

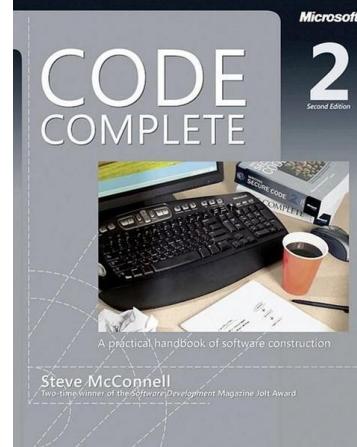
Rigorous Software Engineering

10.03.2021

Veselin Raychev

Defects in software

“Code Complete 2” book is a slightly outdated book (1990s-2000s) with a lot of great and still valid suggestions on how to organize a team and a software projects



The industry average of defect density at the time was about **15-50** errors per 1000 lines of delivered code

Microsoft in the 1990s was an absolute leader and doing better at about **10-20** defects per 1000 lines of code during in-house testing, and **0.5** in released code

Despite doing great then, Microsoft got severely bitten by security bugs in the early 2000s. The whole industry is certainly doing much better now. How?

Software development in the 1990-2000s

What was considered good then:

- Source control (CVS, not git)
- Bug tracking software
- Coding conventions
- Software testers clicking
- Maybe something. Many standard tools did not exist

A lot of projects did not have all of these. No surprise software was commonly with 15-50 errors per 1000 lines. We can still do this bad today :)

Research on understanding bugs

Code analytics was an active research area in the early 2000s. The goal is to collect metrics and statistics and even a machine learning model that would predict which projects are healthy, which code will be buggy or which changes will contain bugs

Some interesting findings

Code that is unnecessarily complex has more defects
(Example metric: [cyclomatic complexity](#))

Code that changes more often has more defects (**code churn**)

The interesting part of it ended after findings that the **best predictor** for bugs is usually project dependent ^[1,2]

Overall: soft metrics. They may help, but they are rarely actionable

And the **author** of the code is often the very best predictor

Not clear yet even if the chosen programming language is a good predictor

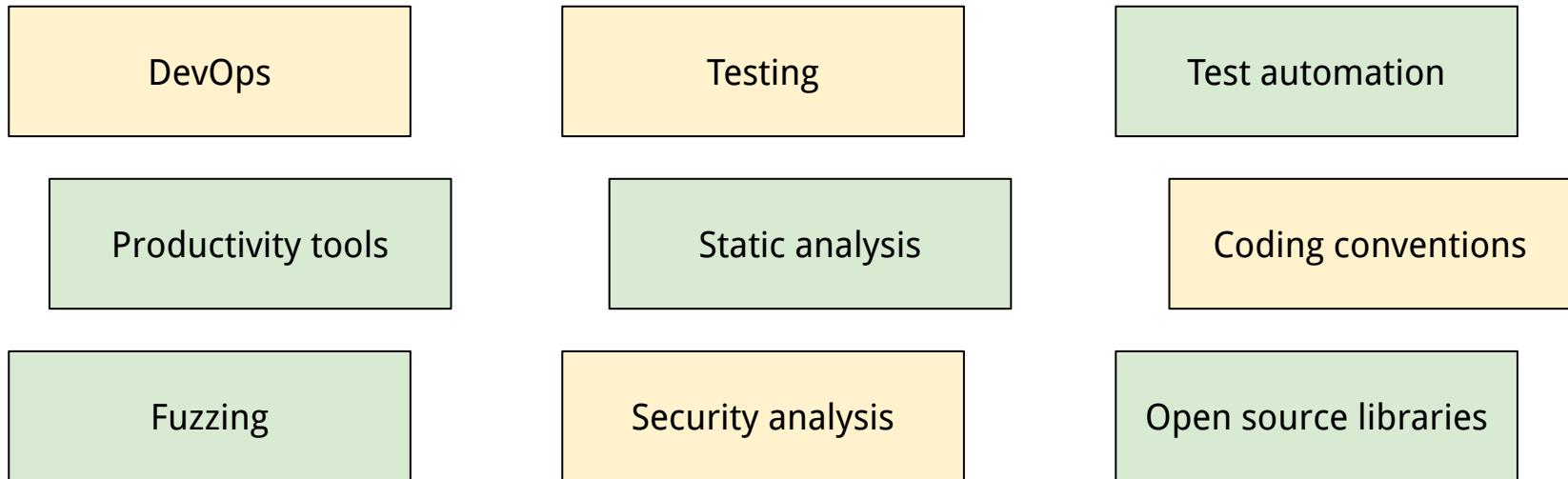
[1] Local vs. Global Models for Effort Estimation and Defect Prediction

[2] Failure is a Four-Letter Word – A Parody in Empirical Research –

Strategies against software defects

Usually defines policies that ensure software is: (all of these help a bit)

shipped on time low number of defects at right cost



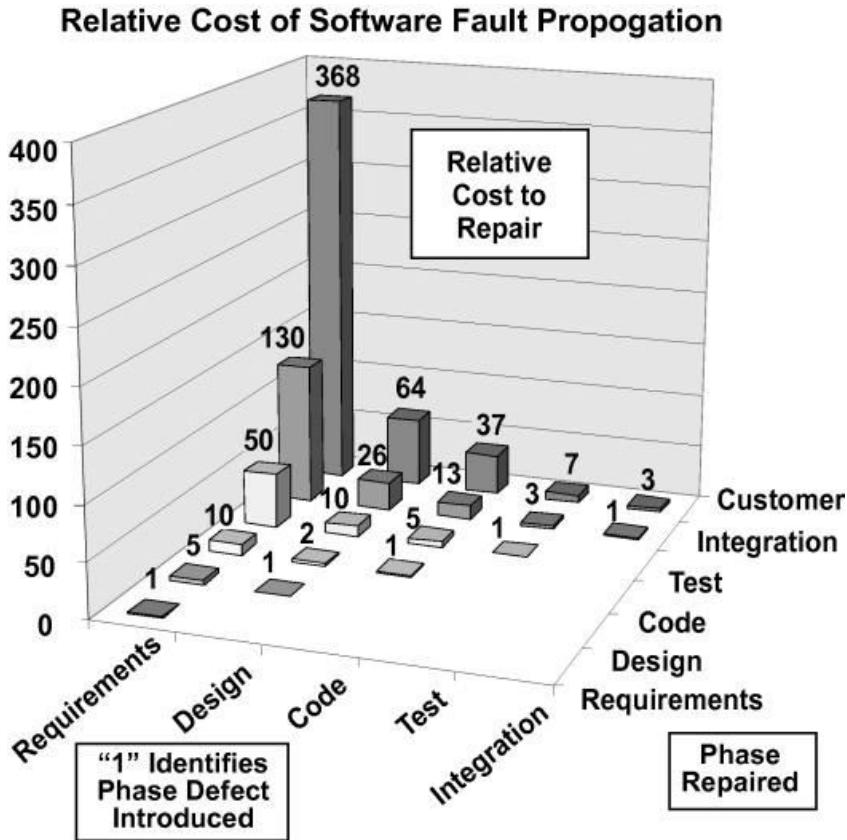
Many tools, little time

Too many tools and techniques

When working for a larger organization (e.g. FAANG), there may be responsible people for most of these pieces

Fewer software engineers usually means there will be shortcuts or more of these tools will come from different vendors

Cost of defects



Two main strategies against cost of bugs:

1. Make the high bars less high (e.g. faster deployment to customers)
2. Find and fix the defect earlier: also known as **shift left**

source:

L. Lazić N. Mastorakis. Cost effective software test metrics

Software shipping

Defects will be inevitable. Even spacecrafts now have the ability to do software updates

Defects may also arise due to changed requirements

Always plan how you will get information about usage of software in the wild
[crash reports, logs]

Then plan how to fix the shipped software
[cloud deployment, automatic updates]

DevOps

A set of practices that usually come with the expectation that the same team **develops** and then **operate** the software

Many sources of the ideas, but very popular among the web companies of last 15 years



Image source:

<https://neonrocket.medium.com/devops-is-a-culture-not-a-role-be1bed149b0>

Usually represents a sequence of actions:

- Ideally done in a quick cycle (release often)
- Sometimes some of these pieces may be quickly-hacked

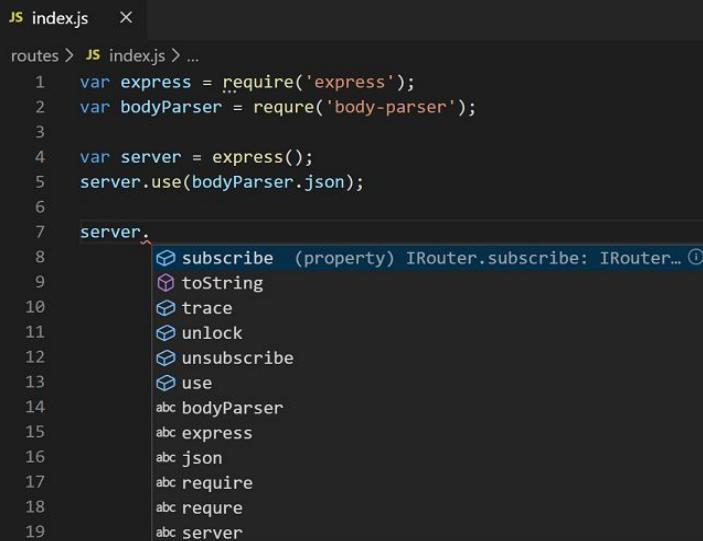
E.g. **YouTube** in its early days did not do must testing, but deployed new version and then watched if users complain on Twitter

1. Code

IDEs such as IntelliJ or Visual Studio Code or Atom or Eclipse or Emacs ...

Make it easier to type faster and avoid typos. Also great at code completion

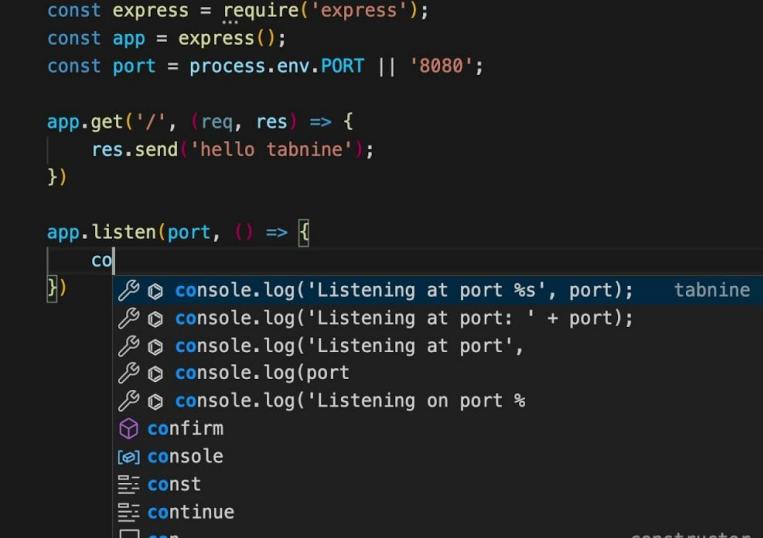
Type based code completion (e.g. in VSCode)



A screenshot of the VSCode interface showing a file named index.js. The code defines a server using the Express framework. A cursor is placed on the word 'server'. A dropdown menu shows various methods and properties available for the server object, such as 'subscribe', 'toString', 'trace', 'unlock', 'unsubscribe', and 'use'. The 'subscribe' method is highlighted.

```
JS index.js ×
routes > JS index.js > ...
1 var express = require('express');
2 var bodyParser = require('body-parser');
3
4 var server = express();
5 server.use(bodyParser.json());
6
7 server:
8     ⚭ subscribe (property) IRouter.subscribe: IRouter... ⓘ
9     ⚭ toString
10    ⚭ trace
11    ⚭ unlock
12    ⚭ unsubscribe
13    ⚭ use
14    abc bodyParser
15    abc express
16    abc json
17    abc require
18    abc require
19    abc server
```

Machine learning based code completion (e.g. tabnine)



A screenshot of the tabnine code completion interface. It shows a portion of an Express.js application. The code includes imports for express and process.env, and defines a get route that sends 'hello tabnine' as a response. Below this, there's a listen call. The completion dropdown is open at the end of the 'listen' call, showing several suggestions starting with 'co'. These suggestions include various logging statements like 'console.log' with different arguments, as well as other common Node.js functions like 'confirm', 'console', 'const', 'continue', and 'super'.

```
const express = require('express');
const app = express();
const port = process.env.PORT || '8080';

app.get('/', (req, res) => {
  res.send('hello tabnine');
}

app.listen(port, () => [
  co
])
  ⚭ console.log('Listening at port %s', port);    tabnine
  ⚭ console.log('Listening at port: ' + port);
  ⚭ console.log('Listening at port',
  ⚭ console.log(port
  ⚭ console.log('Listening on port %
  ⚭ confirm
  ⚭ console
  ⚭ const
  ⚭ continue
  ⚭ super
  ⚭ constructor
```

Static analysis tools (lint rules)

Enforce a coding style. May detect some bugs as well

Problems		Tasks	Console	Properties	Call Graph	Progress
Description		Resource				
▼ 13 errors, 0 warnings, 0 others						
▼	✖ Errors (13 items)					
✖	{ should almost always be at the end of the previous line [whitespace/braces] [4]					main.cpp
✖	{ should almost always be at the end of the previous line [whitespace/braces] [4]					main.cpp
✖	{ should almost always be at the end of the previous line [whitespace/braces] [4]					main.cpp
✖	{ should almost always be at the end of the previous line [whitespace/braces] [4]					main.cpp
✖	Add #include <vector> for vector<> [build/include_what_you_use] [4]					main.cpp
✖	Do not use namespace using-directives. Use using-declarations instead. [build/namespaces] [5]					main.cpp
✖	Do not use namespace using-directives. Use using-declarations instead. [build/namespaces] [5]					main.cpp
✖	Do not use namespace using-directives. Use using-declarations instead. [build/namespaces] [5]					main.cpp
✖	Line ends in whitespace. Consider deleting these extra spaces. [whitespace/end_of_line] [4]					main.cpp

Static analysis tools (rule based tools)

FindBugs is such a tool based on detecting anti-patterns in Java bytecode

eclipse-SDK-3.2-solaris-gtk

HelpDisplay.java in org.eclipse.help.internal.base

```
78     else if (href != null && (href.startsWith("tab=") //$/NON-NLS-1$  
79         || href.startsWith("toc") //$/NON-NLS-1$  
80         || href.startsWith("topic=") //$/NON-NLS-1$  
81     || href.startsWith("contextId="))) { //$/NON-NLS-1$ // assume it is a query string  
82         displayHelpURL(href, forceExternal);  
83     } else { // assume this is a topic  
84         if (getNoframesURL(href) == null) {  
85             try {  
86                 displayHelpURL(  
87                     "topic=" + URLEncoder.encode(href, "UTF-8"), forceExternal);  
88             } catch (UnsupportedEncodingException uee) {  
89             }  
90         } else if (href.startsWith("jar:file:")) { //$/NON-NLS-1$  
91             // topic from a jar to display without frames  
92         }  
93     }  
94 }
```

href could be null and is guaranteed to be dereferenced in org.eclipse.help.internal.base.HelpDisplay.displayHelpResource
At HelpDisplay.java:[line 78]
In method org.eclipse.help.internal.base.HelpDisplay.displayHelpResource(String,boolean) [Lines 71 - 98]
Local variable named href
Dereferenced at HelpDisplay.java:[line 87]
Dereferenced at HelpDisplay.java:[line 90]

Null value is guaranteed to be dereferenced
There is a statement or branch that if executed guarantees that a value is null at this point, and that value that is guaranteed to be dereferenced (except on forward paths involving runtime exceptions).

<http://findbugs.sourceforge.net/>

Nowadays most
Lint tools:

ESLint
PyLint
Clang-tidy

Static analysis tools (deeper analysis)

AI for Code

Find more serious defects, shift left

```
userService.getUserByUsername(req.params.username)
.then(user => {
  if(!user) {
    res.status(404).send({ message: 'Not found'});
    return Promise.reject('User not found');
  }
  util.doSomethingDangerous(user);
})
.catch(err => {
  logger.error(err);
});
```

Unsanitized input from **an HTTP parameter** [:12] **flows** [:12, :13, :20] into eval, where it is executed as JavaScript code. This may result in a Code Injection vulnerability.

 More info

</> Line 20

Also across files and modules

```
module.exports = {
  doSomethingDangerous: function(input) {
    eval(input);
  },
};
```



Security



Maintenance



Bug



Eval



Destructuring

2. Build

Often compiling in a controlled environment

Compilers often detect some errors like static analyzers (e.g. unused variables)

Q: What version of MySql is this going to use?

```
#include <mysql_driver.h>
#include <mysql_connection.h>
```

...

A: Depending on compiler flags. Usually the one installed on the computer. May not compile if the right packages were not installed

1. Making sure that everyone that builds your code has all the dependencies
2. Making sure they are the same

Dependency management:

e.g. pipenv for python, npm for JavaScript

Hermetic builds:

<https://bazel.build/>

Continuous build

Build after every commit

In pull requests

Part of most major repositories now:

GitLab

GitHub

Pull requests 498

Some checks haven't completed yet
15 in progress, 2 pending, and 25 successful checks

Firefox smoke test In progress — node scripts/functional_tests --bail --debug --kibana-install-dir /v... Details

Functional tests / Group 10 In progress — node scripts/functional_tests --debug --bail --kibana-ins... Details

Functional tests / Group 12 In progress — node scripts/functional_tests --debug --bail --kibana-ins... Details

Functional tests / Group 2 In progress — node scripts/functional_tests --debug --bail --kibana-inst... Details

Functional tests / Group 3 In progress — node scripts/functional_tests --debug --bail --kibana-inst... Details

Functional tests / Group 4 In progress — node scripts/functional_tests --debug --bail --kibana-inst... Details

This pull request can be automatically merged by project collaborators
Only those with write access to this repository can merge pull requests.

Block merging

Code review

3. Test

Build unit tests and integration tests

If software release is done rarely, this also involves manual testing

In the past, static analysis was expensive and it was run as part of testing, but the best tools of today are fast enough to run inside the IDEs

Dynamic analysis (testing)

Dynamic analysis executes the same code as in the program or in the test, but modifies the environment under which it executes.

Often, there is performance cost (e.g. 10x slower), but we can accept this when running a few tests

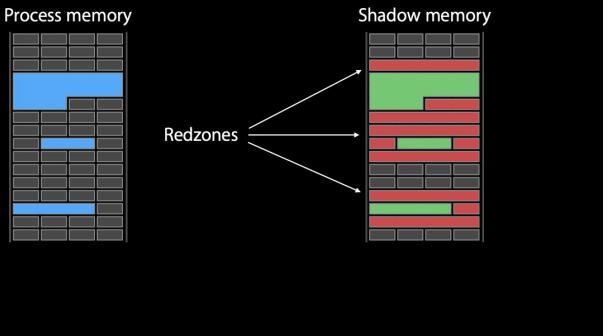
Alternatively, we can write the code to include a lot of checks that are only active in testing mode

```
#ifdef DEBUG
    if (!std::is_sorted(array.begin(), array.end())) fail();
#endif
```

Address sanitizer

Only 2x execution time overhead

Shadow Mapping



Tizen Manifest Editor basicuiasan.c

```
int f(int *a)
{
    int s = 0;
    for (int i = 0; i <= 10; i++)
    {
        s += a[10];
    }
    return s;
}

int
main(int argc, char *argv[])
{
    appdata_s ad = {0,};
    int ret = 0;

    int a[10];
    f(a);
}
```

Report

- stack-buffer-o
- read of size
- Inside f
- Inside f
- address 0x
- Inside f
- Inside f

Problems Console Log

basicuiasan [Tizen Native Application] C:\Users\Samsung\workspace\BasicUIASAN\Debug\basicui
f==15839==ERROR: AddressSanitizer: stack-buffer-overflow on address 0xbff63b18 a
READ of size 4 at 0xbff63b18 thread T0
#0 0xb7673929 in f in file C:\Users\Samsung\workspace\BasicUIASAN\Debug\..\
#1 0xb7673b67 in main in file C:\Users\Samsung\workspace\BasicUIASAN\Debug\/
#2 0xb6e15e4d in __libc_start_main (/lib/libc.so.6+0x17e4d)
#3 0xb7673727 in _start (/opt/usr/apps/org.example.basicuiasan/bin/basicui
Address 0xbff63b18 is located in stack of thread T0 at offset 88 in frame
#0 0xb767395f in main in file C:\Users\Samsung\workspace\BasicUIASAN\Debug\/
This frame has 4 object(s):
[16, 28) 'ad'
[48, 88) 'a' <== Memory access at offset 88 overflows this variable
[128, 148) 'event_callback'
[192, 212) 'handlers'
HINT: this may be a false positive if your program uses some custom stack unwind
(longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow in file C:\Users\Samsung\works
Shadow bytes around the buggy address:
0x37fec710: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x37fec720: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Images:

<https://docs.tizen.org/application/tizen-studio/native-tools/address-sanitizer/>

and

Apple WWDC 2015

Other sanitizers

Undefined behavior sanitizer (ubsan)

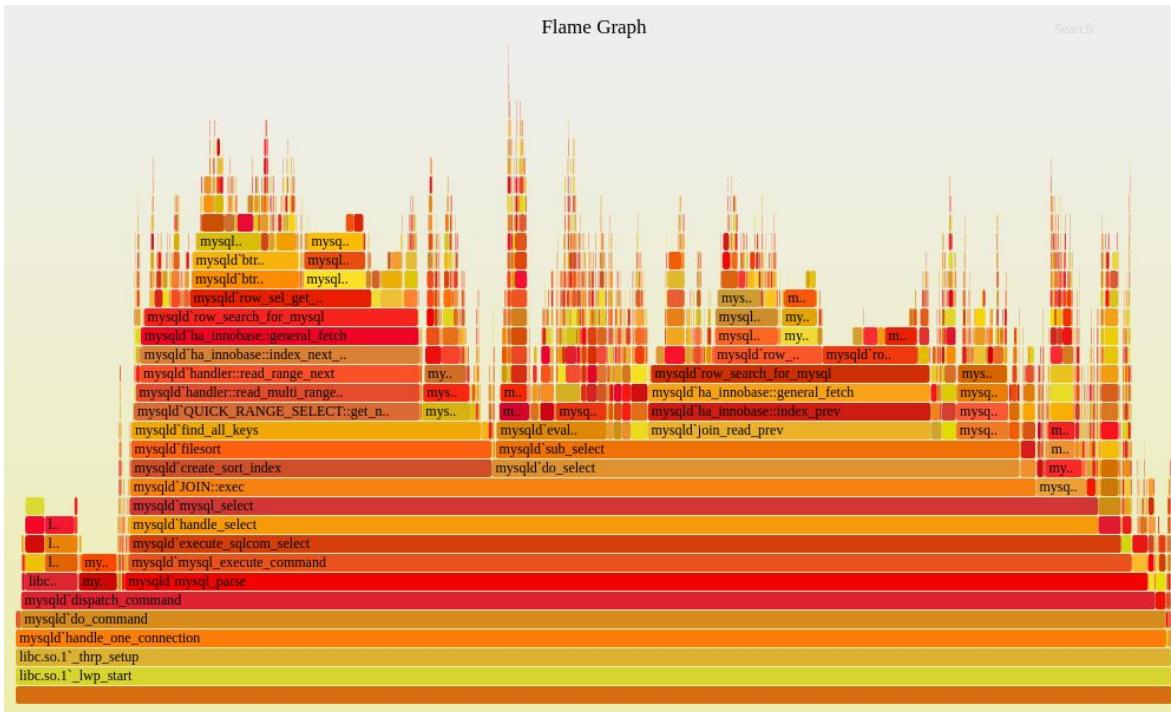
Thread sanitizer (tsan)

Race detectors are very effective in finding concurrency bugs

More info: <https://www.slideshare.net/sermp/sanitizer-cppcon-russia>

Performance testing

Profilers. Record the call stack with some frequency (e.g. every 1ms)



Usually not automated

But once hotspots are discovered:

Microbenchmarks

Testing that this part of the code doesn't get slower

Longer tests: fuzzing

Simple idea: generate [almost] random inputs, check if the program crashes.

Not every invalid memory address is a crash. Often combined with address sanitizer

Great resource: <https://www.fuzzingbook.org/>

Also this course

Release, Deploy and Operate

Part of operations

Often, the same tools used for automated testing and continuous build can be extended to do releases

Can be also as simple as a script to copy to a server and run

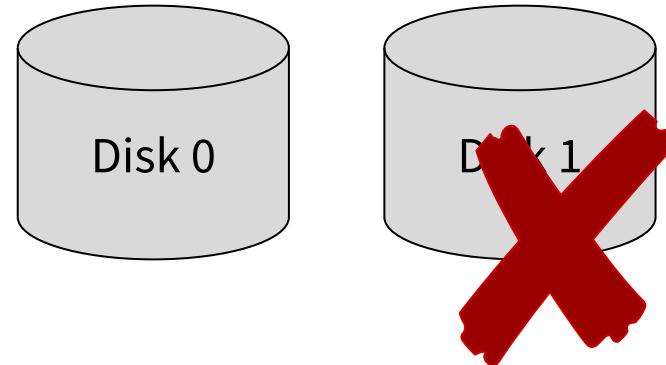
Cloud infrastructure is an entire space on its own. Outside of the scope of this course

Monitor

Always plan how you will get information about usage of software in the wild
[crash reports, logs, metrics]

Example: consider a distributed fault-tolerant storage that keeps the data replicated twice for faster reading and increased reliability

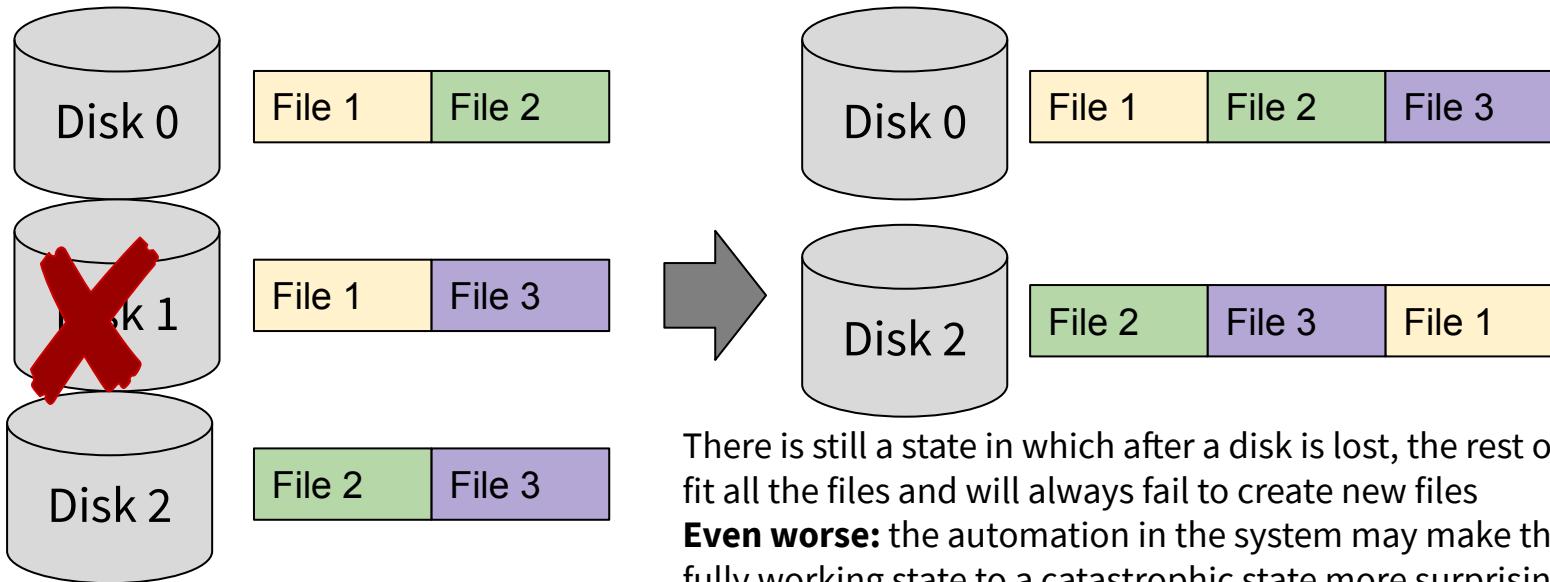
The system is **fault tolerant**. It still “works”, but is not reliable anymore. Needs a separate monitoring system to notify operator to put new disk drive



Monitor more automated systems

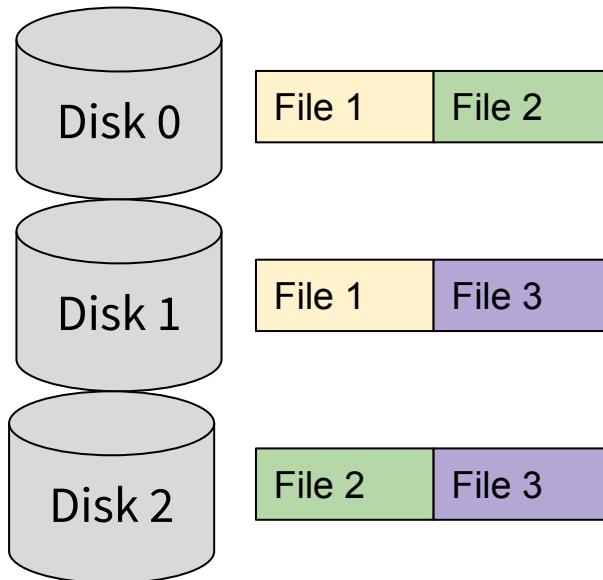
More automation doesn't remove need for monitoring

Now consider a more automated system that keeps at least two (or 3) copies of each file. If a disk dies, it automatically performs copies to maintain the replication



What to monitor?

Need to collect statistics of a running system



How full the disks are?
How many files do not have the sufficient number of replicas?
Do we have any replications in progress?
How large the queue for replication is?
Most importantly: when to raise an alert?

<https://prometheus.io/>

Time series database. Keeps track of metrics and how they move over time.

[Fun images from BorgMon, the Google internal version before Prometheus]

Site Reliability Engineering

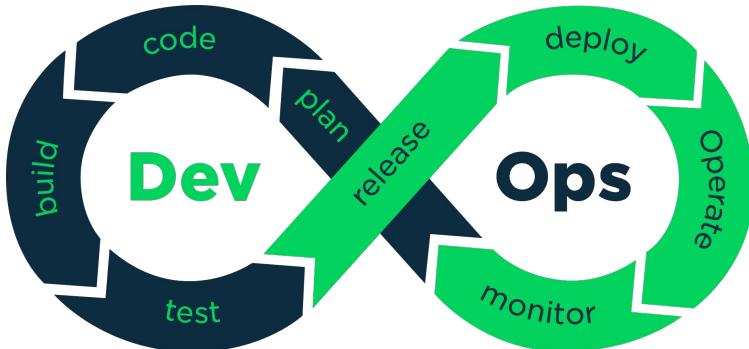
Automating the running of thousands of servers into reliable services

SRE vs SysAdmin

<https://sre.google/sre-book/table-of-contents/>

There is more

In all stages of development, we try to increase confidence the release will be run faster and we will not deploy defects. There are many more things to do. None of this is specific to DevOps, just widely used there.



- Code
 - Run static bug finding tools
- Build
 - Automate with multiple build environments
- Test
 - Canary (staging) testing. Have a separate instance of the site for internal testing only
 - Load testing
- Release
 - Keep track of all previous released configurations. Make it easy to roll-back to a previous version
- Deploy
 - Infrastructure as Code. Define services declaratively
- Operate
 - Build helper tools e.g. to block abusive users
- Monitor
 - Obtain metrics from users, not only from server

Rigorous Software Engineering

Modelling and Specification

Prof. Martin Vechev

- Logic
 - Static Models
 - Dynamic Models
 - Analyzing Models
-

Dynamic Behavior

- Alloy has no built-in model of execution
 - No notion of time or mutable state
- State or time have to be modeled explicitly

```
sig Array {  
    length: Int,  
    data: { i: Int | 0 <= i && i < length } -> lone E  
}
```

```
pred update[ a, a': Array, i: Int, e: E ] {  
    a'.length = a.length &&  
    a'.data = a.data ++ i -> e  
}
```

Describing Mutation via Different Atoms

- Alloy models describe operations declaratively
 - Relating the atoms before and after the operation

```
pred update[ a, a': Array, i: Int, e: E ] {  
    a'.length = a.length &&  
    a'.data = a.data ++ i -> e
```

Equality, not
assignment

A regular
identifier

Abstract Machine Idiom

- Move all relations and operations to a global state

```
sig State { ... }

pred op1[ s, s': State, ... ] { ... }

pred opn[ s, s': State, ... ] { ... }
```

- Operations modify the global state

Abstract Machine: Example

```
abstract sig FSObject { }
sig File, Dir extends FSObject { }
```

FileSystem is the global state

```
sig FileSystem {
    live: set FSObject,
    root: Dir & live,
    parent: (live - root) -> one (Dir & live),
    contents: (Dir & live) -> live
}
{
    contents = ~parent
    live in root.*contents
}
```

root is a directory in this file system

Every object except root has exactly one parent

Abstract Machine: Example (cont'd)

Precondition:
o is a live object
other than root

Remove o and
everything it
(transitively)
contains

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {  
    o in s.live - s.root &&  
    s'.live = s.live - o.*(s.contents) &&  
    s'.parent = s'.live <: s.parent  
}
```

Restrict domain
of parent
relation

Abstract Machine: Example (cont'd)

```
sig FileSystem {  
    live: set FSObject,  
    root: Dir & live,  
    parent: (live - root) -> one (Dir & live),  
    contents: (Dir & live) -> live  
}  
{  
    contents = ~parent  
    live in root.*contents  
}
```

Constraints ensure that
 $s.root = s'.root$
and that
 $s'.contents = \sim(s'.parent)$

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {  
    o in s.live - s.root &&  
    s'.live = s.live - o.*(s.contents) &&  
    s'.parent = s'.live <: s.parent  
}
```

Abstract Machine Idiom (cont'd)

- In static models, invariants are expressed as facts
- In dynamic models, invariants can be asserted as properties maintained by the operations

```
sig State { ... }

pred op1[ s, s': State, ... ] { ... }

pred opn[ s, s': State, ... ] { ... }

pred init[ s': State, ... ] { ... }

pred inv[ s: State ] { ... }
```

```
assert initEstablishes {
    all s': State, ... | init[ s', ... ] => inv[ s' ]
}

check initEstablishes

assert opiPreserves {
    all s, s': State, ... |
    inv[ s ] && opi[ s, s', ... ] => inv[ s' ]
}

check opiPreserves
```

Abstract Machine Example: Initialization

```
sig FileSystem {
    live: set FSObject,
    root: Dir & live,
    parent: (live - root) -> one (Dir & live),
    contents: (Dir & live) -> live
}
```

```
pred inv[ s: FileSystem ] {
    s.contents = ~ (s.parent)
    s.live in s.root.*(s.contents)
}
```

```
pred init[ s': FileSystem ] {
    #s'.live = 1
}
```

```
assert initEstablishes {
    all s': FileSystem |
    init[ s' ] => inv[ s' ]
}
check initEstablishes
```



Abstract Machine Example: Initialization (c'd)

```
sig FileSystem {  
    live: set FSObject,  
    root: Dir & live,  
    parent: (live - root) -> one (Dir & live),  
    contents: (Dir & live) -> live  
}
```

```
pred inv[ s: FileSystem ] {  
    s.contents = ~(s.parent)  
    s.live in s.root.*(s.contents)  
}
```

```
pred init[ s': FileSystem ] {  
    #s'.live = 1 &&  
    s'.contents[ s'.root ] = none  
}
```

```
assert initEstablishes {  
    all s': FileSystem |  
    init[ s' ] => inv[ s' ]  
}  
check initEstablishes
```

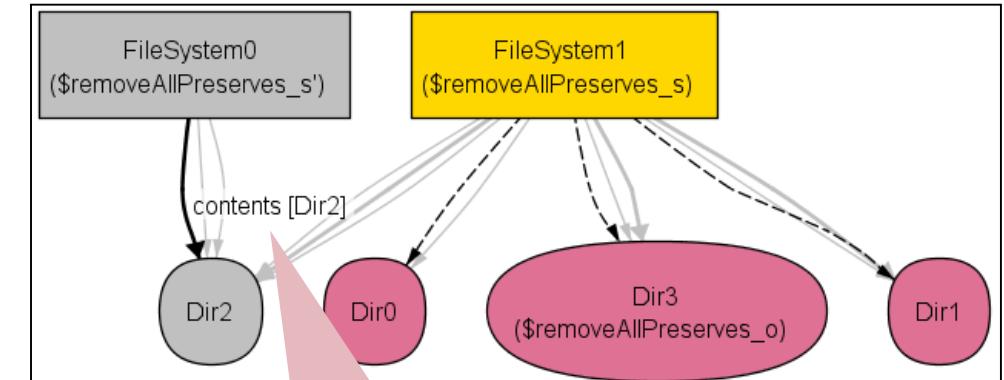


Abstract Machine Example: Preservation

```
pred inv[ s: FileSystem ] {
    s.contents = ~(s.parent)
    s.live in s.root.*(s.contents)
}
```

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {
    o in s.live - s.root &&
    s'.live = s.live - o.*(s.contents) &&
    s'.parent = s'.live <: s.parent
}
```

```
assert removeAllPreserves {
    all s, s': FileSystem, o: FSObject |
    inv[ s ] && removeAll[ s, s', o ] => inv[ s' ]
}
check removeAllPreserves
```



Constraints **no longer** ensure that
 $s'.contents = \sim(s'.parent)$

Abstract Machine Example: Preservation (c't)

```
pred inv[ s: FileSystem ] {  
    s.contents = ~(s.parent)  
    s.live in s.root.*(s.contents)  
}
```

```
pred removeAll[ s, s': FileSystem, o: FSObject ] {  
    o in s.live - s.root &&  
    s'.live = s.live - o.*(s.contents) &&  
    s'.parent = s'.live <: s.parent &&  
    s'.contents = s.contents :> s'.live  
}
```

```
assert removeAllPreserves {  
    all s, s': FileSystem, o: FSObject |  
    inv[ s ] && removeAll[ s, s', o ] => inv[ s' ]  
}
```

```
check removeAllPreserves
```



Temporal Invariants

- The invariants specified and modeled so far were one-state invariants
- Often, one needs to explore or check properties of sequences of states such as temporal invariants
 - 3. When the shared-field is true then shared, elems, and all elements of elems are immutable
- Model sequences of execution steps of an abstract machine ([execution traces](#))

Traces of an Abstract Machine

- Define a linear order on all states
 - First state is the initial state
 - Subsequent states are created by performing operations of the abstract machine

Initial state is
the first in the
order

```
open util/ordering[ State ]  
...  
fact traces {  
    init[ first ] &&  
    all s: State - last |  
        (some ... | op1[ s, s.next, ... ]) or  
        ...  
        (some ... | opn[ s, s.next, ... ])  
}
```

Parametric
library defines
linear order

Subsequent states
are created by one
of the operations

Properties of Traces

- One-state invariants can be asserted more conveniently
 - No separate initialization and preservation checks

```
assert invHolds {  
    all s: State | inv[ s ]  
}
```

- Temporal invariants can be expressed
 - Use `s.next`, `lt[s, s']`, or `lte[s, s']` to relate states

```
assert invtemp {  
    all s, s': FileSystem | s.root = s'.root  
}
```

- Logic
 - Static Models
 - Dynamic Models
 - Analyzing Models
-

Consistency and Validity

- An Alloy model specifies a collection of constraints C that describe a set of structures
- **Consistency:**
A formula F is consistent (satisfiable) if it evaluates to true in at least one of these structures

$$\exists s \bullet C(s) \wedge F(s)$$

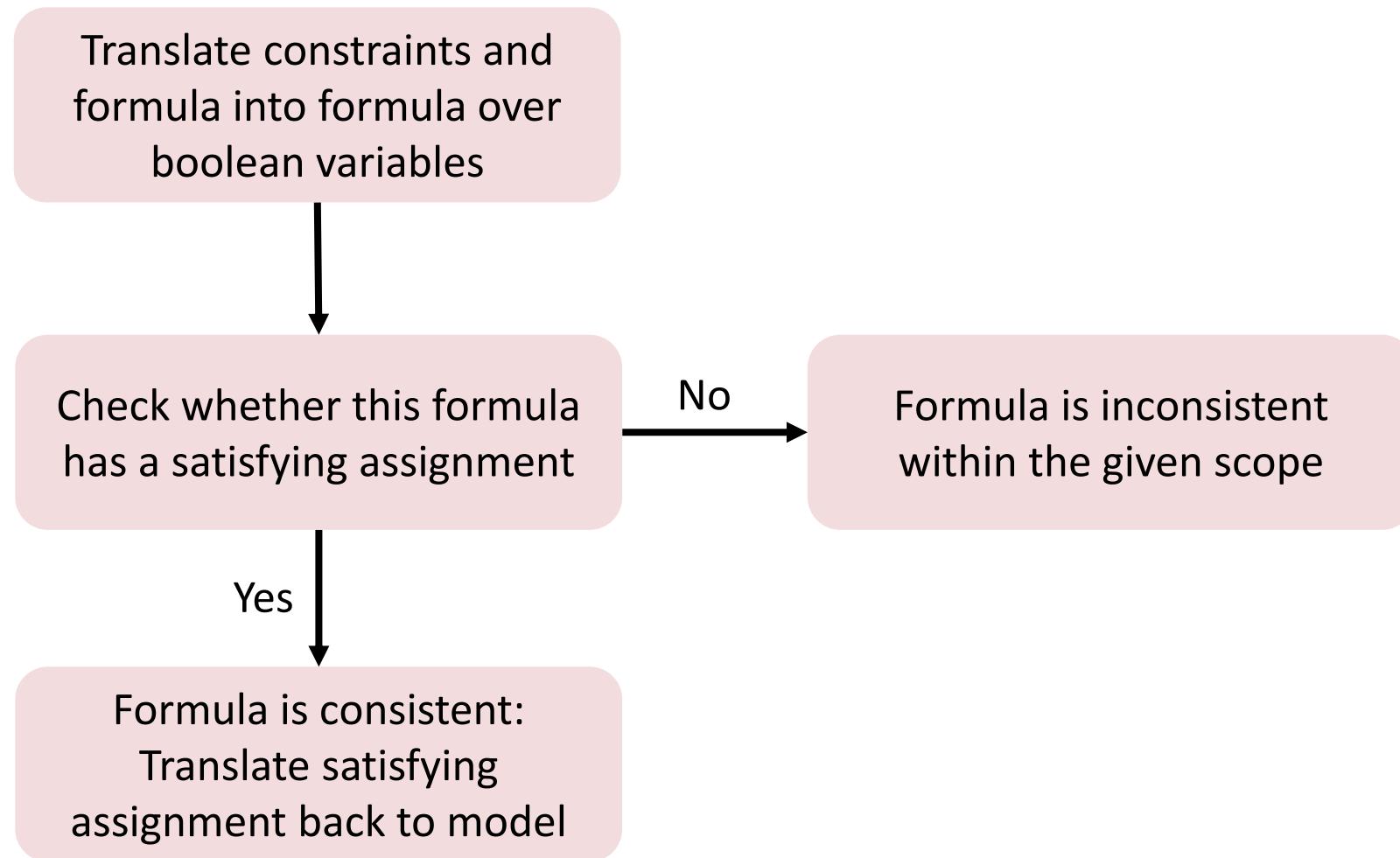
- **Validity:**
A formula F is valid if it evaluates to true in all of these structures

$$\forall s \bullet C(s) \Rightarrow F(s)$$

Analyzing Models within a Scope

- Validity and consistency checking for Alloy is undecidable
- The Alloy analyzer sidesteps this problem by checking validity and consistency [within a given scope](#)
 - A scope gives a [finite bound on the sizes of the sets](#) in the model (which makes everything else in the model also finite)
 - Naïve algorithm: enumerate all structures of a model within the bounds and check formula for each of them

Consistency Checking



Translation into Formula over Boolean Vars

- Internally, Alloy represents all data types as relations
 - A relation is a set of tuples

```
sig Node {  
    next: lone Node  
}
```

next is a binary
relation in
 $\text{Node} \times \text{Node}$

- Constraints and formulas in the model are represented as formulas over relations

```
fact {  
    all n: Node | n != n.next  
}
```

$\forall n \bullet (n, n) \notin \text{next}$

Translation into Boolean Formula (cont'd)

- A relation is translated into boolean variables
 - Introduce one boolean variable for each tuple that is potentially contained in the relation

```
sig Node {
    next: lone Node
}
pred show { }
run show for 3
```

next is a binary
relation in
 $\text{Node} \times \text{Node}$

$n_{00}, n_{01}, n_{02},$
 $n_{10}, n_{11}, n_{12},$
 n_{20}, n_{21}, n_{22}

For the given
scope, the next
relation may
contain nine
different tuples

- Constraints and formulas are translated into boolean formulas over these variables

```
fact {
    all n: Node | n != n.next
}
```

$$\neg(n_{00} \wedge n_{01}) \wedge \neg(n_{00} \wedge n_{02}) \wedge \neg(n_{01} \wedge n_{02}) \wedge \\ \neg(n_{10} \wedge n_{11}) \wedge \neg(n_{10} \wedge n_{12}) \wedge \neg(n_{11} \wedge n_{12}) \wedge \\ \neg(n_{20} \wedge n_{21}) \wedge \neg(n_{20} \wedge n_{22}) \wedge \neg(n_{21} \wedge n_{22}) \wedge \\ \neg n_{00} \wedge \neg n_{11} \wedge \neg n_{22}$$

Check for Satisfying Assignments

- Satisfiability of formulas over boolean variables is a well understood problem
 - Find a satisfying assignment if one exists and return UNSAT otherwise
 - The problem is NP-complete
- In practice, SAT solvers are extremely efficient

$$\begin{aligned} & \neg(n_{00} \wedge n_{01}) \wedge \neg(n_{00} \wedge n_{02}) \wedge \neg(n_{01} \wedge n_{02}) \wedge \\ & \neg(n_{10} \wedge n_{11}) \wedge \neg(n_{10} \wedge n_{12}) \wedge \neg(n_{11} \wedge n_{12}) \wedge \\ & \neg(n_{20} \wedge n_{21}) \wedge \neg(n_{20} \wedge n_{22}) \wedge \neg(n_{21} \wedge n_{22}) \wedge \\ & \quad \neg n_{00} \wedge \neg n_{11} \wedge \neg n_{22} \end{aligned}$$

n	0	1	2
0	F	F	F
1	F	F	T
2	F	T	F

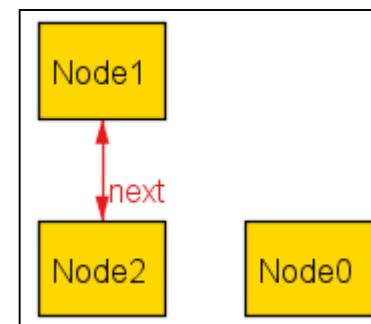
Translation Back to Model

- A satisfying assignment can be translated back to relations

n	0	1	2
0	F	F	F
1	F	F	T
2	F	T	F

next = { (1,2), (2,1) }

and then visualized



Interpretation of UNSAT

- If a boolean formula has no satisfying assignment, the SAT solver returns UNSAT
- The boolean formula encodes an Alloy model **within a given scope**
 - There are no structures within this scope,
but **larger structures may exist**

Validity and Invalidity Checking

- A formula F is valid if it evaluates to true in **all** structures that satisfy the constraints C of the model

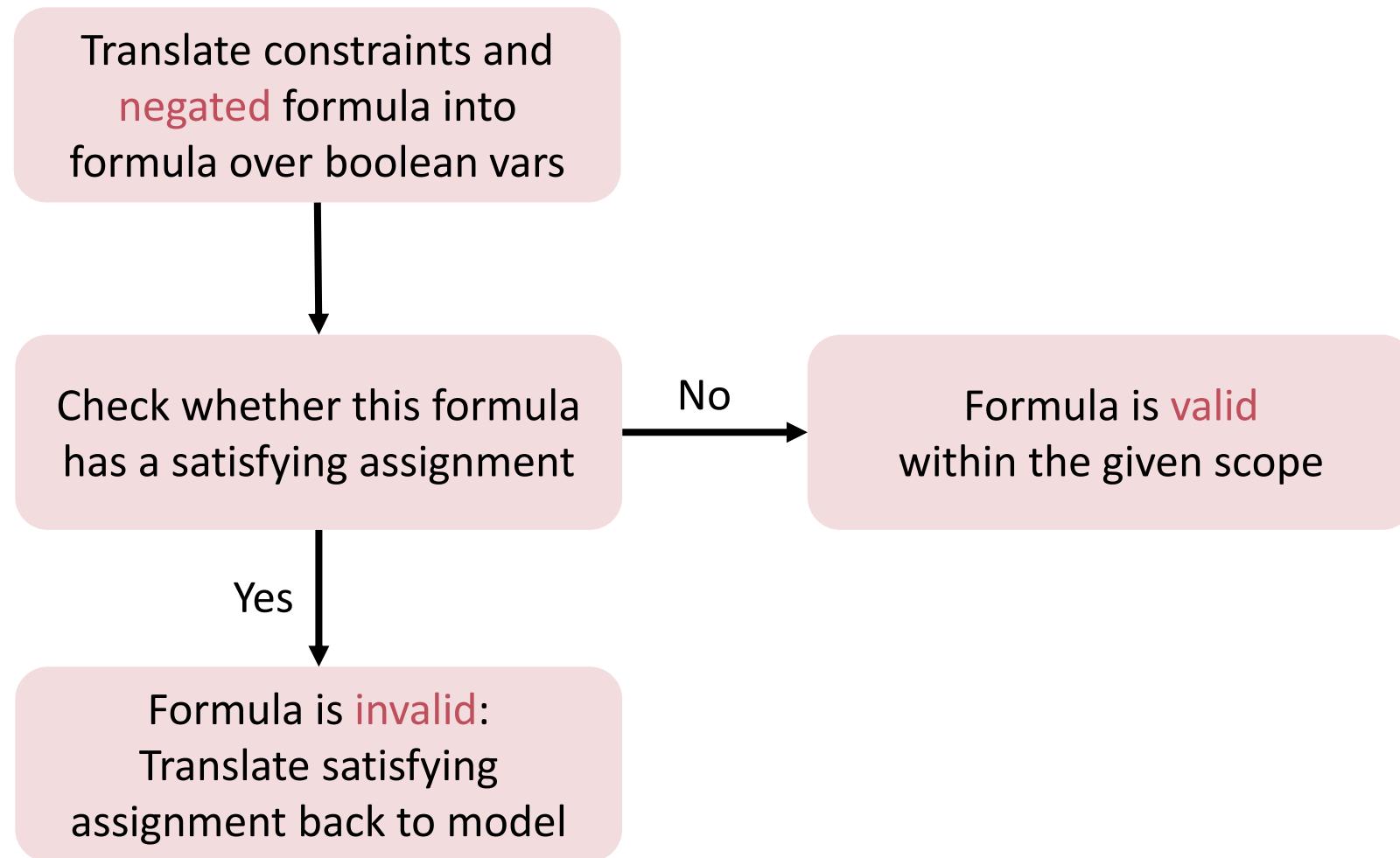
$$\forall s \bullet C(s) \Rightarrow F(s)$$

- Enumerating all structures within a given scope is possible, but would be too slow
- Instead of checking validity, the Alloy Analyzer **checks for invalidity**, that is, looks for **counterexamples**

This is a consistency check

$$\neg(\forall s \bullet C(s) \Rightarrow F(s)) \equiv (\exists s \bullet C(s) \wedge \neg F(s))$$

Validity Checking



Analyzing Models: Summary

- Consistency checking
 - Performed by **run** command within a scope
 - Positive answers are definite (structures)
- Validity checking
 - Performed by **check** command within a scope
 - Negative answers are definite (counterexamples)
- Small model hypothesis:
Most interesting errors are found by looking at small instances

Abstract Interpretation: Step 3

1. select/define an abstract domain
 - selected based on the type of **properties** you want to prove
2. define abstract semantics **for the language** w.r.t. to the domain
 - prove **sound** w.r.t **concrete semantics**
 - involves defining abstract transformers
 - that is, effect of statement / expression on the abstract domain
3. iterate abstract transformers over the abstract domain
 - until we reach a **fixed point**

F^i on our example

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

$$F^i(m)1 = \lambda v. [-\infty, \infty]$$

$$F^i(m)2 = \llbracket x := 5 \rrbracket_i(m(1))$$

$$F^i(m)3 = \llbracket y := 7 \rrbracket_i(m(2)) \sqcup \llbracket \text{goto 3} \rrbracket_i(m(6))$$

$$F^i(m)4 = \llbracket i \geq 0 \rrbracket_i(m(3))$$

$$F^i(m)5 = \llbracket y := y + 1 \rrbracket_i(m(4))$$

$$F^i(m)6 = \llbracket i := i - 1 \rrbracket_i(m(5))$$

$$F^i(m)7 = \llbracket i < 0 \rrbracket_i(m(3))$$

Fixed point of F^i

Let us compute the least fixed point of F^i

Iterate 0

The collection of these lines denote the current iterate. The iterate is a map

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 2: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 3: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 4: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$

Iterate 1

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 3: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 4: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$

Iterate 2

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 4: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$

Iterate 3

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$

Iterate 4

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7:  
}
```

Notice how we propagated to both
labels 4 and 7 *at the same time*

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 5

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 6

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 7

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 8

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, -1]$

Iterate 9

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, -1]$

Iterate 10

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, -1]$

Iterate 11

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, -1]$

Iterate 12

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, -1]$

Iterate 13

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 10]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 9]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, -1]$

Iterate 14

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 10]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 10]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, -1]$

Iterate 15

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 10]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 10]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 10]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 9]$, $i \rightarrow [-\infty, -1]$

Iterate 16

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 10]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 10]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 10]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 10]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 10]$, $i \rightarrow [-\infty, -1]$

The issue is that the iterates:

$$F^{i(0)}(\lambda \ell . \lambda v. \perp_i), F^{i(1)}(\lambda \ell . \lambda v. \perp_i), F^{i(2)}(\lambda \ell . \lambda v. \perp_i), \dots$$

will keep going on forever as the value of variable y will keep increasing. Hence, we will not be able to compute all of the iterates that we need in order to apply the fixed point theorem.

what should we do in this case ?

Generally, if we have a complete lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ and a monotone function F , if the height is infinite or the computation of the iterates of F takes too long, we need to find a way to **approximate** the least fixed point of F .

The interval domain and its function F^i is an example of this case.

We need to find a way to compute an element A such that:

$$\text{lfp}^{\sqsubseteq} F \sqsubseteq A$$

Widening Operator

The operator $\nabla: L \times L \rightarrow L$ is called a **widening** operator if:

- $\forall a, b \in L: a \sqcup b \sqsubseteq a \nabla b$ (widening approximates the join)
- if $x^0 \sqsubseteq x^1 \sqsubseteq x^2 \sqsubseteq \dots \sqsubseteq x^n \sqsubseteq \dots$ is an increasing sequence then $y^0 \sqsubseteq y^1 \sqsubseteq y^2 \sqsubseteq \dots \sqsubseteq y^n$ **stabilizes** after a **finite number of steps**

where $y^0 = x^0$ and $\forall i \geq 0: y^{i+1} = y^i \nabla x^{i+1}$

Note that widening is completely **independent of the function F.**

Much like the join, it is an operator defined for the **particular domain**.

Useful Theorem

If L is a complete lattice, $\sqvee: L \times L \rightarrow L$, $F: L \rightarrow L$ is monotone

Then the sequence:

$$y^0 = \perp$$

$$y^1 = y^0 \sqvee F(y^0)$$

$$y^2 = y^1 \sqvee F(y^1)$$

...

$$y^n = y^{n-1} \sqvee F(y^{n-1})$$

will stabilize after a finite number of steps with y^n being a **post-fixedpoint** of F .

By Tarski's theorem, we know that a post-fixedpoint is above the least fixed point. Hence, it follows that: $\text{lfp}^\sqsubseteq F \sqsubseteq y^n$

Widening for Interval Domain

Let us define a widening operator for the intervals

$$\nabla_i: \mathbb{L}^i \times \mathbb{L}^i \rightarrow \mathbb{L}^i$$

$[a, b] \nabla_i [c, d] = [e, f]$ where:

$$\begin{aligned} &\text{if } c < a, \text{ then } e = -\infty, \text{ else } e = a \\ &\text{if } d > b, \text{ then } f = \infty, \text{ else } f = b \end{aligned}$$

if one of the operands is \perp the result is the other operand.

The basic intuition is that if we see that an end point is unstable, we move its value to the extreme case.

Exercise: show this operator satisfies the conditions for widening.

Examples: widening for Interval

$$[1, 2] \ \nabla_i [0, 2] =$$

$$[0, 2] \ \nabla_i [1, 2] =$$

$$[1, 5] \ \nabla_i [1, 5] =$$

$$[2, 3] \ \nabla_i [2, 4] =$$

Examples: widening for Interval

$$[1, 2] \ \nabla_i [0, 2] = [-\infty, 2]$$

$$[0, 2] \ \nabla_i [1, 2] = [0, 2]$$

$$[1, 5] \ \nabla_i [1, 5] = [1, 5]$$

$$[2, 3] \ \nabla_i [2, 4] = [2, \infty]$$

Let us again consider our program with the Interval domain
but this time we will apply the widening operator

Iterate 0

$$\mathbf{it}^0 = \perp$$

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 2: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 3: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 4: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$

Iterate 1

$$\begin{aligned} \text{it}^1 &= \text{it}^0 \downarrow F(\text{it}^0) \\ &= \perp \downarrow F(\perp) \\ &= F(\perp) \end{aligned}$$

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty], y \rightarrow [-\infty, \infty], i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 3: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 4: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$

Iterate 2

$$\text{it}^2 = \text{it}^1 \triangleright F(\text{it}^1)$$

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 4: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$

Iterate 3

$$\text{it}^3 = \text{it}^2 \triangleright F(\text{it}^2)$$

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 5: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$

Iterate 4

Notice how we propagated to both
labels 4 and 7 **at the same time**

$$it^4 = it^3 \triangleright F(it^3)$$

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty], y \rightarrow [-\infty, \infty], i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5], y \rightarrow [-\infty, \infty], i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5], y \rightarrow [7, 7], i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5], y \rightarrow [7, 7], i \rightarrow [0, \infty]$
- 5: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 6: $x \rightarrow \perp_i, y \rightarrow \perp_i, i \rightarrow \perp_i$
- 7: $x \rightarrow [5, 5], y \rightarrow [7, 7], i \rightarrow [-\infty, -1]$

Iterate 5

$$\text{it}^5 = \text{it}^4 \triangleright F(\text{it}^4)$$

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$, $i \rightarrow \perp_i$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 6

$$\text{it}^6 = \text{it}^5 \triangleright F(\text{it}^5)$$

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 7: first compute $F(it^6)$

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7:  
}
```

$$it^7 = it^6 \triangleright F(it^6)$$

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, 8]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 7: then $\text{it}^6 \triangleright F(\text{it}^6)$

we have:

$$3: x \rightarrow [5, 5], \quad y \rightarrow [7, 7], \quad i \rightarrow [-\infty, \infty]$$



$$3: x \rightarrow [5, 5], \quad y \rightarrow [7, 8], \quad i \rightarrow [-\infty, \infty]$$

=

$$3: x \rightarrow [5, 5], \quad y \rightarrow [7, \infty], \quad i \rightarrow [-\infty, \infty]$$

Iterate 7: final result

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7:  
}
```

$$\text{it}^7 = \text{it}^6 \triangleright F(\text{it}^6)$$

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, 7]$, $i \rightarrow [-\infty, -1]$

Iterate 8

$$\text{it}^8 = \text{it}^7 \triangleright F(\text{it}^7)$$

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, -1]$

Iterate 9

$$\text{it}^9 = \text{it}^8 \triangleright F(\text{it}^8)$$

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, \infty]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, 8]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, -1]$

Iterate 10

$$\text{it}^{10} = \text{it}^9 \triangledown F(\text{it}^9)$$

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, \infty]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, \infty]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, -1]$

Iterate 11: a post fixed point is reached

$$\mathbf{it}^{11} = \mathbf{it}^{10} \triangleright F(\mathbf{it}^{10})$$

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7:  
}
```

- 1: $x \rightarrow [-\infty, \infty]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 2: $x \rightarrow [5, 5]$, $y \rightarrow [-\infty, \infty]$, $i \rightarrow [-\infty, \infty]$
- 3: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, \infty]$
- 4: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [0, \infty]$
- 5: $x \rightarrow [5, 5]$, $y \rightarrow [8, \infty]$, $i \rightarrow [0, \infty]$
- 6: $x \rightarrow [5, 5]$, $y \rightarrow [8, \infty]$, $i \rightarrow [-1, \infty]$
- 7: $x \rightarrow [5, 5]$, $y \rightarrow [7, \infty]$, $i \rightarrow [-\infty, -1]$

Chaotic (Asynchronous) Iteration

```
x1 := ⊥; x2 = ⊥; ...; xn = ⊥;  
W := {1,...,n};
```

```
while (W ≠ {}) do {  
    ℓ := removeLabel(W);  
    prevℓ := xℓ;  
    xℓ := prevℓ ∇ fℓ(x1,...,xn) ;  
    if (xℓ ≠ prevℓ)  
        W := W ∪ influence(ℓ);  
}
```

- W is the worklist , a set of labels left to be processed
- influence(ℓ) returns the set of labels where the value at those labels is influenced by the result at ℓ
- Re-compute only when necessary, thanks to influence (ℓ)
- Asynchronous computation can be parallelized

Rigorous Software Engineering

Modularity

Samuel Steffen

Slides adapted from previous years
(special thanks to Peter Müller, Felix Friedrich, Hermann Lehner)

Mastering Complexity

- *The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and Rule)* [Dijkstra, 1965]
- Benefits of decomposition
 - Partition the overall development effort
 - Support independent testing and analysis !
 - Decouple parts of a system so that changes to one part do not affect other parts
 - Permit system to be understood as a composition of mind-sized chunks
 - Enable reuse of components

Modularity

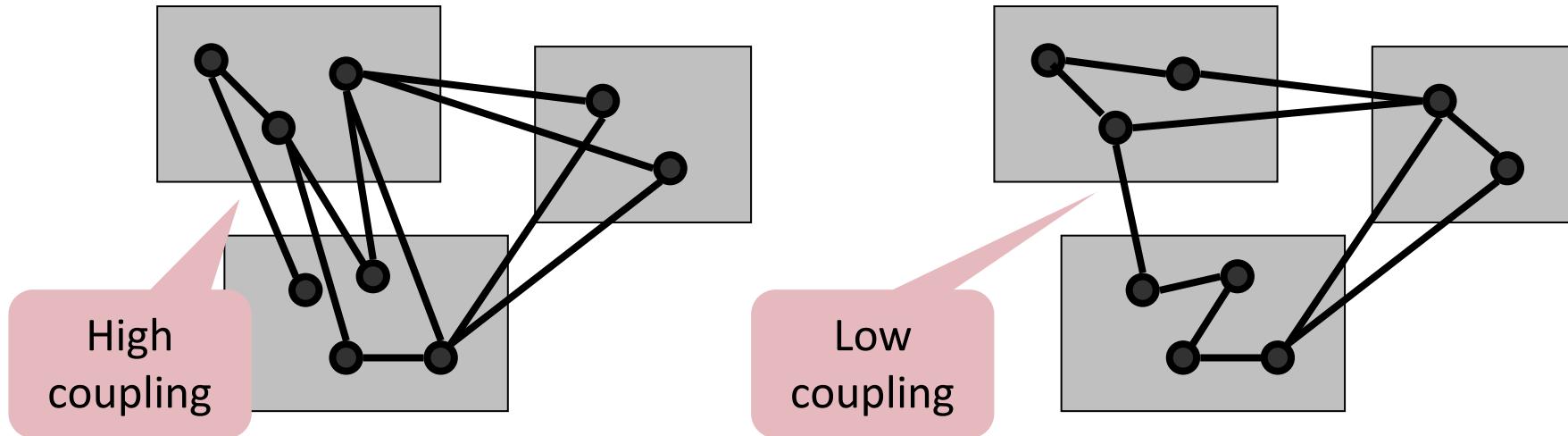
1. Coupling

- Data Coupling
- Procedural Coupling
- Class Coupling

2. Adaptation

Coupling

- Coupling measures **interdependence** between different **modules**



- Tightly-coupled modules **cannot** be developed, tested, changed, understood, or reused in isolation

Coupling (cont'd)

- Low coupling is a key concern when developing correct and maintainable software
- Next, we'll discuss common coupling problems and general approaches to avoid these

Modularity

1. Coupling

- Data Coupling
- Procedural Coupling
- Class Coupling

2. Adaptation

Representation Exposure

- Modules that expose their internal data representation become tightly coupled to their clients

```
class Coordinate {  
    public double radius, angle;  
  
    public double getX( ) {  
        return Math.cos( angle ) * radius;  
    }  
}
```

Representation Exposure: Problems

- Data representation is **difficult to change** during maintenance
- Modules cannot maintain strong invariants
- Concurrency requires complex synchronization

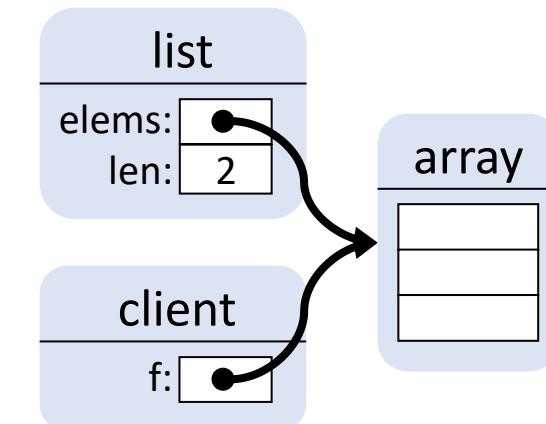
```
class Coordinate {  
    public double radius, angle;  
    invariant 0 <= radius;  
  
    public double getX( ) {  
        synchronized( this ) {  
            return Math.cos( angle ) * radius;  
        }  
    }  
}
```

```
class Coordinate {  
    public double x,y;  
  
    public double getX( ) { return x; }  
}
```

Representation Exposure: Problems (cont'd)

- Data representations often include **sub-objects** or entire sub-object structures
- Exposing sub-objects may lead to **unexpected side effects**

```
class List<E> {  
    E[ ] elems;  
    int len;  
  
    void set( int index, E e )  
    { elems[ index ] = e; }  
  
    E[ ] toArray( )  
    { return elems; }  
}
```



Approach 1: Restricting Access to Data

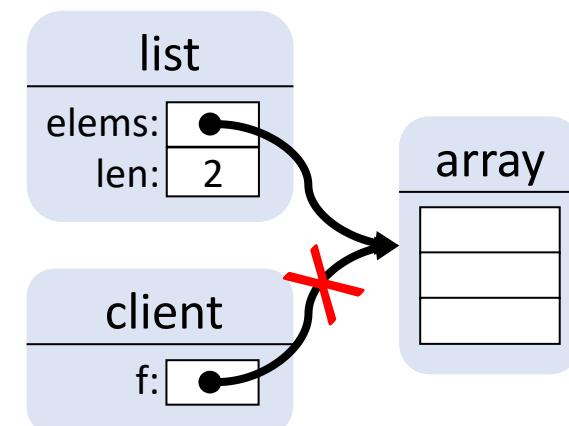
- Force clients to access the data representation through a **narrow interface**
- **Information hiding:** Hide implementation details behind interface
- Use interface for necessary **checks**

```
class Coordinate {  
    private double radius, angle;  
    invariant 0 <= radius;  
  
    public void setRadius( double r )  
        requires 0 <= r;  
    { synchronized( this ) { radius = r; } }  
  
    public double getX( ) {  
        synchronized( this ) {  
            return Math.cos( angle ) * radius;  
        }  
    }  
}
```

Approach 1: Restricting Access to Data (cont'd)

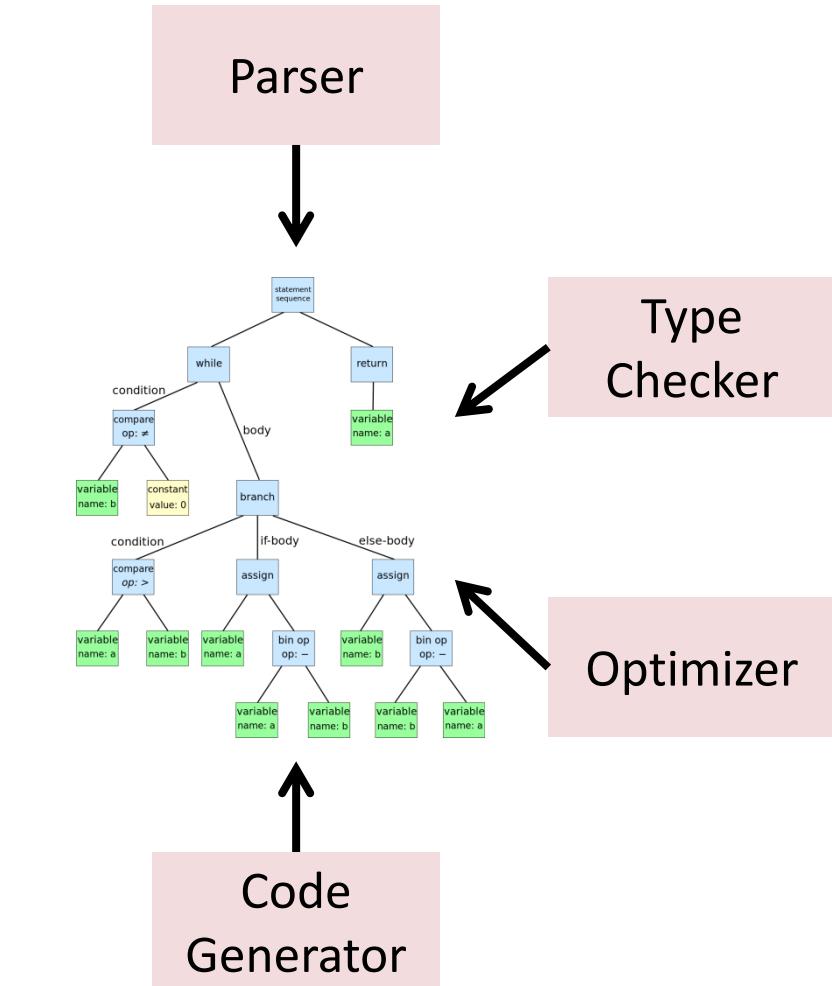
- Avoid exposure of sub-objects
- No leaking: Do not return references to sub-objects
- No capturing: Do not store arguments as sub-objects
- Clone objects if necessary

```
class List<E> {  
    E[ ] elems;  
    int len;  
  
