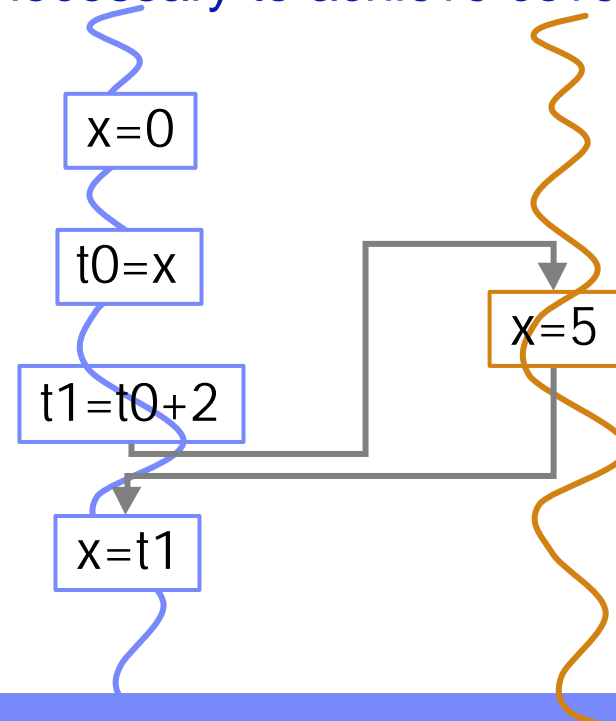
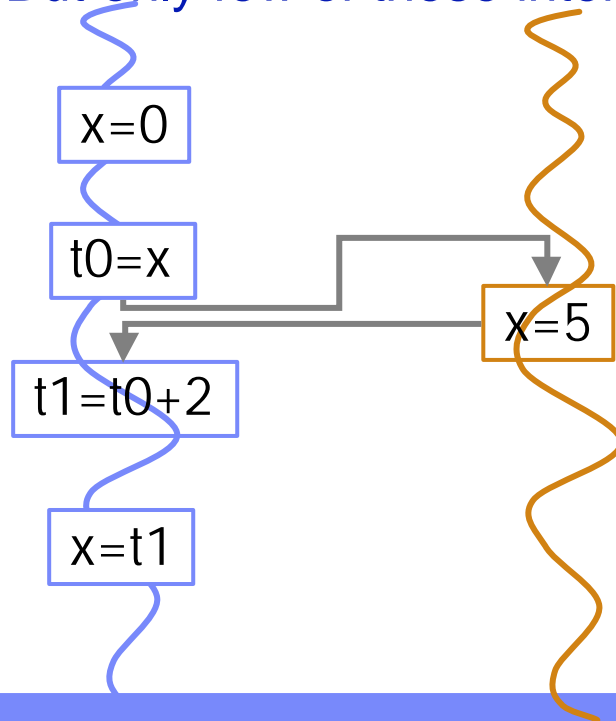
- ◇ Only few of the possible interleavings are usually generated for a given environment
- ◇ There are a lot of possible interleavings





## Some problems in testing multithreaded programs

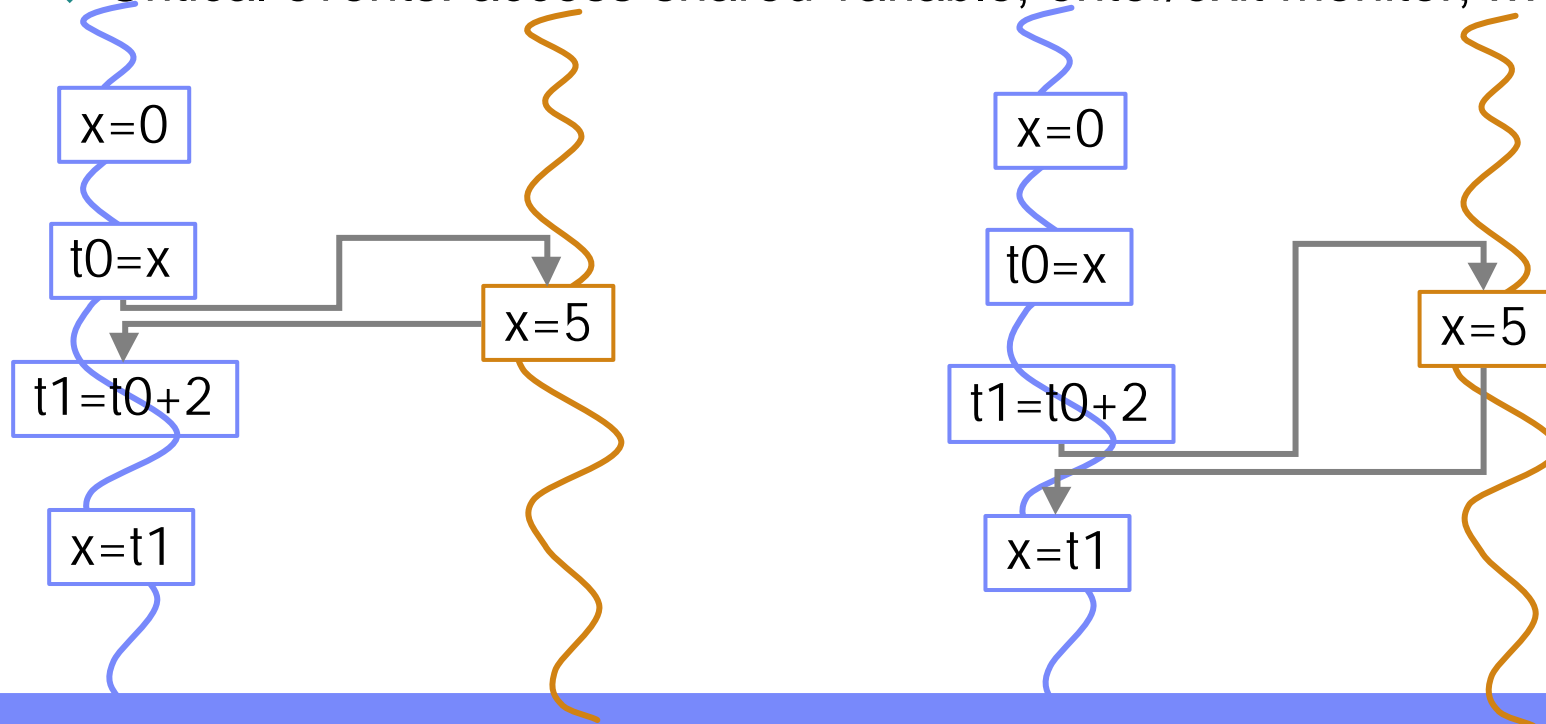
- ◇ Only few of the possible interleavings are usually generated for a given environment
- ◇ There are a lot of possible interleavings
- ◇ But only few of these interleavings are necessary to achieve coverage!