    E[ ] toArray( ) {  
        E[ ] res = new E[ len ];  
        System.arraycopy( ... );  
        return res;  
    }  
}
```



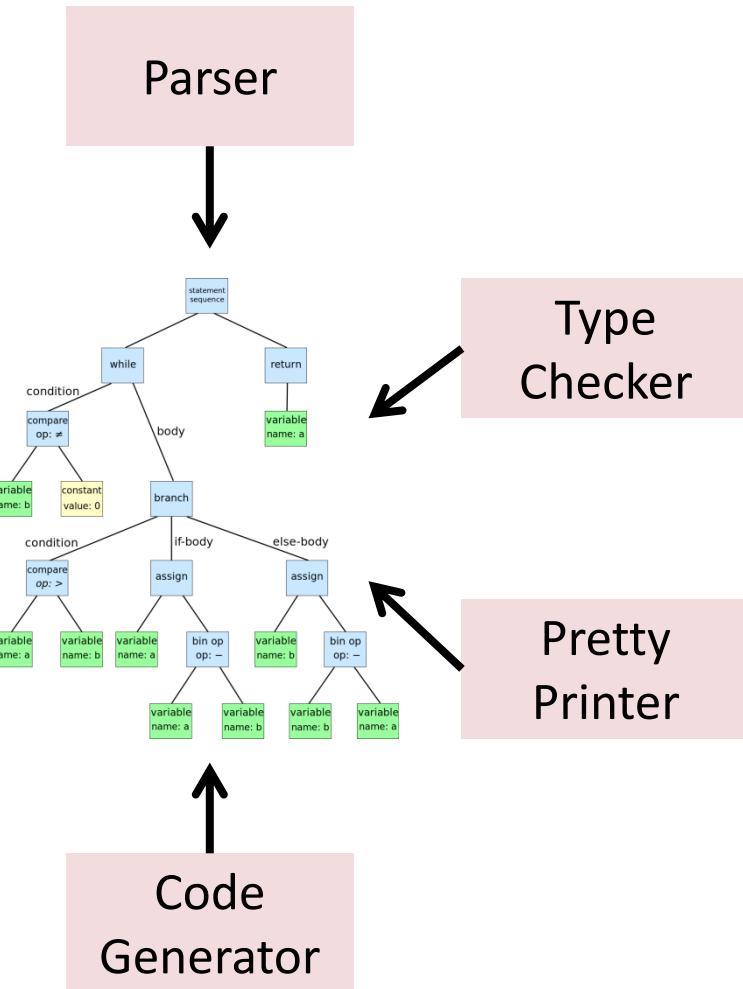
Shared Data Structures

- Modules get coupled by operating on **shared** data structures
 - Example: compiler working on syntax tree
- Problems caused by
 - Changes in data representation
 - Unexpected side effects
 - Concurrency



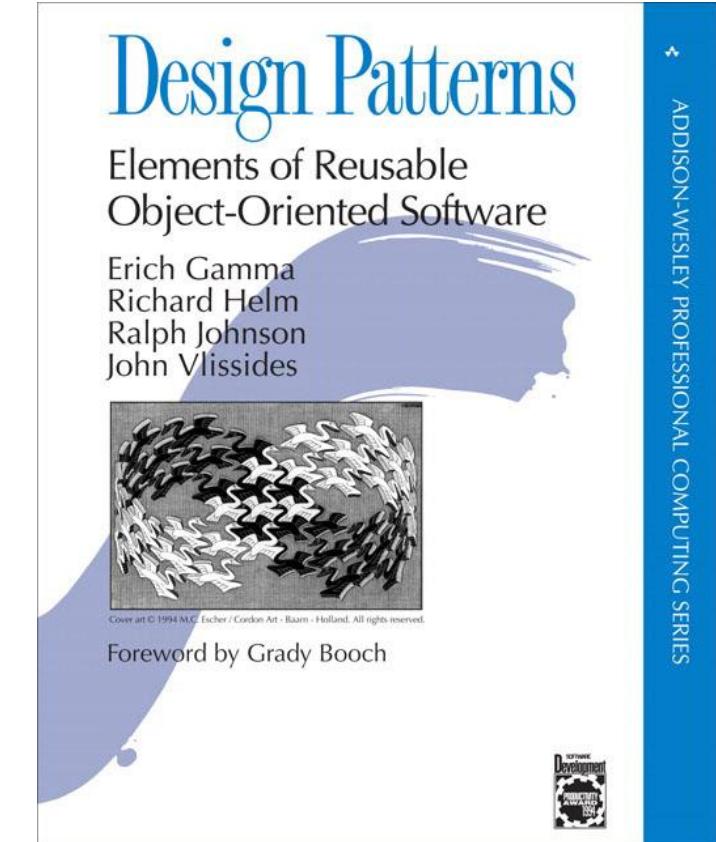
Approach 2: Making Shared Data Immutable

- Some drawbacks of shared data only apply to **mutable** shared data
 - Maintaining invariants
 - Thread synchronization
 - Unexpected side effects
- However:
 - Changing the data representation remains a problem
 - Copies can lead to run-time and memory overhead



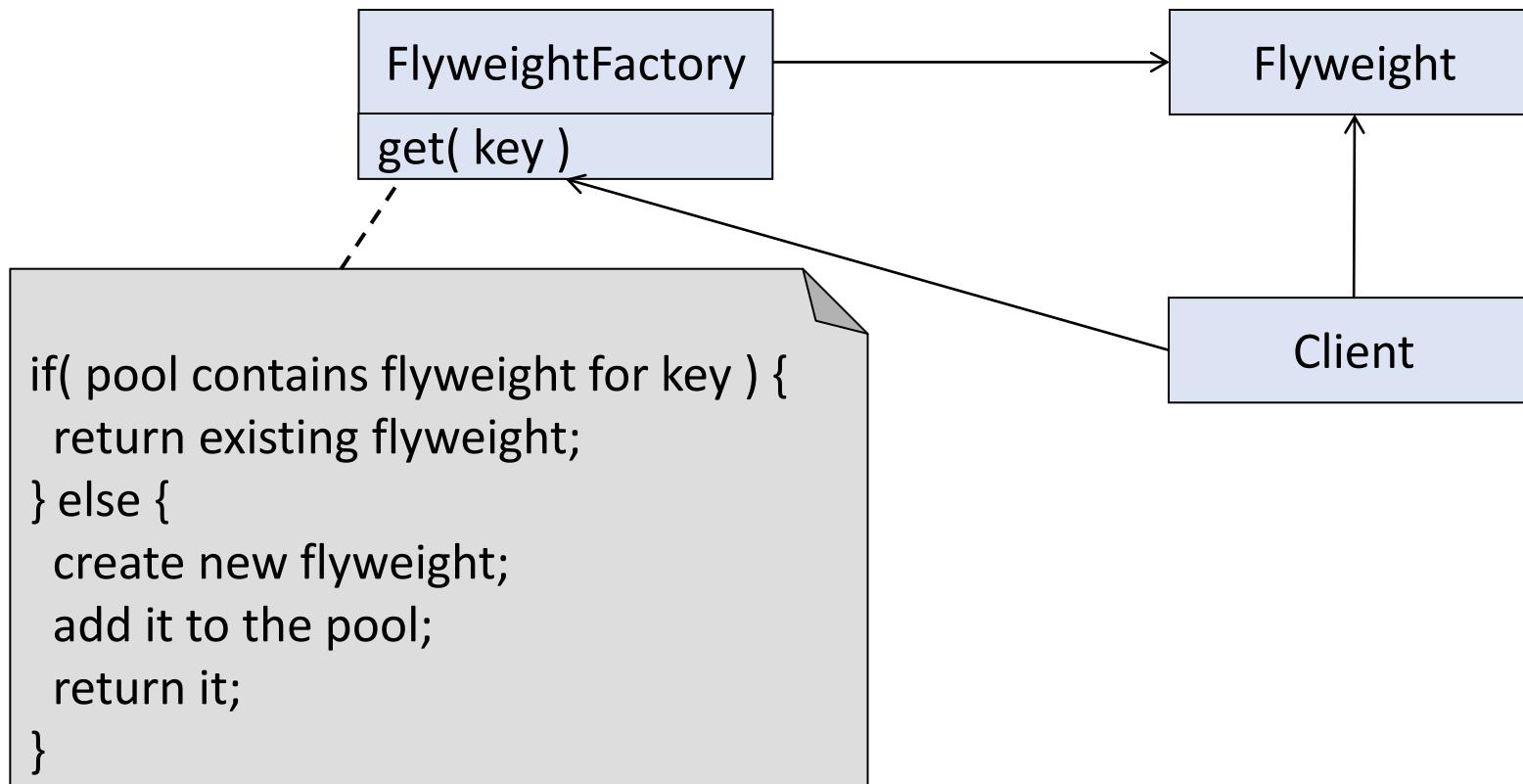
Design Patterns

- Design patterns are **general, reusable solutions** to commonly occurring design problems
- Capture best practices in detailed design



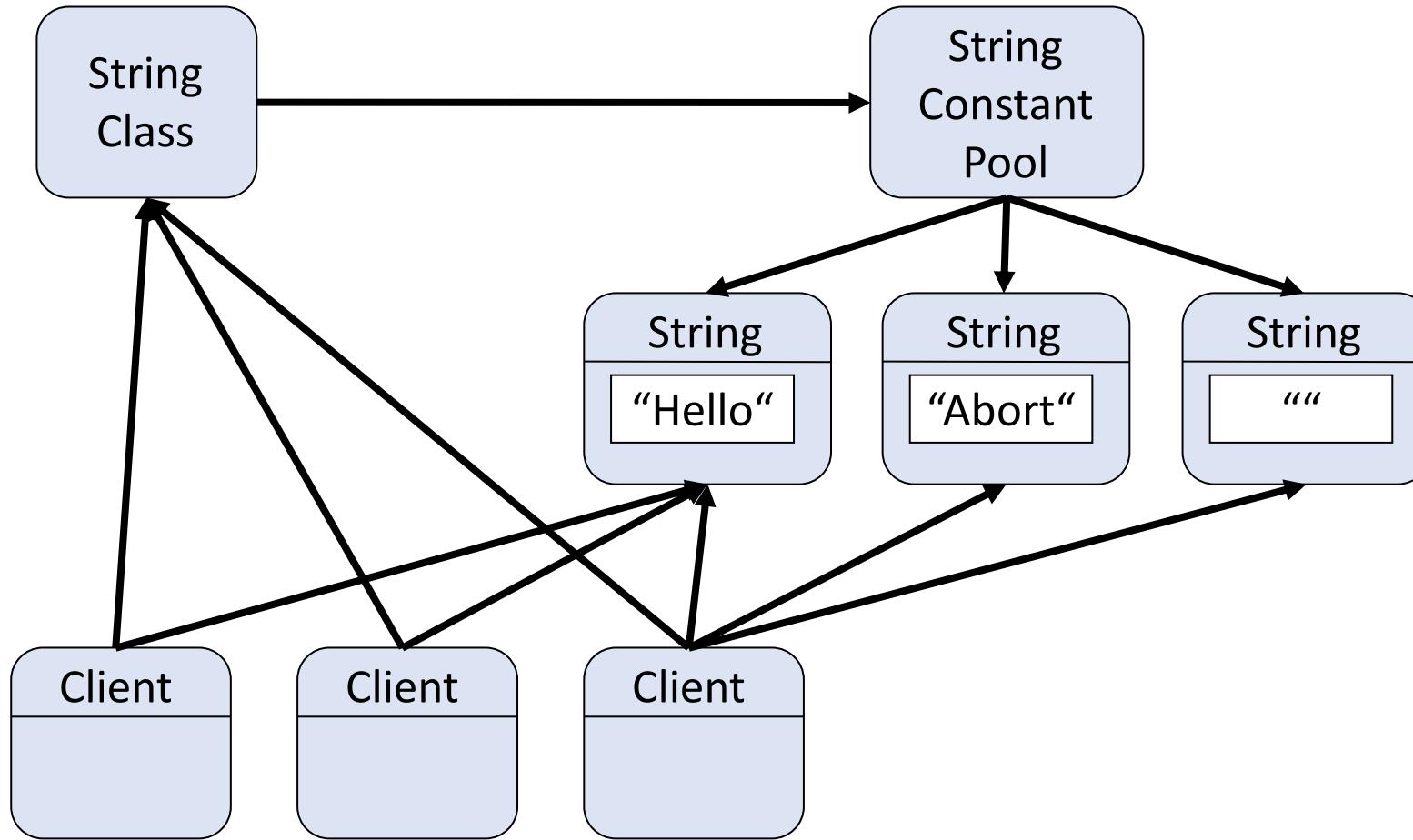
Flyweight Pattern

- The flyweight pattern maximizes sharing of immutable objects



Flyweight Pattern Example

- Java uses the flyweight pattern for constant strings



Modularity

1. Coupling

- Data Coupling
- Procedural Coupling
- Class Coupling

2. Adaptation

Problems of Procedural Coupling

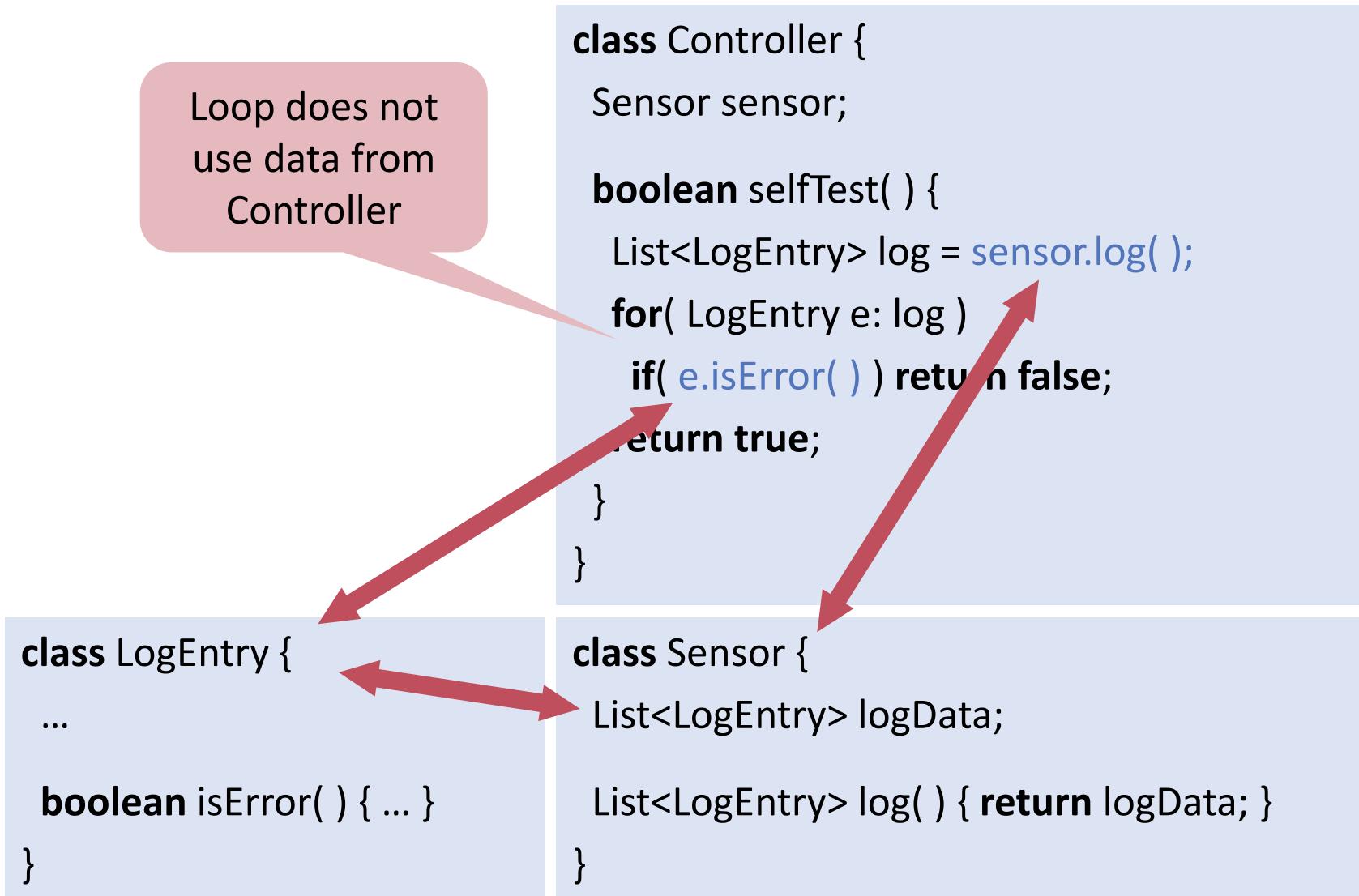
- Modules are coupled to other modules whose methods they call
- Reuse: Callers cannot be reused without callee modules
- Adaptation: Changing signature in callee requires changing caller

```
class LogEntry {  
    boolean isError( ) { ... }  
}
```

```
class Controller {  
    Sensor sensor;  
  
    public boolean selfTest( ) {  
        List<LogEntry> log = sensor.log( );  
        for( LogEntry e: log )  
            if( e.isError( ) ) return false;  
        return true;  
    }  
}
```

```
class Sensor {  
    List<LogEntry> log( ) { return ...; }  
}
```

Approach 1: Moving Code

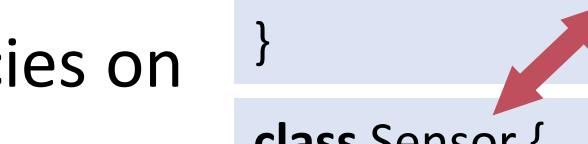


Approach 1: Moving Code (cont'd)

- Moving code may reduce procedural coupling
- It is common to even duplicate functionality to avoid dependencies on other code

```
class LogEntry {  
    ...  
  
    boolean isError( ) { ... }  
}
```

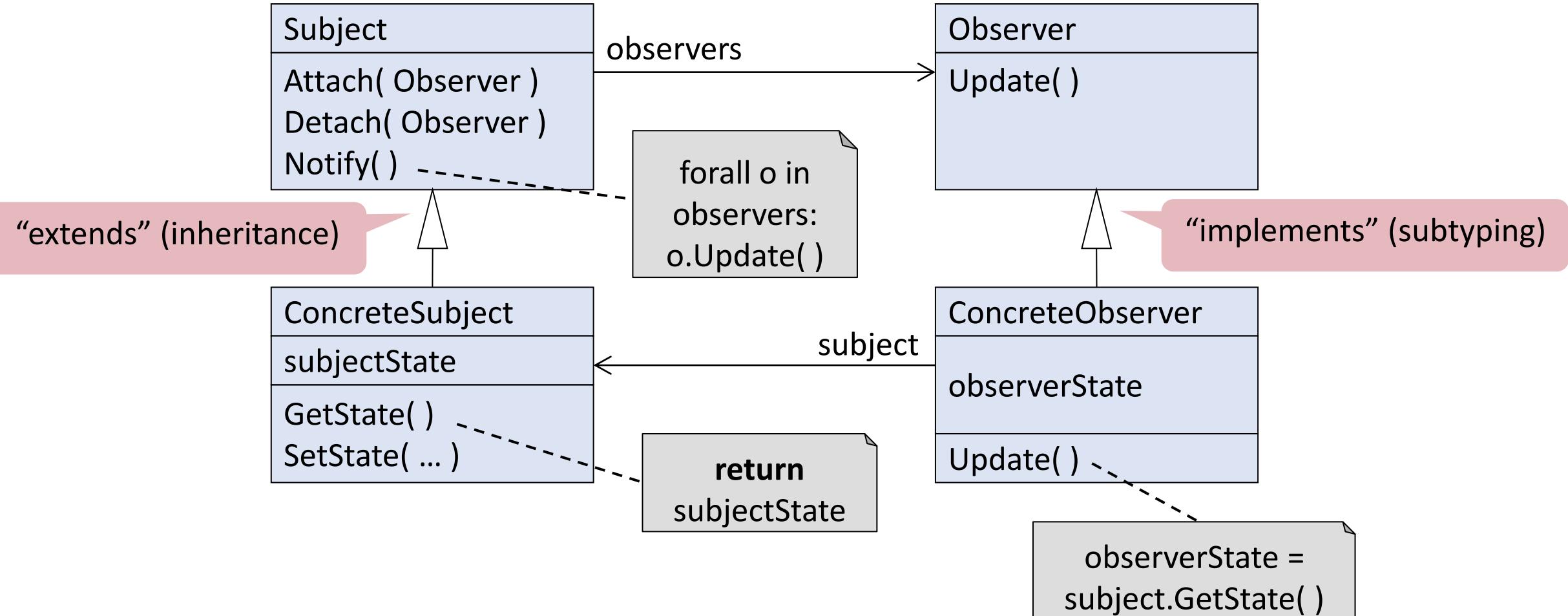
```
class Controller {  
    Sensor sensor;  
  
    boolean selfTest( )  
    { return sensor.noError( ); }  
}  
  
class Sensor {  
    List<LogEntry> logData;  
  
    boolean noError( ) {  
        for( LogEntry e: logData )  
            if( e.isError( ) ) return false;  
  
        return true;  
    }  
}
```



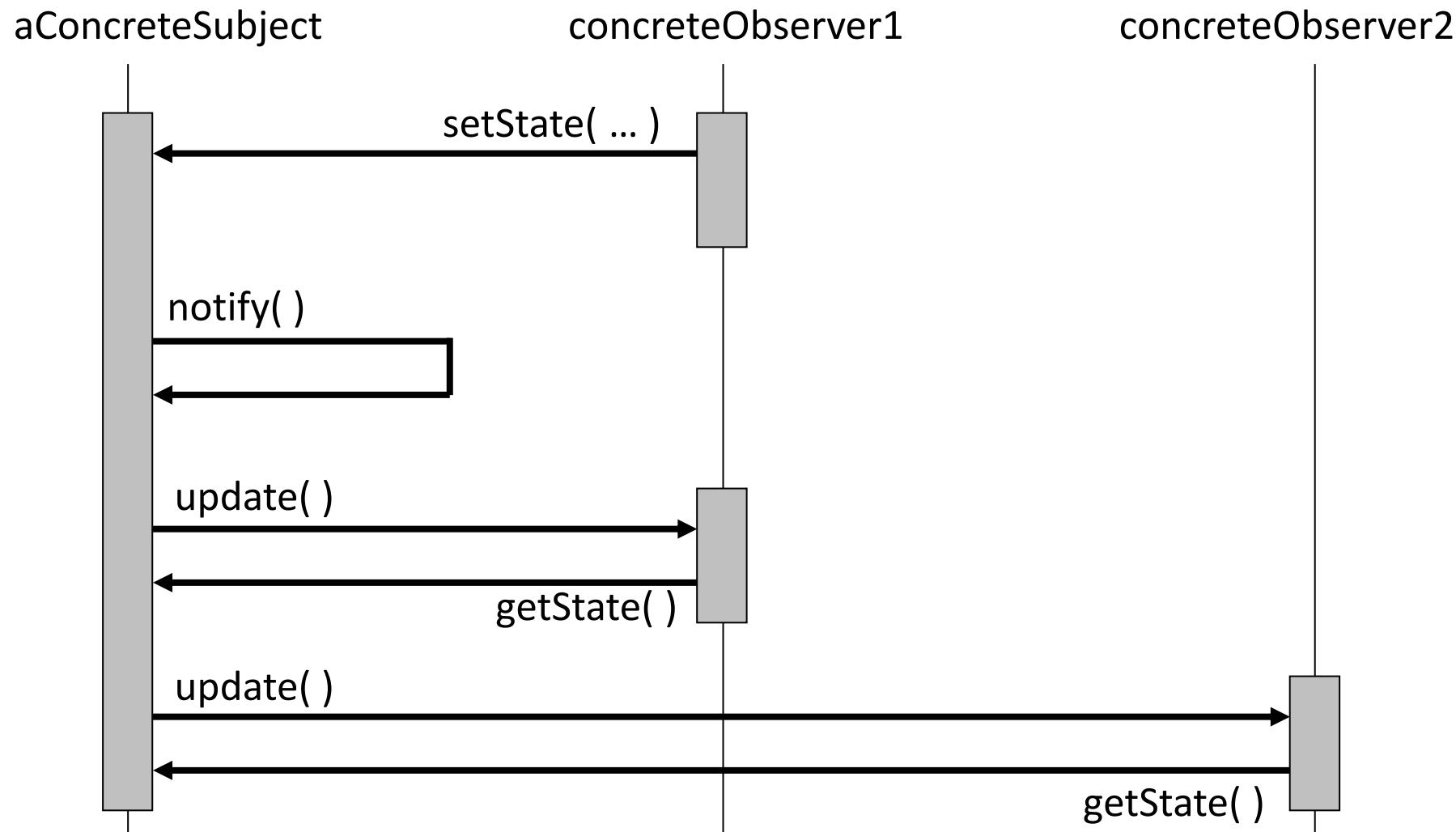
Approach 2: Event-Based Style

- Components may
 - Generate events
 - Register for events of other components using a callback
- Generators of events **do not know** which components will be affected by their events
- Commonly used in user interfaces and web sites
 - Model-View-Controller Architecture, see later

Observer Pattern



Observer Pattern (cont'd)



Observer Pattern: Example

- Debugger has a generic list of observers
- Debugger generates event when breakpoint is reached
- Observers decide how to handle this event

```
class Debugger extends Subject {  
    void processBreakPoint( ... ) {  
        ...  
        notify( ... );  
    }  
}
```

```
class Editor  
    implements Observer {  
    void showContext( ... ) { ... }  
  
    void update ( ... ) {  
        showContext( ... );  
    }  
}
```

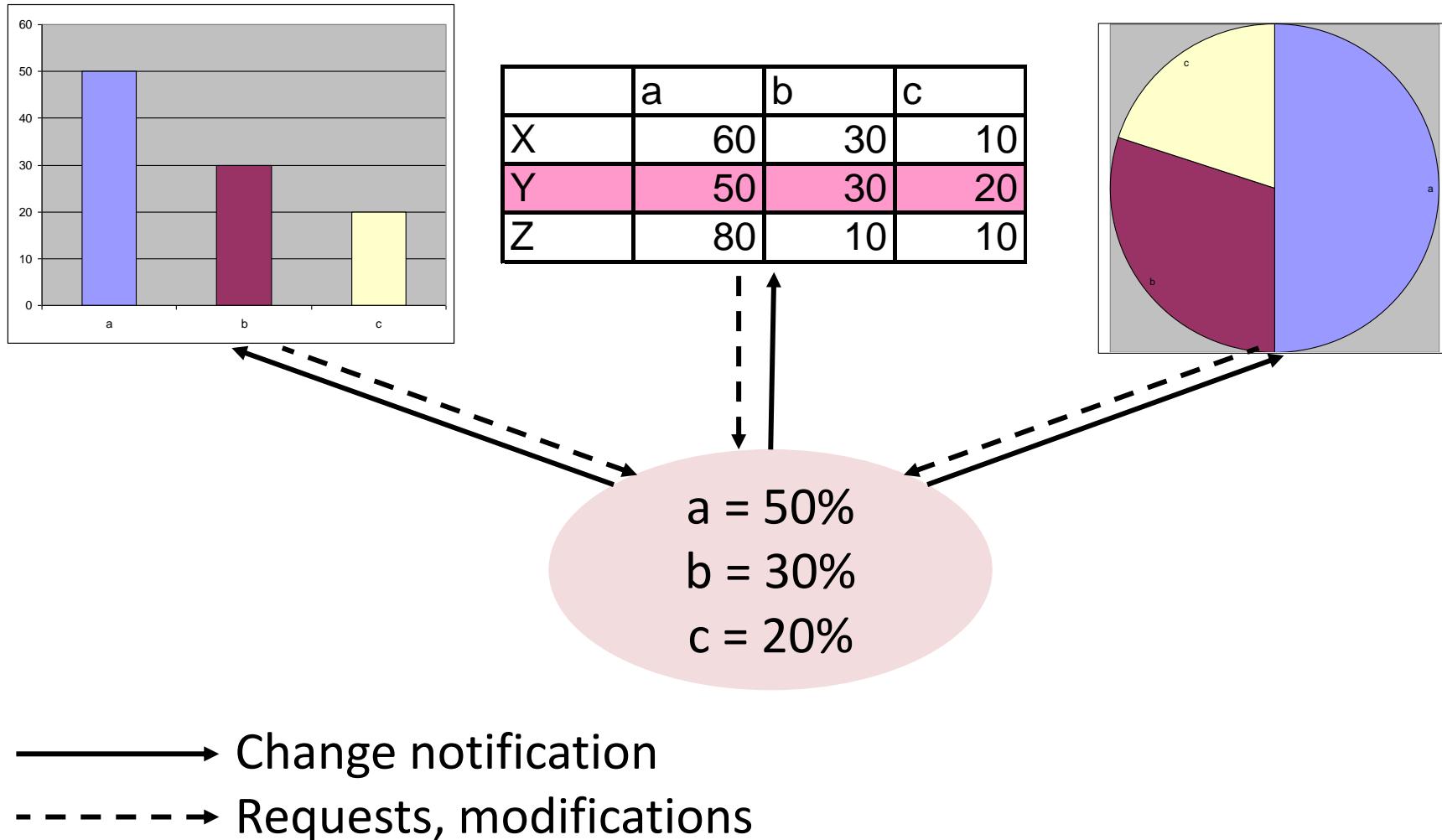
Adaptation: Add StackViewer

- New requirement: Display stack trace when breakpoint is reached
- StackViewer is just another observer
- Debugger does **not** have to be adapted

```
class StackViewer
    implements Observer {
    ...
    void showStackTrace( ... )
    { ... }

    void update ( ... ) {
        showStackTrace( ... );
    }
}
```

Event-Based Style in User Interfaces



Model-View-Controller Architecture

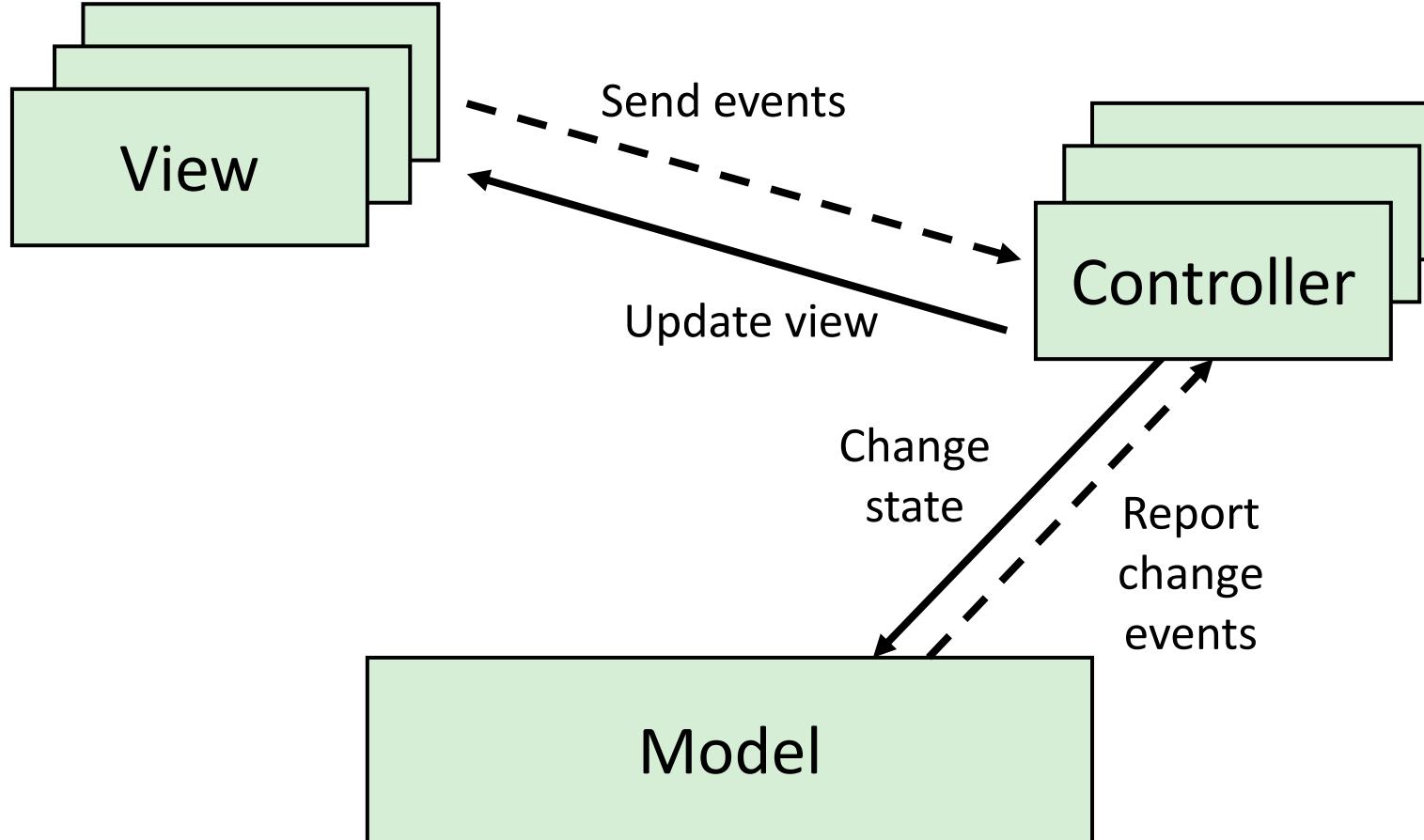
■ Components

- Model contains the core functionality and data
- One or more views display information to the user
- One or more controllers handle user input

■ Communication

- Change-propagation mechanism via events ensures consistency between user interface and model
- If the user changes the model through the controller of one view, the other views will be updated automatically

Model-View-Controller Architecture



Event-Based Style: Discussion

Strengths

- Reuse: plug in new components by registering it for events
- Adaptation: add, remove, and replace components with minimum effect on other components in the system

Weaknesses

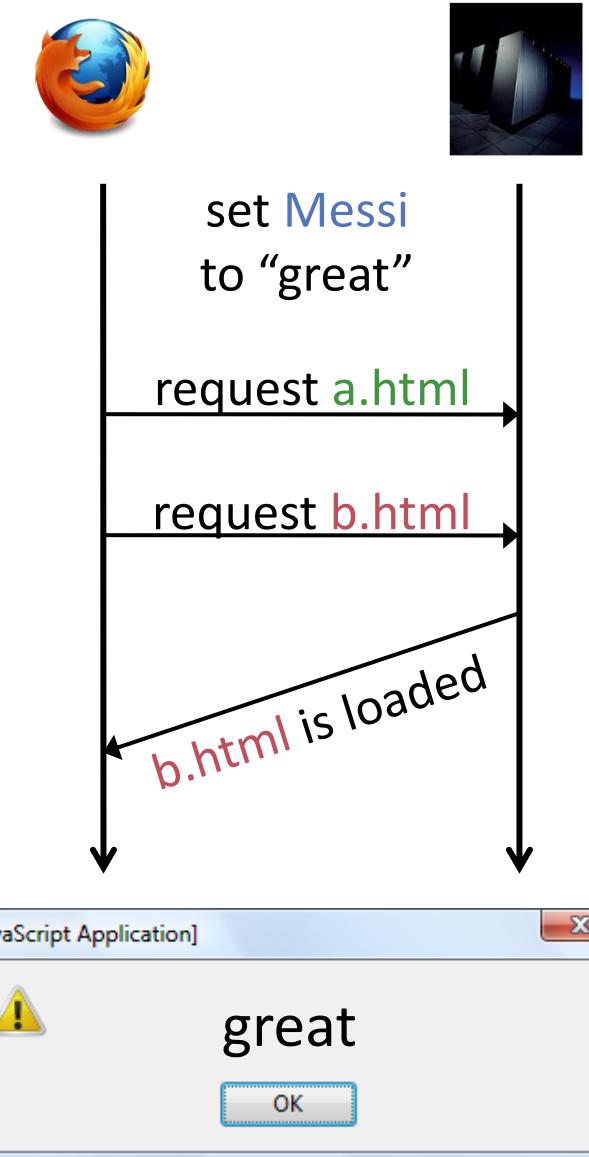
- Loss of control
 - What components will respond to an event?
 - In which order will components be invoked?
 - Are invoked components finished?
- Ensuring correctness is difficult because it depends on context in which invoked

Loss of Control: Example

```
<html><body>
<script>document.Messi = "great";</script>
<iframe src="a.html" ></iframe>
<iframe src="b.html" ></iframe>
</body></html>
```

```
<html><body>
<script>parent.document.Messi = "poor";
</script>
</body></html>
```

```
<html><body>
<script>alert(parent.document.Messi);</script>
</body></html>
```

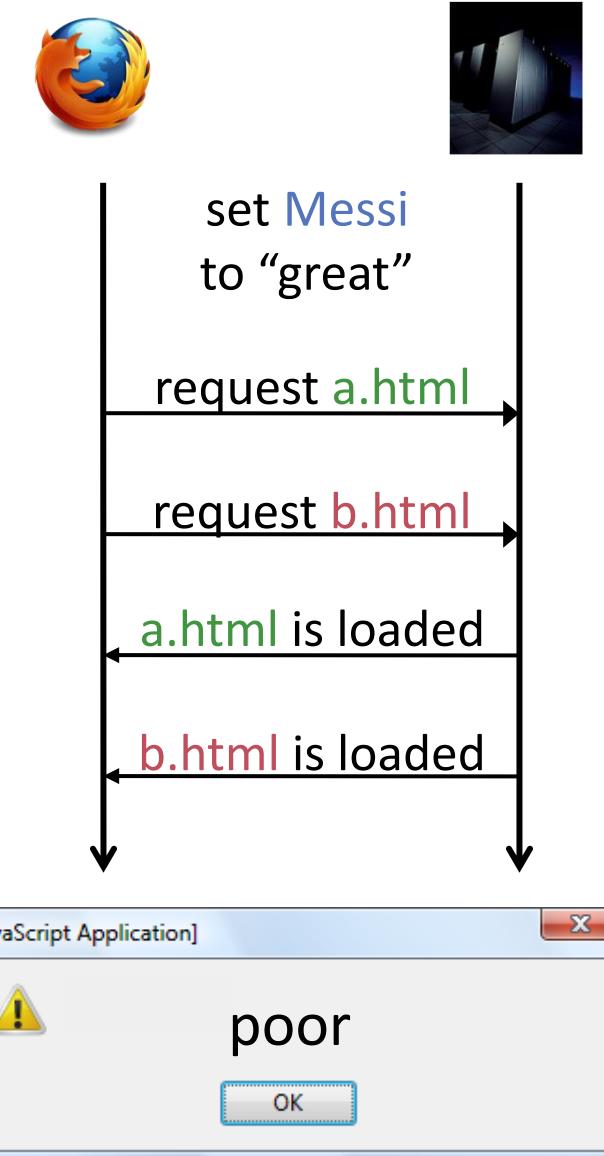


Loss of Control: Example (cont'd)

```
<html><body>
<script>document.Messi = "great";</script>
<iframe src="a.html" ></iframe>
<iframe src="b.html" ></iframe>
</body></html>
```

```
<html><body>
<script>parent.document.Messi = "poor";
</script>
</body></html>
```

```
<html><body>
<script>alert(parent.document.Messi);</script>
</body></html>
```



Modularity

1. Coupling

- Data Coupling
- Procedural Coupling
- Class Coupling

2. Adaptation

Inheritance

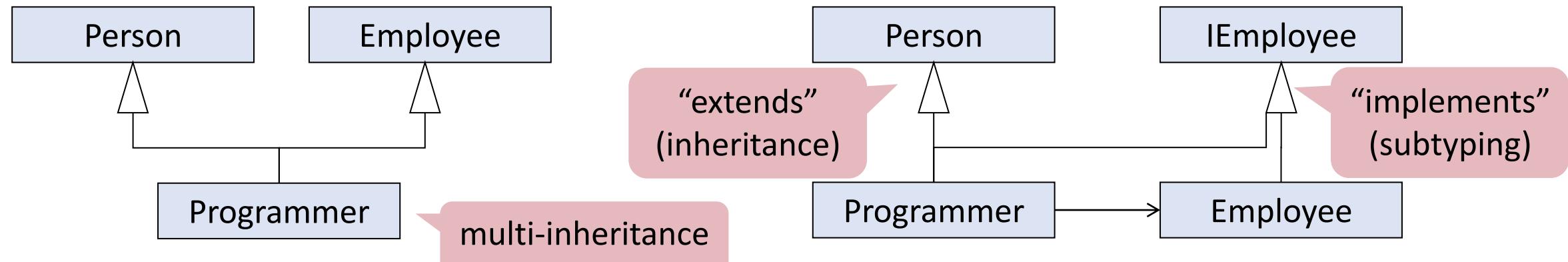
- Inheritance couples the subclass to the superclass
- Changes in the superclass may break the subclass
 - Fragile baseclass problem

```
class SymbolTable  
    extends TreeMap<Ident, Type> {  
}
```

- Limits options for other inheritance relations
 - Only one superclass in single inheritance languages
 - Multiple inheritance may cause conflicts

Approach 1: Inheritance by Subtyping and Delegation

- (Multiple) inheritance can be replaced by **subtyping** and delegation



- Generally, delegation can be used to avoid coupling through inheritance

```
class SymbolTable {  
    TreeMap<Ident, Type> types;  
  
    Type getType( Ident id )  
    { return types.get( id ); }  
}
```

Type Declarations

- Using class names in declarations of methods, fields, and local variables couples the client to the used classes
- Data structures are difficult to change during maintenance

```
class SymbolTable {  
    TreeMap<Ident, Type> types;  
  
    TreeMap<Ident, Type> getTypes( ) {  
        return types.clone( );  
    }  
}
```

Approach 2: Using Interfaces

- Replace occurrences of class names by **supertypes** (interfaces)
- Use the **most general supertype** that offers all required operations
- Data structures can be changed without affecting the code

```
class SymbolTable {  
    TreeMap<Ident, Type> types;  
  
    TreeMap<Ident, Type> getTypes( ) {  
        return types.clone( );  
    }  
}
```



```
class SymbolTable {  
    Map<Ident, Type> types;  
  
    Map<Ident, Type> getTypes( ) {  
        return types.clone( );  
    }  
}
```

Object Allocation

- **Allocations** couple clients to the instantiated class
- Attempt: let client allocate
 - Problem is shifted to clients
 - Difficult to create objects for testing

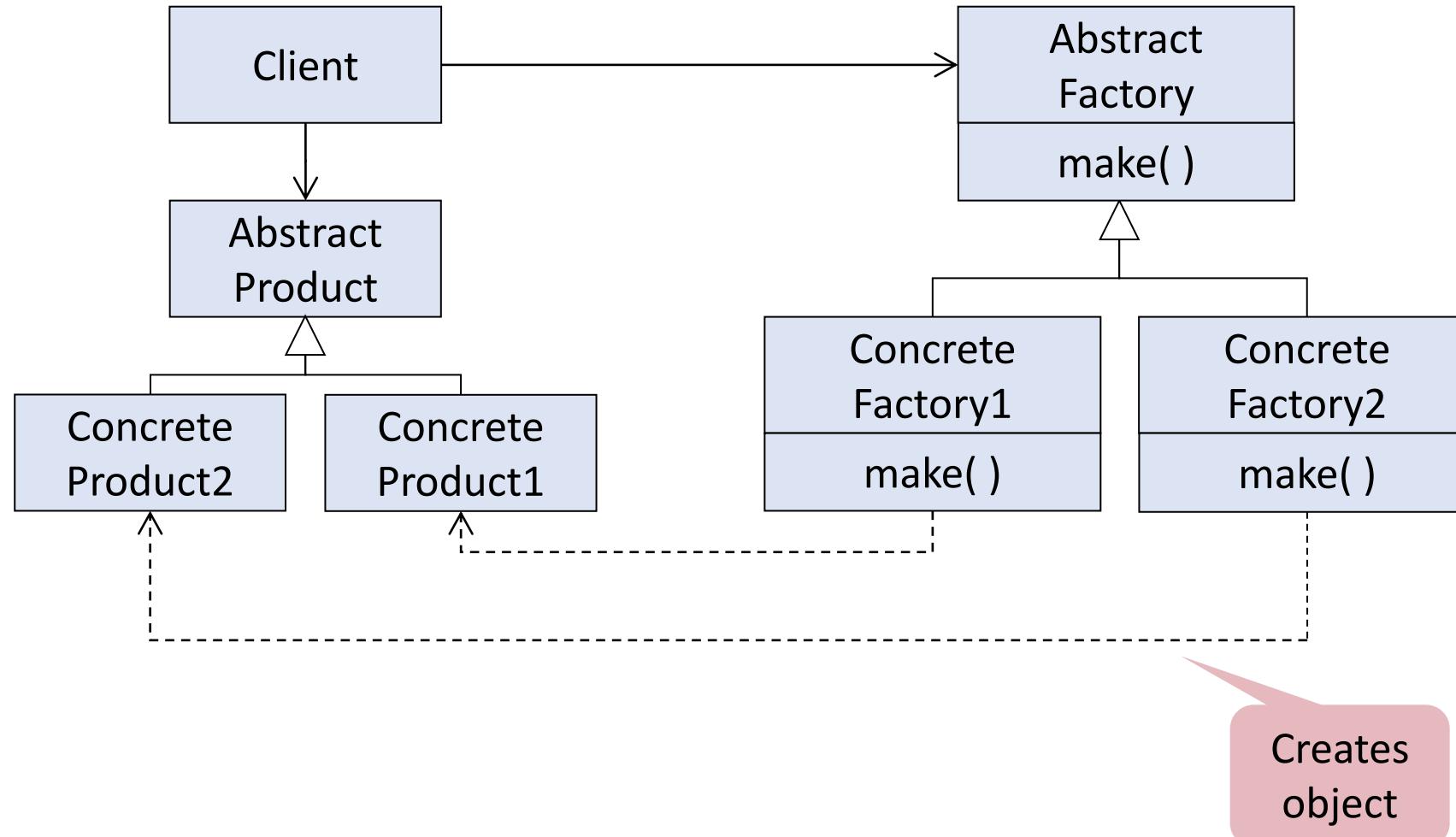
```
class SymbolTable {  
    Map<Ident, Type> types;  
  
    SymbolTable( ) {  
        types = new TreeMap<Ident, Type>();  
    }  
}
```

```
class SymbolTable {  
    Map<Ident, Type> types;  
  
    SymbolTable( Map<Ident, Type> t ) {  
        types = t;  
    }  
}
```

Approach 3: Delegating Allocations via Factories

- Delegate allocations to a dedicated class called an **abstract factory**
- Different **concrete factory** classes make objects of different classes
- The concrete factory to be used is **chosen by the client**

Abstract Factory Pattern



Abstract Factory Example

```
interface MapFactory<K,V> { Map<K,V> make( ); }
```

```
class TreeMapFactory implements MapFactory<K,V> {  
    Map<K,V> make( ) { return new TreeMap<K,V>( ); }  
}
```

```
class SymbolTable {  
    MapFactory<Ident, Type> factory;  
    Map<Ident, Type> types;  
  
    SymbolTable( MapFactory<Ident, Type> f ) {  
        factory = f;  
        types = factory.make( );  
    }  
}
```

SymbolTable no
longer coupled to
TreeMap

Modularity

1. Coupling

- Data Coupling
- Procedural Coupling
- Class Coupling

2. Adaptation

Adaptation

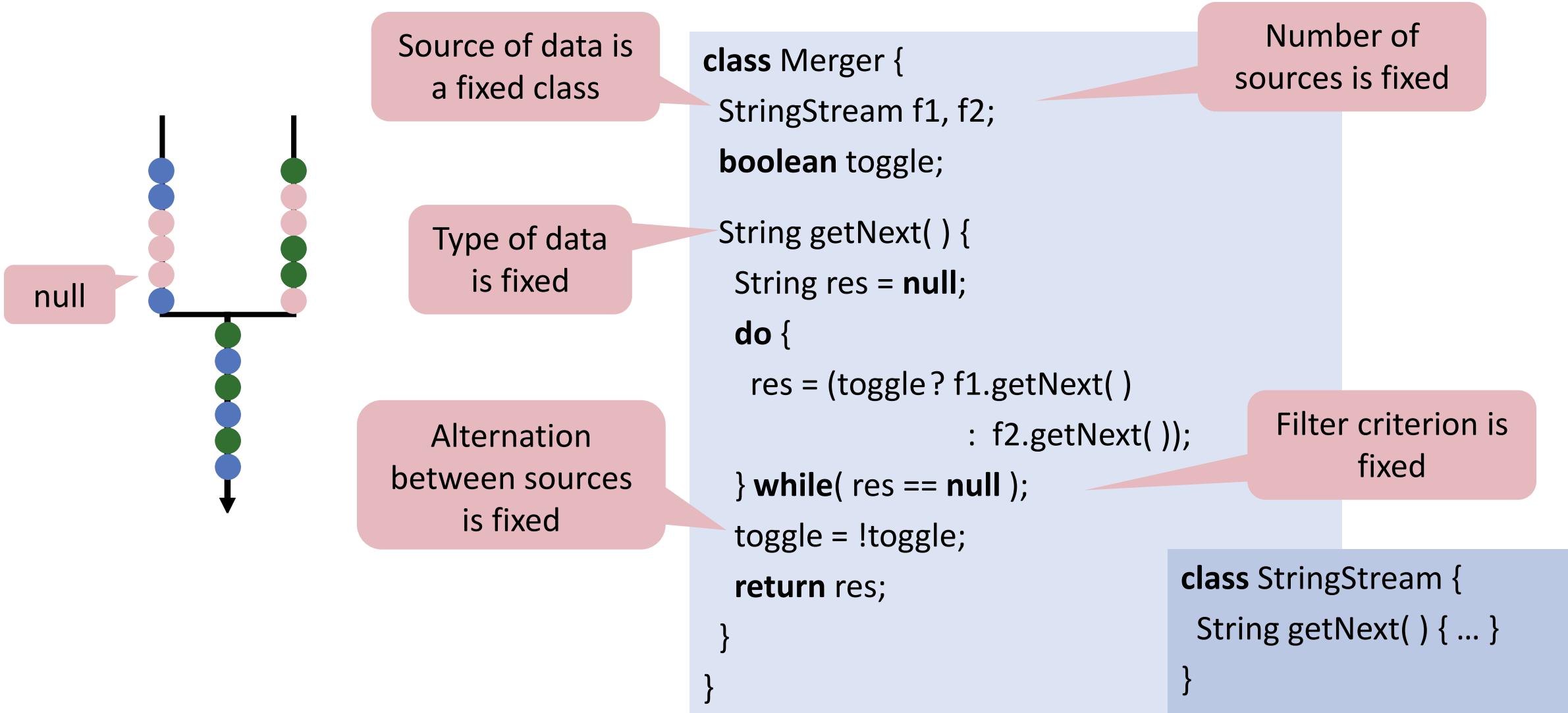
- Since software is (perceived as being) easy to change, software systems often deviate from their initial design
- Typical changes:
 - New features (requested by customers or management)
 - New interfaces (new hardware, new or changed interfaces to other software systems)
 - Bug fixing, performance tuning
- Changes often erode the structure of the system

Parameterization

- Modules can be **prepared for change** by allowing clients to influence their behavior
- Make modules **parametric** in:
 - The values they manipulate
 - The data structures they operate on
 - The types they operate on
 - The algorithms they apply
- *One man's constant is another man's variable.*

[Alan J. Perlis]

Parameterization: Example



Parameterizing Values

- Modules can be made parametric by using **variable values** instead of constant values

```
class Merger {  
    StringStream[ ] streams;  
    int next;  
  
    String getNext( ) {  
        String res = null;  
        do {  
            res = streams[ next ].getNext( );  
        } while( res == null );  
        next = (next + 1) % streams.length;  
        return res;  
    }  
}
```

Parameterizing Data Structures

- Modules can be made parametric by using **interfaces** and **factories** instead of concrete classes

```
class StringStream
    implements Filter {
    String getNext( ) { ... }
}
```

```
class Merger {
    Filter[ ] filters;
    int next;

    String getNext( ) {
        String res = null;
        do {
            res = filters[ next ].getNext( );
        } while( res == null );
        next = (next + 1) % filters.length;
        return res;
    }
}
```

Parameterizing Types

- Modules can be made parametric by using **generic types**

```
class StringStream
    implements Filter<String> {
    String getNext( ) { ... }
}
```

```
class Merger<D> {
    Filter<D>[ ] filters;
    int next;

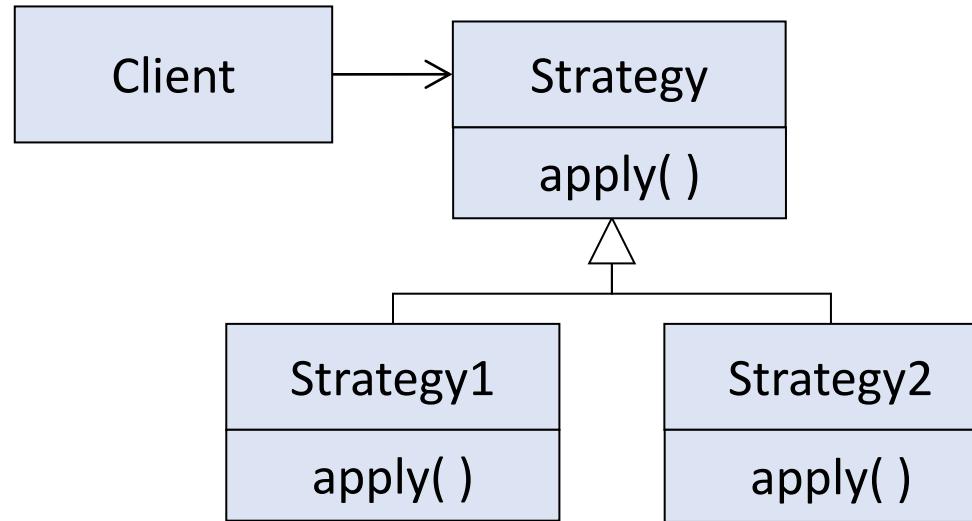
    D getNext( ) {
        D res = null;
        do {
            res = filters[ next ].getNext( );
        } while( res == null );
        next = (next + 1) % filters.length;
        return res;
    }
}
```

Parameterizing Algorithms

- Can parameterize the selector function too!

```
class Merger<D> {  
    Filter<D>[ ] filters;  
    int next;  
    Selector<D> s;  
  
    D getNext( ) {  
        D res = null;  
        do {  
            res = filters[ next ].getNext( );  
        } while( !s.select( res ) );  
        next = (next + 1) % filters.length;  
        return res;  
    }  
}
```

Strategy Pattern

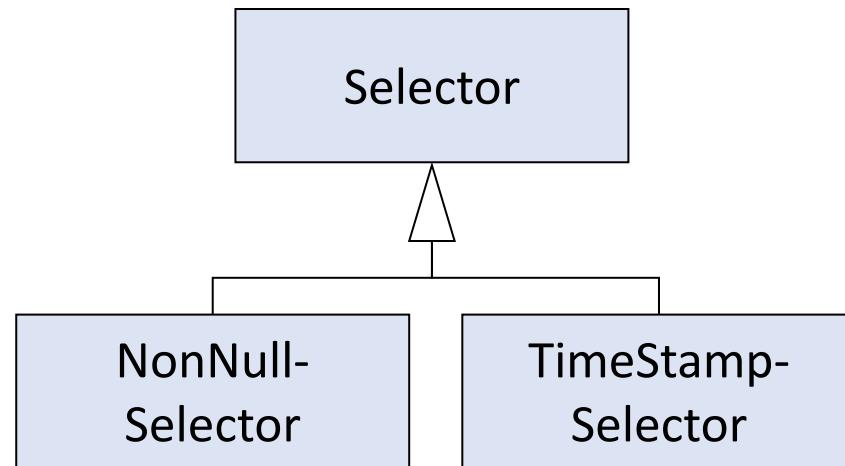


```
interface Selector<D> {  
    boolean select( D val );  
}
```

```
class NonNullSelector<D>  
    implements Selector<D> {  
    boolean select( D val ) {  
        return val != null;  
    }  
}
```

Dynamic Method Binding

- In object-oriented programs, behaviors can be specialized via **overriding** and **dynamic method binding**

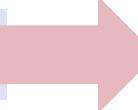


```
class Merger<D> {  
    Filter<D>[ ] filters;  
    int next;  
    Selector<D> s;  
  
    D getNext( ) {  
        D res = null;  
        do {  
            res = filters[ next ].getNext( );  
        } while( !s.select( res ) );  
        next = (next + 1) % filters.length;  
        return res;  
    }  
}
```

Dynamic Method Binding as Case Distinction

- Dynamic method binding is a case distinction on the **dynamic type** of the **receiver** object

```
s.select( res );
```



```
if( s instanceof NonNullSelector )  
    s.NonNullSelector::select( res );  
else if( s instanceof TimeStampSelector )  
    s.TimeStampSelector::select( res );  
else if ...
```

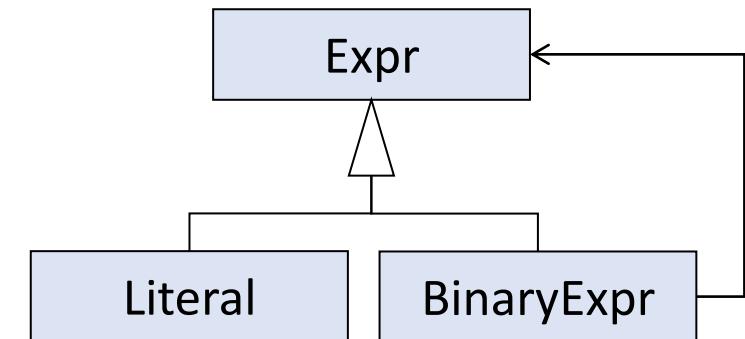
- Adding or removing cases does not require changes in the caller
 - Client code is adaptable

Case Distinction on Arguments

- It can be useful to select an operation based on the **dynamic type** of the receiver object **and the argument(s)**

Example: Syntax Tree

- The behavior of operations depends on the type of node it is applied to
- The set of operations is **not** fixed (type checking, evaluation, code generation, ...)



Double Invocation

Goal: Select method based on dynamic type of *receiver and argument*

based on Expr

```
abstract class Expr {  
    abstract void accept( Visitor v );  
}
```

```
class Literal extends Expr {  
    int val;  
  
    void accept( Visitor v ) {  
        v.visitLiteral( this );  
    }  
}
```

```
class Binary extends Expr {  
    void accept( Visitor v ) {  
        v.visitBinary( this );  
    }  
}
```

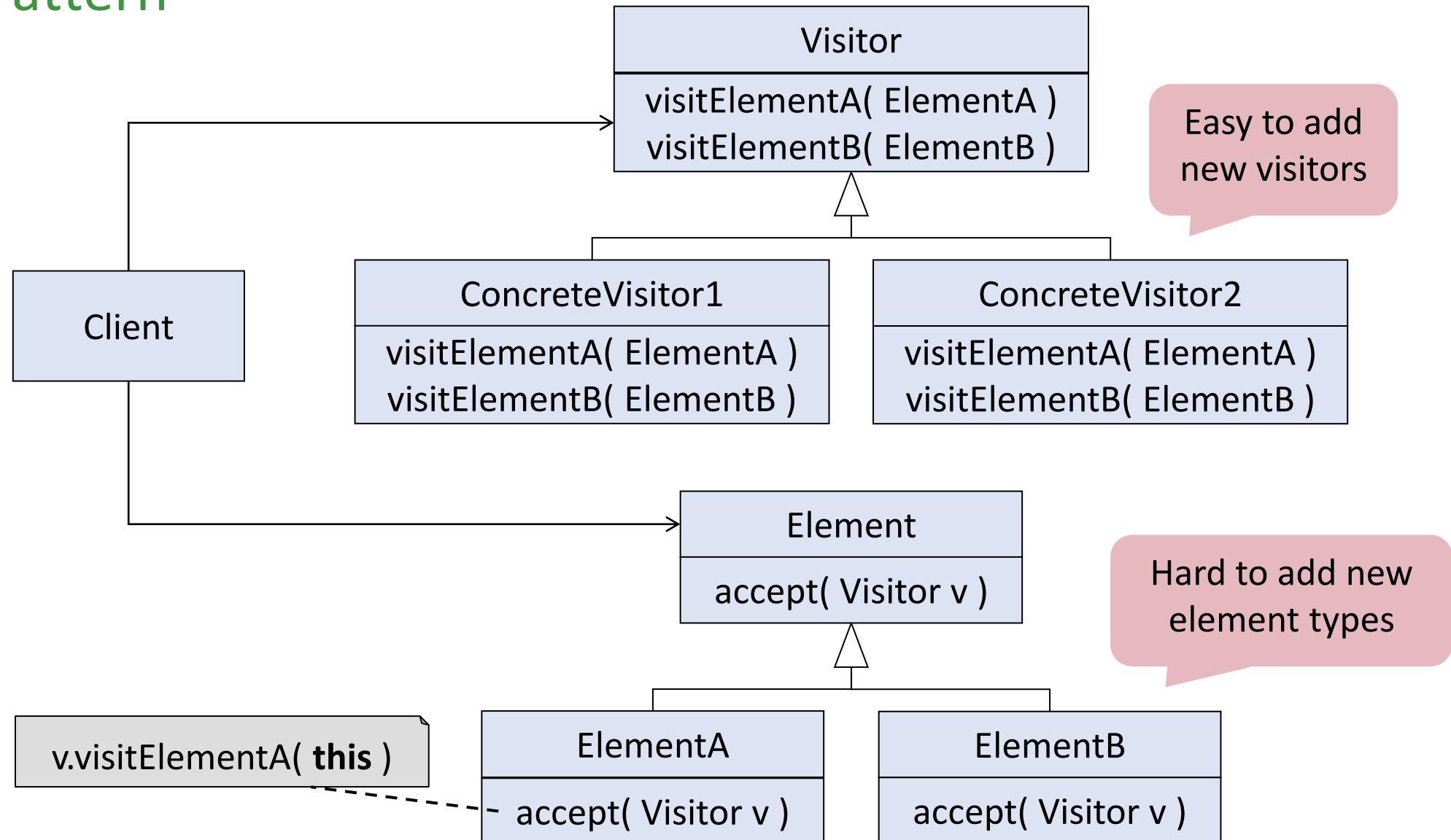
```
abstract class Visitor {  
    abstract void visitLiteral( Literal e );  
    abstract void visitBinary( Binary e );  
}
```

```
class Evaluator extends Visitor {  
    int value;  
    void visitLiteral( Literal e ) {  
        value = e.val;  
    }  
  
    void visitBinary( Binary e ) { ... }  
}
```

```
class PrettyPrinter extends Visitor {  
    ...  
}
```

based on
Visitor

Visitor Pattern



Summary

- Low coupling is a general design goal, but there are trade-offs:
 - Performance and convenience (e.g., cloning overhead for immutable objects)
 - Adaptability (e.g., visitor pattern improves adaptability but induces coupling)
 - Coupling to stable classes (e.g., library classes) is less critical
- Designing adaptable modules facilitates inevitable changes and reuse

Constraint Solving: challenges

Constraint solving is fundamental to symbolic execution as a constraint solver is continuously invoked during analysis. Often, the main **roadblock to performance** of symbolic execution engines is the time spent in constraint solving. Therefore, it is important that:

1. The SMT solver supports as many decidable logical fragments as possible. Some tools use more than one SMT solver.
2. The SMT solver can solve large formulas quickly.
3. The symbolic execution engines tries to reduce the burden in calling the SMT solver by exploring domain specific insights.

Key Optimization: Caching

The basic insight here is that often, the analysis will invoke the SMT solver with **similar formulas**. Therefore, the symbolic execution system can keep a **map (cache)** of formulas to a satisfying assignment for the formula.

Then, when the engine builds a new formula and would like to find a satisfying assignment for that formula, it can first access the cache, **before calling** the SMT solver.

Key Optimization: Caching

Suppose the cache contains the mapping:

Formula:		Solution:
$(x + y < 10) \wedge (x > 5)$	→	$\{x = 6, y = 3\}$

If we get a weaker formula as a query, say $(x + y < 10)$, then we can immediately reuse the solution already found in the cache, without calling the SMT solver.

If we get a stronger formula as a query, say $(x + y < 10) \wedge (x > 5) \wedge (y \geq 0)$, then we can quickly try the solution in the cache and see if it works, without calling the solver (in this example, it works).

When Constraint Solving Fails

Despite best efforts, the program may be using constraints in a logical fragment which the SMT solver does not handle well.

For instance, suppose the SMT solver does not handle **non-linear constraints** well.

Let us consider a modification of our running example.

Modified Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```



Here, we changed the `twice()` function to contain a **non-linear** result.

Let us see what happens when we symbolically execute the program now...

Modified Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

This is the result if $x = z$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto y_0 * y_0\end{aligned}$$

$$pct : x_0 = y_0 * y_0$$

Now, if we are to invoke the SMT solver with the `pct` formula, it may be **unable** to compute satisfying assignments, precluding us from knowing whether the path is feasible or not.

Solution: Concolic Execution

Concolic Execution: combines **both** symbolic execution and concrete (normal) execution.

The basic idea is to have the concrete execution **drive** the symbolic execution.

Here, the program runs as usual (it needs to be given some input), but in addition it also maintains the usual symbolic information.

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

The `read()` functions read a value from the input. Suppose we read $x = 22$ and $y = 7$.

We will keep both the concrete store and the symbolic store and path constraint.

$$\sigma : x \mapsto 22, \\ y \mapsto 7$$

$$\sigma_s : x \mapsto x_0, \\ y \mapsto y_0$$

pct : true

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

$$\begin{aligned}\sigma : x &\mapsto 22, \\ y &\mapsto 7, \\ z &\mapsto 14\end{aligned}$$

$$\begin{aligned}\sigma_s : x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2^*y_0\end{aligned}$$

pct : true

The concrete execution will now take the ‘else’ branch of $z == x$.

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}
```

```
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}
```

```
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

Hence, we get:

$$\sigma : x \mapsto 22,
y \mapsto 7,
z \mapsto 14$$

$$\sigma_s : x \mapsto x_0,
y \mapsto y_0
z \mapsto 2 * y_0$$

$$pct : x_0 \neq 2 * y_0$$

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

At this point, concolic execution decides that it would like to explore the “true” branch of $x == z$ and hence it needs to generate concrete inputs in order to explore it. Towards such inputs, it negates the `pct` constraint, obtaining:

```
pct : x0 = 2*y0
```

It then calls the SMT solver to find a satisfying assignment of that constraint. Let us suppose the SMT solver returns:

```
x0 ↦ 2, y0 ↦ 1
```

The concolic execution then runs the program with this input.

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

With the input $x \mapsto 2, y \mapsto 1$ we reach this program point with the following information:

$$\sigma : x \mapsto 2,
y \mapsto 1,
z \mapsto 2$$

$$\sigma_s : x \mapsto x_0,
y \mapsto y_0
z \mapsto 2 * y_0$$

$$pct : x_0 = 2 * y_0$$

Continuing further we get:

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

We reach the “else” branch of $x > y + 10$

$$\begin{aligned}\sigma : x &\mapsto 2, \\ y &\mapsto 1, \\ z &\mapsto 2\end{aligned}$$

$$\begin{aligned}\sigma_s : x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$\begin{aligned}pct : x_0 &= 2*y_0 \\ &\wedge \\ x_0 &\leq y_0 + 10\end{aligned}$$

Again, concolic execution may want to explore the ‘true’ branch of $x > y + 10$.

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

We reach the “else” branch of $x > y + 10$

$$\begin{aligned}\sigma : x &\mapsto 2, \\ y &\mapsto 1, \\ z &\mapsto 2\end{aligned}$$

$$\begin{aligned}\sigma_s : x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$\begin{aligned}pct : x_0 &= 2*y_0 \\ &\wedge \\ x_0 &\leq y_0+10\end{aligned}$$

Concolic execution now negates the conjunct
 $x_0 \leq y_0+10$ obtaining:

$$x_0 = 2*y_0 \wedge x_0 > y_0+10$$

A satisfying assignment is: $x_0 \mapsto 30, y_0 \mapsto 15$

Concolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

If we run the program with the input:

$x_0 \mapsto 30, y_0 \mapsto 15$

we will now reach the **ERROR** state.

As we can see from this example, by keeping the symbolic information, the concrete execution can use that information in order to obtain new inputs.

Non-linear constraints

Let us return to the problem of **non-linear constraints**

We remark that non-linear constraints are just a representative of a feature which is difficult for SMT solvers to deal with and hence they need to under-approximate (via concolic execution)

Non-linear constraints

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```



Let us again consider our example and see what concolic execution would do with non-linear constraints.

Concolic Execution: Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

The `read()` functions read a value from the input. Suppose we read $x = 22$ and $y = 7$.

$$\sigma : x \mapsto 22, \\ y \mapsto 7$$

$$\sigma_s : x \mapsto x_0, \\ y \mapsto y_0$$

pct : true

Concolic Execution: Example

```
int twice(int v) {  
    return v * v;  
}
```

```
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}
```

```
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

$$\sigma : x \mapsto 22,
y \mapsto 7,
z \mapsto 49$$

$$\sigma_s : x \mapsto x_0,
y \mapsto y_0
z \mapsto y_0 * y_0$$

pct : true

The concrete execution will now take the ‘else’ branch of $x == z$.