## Schedules: logical vs. physical

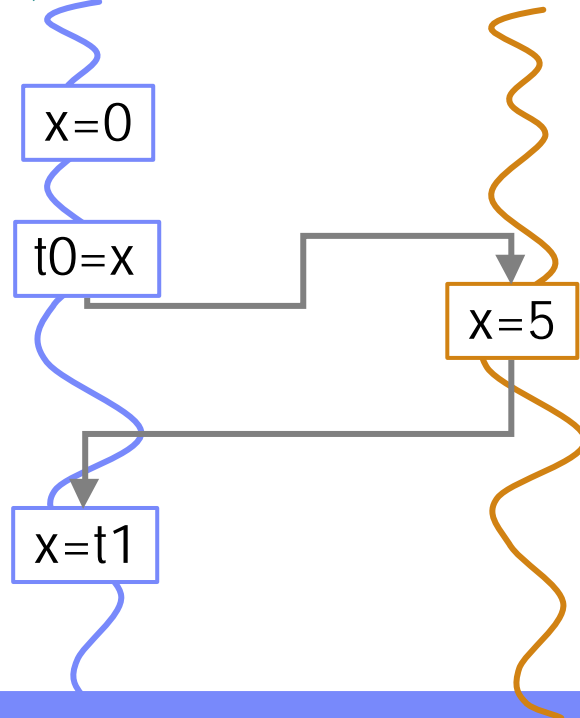
- Physical schedule: a linear ordering of all events
- Logical schedule: equivalence class of all physical schedules that agree on critical events (Choi & Srinivasan, '98)
- Critical events: access shared variable, enter/exit monitor, ...





## Schedules: logical vs. physical

- Physical schedule: a linear ordering of all events
- Logical schedule: equivalence class of all physical schedules that agree on critical events (Choi & Srinivasan, '98)
  - Critical events: access shared variable, enter/exit monitor, ...





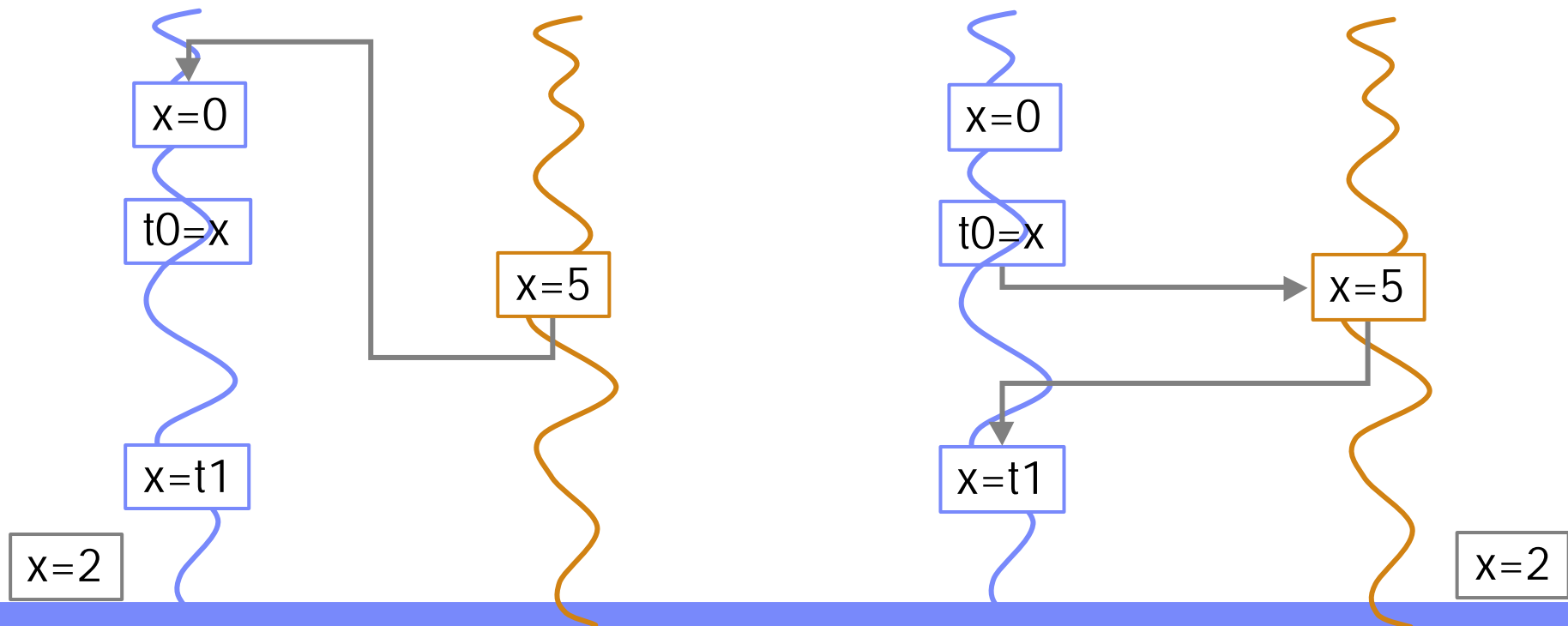
Let's take the idea another step forward...





## Schedules: value vs. logical

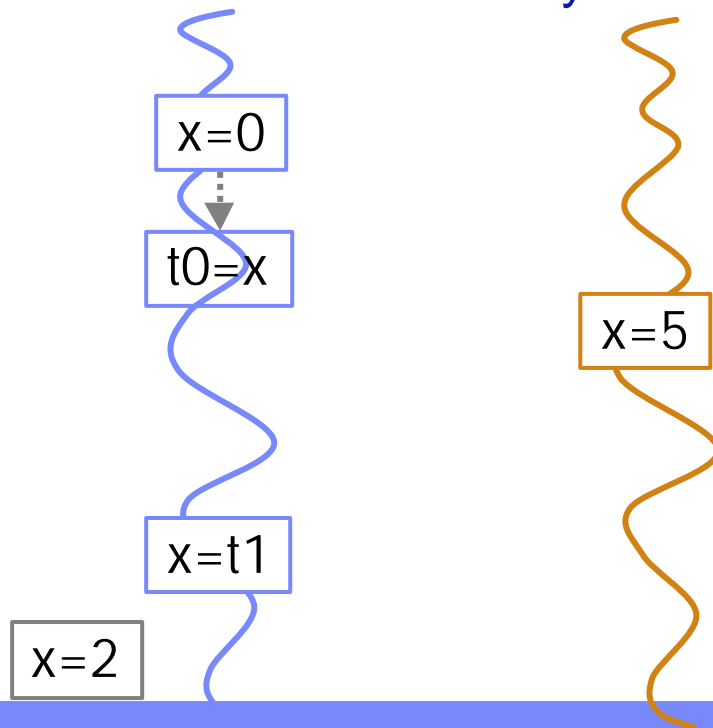
- ◆ Logical schedules that agree on values read by all read events – produce the same results





## Schedules: value vs. logical

- ◇ Logical schedules that agree on values read by all read events – produce the same results
- ◇ **Value schedule:** equivalence class of all logical schedules that agree on values consumed by read events





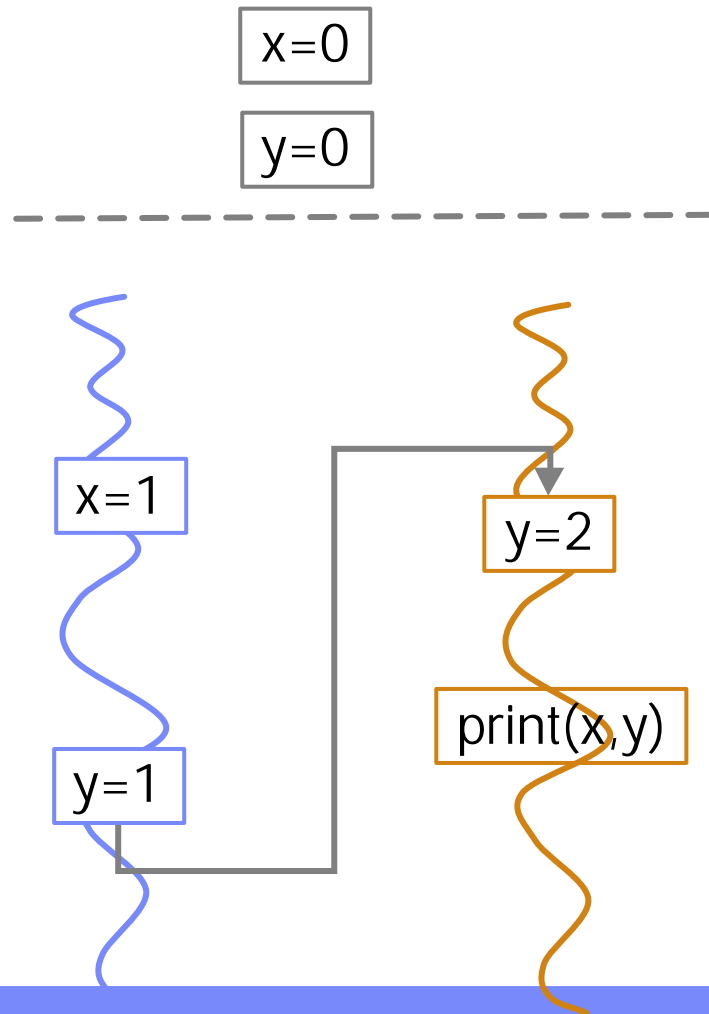


## Choosing among alternative pasts

- ◆ Testing goal:
  - ◆ Generate runs with different outcomes
  - Interfere with runtime to generate many different value schedules
- ◆ Value substitution process:
  - ◆ Execute the program, record critical events by thread
  - ◆ Interfere at shared variables' reads
    - ◆ Provide one of older values instead of the current one
  - ◆ Observation: the same effect as if a different value schedule had actually taken place