Concolic Execution: Example

```
int twice(int v) {  
    return v * v;  
}
```

```
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}
```

```
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Hence, we get:

$$\sigma : x \mapsto 22,
y \mapsto 7,
z \mapsto 49$$

$$\sigma_s : x \mapsto x_0,
y \mapsto y_0
z \mapsto y_0 * y_0$$

$$pct : x_0 \neq y_0 * y_0$$

Concolic Execution: Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

However, here we have a non-linear constraint $x_0 \neq y_0 * y_0$. If we would like to explore the true branch we negate the constraint, obtaining $x_0 = y_0 * y_0$ but again we have a **non-linear constraint**!

In this case, concolic execution simplifies the constraint by plugging in the concrete values for y_0 in this case, 7, obtaining the simplified constraint:

$$x_0 = 49$$

Hence, it now runs the program with the input

$$x \mapsto 49, \quad y \mapsto 7$$

Concolic Execution: Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Running with the input

$x \mapsto 49$, $y \mapsto 7$

will reach the error state.

However, notice that with these inputs, if we try to simplify non-linear constraints by plugging in concrete values (as concolic execution does), then concolic execution we will never reach the else branch of the `if (x > y + 10)` statement.

Concolic Execution: External World

```
int F(char *f) {  
    FILE *fp;  
    fp = fopen(f, "r");  
    ...  
}
```

Often, we are not interested in symbolically executing OS code or native libraries. By keeping both the concrete and the symbolic values, we can always invoke **system calls** or **library API** with the concrete values and simply continue concrete execution as usual.

Symbolic Execution: Selected Tools

- Stanford's KLEE: <http://klee.llvm.org/>
- NASA's Java PathFinder: <http://javapathfinder.sourceforge.net/>
- Microsoft Research's SAGE
- UC Berkeley's CUTE
- EPFL's S2E: <http://dslab.epfl.ch/proj/s2e>

Summary

- *Symbolic execution* is a versatile technique for analysing programs
 - Fully automated, relies on SMT solvers
 - Scales to large programs
 - Different flavours, with different goals, have been proposed over the decades
- *Concolic execution* combines concrete with symbolic execution
 - To overcome SMT solver limitations (theory undecidability)
 - Is ultimately a search-based *testing technique* with the goal of improving branch coverage in testing
- Advantages of concolic execution:
 - Handling SMT solver limitations
 - Yields actual test inputs → improves test suite
 - Can handle native functions/code
 - Can handle hard problems (e.g. bit manipulations, crypto functions)

Summary

- Concolic execution is potentially **unsound**:
 - Bounding loops (and recursion), e.g. by bounded unrolling
 - Plugging in concrete values
- Concolic execution, despite being a testing technique, is **sound and complete** (neither false negatives nor positives) if all loops are input-independent and concrete values never plugged in (see e.g. <https://mariachris.github.io/Pubs/IT-2017.pdf>)

Terminology

- 1976: King, Clarke – *Symbolic Execution*, as an improvement over hand-written tests with manually specified test inputs (slides 8ff)
- 2005: Combine symbolic and concrete executions to overcome limitations of purely symbolic execution
 - Goidefroid, Sen et. al (DART, CUTE) – *Concolic Execution* (slides 36ff), also called *concolic testing*
 - Cedar et al. (ExE, KLEE) – *Execution-Generated Testing*, similar approach
- 2013: Cadar, Sen – *Dynamic Symbolic Execution* as an umbrella term for both *Concolic Testing* and *Execution-Generated Testing*

Further notes:

- Term *dynamic symbolic execution* appears to originate from a patent application from 1982, from Johns Hopkins University
- O'Hearn et. al develop SMALLFOOT in 2005, symbolic execution for separation logic; not a testing technique, irrelevant for this course; used by Viper (remember FMFP)

Rigorous Software Engineering

Applications of Analysis: Intervals

Prof. Martin Vechev

Tarski's theorem tells us that a fixed point exists, but does not actually suggest an algorithm for computing it.

Next: we look at ways to compute a fixed point

Function Iterates

For a poset (L, \sqsubseteq) , a function $f: L \rightarrow L$, an element $a \in L$, the iterates of the function from a are:

$$f^0(a), f^1(a), f^2(a) \dots$$

$$\text{where } f^{n+1}(a) = f(f^n(a))$$

Note that $f^0(a) = a$

In program analysis, we usually take a to be \perp

A useful fixed point theorem

Given a poset of finite height, a least element \perp , a **monotone** f .

Then the iterates $f^0(\perp), f^1(\perp), f^2(\perp) \dots$ form an increasing sequence which eventually stabilizes from some $n \in \mathbb{N}$, that is:
 $f^n(\perp) = f^{n+1}(\perp)$ and:

$$\text{lfp}^{\sqsubseteq} f = f^n(\perp)$$

This leads to a simple algorithm for computing $\text{lfp}^{\sqsubseteq} f$

Cheat Sheet: Connecting Math and Analysis

Mathematical Concept	Use in Static Analysis
Complete Lattice	Defines Abstract Domain and ensure joins exist.
Joins (\sqcup)	Combines facts arriving at a program point
Bottom (\perp)	Used for initialization of all but initial elements
Top (T)	Used for initialization of initial elements
Widening (\triangledown)	Used to guarantee analysis termination
Function Approximation	Critical to make sure abstract semantics approximate the concrete semantics
Fixed Points	This is what is computed by the analysis
Tarski's Theorem	Ensures fixed points exist.

So Far

So far, we saw a bunch of mathematical concepts such as lattices, functions, fixed points, function approximation, etc.

Next, we will see how to put these together in order to build static analyzers.

Starting point

Our starting point is a domain where each element of the domain is a **set of states**. The domain of states is a **complete lattice**:

$$(\wp(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma)$$

$$\Sigma = \text{Lab} \times \text{Store}$$

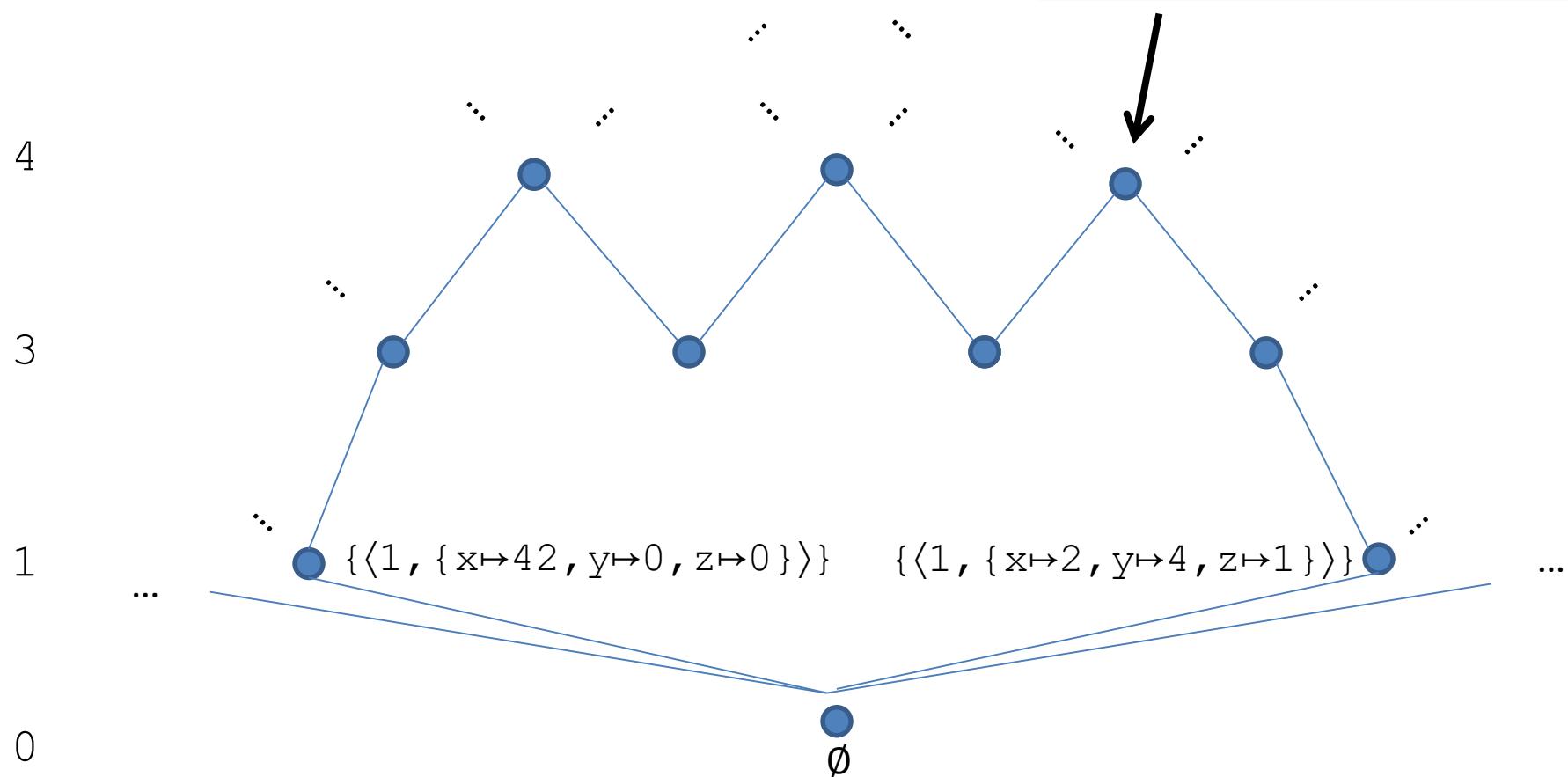
Starting Point: Domain of States

Size of Set:

$$n > 0$$

$$\Sigma$$

Each element is a finite set of states, e.g., $\llbracket P \rrbracket$



Recall: Representing $\llbracket P \rrbracket$

Let $\llbracket P \rrbracket$ be the set of reachable state of a program P .

Let function F be (where I is an initial set of states):

$$F(S) = I \cup \{ c' \mid c \in S \wedge c \rightarrow c' \}$$

Then, $\llbracket P \rrbracket$ is a **fixed point** of F : $F(\llbracket P \rrbracket) = \llbracket P \rrbracket$

(in fact, $\llbracket P \rrbracket$ is the **least fixed point** of F)

Abstract Interpretation: step-by-step

1. select/define an abstract domain
 - selected based on the type of **properties** you want to prove
2. define abstract semantics **for the language** w.r.t. to the domain
 - prove **sound** w.r.t **concrete semantics**
 - involves defining abstract transformers
 - that is, effect of statement / expression on the abstract domain
3. iterate abstract transformers over the abstract domain
 - until we reach a **fixed point**

The **fixed point** is the **over-approximation** of the program

Abstract Interpretation: Step 1

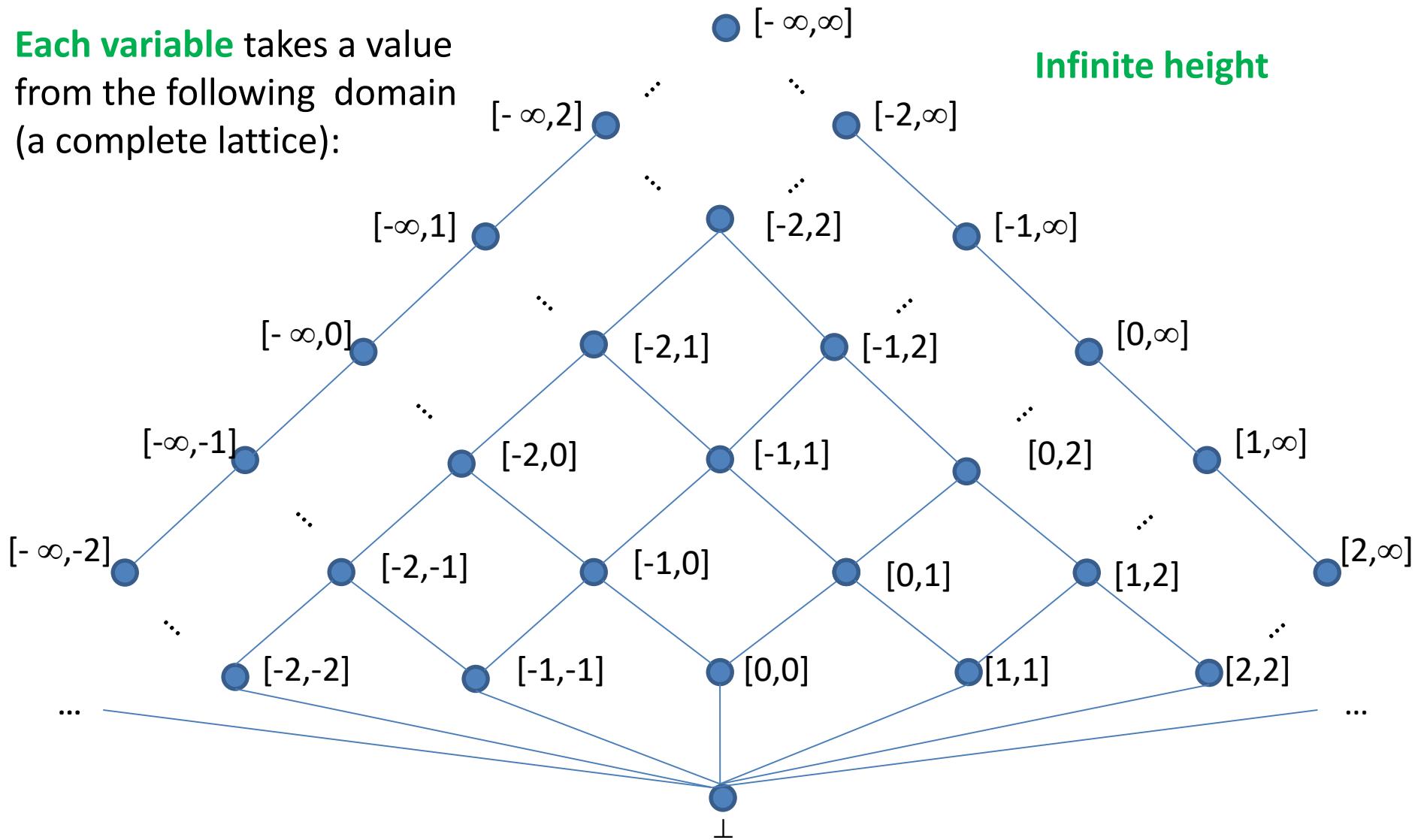
1. select/define an abstract domain
 - selected based on the type of **properties** you want to prove

Interval Domain

If we are interested in properties that involve the range of values that a variable can take, we can abstract the set of states into a map which captures the range of values that a variable can take.

Interval Domain

Each variable takes a value from the following domain (a complete lattice):



Infinite height

Interval Domain: Lets Define it

Let the interval domain be: $(L^i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \perp_i, [-\infty, \infty])$

We denote $Z^\infty = Z \cup \{-\infty, \infty\}$

The set $L^i = \{ [x, y] \mid x, y \in Z^\infty, x \leq y \} \cup \{\perp_i\}$

For a set $S \subseteq Z^\infty$, $\min(S)$ returns the minimum number in S , $\max(S)$ returns the maximum number in S .

- $[a, b] \sqsubseteq_i [c, d]$ if $c \leq a$ and $b \leq d$
- $[a, b] \sqcup_i [c, d] = [\min(\{a, c\}), \max(\{b, d\})]$
- $[a, b] \sqcap_i [c, d] = \text{meet}(\max(\{a, c\}), \min(\{b, d\}))$
where $\text{meet}(a, b)$ returns $[a, b]$ if $a \leq b$ and \perp_i otherwise

Ex: $[2, 2] \sqcup_i [3, 3] = ?$

Ex: $[2, 2] \sqcap_i [3, 3] = ?$

Intervals: Applied to Programs

The L^i domain simply defines intervals, but to apply it to programs we need to take into account program labels (program counters) and program variables.

Therefore, for programs, we use the domain $\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)$

That is, at each label and for each variable, we will keep the range for that variable. This domain is also a **complete lattice**.

The operators of L^i $\sqsubseteq_i, \sqcup_i, \sqcap_i$ are **lifted directly** to both domains:

- $\text{Var} \rightarrow L^i$
- $\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)$

Example of the lifting

Suppose we have two variables x and y in our program. Then:

Let map1 be: $x \rightarrow [1, 5]$, $y \rightarrow [7, 10]$

Let map2 be: $x \rightarrow [2, 6]$, $y \rightarrow [6, 9]$

Then $\text{map3} = \text{map1} \sqcap \text{map2} = x \rightarrow [2, 5]$, $y \rightarrow [7, 9]$

Then $\text{map4} = \text{map1} \sqcup \text{map2} = x \rightarrow [1, 6]$, $y \rightarrow [6, 10]$

Here, $\text{map3} \sqsubseteq \text{map4}$

Here \perp is the map: $x \rightarrow \perp_i$, $y \rightarrow \perp_i$

Intervals: Applied to Programs

$$\begin{aligned}\alpha^i : \wp(\Sigma) &\rightarrow (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)) \\ \gamma^i : (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)) &\rightarrow \wp(\Sigma)\end{aligned}$$

Using α^i , we abstract a **set of states** into a map from program labels to interval ranges for each variable.

Using γ^i , we concretize the intervals maps to a set of **states**

Formal definition of α^i and γ^i is left as an exercise.

Let us consider an example to give an intuition.

Example of Abstraction and Concretization

```
αi ( { ⟨1, {x↦1, y↦9, q↦-2}⟩, ⟨1, {x↦5, y↦9, q↦-2}⟩, ⟨1, {x↦8, y↦9, q↦-2}⟩,  
⟨1, {x↦1, y↦9, q↦4}⟩,   ⟨1, {x↦5, y↦9, q↦4}⟩,   ⟨1, {x↦8, y↦9, q↦4}⟩ } )  
= 1 → (x ↠ ??? , y ↠ ??? , q ↠ ??? )
```

Example of Abstraction and Concretization

```
 $\alpha^i ($ 
{ ⟨1, {x ↦ 1, y ↦ 9, q ↦ -2}⟩, ⟨1, {x ↦ 5, y ↦ 9, q ↦ -2}⟩, ⟨1, {x ↦ 8, y ↦ 9, q ↦ -2}⟩,
⟨1, {x ↦ 1, y ↦ 9, q ↦ 4}⟩, ⟨1, {x ↦ 5, y ↦ 9, q ↦ 4}⟩, ⟨1, {x ↦ 8, y ↦ 9, q ↦ 4}⟩}
 $)$ 
 $= 1 \rightarrow (x \mapsto [1, 8], y \mapsto [9, 9], q \mapsto [-2, 4])$ 
```

```
 $\gamma^i (1 \rightarrow (x \mapsto [1, 8], y \mapsto [9, 9], q \mapsto [-2, 4]))$ 
 $= \{ \langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto -2 \} \rangle, \langle 1, \{x \mapsto 5, y \mapsto 9, q \mapsto -2 \} \rangle, \langle 1, \{x \mapsto 8, y \mapsto 9, q \mapsto -2 \} \rangle,$ 
 $\langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto 4 \} \rangle, \langle 1, \{x \mapsto 5, y \mapsto 9, q \mapsto 4 \} \rangle, \langle 1, \{x \mapsto 8, y \mapsto 9, q \mapsto 4 \} \rangle,$ 
 $\langle 1, \{x \mapsto 7, y \mapsto 9, q \mapsto 3 \} \rangle, \langle 1, \{x \mapsto 3, y \mapsto 9, q \mapsto 4 \} \rangle, \langle 1, \{x \mapsto 1, y \mapsto 9, q \mapsto -1 \} \rangle,$ 
 $\dots, \dots, \dots \}$ 
```

Concretization includes many more states (in red) than what was abstracted...

Abstract Interpretation: Step 2

1. select/define an abstract domain
 - selected based on the type of **properties** you want to prove
2. define abstract semantics **for the language** w.r.t. to the domain
 - prove **sound** w.r.t **concrete semantics**
 - involves defining abstract transformers
 - that is, effect of statement / expression on the abstract domain

we still need to actually **compute** $\alpha^i (\llbracket P \rrbracket)$
(or an **over-approximation** of it)

We need to approximate F

We want a function F^i where:

$$F^i : (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)) \rightarrow (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i))$$

such that: $\alpha^i(\text{lfp } F) \sqsubseteq \text{lfp } F^i$

The best transformer is: $\alpha^i \circ F \circ \gamma^i$

Recall: $F^\#$

$F^\# : (\text{Lab} \rightarrow A) \rightarrow (\text{Lab} \rightarrow A)$

$$F^\#(m)\ell = \begin{cases} T & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell', \text{action}, \ell)} [\![\text{action}]\!](m(\ell')) & \text{otherwise} \end{cases}$$

$[\![\text{action}]\!]: A \rightarrow A$

$[\![\text{action}]\!]$ is the key ingredient here. It captures the effect of a language statement on the abstract domain A . Once we define it, we have $F^\#$

$[\![\text{action}]\!]$ is often referred to as the **abstract transformer**.

Now just instantiate $F^\#$ to obtain F^i

F^i

$$F^i : (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i)) \rightarrow (\text{Lab} \rightarrow (\text{Var} \rightarrow L^i))$$

Here is a definition of F^i which **approximates** the best transformer but **works only on the abstract domain**:

$$F^i(m)^\ell = \begin{cases} \lambda v. [-\infty, \infty] & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell', \text{action}, \ell)} \llbracket \text{action} \rrbracket_i(m(\ell')) & \text{otherwise} \end{cases}$$

$$\llbracket \text{action} \rrbracket_i : (\text{Var} \rightarrow L^i) \rightarrow (\text{Var} \rightarrow L^i)$$

what is $(\ell', \text{action}, \ell)$?

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (0 ≤ i) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7: }
```

Actions:

```
(1, x := 5, 2)  
(2, y := 7, 3)  
(3, 0 ≤ i, 4)  
(3, 0 > i, 7)  
(4, y = y + 1, 5)  
(5, i := i - 1, 6)  
(6, goto 3, 3)
```

Multiple (two) actions reach label 3

what is $(\ell', \text{action}, \ell)$?

- $(\ell', \text{action}, \ell)$ is an edge in the control-flow graph
- More formally, if **there exists** a transition $t = \langle \ell', \sigma' \rangle \rightarrow \langle \ell, \sigma \rangle$ in a program trace in P , where t was performed by statement called **action**, then $(\ell', \text{action}, \ell)$ must exist. This says that we are **sound**: we never miss a flow.
- However, $(\ell', \text{action}, \ell)$ may exist even if no such transition t above occurs. In this case, the analysis would be imprecise as we would unnecessarily create more flows.

what is $(\ell', \text{action}, \ell)$?

An **action** can be:

- $b \in \text{BExp}$ boolean expression in a conditional here, $a \in \text{AExp}$
- $x := a$
- skip

Next, we will define the effect of some of these things formally, while with others we will proceed by example.

The key point is to make sure that $\llbracket \text{action} \rrbracket_i$ produces sound and precise results.

F^i on our example

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
7:  
}
```

$$F^i(m)1 = \lambda v. [-\infty, \infty]$$

$$F^i(m)2 = \llbracket x := 5 \rrbracket_i(m(1))$$

$$F^i(m)3 = \llbracket y := 7 \rrbracket_i(m(2)) \sqcup \llbracket \text{goto 3} \rrbracket_i(m(6))$$

$$F^i(m)4 = \llbracket i \geq 0 \rrbracket_i(m(3))$$

$$F^i(m)5 = \llbracket y := y + 1 \rrbracket_i(m(4))$$

$$F^i(m)6 = \llbracket i := i - 1 \rrbracket_i(m(5))$$

$$F^i(m)7 = \llbracket i < 0 \rrbracket_i(m(3))$$

$\llbracket x := a \rrbracket_i$

$$\llbracket x := a \rrbracket_i(m) = m[x \mapsto v] \quad , \quad \text{where } \langle a, m \rangle \downarrow_i v$$

$\langle a, m \rangle \downarrow_i v$ says that given a map m , the expression a evaluates to a value $v \in L^i$

The operational semantics rules for expression evaluation are as before, except:

- any constant Z is abstracted to an element in L^i
- operators $+$, $-$ and $*$ are re-defined for the Interval domain

Arithmetic Expressions

If we **add** \perp_i to any other element, we get \perp_i .

If both operands are not \perp_i , we get:

$$[x, y] + [z, q] = [x + z, y + q]$$

what about $*$?

is $[x, y] * [z, q] = [x * z, y * q]$ **sound** ?

Ex: $[-1,2] * [1,2] = ?$

$\llbracket b \rrbracket_i$

Let us first look at the expression: $a_1 \ c \ a_2$

For a map m , we need to define : $\llbracket a_1 \ c \ a_2 \rrbracket_i(m)$

which produces another map as a result.

Here, c is a condition such as: $\leq, =, <$

Recall our example: what is $\llbracket x \leq y \rrbracket$?

Suppose we have the program:

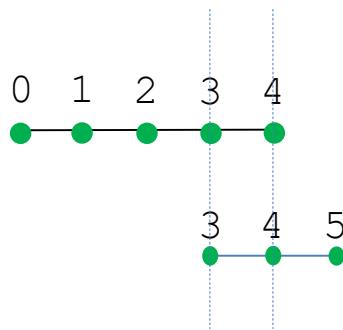
```
// Here, x is [0, 4] and y is [3, 5]
if (x ≤ y) {
    1: ...
}
```

What does $\llbracket x \leq y \rrbracket$ produce at label 1 ?

That is, what are the intervals for x and y at label 1 ?

Definition of $[l_1, u_1] \leq [l_2, u_2]$

What should $[0, 4] \leq [3, 5]$ produce?



One answer is: $([0, 3], [3, 5])$. Is it **sound**?

Another, non-comparable answer is: $([0, 4], [4, 5])$. Is it **sound**?

What about (\top, \top) – **sound**?

Can you find a **more precise** answer ?

Definition of $[l_1, u_1] \leq [l_2, u_2]$

$$[l_1, u_1] \leq [l_2, u_2] = ([l_1, u_1] \sqcap_i [-\infty, u_2], [l_1, \infty] \sqcap_i [l_2, u_2])$$

$$\begin{aligned}[0, 4] \leq [3, 5] &= ([0, 4] \sqcap_i [-\infty, 5], [0, \infty] \sqcap_i [3, 5]) \\ &= ([0, 4], [3, 5])\end{aligned}$$

Exercise: define $<$ and $=$

Evaluating $\llbracket b \rrbracket_i$

$$\llbracket b_1 \vee b_2 \rrbracket_i (m) = \llbracket b_1 \rrbracket_i (m) \sqcup \llbracket b_2 \rrbracket_i (m)$$

$$\llbracket b_1 \wedge b_2 \rrbracket_i (m) = \llbracket b_1 \rrbracket_i (m) \sqcap \llbracket b_2 \rrbracket_i (m)$$

Abstract Interpretation: Step 3

1. select/define an abstract domain
 - selected based on the type of **properties** you want to prove
2. define abstract semantics **for the language** w.r.t. to the domain
 - prove **sound** w.r.t **concrete semantics**
 - involves defining abstract transformers
 - that is, effect of statement / expression on the abstract domain
3. iterate abstract transformers over the abstract domain
 - until we reach a **fixed point**

Rigorous Software Engineering

Rigorous Testing: Symbolic and Concolic Execution

Prof. Martin Vechev

Soundiness...

soundiness.org

Soundiness Home Page

If you wanted to learn about soundiness, you are in the right place. Below is a brief excerpt from our Soundiness manifesto...

Static program analysis is a key component of many software development tools, including compilers, development environments, and verification tools. Practical applications of static analysis have grown in recent years to include tools by companies such as Coverity, Fortify, GrammaTech, IBM, and others. Analyses are often expected to be

Question

Can you build an **automatic** analyzer which takes as input an **arbitrary** program and an **arbitrary** property such that if the analyzer answers:

- “**Yes**”, then it is certain that the property holds
- “**No**”, then it is certain that the property does not hold

Question

Can you build an **automatic** analyzer which takes as input an **arbitrary** program and an **arbitrary** property such that if the analyzer answers:

- “**Yes**”, then it is certain that the property holds
- “**No**”, then it is certain that the property does not hold

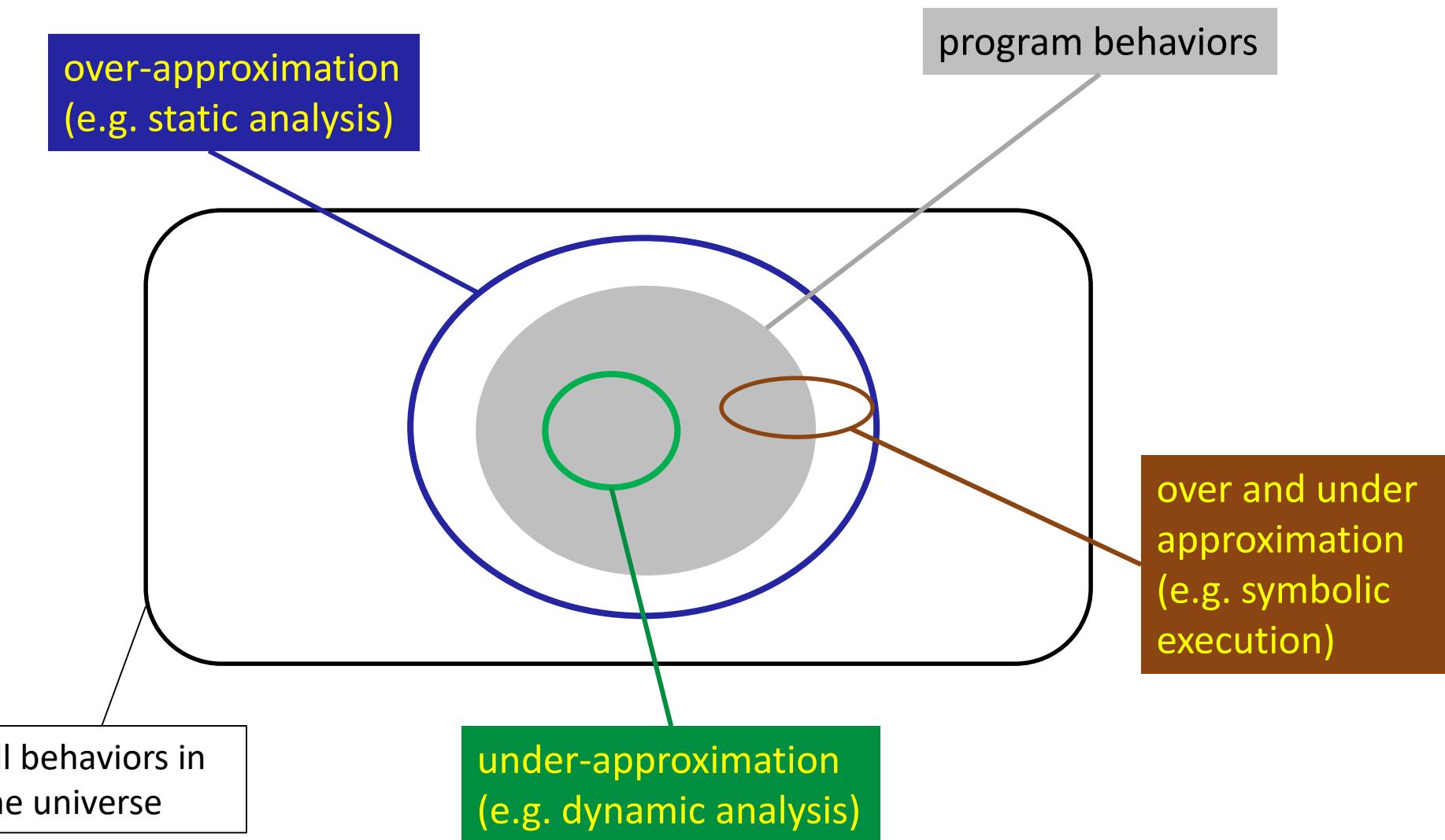


Answer:

No. The problem is undecidable

Alan Turing

Approaches to Program Analysis



Today: Symbolic Execution

Symbolic Execution

Symbolic execution is a technique which sits in between testing and static analysis.

It is completely automatic, and aims to explore as many program executions as possible, with the expense that it has false negatives: it may miss program executions, that is, may miss errors.

Hence, symbolic execution is a particular instance of an under-approximation (some versions are actually both under- and over- approximations)

Symbolic Execution: Applications

Symbolic execution is widely used in practice. Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems:

- Network servers
- File systems
- Device drivers
- Unix utilities
- Computer vision code
- ...

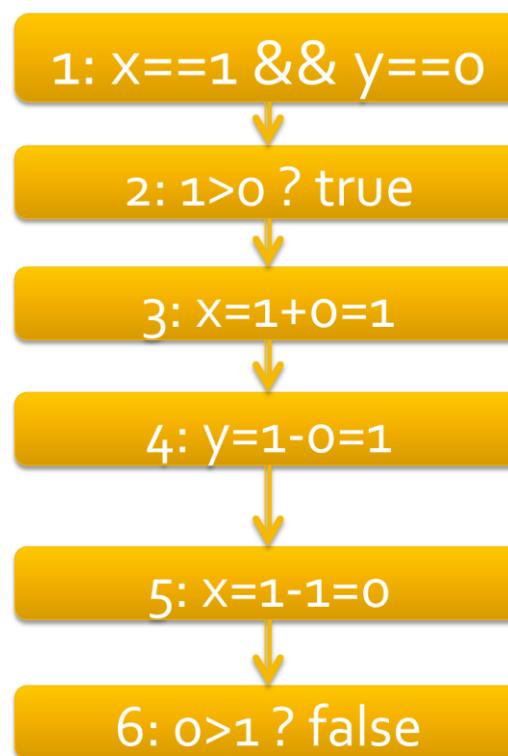
Symbolic Execution: The Idea

In classic symbolic execution, we associate with each variable a **symbolic value** instead of a concrete value. We then run the program with the symbolic values obtaining a **big constraint formula** as we run the program. Hence, the name **symbolic execution**.

At any program point we can invoke a constraint (SMT) solver to find **satisfying assignments** to the formula. These satisfying assignments can be used to indicate **real concrete inputs** for which the program reaches a program point or to **steer the analysis** to another part of the program.

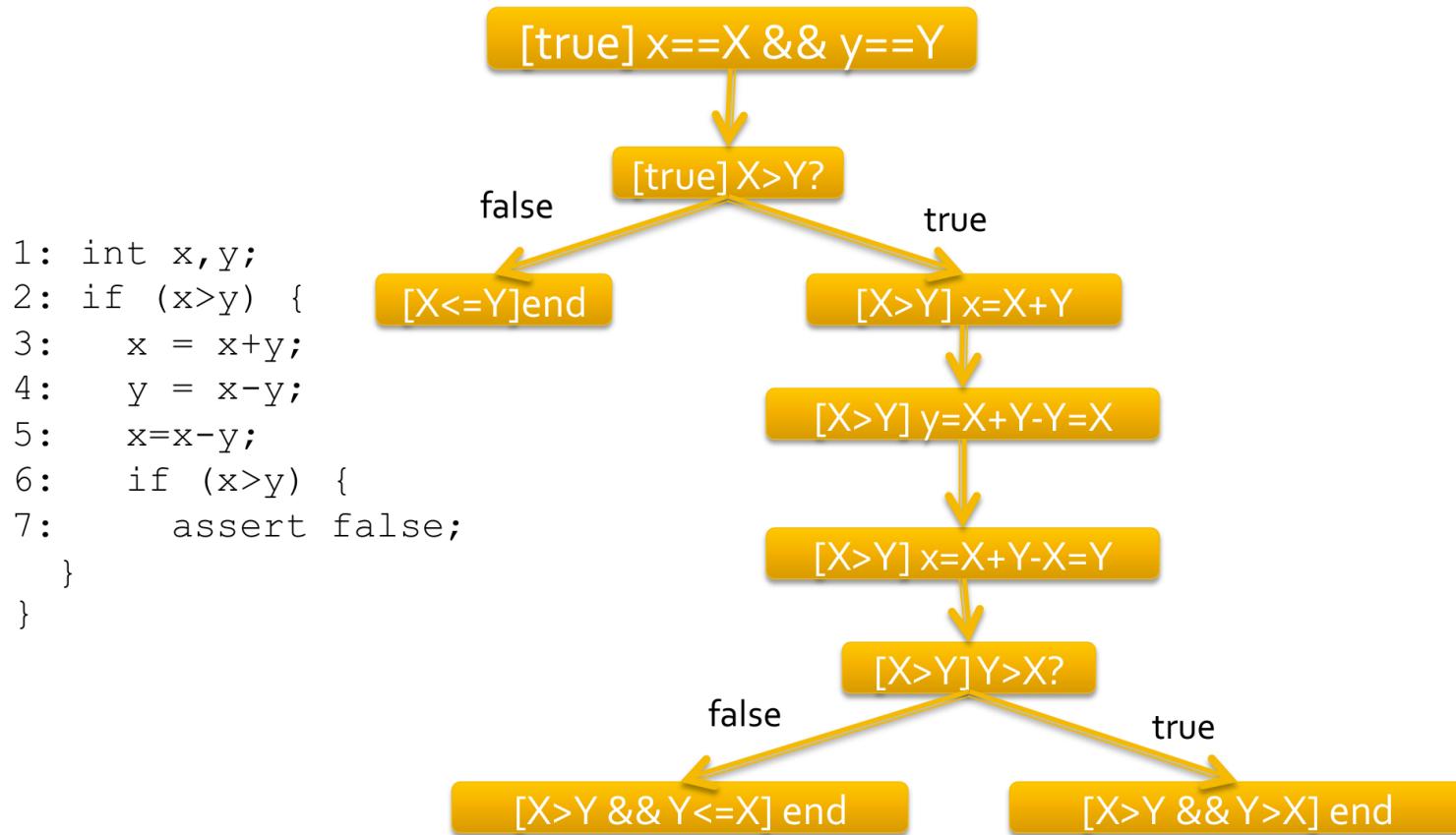
Example: Concrete Execution

```
1: int x,y;  
2: if (x>y) {  
3:     x = x+y;  
4:     y = x-y;  
5:     x=x-y;  
6:     if (x>y) {  
7:         assert false;  
    }  
}
```



- `assert false` abstracts over any failing statement
- Flowchart corresponds to execution of a single unit test
- This and next slide taken from Galeotti/Gorla/Rau, Saarland University (<https://www.st.cs.uni-saarland.de/edu/automatedtestingverification12/>)

Example: Symbolic Execution



- Flowchart visualises symbolic execution of the program
- Parts in brackets, e.g. $[X > Y]$ are *path constraints*, i.e. constraints under which a path is executed
- Notice that symbolic executions *branches*, i.e. explores both then- and else-branches

Constraints: Examples

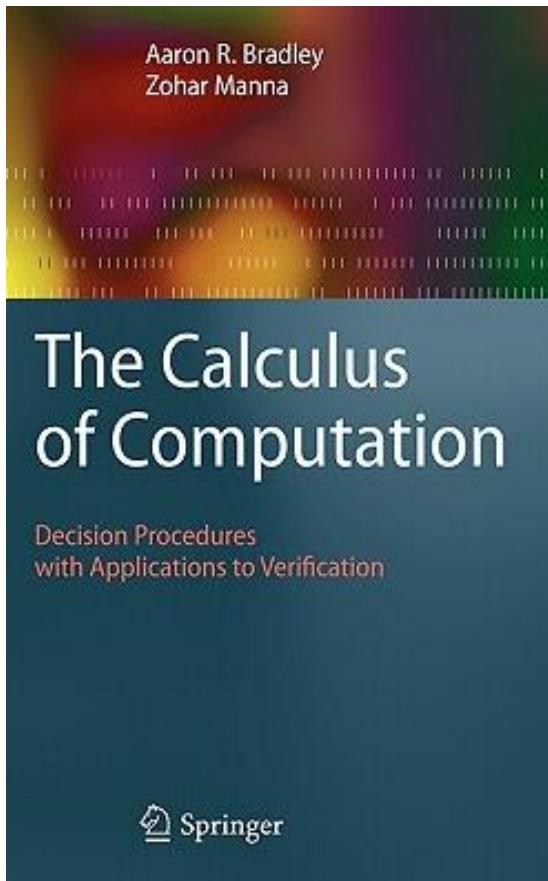
Linear constraint: $5*x + 6 < 100$

Non-linear constraint: $x * y + 12 < 29$

Uninterpreted functions: $f(x) < 30$

A constraint solver, typically an SMT solver, finds satisfying assignments to constraints. An example of SMT solvers are Z3 and Yices.

Logical Fragments of Constraints



What is decidable ?

Theory	Description	Full	QFF
T_E	equality	no	yes
T_{PA}	Peano arithmetic	no	no
T_N	Presburger arithmetic	yes	yes
T_Z	linear integers	yes	yes
T_R	reals (with \cdot)	yes	yes
T_Q	rationals (without \cdot)	yes	yes
T_{RDS}	recursive data structures	no	yes
T_{RDS}^+	acyclic recursive data structures	yes	yes
T_A	arrays	no	yes
$T_A^=$	arrays with extensionality	no	yes

Theory of Equality: Example

$$a = b \wedge b = c \rightarrow g(f(a), b) = g(f(c), a)$$

Is this a valid formula?

Theory of Integers

$$\forall x, y, z. \quad x > z \wedge y \geq 0 \rightarrow x + y > z$$

Is this formula valid?

How about this one?

$$\forall x, y. \quad x > 0 \wedge (x = 2y \vee x = 2y + 1) \rightarrow x - y > 0$$

Symbolic Execution: Technically

At any point during program execution, symbolic execution keeps two formulas:

symbolic store and a path constraint

Therefore, at any point in time the symbolic state is described as the conjunction of these two formulas.

Symbolic Store

- The values of variables at any moment in time are given by a function $\sigma_s \in \text{SymStore} = \text{Var} \rightarrow \text{Sym}$
 - Var is the set of variables as before
 - Sym is a set of symbolic values
 - σ_s is called a **symbolic store**
- Example: $\sigma_s : x \mapsto x_0, y \mapsto y_0$

Semantics

- Arithmetic expression evaluation simply manipulates the symbolic values
- Let $\sigma_s : x \mapsto x0, y \mapsto y0$
- Then, $z = x + y$ will produce the symbolic store:
 $x \mapsto x0, y \mapsto y0, z \mapsto x0+y0$

That is, we literally keep the **symbolic expression** $x0+y0$

Path Constraint

- The analysis keeps a path constraint (**pct**) which records the **history of all branches taken so far**. The path constraint is simply a formula.
- The formula is typically in a decidable logical fragment without quantifiers
- At the start of the analysis, the path constraint is **true**
- Evaluation of **conditionals** affects the path constraint , but not the symbolic store.

Path Constraint: Example

Let $\sigma_s : x \mapsto x_0, y \mapsto y_0$

Let pct = $x_0 > 10$

Lets evaluate: if $(x > y + 1)$ { 5: ... }

At label 5, we will get the symbolic store σ_s . It does not change. But we will get an **updated path constraint**:

$$\text{pct} = x_0 > 10 \wedge x_0 > y_0 + 1$$

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

What inputs cause the program to reach the error?

Here, we use **ERROR** for illustration purposes, but in practice, **ERROR** can be an array out of bounds, assertion violation or some other problem

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Lets execute this example
with classic symbolic execution

In practice, we don't know where
the errors is, so we need to search
and explore all paths (or as many
symbolic paths as we can)

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

The `read()` functions read a value from the input and because we don't know what those read values are, we set the values of `x` and `y` to fresh symbolic values called `x0` and `y0`

`pct` is true because so far we have not executed any conditionals

$$\sigma_s : \begin{aligned} x &\mapsto x0, \\ y &\mapsto y0 \end{aligned} \quad pct : \text{true}$$

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}
```

```
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}
```

```
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

pct : true

Here, we simply executed the function `twice()` and added the new symbolic value for `z`.

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

We forked the analysis into 2 paths: the true and the false path. So we **duplicate** the state of the analysis.

This is the result if $x = z$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$pct : x_0 = 2*y_0$$

This is the result if $x \neq z$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$pct : x_0 \neq 2*y_0$$

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

We can avoid further exploring a path if we know the constraint `pct` is **unsatisfiable**. In this example, both `pct`'s are **satisfiable** so we need to keep exploring both paths.

This is the result if $x = z$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$pct : x_0 = 2*y_0$$

This is the result if $x \neq z$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$pct : x_0 \neq 2*y_0$$

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

Lets explore the path when $x == z$ is true.
Once again we get 2 more paths.

This is the result if $x > y + 10$:

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

$$\text{pct} : \begin{aligned} x_0 &= 2*y_0 \\ &\wedge \\ x_0 &> y_0 + 10 \end{aligned}$$

This is the result if $x \leq y + 10$:

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

$$\text{pct} : \begin{aligned} x_0 &= 2*y_0 \\ &\wedge \\ x_0 &\leq y_0 + 10 \end{aligned}$$

Symbolic Execution: Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

So the following path reaches “**ERROR**”.

This is the result if $x > y + 10$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ &y \mapsto y_0 \\ &z \mapsto 2*y_0\end{aligned}$$

$$\begin{aligned}pct : \quad x_0 &= 2*y_0 \\ &\wedge \\ &x_0 > y_0 + 10\end{aligned}$$

We can now ask the SMT solver for a satisfying assignment to the pct formula.

For instance, $x_0 = 40$, $y_0 = 20$ is a satisfying assignment. That is, running the program with these inputs triggers the error.

Handling Loops: a limitation

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < k; i++)  
        sum += i;  
    return sum;  
}
```

A serious limitation of symbolic execution is handling unbounded loops. Symbolic execution runs the program for a finite number of paths. But what if we do not know the bound on a loop ? The symbolic execution will keep running **forever** !

Handling Loops: bound loops

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < 2; i++)  
        sum += i;  
    return sum;  
}
```

A **common solution in practice** is to provide some loop bound. In this example, we can bound k , to say 2. This is an example of an **under-approximation**. Practical symbolic analyzers usually under-approximate as most programs have unknown loop bounds.

Constraint Solving: challenges

Constraint solving is fundamental to symbolic execution as a constraint solver is continuously invoked during analysis. Often, the main **roadblock to performance** of symbolic execution engines is the time spent in constraint solving. Therefore, it is important that:

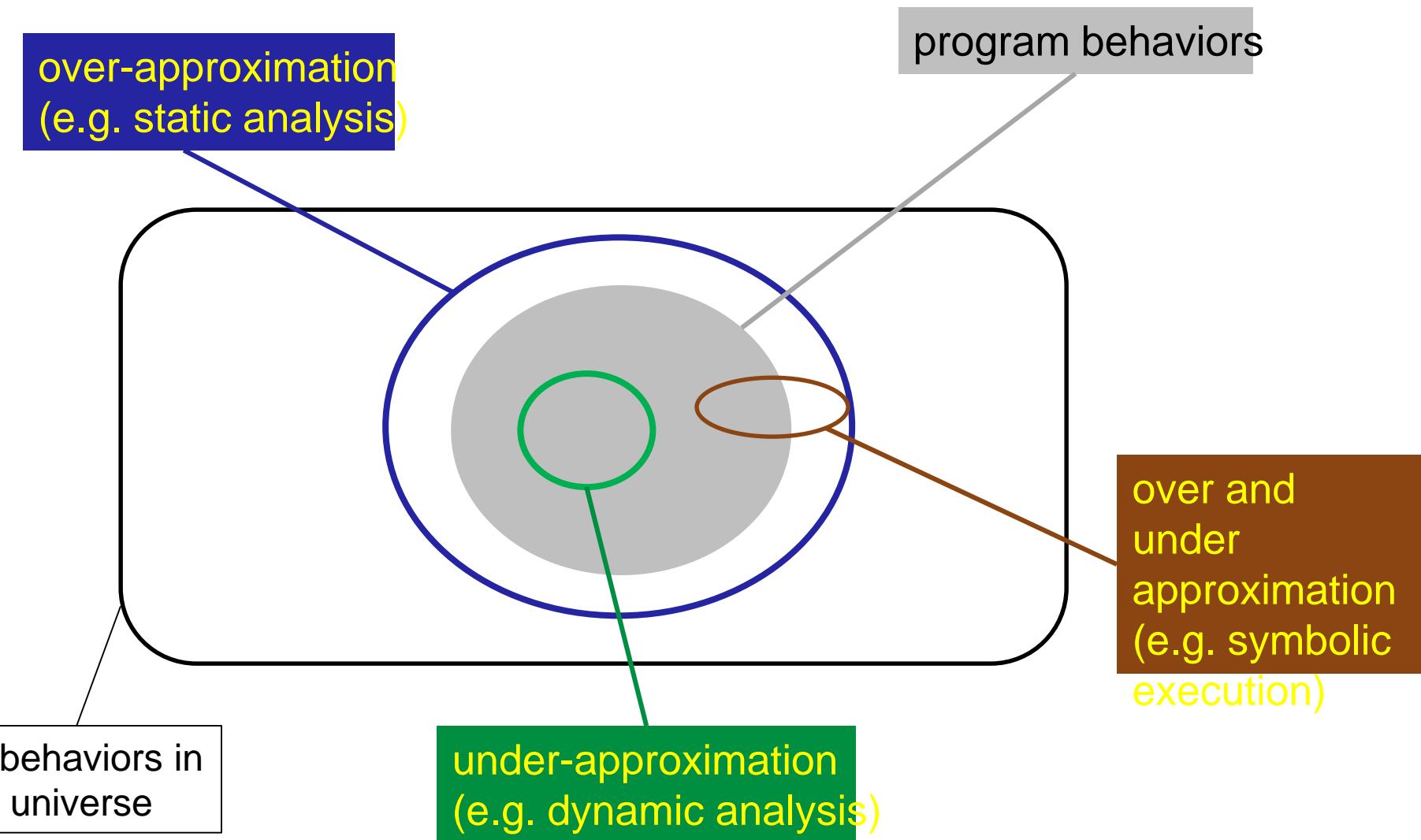
1. The SMT solver supports as many decidable logical fragments as possible. Some tools use more than one SMT solver.
2. The SMT solver can solve large formulas quickly.
3. The symbolic execution engines tries to reduce the burden in calling the SMT solver by exploring domain specific insights.

Rigorous Software Engineering

Dynamic Analysis: Dynamic Race Detection

Prof. Martin Vechev

Approaches to Program Analysis



Dynamic Race Detection

- A popular kind of **dynamic analysis**
 - The analysis is an under-approximation: it considers a subset of the program behaviors
- **Highly effective** for finding concurrency bugs
- Many **different variants**
 - Interesting trade-off between **asymptotic complexity** and **precision** of the analysis

Today

We will illustrate the **key concepts** of race detection on a rich application domain that is quite prevalent today, namely **event-driven** applications such as Web pages and Android

All concepts we study today apply to other settings: e.g. regular concurrent Java programs.

Motivation: Event-Driven Applications



~ 3.5 billion smartphones

A screenshot of a CNN news article. The header reads "India's Covid catastrophe could make global shortages even worse". Below the headline is a photograph of a street scene in India where many shop fronts have their metal roll-up shutters closed. To the right of the main article are two smaller images: one showing a group of people in a hallway, and another showing a large plume of smoke or fire. The CNN navigation bar at the top includes links for World, US Politics, Business, Health, Entertainment, Style, Travel, Sports, and Videos. A COVID-19 update section is also visible.

~ 4.2 billion web pages

Reacts to events: user clicks, arrival of network requests

Event-Driven Applications

Wanted: fast response time

Highly Asynchronous,
Complex control flow

Looks Like This

CNN World US Politics Business Health Entertainment Style Travel Sports Videos

COVID-19 Live updates | Vaccinations | US Coast Guard cutter | Caitlyn Jenner | Jeff Bezos | NASA spacecraft |

India's Covid catastrophe could make global shortages even worse



A man walks past a row of closed shop shutters in a market area. The shutters are made of metal and are all closed. The man is wearing a dark shirt and light trousers. The background shows other closed shutters and some signs.

SAROB/MEEED/ZUMA PRESS/GETTY IMAGES

<https://edition.cnn.com/2021/05/10/business/india-covid-industries-intl-hnk/index.html>



ANALYSIS

Trump drags House GOP deeper into his theater of lies

- Kinzinger comes out swinging at McCarthy and Trump as he defends Cheney
- **Analysis:** A stunning revelation about the GOP's willful blindness on Trump



This is what runs

Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```



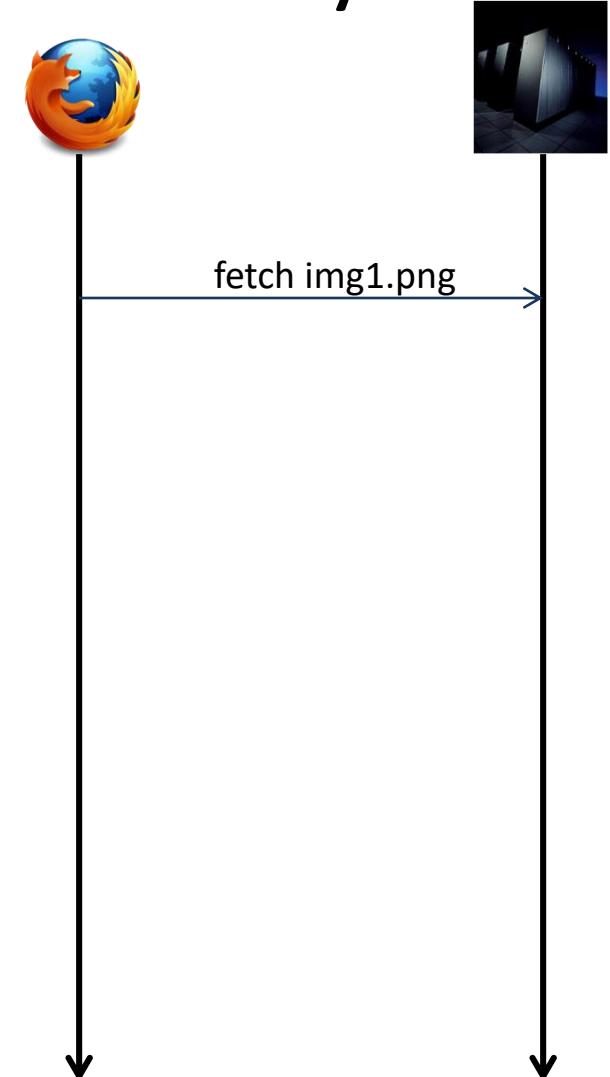
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```



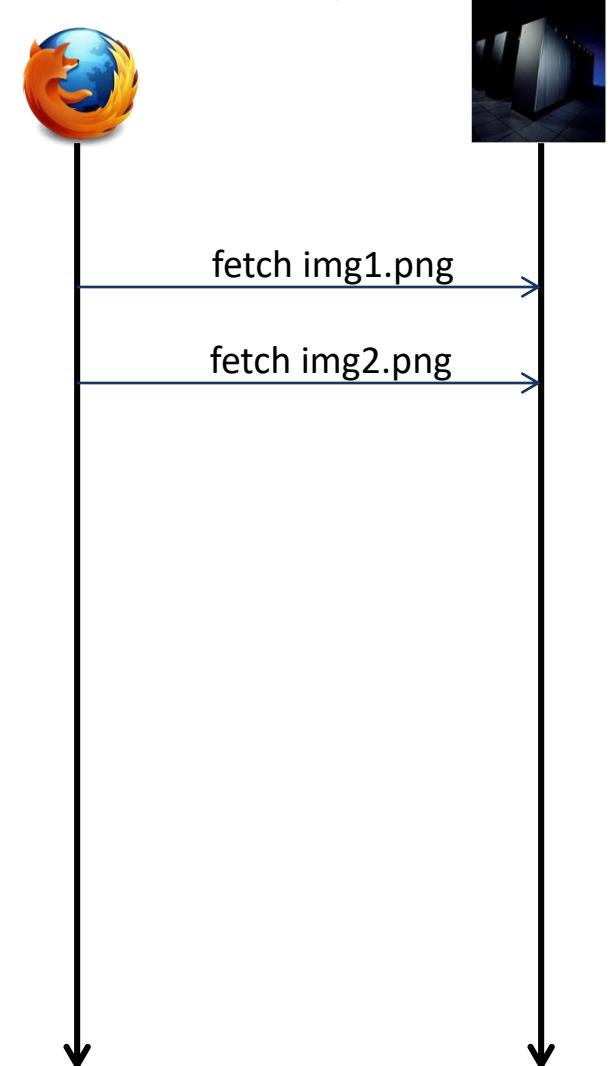
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```



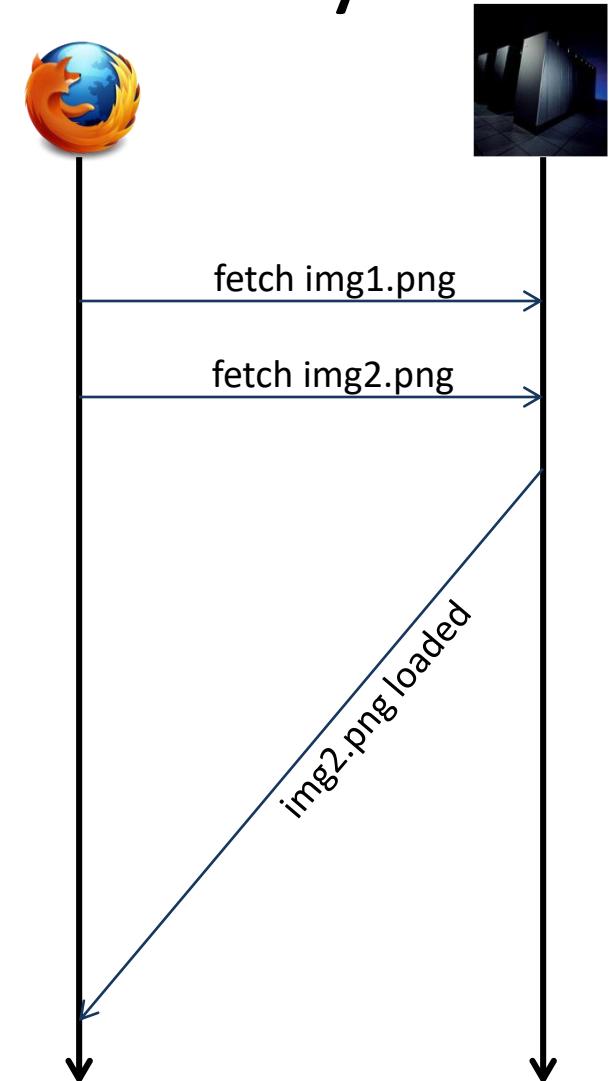
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```

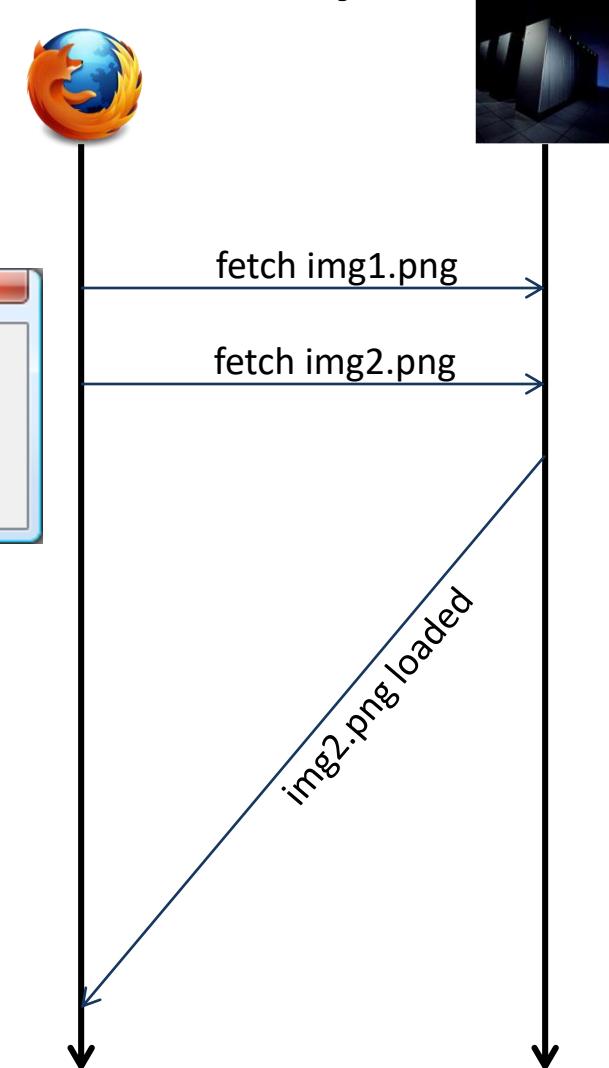
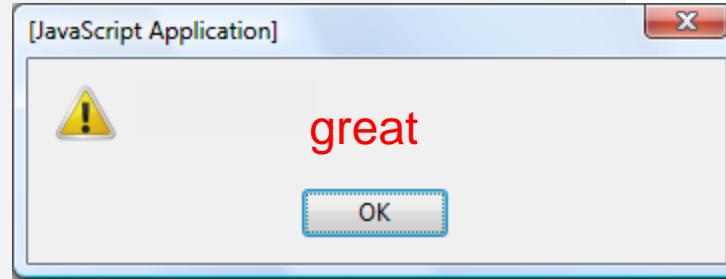


Non-determinism: network latency

```
<html>
<head></head>
<body>
<script>
var RSE = "great";
</script>




</body>
</html>
```



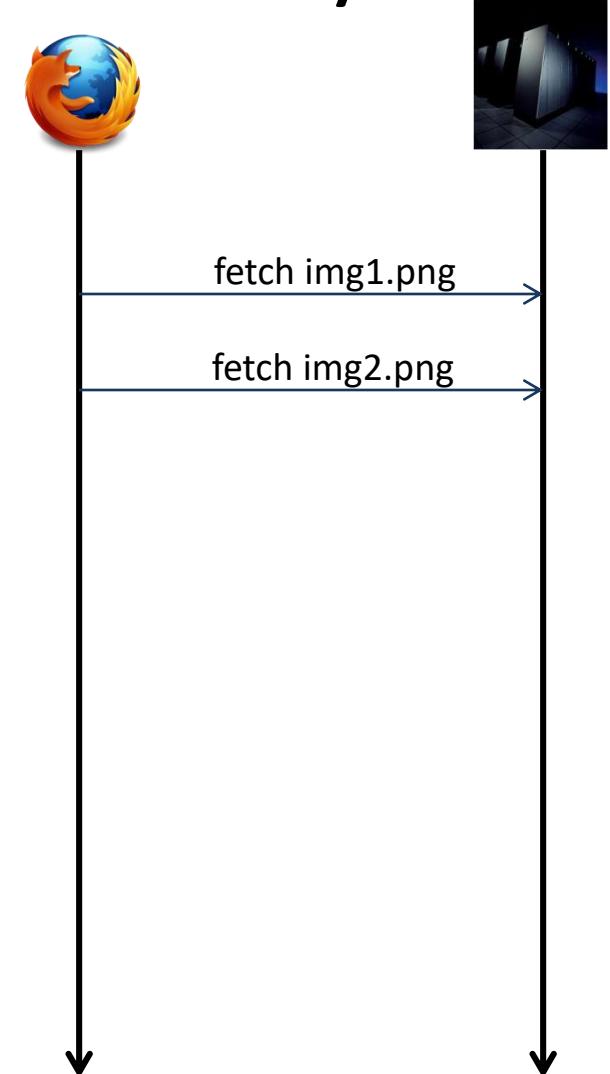
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```



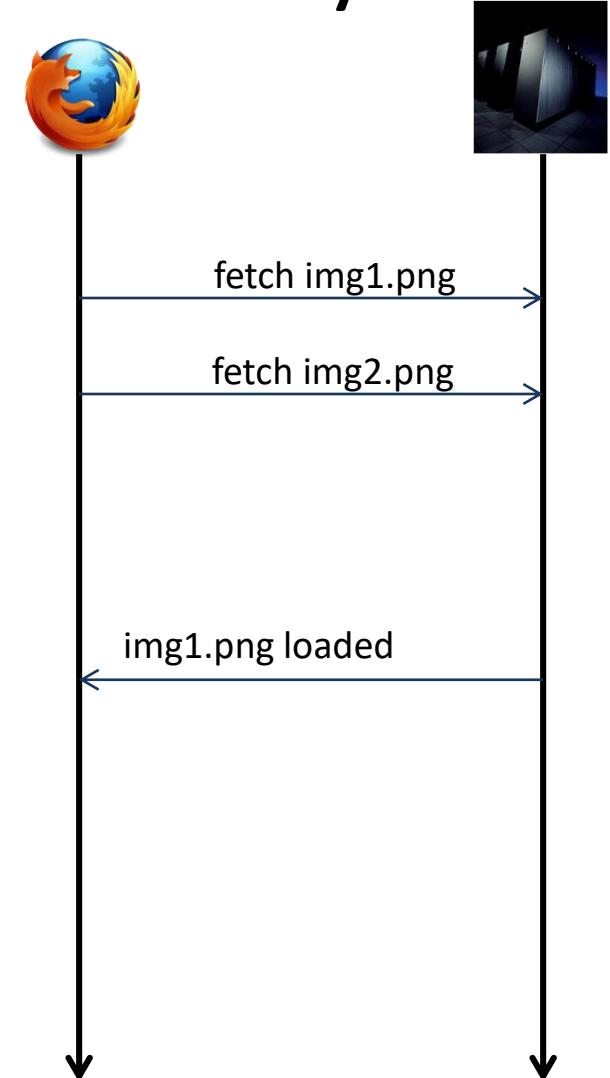
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```



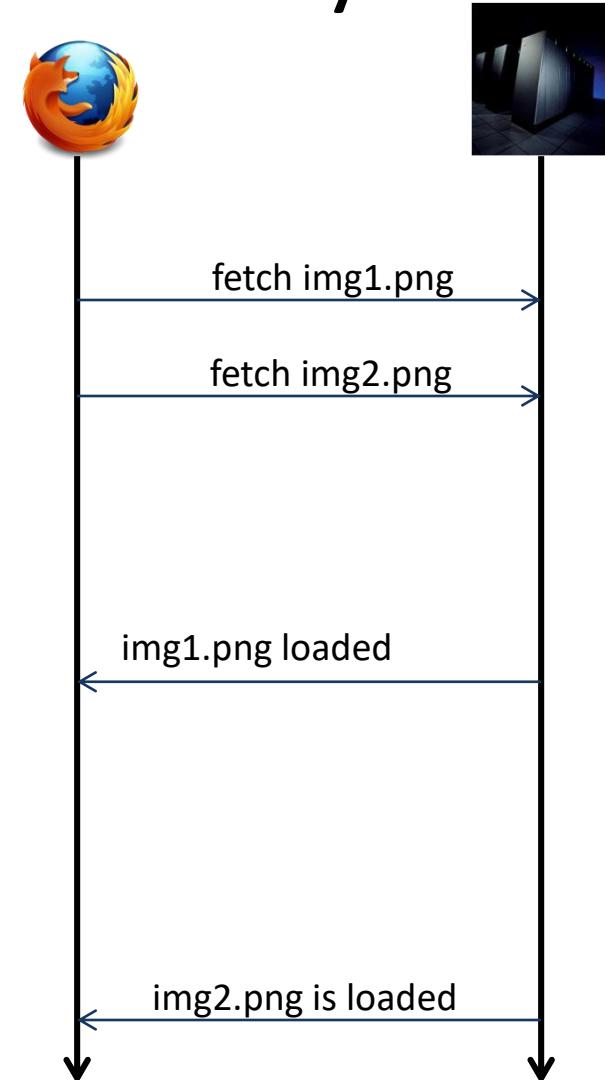
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```

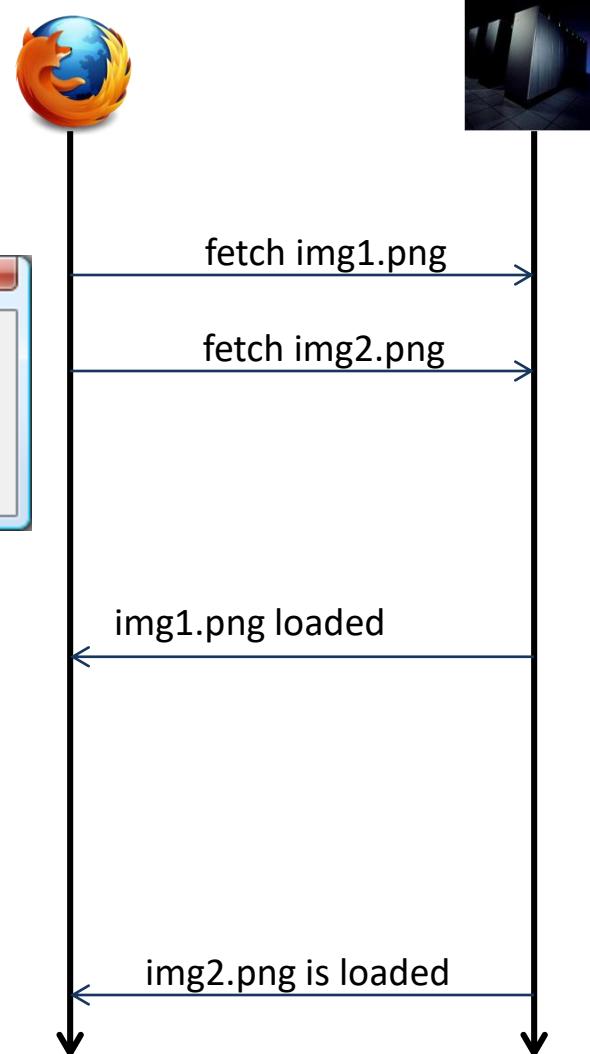
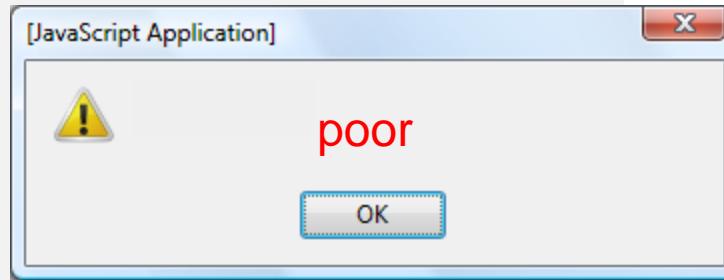


Non-determinism: network latency

```
<html>
<head></head>
<body>
<script>
var RSE = "great";
</script>




</body>
</html>
```



What do we learn from these?

Asynchrony + Shared Memory



Non-Determinism



Unwanted Behavior

What do we learn from these?

Asynchrony + Shared Memory



Non-Determinism



Unwanted Behavior

Lets phrase the problem as **data race detection**

What is a Data Race ?

What is a Data Race ?

Semantically, a data race occurs when we have a reachable program state where:

- we have two outgoing transitions by two different threads
- the two threads access the same memory location
- one of the accesses is a write

Examples [Recap]

Data Race on X

Thread T₁:

fork T₂

X = 1

Thread T₂:

X = 2

Program has No Data Races

Thread T₁:

X = 1

fork T₂

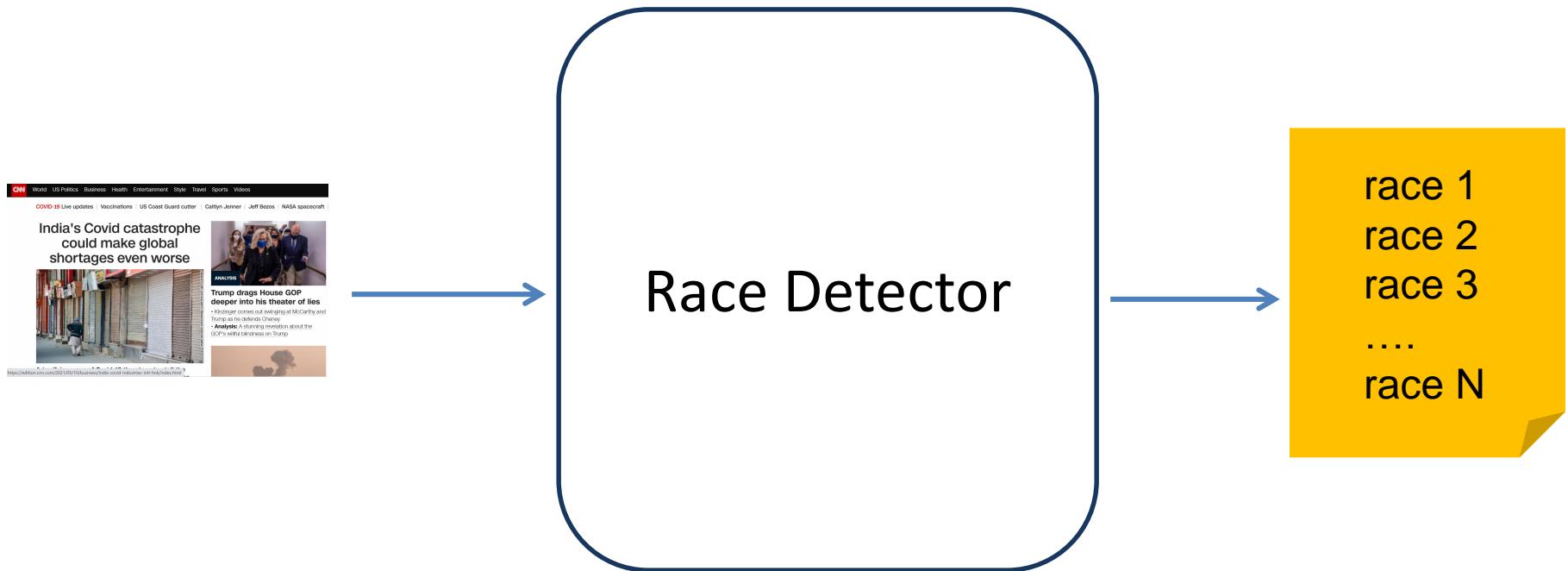
Thread T₂:

X = 2

The program **has a reachable state** where both X = 1 and X = 2 are enabled

The program **does not have a reachable state** where both X = 1 and X = 2 are enabled

Wanted



Naïve Algorithm

The definition of a data race suggests a naïve algorithm which finds **all races of a program** given some input states.

The algorithm simply enumerates all reachable states of the concurrent program from the initial input states and checks the definition on each such reachable state.

Naïve Algorithm

The definition of a data race suggests a naïve algorithm which finds **all races of a program** given some input states.

The algorithm simply enumerates all reachable states of the concurrent program from the initial input states and checks the definition on each such reachable state.

Does Not Scale to Real-World Programs

In Practice

In practice, algorithms aim to scale to large programs by being more efficient and not keeping program states around. To accomplish that, they **weaken their guarantees**.

We will see the guarantees they provide a little later, but at this point it is sufficient to mention that a typical guarantee is that the **first race the algorithm reports is a real race**, but any subsequent reported races after the first race are not guaranteed to exist, that is, they may be **false positives**, a major issue to deal with for any modern analyzer.

False positives exist because of **user-defined synchronization**.

Example of a False Positive Race (on variable X)

```
Initially: X = Y = 0

Thread T1:      ||      Thread T2:

while(Y == 0);
X = 1                      X = 0
                            Y = 1
```

A state of the art race detector may report a **race on X and Y**

Modern Dynamic Race Detection: 5 Steps

Step 1: Define Memory locations (on which races can happen)

Usually easy but there can be issues (framework vs. user-code)

Step 2: Define Happens-Before Model (how operations are ordered)

Can be tricky to get right due to subtleties of concurrency

Step 3: Come up with an Algorithm to detect races

Hard to get good asymptotic complexity + correctness

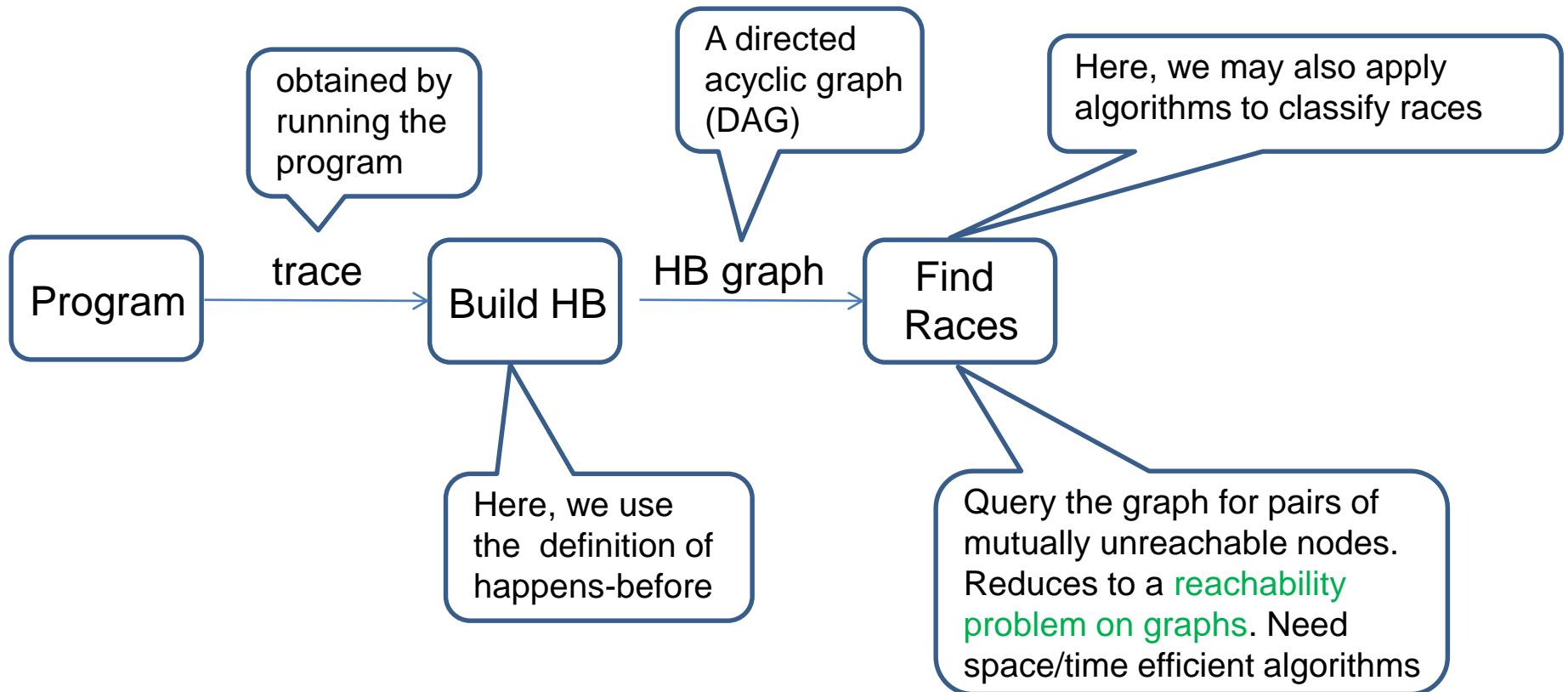
Step 4: Come up with techniques (algorithm, filters) to remove harmless races

Needs to answer what harmless means

Step 5: Implement Algorithm and Evaluate

Important to have low instrumentation overhead

Dynamic Race Detection: Flow



(some of these boxes will become clear later in the slides)

Let us now discuss these 5 steps in our example domain: event-driven applications

These 5 steps need to be taken for **any other domain**

Step 1:



Memory
Locations

- "Normal", C-like, memory locations for JavaScript variables
- Functions are treated like "normal" locations
- HTML DOM elements
- Event, event-target and event-handler tuple

Memory Locations: Example