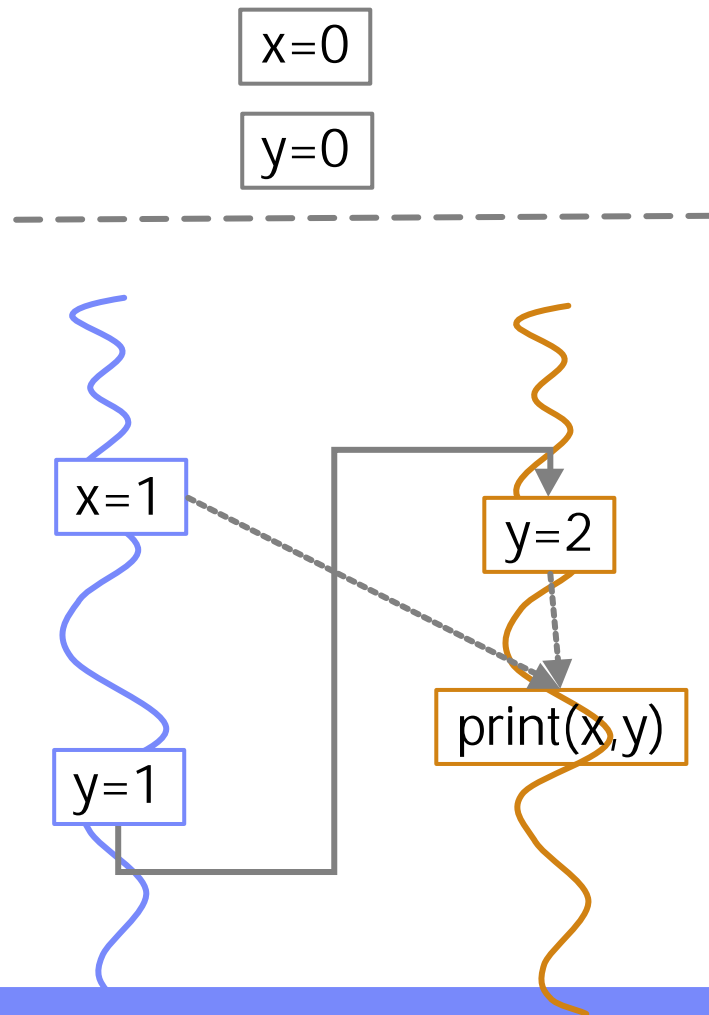


## Choosing among alternative pasts



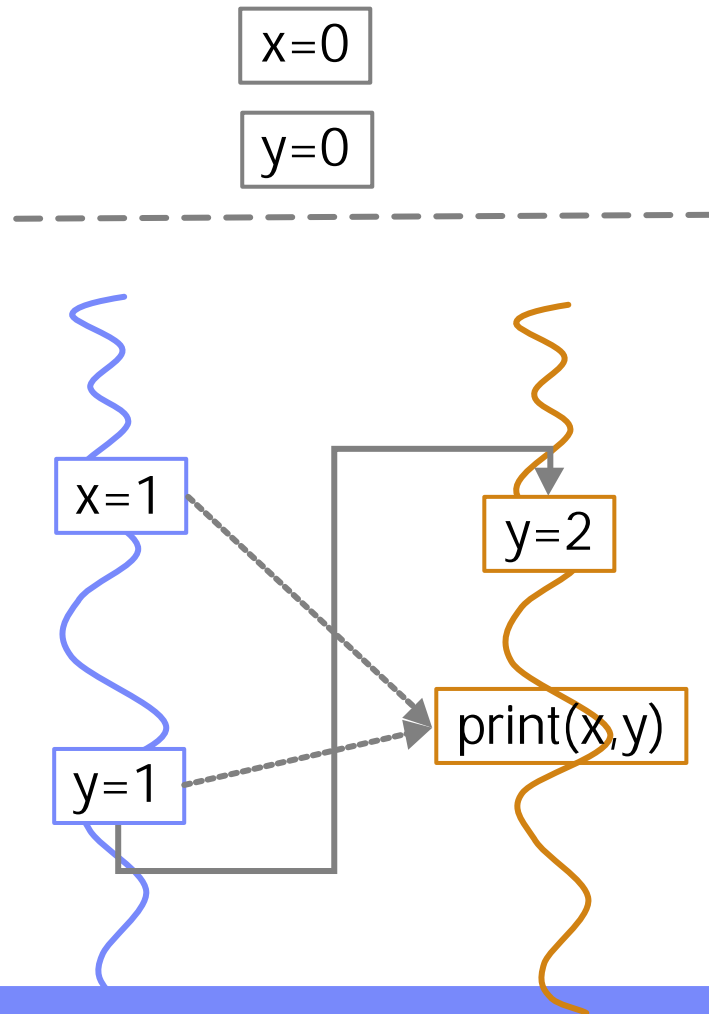


## Choosing among alternative pasts



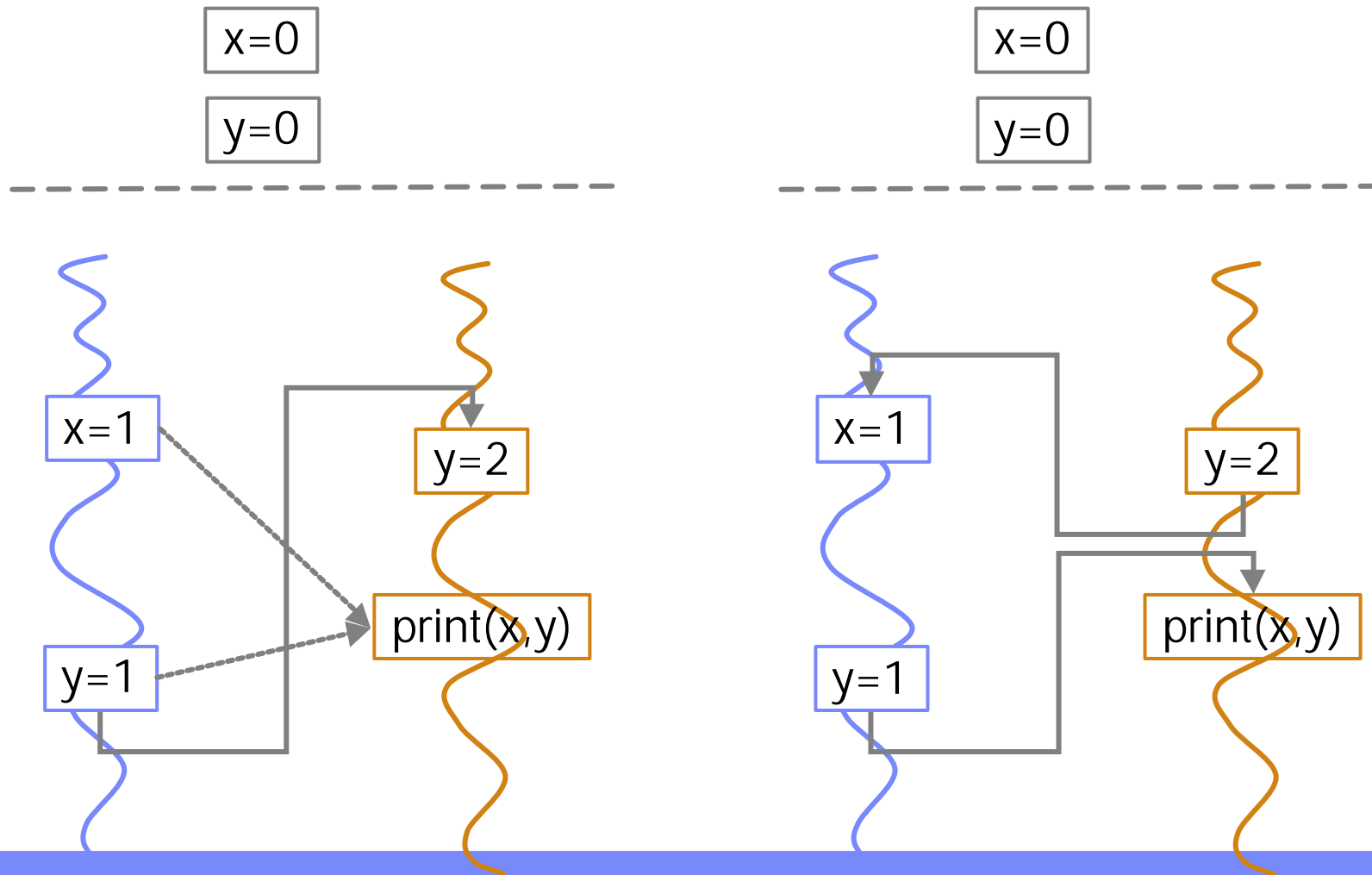


## Choosing among alternative pasts





## Choosing among alternative pasts



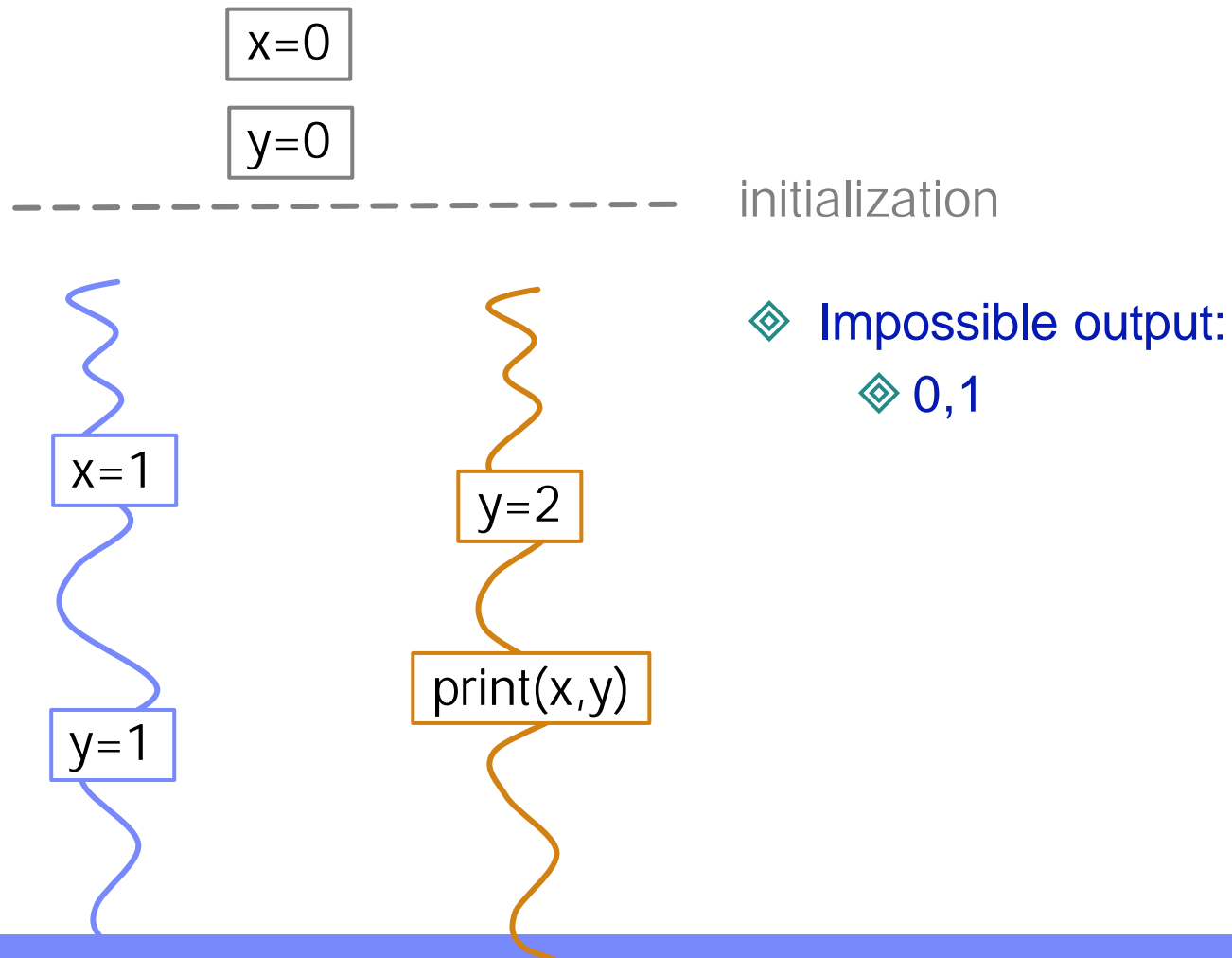


## Sound value substitutions

- ◆ Problem: illegal value choices
  - ◆ Values that are impossible to obtain in a legal run