```
<html>
<head></head>
<body>

<script>
var RSE = "great";
</script>




</body>
</html>
```

Step 2:

Happens-
Before
Model

... is a partial order (A, \leq)

Step 2:

Happens-
Before
Model

... is a partial order (A, \leq)

First, define the contents of A , i.e. atomic **action**

- E.g.: parsing a single HTML element, executing a script, processing an event handler

Step 2:

Happens-
Before
Model

... is a partial order (A, \leq)

First, define the contents of A , i.e. atomic **action**

- E.g.: parsing a single HTML element, executing a script, processing an event handler

Then, define \leq , i.e. how to **order** actions

- E.g.: parsing of HTML elements of the web page is ordered

Happens-Before: Example

```
<html>
<head></head>
<body>
```

```
<script>
var RSE = "great";
</script>
```

```

```

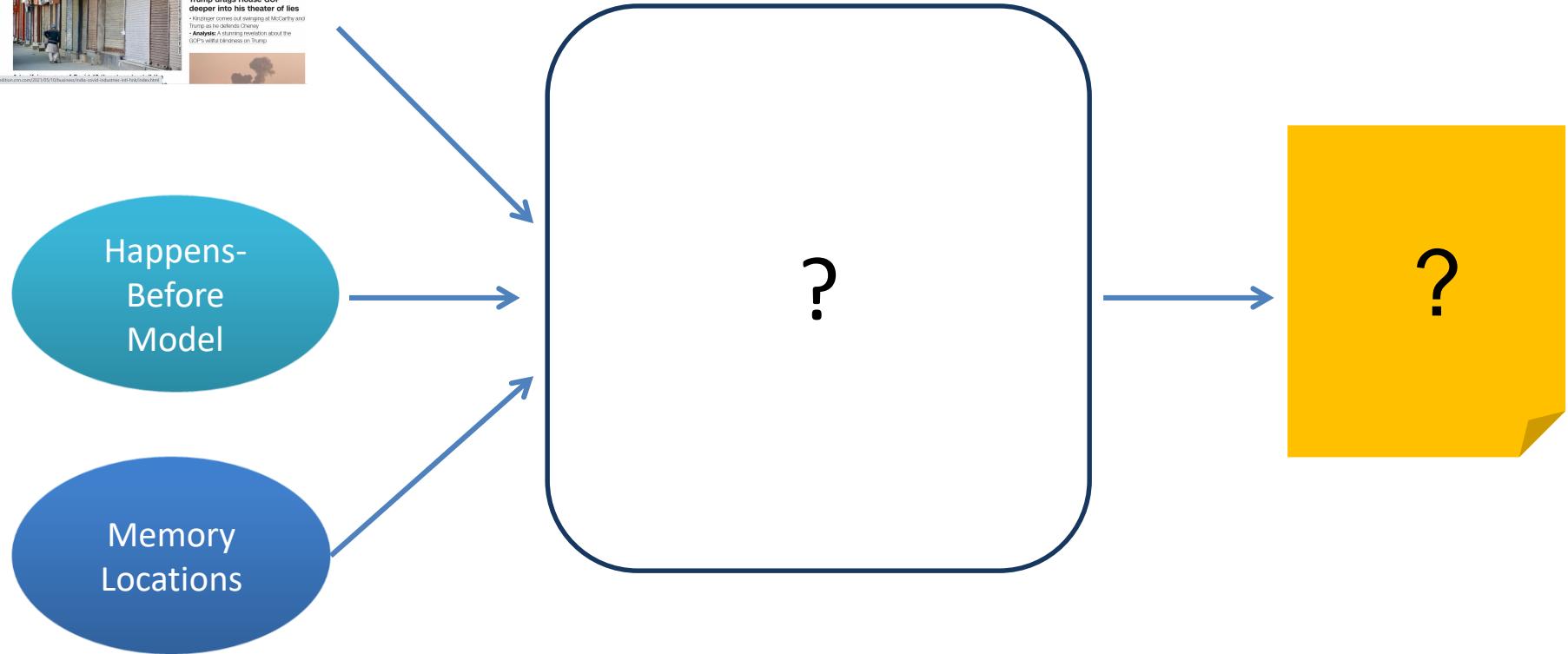
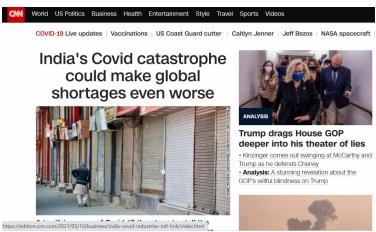
```

```

```
</body>
</html>
```

a data race on RSE

Steps 3 and 4 : Define Race Detection Algorithm



Dynamic Race Detection: Theorems (that an analyzer should ensure)

No false negatives: if the Analysis reports **no races** on an execution, then the execution **must not contain a race**

No false positives: if the Analysis reports **a race** for a given execution then the execution **for sure contains a race**

Two Challenges Affecting Steps 3 and 4

Synchronization done with read/writes



quickly leads to **thousands of false races**

Massive number of event handlers



quickly causes **space blow-up** in analysis data structures

False Positives: Example

```
<html><body>

<script>
var init = false, y = null;
function f() {
    if (init)
        alert(y.g);
    else
        alert("not ready");
}
</script>

<input type="button" id="b1"
       onclick="javascript:f ()">

<script>
y = { g:42 };
init = true;
</script>

</body></html>
```

- 3 variables with races:
init
y
y.g
- some races are synchronization:
init
- reports **false races** on variables:
y
y.g

Wanted: “guaranteed” races

```
<html><body>

<script>
var init = false, y = null;
function f() {
    if (init)
        alert(y.g);
    else
        alert("not ready");
}
</script>

<input type="button" id="b1"
       onclick="javascript:f()">

<script>
y = { g:42 };
init = true;
</script>

</body></html>
```

Intuition: identify races that are guaranteed to exist.

We report races on variable **init**

But not on:

y
y.g

Because races on **y** and **y.g** are covered by the race on **init**

Two Challenges Affecting Steps 3 and 4

Synchronization with read/writes

race coverage eliminates false races



Massive number of event handlers

quickly causes space blow-up in analysis data structures



Computing Races

A race detector should compute races. The basic query is whether two operations **a** and **b** are ordered:

$$a \preccurlyeq b$$

Observation: represent \preccurlyeq (the happens-before of an execution trace) as a **directed acyclic graph** and perform graph connectivity queries to answer $a \preccurlyeq b$

Report a race if **a** and **b** are not reachable from one another, they touch the same memory location and one is a write.

Example \leqslant built from a trace

Lets take the trace: ABCDE.

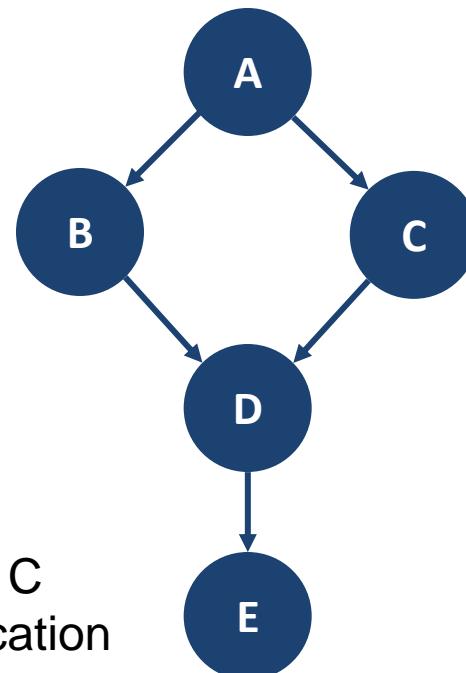
If the happens-before tells us that B and C need not be ordered, but all others are ordered, then we obtain the following graph on the right, also written in text as:

$$\leqslant = \{ (A, B), (A, C), (B, D), (C, D), (D, E), \\ [+ \text{transitive, reflexive closure}] \dots \}$$

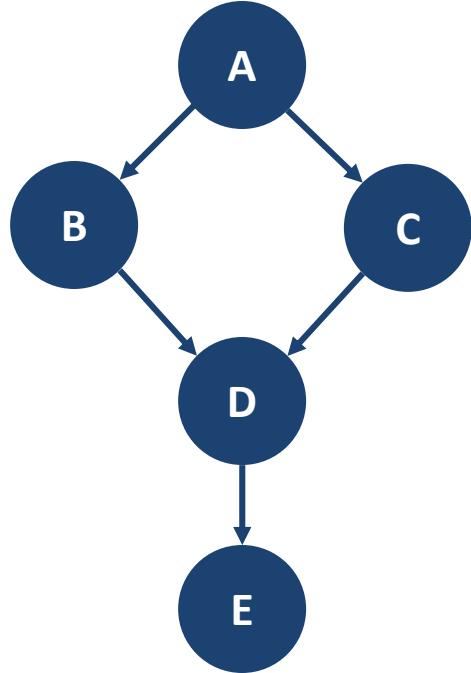
This graph captures that we not only have ABCDE as a trace but we also have ACBDE as a trace

In this example, we would **have a race** between B and C if actions B and C were touching the same memory location and one of them was writing to that location.

The DAG representing \leqslant
(Hasse diagram)



?
a ≼ b via BFS

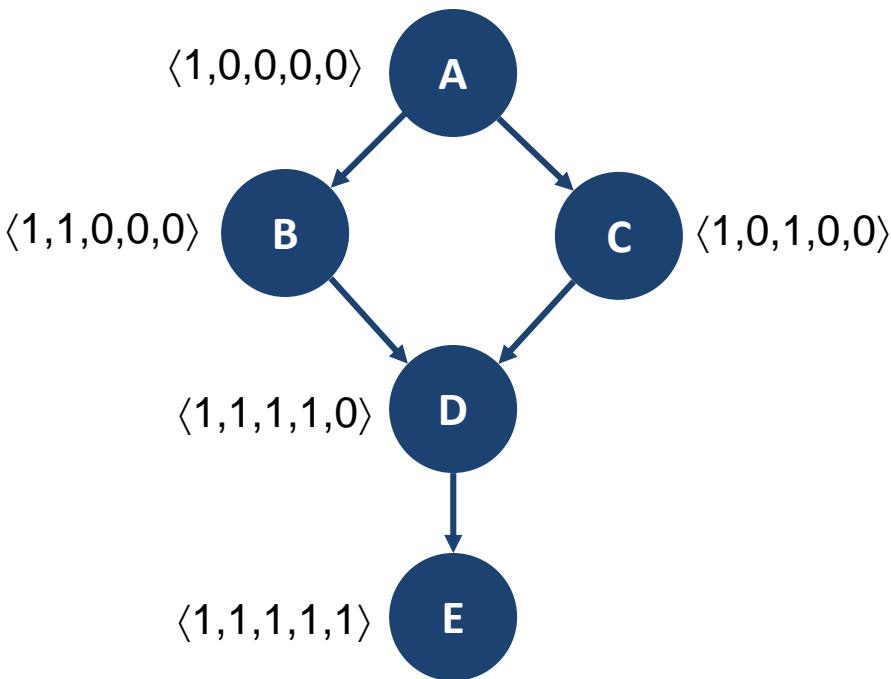


M - number of edges
N - number of nodes

Query Time: O(M)
Space : O(N)

?

$a \preccurlyeq b$ via vector clocks



A vector clock vc is a map:

$$vc \in \text{Nodes} \rightarrow \text{Nat}$$

associate a vector clock
with each node

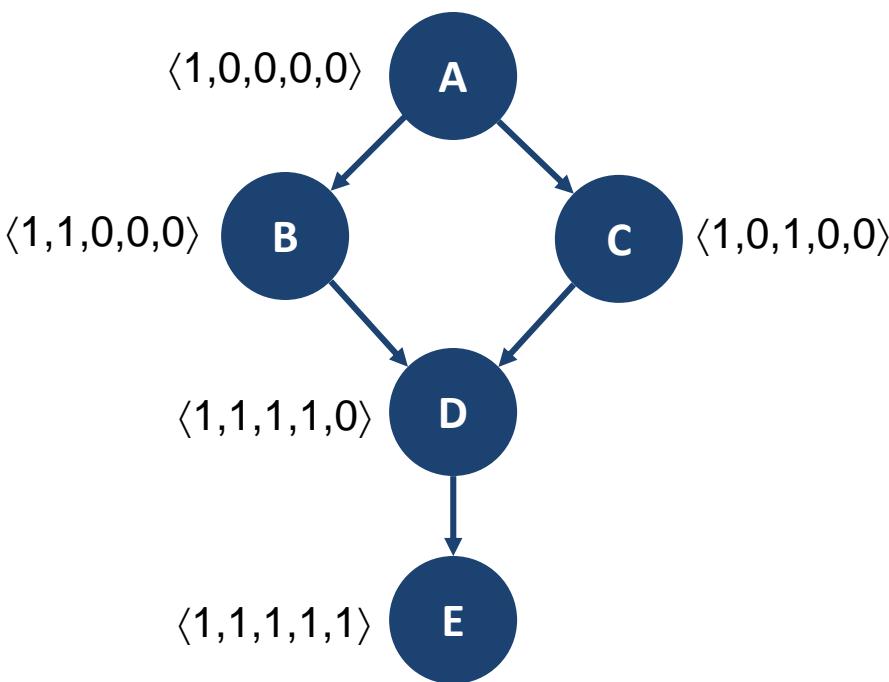
$\langle 1,0,0,0,0 \rangle \sqsubseteq \langle 1,1,1,1,0 \rangle$
it follows that $A \preccurlyeq D$

$\langle 1,1,0,0,0 \rangle \not\sqsubseteq \langle 1,0,1,0,0 \rangle$
it follows that $B \not\preccurlyeq C$

In this example graph, Nodes = {A,B,C,D,E}

?

$a \preccurlyeq b$ via vector clocks



At a given node, its vector clock captures who can reach that node.

For example, for node C, its vector clock $vc-C\langle 1, 0, 1, 0, 0 \rangle$ denotes that:

- A can reach C: because $vc-C(A) = 1$
- B cannot reach C: because $vc-C(B) = 0$
- C can reach C: because $vc-C(C) = 1$
- D cannot reach C: because $vc-C(D) = 0$
- E cannot reach C: because $vc-C(E) = 0$

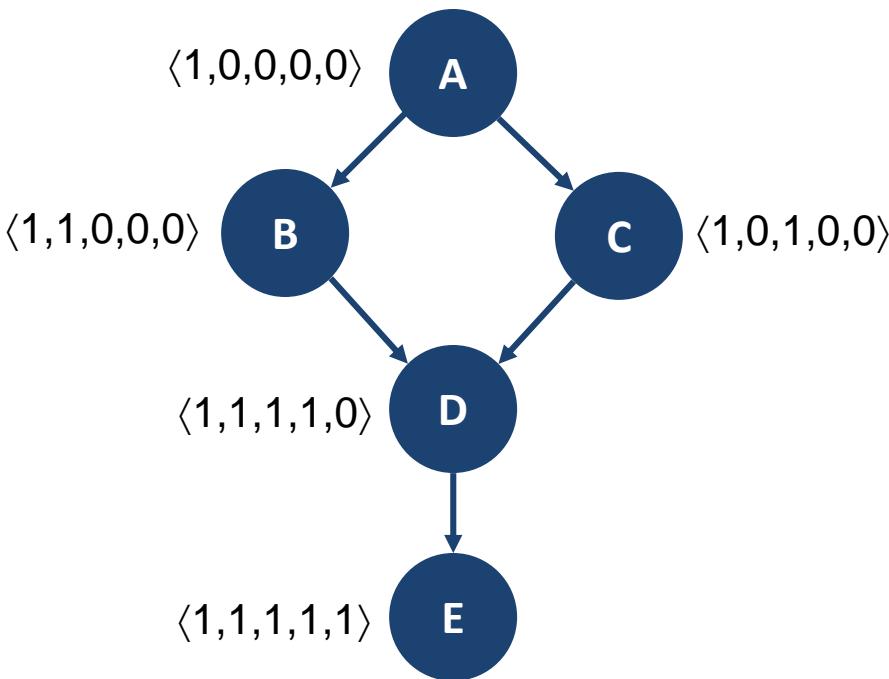
Given two nodes, say B and C, we can determine whether they are mutually unreachable by just checking:

whether $vc-C(B) = 0$ and $vc-B(C) = 0$

This is constant-time work.

?

$a \leq b$ via vector clocks



To compute the vector clocks, simply process each edge of the graph and join the vector clocks.

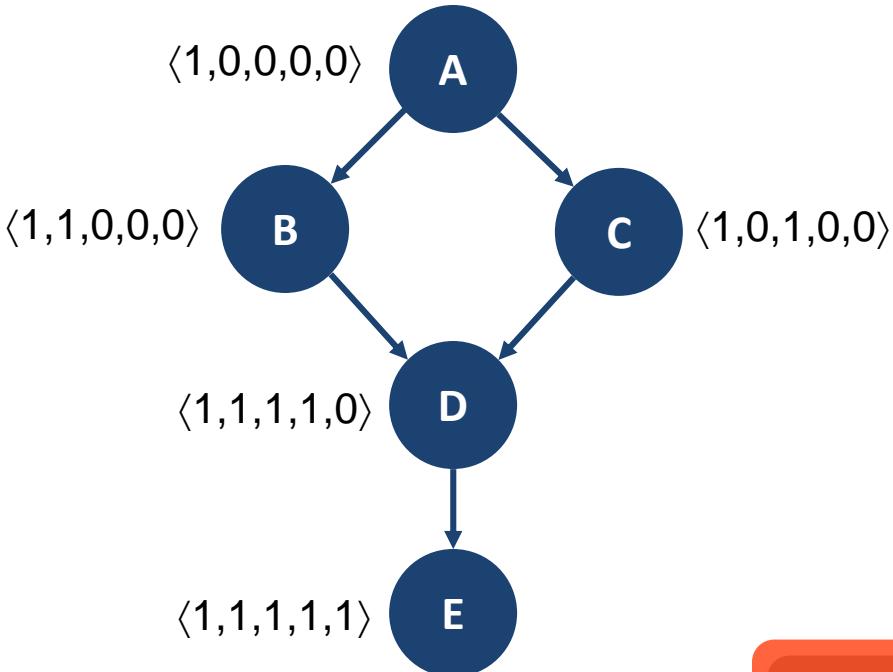
For instance, to compute the vector clock for node D, we may first process the edge from to B \rightarrow D, thereby copying the vector clock $\langle 1,1,0,0,0 \rangle$ from B to D.

Then, when we process the edge C \rightarrow D, we will join (take the max) of the current vector clock at D ($\langle 1,1,0,0,0 \rangle$) and the vector clock coming from C ($\langle 1,0,1,0,0 \rangle$).

That is, for each edge we process, we do $O(N)$ work (as we need to iterate over each entry in the vector clock and the number of such entries is N).

?

$a \preccurlyeq b$ via vector clocks



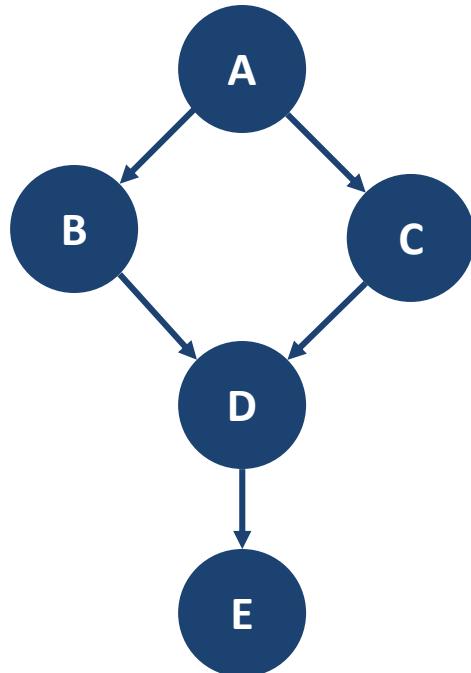
Pre-computation Time: $O(M \cdot N)$
(to obtain all vector clocks)

Query Time: $O(1)$
(for a pair of nodes)

Space: $O(N^2)$

Space Explosion

?
a \leq b via combining chain
decomposition with vector clocks



Key idea: Re-discover threads by partitioning the nodes into chains.

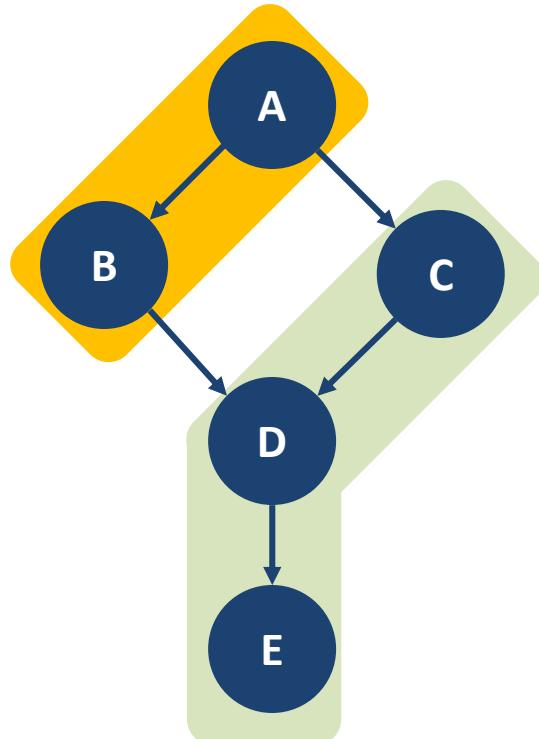
computes a map:

$c \in \text{Nodes} \rightarrow \text{ChainIDs}$

associate a chain with each node

?
a \leq b via combining chain
decomposition with vector clocks

Key idea: Re-discover threads by partitioning the nodes into chains.

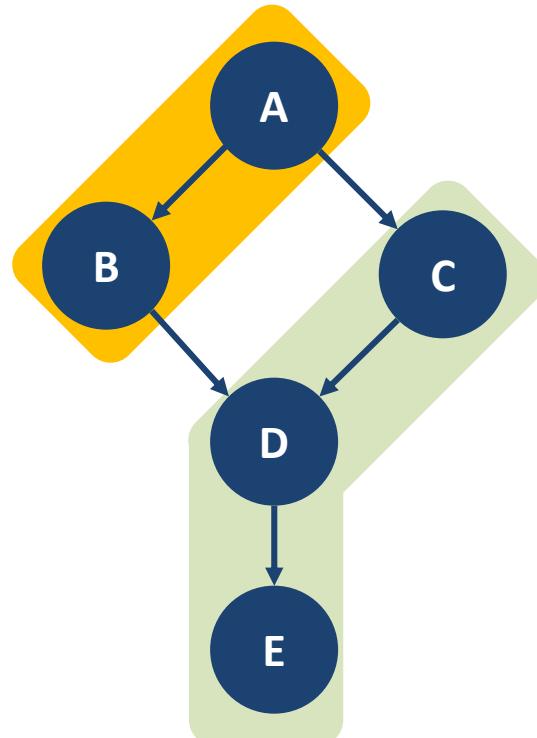


computes a map:

$$c \in \text{Nodes} \rightarrow \text{ChainIDs}$$

associate a chain with each node

$a \leq b$ via combining chain decomposition with vector clocks (optimal version)



$C = \text{number of chains}$

Chain Computation Time: $O(N^3 + C \cdot M)$

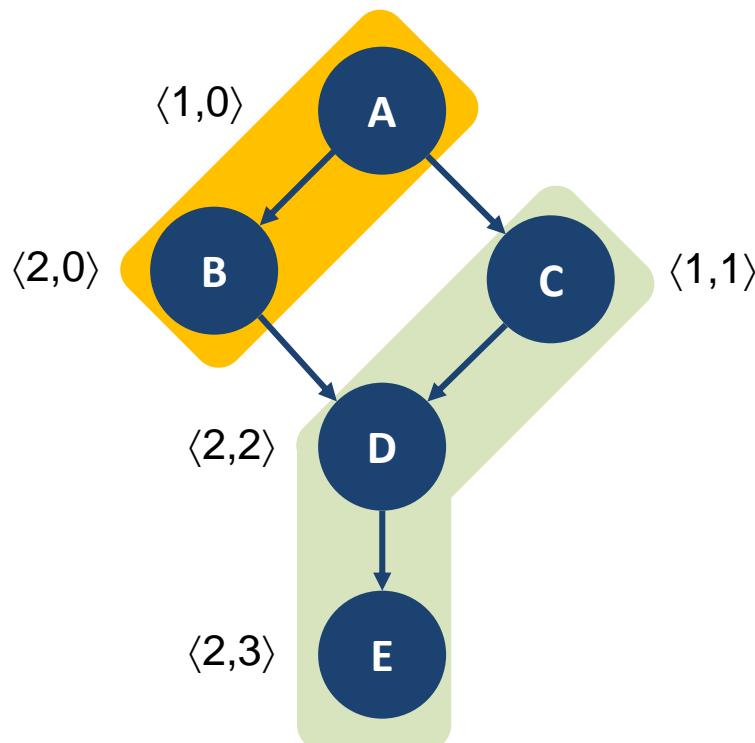
Vector clock computation: $O(C \cdot M)$

Query Time: $O(1)$

Space: $O(C \cdot N)$

Improved

?
a \leq b via combining chain
decomposition with vector clocks
(optimal version)



C = number of chains

Chain Computation Time: $O(N^3 + C \cdot M)$

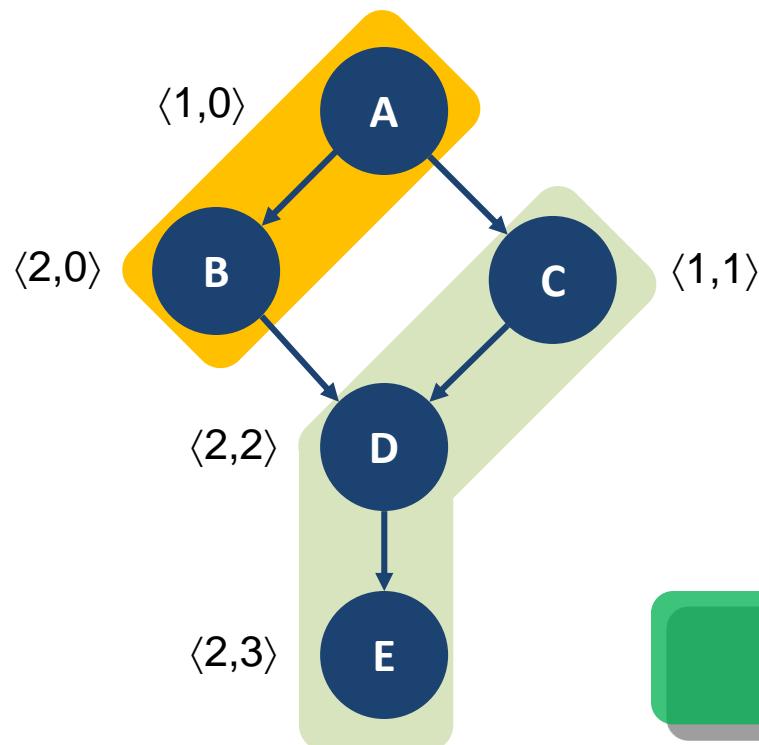
Vector clock computation: $O(C \cdot M)$

Query Time: $O(1)$

Space: $O(C \cdot N)$

Improved

?
a \leq b via combining chain
decomposition with vector clocks
(greedy version)



C = number of chains

Chain Computation Time: $O(C \cdot M)$

Vector clock computation: $O(C \cdot M)$

Query Time: $O(1)$

Space: $O(C \cdot N)$

Improved

Two Challenges Affecting Steps 3 and 4

Synchronization with read/writes

race coverage **eliminates all false races**



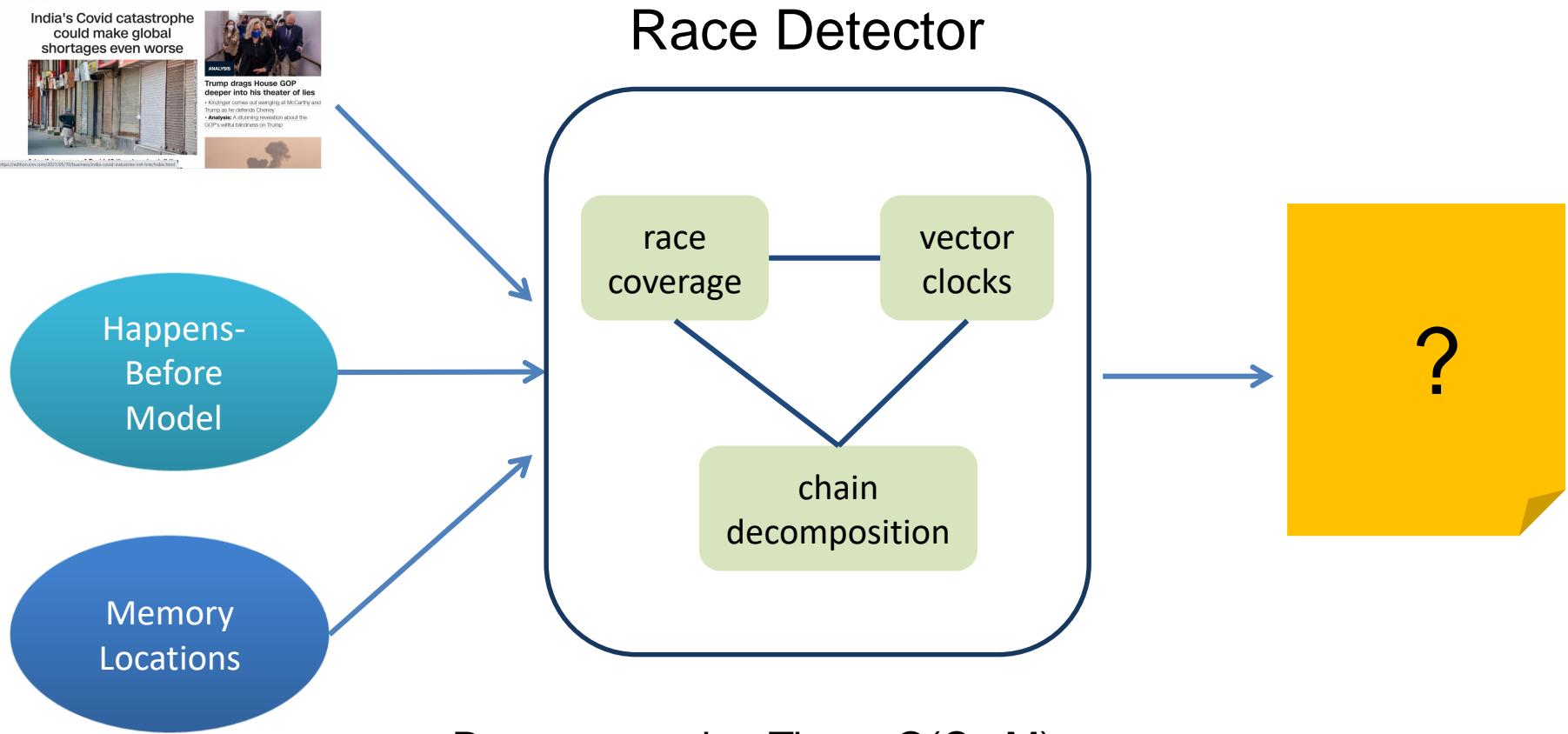
Massive number of event handlers

greedy chain decomposition + vector clocks

space: **$O(C \cdot N)$** where $C \ll N$



Race Detection: Web



Pre-computation Time: $O(C \cdot M)$
Query Time: $O(1)$
Space: $O(C \cdot N)$

Step 5: Implement and Evaluate

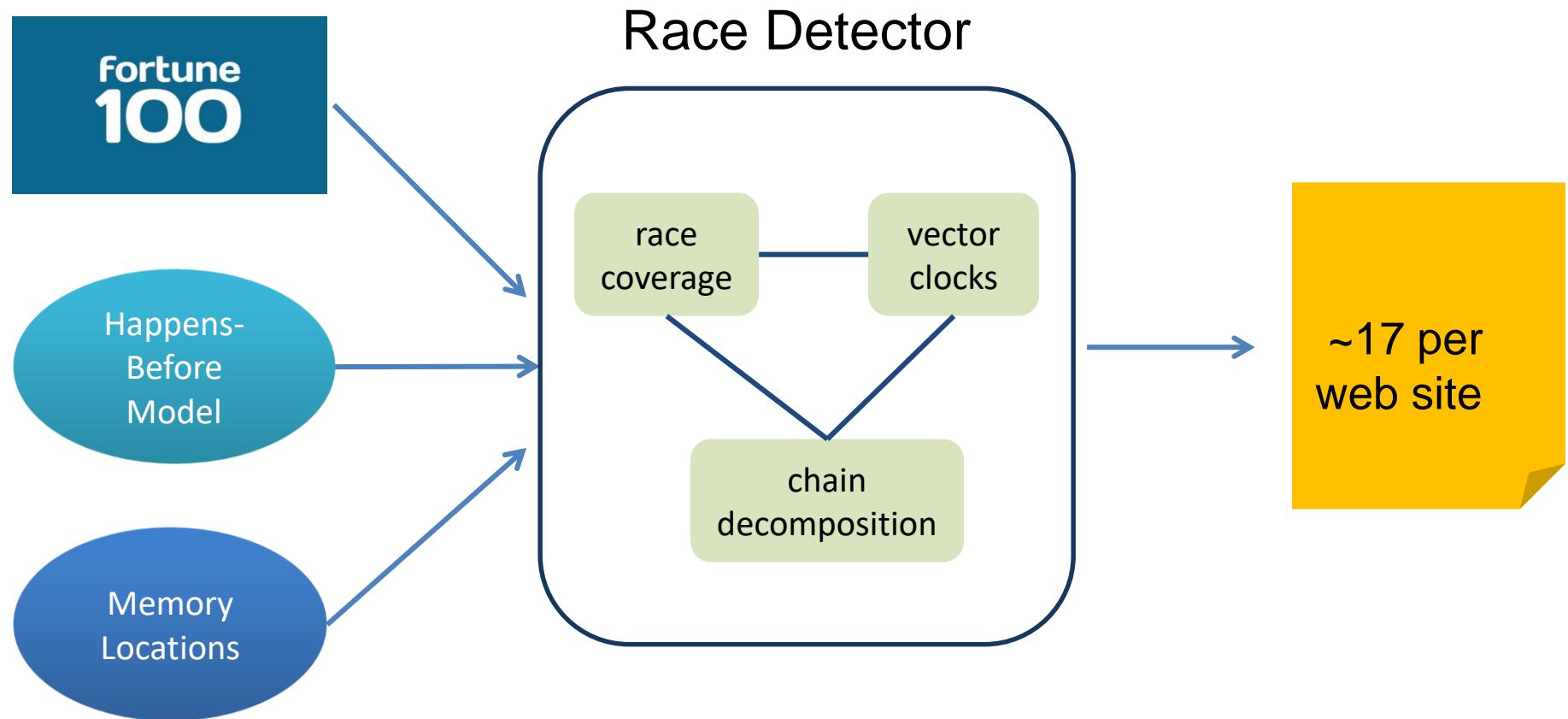
- Based on WebKit Browser
 - Used by Apple's Safari and Google's Chrome
- Check it out: <https://github.com/eth-sri/EventRacer>

Step 5: Implement and Evaluate

We evaluate algorithm performance and precision

Hopefully algorithm is fast and does not report too many false positives on a wide range of applications

Experiments: Fortune 100 web sites



Race coverage: benefit

Metric	Mean # race vars
All	634.6
Only uncovered races	45.3
Filtering methods	
Writing same value	0.75
Only local reads	3.42
Late attachment of event handler	16.7
Lazy initialization	4.3
Commuting operations - className, cookie	4.0
Race with unload	1.1
Remaining after all filters	17.8

Algorithm: Space

Metric	Mean	Max
Number of event actions	5868	114900
Number of chains	175	792
Graph connectivity algorithm		
Vector clocks w/o chain decomposition	544MB	25181MB
Vector clocks + chain decomposition	5MB	171MB

Algorithm: Time

Metric	Mean	Max
Number of event actions	5868	114900
Number of chains	175	792
Graph connectivity algorithm		
Vector clocks w/o chain decomposition	>0.1sec	OOM
Vector clocks + chain decomposition	0.04sec	2.4sec
Breadth-first search	>22sec	TIMEOUT

Modern Dynamic Race Detection: 5 Steps

Step 1: Define Memory locations (on which races can happen)

Usually easy but there can be issues (framework vs. user-code)

Step 2: Define Happens-Before Model (how operations are ordered)

Can be tricky to get right due to subtleties of concurrency

Step 3: Come up with an Algorithm to detect races

Hard to get good asymptotic complexity + correctness

Step 4: Come up with techniques (algorithm, filters) to remove harmless races

Needs to answer what harmless means

Step 5: Implement Algorithm and Evaluate

Important to have low instrumentation overhead

Function Iterates

For a poset (L, \sqsubseteq) , a function $f: L \rightarrow L$, an element $a \in L$, the iterates of the function from a are:

$$f^0(a), f^1(a), f^2(a) \dots$$

$$\text{where } f^{n+1}(a) = f(f^n(a))$$

Note that $f^0(a) = a$

In program analysis, we usually take a to be \perp

A useful fixed point theorem

Given a poset of finite height, a least element \perp , a **monotone** f .

Then the iterates $f^0(\perp), f^1(\perp), f^2(\perp) \dots$ form an increasing sequence which eventually stabilizes from some $n \in \mathbb{N}$, that is:
 $f^n(\perp) = f^{n+1}(\perp)$ and:

$$\text{lfp}^{\sqsubseteq} f = f^n(\perp)$$

This leads to a simple algorithm for computing $\text{lfp}^{\sqsubseteq} f$

Concepts

- Structures: posets, lattices
- Functions: monotone, fixed points
- Approximating functions

Representing $\llbracket P \rrbracket$

Let $\llbracket P \rrbracket$ be the set of reachable states of a program P .

Let function F be (where I is an initial set of states and \rightarrow is the transition relation between states):

$$F(S) = I \cup \{ c' \mid c \in S \wedge c \rightarrow c' \}$$

Then, $\llbracket P \rrbracket$ is a **fixed point** of F : $F(\llbracket P \rrbracket) = \llbracket P \rrbracket$

(in fact, $\llbracket P \rrbracket$ is the **least fixed point** of F)

The Art of Approximation: Static Program Analysis

- Define a function $F^\#$ such that $F^\#$ approximates F . This is typically done manually and can be tricky but is done once and for a programming language.
- Then, use existing theorems which state that the least fixed point of $F^\#$, e.g. some V , approximates the least fixed point of F , e.g. $\llbracket P \rrbracket$
- Finally, automatically compute a fixed point of $F^\#$, that is a V where $F^\#(V) = V$

Approximating a Function

given functions:

$$F: C \rightarrow C$$

$$F^\#: C \rightarrow C$$

what does it mean for $F^\#$ to approximate F ?

$$\forall x \in C : F(x) \sqsubseteq_c F^\#(x)$$

Approximating a Function

What about when:

$$F: C \rightarrow C$$

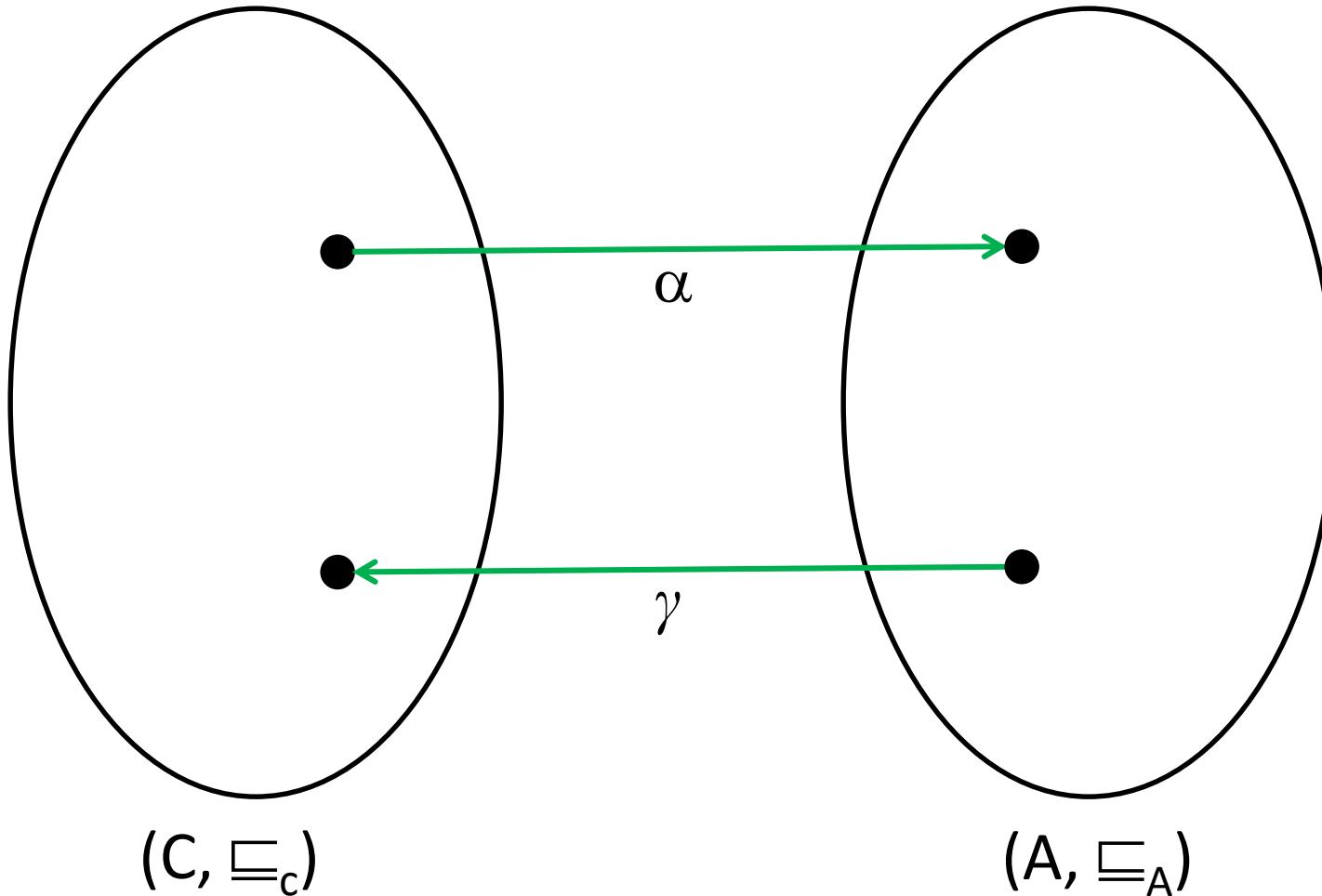
$$F^\#: A \rightarrow A$$

We need to connect the concrete C and the abstract A
We will connect them via two functions α and γ

$\alpha : C \rightarrow A$ is called the **abstraction** function

$\gamma : A \rightarrow C$ is called the **concretization** function

Connecting Concrete with Abstract



Approximating a Function: Definition 1

So we have the 2 functions:

$$F: C \rightarrow C$$

$$F^\#: A \rightarrow A$$

If we know that α and γ form a **Galois Connection**, then we can use the following definition of approximation:

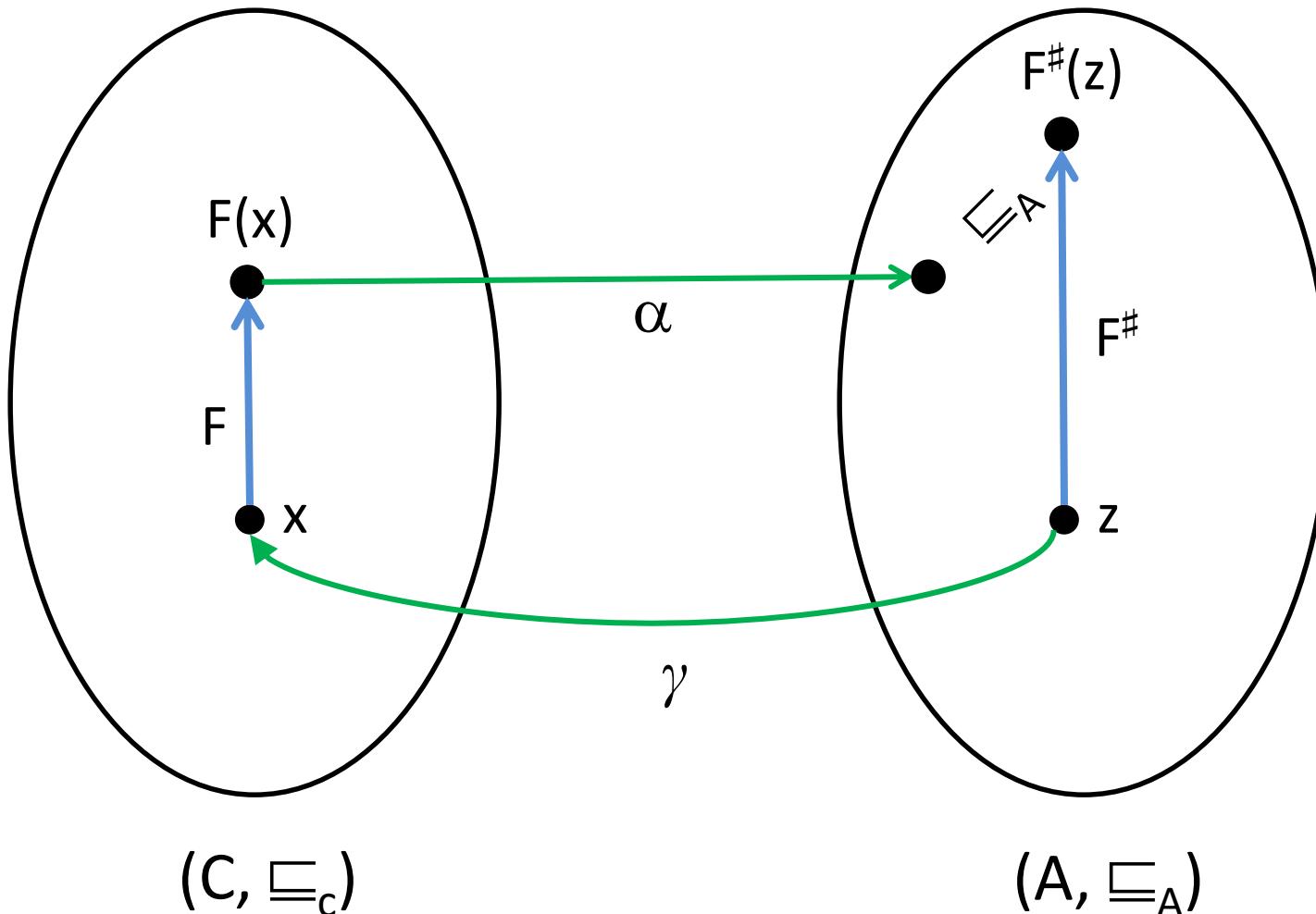
$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$$

For the course, **it is not important** to know what Galois Connections are.

The only point to keep in mind that is that they place some **restrictions** on what α and γ can be.

For instance, among other things, they require α to be monotone.

Visualizing Definition 1



Approximating a Function

what this equation:

$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$$

says is that if we have some function in the abstract that we think **should approximate** the concrete function, then to check that this is indeed true, we need to prove that for any abstract element, concretizing it, applying the concrete function and abstracting back again is **less than** applying the function in the abstract directly.

Least precise approximation

To approximate F , we can always define $F^\#(z) = T$

This solution is always **sound** as: $\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A T$

However, it is not practically useful as it is **too imprecise**

Most precise approximation

What if $F^\#(z) = \alpha(F(\gamma(z)))$? This is the **best abstract function**.

The problem is that we often **cannot implement** such a $F^\#(z)$ algorithmically.

However, we can come up with a $F^\#(z)$ that has the **same behavior** as $\alpha(F(\gamma(z)))$ but a **different implementation**.

Any such $F^\#(z)$ is referred to as the **best transformer**.

Key Theorem I: Least Fixed Point Approximation

If we have:

1. **monotone** functions $F: C \rightarrow C$ and $F^\#: A \rightarrow A$
2. $\alpha: C \rightarrow A$ and $\gamma: A \rightarrow C$ forming a Galois Connection
3. $\forall z \in A: \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$ (that is, $F^\#$ approximates F)

then:

$$\alpha(\text{lfp}(F)) \sqsubseteq_A \text{lfp}(F^\#)$$

This is important as it goes from **local** function approximation to **global** approximation. This is a key theorem in program analysis.

Least Fixed Point Approximation

The 3 premises to the theorem are usually proved **manually**.

Once proved, we can now **automatically** compute a least fixed point in the abstract and be sure that our result is **sound** !

Approximating a Function: Definition 2

So we have the 2 functions:

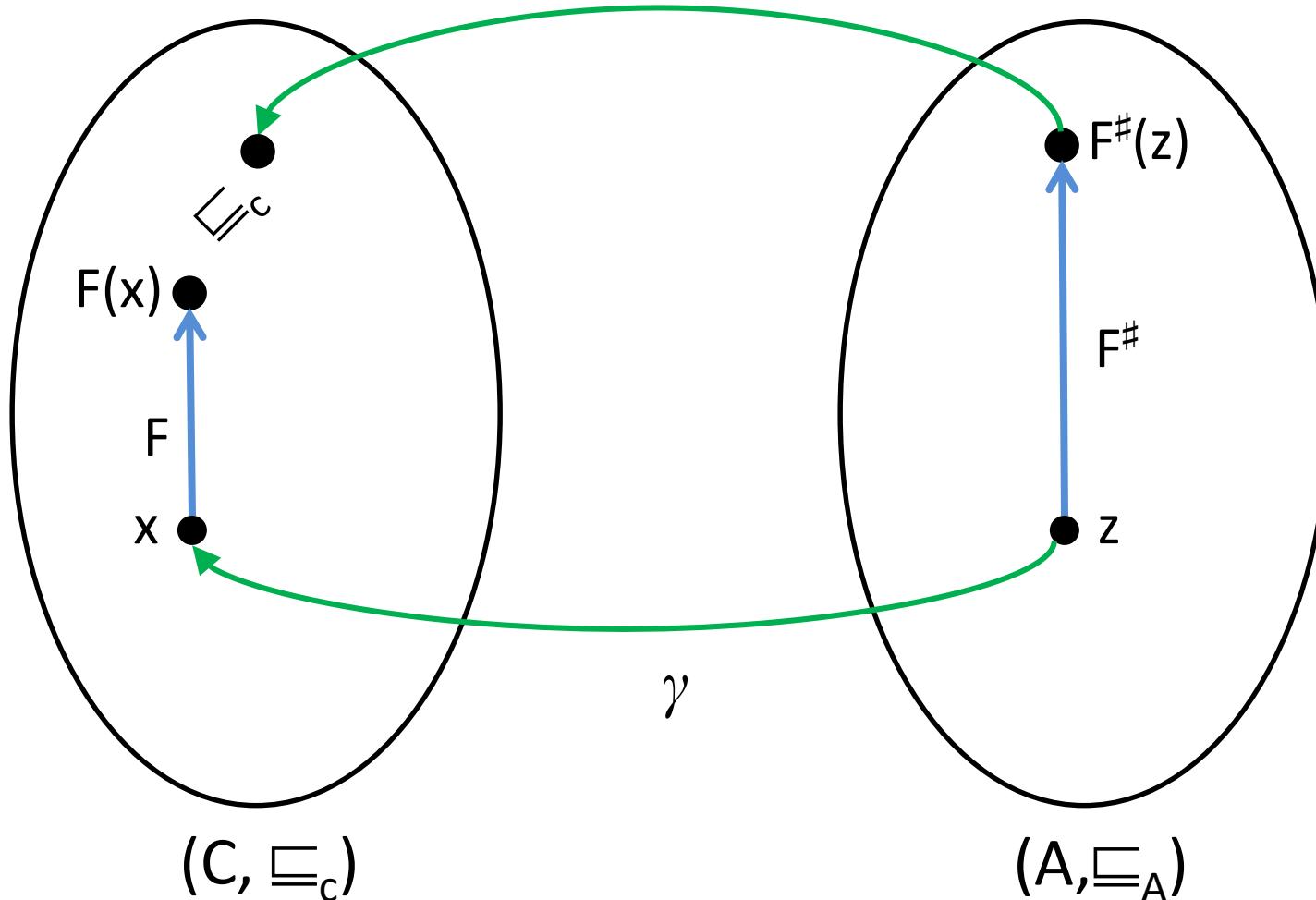
$$F: C \rightarrow C$$

$$F^\#: A \rightarrow A$$

But what if α and γ do not form a **Galois Connection**? For instance, α is not monotone. Then, we can use the following definition of approximation:

$$\forall z \in A : F(\gamma(z)) \sqsubseteq_c \gamma(F^\#(z))$$

Visualizing Definition 2 (concretization-based)



Key Theorem II: Least Fixed Point Approximation

If we have:

1. monotone functions $F: C \rightarrow C$ and $F^\#: A \rightarrow A$
2. $\gamma: A \rightarrow C$ is monotone
3. $\forall z \in A: F(\gamma(z)) \sqsubseteq_c \gamma(F^\#(z))$ (that is, $F^\#$ approximates F)

then:

$$\text{lfp}(F) \sqsubseteq_c \gamma(\text{lfp}(F^\#))$$

This is important as it goes from **local** function approximation to **global** approximation. Another key theorem in program analysis.

So what is $F^\#$ then ?

$F^\#$ is to be defined for the particular abstract domain A that we work with. The domain A can be Sign, Parity, Interval, Octagon, Polyhedra, and so on.

In our setting and commonly, we simply keep a map from every label (program counter) in the program to an abstract element in A

Then $F^\#$ simply updates the mapping from labels to abstract elements.

$F^\#$

$F^\#: (\text{Lab} \rightarrow A) \rightarrow (\text{Lab} \rightarrow A)$

$$F^\#(m)\ell = \begin{cases} T & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell', \text{action}, \ell)} [\![\text{action}]\!](m(\ell')) & \text{otherwise} \end{cases}$$

$[\![\text{action}]\!]: A \rightarrow A$

$[\![\text{action}]\!]$ is the key ingredient here. It captures the effect of a language statement on the abstract domain A . Once we define it, we have $F^\#$

$[\![\text{action}]\!]$ is often referred to as the **abstract transformer**.

what is $(\ell', \text{action}, \ell)$?

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (0 ≤ i) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7: }
```

Actions:

```
(1, x := 5, 2)  
(2, y := 7, 3)  
(3, 0 ≤ i, 4)  
(3, 0 > i, 7)  
(4, y = y + 1, 5)  
(5, i := i - 1, 6)  
(6, goto 3, 3)
```

Multiple (two) actions reach label 3

what is action ?

An **action** can be:

- $b \in BExp$ boolean expression in a conditional here, $a \in AExp$
- $x := a$
- skip

In performing an action, the assignment and the boolean expression of a conditional is **fully evaluated**

$$\begin{array}{ccc} x := y + x & & \text{if } (x > 5) \dots \\ \{x \mapsto 2, y \mapsto 2\} & \rightarrow & \{x \mapsto 4, y \mapsto 2\} \\ & & \{x \mapsto 2, y \mapsto 0\} \end{array}$$

Defining $\llbracket \text{action} \rrbracket$

As we said, $\llbracket \text{action} \rrbracket$ captures the abstract semantics of the language for a particular abstract domain.

In later lectures we will see precise definitions for some actions in the Interval domain. Defining $\llbracket \text{action} \rrbracket$ for complex domains such as say Octagon (see later) can be quite tricky.

Lets just have a brief example now to what it entails even for Intervals...

Example: what is $\llbracket x \leq y \rrbracket$ for Intervals ?

Suppose we have the program:

```
// Here, x is [0, 4] and y is [3, 5]
if (x ≤ y) {
    1: ...
}
```

What does $\llbracket x \leq y \rrbracket$ produce at label 1 ?

That is, what are x and y at label 1 ?

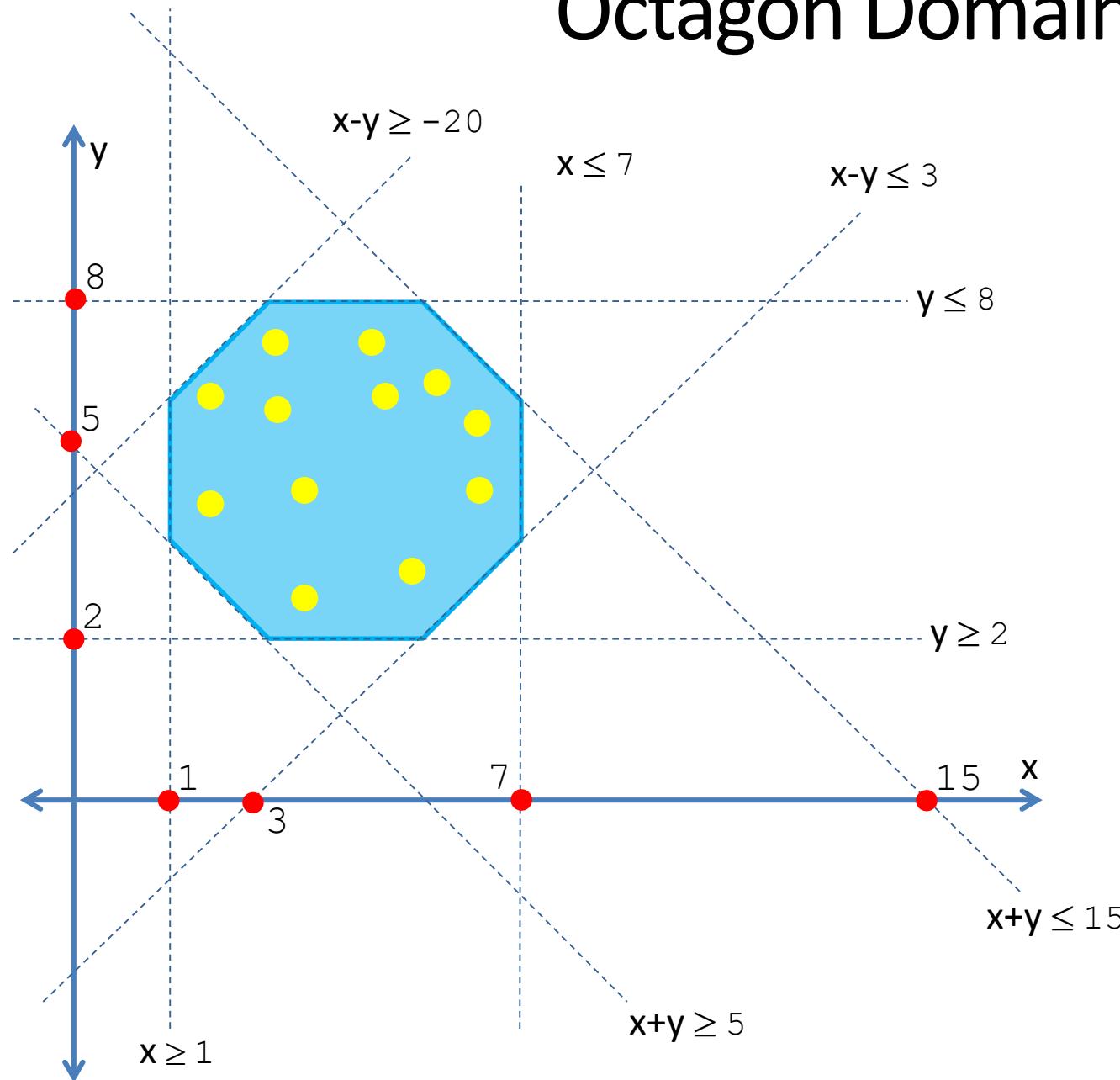
Relational Abstractions

The Interval domain is an example of a **non-relational** domain. It does not explicitly keep the relationship between variables.

In some cases however, it may be necessary to keep this relationship in order to be more precise. Next, we show two examples of abstractions (Octagon and Polyhedra) where the relationship is kept. These domains are called **relational domains**.

In the project, you will use the Polyhedra domain, already implemented as part of the Apron library.

Octagon Domain



constraints are of
the following form:

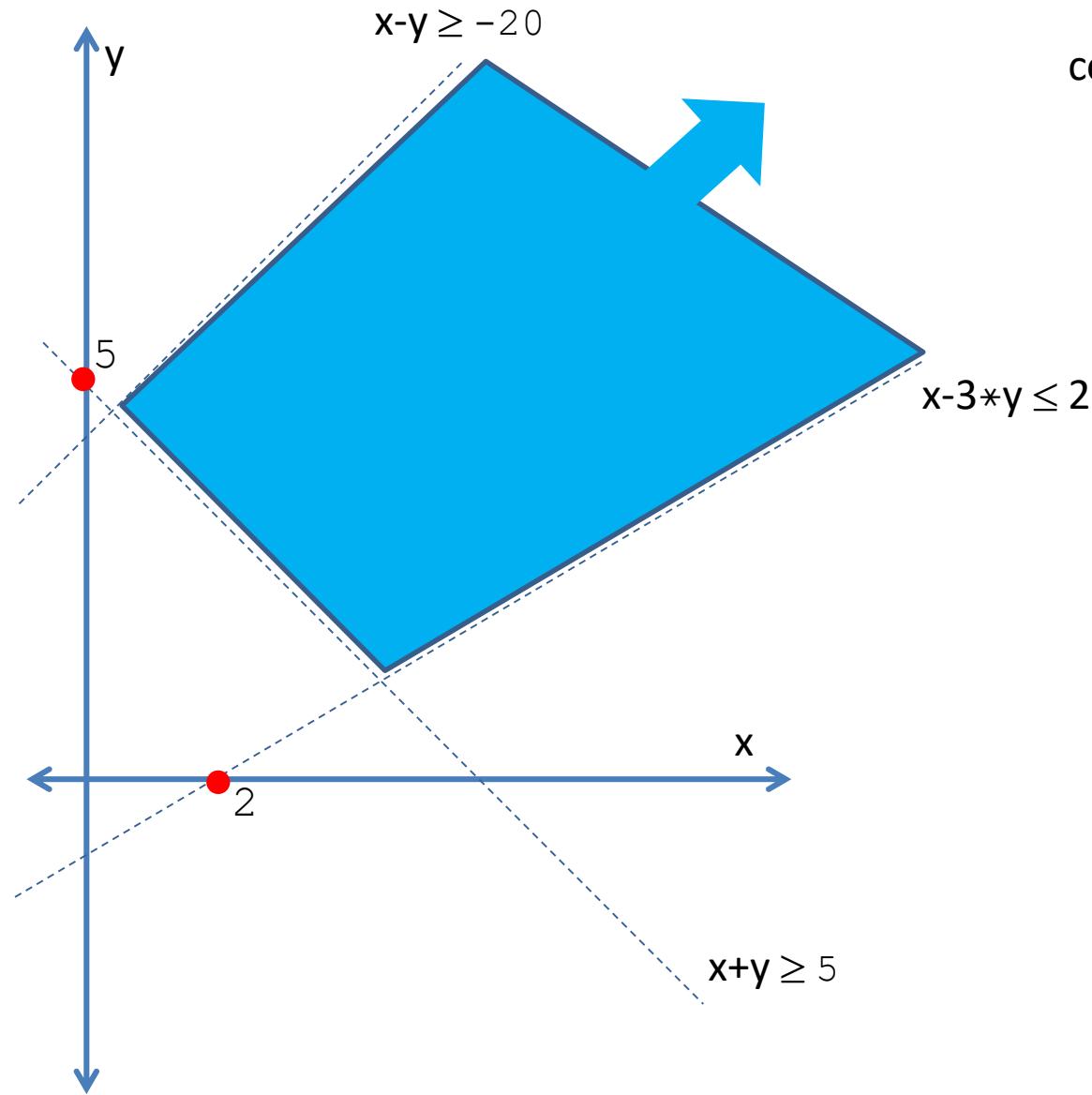
$$\pm x \pm y \leq c$$

The slope is fixed

an abstract state is a map
from labels to conjunction
of constraints

$$\begin{aligned}
 & x - y \leq 3 \quad \wedge \\
 & y \leq 8 \quad \wedge \\
 & y \geq 2 \quad \wedge \\
 & x + y \leq 15 \quad \wedge \\
 & x + y \geq 5 \quad \wedge \\
 & x \geq 1 \quad \wedge \\
 & x - y \geq -20 \quad \wedge \\
 & x \leq 7
 \end{aligned}$$

Polyhedra Domain



constraints are of the following form:

$$c_1x_1 + c_2x_2 \dots + c_nx_n \leq c$$

the slope can vary

an abstract state is again a map
from labels to conjunction of
constraints:

$$\begin{aligned} x - y &\geq -20 \wedge \\ x - 3y &\leq 2 \wedge \\ x + y &\geq 5 \end{aligned}$$

Rigorous Software Engineering

Documentation

Benjamin Bichsel

Slides adapted from previous years
(special thanks to Peter Müller, Felix Friedrich, Hermann Lehner)

Documentation

1. Why should we document?
2. What to document?
3. How to document?

Source Code is Insufficient

- Wide-spread mentality: Code is enough
- Developers require information that is **difficult to extract** from source code
 - Possible result values of a method, and when they occur
 - Possible side effects of methods
 - Consistency conditions of data structures
 - How data structures evolve over time
 - Whether objects are shared among data structures
- Details in the source code may be **overwhelming**

Why Documentation? Essential vs Incidental Properties

- Source code does not express **which properties are stable** during software evolution
 - Which details are essential, and which are incidental?

```
int find( int[ ] array, int v ) {  
    for( int i = 0; i < array.length; i++ )  
        if( array[ i ] == v ) return i;  
    return -1;  
}
```

Can we rely on the result *r* being the smallest index such that $\text{array}[r] == v$?

```
int find( int[ ] array, int v ) {  
    if( 256 <= array.length ) {  
        // perform parallel search and  
        // return first hit  
    } else {  
        // sequential search like before  
    }  
}
```

Why Documentation? Invariants

```
V get( Object key ) {  
    if( key == null )  
        return getForNullKey( );  
    int hash = hash( key.hashCode() );  
    for( Entry<K,V> e = table[ indexFor(hash, table.length) ];  
         e != null; e = e.next ) {  
        Object k;  
        if( e.hash == hash &&  
            ( (k = e.key) == key || key.equals(k) ) )  
            return e.value;  
    }  
    return null;  
}
```

key was
not found

```
HashMap<String,String> m;  
m = SomeLibrary.foo( );  
String s = m.get( "key" );  
// can s be null?
```

Iterate over all entries
for this key's hash
code

Real code:
<http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/tip/src/share/classes/java/util/HashMap.java#l316>

Why Documentation? Invariants (cont'd)

```
V get( Object key ) {  
    if( key == null )  
        return getForNullKey( );  
    int hash = hash( key.hashCode() );  
    for( Entry<K,V> e = table[ indexFor(hash, table.length) ];  
         e != null; e = e.next ) {  
  
        Object k;  
        if( e.hash == hash &&  
            ( (k = e.key) == key || key.equals(k) ) )  
            return e.value;  
    }  
    return null;  
}
```

Is [hash, **null**] a valid entry?
Need to find and check all ways of
entering information into table

```
HashMap<String,String> m;  
m = SomeLibrary.foo( );  
if( m.containsKey( "key" ) ) {  
    String s = m.get( "key" );  
    // can s be null?  
    ...  
}
```

Why Documentation? Overriding methods

- Initialize the fields of an object when the object is created or when the fields are accessed for the first time?

```
class ImageFile {  
    String file;  
    Image image;  
  
    ImageFile( String f ) {  
        file = f;  
        // load the image  
    }  
  
    Image getImage( ) {  
        return image;  
    }  
}
```

```
class ImageFile {  
    String file;  
    Image image;  
  
    ImageFile( String f ) {  
        file = f;  
    }  
  
    Image getImage( ) {  
        if( image == null ) {  
            // load the image  
        }  
        return image;  
    }  
}
```

Why Documentation? Overriding methods

```
class ImageFile {  
    String file;  
    Image image;  
  
    ...  
  
    boolean equals( Object o ) {  
        if( o.getClass( ) != getClass( ) ) return false;  
        return file.equals( ( (ImageFile) o ).file );  
    }  
  
    int hashCode( ) {  
        if( image == null )  
            return file.hashCode( );  
        else  
            return image.hashCode( ) + file.hashCode( );  
    }  
}
```

Is this a suitable implementation of hashCode?

Why Documentation? Overriding methods (cont'd)

```
class ImageFile {  
    String file;  
    Image image;  
  
    ...  
  
    boolean equals( Object o ) {  
        if( o.getClass( ) != getClass( ) ) return false;  
        return file.equals( ( (ImageFile) o ).file );  
    }  
  
    int hashCode( ) {  
        if( image == null )  
            return file.hashCode( );  
        else  
            return image.hashCode( ) + file.hashCode( );  
    }  
}
```

```
void demo(  
    HashMap<ImageFile, String> m,  
    ImageFile f ) {  
    m.put( f, "Hello" );  
    Image i = f.getImage( );  
    int l = m.get( f ).length( );  
    ...  
}
```

With lazy initialization,
getter may change hash
code

Need to determine
whether image may
be modified

Why Documentation? Overriding methods (cont'd)

```
class ImageFile {  
    String file;  
    Image image;  
  
    ...  
  
    boolean equals( Object o ) {  
        if( o.getClass( ) != getClass( ) ) return false;  
        return file.equals( ( (ImageFile) o ).file );  
    }  
  
    int hashCode( ) {  
        return getImage( ).hashCode( ) +  
            file.hashCode( );  
    }  
}
```

```
void demo(  
    HashMap<ImageFile, String> m,  
    ImageFile f ) {  
    m.put( f, "Hello" );  
    Image i = f.getImage( );  
    int l = m.get( f ).length( );  
    ...  
}
```

Hash code is not affected by lazy initialization

Need to determine whether the result of getImage may be modified

Why Documentation? Effect of Aliasing

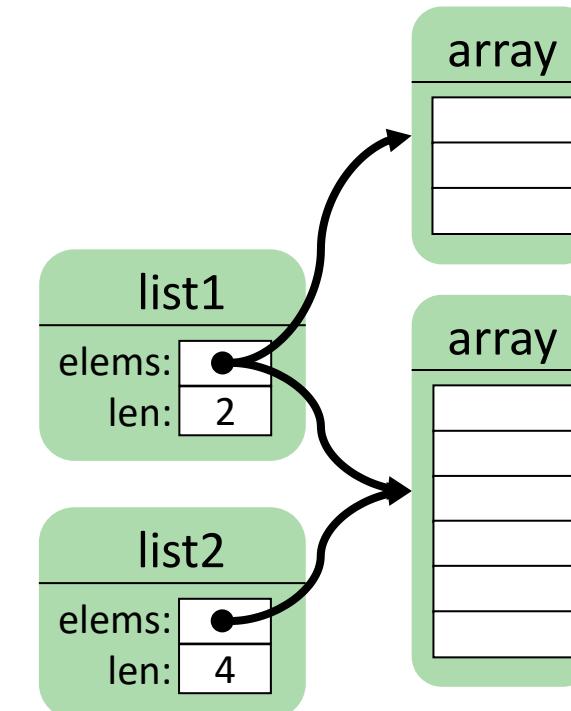
```
class List<E> {  
    E[ ] elems;  
    int len;  
  
    void set( int index, E e ) {  
        elems[ index ] = e;  
    }  
  
    E get( int index ) {  
        return elems[ index ];  
    }  
  
    List<E> clone( ) {  
        return new List<E>( rep, len );  
    }  
}
```

```
class SmallList<E> extends List<E> {  
    void shrink( ) {  
        // reduce array size if the array  
        // is not fully used  
    }  
}
```

Is this an optimization or does it change the behavior?

Why Documentation? Effect of Aliasing (cont'd)

```
class SmallList<E> extends List<E> {  
    void shrink( ) {  
        int l = elems.length / 2;  
        if( len <= l ) {  
            E[ ] tmp = new E[ l ];  
            System.arraycopy( elems, 0, tmp, 0, len );  
            elems = tmp;  
        }  
    }  
}
```

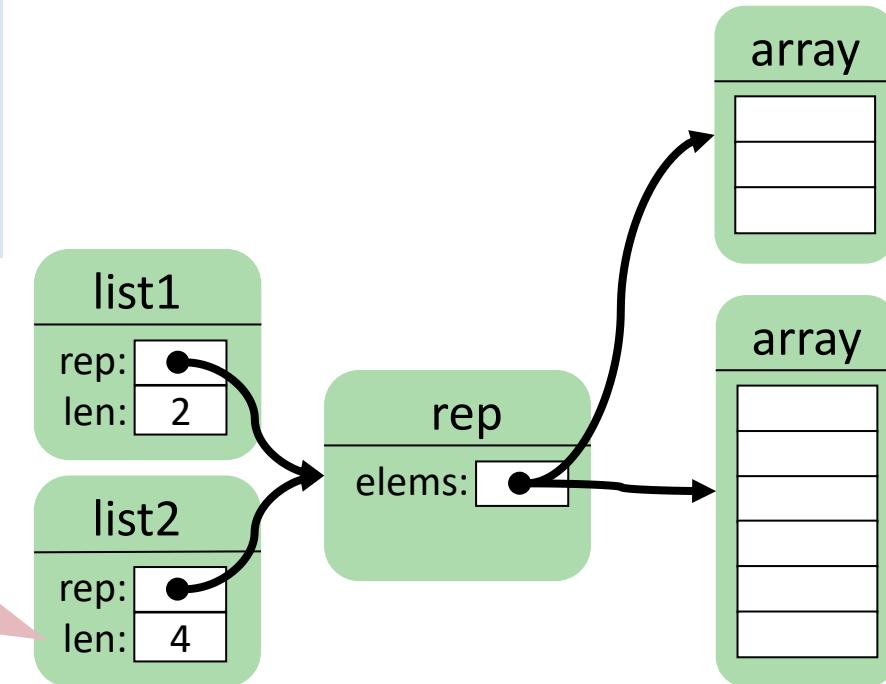


Extending List: Reference Counting

```
class SmallList<E> extends List<E> {  
    void shrink( ) {  
        int l = rep.elems.length / 2;  
        if( len <= l ) {  
            E[ ] tmp = new E[ l ];  
            System.arraycopy( rep.elems, 0, tmp, 0, len );  
            rep.elems = tmp;  
        }  
    }  
}
```

May introduce
out-of-bounds
exception for
list2

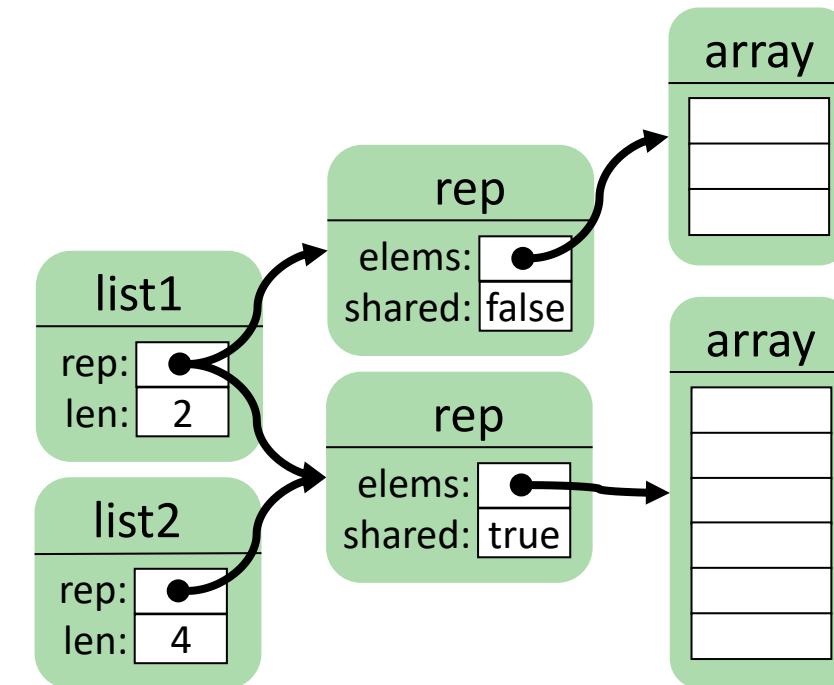
Invariant: Array is
not shared



Extending List: Reference Counting (cont'd)

```
class SmallList<E> extends List<E> {  
    void shrink( ) {  
        int l = rep.elems.length / 2;  
        if( len <= l ) {  
            E[ ] tmp = new E[ l ];  
            System.arraycopy( rep.elems, 0, tmp, 0, len );  
            if( rep.shared ) rep = new ListRep<E>();  
            rep.elems = tmp;  
        }  
    }  
}
```

Ensure that a shared array is never modified



Documentation

1. Why should we document?
2. What to document?
3. How to document?

Documentation

- Essential properties must be documented explicitly

For clients:

How to **use** the code?

Document the **interface**

For implementors:

How does the code **work**?

Document the **implementation**

- Documentation should focus on **what** the essential properties are, **not how** they are achieved

- “Whenever a ListRep object’s shared-field is false, it is used as representation of at most one List object”

Rather than

- “When creating a new List object with an existing ListRep object, the shared-field is set to true”

Interface Documentation

For clients:
How to **use** the code?
Document the **interface**

- The client interface of a class consists of
 - Constructors
 - Methods
 - Public fields
 - Supertypes
- We focus on methods here
 - Constructors are analogous
 - Fields can be viewed as getter and setter methods

Method Documentation: Call

- Clients need to know **how to call** a method correctly

```
class InputStreamReader {  
    int read( char cbuf[ ], int offset, int len ) throws IOException  
    ...  
}
```

- Parameter values
 - `cbuf` is non-null
 - `offset` is non-negative
 - `len` is non-negative
 - `offset + len` is at most `cbuf.length`
- Input state
 - The receiver is open

Method Documentation: Results

- Clients need to know how what a method **returns**

```
class InputStreamReader {  
    int read( char cbuf[ ], int offset, int len ) throws IOException  
    ...  
}
```

- **Result values**
 - The method returns -1 if the end of the stream has been reached before any characters are read
 - Otherwise, the result is between 0 and len, and indicates how many characters have been read from the stream

Method Documentation: Effects

- Clients need to know how a method **affects the state**
- **Heap effects**
 - “result” characters have been consumed from the stream and stored in cbuf, from offset onwards
 - If the result is -1, no characters are consumed and cbuf is unchanged
- **Other effects**
 - The method throws an IOException if the stream is closed or an I/O error occurs
 - It does not block

Method Documentation: Another Example

```
class List<E> {  
    ...  
    List<E> clone( ) {  
        rep.shared = true;  
        return new List<E>( rep, len );  
    }  
}
```

- The method returns a **shallow copy** of its receiver
 - The list is copied, but not its contents
- The result is a **fresh object**
- The method requires **constant time and space**

Interface Documentation: Global Properties

- Some implementations have properties that affect all methods
 - Properties of the data structure, that is, guarantees that are maintained by all methods together
- **Consistency:** properties of states
 - Example: a list is sorted
 - Client-visible **invariants**

```
int a = list.first( );
int b = list.itemAt( 1 );
int c = list.last( );
// a <= b <= c
```

Interface Documentation: Global Properties (cont'd)

- **Evolution:** properties of sequences of states

- Example: a list is immutable
 - **Invariants** on sequences of states

```
int a = list.first( );
// arbitrary operations
int b = list.first( );
// a == b
```

- **Abbreviations:** requirements or guarantees for all methods

- Example: a list is not thread-safe

Clients must ensure they have exclusive access to the list, for instance, because the execution is sequential, the list is thread-local, or they have acquired a lock

Implementation Documentation

For implementors:
How does the code work?
Document the implementation

- Method documentation is similar to interfaces
- Data structure documentation is more prominent
 - Properties of fields, internal sharing, etc.
 - Implementation invariants
- Documentation of the algorithms inside the code
 - For instance, justification of assumptions

Implementation Documentation: Example

```
class ListRep<E> {  
    E[ ] elems;  
    boolean shared;  
    ...  
}
```

```
class List<E> {  
    ListRep<E> rep;  
    int len;  
    ...  
}
```

1. `elems` is non-null
2. `elems` is pointed to by only one object
3. When the `shared`-field is true then `shared`, `elems`, and all elements of `elems` are immutable
4. When the `shared`-field is false, the `ListRep` object is used as representation of at most one `List` object
5. `rep` is pointed to only by `List` objects
6. `rep` is non-null
7. $0 \leq \text{len} \leq \text{rep.elems.length}$

Implementation Documentation: Example (cont'd)

```
/* This method reduces the memory footprint of the list if it uses at most
 * 50% of its capacity, and does nothing otherwise. It optimizes the
 * memory consumption if the underlying array is not shared or if it is
 * shared but will be copied several times after shrinking. The list content
 * remains unchanged. */
void shrink( ) {
    // perform array copy only if array size can be reduced by 50%
    if( len <= rep.elems.length / 2 ) {
        E[ ] tmp;
        tmp = new E[ rep.elems.length / 2 ];
        System.arraycopy( rep.elems, 0, tmp, 0, len );
        if( rep.shared ) rep = new ListRep<E>();
        // rep is not shared, so we may update its array
        rep.elems = tmp;
    }
}
```

Implementation Documentation: Example (cont'd)

```
void shrink( ) {  
    int l = rep.elems.length / 2;  
    if( len <= l ) {  
        E[ ] tmp;  
        tmp = new E[ l ]; ⑦  
        System.arraycopy( ... );  
        if( rep.shared )  
            rep = new ListRep<E>();  
        ⑥  
        rep.elems = tmp;  
    } ① ② ④→7  
}
```

1. elems is non-null
2. elems is pointed to by only one object
3. When the shared-field is true then shared, elems, and all elements of elems are immutable
4. When the shared-field is false, the ListRep object is used as representation of at most one List object
5. rep is pointed to only by List objects
6. rep is non-null
7. $0 \leq len \leq \text{rep.elems.length}$

Implementation Documentation: Example (cont'd)

```
void shrink( ) {  
    int l = rep.elems.length / 2;  
    if( len <= l ) {  
        E[ ] tmp;  
        tmp = new E[ l ];  
        System.arraycopy( ... );  
        rep.elems = tmp;  
    }  
}
```

3 7

1. elems is non-null
2. elems is pointed to by only one object
3. When the shared-field is true then shared, elems, and all elements of elems are immutable
4. When the shared-field is false, the ListRep object is used as representation of at most one List object
5. rep is pointed to only by List objects
6. rep is non-null
7. $0 \leq len \leq \text{rep.elems.length}$

Documentation: Key Properties

■ Methods and constructors

- Arguments and input state
- Results and output state
- Effects

For clients:

How to **use** the code?

Document the **interface**

■ Data structures

- Value and structural invariants
- One-state and temporal invariants

For implementors:

How does the code **work**?

Document the **implementation**

■ Algorithms

- Behavior of code snippets (analogous to methods)
- Explanation of control flow
- Justification of assumptions

Documentation

1. Why should we document?
 2. What to document?
 3. How to document?
-

Comments

- Simple, flexible way of documenting interfaces and implementations
- Tool support is limited
 - HTML generation
 - Not present in executable code
 - Relies on conventions (cp. Python)
- Javadoc
 - Textual descriptions
 - Tags

```
/**  
 * Returns the value to which the  
 * specified key is mapped, or  
 * {@code null} if this map contains no  
 * mapping for the key.  
 *  
 * @param key the key whose associated  
 *             value is to be returned  
 * @return the value to which the  
 *         specified key is mapped, or  
 *         {@code null} if this map contains  
 *         no mapping for the key  
 * @throws NullPointerException if the  
 *             specified key is null and this map  
 *             does not permit null keys  
 */  
V get( Object key );
```

Types and Modifiers

- Types are a powerful documentation tool

```
HashMap<String, String> m;  
m = SomeLibrary.foo();  
String s = m.get("key");
```

- Modifiers can express some specific semantic properties

```
from SomeLibrary import foo  
m = foo()  
s = m[ 'key' ]
```

Python

- Tool support
 - Static checking
 - Run-time checking
 - Auto-completion

```
class HashMap<K,V> ... {  
    final float loadFactor;  
    ...  
}
```

Effect Systems

- Effect systems are extensions of type systems that describe computational effects
 - Read and write effects
 - Allocation and de-allocation
 - Locking
 - Exceptions
- Tool support
 - Static checking
- Trade-off between overhead and benefit

```
class InputStreamReader {  
    int read( ) throws IOException  
    ...  
}
```

```
try {  
    int i = isr.read( );  
} catch( IOException e ) {  
    ...  
}
```

Metadata

- Annotations allow one to attach additional syntactic and semantic information to declarations
- Tool support
 - Type checking of annotations
 - Static processing through compiler plug-ins
 - Dynamic processing

```
@interface NonNull{ }
```

```
@NonNull Image getImage( ) {  
    if( image == null ) {  
        // load the image  
    }  
    return image;  
}
```

Assertions

- Assertions specify semantic properties of implementations
 - Boolean conditions that need to hold

- Tool support
 - Run-time checking
 - Static checking
 - Test case generation

```
void shrink( ) {  
    int l = rep.elems.length / 2;  
    if( len <= l ) {  
        E[ ] tmp = new E[ l ];  
        System.arraycopy( ... );  
        if( rep.shared )  
            rep = new ListRep<E>();  
        assert !rep.shared;  
        rep.elems = tmp;  
    }  
}
```

Contracts

- Contracts are stylized assertions for the documentation of interfaces and implementations
 - Method pre and postconditions
 - Invariants
- Tool support
 - Run-time checking
 - Static checking
 - Test case generation

```
class ImageFile {  
    String file;  
    invariant file != null;  
  
    Image image;  
    invariant old( image ) != null ==>  
              old( image ) == image;  
  
    ImageFile( String f )  
        requires f != null;  
        { file = f; }  
  
    Image getImage( )  
        ensures result != null;  
        {  
            if( image == null ) { // load the image }  
            return image;  
        }  
    }
```

Documentation: Techniques

- Trade-off between overhead, expressiveness, precision, and benefit
 - Formal techniques require more overhead, but enable better tool support
 - In practice, a mix of the different techniques is useful
- It is better to **simplify** than to describe complexity!
 - *If you have a procedure with ten parameters, you probably missed some.*
[Alan J. Perlis]

5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.3.1 Partition Testing

5.3.2 Selecting Representative Values

5.3.3 Combinatorial Testing

5.4 Structural Testing

Combinatorial Testing

- Combining equivalence classes and boundary testing leads to many values for each input
 - Twelve values for month and 17 values for year in the Leap Year example
- Testing all possible combinations leads to a combinatorial explosion ($12 \times 17 = 204$ tests)
- Reduce test cases to make effort feasible
 - Semantic constraints
 - Combinatorial selection
 - Random selection

Eliminating Combinations

- Inspect test cases for unnecessary combinations
 - Especially for invalid values
 - Use problem domain knowledge

month	
Month with 28 or 29 days	month = 2
Months with 30 days	month $\in \{4, 6, 9, 11\}$
Months with 31 days	month $\in \{1, 3, 5, 7, 8, 10, 12\}$
Invalid	month < 1 or month > 12

Test all combinations with year

Behavior is independent of year

- Reduces test cases from 204 to $17 + 4 + 3 + 4 = 28$

Semantic Constraints: Discussion

- Semantic constraints potentially reduce the number of test cases
 - They also help increasing the coverage
- But too many combinations remain
 - Especially when there are many input values, for instance, for the fields of objects

Influence of Variable Interactions

- Empirical evidence suggests that most errors do not depend on the interaction of many variables
- Interactions of **two or three variables** trigger most errors

Pairwise-Combinations Testing

- Instead of testing all possible combinations of all inputs, focus on all possible combinations of each pair of inputs
 - Pairwise-combinations testing is identical to combinatorial testing for two or less inputs
- Example: Consider a method with four boolean parameters
 - Combinatorial testing requires $2^4 = 16$ test cases
 - Pairwise-combinations testing requires 5 test cases:
TTTT, TFFF, FTFF, FFTF, FFFT
- Can be generalized to k-tuples (k-way testing)

Pairwise-Combinations Testing: Complexity

- For n parameters with d values per parameter, the number of test cases grows logarithmically in n and quadratic in d
 - Handles larger number of parameters, for instance, fields of objects
 - The number d can be influenced by the tester
- Result holds for large n and d , and for all k in k -way testing

Pairwise-Combinations Testing: Discussion

- Pairwise-combinations testing (or k-way testing) reduces the number of test cases significantly while detecting most errors
- Pairwise-combinations testing is especially important when many system configurations need to be tested
 - Hardware, operating system, database, application server, etc.
- Should be combined with other approaches to detect errors that are triggered by more complex interactions among parameters

5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.4 Structural Testing

Motivating Example

Given a non-null array of integers, sort the array in-place in ascending order

```
public void sort( int[ ] a ) {  
    if( a == null || a.length < 2 ) // array is trivially sorted  
        return;  
    // check if array is already sorted  
    int i;  
    for( i = 0; i < a.length - 1; i++ )  
        if( a[ i ] < a[ i + 1 ] )  
            break;  
    if( i >= a.length - 1 ) // array is already sorted  
        return;  
    // use quicksort to sort the array in ascending order  
}
```

Error: check for sortedness should use '>'

Motivating Example: Functional Testing

Given a non-null array of integers, sort the array in-place in ascending order

	a
Valid	any non-null array
Invalid	null

Choose for instance
{}, { 1 }, { 1, 2, 3 }

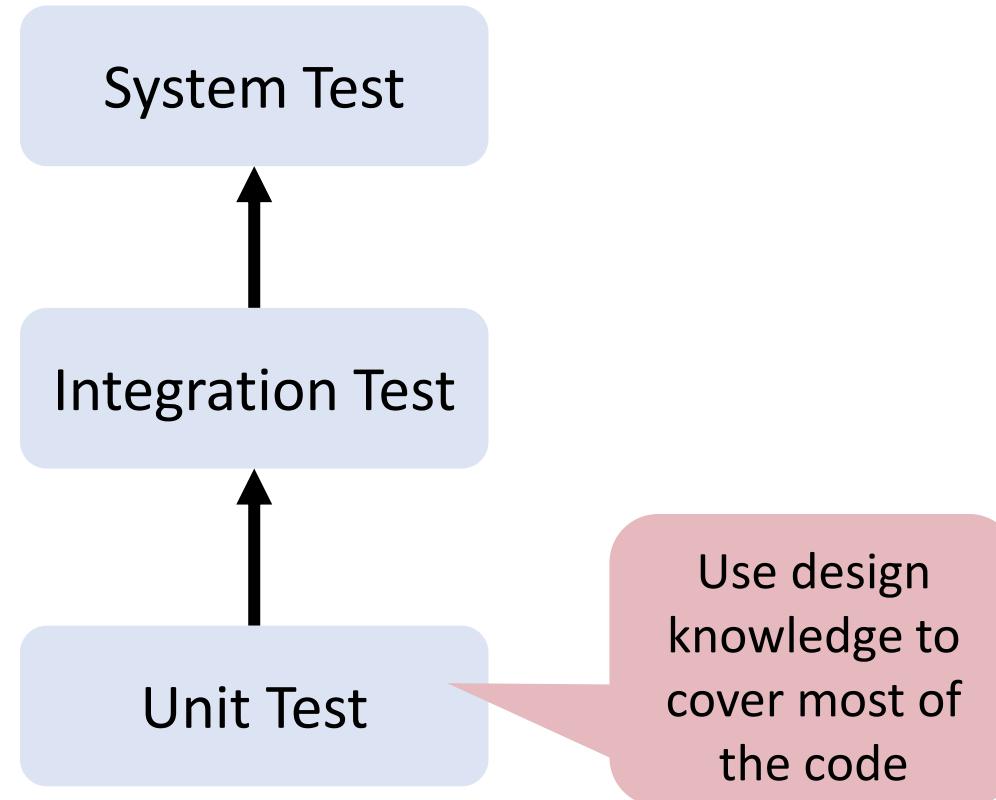
- The requirements give no clue that one should test with an array that is sorted in descending order

Motivating Example: Discussion

- Detailed design and coding introduce many behaviors that are not present in the requirements
 - Choice of data structures
 - Choice of algorithms
 - Optimizations such as caches
- Functional testing generally does not thoroughly exercise these behaviors
 - No data structure specific test cases, e.g., rotation of AVL-tree
 - No test cases for optimizations, e.g., cache misses

Applications of Structural Testing

- White-box test a unit to cover a large portion of its code



5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.4 Structural Testing

Basic Blocks

- A **basic block** is a sequence of statements such that the code in a basic block:
 - has **one entry point**: no code within it is the destination of a jump instruction anywhere in the program
 - has **one exit point**: only the last instruction causes the program to execute code in a different basic block
- Whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order

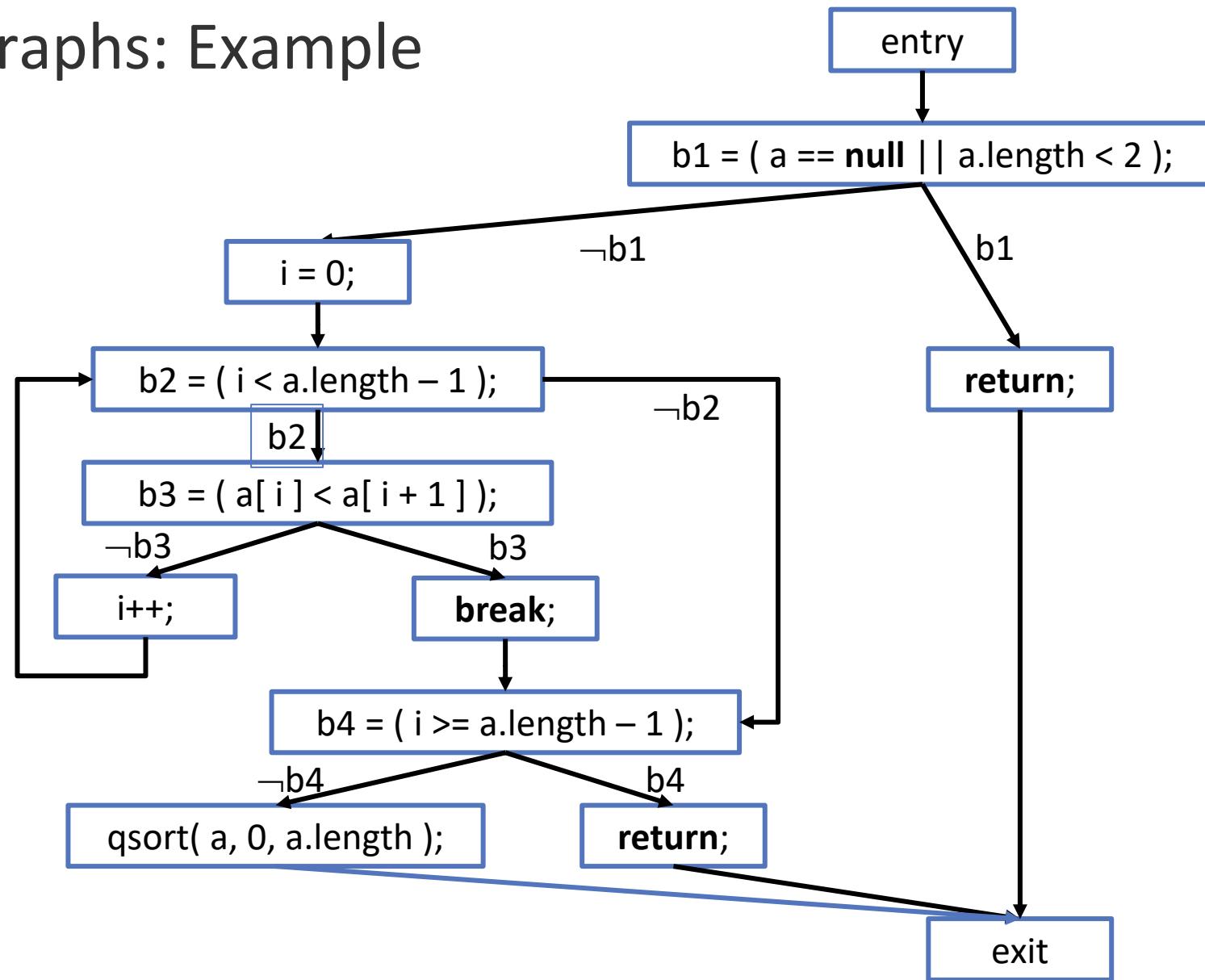
Basic Blocks: Example

```
public void sort( int[ ] a ) {  
    if( a == null || a.length < 2 )  
        return;  
  
    int i;  
  
    for( i = 0; i < a.length - 1; i++ ) {  
        if( a[ i ] < a[ i + 1 ] )  
            break;  
    }  
    if( i >= a.length - 1 )  
        return;  
    qsort( a, 0, a.length );  
}
```

Intraprocedural Control Flow Graphs

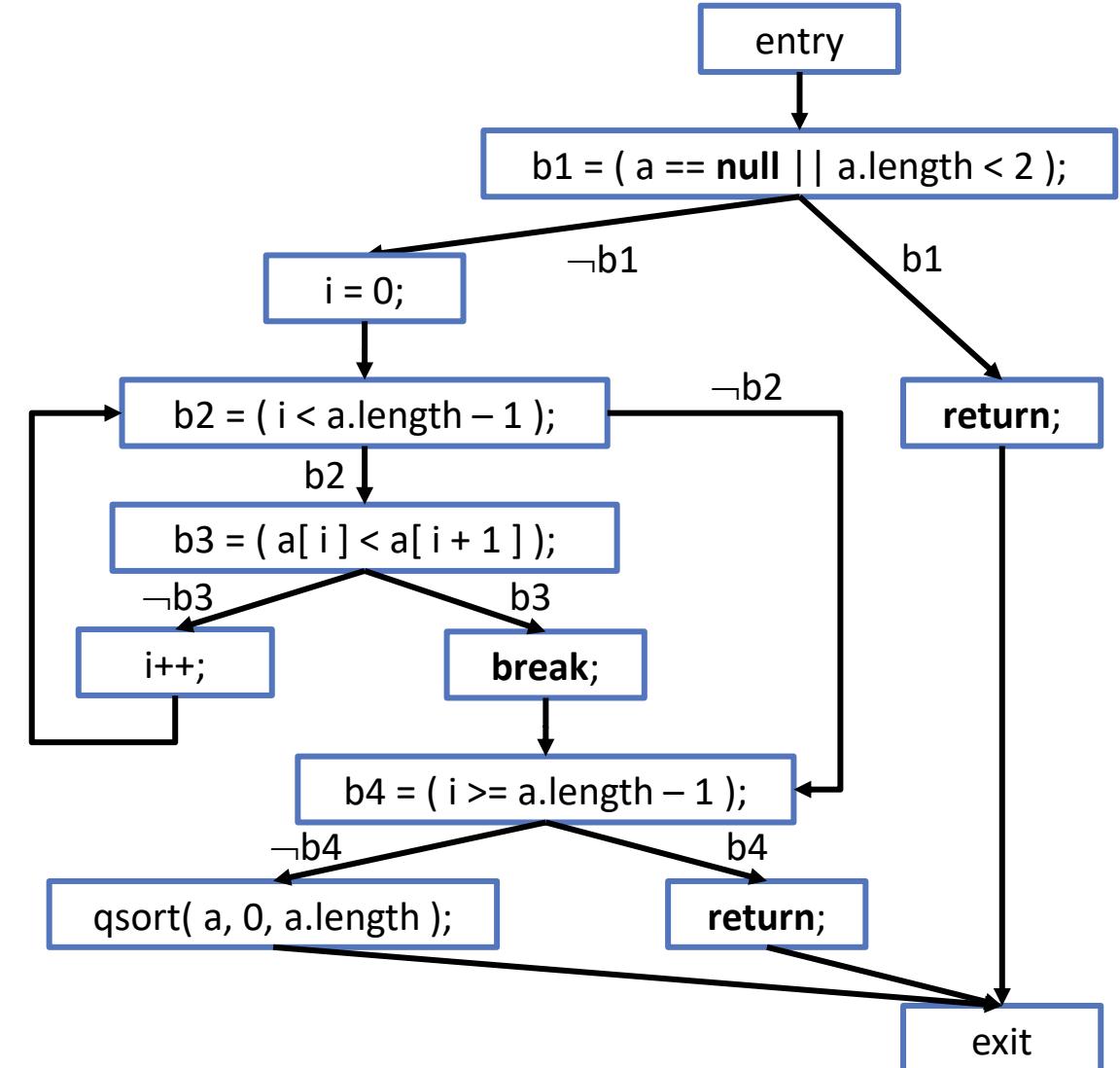
- An **intraprocedural control flow graph (CFG)** of a procedure p is a graph (N, E) where:
 - N is the set of basic blocks in p plus designated entry and exit blocks
 - E contains
 - an edge from a to b with condition c iff the execution of basic block a is succeeded by the execution of basic block b if condition c holds
 - an edge $(\text{entry}, a, \text{true})$ if a is the first basic block of p
 - edges $(b, \text{exit}, \text{true})$ for each basic block b that ends with a (possibly implicit) return statement

Control Flow Graphs: Example



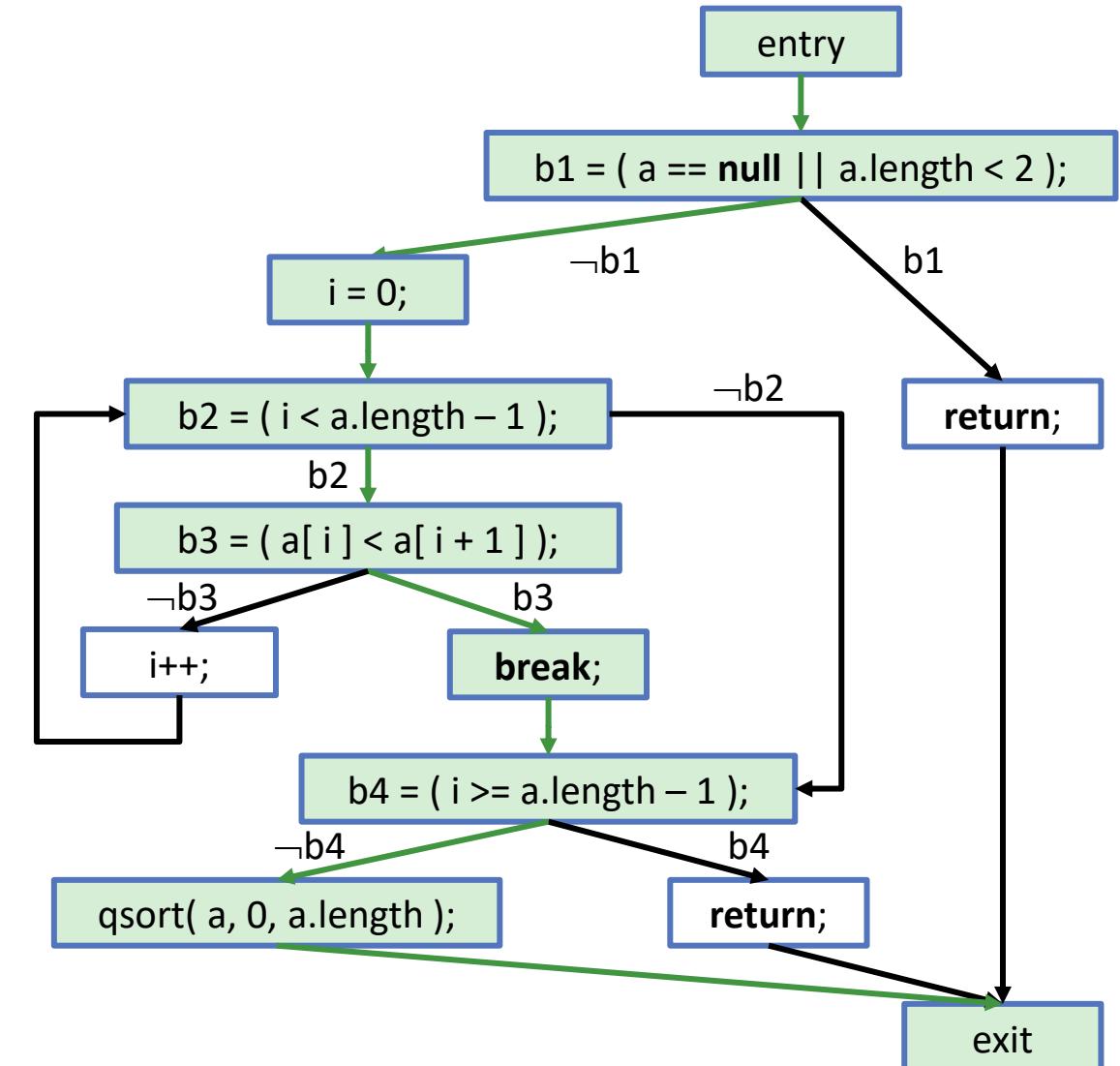
Test Coverage

- The CFG can serve as an **adequacy criterion** for test cases
- The more parts are executed, the higher the chance to uncover a bug
- “parts” can be nodes, edges, paths, etc.



Test Coverage: Example

- Consider the input $a = \{ 3, 7, 5 \}$



Statement Coverage

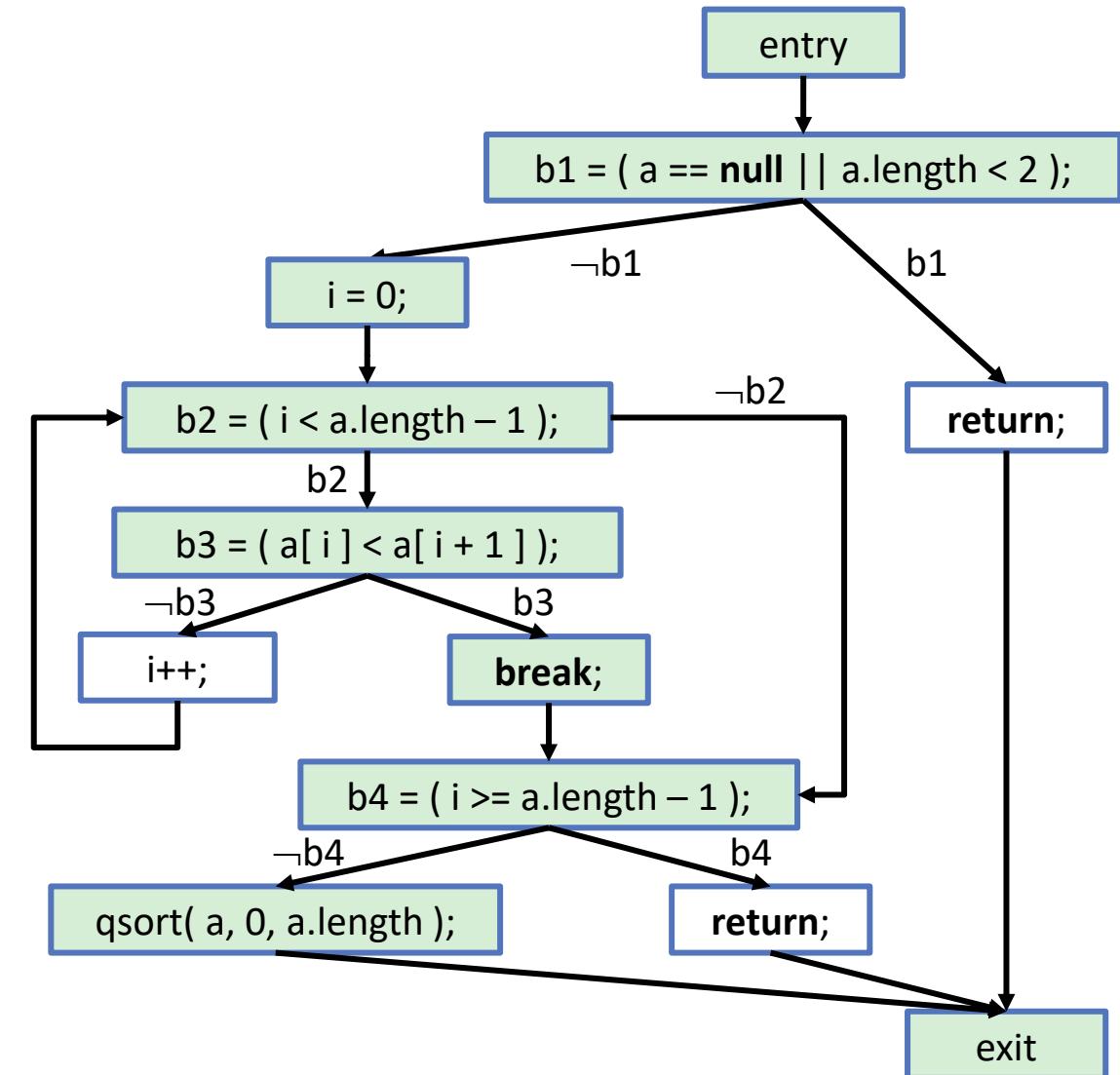
- Assess the quality of a test suite by measuring how much of the CFG it executes
- Idea: one can detect a bug in a statement only by executing the statement

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

- Can also be defined on basic blocks

Statement Coverage: Example

- Consider the input $a = \{ 3, 7, 5 \}$
- This test case executes 7 out of 10 basic statements
- Statement coverage: 70%

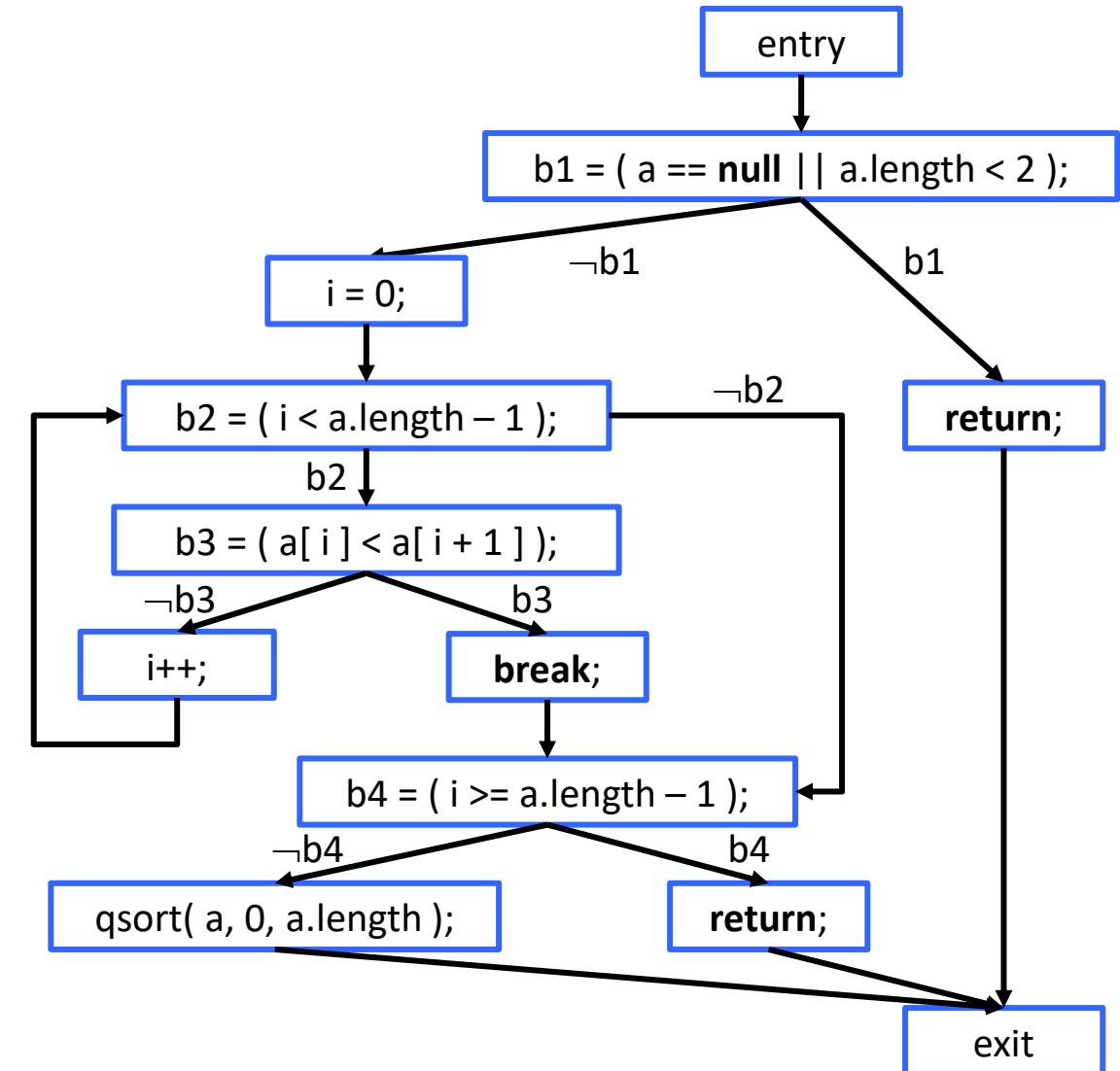


Statement Coverage: Example (cont'd)

- We can achieve 100% statement coverage with three test cases

- `a = { 1 }`
- `a = { 5, 7 }`
- `a = { 7, 5 }`

- The last test case detects the bug

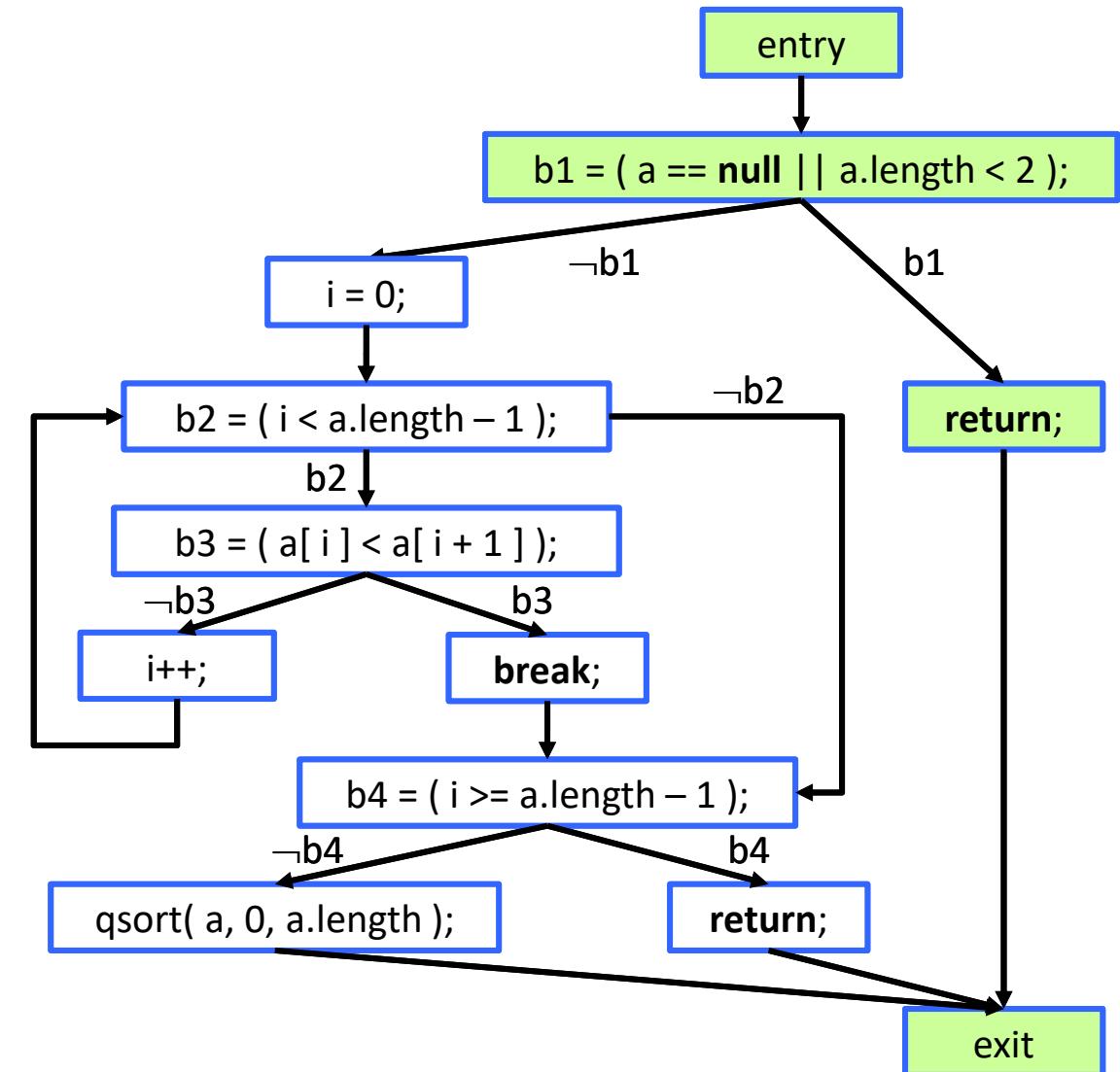


Statement Coverage: Example (cont'd)

- We can achieve 100% statement coverage with three test cases

- $a = \{ 1 \}$
- $a = \{ 5, 7 \}$
- $a = \{ 7, 5 \}$

- The last test case detects the bug

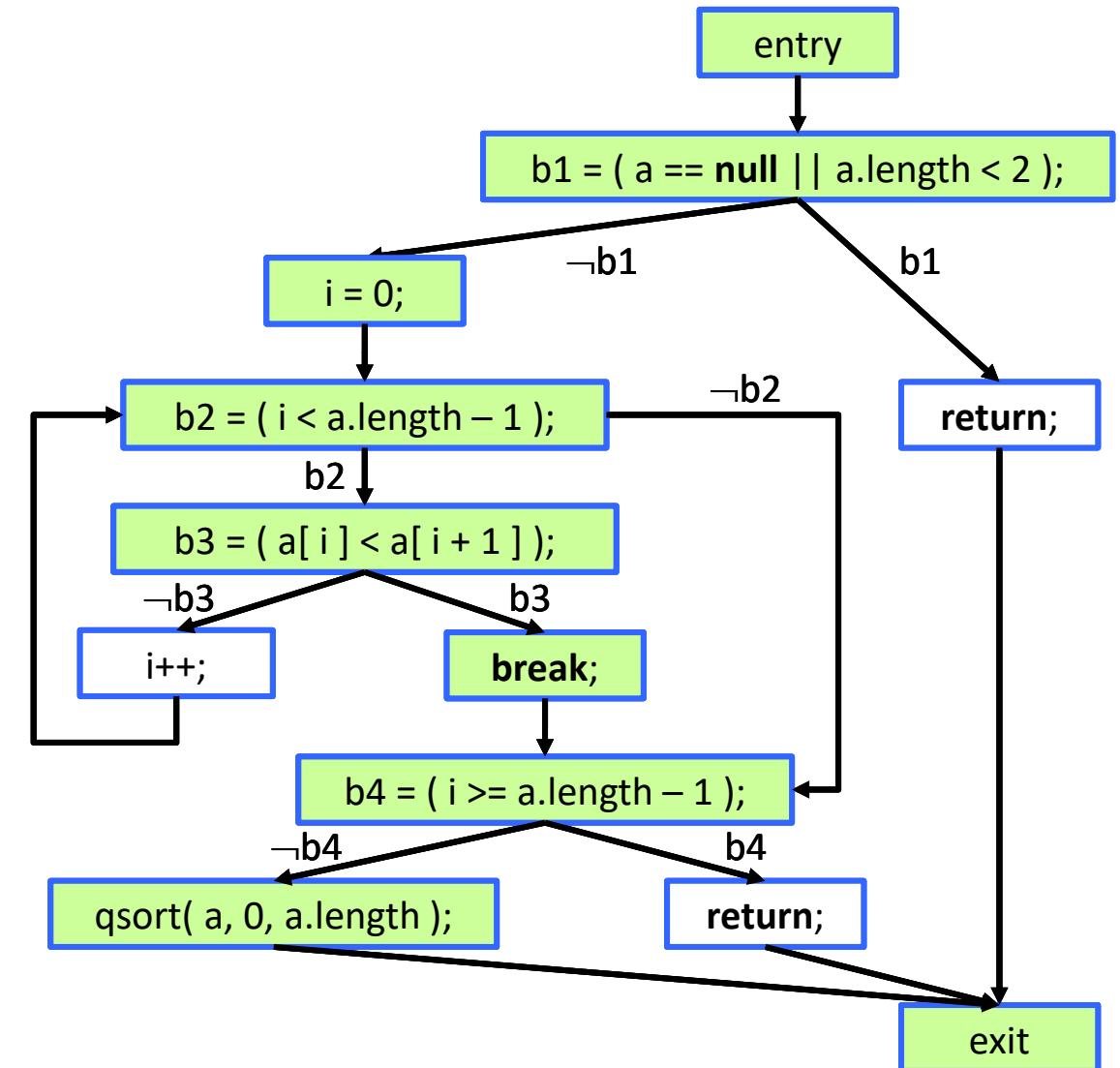


Statement Coverage: Example (cont'd)

- We can achieve 100% statement coverage with three test cases

- $a = \{ 1 \}$
- $a = \{ 5, 7 \}$
- $a = \{ 7, 5 \}$

- The last test case detects the bug

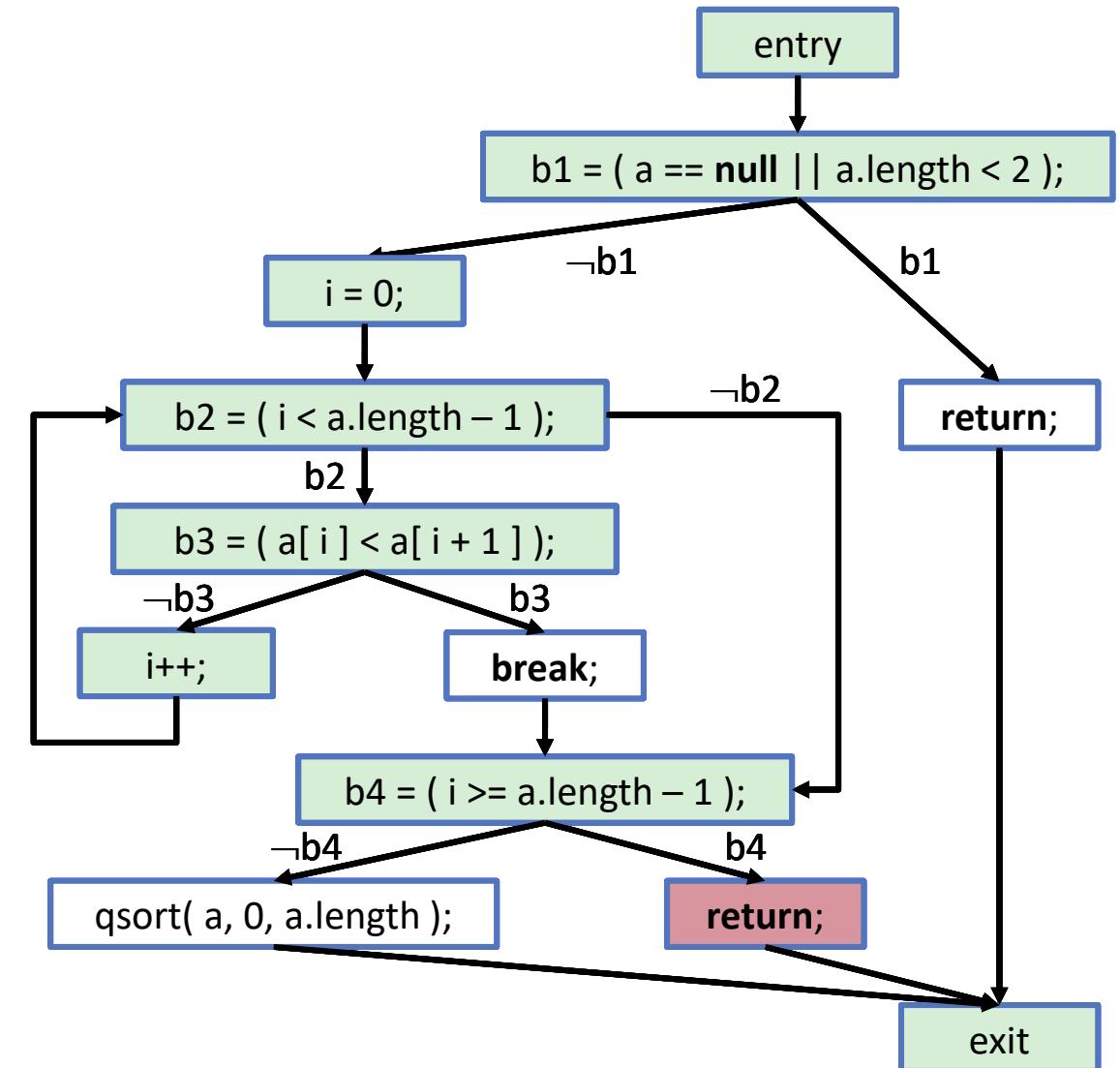


Statement Coverage: Example (cont'd)

- We can achieve 100% statement coverage with three test cases

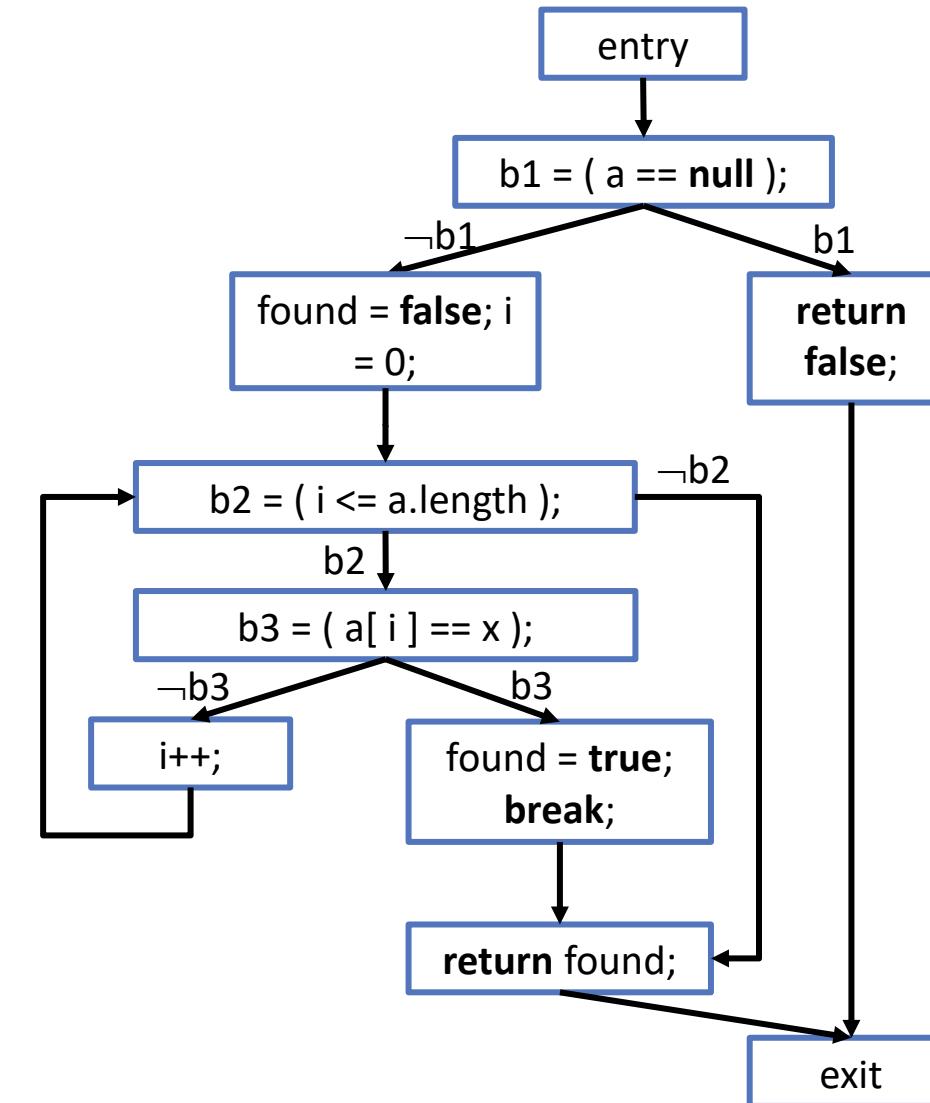
- $a = \{ 1 \}$
- $a = \{ 5, 7 \}$
- $a = \{ 7, 5 \}$

- The last test case detects the bug



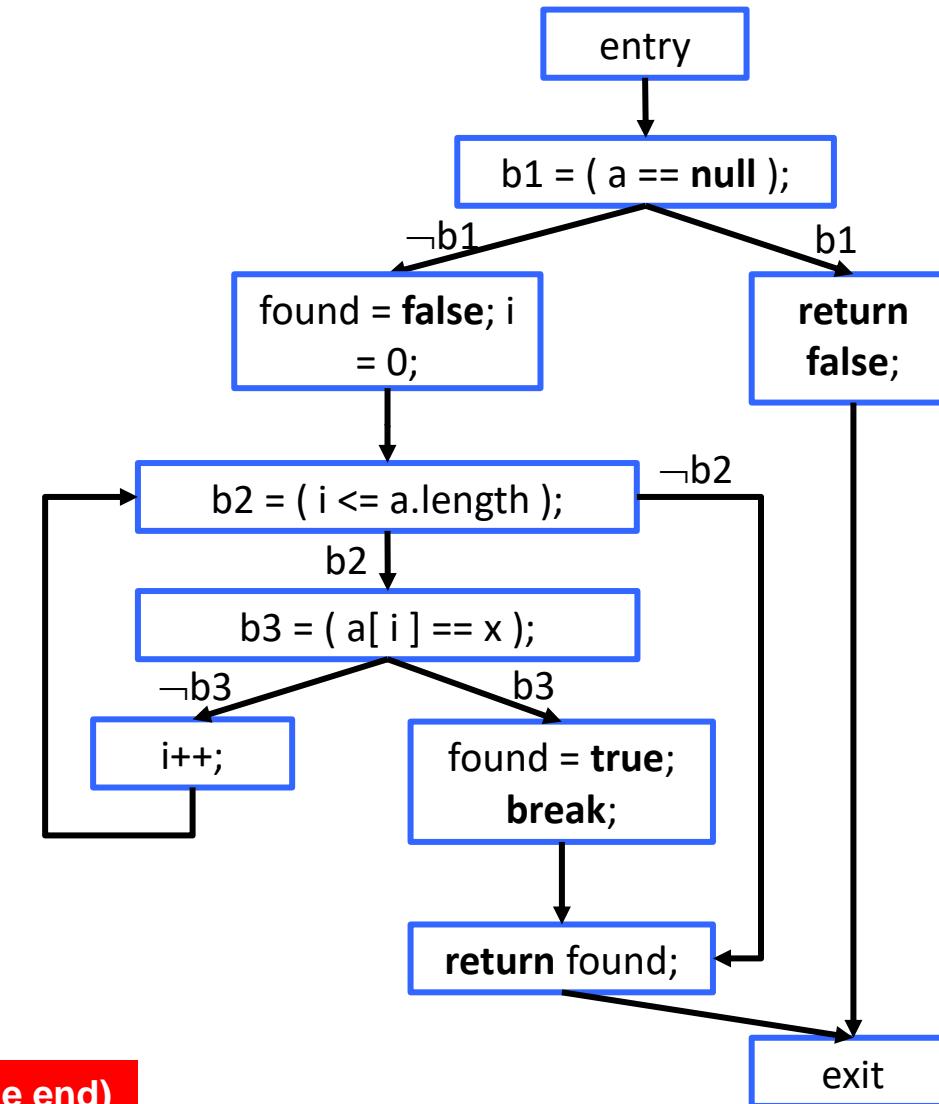
Statement Coverage: Discussion

```
boolean contains( int[ ] a, int x ) {  
    if( a == null ) return false;  
  
    boolean found = false;  
  
    for( int i = 0; i <= a.length; i++ ) {  
        if( a[ i ] == x ) {  
            found = true;  
            break;  
        }  
    }  
    return found;  
}
```



Statement Coverage: Discussion (cont'd)

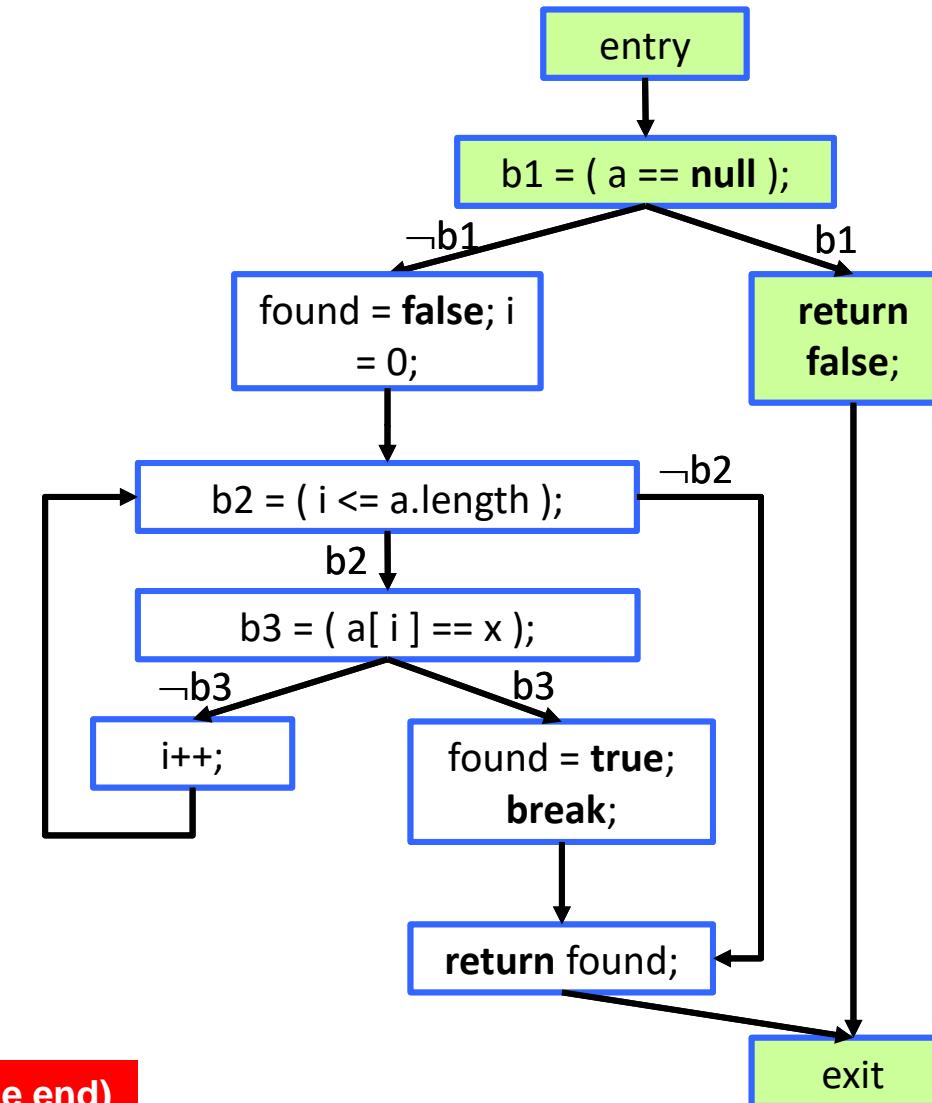
- We can achieve 100% statement coverage with two test cases
 - $a = \text{null}$
 - $a = \{ 1, 2 \}, x = 2$
- The test cases do not detect the bug!
- More thorough testing is necessary



Problem: we never take the $\neg b_2$ edge (we never run the loop till the end)

Statement Coverage: Discussion (cont'd)

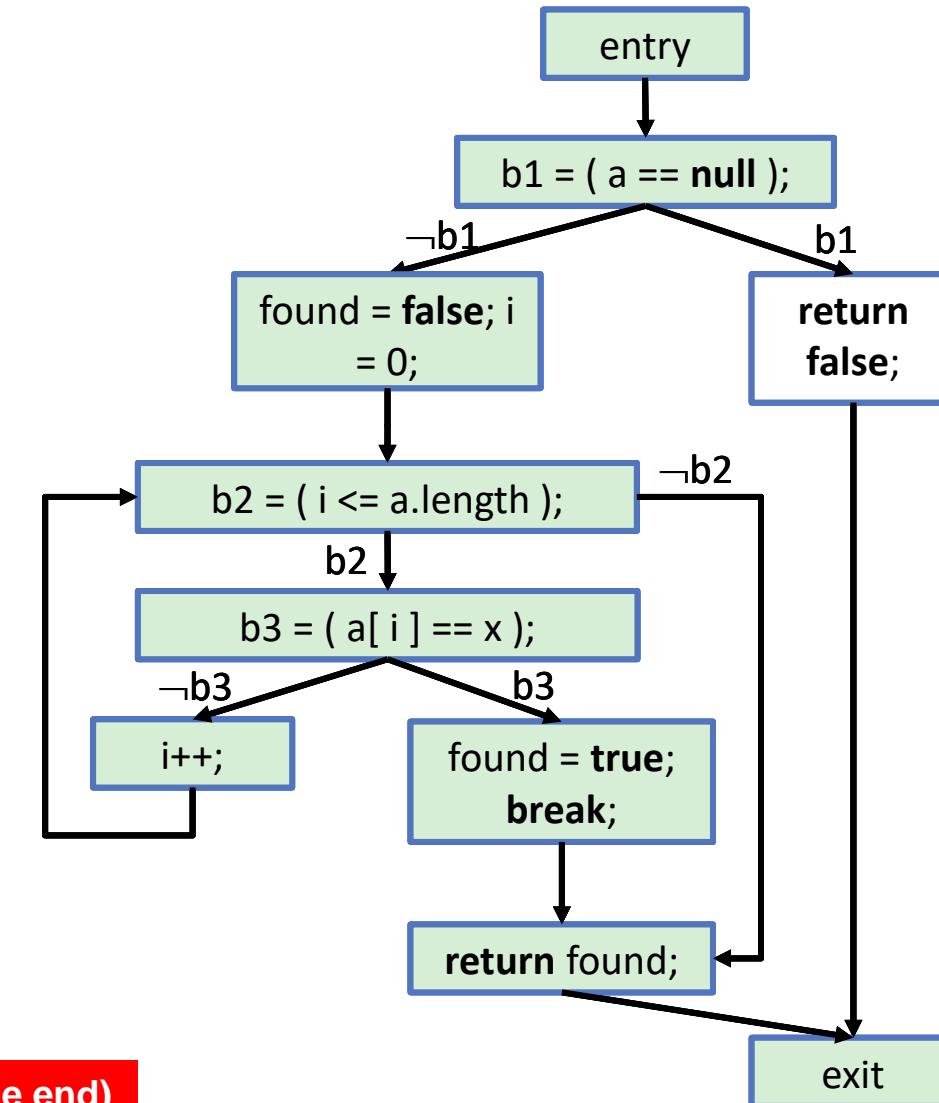
- We can achieve 100% statement coverage with two test cases
 - $a = \text{null}$
 - $a = \{ 1, 2 \}, x = 2$
- The test cases do not detect the bug!
- More thorough testing is necessary



Problem: we never take the $\neg b2$ edge (we never run the loop till the end)

Statement Coverage: Discussion (cont'd)

- We can achieve 100% statement coverage with two test cases
 - $a = \text{null}$
 - $a = \{ 1, 2 \}, x = 2$
- The test cases do not detect the bug!
- More thorough testing is necessary



Problem: we never take the $\neg b_2$ edge (we never run the loop till the end)

Branch Coverage

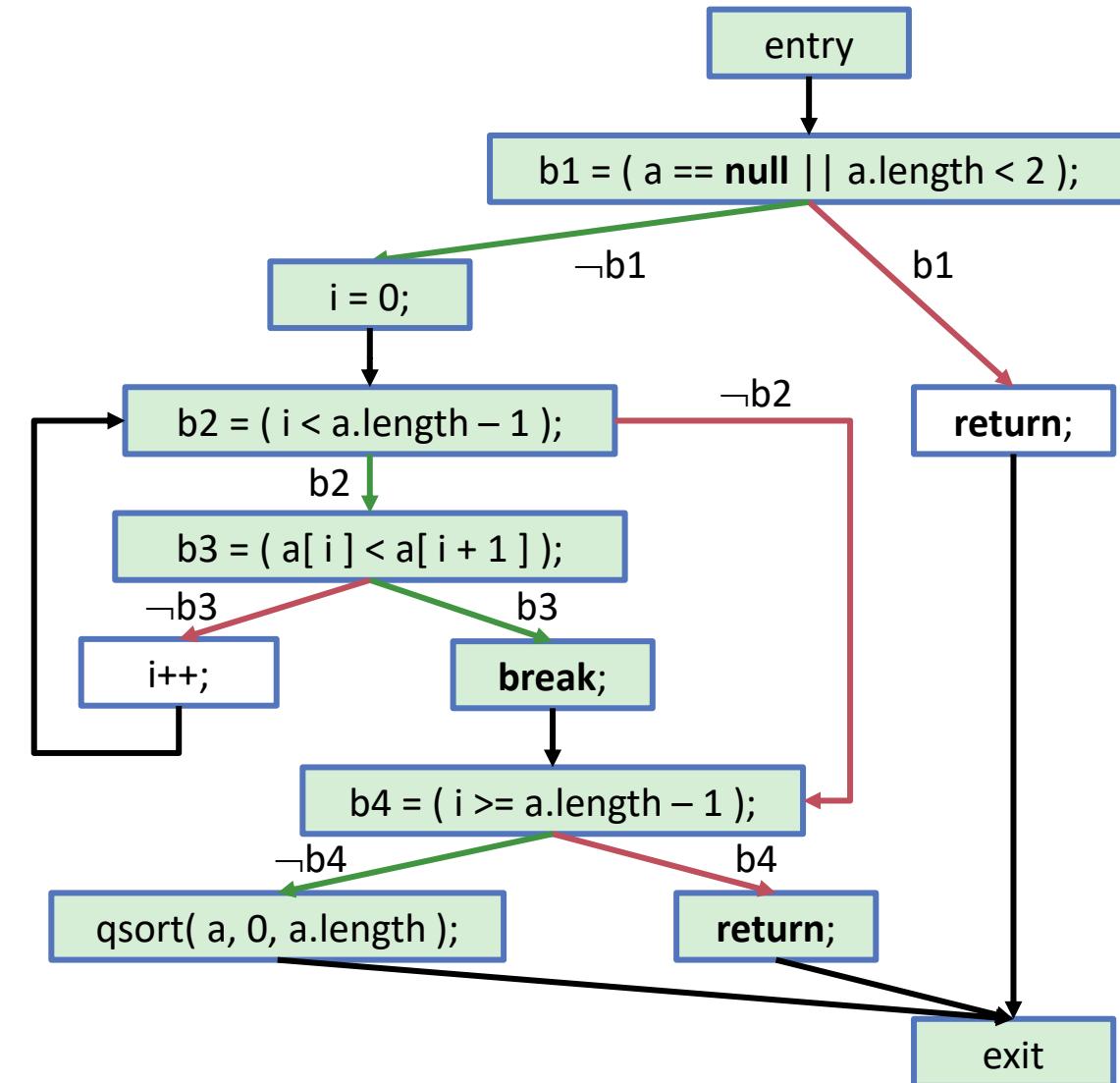
- Idea: test all possible branches in the control flow
- An edge (m, n, c) in a CFG is a branch iff there is another edge (m, n', c') in the CFG with $n \neq n'$

$$\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}$$

- Conveniently define branch coverage to be 100% if the code contains no branches