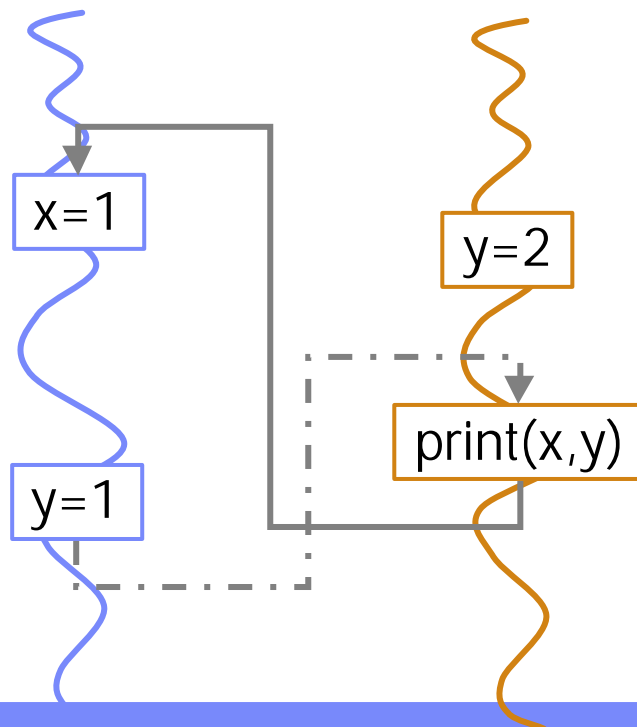


## Sound value substitutions





## Sound value substitutions



◇ Impossible output:  
◇ 0,1



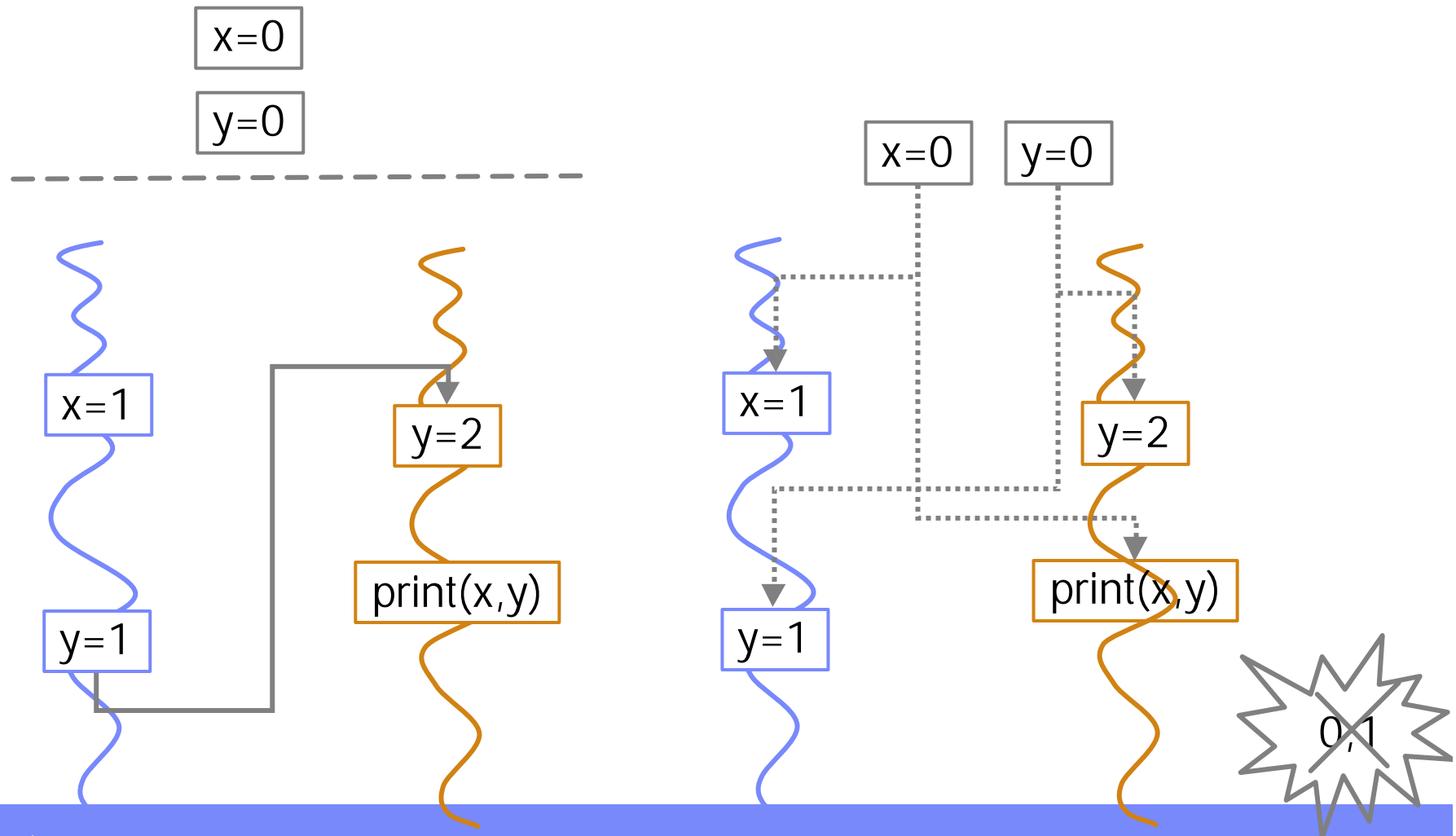


## Sound value substitutions

- ◇ Problem with value substitution: illegal value choices
  - ◇ Values that are impossible to obtain in a legal run
- ◇ How can we identify the sound choices?