Branch Coverage: Example 1

- Consider the input $a = \{ 3, 7, 5 \}$
- This single test case executes 4 out of 8 branches
- Branch coverage: 50%
- Three test cases needed for 100% branch coverage



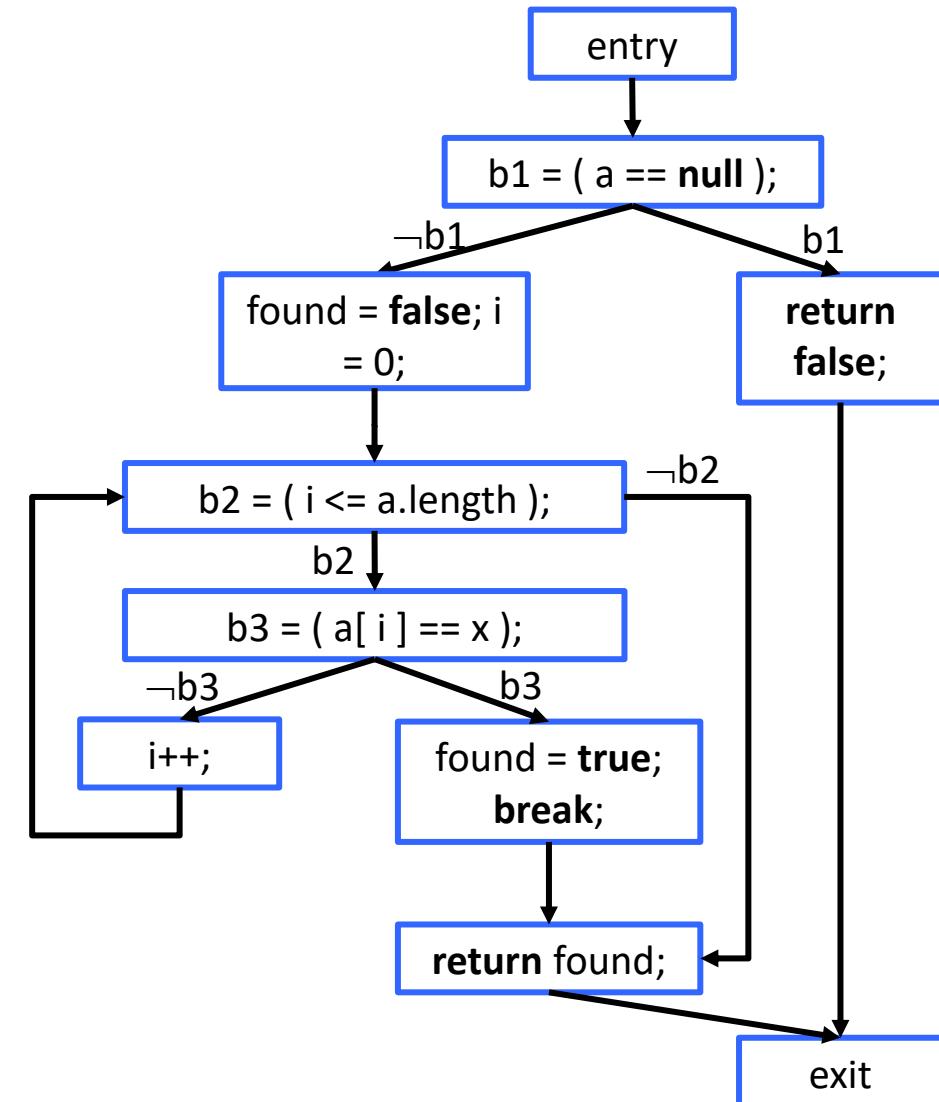
Branch Coverage: Example 2

- The two test cases

- `a = null`
- `a = { 1, 2 }, x = 2`

execute 5 out of 6 branches

- Branch coverage: 83%



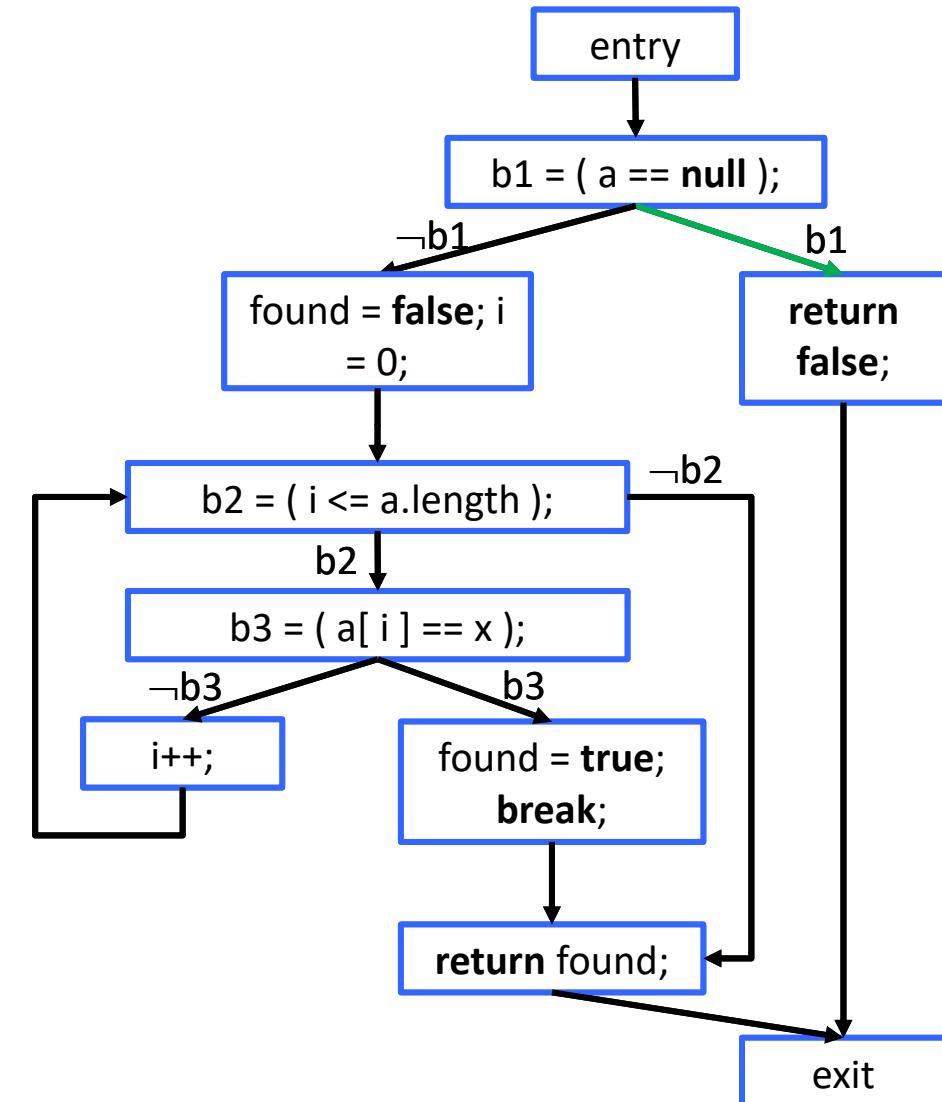
Branch Coverage: Example 2

- The two test cases

- `a = null`
- `a = { 1, 2 }, x = 2`

execute 5 out of 6 branches

- Branch coverage: 83%



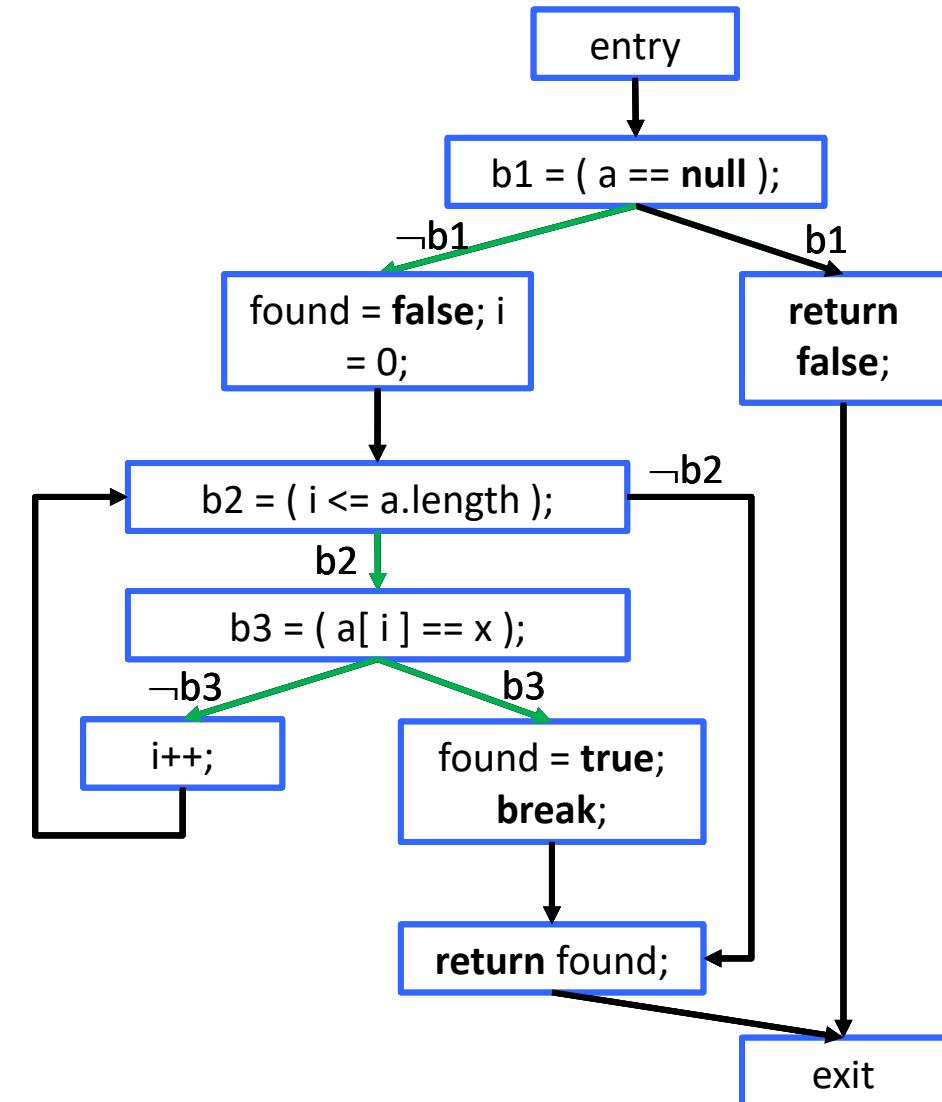
Branch Coverage: Example 2

- The two test cases

- `a = null`
- `a = { 1, 2 }, x = 2`

execute 5 out of 6 branches

- Branch coverage: 83%



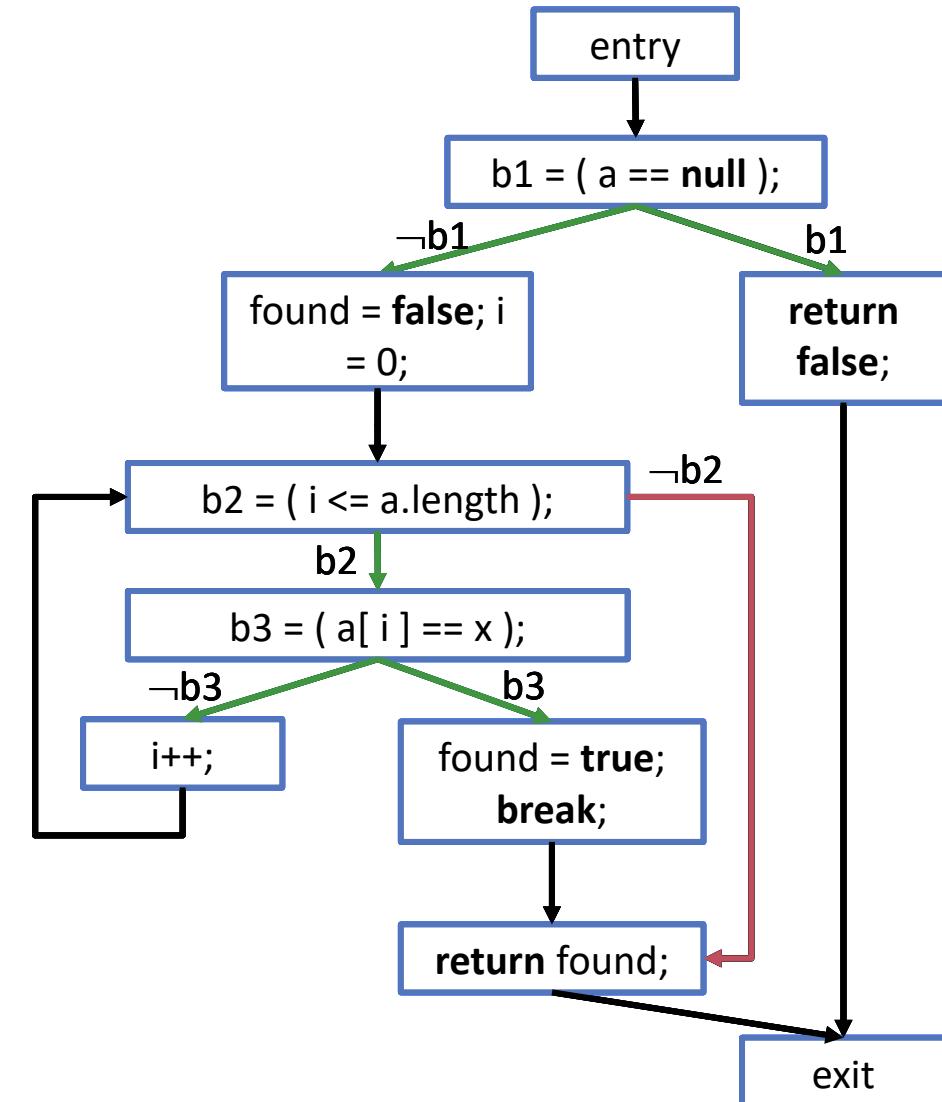
Branch Coverage: Example 2

- The two test cases

- `a = null`
- `a = { 1, 2 }, x = 2`

execute 5 out of 6 branches

- Branch coverage: 83%

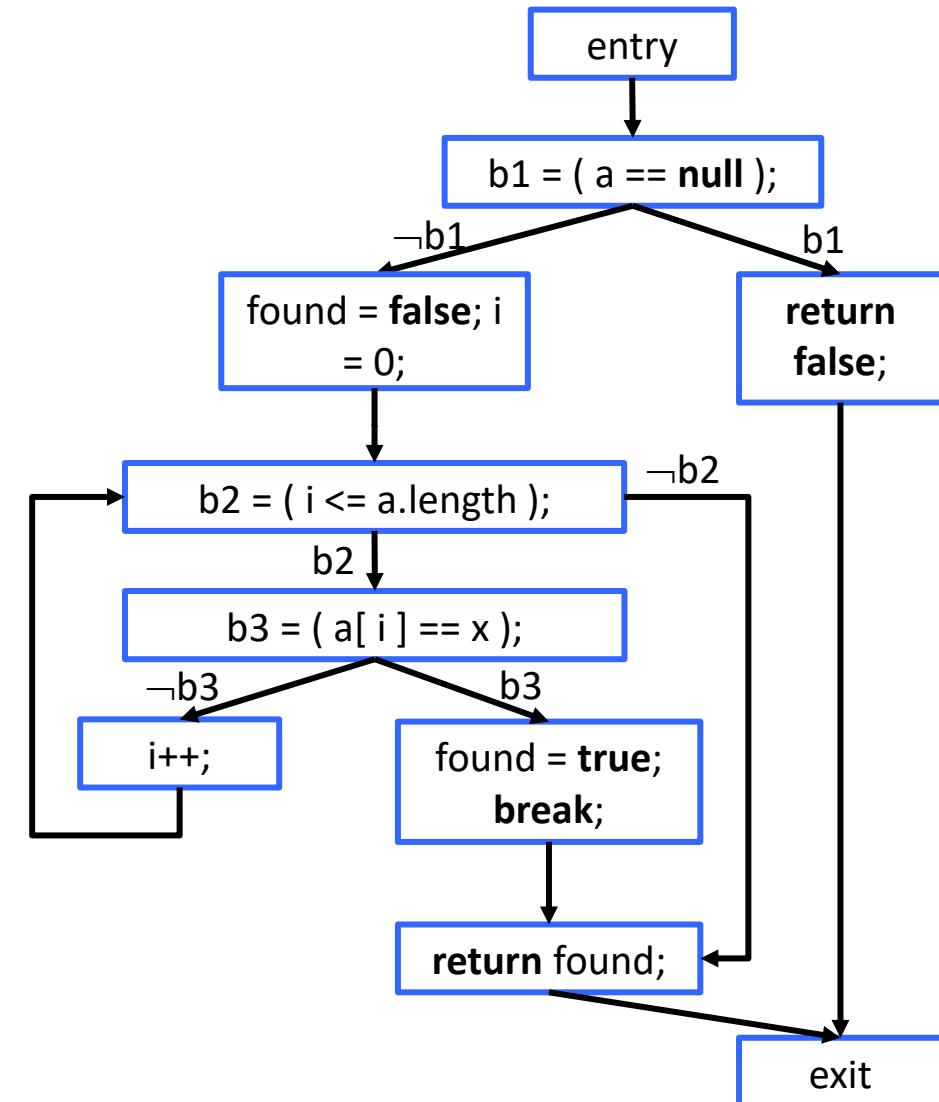


Branch Coverage: Example 2 (cont'd)

- Achieving 100% branch coverage would require a test case that runs the loop to the end

- `a = null`
- `a = { 1 }, x = 1`
- `a = { 1 }, x = 3`

- The last test case detects the bug

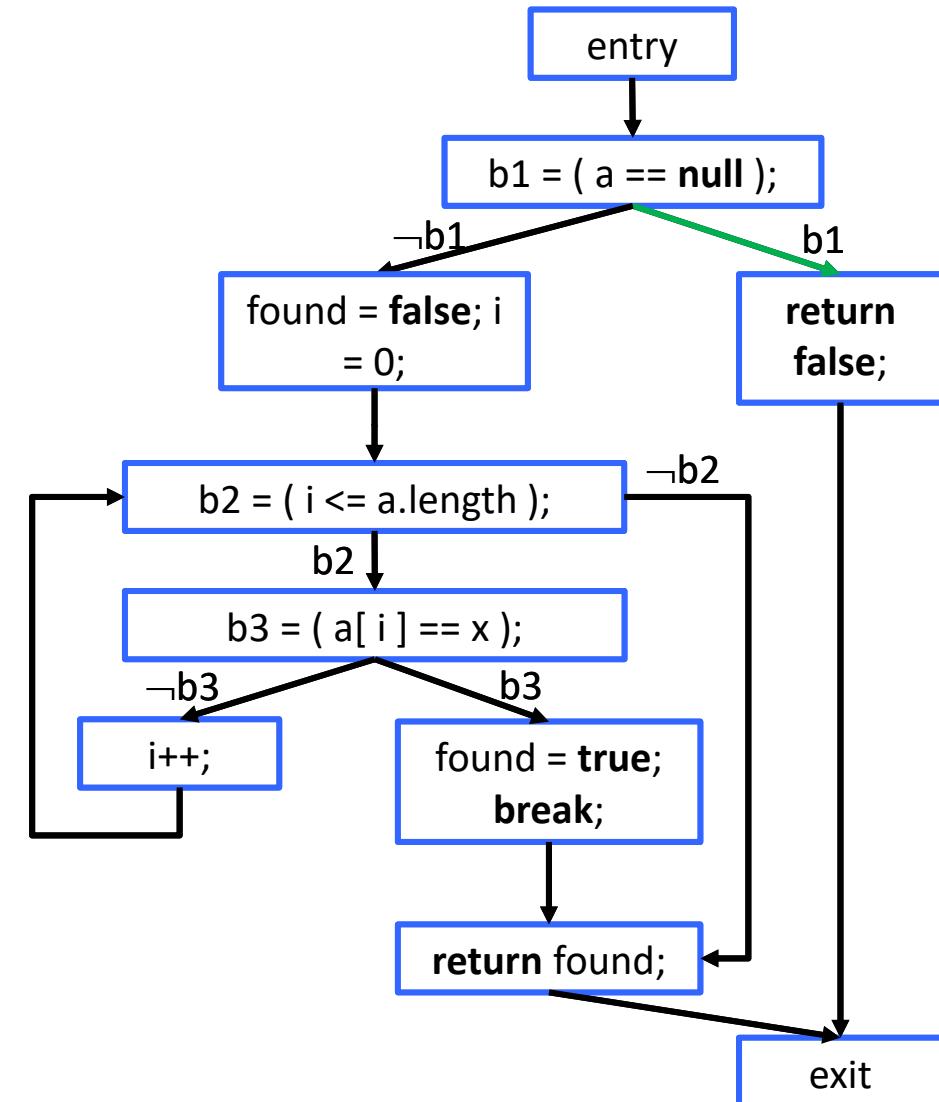


Branch Coverage: Example 2 (cont'd)

- Achieving 100% branch coverage would require a test case that runs the loop to the end

- `a = null`
- `a = { 1 }, x = 1`
- `a = { 1 }, x = 3`

- The last test case detects the bug

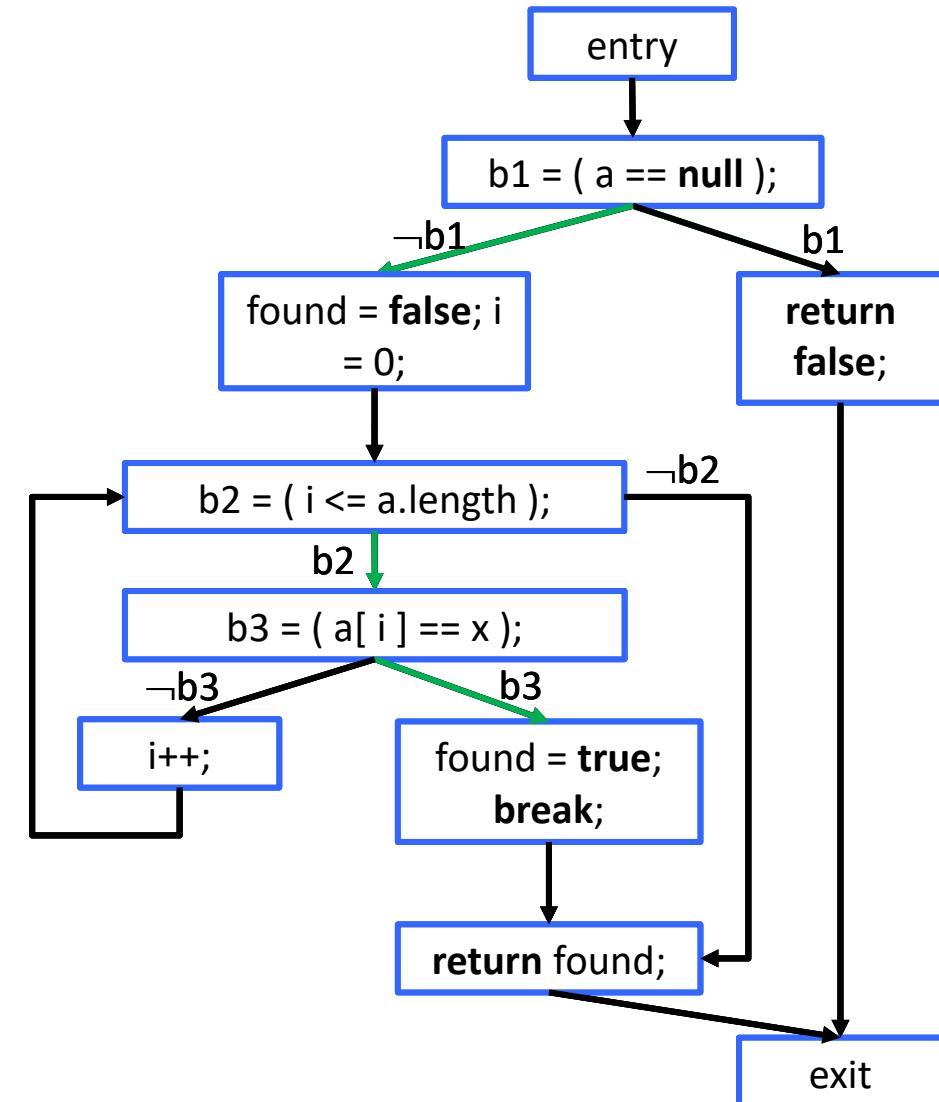


Branch Coverage: Example 2 (cont'd)

- Achieving 100% branch coverage would require a test case that runs the loop to the end

- `a = null`
- `a = { 1 }, x = 1`
- `a = { 1 }, x = 3`

- The last test case detects the bug

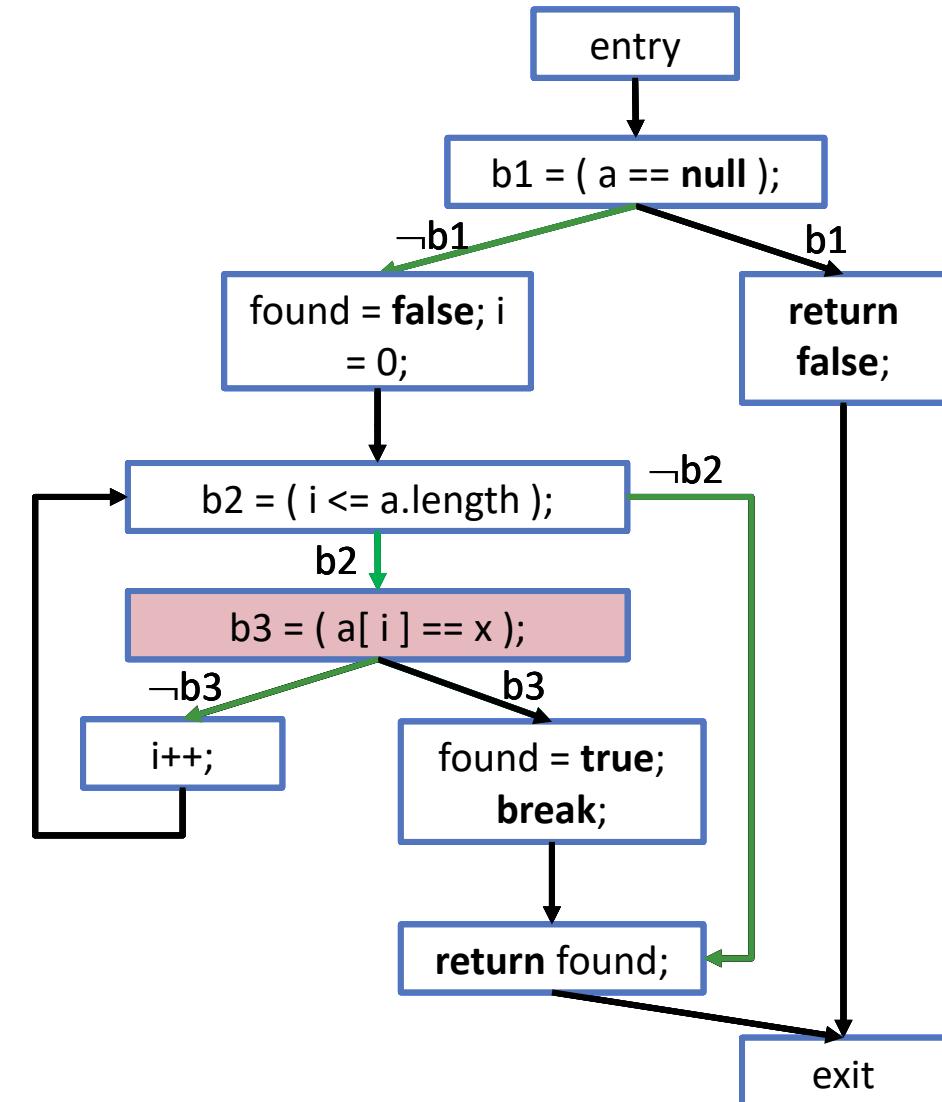


Branch Coverage: Example 2 (cont'd)

- Achieving 100% branch coverage would require a test case that runs the loop to the end

- $a = \text{null}$
- $a = \{ 1 \}, x = 1$
- $a = \{ 1 \}, x = 3$

- The last test case detects the bug

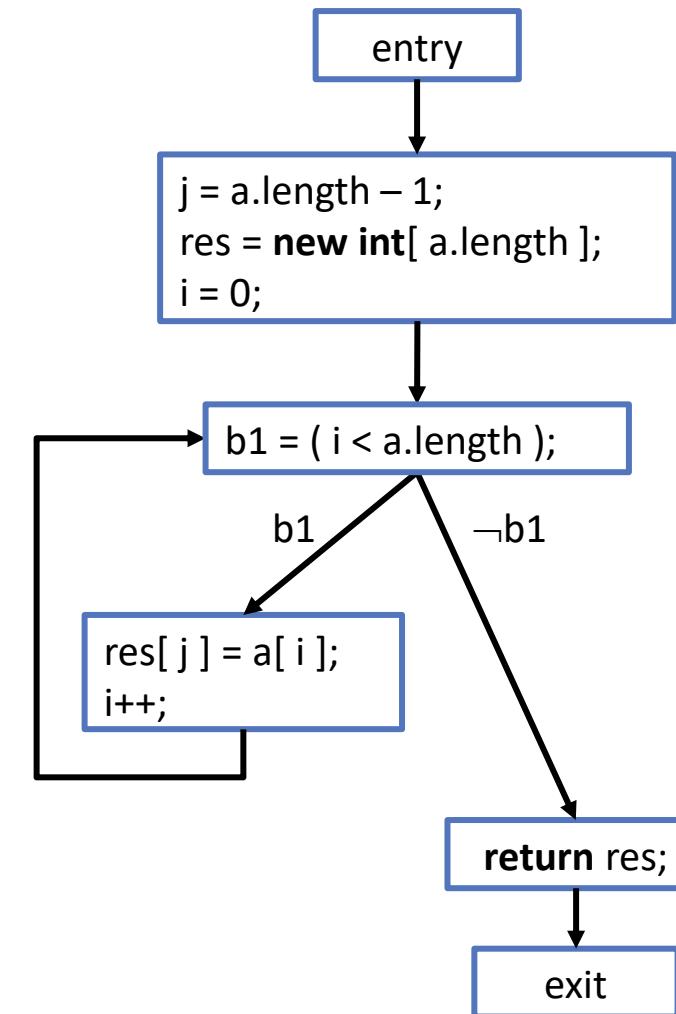


Branch Coverage: Discussion

- Branch coverage leads to more thorough testing than statement coverage
 - Complete branch coverage implies complete statement coverage
 - But “at least n% branch coverage” does not generally imply “at least n% statement coverage”
- Most widely-used adequacy criterion in industry

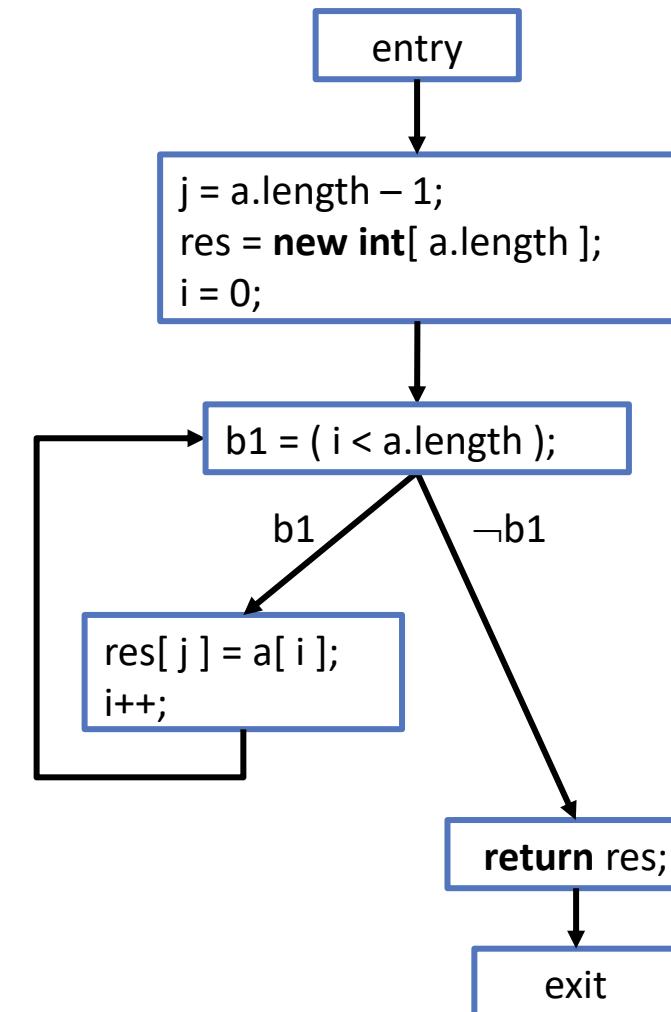
Branch Coverage: Discussion (cont'd)

```
int[ ] reverse( int[ ] a ) {  
    int j = a.length - 1;  
    int[ ] res = new int[ a.length ];  
    for( int i = 0; i < a.length; i++ ) {  
        res[ j ] = a[ i ];  
    }  
    return res;  
}
```



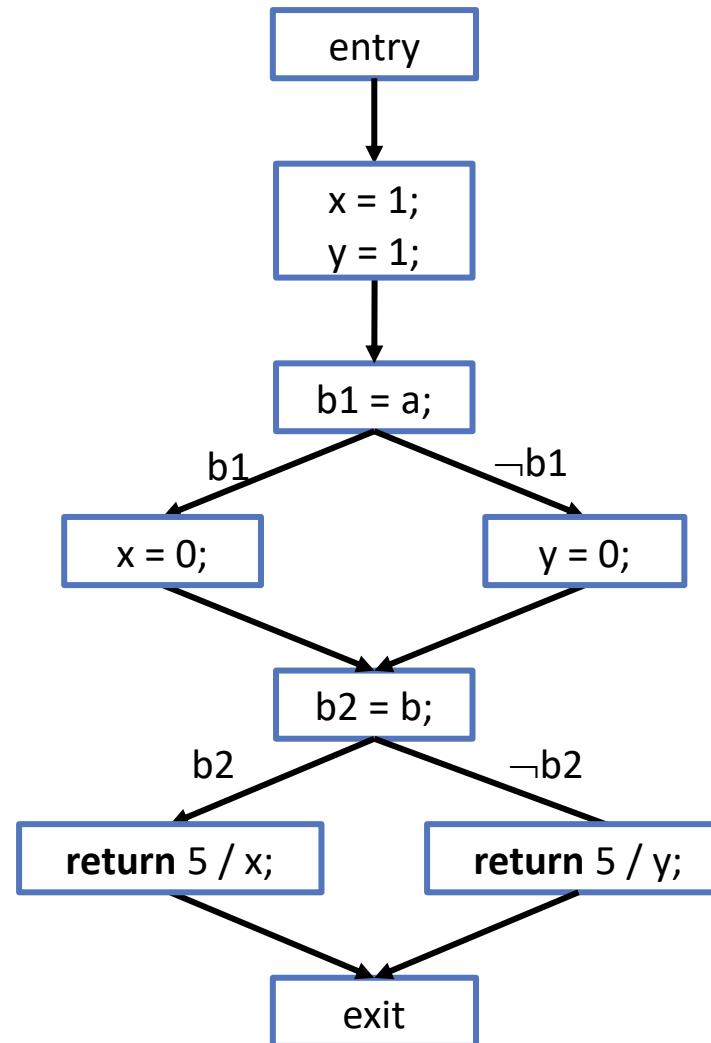
Branch Coverage: Discussion (cont'd)

- We can achieve 100% branch coverage with one test case
 - `a = { 1 }`
- The test case does not detect the bug!
- More thorough testing is necessary



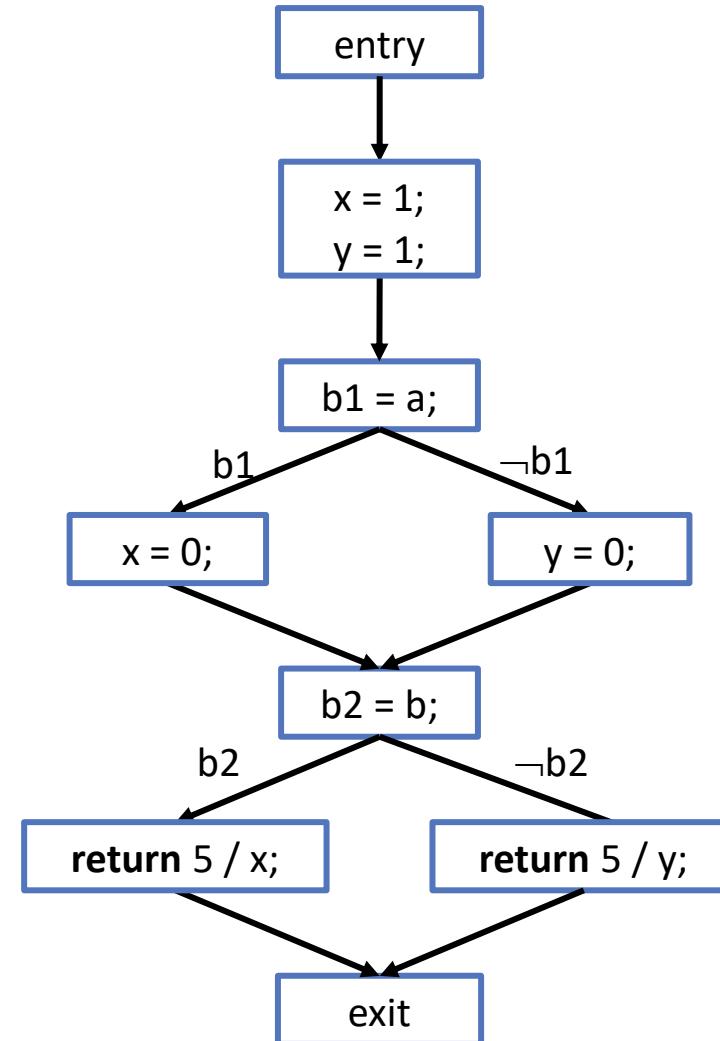
Branch Coverage: Discussion (cont'd)

```
int foo( boolean a, boolean b ) {  
    int x = 1;  
    int y = 1;  
    if( a )  
        x = 0;  
    else  
        y = 0;  
    if( b )  
        return 5 / x;  
    else  
        return 5 / y;  
}
```



Branch Coverage: Discussion (cont'd)

- We can achieve 100% branch coverage with two test cases
 - **a = true, b = false**
 - **a = false, b = true**
- The test cases do not detect the bug!
- More thorough testing is necessary



Path Coverage

- Idea: test all possible paths through the CFG
- A path is a sequence of nodes n_1, \dots, n_k such that
 - $n_1 = \text{entry}$
 - $n_k = \text{exit}$
 - There is an edge (n_i, n_{i+1}, c) in the CFG

Path Coverage =

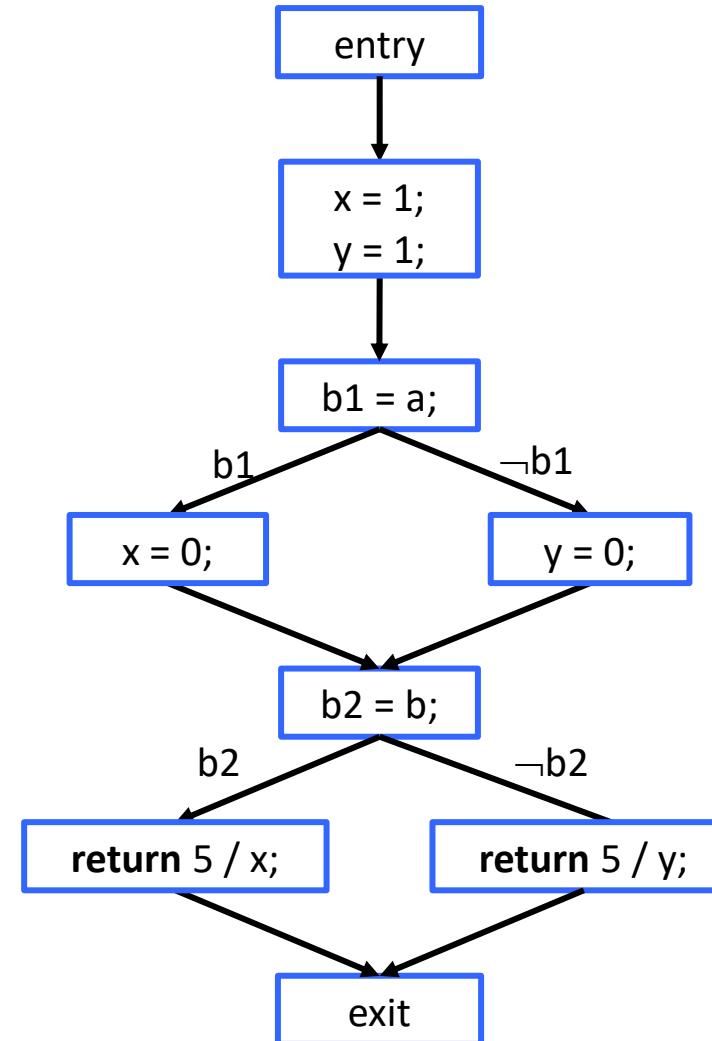
$$\frac{\text{Number of executed paths}}{\text{Total number of paths}}$$

Path Coverage: Example 1

- The two test cases
 - **a = true, b = false**
 - **a = false, b = true**

execute two out of four paths

- Path coverage: 50%

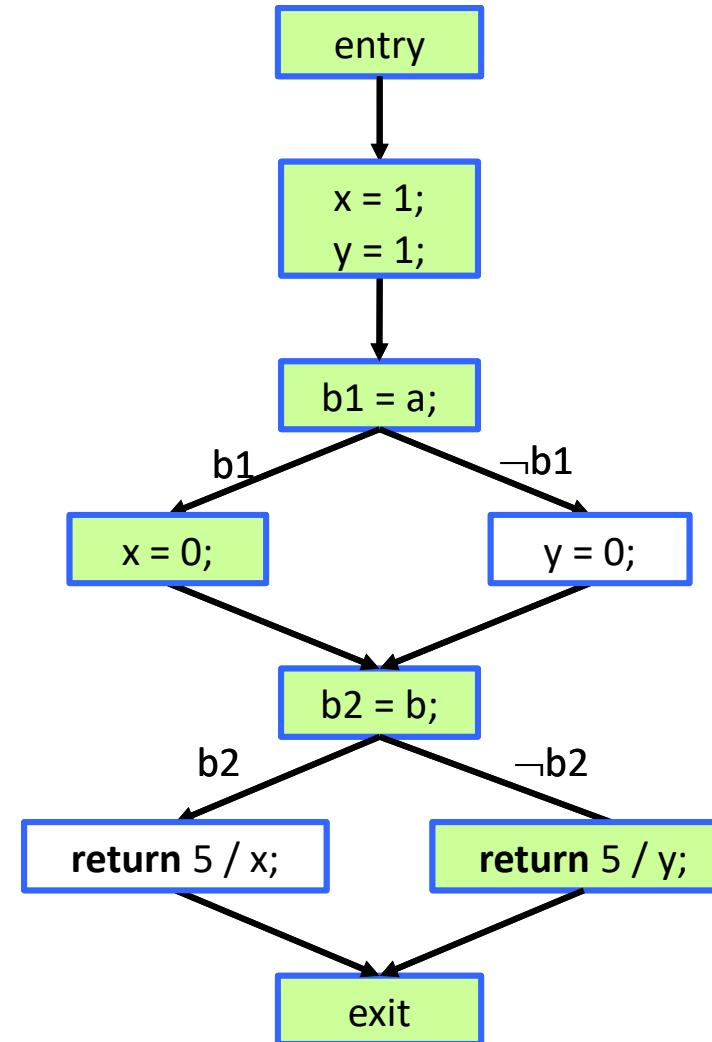


Path Coverage: Example 1

- The two test cases
 - **a = true, b = false**
 - **a = false, b = true**

execute two out of four paths

- Path coverage: 50%

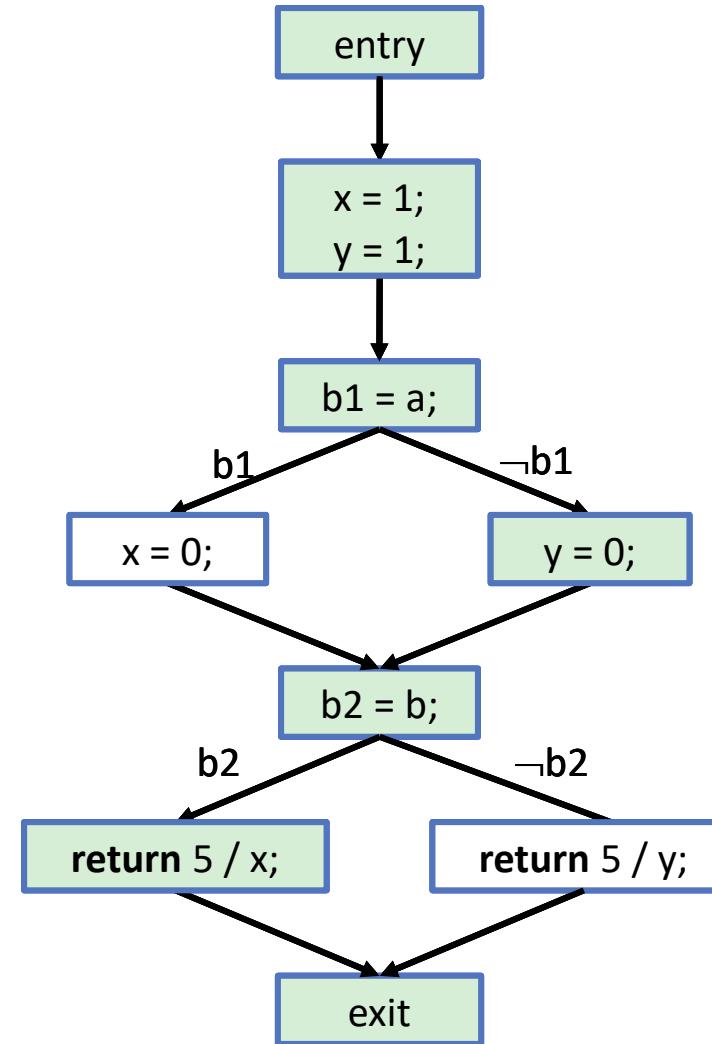


Path Coverage: Example 1

- The two test cases
 - **a = true, b = false**
 - **a = false, b = true**

execute two out of four paths

- Path coverage: 50%

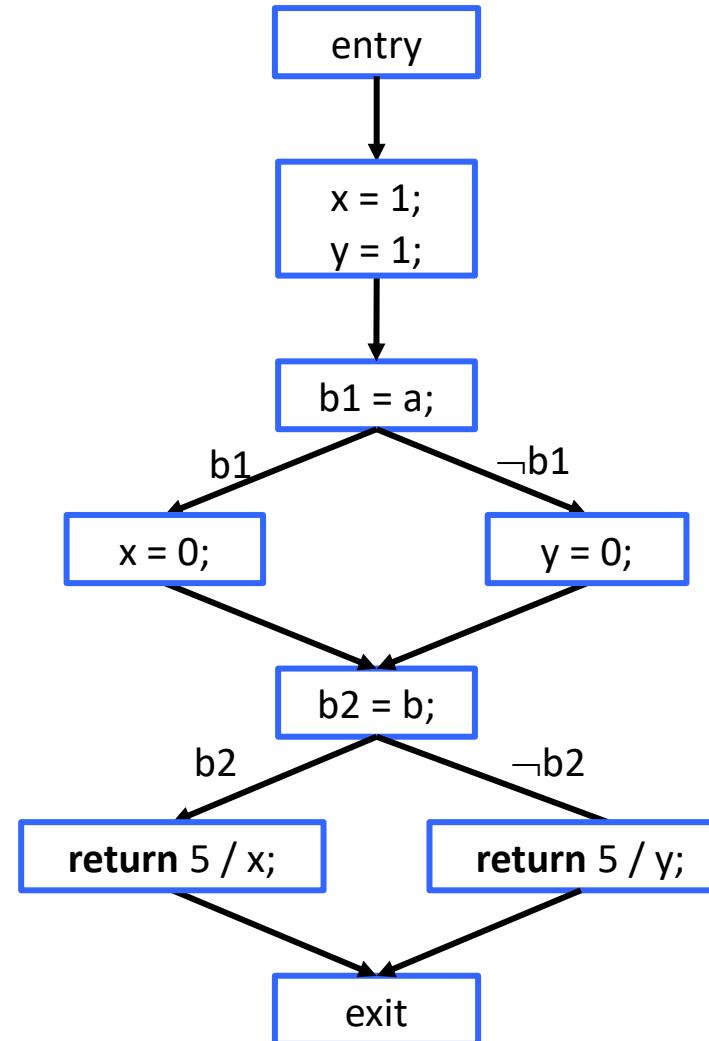


Path Coverage: Example 1 (cont'd)

- We can achieve 100% path coverage with four test cases

- a = **true**, b = **false**
- a = **false**, b = **true**
- a = **true**, b = **true**
- a = **false**, b = **false**

- The two additional test cases detect the bugs

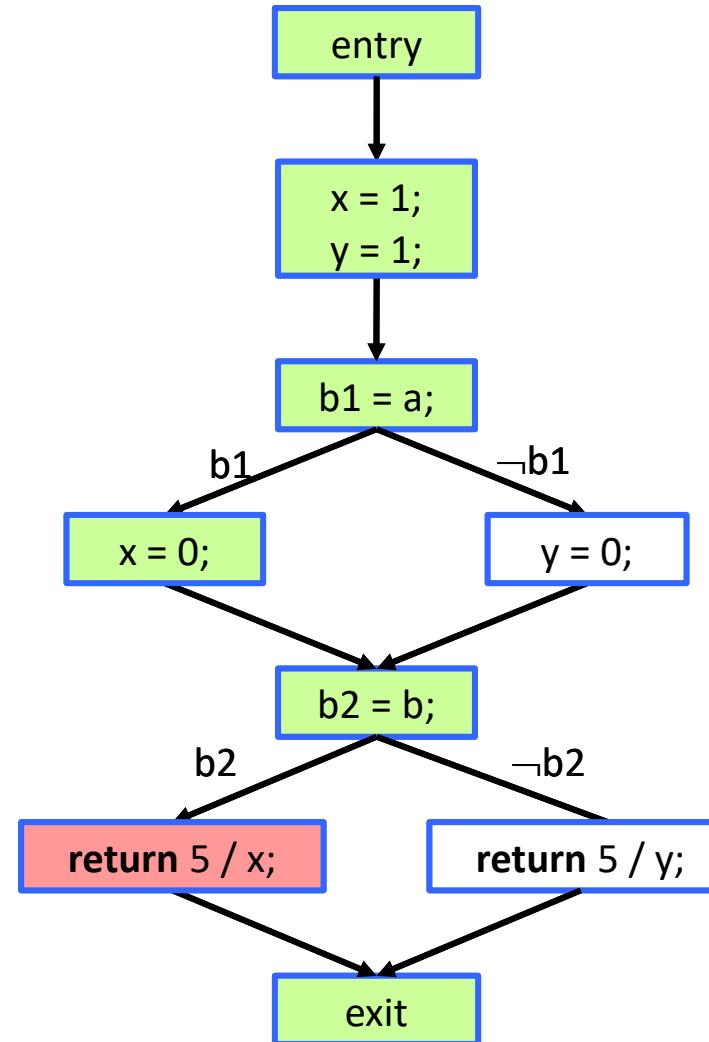


Path Coverage: Example 1 (cont'd)

- We can achieve 100% path coverage with four test cases

- a = **true**, b = **false**
- a = **false**, b = **true**
- a = **true**, b = **true**
- a = **false**, b = **false**

- The two additional test cases detect the bugs

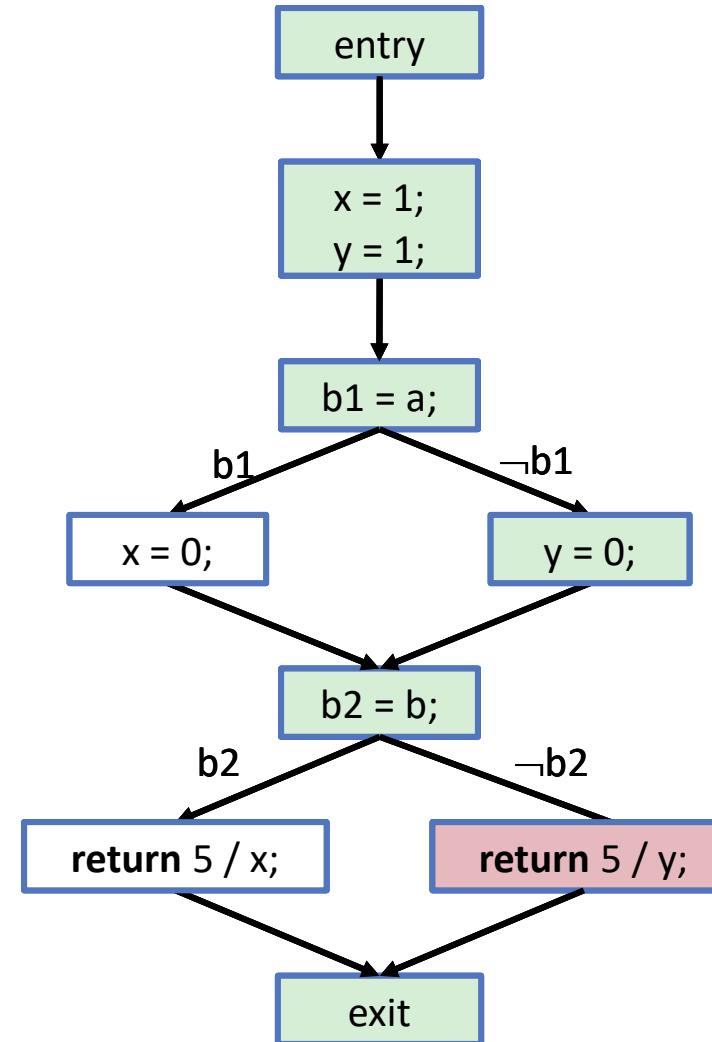


Path Coverage: Example 1 (cont'd)

- We can achieve 100% path coverage with four test cases

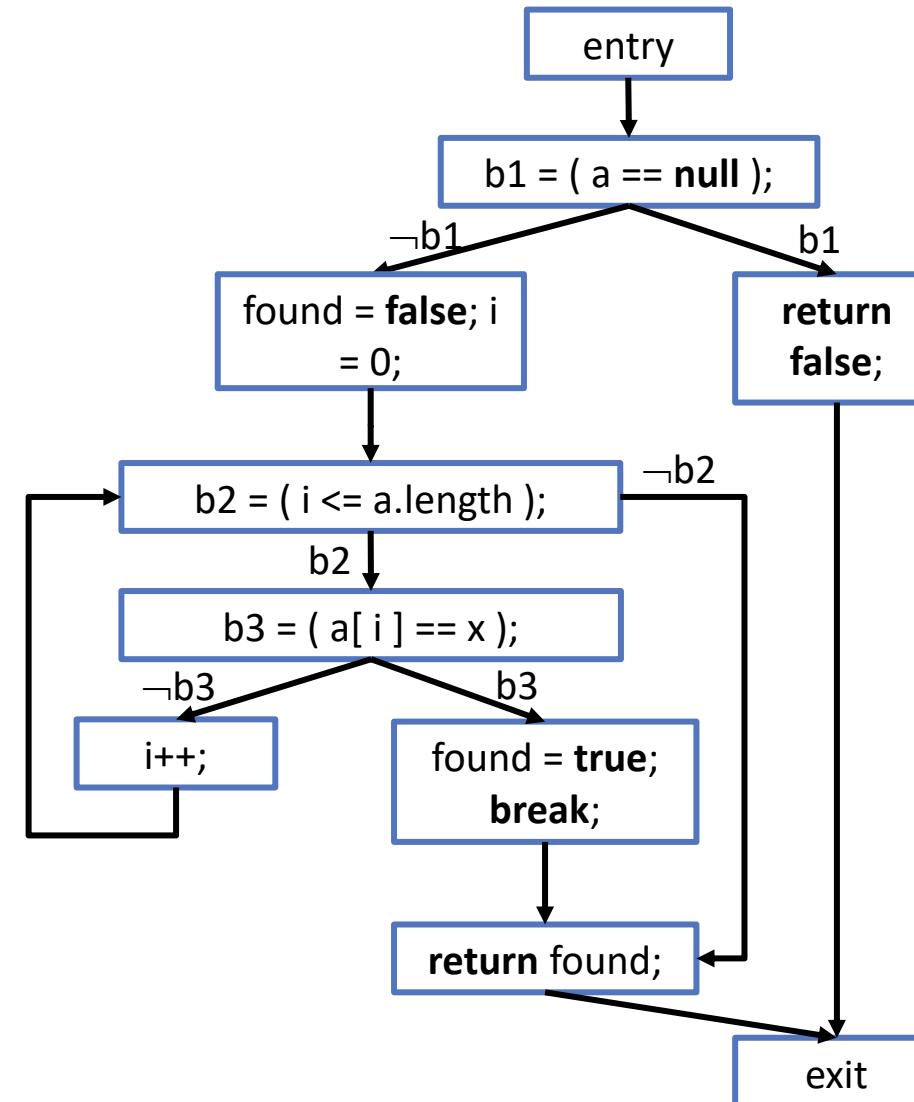
- a = **true**, b = **false**
- a = **false**, b = **true**
- a = **true**, b = **true**
- a = **false**, b = **false**

- The two additional test cases detect the bugs



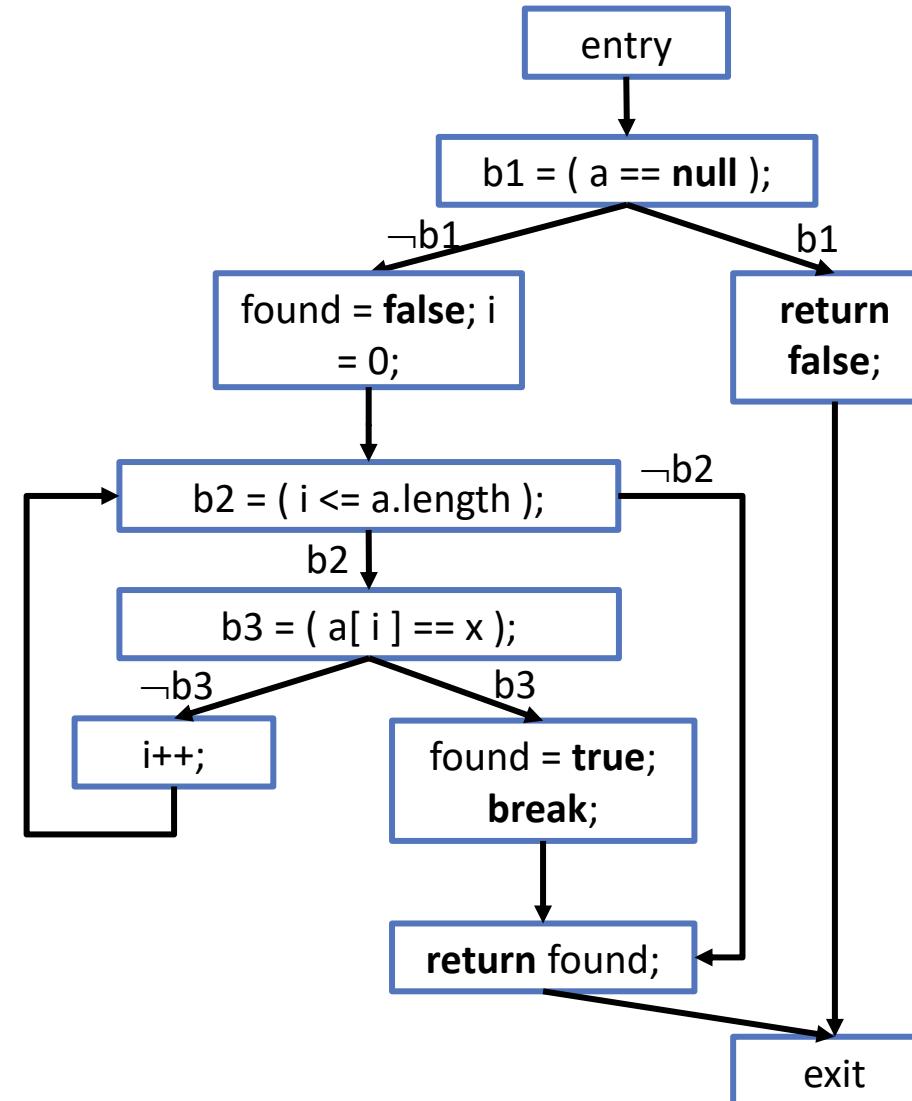
Path Coverage: Example 2

```
boolean contains( int[ ] a, int x ) {  
    if( a == null ) return false;  
  
    boolean found = false;  
  
    for( int i = 0; i <= a.length; i++ ) {  
  
        if( a[ i ] == x ) {  
  
            found = true;  
  
            break;  
        }  
    }  
  
    return found;  
}
```



Path Coverage: Example 2 (cont'd)

- Number of loop iterations is not known statically
- An arbitrarily large number of test cases is needed for complete path coverage



Path Coverage: Discussion

- Path coverage leads to more thorough testing than both statement and branch coverage
 - Complete path coverage implies complete statement coverage and complete branch coverage
 - But “at least n% path coverage” does not generally imply “at least n% statement coverage” or “at least n% branch coverage”
- Complete path coverage is not feasible for loops
 - Unbounded number of paths

Loop Coverage

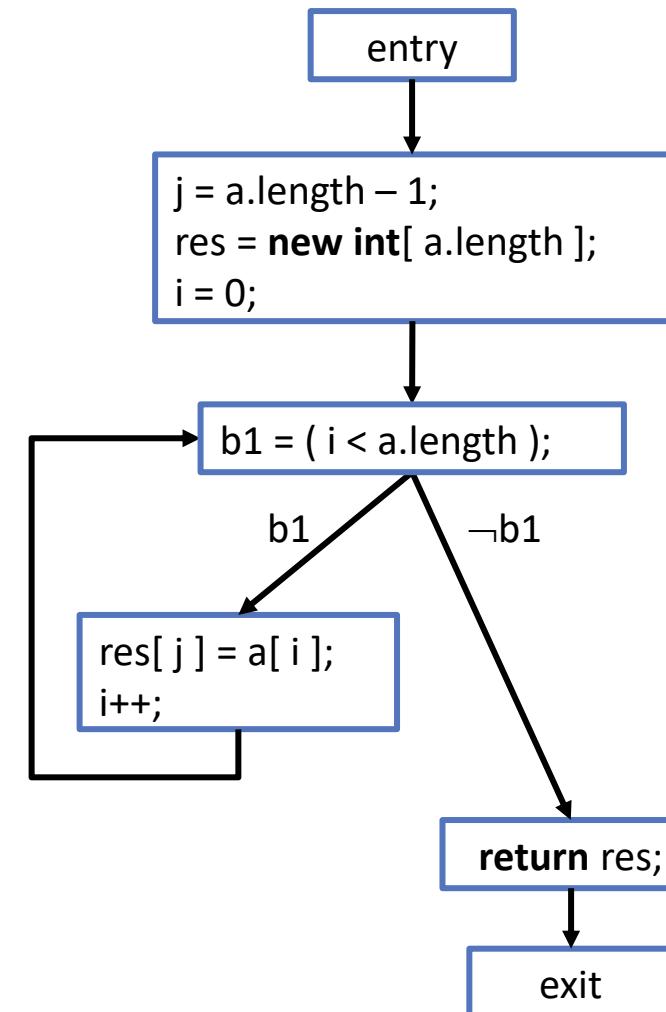
- Idea: for each loop, test zero, one, and more than one (consecutive) iterations

$$\text{Loop Coverage} = \frac{\text{Number of executed loops with 0, 1, and more than 1 iterations}}{\text{Total number of loops} * 3}$$

- Loop coverage is typically combined with other adequacy criteria such as statement or branch coverage

Loop Coverage: Example

- The test case
 - `a = { 1 }`executes one out of three possible cases for the loop
- Loop coverage: 33%

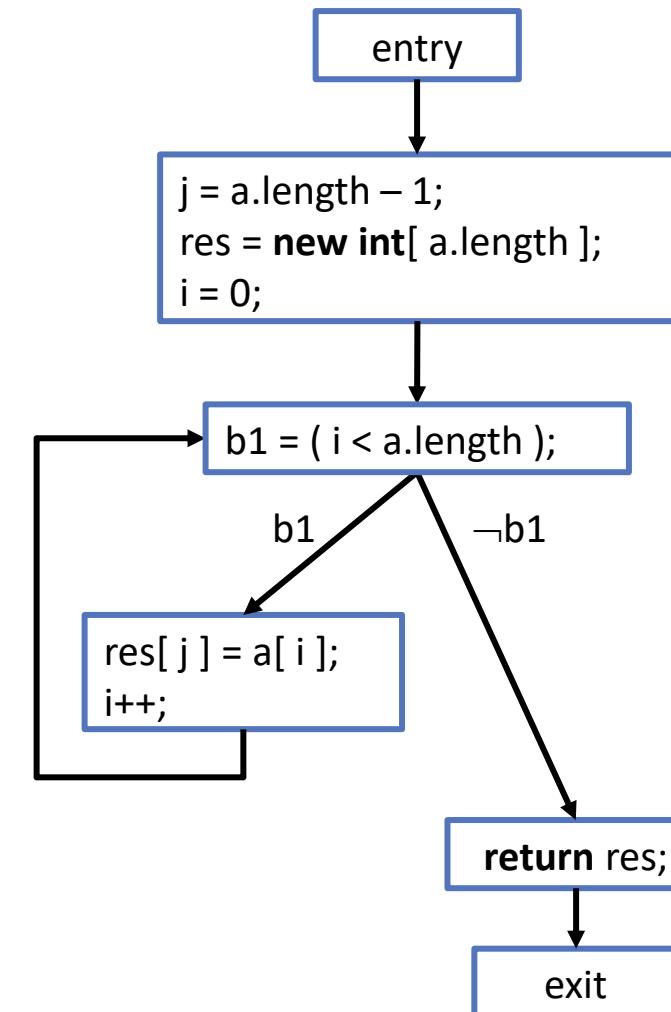


Loop Coverage: Example

- We can achieve 100% loop coverage with three test cases

- `a = {}`
- `a = { 1 }`
- `a = { 1, 2 }`

- The last test case detects the bug



Measuring Coverage

- Coverage information is collected while the test cases execute
- Use code instrumentation or debug interface to count executed basic blocks, branches, etc.

```
int foo( boolean a, boolean b ) {  
    int x = 1; int y = 1;  
    if( a ) {  
        branchCovered[ 0 ] = true; x = 0;  
    } else {  
        branchCovered[ 1 ] = true; y = 0;  
    }  
    if( b ) {  
        branchCovered[ 2 ] = true;  
        return 5 / x;  
    } else {  
        branchCovered[ 3 ] = true;  
        return 5 / y;  
    }  
}
```

Interpreting Coverage

- High coverage does not mean that code is well tested
 - But: low coverage means that code is not well tested
 - Make sure you do not blindly optimize coverage but develop test suites that test the code well
- Coverage tools do not only measure coverage metrics, they also identify which parts of the code have not been tested

Experimental Evaluation: Approach

- Several studies investigate the benefit of coverage metrics
 - Andrews et al.: “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria”, TR SCE-06-02, 2006
- Approach
 - Seed defects in the code
 - Develop test suites that satisfy various coverage criteria
 - Measure how many of the seeded defects are found by the test suits
 - Extrapolate to “real” defects in the code

5. Testing Summary

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.4 Structural Testing

Rigorous Software Engineering

Introduction

Martin Vechev

Slides adapted from previous years
(special thanks to Peter Müller, Felix Friedrich, Hermann Lehner)



whoami

Professor of Computer Science at ETH since January 2012



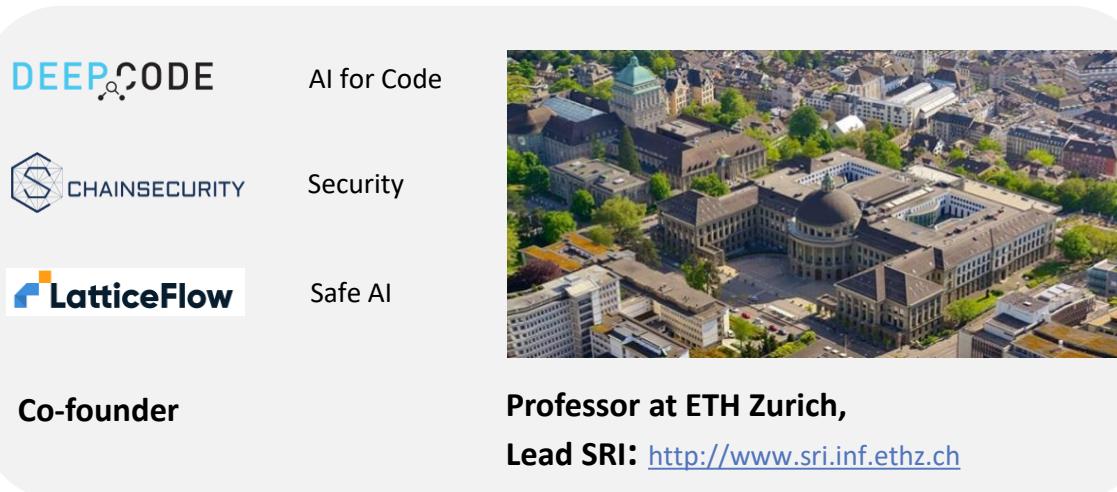
Sofia, Bulgaria



SFU, Canada, B.Sc.



Cambridge, England, PhD



Researcher @
IBM T.J. Watson Research
Center, New York, USA

1. Introduction

1.1 Software Failures

1.2 Challenges

1.3 Clean Code

1.4 Solution Approaches (Course Outline)

Software is Everywhere



Bad Software is Also Everywhere



The Ariane 5 Desaster (4th June 1997)

- 39 seconds after take-off, the rocket Ariane 5, successor of Ariane 4, ran dramatically off-course, leading to separation of the boosters, in turn triggering the self-destruct system
- The reason for the fault was eventually attributed to a **floating-point conversion error** in both the inertial reference system SRI1 and its replicate SRI2.
- The code (that was even not necessary after lift-off) had worked for Ariane 4 because of a less curved trajectory.



The Patriot Accident

- The Patriot missile air defense system tracks and intercepts incoming missiles
- On February 25, 1991, a Patriot system ignored an incoming Scud missile
- 28 soldiers died; 98 were injured



Patriot Bug – Rounding Error

- The tracking algorithm measures time in 1/10s
- Time is stored in a 24-bit register
 - Precise binary representation of 1/10 (infinite):
0.0001100110011001100110011001...
 - Truncated value in 24-bit register:
0.00011001100110011001100
 - Rounding error: ca. 0.00000095s every 1/10s
- After 100 hours of operation, the error is
 $0.00000095s \times 10 \times 3600 \times 100 = 0.34s$
- A Scud travels at about 1.7km/s, and so travels > 0.5km in this time

The Therac-25 Accident

- Therac-25 is a medical linear accelerator
- High-energy X-ray and / or electron beams destroy tumors
- Six people died between 1985 and 1987



N. Levenson, C.Turner, *An Investigation of the Therac-25 Accidents*, IEEE Computer, 1993
N. Levenson, *Safeware,: System Safety and Computers*, Addison Wesley 1995

Analysis of the Therac-25 Accident

- **Changed requirements** were not considered
 - In Therac-25, software is safety-critical (in contrast to Therac 20)
- **Design** is too complex
 - Concurrent system, shared variables (race conditions)
- **Code** is buggy
 - Check for changes done at wrong place
- **Testing** was insufficient
 - System test only, almost no separate software test
- **Maintenance** was poor
 - Correction of bug instead of re-design (root cause)

Boeing finds another software problem on the 737 Max

It's at least the third flaw found since the plane was grounded last year

By Sean O'Kane | @sokane1 | Feb 6, 2020, 12:42pm EST



Listen to this article

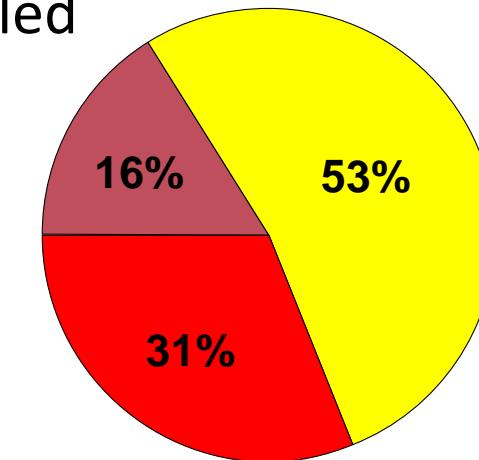


SHARE



Software – a Poor Track Record

- Software bugs cost the U.S. economy an estimated \$59.5 billion annually, or about 0.6 percent of the gross domestic product
- 84% of all software projects are **unsuccessful**
 - Late, over budget, less features than specified, cancelled
- The average unsuccessful project
 - 222% longer than planned
 - 189% over budget
 - 61% of originally specified features



1. Introduction

1.1 Software Failures

1.2 Challenges

1.3 Clean Code

1.4 Solution Approaches (Course Outline)

Why is Software so Difficult to Get Right?