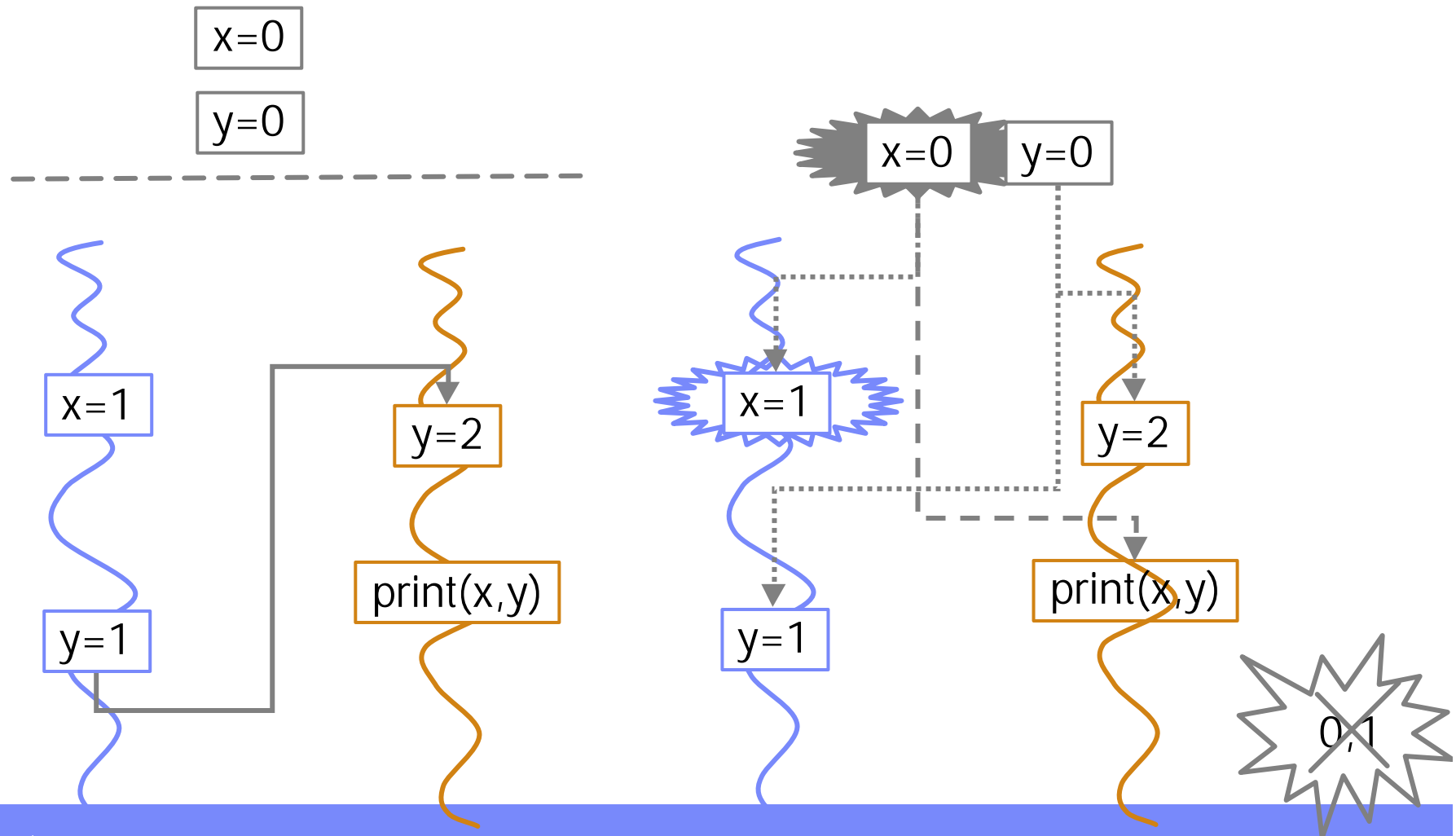


## Sound value substitutions



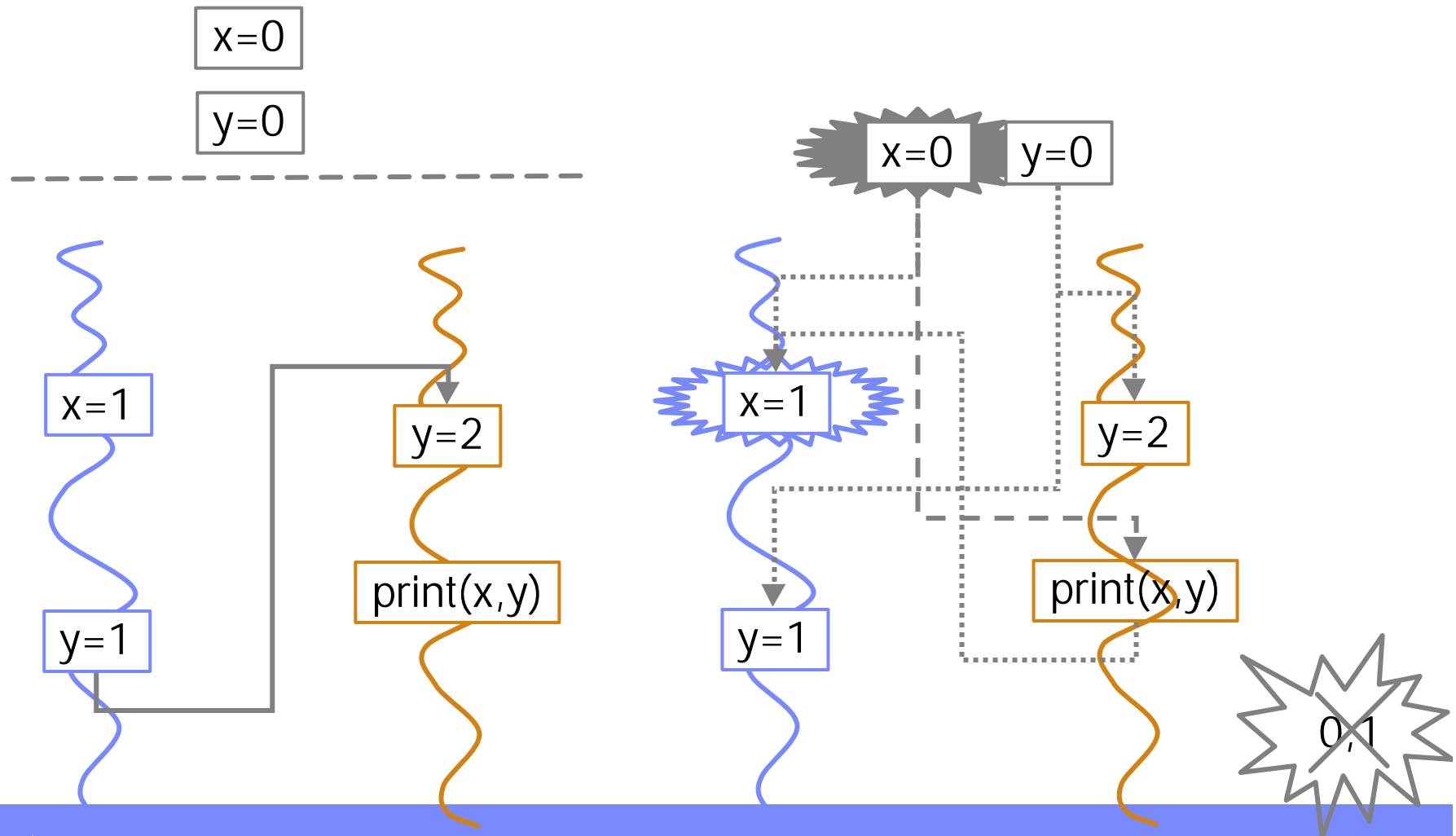


## Sound value substitutions



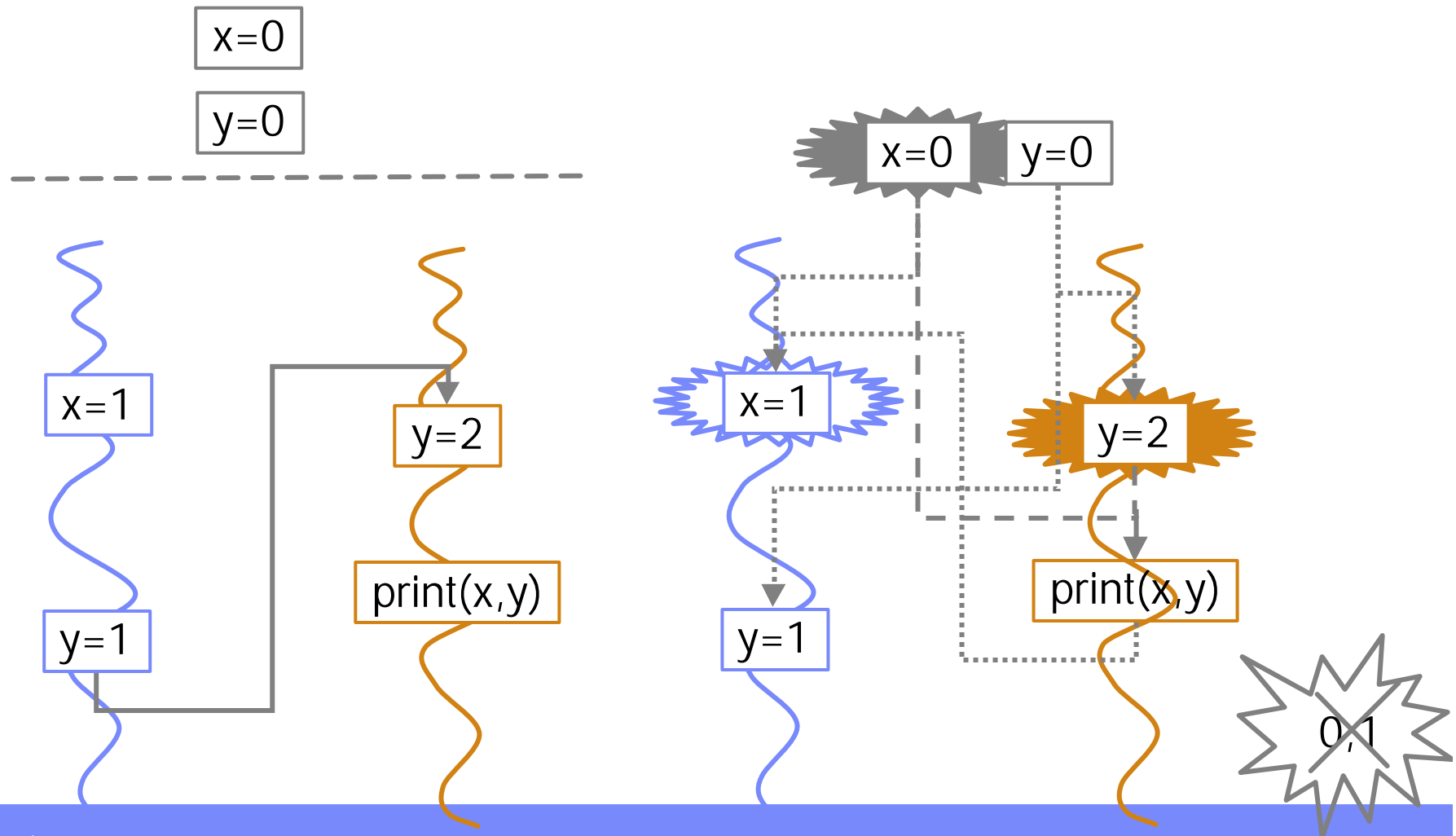


## Sound value substitutions





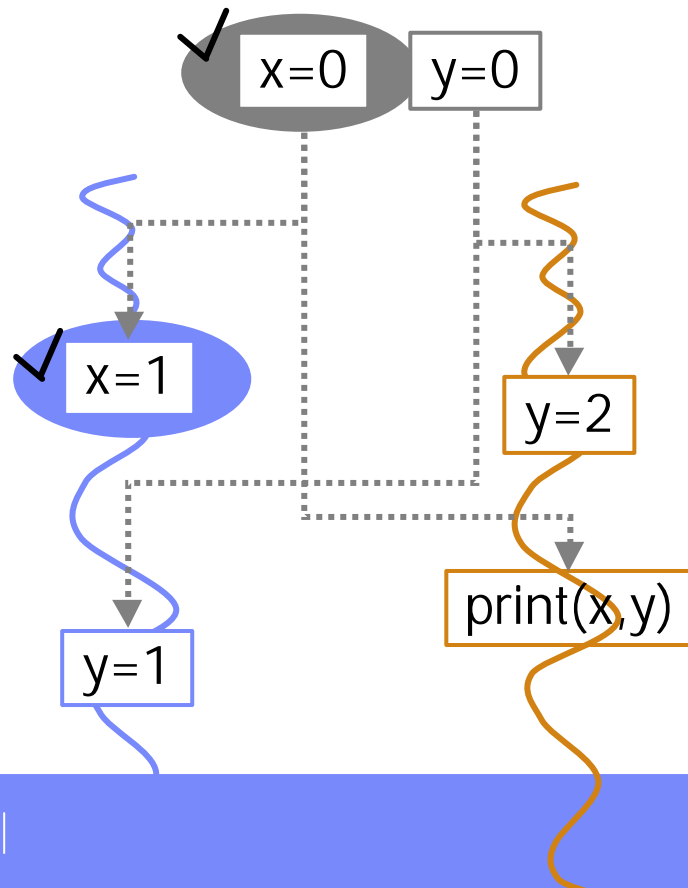
## Sound value substitutions





## Visibility

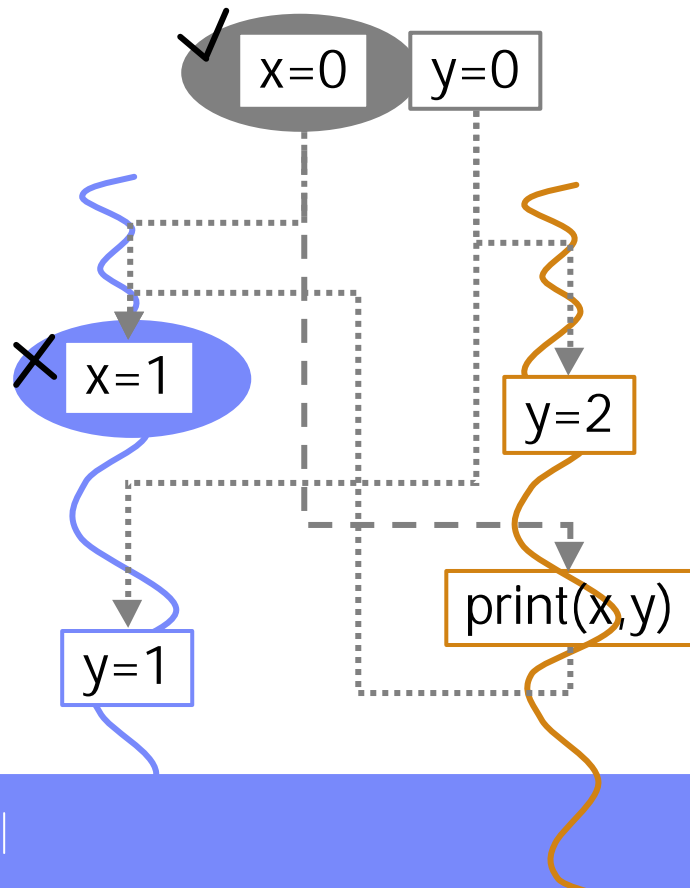
- ◇ A write event  $w$  is visible from a read event  $r$  if
  - ◇  $r$  does not precede  $w$