Complexity

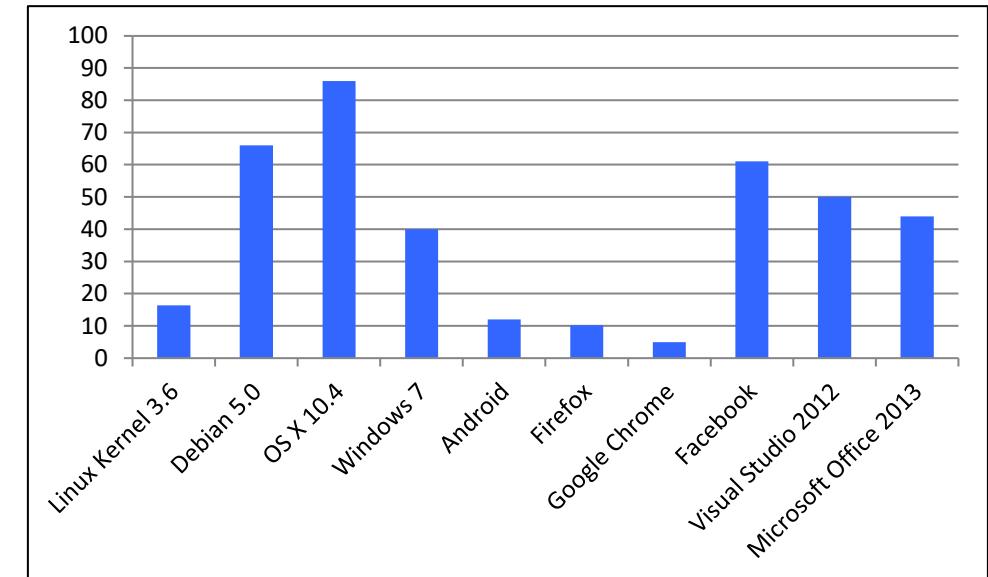
Change

Competing
Objectives

Constraints

Complexity

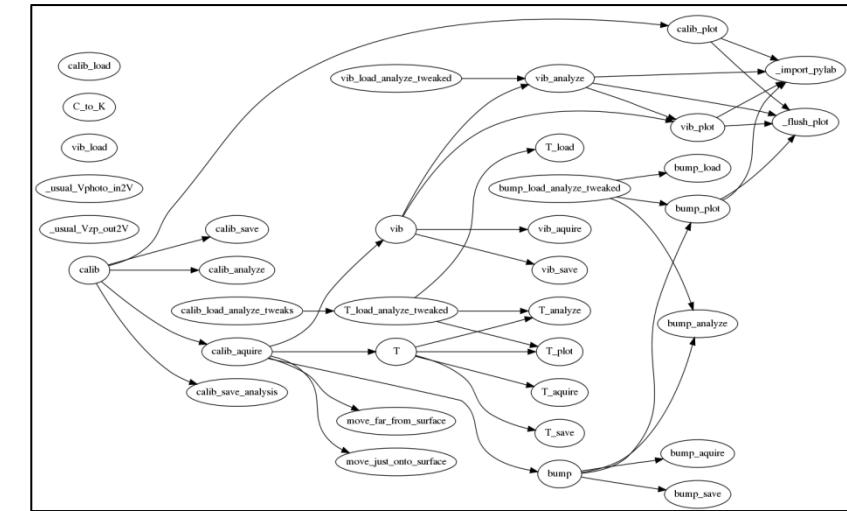
- Modern software systems are huge
 - Created by many developers over several years
- They have a very high number of:
 - Discrete states (infinite if the memory is not bounded)
 - Execution paths (infinite if the system may not terminate)



Size of software systems in MLOC

Complexity (cont'd)

- Small programs tend to be simple

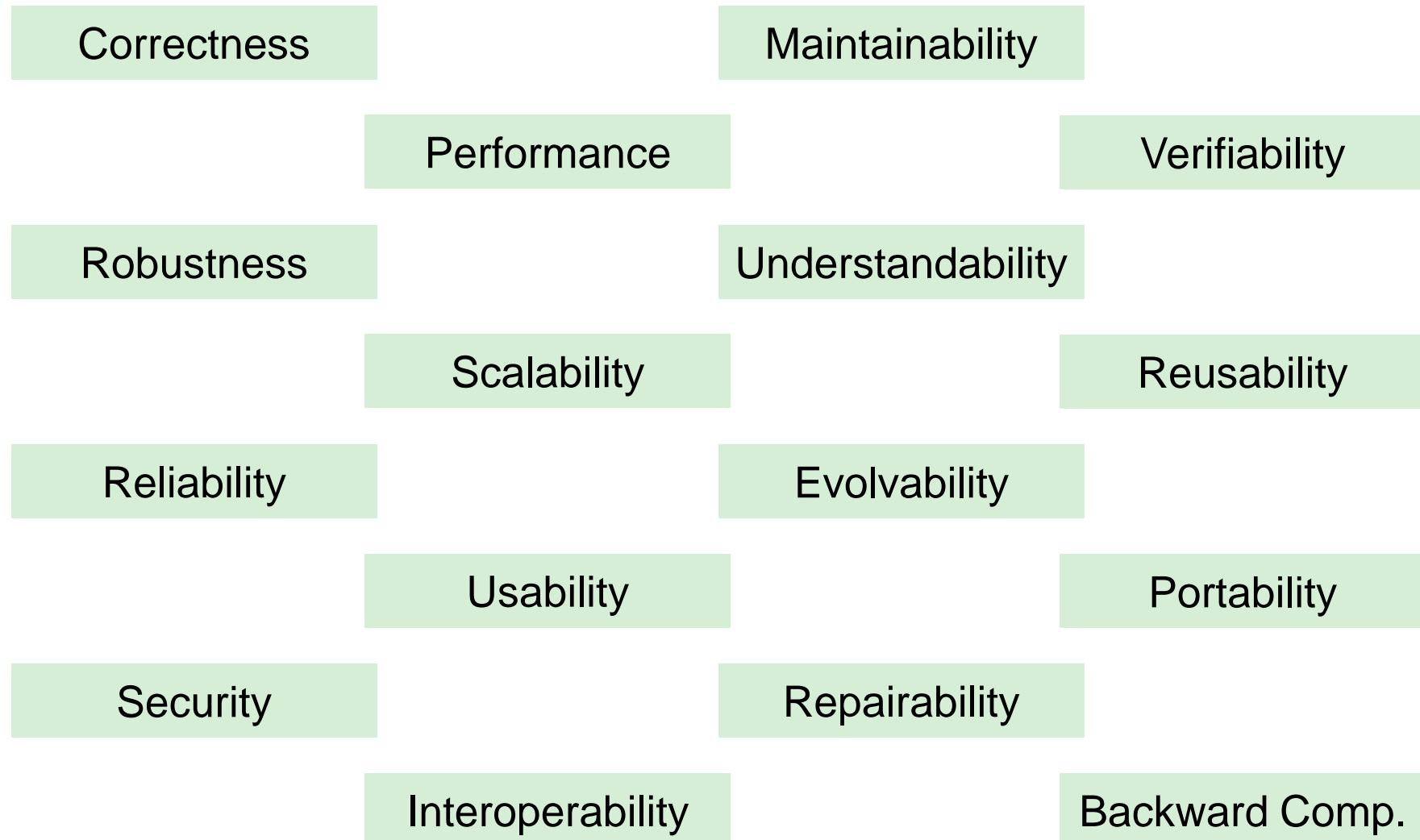


- Big programs tend to be complex
 - Complexity grows worse than linearly with size

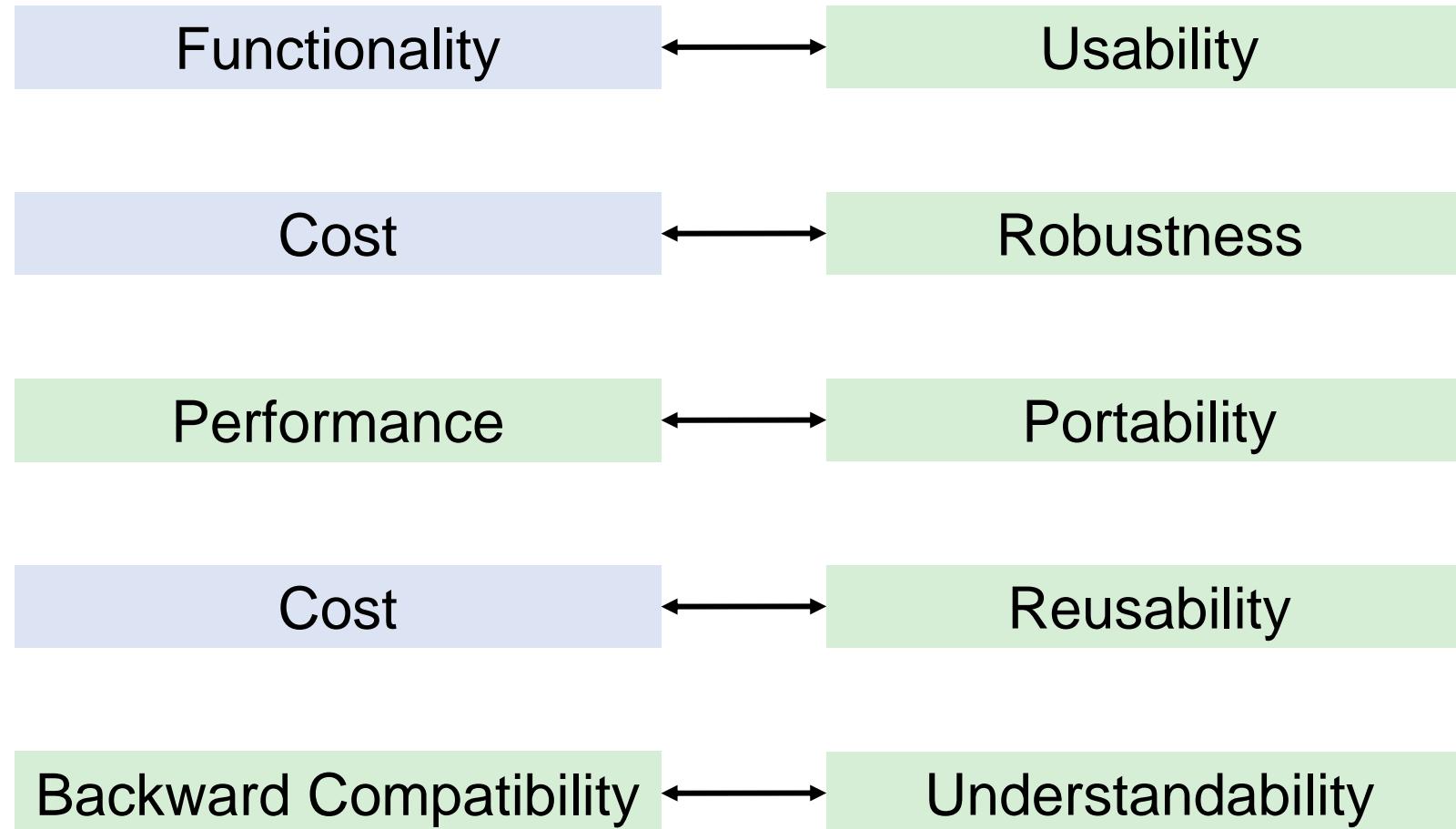
Change

- Since software is (*perceived as being*) easy to change, software systems often deviate from their initial design
- Typical changes include
 - New features (requested by customers or management)
 - New interfaces (new hardware, new or changed interfaces to other software systems)
 - Bug fixing, performance tuning
- Changes often **erode** the structure of the system

Competing Objectives: Design Goals



Competing Objectives: Typical Trade-Offs



Constraints

Software development (like all projects) is constrained by limited resources

- Budget
 - Marketing,
management priorities
- Time
 - Market opportunities,
external deadlines
- Staff
 - Available skills



Software Engineering

Complexity

Change

Competing
Objectives

Constraints

A collection of techniques, methodologies, and tools that help with the production of

- a high quality software system
- with a given budget
- before a given deadline
- while change occurs

[Brügge]

1. Introduction

1.1 Software Failures

1.2 Challenges

1.3 Clean Code

1.4 Solution Approaches (Course Outline)

WHAT IS CLEAN CODE?

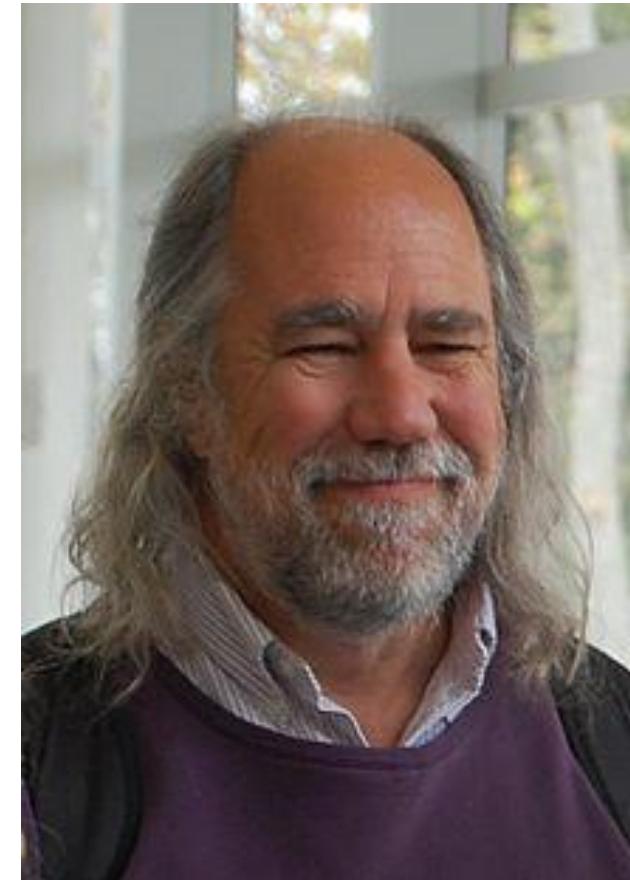
Bjarne Stroustrup, inventor of C++, author of *The C++ Programming Language*

I like my code to be elegant and efficient. The logic should be straightforward to **make it hard for bugs to hide**, the **dependencies minimal** to ease maintenance, **error handling complete** according to an articulated strategy and **perfomance close to optimal** so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does **one thing** well.



Grady Booch author of *Object Oriented Analysis and Design with Applications*

Clean code is **simple** and direct. Clean code
reads like well-written prose. Clean code
never obscures the designers intent but
rather is full of **crisp abstractions** and
straightforward lines of control.



Dave Thomas founder of OTI, godfather of the *Eclipse* strategy

Clean code **can be read** and enhanced by a developer other than its original author. It has **unit and acceptance tests**. It has meaningful names. It provides one way rather than many ways for **doing one thing**. It has **minimal dependencies**, which are explicitly defined, and provides a clear and **minimal API**. Code should be **literate** since depending on the language, not all necessary information can be expressed clearly in code alone.



Ron Jeffries

author of *Extreme Programming Installed*

In recent years, I begin and nearly end, with Beck's rules of simple code. In priority order, simple code

- Runs all the **tests**
- Contains **no duplication**
- Expresses all the **design ideas** that are in the system
- **Minimizes the number of entities** such as classes, methods, functions, and the like

(...)



Ward Cunningham

inventor of Wiki, coinventor of eXtreme Programming (...)

You know you are working on clean code
when each routine you read turns out to be
pretty much **what you expected**. You can
call it beautiful code when the code also
makes it **look like the language was made
for the problem**.



1. Introduction

1.1 Software Failures

1.2 Challenges

1.3 Clean Code

1.4 Solution Approaches (Course Outline)

Course Outline

- We will study several key **principles** for creating code
- We will cover both **established practices** and **innovative approaches**
- We will emphasize **software reliability and correctness techniques**

Part I: Writing clean code [2]

Modularity, Coupling, Design Patterns

Part II: Software Testing [2+1]

Metrics, exhaustive, random, functional, structural testing.

Part III: Software Analysis at Scale [13+1]

Math, heap, numerical, symbolic execution, concolic execution, fuzzing

Part IV: Modeling [3+1]

Model finding, Alloy, applications to memory models

Lectures

Almost all lectures will be given by Prof. Martin Vechev

However, the first two lectures after this one will be taught by Benjamin Bichsel and Samuel Steffen (Part I).

We also have 3 planned guest lectures so to learn from best practices in the field, including CTOs of successful ETH spin-offs (former Google Tech Leads), and industry labs, connecting to topics in the course.

Project

- There will be a project to help you master the techniques introduced in lectures:

Implement a component of an automated static analyzer to find bugs

- Done in a group of 2 or 3, never solo
 - **Select your team soon** (watch for announcements, about in week 4)
- Details will be explained later

Organization of the Course

■ Prerequisites

- Course is **self-contained**
- But it combines well with other courses:
 - Formal Methods and Functional Programming
 - Compiler Design
 - Software Engineering Seminar

■ Grading

- 20% project
- 80% final exam

Course Infrastructure

- Web page: <https://www.sri.inf.ethz.ch/teaching/rse2021>
 - All up-to-date info is there: lectures, exercises, slides, etc.
- Lectures are pre-recorded
- Moodle Forum (link on course homepage)
- If you have questions, contact your assistants first

Rigorous Software Engineering

Prof. Martin Vechev



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Motivation for Material

- Programs are everywhere
 - e.g., cars, smart phones, mobile apps, software-defined networks, spreadsheets, probabilistic programs, etc.
- Hard to manually reason about **security and reliability** of programs: drivers, security of blockchain smart contracts, reliable artificial intelligence
- Defects have a massive economic effect
 - ~60 billion USD annually and growing

Wanted: Automated techniques for ensuring security and reliability of programs and computations.

Question

Can you build an **automatic** analyzer which takes as input an **arbitrary** program and an **arbitrary** property such that if the analyzer answers:

- “**Yes**”, then it is certain that the property holds
- “**No**”, then it is certain that the property does not hold

?

Question

Can you build an **automatic** analyzer which takes as input an **arbitrary** program and an **arbitrary** property such that if the analyzer answers:

- “**Yes**”, then it is certain that the property holds
- “**No**”, then it is certain that the property does not hold



Answer:

No. The problem is undecidable

Alan Turing

What now?

Change the question

New Question

Can you build an **automatic** analyzer which takes as input an **arbitrary** program and an **arbitrary** property such that if the analyzer answers:

- “**Yes**” , then it is certain that the property holds
unknown if the property holds or not
- “**No**” , then it is ~~certain that the property does not hold~~

?

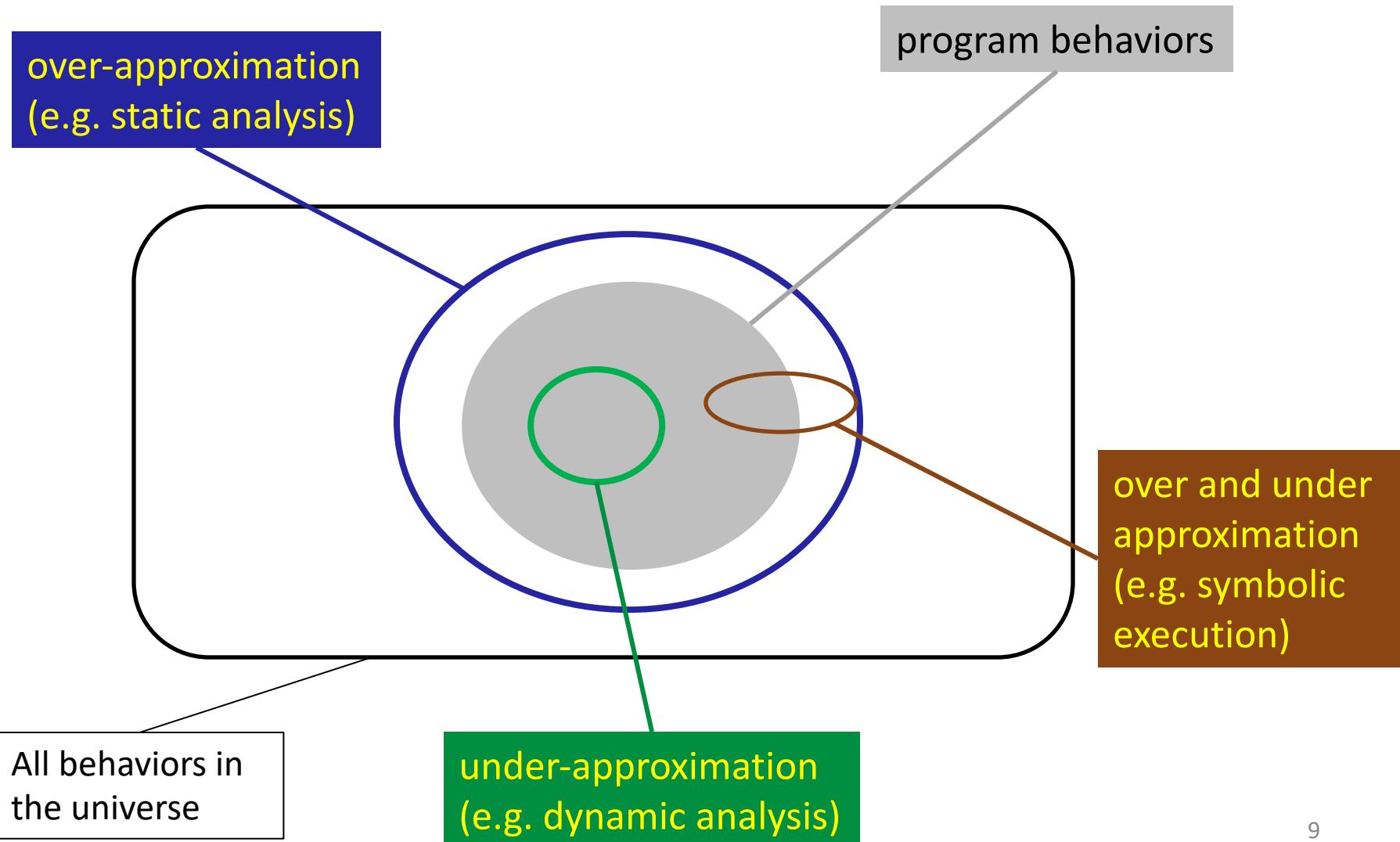
Trivial Solution

```
StaticAnalyzer(Program, Property)
{
    return "No";
}
```

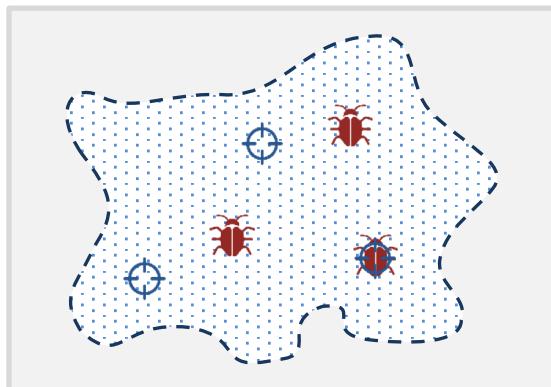
Static Program Analysis: Challenge

The challenge is to build a static analyzer that is able to answer “Yes” for as many programs as possible that satisfy the property.

Approaches to Program Analysis

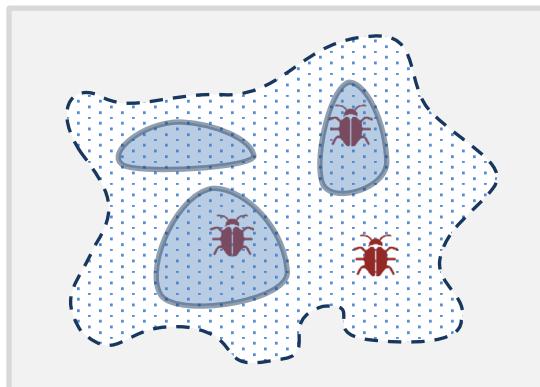


Another view...



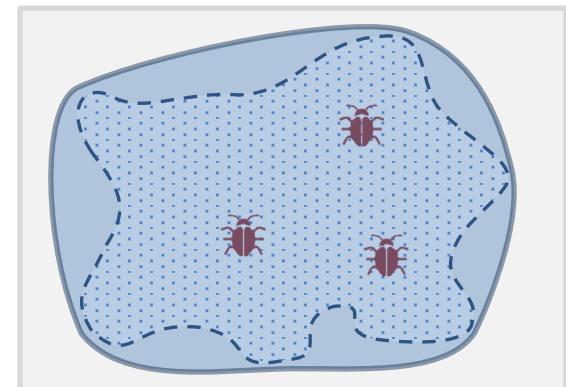
Testing

Report true bugs
Can miss bugs



Dynamic (symbolic)
analysis

Report true bugs
Can miss bugs



Automated verification

Can report false alarms
No missed bugs

Plan for Today

- Informal introduction to static program analysis
- Goal: get an intuition
- Understand **why** we need the math later

Static Program Analysis: cool facts

- Can automatically **prove** interesting properties such as
 - absence of null pointer dereferences, assertions at a program point, termination, absence of data races, information flow,...
- Can automatically find bugs in large scale programs, or detect bad patterns
 - for instance: the program fails to call the API “close” on a File object
 - or detect stylistic patterns..
 - **what else?**
- Nice combination of math and system building
 - combines program semantics, data structures, discrete math, logic, algorithms, decision procedures, ...

Static Program Analysis: cool facts

- Can run the program **without** giving a concrete input
 - abstractly execute a piece of code from any point
- **No need for manual annotations** such as loop invariants
 - they are automatically inferred
 - what is a loop invariant ?

Lets look at a couple of examples what static analysis can do for us...

What is the result of this program ?

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
  if (n>100) then
    r = n-10;
  else
    t1 = n + 11;
    t2 = MC(t1);
    r = MC(t2);
  endif;
end

var a:int, b:int;
begin
  b = MC(a);
end
```

The McCarthy 91 function: if ($n \geq 101$) then $n - 10$ else 91

```
proc MC(n:int) returns (r:int)
var t1:int, t2:int;
begin
  if (n>100) then
    r = n-10;
  else
    t1 = n + 11;
    t2 = MC(t1);
    r = MC(t2);
  endif;
end

var a:int, b:int;
begin
  b = MC(a);
end
```

Invariants per program point
(automatically computed):

top

$n-101 \geq 0$

$-n+r+10 = 0; n-101 \geq 0$

$-n+100 \geq 0$

$-n+t1-11 = 0; -n+100 \geq 0$

$-n+t1-11 = 0; -n+100 \geq 0;$
 $-n+t2-1 \geq 0; t2-91 \geq 0$

$-n+t1-11=0; -n+100 \geq 0; -n+t2-1 \geq 0;$
 $t2-91 \geq 0; r-t2+10 \geq 0; r-91 \geq 0$

$-n+r+10 \geq 0; r-91 \geq 0$

top

$-a+b+10 \geq 0; b-91 \geq 0$

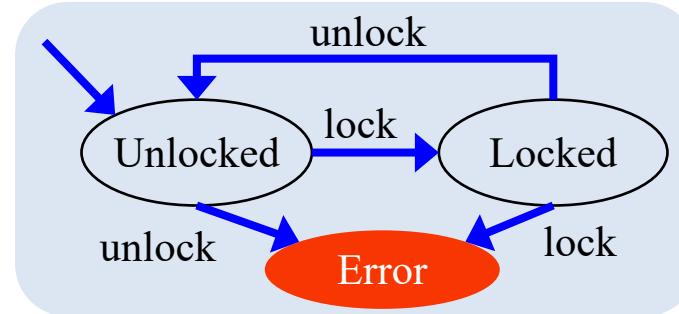
Automatically inferred using numerical abstract domains

Network driver: does it do what it is supposed to?

```
driver(int req) {  
  
1: initlock();  
2: lock();  
3: old = packets;  
  
4: if (req) {  
5:   req = req->next;  
6:   unlock();  
7:   packets++;  
}  
  
8: if (packets != old)  
    goto 2;  
  
9: unlock();  
}
```

```
enum {Locked,Unlocked}  
initlock() {s = Unlocked;}  
lock() {  
  if (s == Locked) abort;  
  else s = Locked; }  
unlock() {  
  if (s == Unlocked) abort;  
  else s = Unlocked; }
```

Property we want to check:

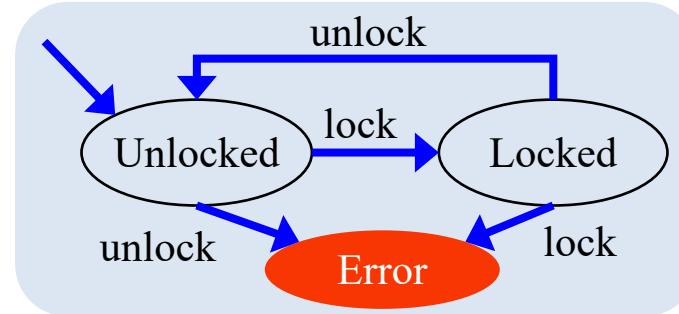


Network driver: does it do what it is supposed to?

```
driver(int req) {  
  
1: initlock();  
2: lock();  
3: old = packets;  
  
4: if (req) {  
5:   req = req->next;  
6:   unlock();  
7:   packets++;  
}  
  
8: if (packets != old)  
    goto 2;  
  
9: unlock();  
}
```

```
enum {Locked,Unlocked}  
initlock() {s = Unlocked;}  
lock() {  
  if (s == Locked) abort;  
  else s = Locked; }  
unlock() {  
  if (s == Unlocked) abort;  
  else s = Unlocked; }
```

Property we want to check:



Automatically verified using SLAM based on predicate abstraction

Real-World Program Analyzers (small sample)

A significantly increased interest in the last few years



Microsoft®
Research
Klocwork

TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS

ACM.org | Join ACM | About Communications | ACM Resources | Alerts & Feeds

SIGN IN

COMMUNICATIONS
OF THE ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRACTICE | CAREERS | MAGAZINE ARCHIVE

Home / Magazine Archive / February 2010 (Vol. 53, No. 2) / A Few Billion Lines of Code Later: Using Static Analysis... / Full Text

CONTRIBUTED ARTICLES

A Few Billion Lines of Code Later: Using Static Analysis to
Find Bugs in the Real World



PANAYA
Making ERP Easy

Facebook: last 2 years...

INFER: <http://fbinfer.com/>



Infer is an automated code review tool based on deep program analysis.

It is fully integrated and is a key part of FB's software development process. Fast and precise, it fits FB's culture on move fast and break things 😊

Flow: <http://flowtype.org/>

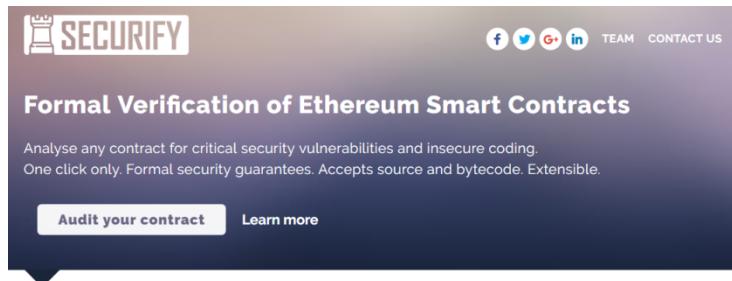


Flow is a static type checker and can automatically infer types of JavaScript programs, both user code and libraries.

Both tools are open sourced and are used by companies beyond Facebook.

Security of Blockchain

Securify: <http://securify.ch>

A screenshot of the Securify interface showing a Solidity code editor. The code is for a "SimpleBank" smart contract, containing functions for depositing and withdrawing funds. The interface includes tabs for "Solidity", "Bytecode", and "Address".

```
1 contract SimpleBank {
2
3     mapping(address => uint) balances;
4
5     function deposit(uint amount) {
6         balances[msg.sender] += amount;
7     }
8
9     function withdraw() {
10        msg.sender.call.value(balances[msg.sender])();
11        balances[msg.sender] = 0;
12    }
13
14 }
```

Securify is an automated static analyzer we developed in our lab which finds common errors in Ethereum smart contracts (or verifies their absence).

This analyzer is widely used by industry to perform audits of smart contracts automatically.

Security of Blockchain

VerX: <http://verx.ch>

The screenshot shows the VerX web application interface. At the top, there's a navigation bar with links for Documentation and API Docs. Below the navigation, a message says "Created by ChainSecurity AG, based on research from the ICE center, ETH Zurich." On the left, a sidebar menu is open under the "Specification" tab, showing a list of smart contracts including SafeERC20, RoentrancyGuard, FinalizableCrowdsale, RefundEscrow, RefundableCrowdsale, ERC20, MintorRole, MintedCrowdsale, ERC20Detailed, and SampleCrowdsale. The "Crowdsale" item is currently selected and highlighted in blue. The main content area displays a large block of Solidity code for the "SampleCrowdsale" contract. The code includes several properties and always clauses defining various states and transitions of the crowdsale, escrow, and wallet. At the bottom of the code area is a blue "Verify" button.

```
1 // The token distributed by the crowdsale is Token
2 property crowdsale_token_does_not_change {
3     always(SampleCrowdsale._token == SampleCrowdsaleToken);
4 }
5
6 // The escrow of the crowdsale does not change
7 property crowdsale_escrow_does_not_change {
8     always(SampleCrowdsale._escrow == RefundEscrow);
9 }
10
11 // The wallet of the crowdsale cannot be modified
12 property wallet_does_not_change {
13     always((SampleCrowdsale._wallet == RefundEscrow._beneficiary)
14         && (SampleCrowdsale._wallet == 0x55555555555555555555555555555555));
15 }
16
17 // The refunding state of the escrow is forever and cannot be changed
18 property escrow_refunding_state_is_forever {
19     always((prev(RefundEscrow._state) == 1) => (RefundEscrow._state == 1));
20 }
21
22 // The closed state of the escrow is forever and cannot be changed
23 property escrow_close_state_is_forever {
24     always((prev(RefundEscrow._state) == 2) => (RefundEscrow._state == 2));
25 }
26
27 // Refunds are possible only if the crowdsale is finalized
28 nonrevert refundable_only_when_crowdsale_is_finalized;
```

VerX is a deep automated static verifier we developed in our lab that can automatically verify Ethereum smart contracts for custom properties.

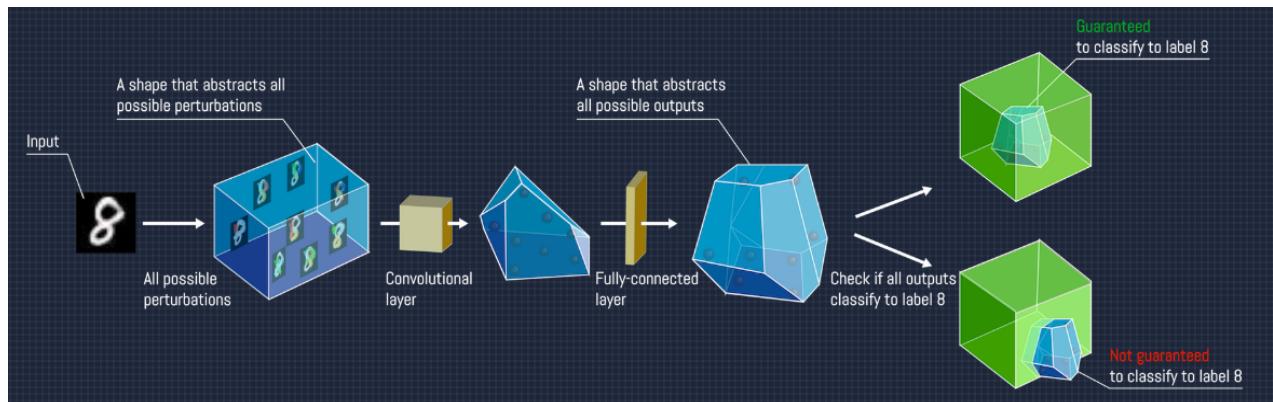
This verifier heavily used by industry to perform formally verified audits of smart contracts automatically.

Reliable Artificial Intelligence

<http://safeai.ethz.ch>

<https://github.com/eth-sri/eran>

Eran is an automated analyzer we built which ensures safety of deep learning. It uses numerical domains, like the ones taught in this Part II.



Questions

- What are the core principles behind these automated security and code analysis tools? Is there a general theory?
- What kind of properties/defects/security issues can these tools detect?
- How do you go about building one these tools?
- What concerns should be addressed to have such tools be practically adopted by software engineers?

Lets begin...

Static Analysis via Abstract Interpretation

- We will learn a style called **abstract interpretation**
 - a general theory of how to do approximation **systematically**
- Abstract interpretation is a very useful **thinking framework**
 - relate the concrete with the abstract, the infinite with the finite
- Many existing analyses can be seen as abstract interpreters
 - type systems, data-flow analysis, model checking, etc...

Abstract Interpretation: step-by-step

1. select/define an abstract domain
 - selected based on the type of **properties** you want to prove
2. define abstract semantics **for the language** w.r.t. to the domain
 - prove **sound** w.r.t **concrete semantics**
 - involves defining abstract transformers
 - that is, effect of statement / expression on the abstract domain
3. iterate abstract transformers over the abstract domain
 - until we reach a **fixed point**

The **fixed point** is the **over-approximation** of the program

Lets prove an assertion...

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
  
7: assert 0 ≤ x + y  
}
```

There are infinitely many executions here.
We cannot just enumerate them all.

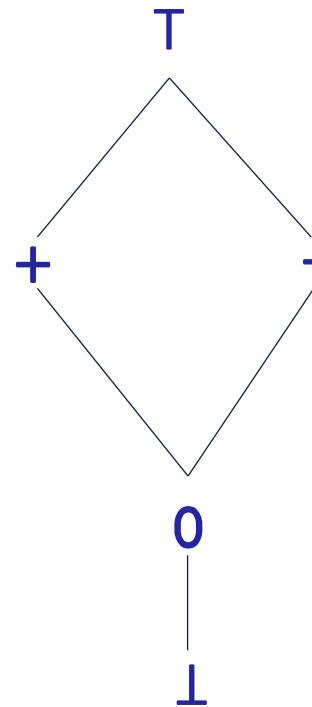
And even if they were finite, it would still
take us a long time to enumerate them
all...

Instead, let us do some over-
approximation, so that we can reduce the
space of what we need to enumerate...

Step 1: Select abstraction

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```

Lets pick the **sign** abstraction



Why this abstract domain ?

Question: what does + represent in the sign abstraction?

Question: what does + represent in the sign abstraction?

Answer: + represents all positive numbers and 0.

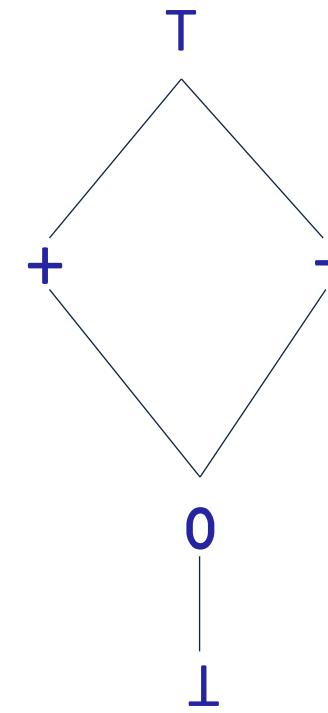
What about -, what does it mean ?

Step 1: Select abstraction

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
  
7: assert 0 ≤ x + y  
}
```

pc	x	y	i
2	+	⊥	T

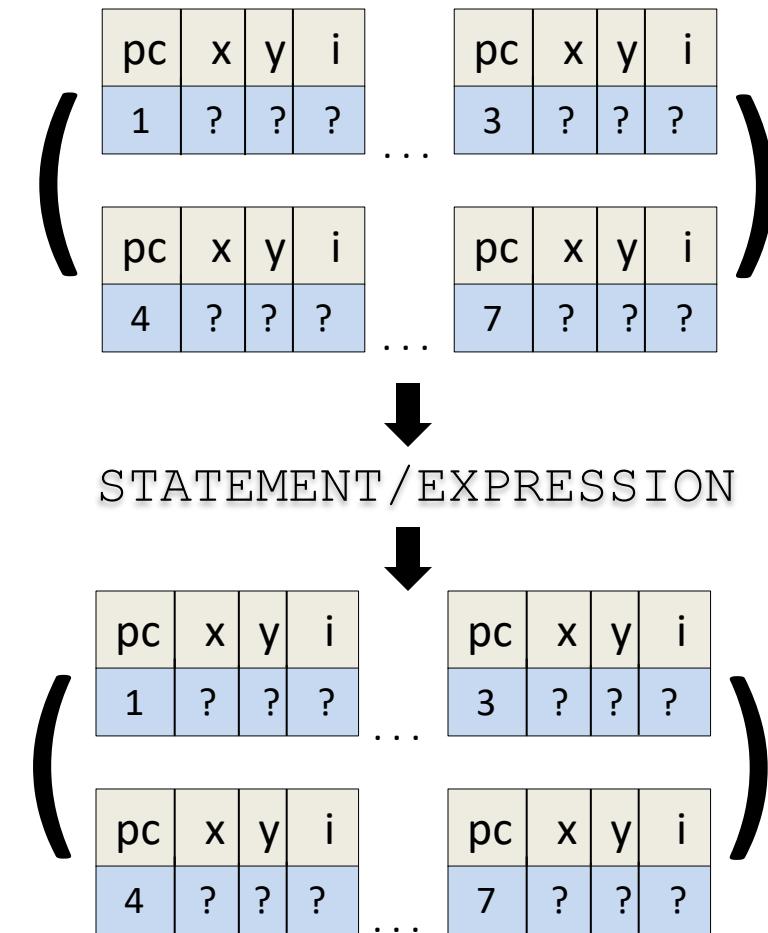
Example Abstract State



An **abstract** program state is a map from variables to **elements in the domain**

Step 2: Define Transformers

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



An **abstract transformer** describes the effect of statement and expression evaluation on an **abstract state**

Important Point

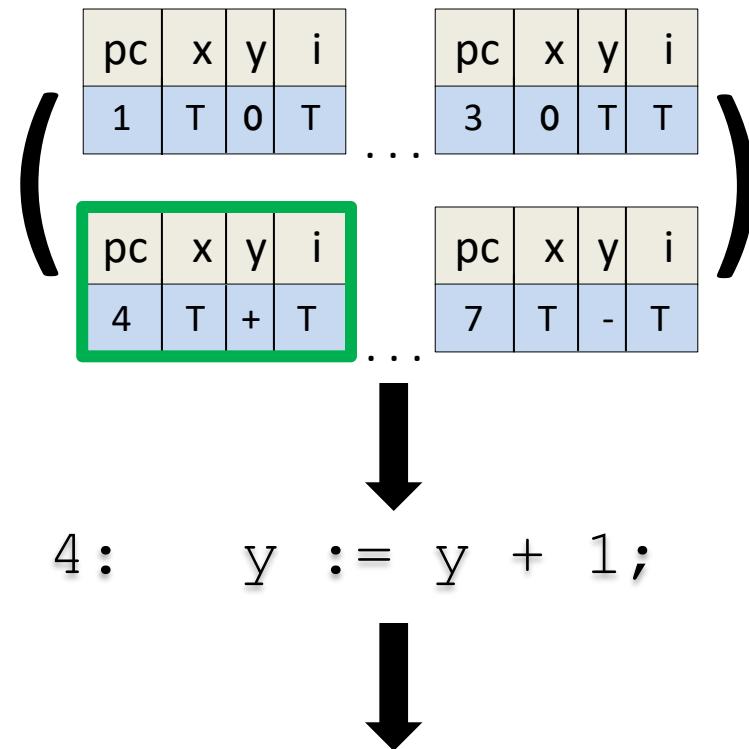
It is important to remember that abstract transformers are defined per **programming language** once and for all, and not per-program !

That is, they essentially define the new (abstract) semantics of the language (we will see them formally defined later)

This means that **any program** in the programming language can use the **same transformers**.

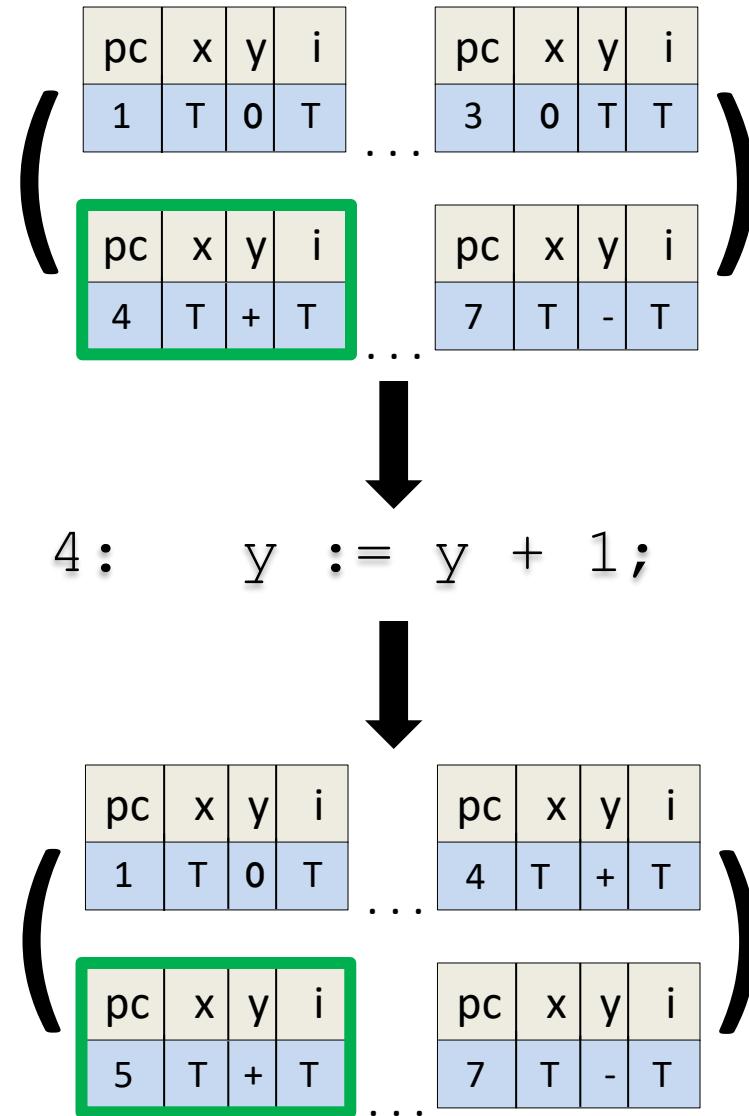
Step 2: Define Transformers

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



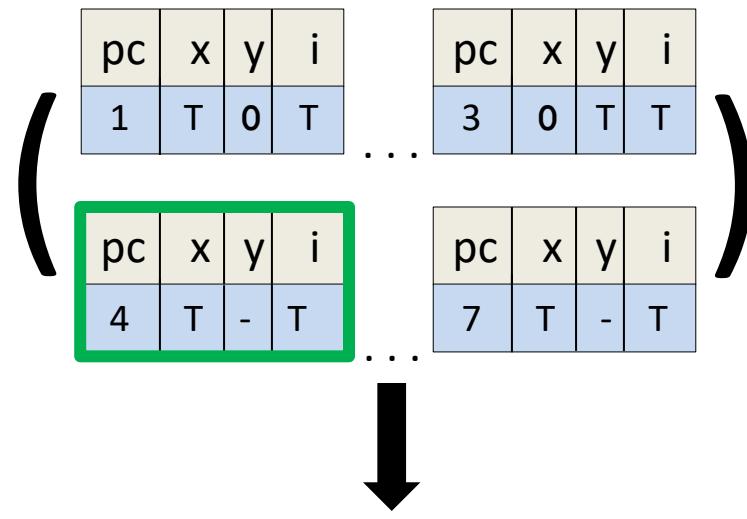
Step 2: Define Transformers

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



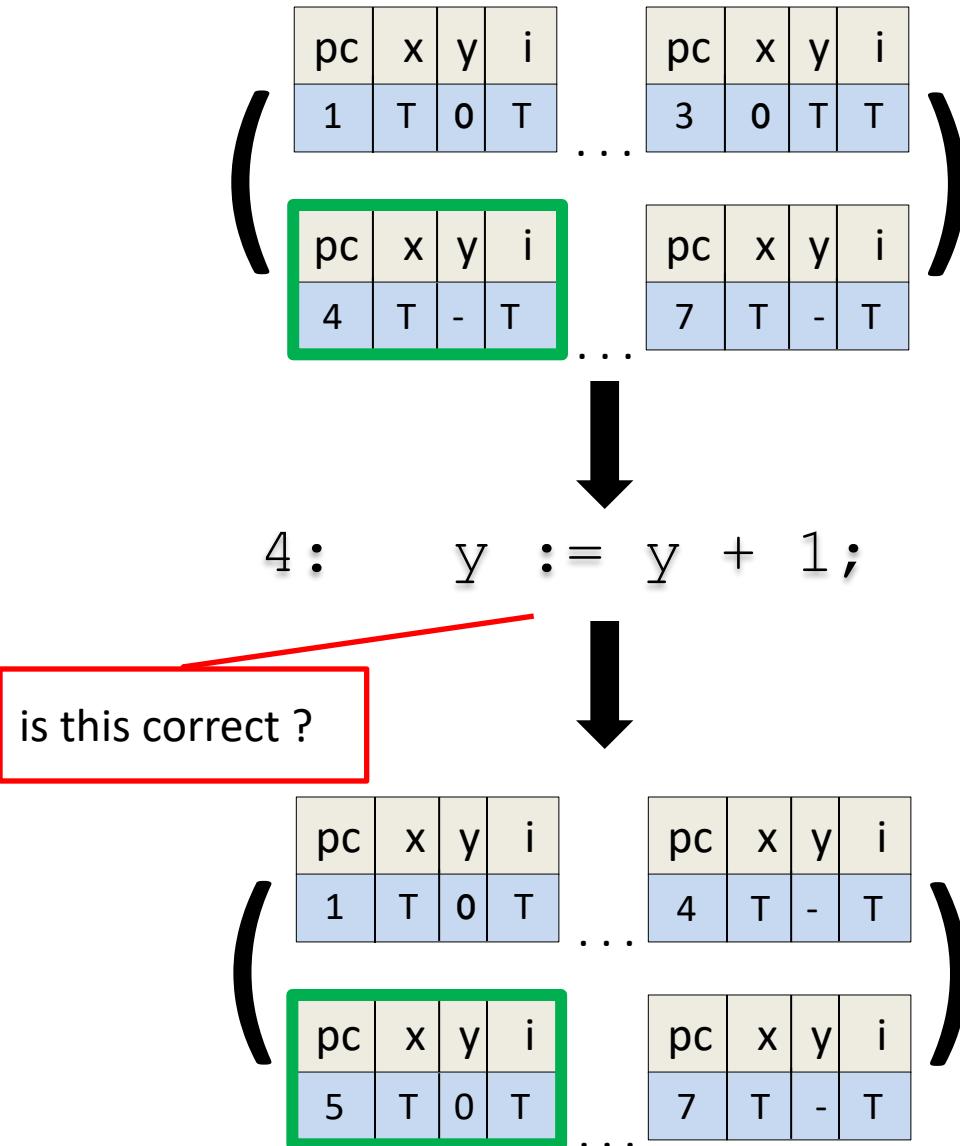
Step 2: Define Transformers

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



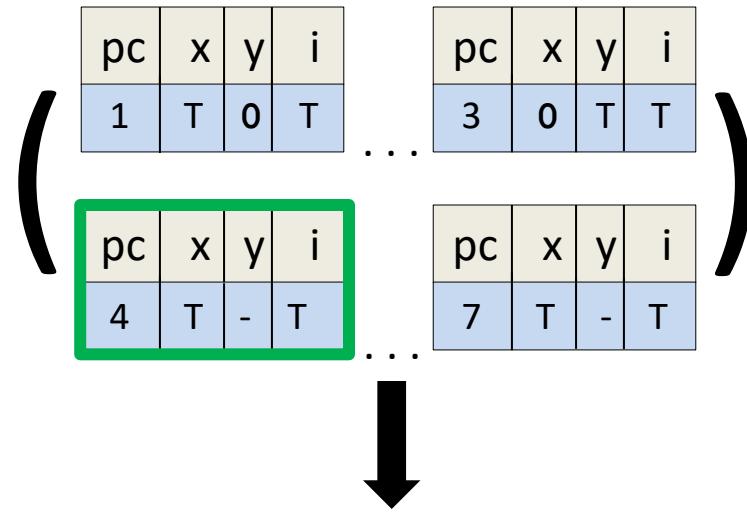
Step 2: Define Transformers

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



Step 2: Define Transformers

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



What does correct mean ?

Transformer Correctness

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
  
7: assert 0 ≤ x + y  
}
```

A **correct abstract transformer** should always produce results that are a **superset** of what a **concrete transformer** would produce

Unsound Transformer

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

This abstract state:

pc	x	y	i
4	T	-	T

represents **infinitely many** concrete states including:

pc	x	y	i
4	1	-3	2

If we perform $y := y + 1$ on this concrete state, we get:

pc	x	y	i
5	1	-2	2

However, the **abstract** transformer produced an abstract state:

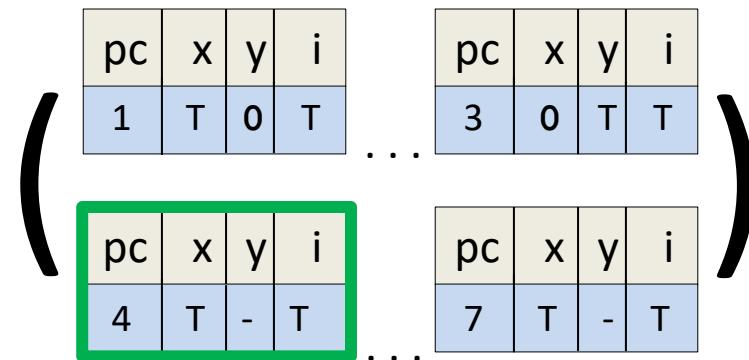
pc	x	y	i
5	T	0	T

This abstract state **does not** represent any state where $y = -2$

The abstract transformer is **unsound** ! 😞

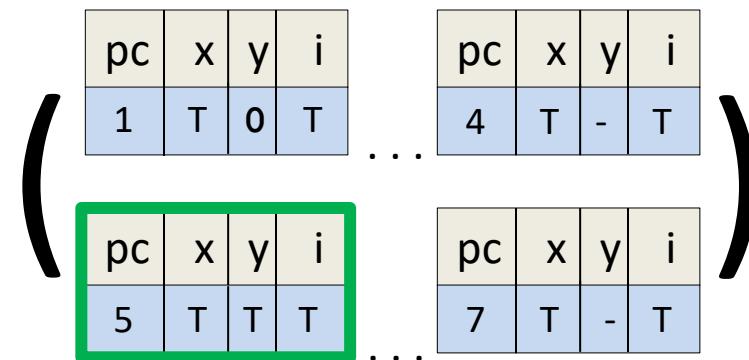
How about this ?

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



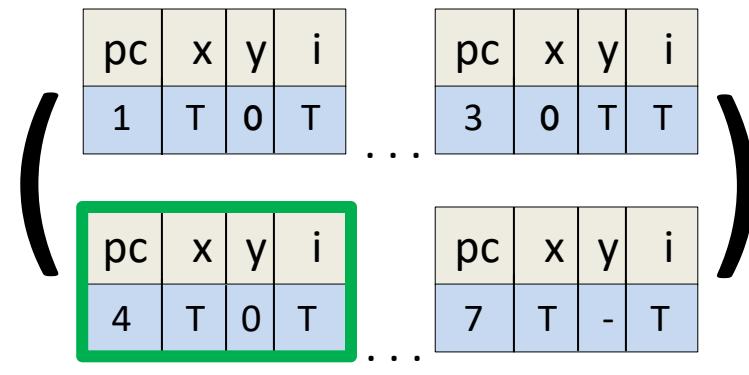
4 : y := y + 1;

This is correct, why?



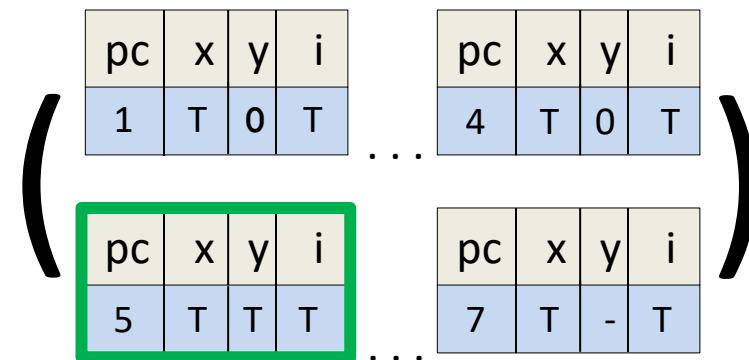
How about this ?

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



4 : y := y + 1;

Is this sound ? Yes
Is it precise ? No



Imprecise Transformer

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

This abstract state:

pc	x	y	i
4	T	0	T

represents **infinitely** many concrete states where y is always **0**, including:

pc	x	y	i
4	1	0	2

If we perform $y := y + 1$ on **any** of these concrete states, we will always get states where y **is always positive**, such as:

pc	x	y	i
5	1	1	2

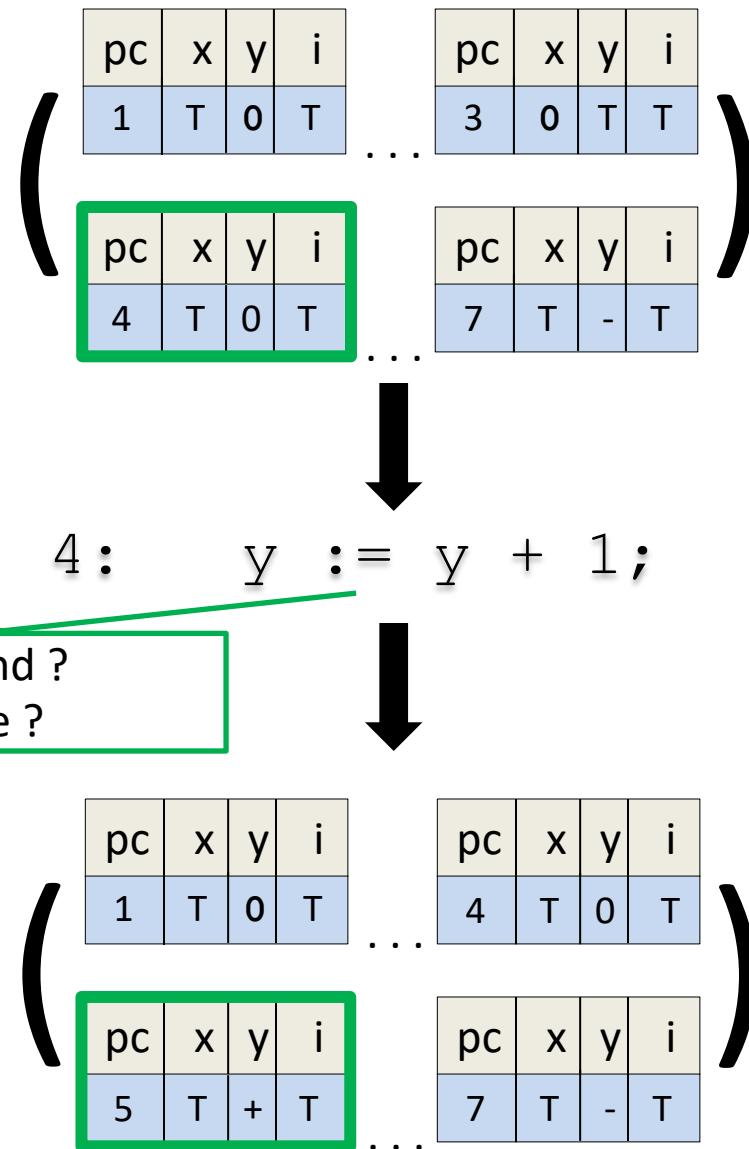
However, the **abstract** transformer produces an abstract state where y **can be any value**, such as:

pc	x	y	i
5	T	T	T

The abstract transformer is **imprecise** ! 😞

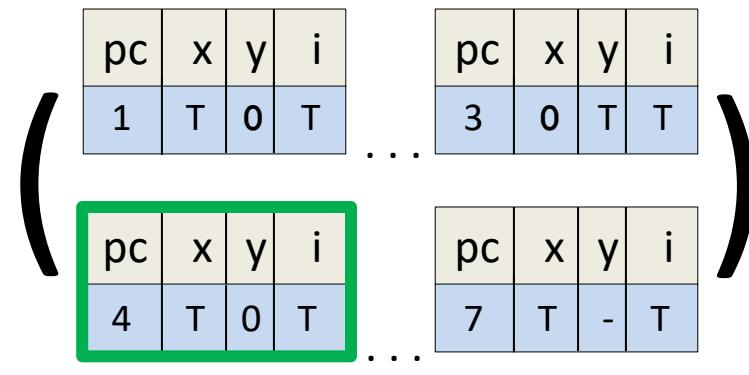
How about this ?

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



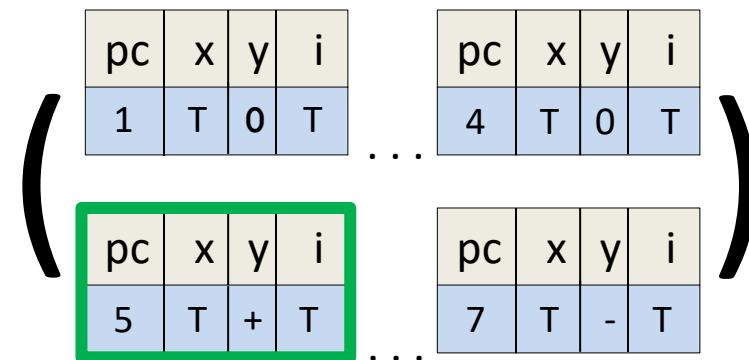
How about this ?

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



4 : y := y + 1;

Is this sound ? Yes
Is it precise ? Yes



Abstract Transformers

It is easy to be **sound** and **imprecise**: simply output T

It is desirable to be both **sound** and **precise**. If we lose precision, it needs to be clear why and where:

- sometimes, computing the most precise transformer (also called the **best transformer**) is **impossible**
- for efficiency reasons, we may sacrifice some precision

Step 3: Iterate to a fixed point

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```

Start with the **least** abstract element

pc	x	y	i
1	⊥	⊥	⊥
2	⊥	⊥	⊥
3	⊥	⊥	⊥
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥

Lets do some iterations...

Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

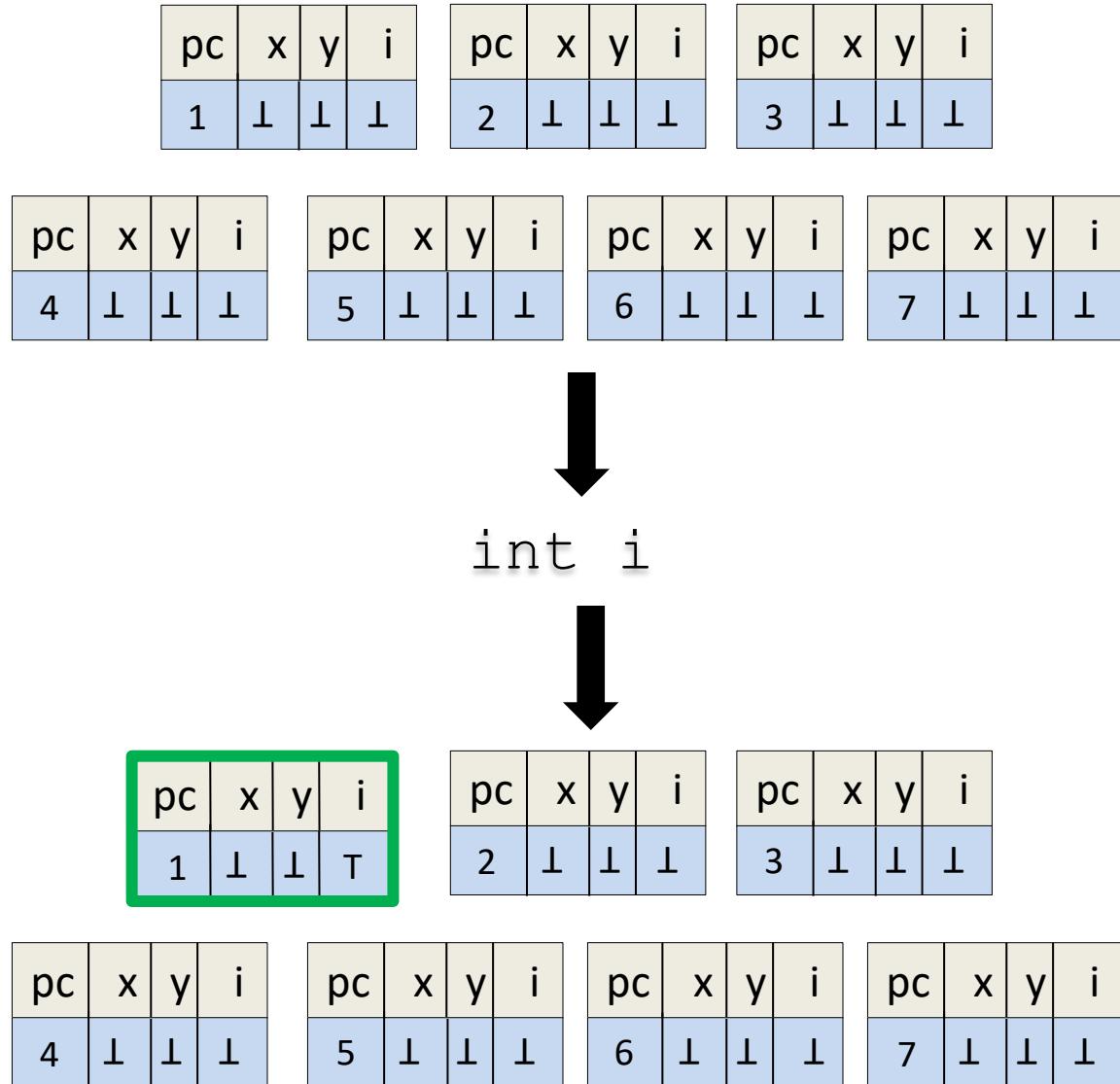
pc	x	y	i
1	⊥	⊥	⊥
2	⊥	⊥	⊥
3	⊥	⊥	⊥
pc	x	y	i
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥

↓
int i

↓

Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x :=5;  
    2: int y :=7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

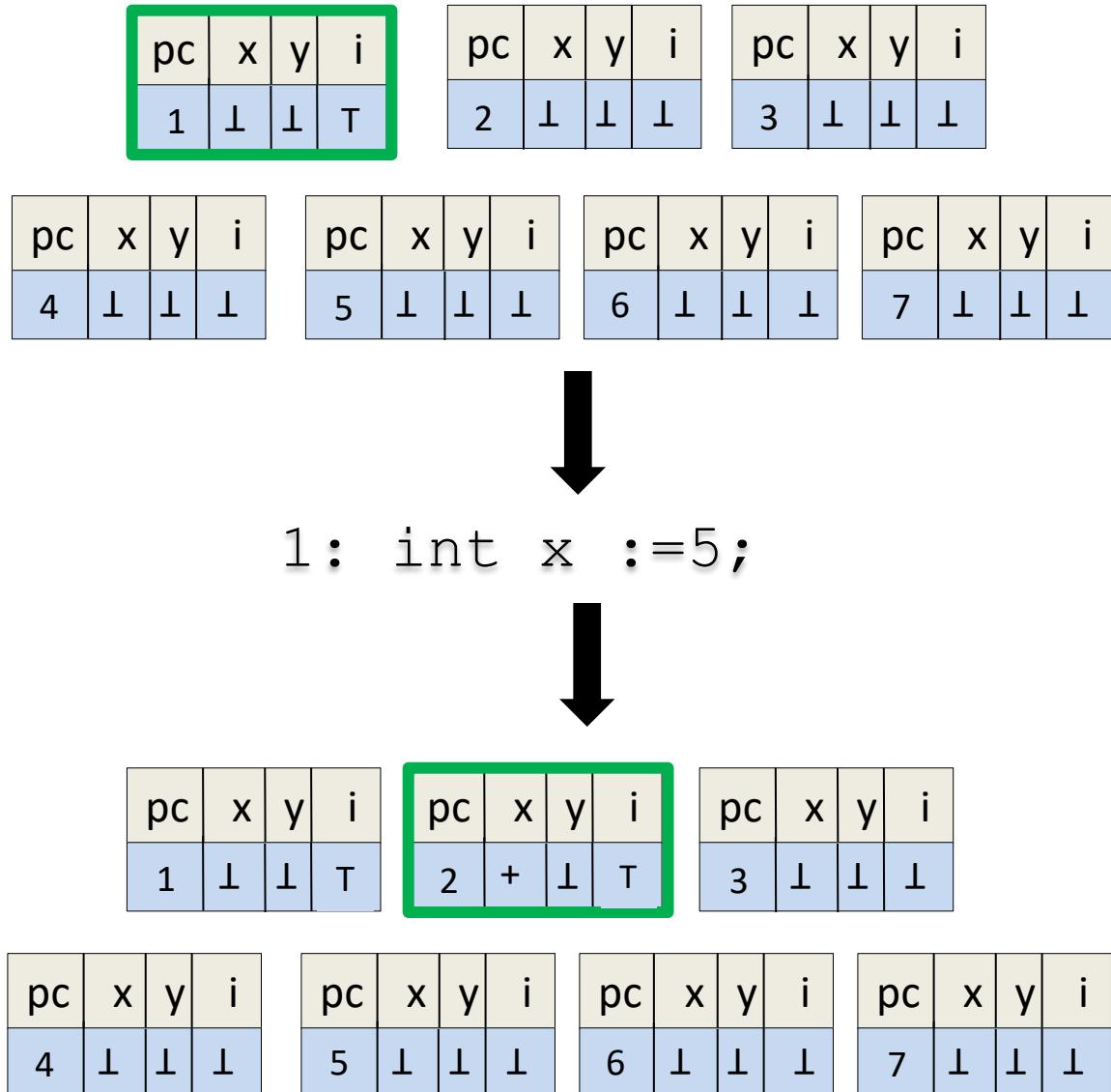
pc	x	y	i
1	⊥	⊥	⊤
2	⊥	⊥	⊥
3	⊥	⊥	⊥
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥

1: int x :=5;



Step 3: Iterate to a fixed point

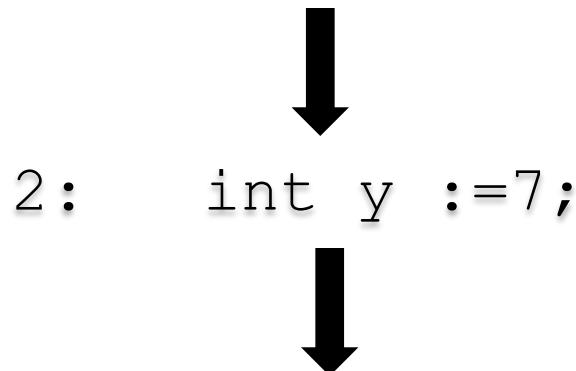
```
foo (int i) {  
    1: int x :=5;  
    2: int y :=7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



Step 3: Iterate to a fixed point

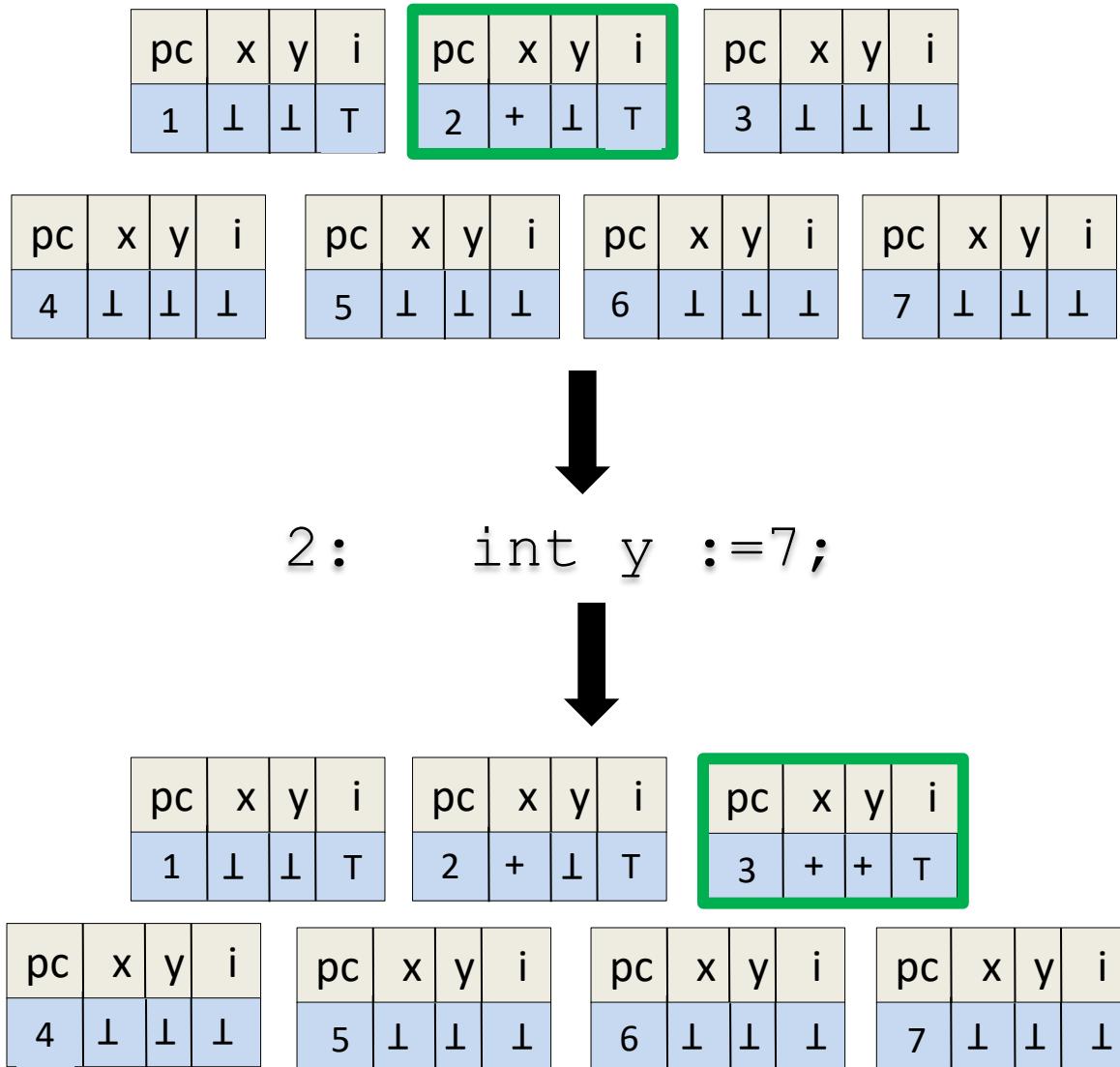
```
foo (int i) {  
    1: int x :=5;  
    2: int y :=7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	⊥	⊥	⊥
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥



Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

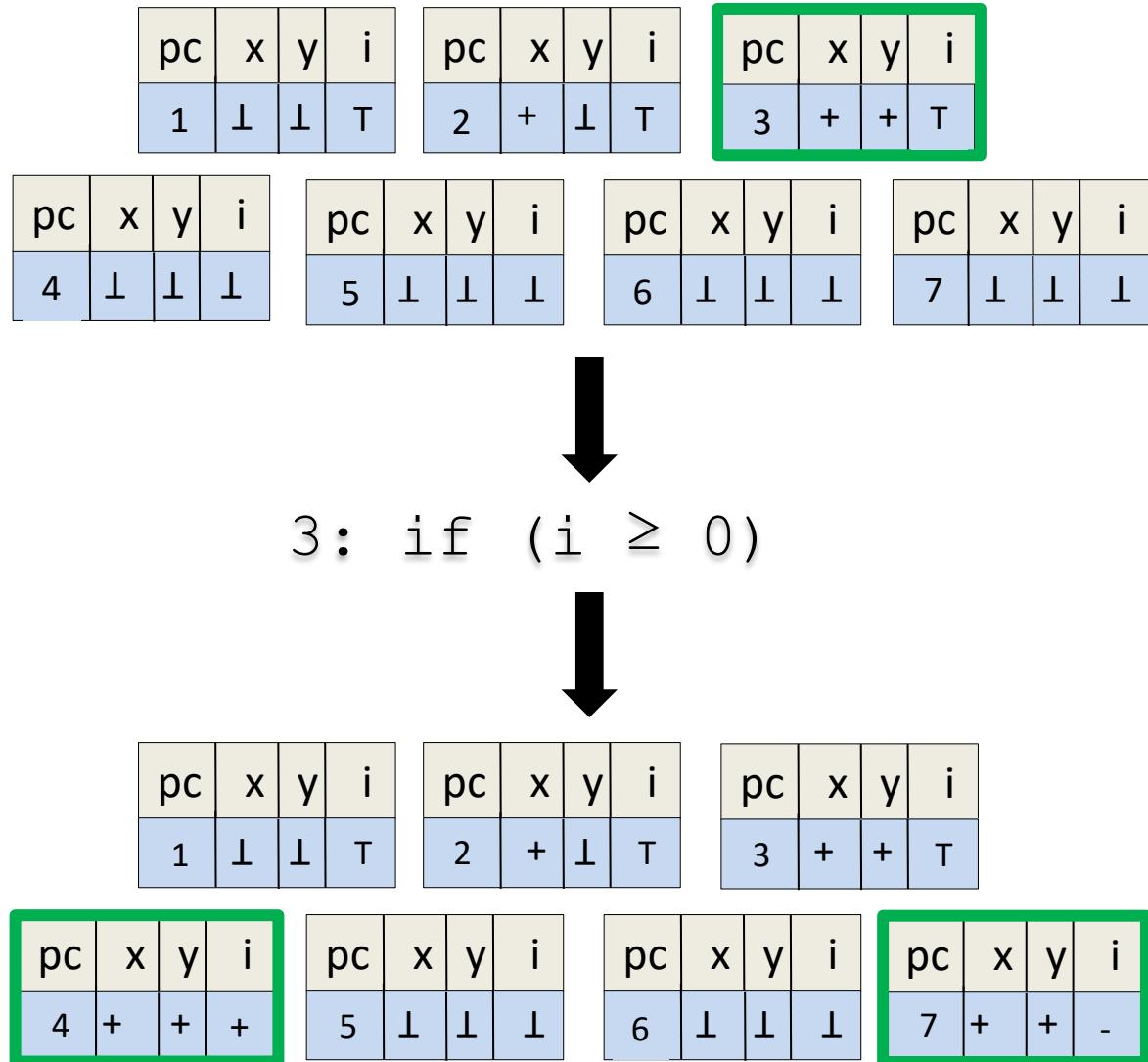
pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	⊥	⊥	⊥
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	⊥	⊥	⊥

3: if (i ≥ 0)



Step 3: Iterate to a fixed point

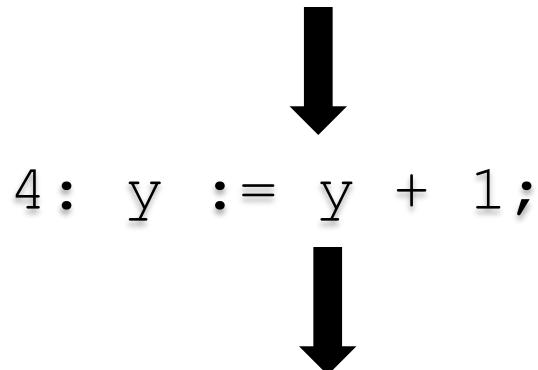
```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```



Step 3: Iterate to a fixed point

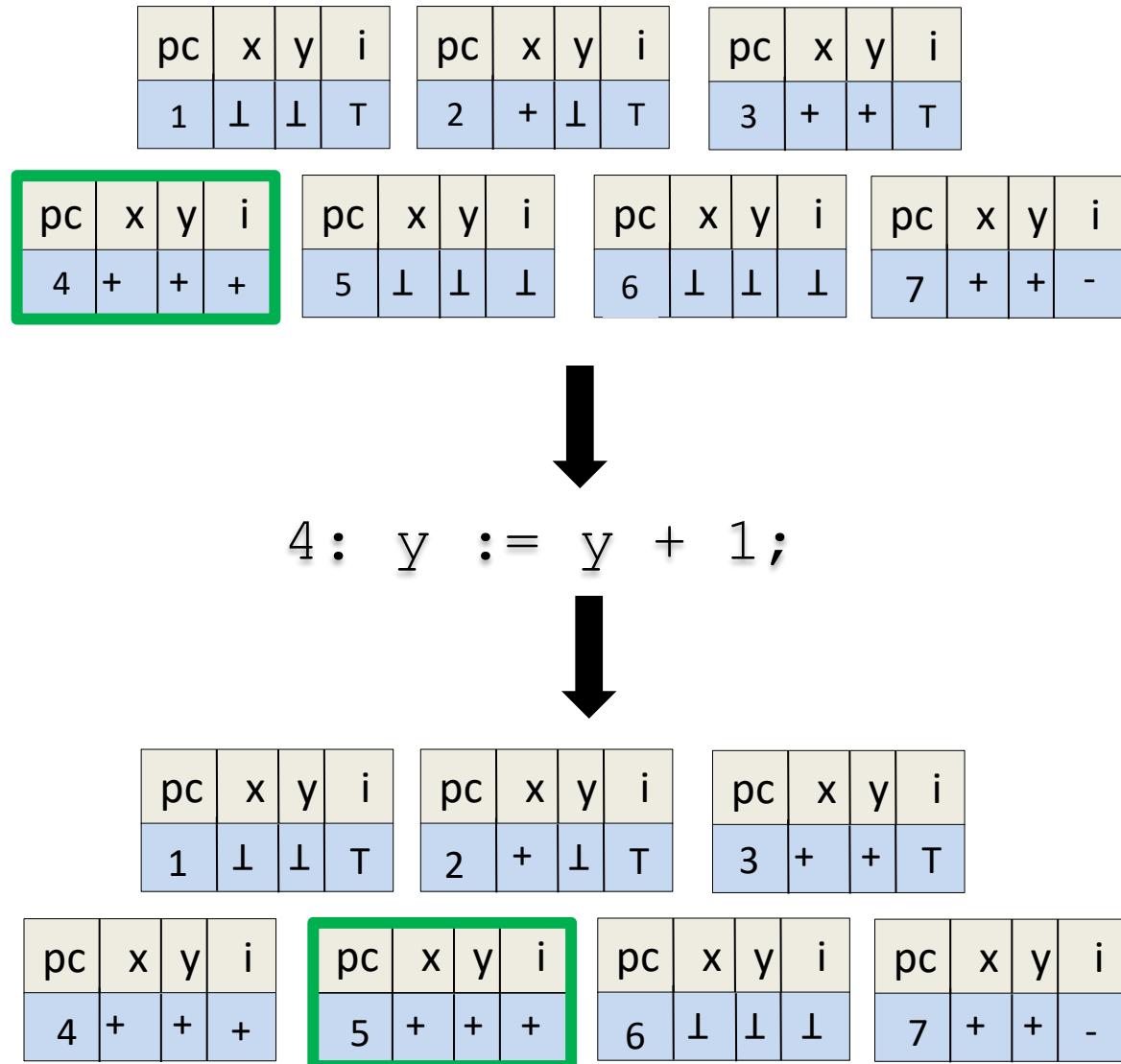
```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	+	+	+
5	⊥	⊥	⊥
6	⊥	⊥	⊥
7	+	+	-



Step 3: Iterate to a fixed point

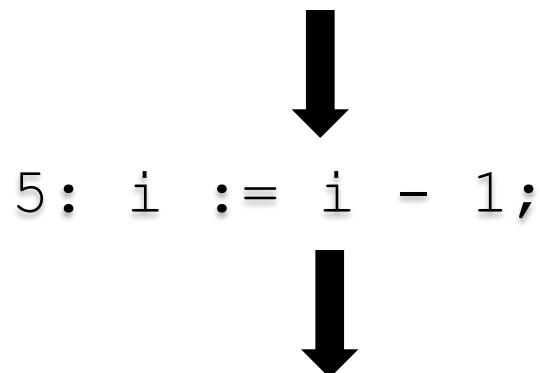
```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



Step 3: Iterate to a fixed point

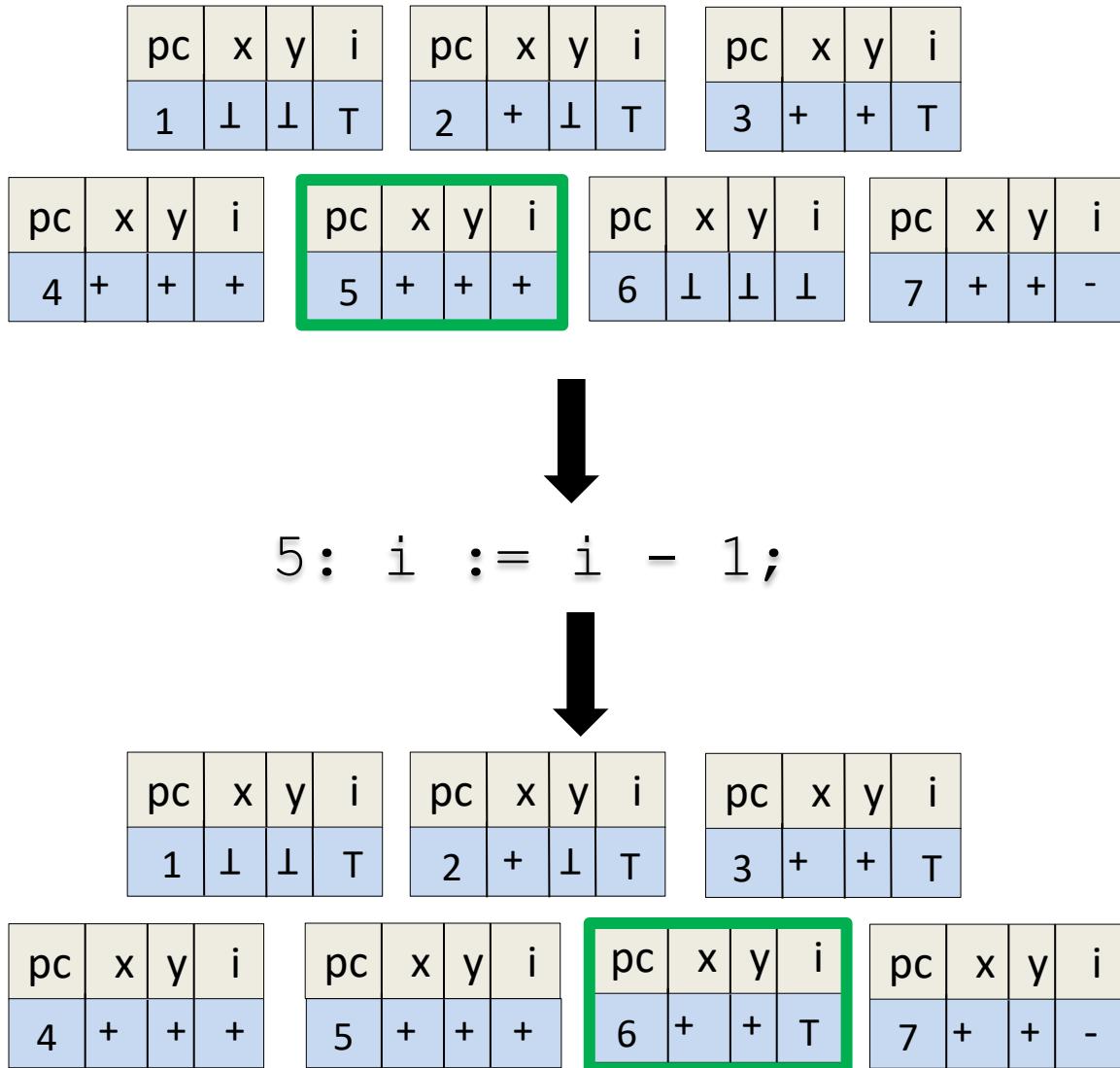
```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
pc	x	y	i
4	+	+	+
5	+	+	+
6	⊥	⊥	⊥
7	+	+	-



Step 3: Iterate to a fixed point

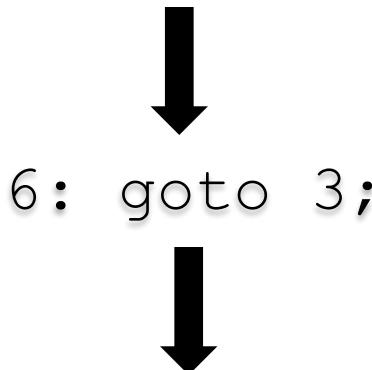
```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



Step 3: Iterate to a fixed point

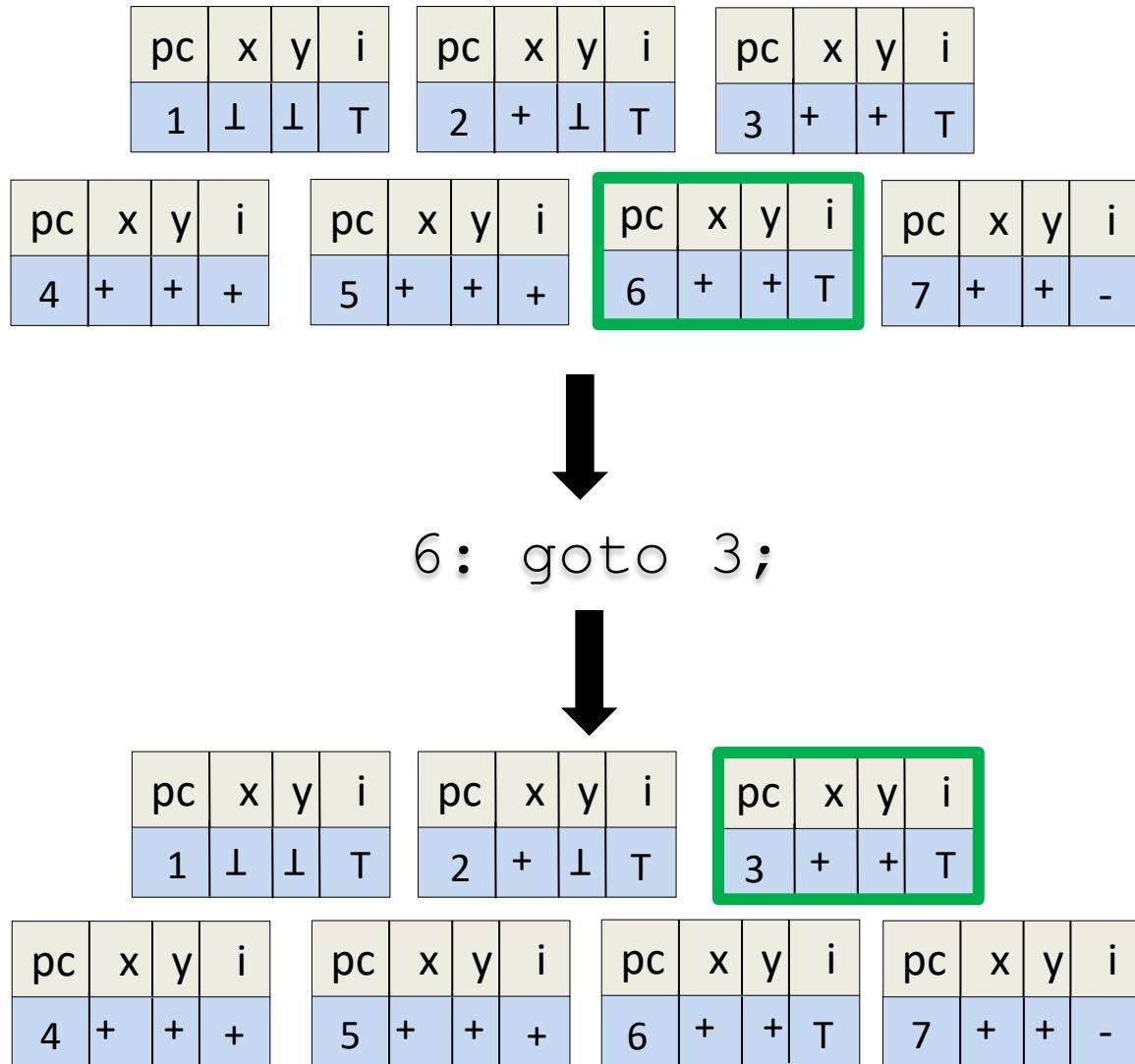
```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
pc	x	y	i
4	+	+	+
5	+	+	+
6	+	+	T
7	+	+	-



Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ x + y  
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```

pc	x	y	i
1	⊥	⊥	T
pc	x	y	i
2	+	⊥	T
pc	x	y	i
3	+	+	T
pc	x	y	i
4	+	+	+
pc	x	y	i
5	+	+	+
pc	x	y	i
6	+	+	T
pc	x	y	i
7	+	+	-



ANY STATEMENT

No matter what statement we execute from this state, we reach that same state

What is the loop invariant ?

Step 4: Check property

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ x + y  
}
```

pc	x	y	i
1	⊥	⊥	T
pc	x	y	i
2	+	⊥	T
pc	x	y	i
3	+	+	T
pc	x	y	i
4	+	+	+
pc	x	y	i
5	+	+	+
pc	x	y	i
6	+	+	T
pc	x	y	i
7	+	+	-

$$P \models (0 \leq x + y)$$



$$P_{\text{sign}} \models (0 \leq x + y)$$



sign domain precise enough
to prove property

Lets change the property

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7: assert 0 ≤ x - y  
}
```

pc	x	y	i
1	⊥	⊥	T
2	+	⊥	T
3	+	+	T
4	+	+	+
5	+	+	+
6	+	+	T
7	+	+	-

$$P \not\models (0 \leq x - y) \quad \text{X}$$

$$P_{\text{sign}} \not\models (0 \leq x - y) \quad \text{X}$$

sign domain is sound: property does not hold and it confirms it

Lets change the property again

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

pc	x	y	i
1	⊥	⊥	T
pc	x	y	i
2	+	⊥	T
pc	x	y	i
3	+	+	T
pc	x	y	i
4	+	+	+
pc	x	y	i
5	+	+	+
pc	x	y	i
6	+	+	T
pc	x	y	i
7	+	+	-

$$P \models (0 \leq y - x) \quad \checkmark$$

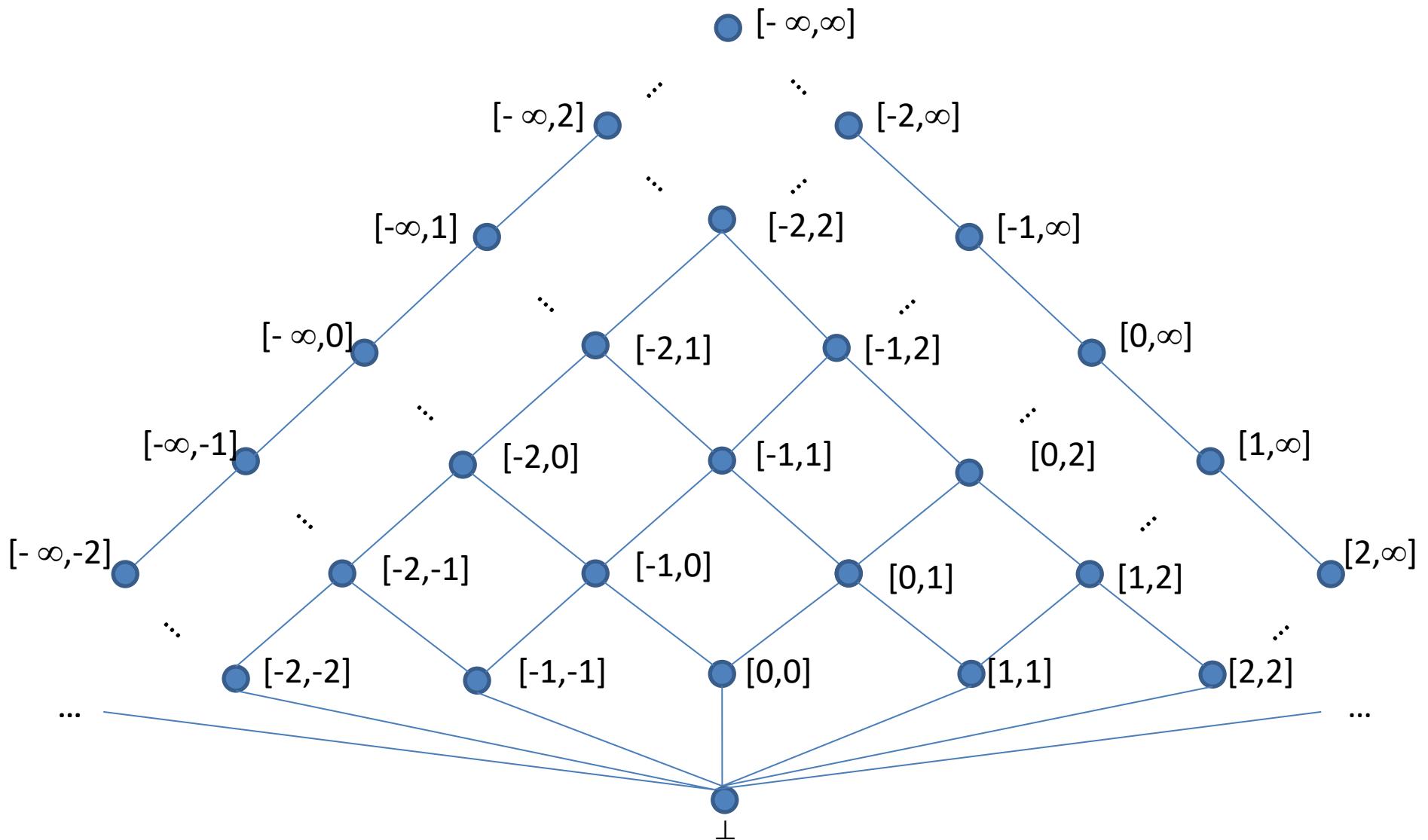
$$P_{\text{sign}} \not\models (0 \leq y - x) \quad \times$$

sign domain **too imprecise** to prove property

Lets try another abstraction

This time, instead of abstracting variable values using the **sign** of the variable, we will abstract the values using an interval

Step 1: Select interval domain



Step 1: Select abstract domain

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ y - x  
}
```

An abstract program state now looks like:

pc	x	y	i
2	$[-2, \infty]$	$[1, 7]$	$[1, 2]$

Step 2: Define Transformers

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

(...)

pc	x	y	i
4	$[-2, \infty]$	$[1, 7]$	$[1, 2]$

...)

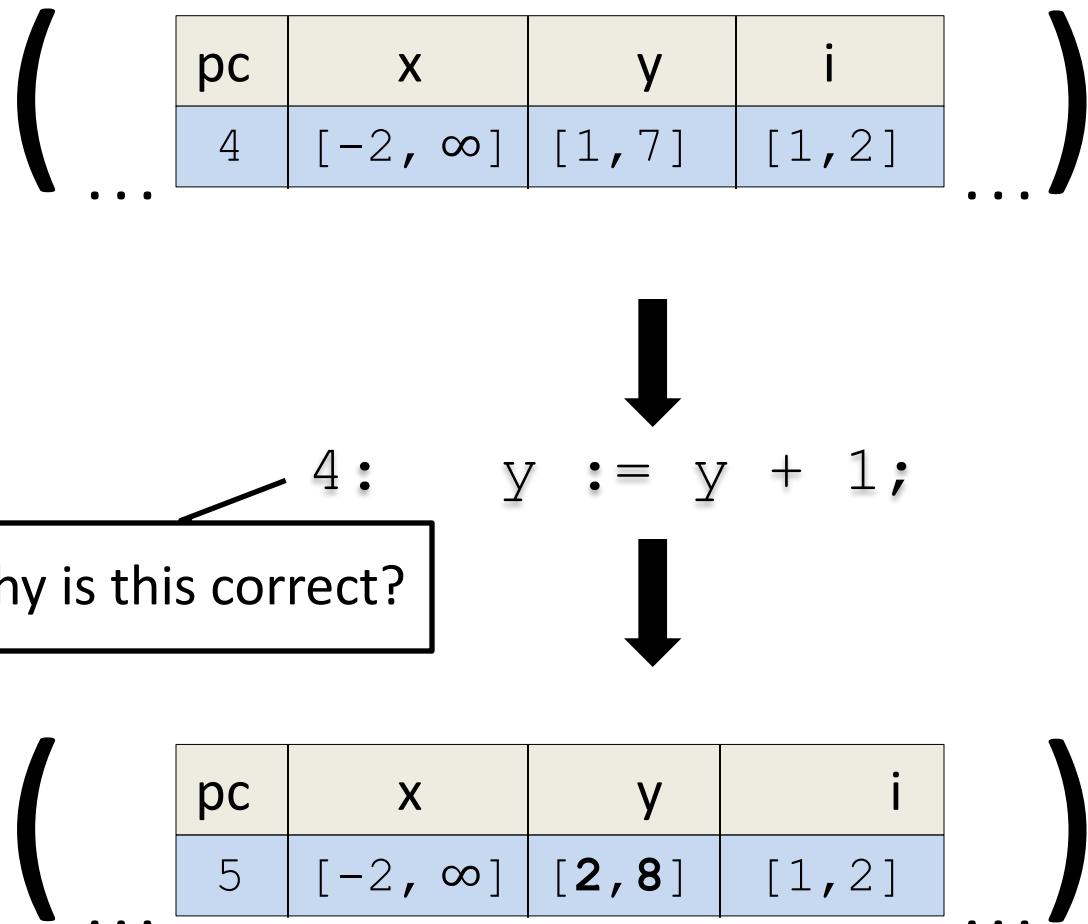
4 : y := y + 1;



?

Step 2: Define Transformers

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
7: assert 0 ≤ y - x  
}
```



Step 2: Define Transformers

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

(...)

pc	x	y	i
5	[−2, ∞]	[1, 7]	[1, 2]

...)

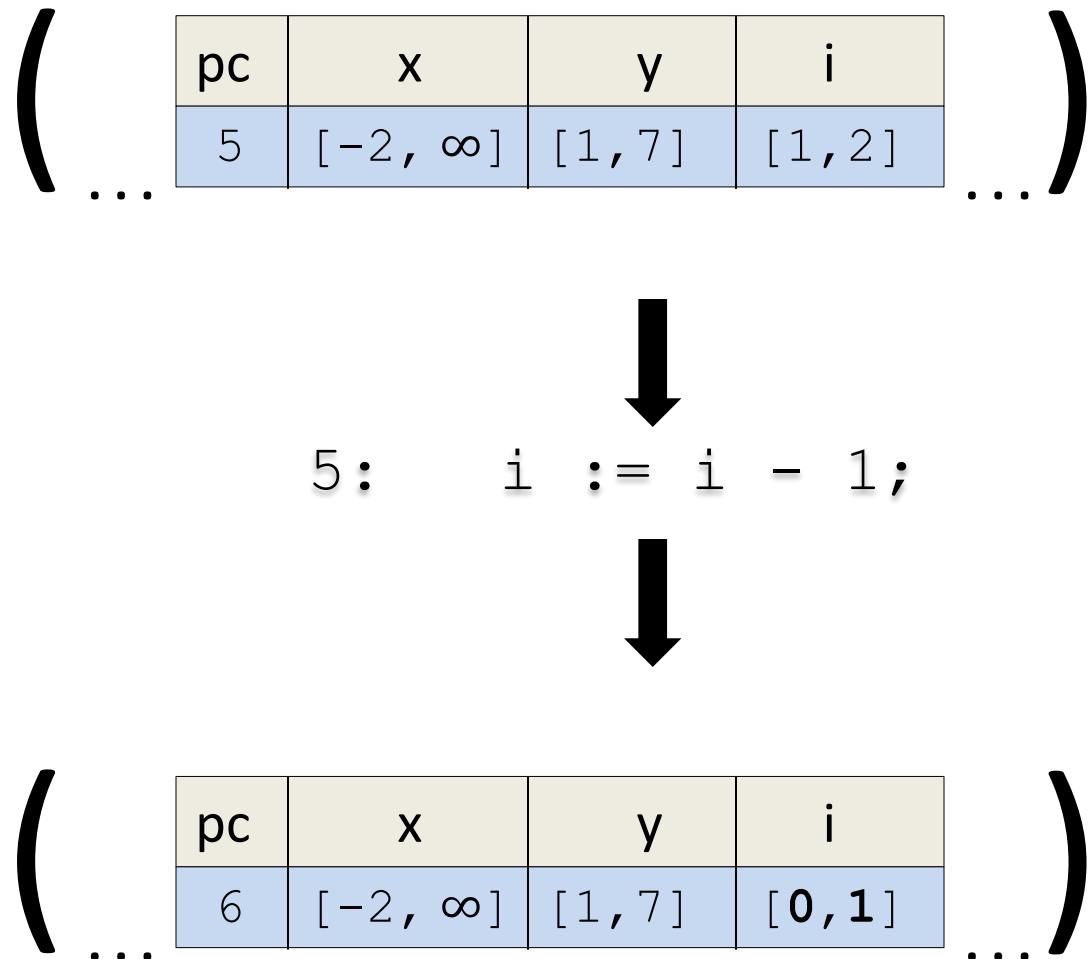
5: i := i - 1;



?

Step 2: Define Transformers

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {  
  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

Again, we start with
the **least** abstract element

pc	x	y	i
1	⊥	⊥	⊥
pc	x	y	i
2	⊥	⊥	⊥
pc	x	y	i
3	⊥	⊥	⊥
pc	x	y	i
4	⊥	⊥	⊥
pc	x	y	i
5	⊥	⊥	⊥
pc	x	y	i
6	⊥	⊥	⊥
pc	x	y	i
7	⊥	⊥	⊥

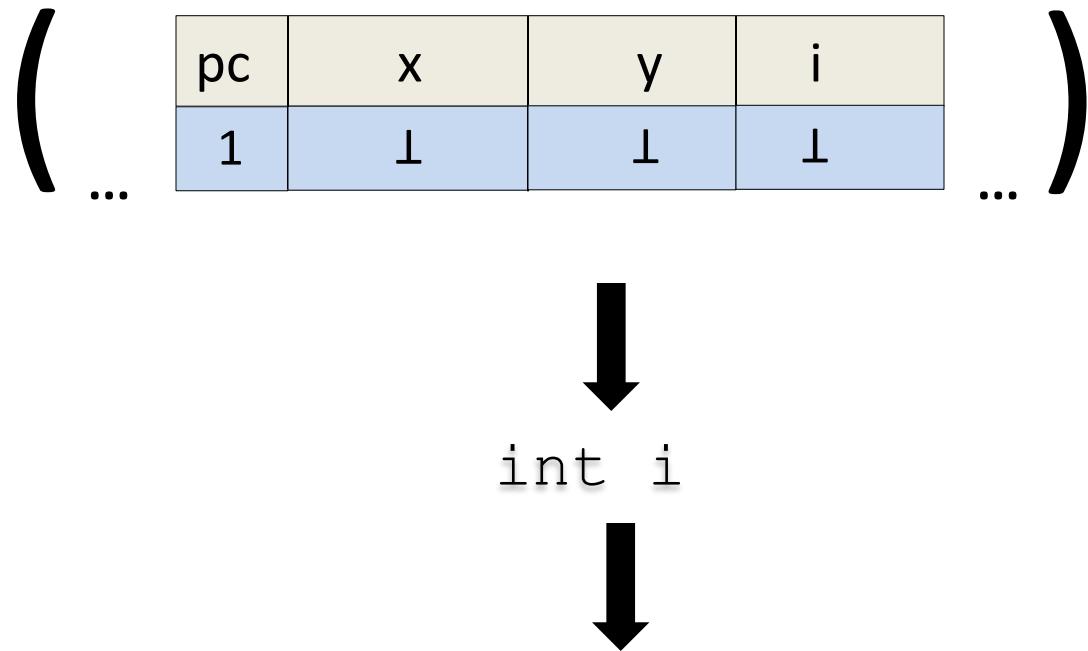
Lets the iterations begin !

Note:

we only show the change of 1 component

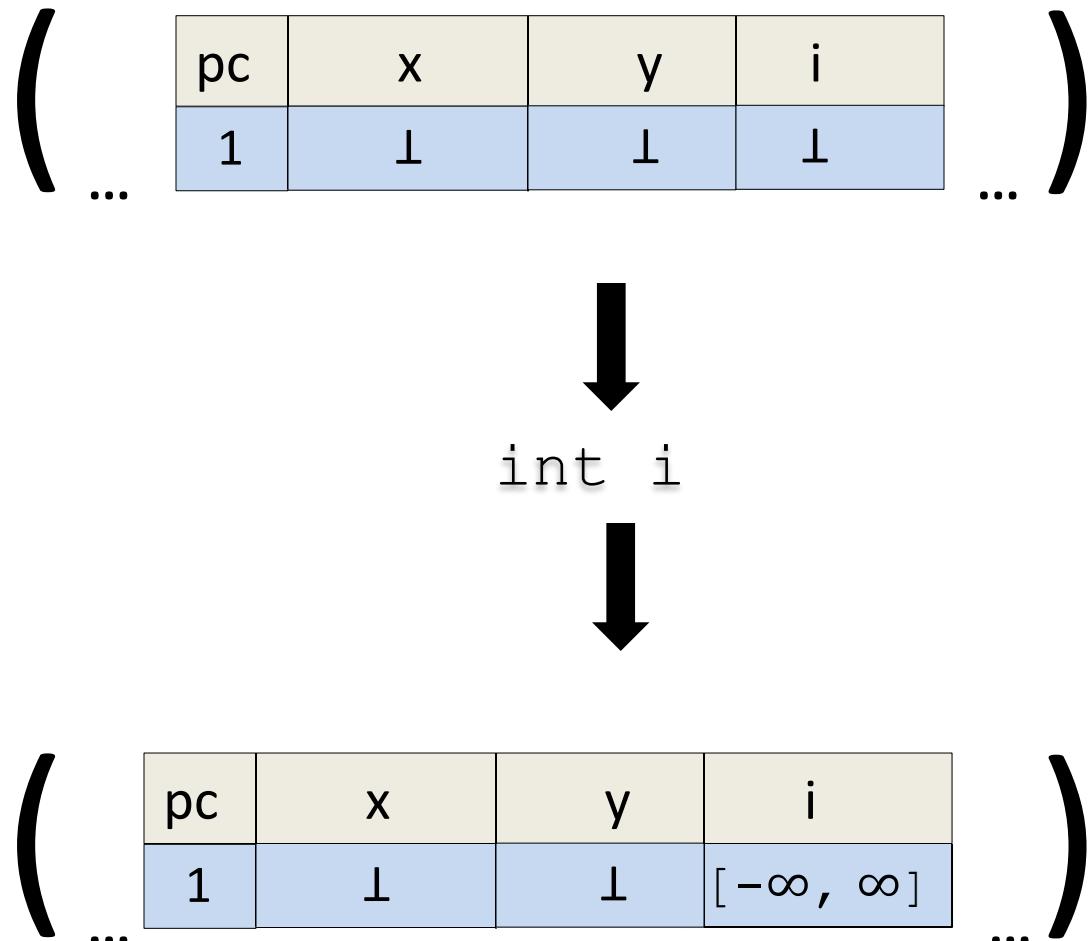
Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```



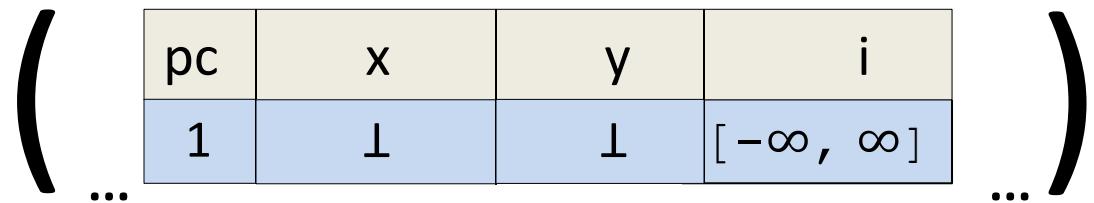
Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

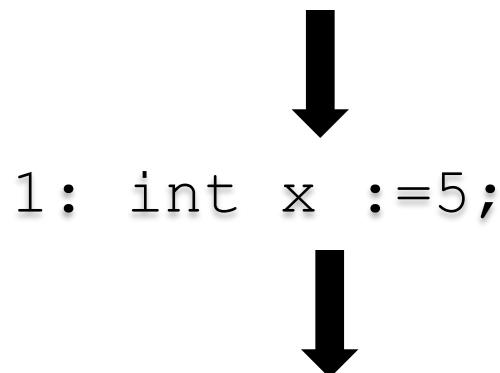


Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x :=5;  
    2: int y :=7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ y - x  
}
```

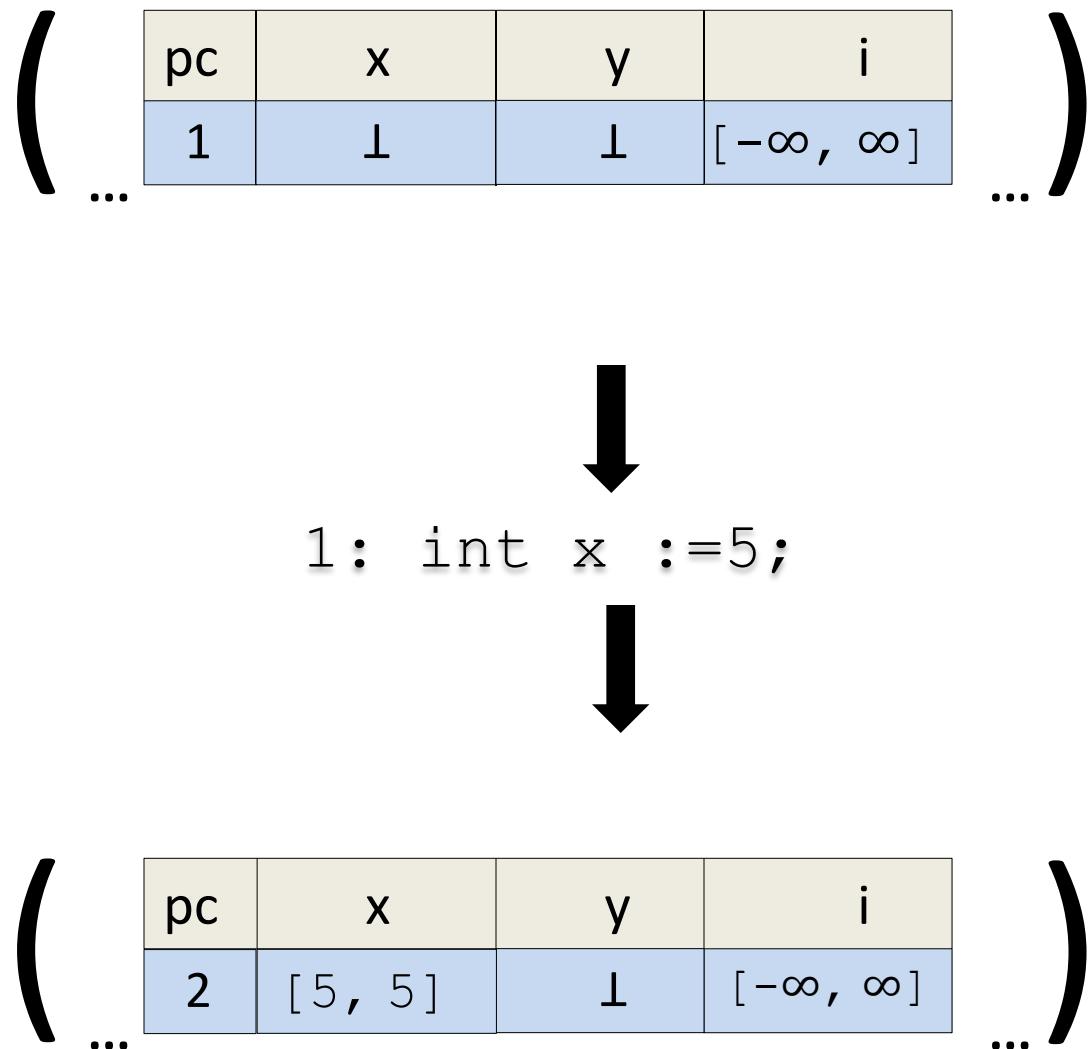


pc	x	y	i
1	⊥	⊥	$[-\infty, \infty]$



Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x :=5;  
    2: int y :=7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ y - x  
}
```

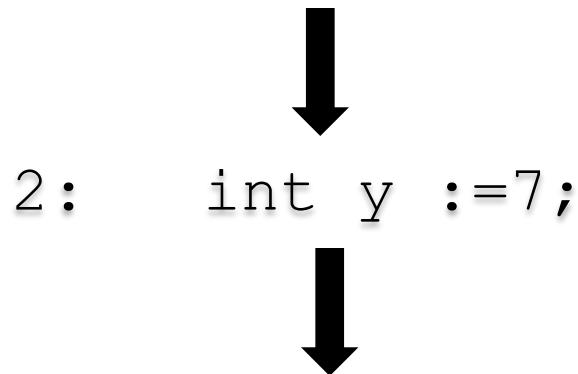


Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

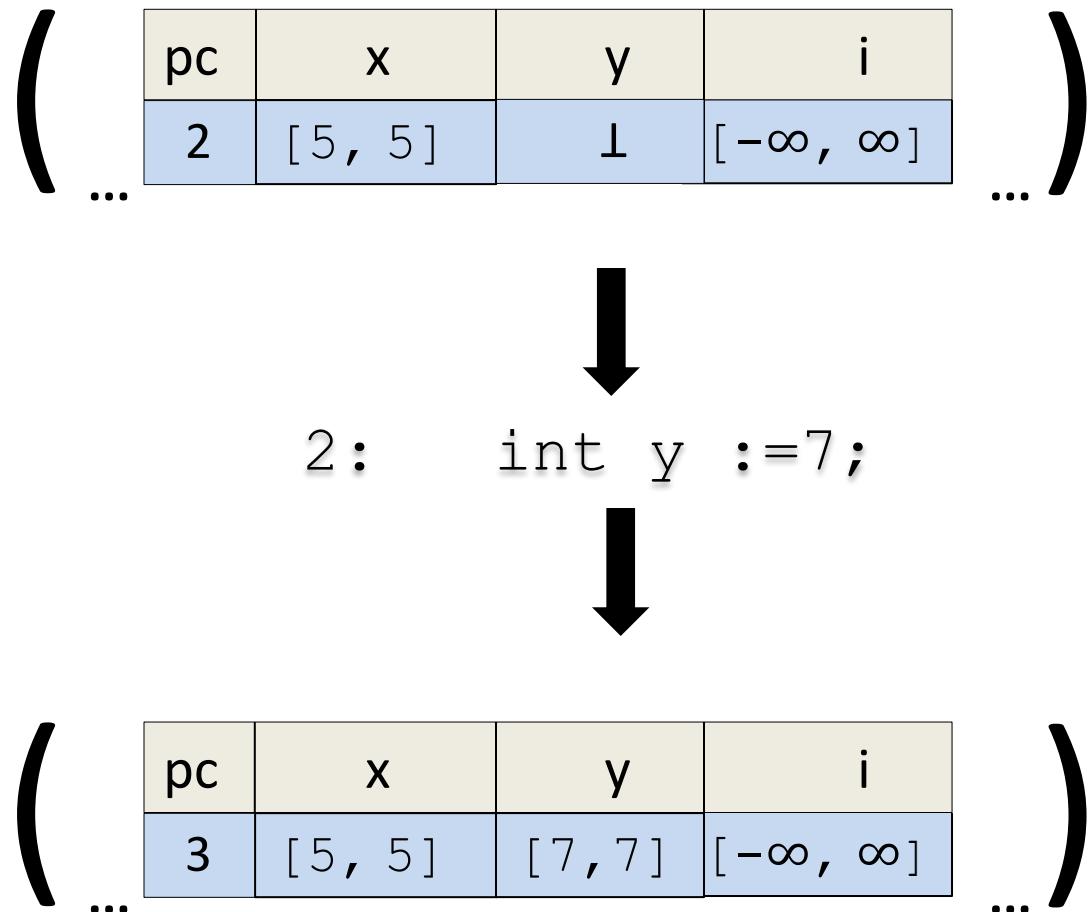
(... | pc | x | y | i | ...)

...	pc	x	y	i	...
...	2	[5, 5]	⊥	[-∞, ∞]	...



Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

()

pc	x	y	i
3	[5, 5]	[7, 7]	[-∞, ∞]
...			

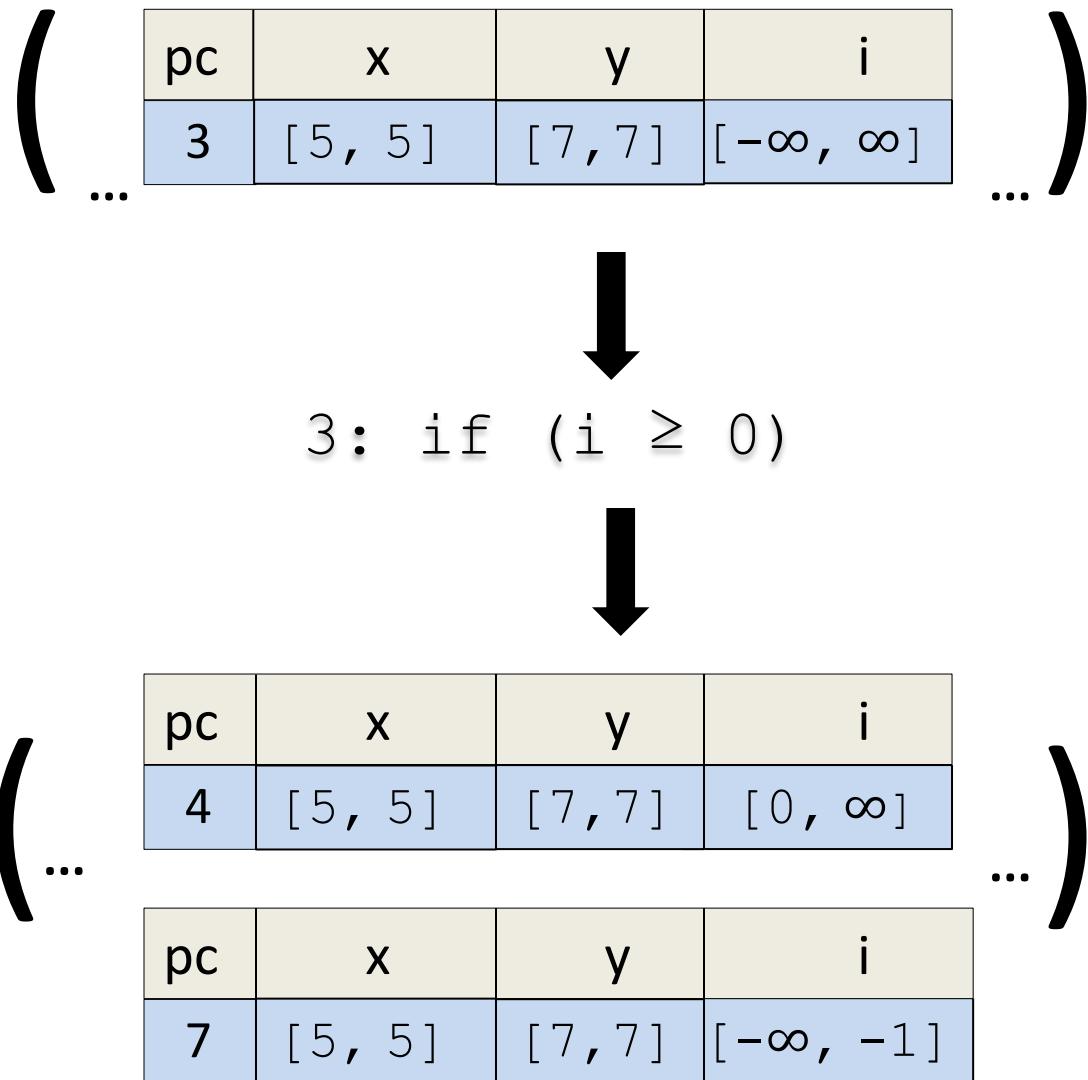
...)



3: if (i ≥ 0)

Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ y - x  
}
```



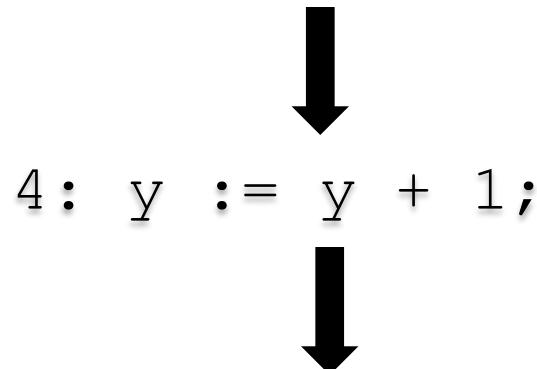
Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
    4:     y := y + 1;  
    5:     i := i - 1;  
    6:     goto 3;  
    }  
  
    7: assert 0 ≤ y - x  
}
```

(...)

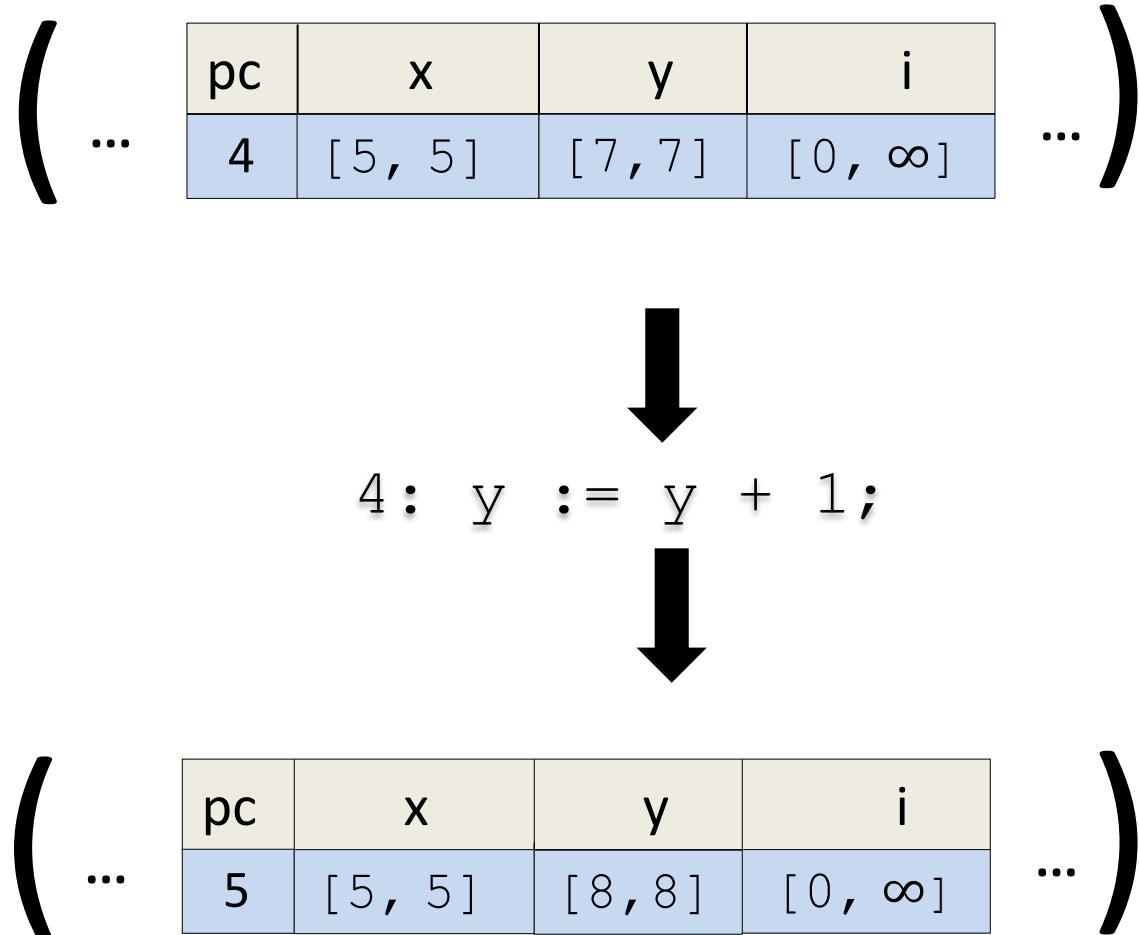
pc	x	y	i
4	[5, 5]	[7, 7]	[0, ∞]

...)



Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```



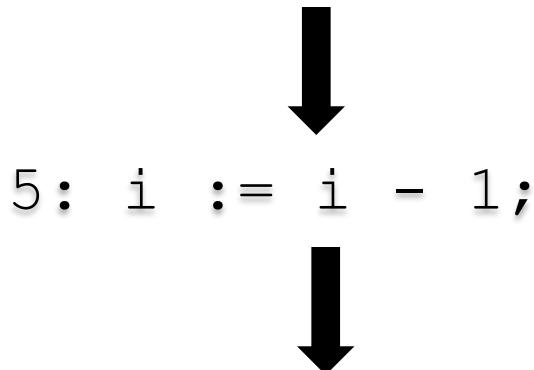
Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

(...)

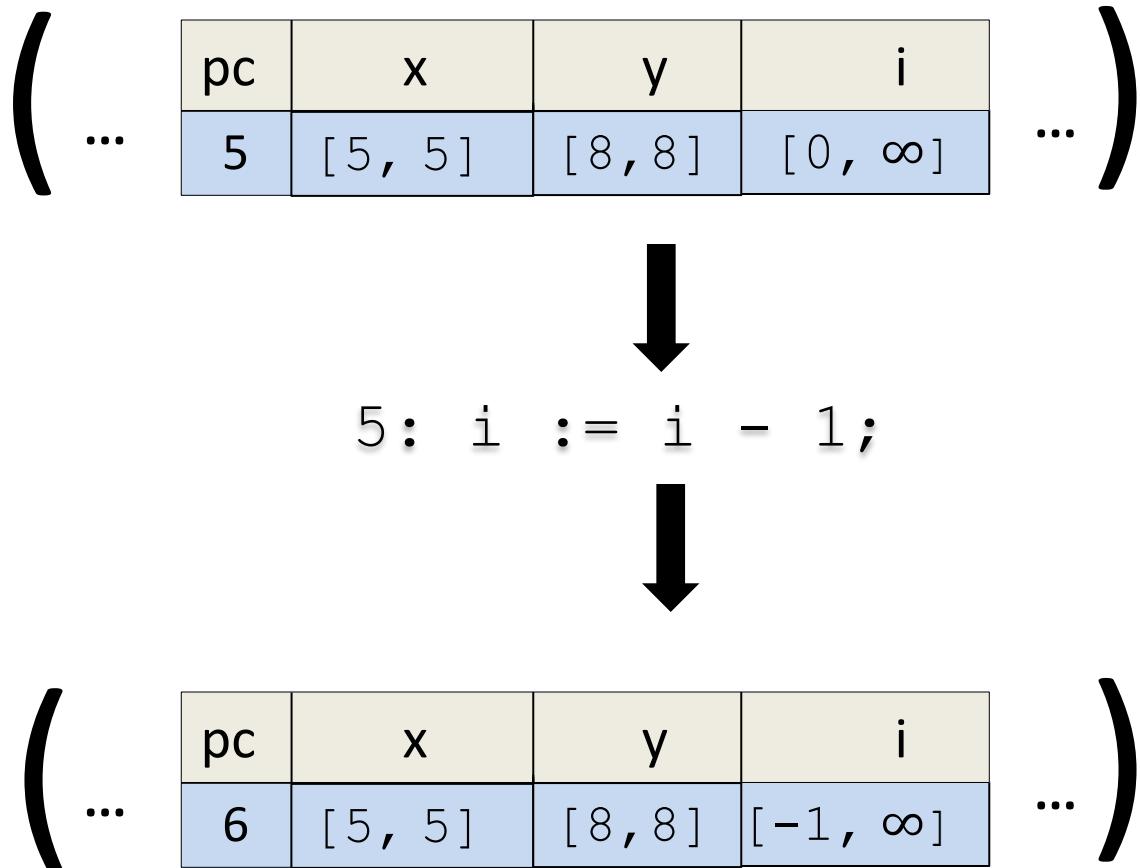
pc	x	y	i
5	[5, 5]	[8, 8]	[0, ∞]

...)



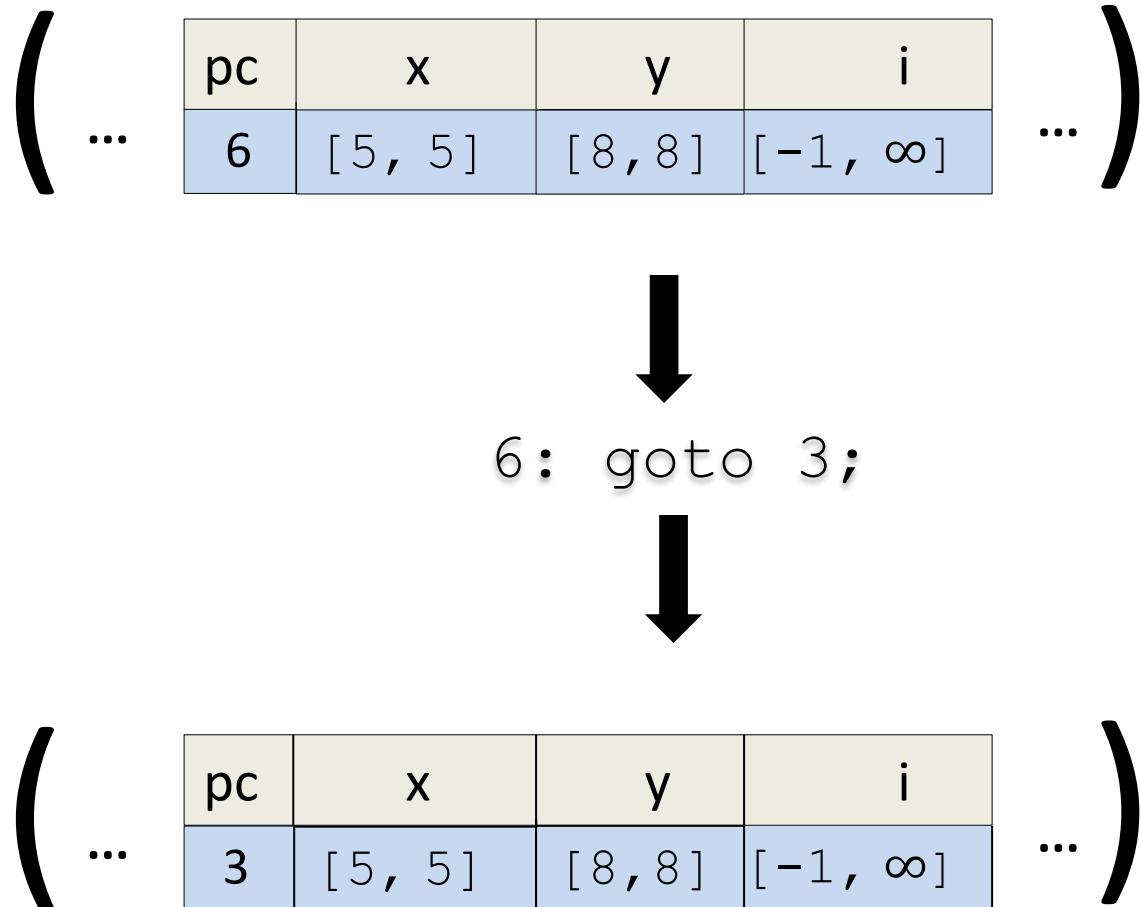
Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

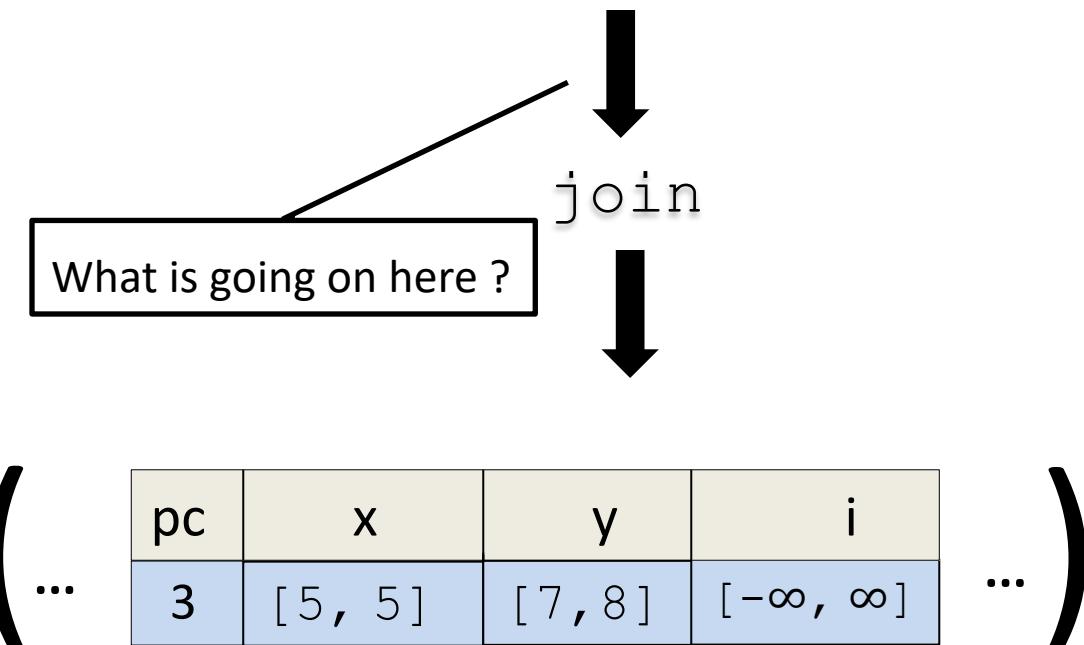


Step 3: Iterate to a fixed point

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
  
    7: assert 0 ≤ y - x  
}
```

pc	x	y	i
3	[5, 5]	[8, 8]	[-1, ∞]

pc	x	y	i
3	[5, 5]	[7, 7]	[-∞, ∞]



Joins

When we have two abstract elements A and B, we can **join** them to produce their (least) **upper bound**

denoted by: $A \sqcup B$

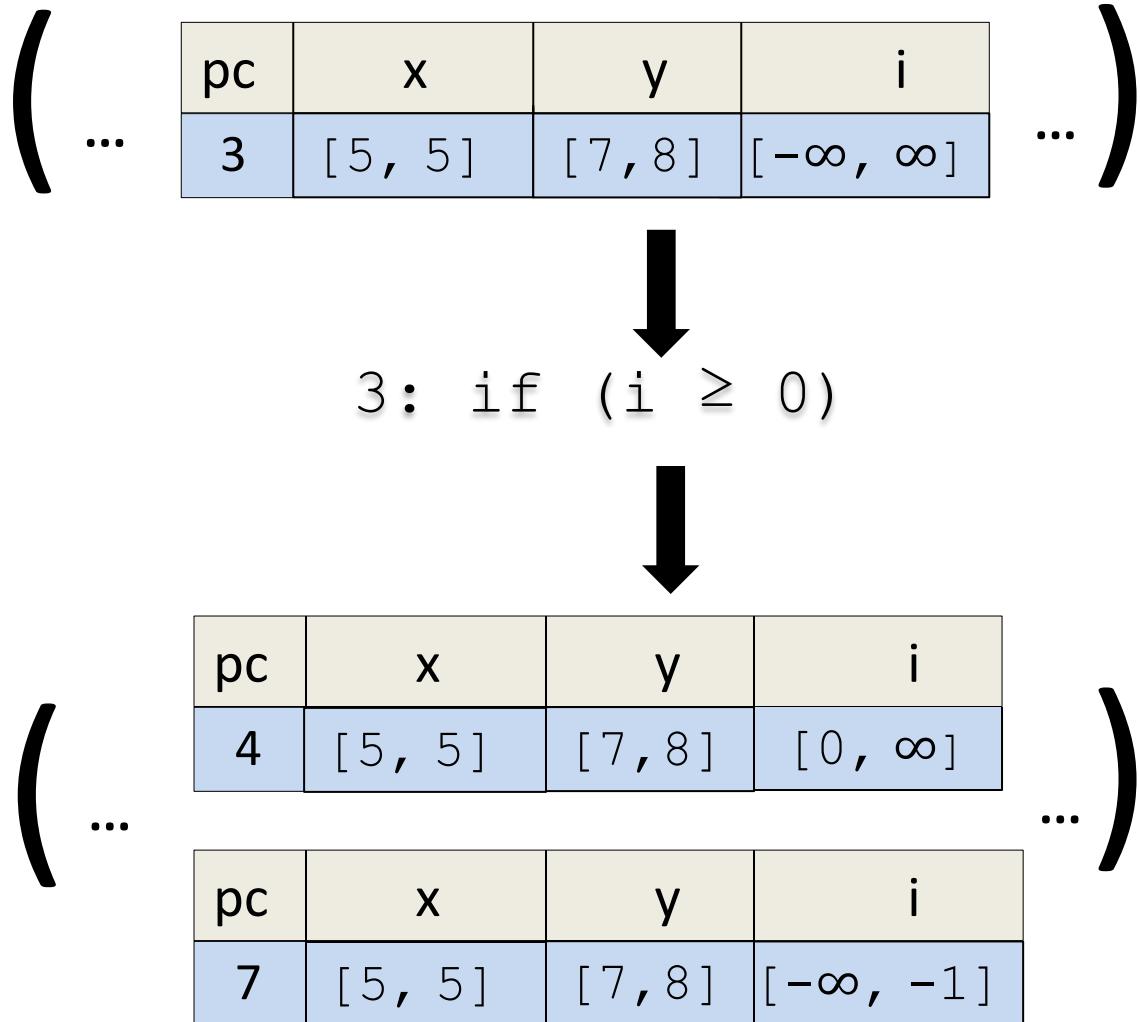
we have that $A \sqsubseteq A \sqcup B$ and $B \sqsubseteq A \sqcup B$

$D \sqsubseteq E$ means that E is **more abstract** than D

In our example, we join the abstract states that occur at the same program label

Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```



Step 3: Iterate to a fixed point

```
foo (int i) {  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

(...)

pc	x	y	i
4	[5, 5]	[7, 8]	[0, ∞]

... ()

↓
4 : y := y + 1

This will never terminate !

y will keep increasing forever !

But states reaching label 7 will always satisfy the assertion !

Cannot reach a Fixed point

```
foo (int i) {  
1: int x :=5;  
2: int y :=7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
    }  
  
7: assert 0 ≤ y - x  
}
```

With the interval abstraction we could not reach a fixed point.

The domain has infinite height.

What should we do ?

Introduce a special operator called the widening operator !

It ensures termination at the expense of precision

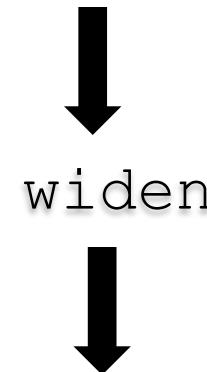
Widening instead of Join

```
foo (int i) {  
    1: int x := 5;  
    2: int y := 7;  
  
    3: if (i ≥ 0) {  
        4:     y := y + 1;  
        5:     i := i - 1;  
        6:     goto 3;  
    }  
    7: assert 0 ≤ y - x  
}
```

pc	x	y	i
3	[5, 5]	[7, 7]	[-∞, ∞]

pc	x	y	i
3	[5, 5]	[8, 8]	[0, ∞]

y is increasing,
go directly to ∞



(...)

pc	x	y	i
3	[5, 5]	[7, ∞]	[-