## Visibility

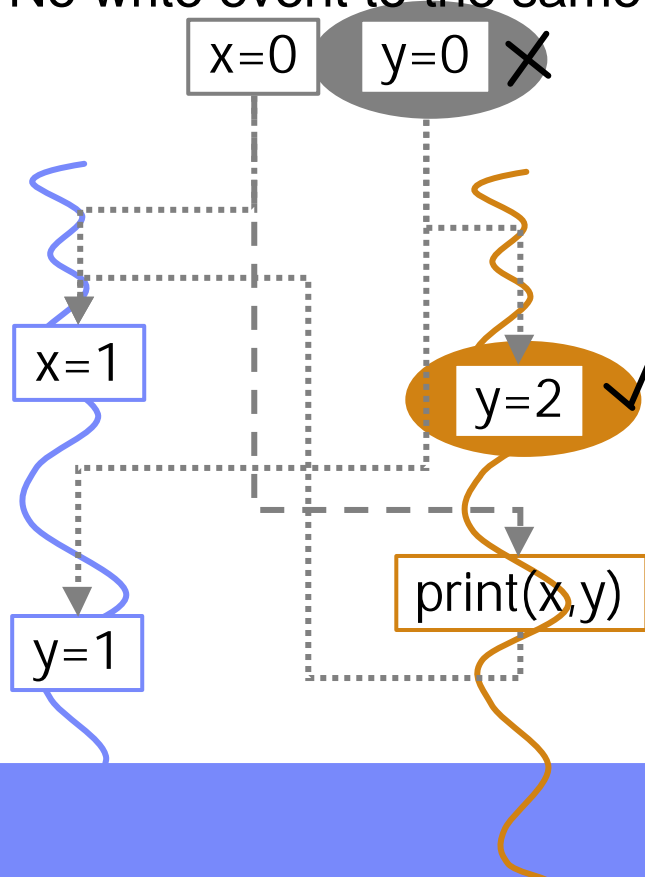
- ◇ A write event  $w$  is visible from a read event  $r$  if
  - ◇  $r$  does not precede  $w$





## Visibility

- ◆ A write event  $w$  is visible from a read event  $r$  if
  - ◆  $r$  does not precede  $w$
  - ◆ No write event to the same variable intervenes between  $w$  and  $r$





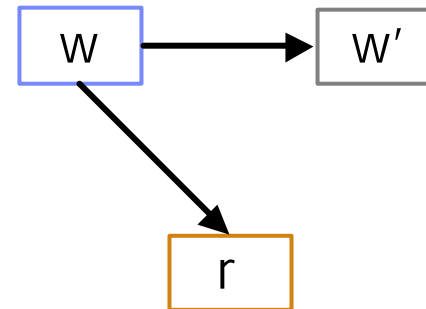
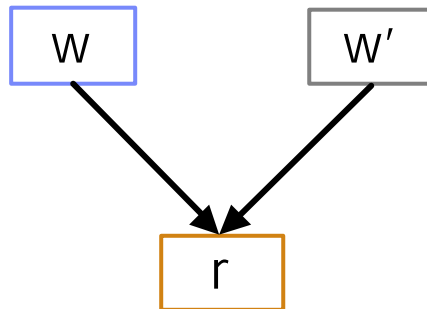


## Generating sound value substitutions

- ◆ When a thread event  $r$  requests value of a shared variable  $x$ 
  - ◆ Find all events  $w$  that write  $x$  and are visible from  $r$ 
    - ◆ There will always be such a  $w$  if the variables are initialized
  - ◆ Select one such  $w$  to be the value producer
  - ◆ Make all other  $w$ -s invisible from  $r$ 
    - ◆ How?

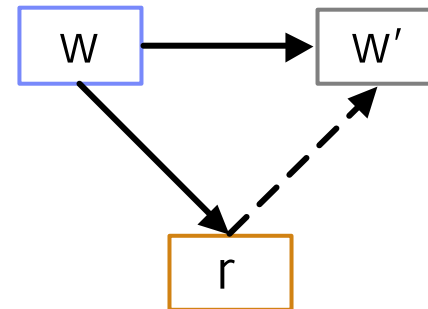
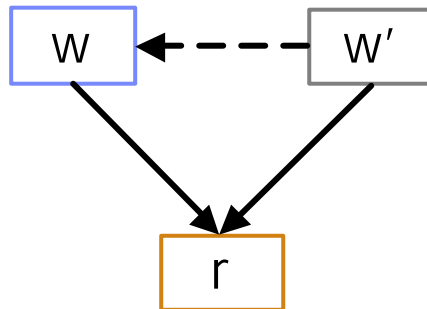


## Hiding the write event





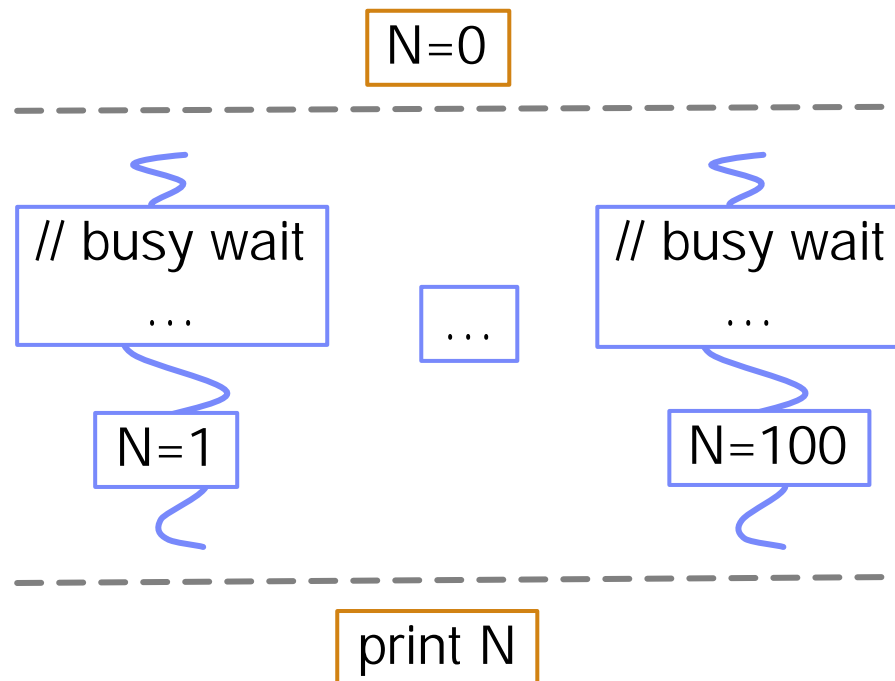
## Hiding the write event





## Conclusions

- ◆ Algorithm works fine for programs composed solely of reads/writes
- ◆ Compares favorably to other tools
  - ◆ Especially for long-distance races



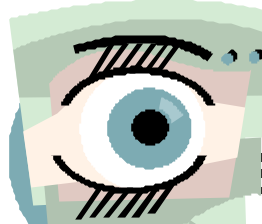
- ◆ **Normal execution:**  $N=100$
- ◆ **Noise-maker:** depends on heuristics and busy-wait
- ◆ **Alternative pasts:** 1...100 with equal probability



## Conclusions

- ◆ Algorithm works fine for programs composed solely of reads/writes
- ◆ Compares favorably to other tools
  - ◆ Especially for long-distance races
- ◆ Challenges:
  - ◆ Synchronized blocks
    - ◆ The position of the block is determined before the block is executed
    - ◆ Need static analysis to identify all reads/writes
  - ◆ Time and space consumption
    - ◆ Several ways to reduce the number of graphs and size of each graph
    - ◆ Slicing could help
    - ◆ Can just use insights to find new heuristics for noise-generation tools





There once was a man who said, "God  
Must think it exceedingly odd  
If He finds that this tree  
Continues to be  
When there's no one about in the Quad."

"Dear Sir:

Your astonishment's odd:  
I am always about in the Quad  
And that's why the tree  
Will continue to be,  
Since observed by,  
Yours faithfully,  
God."

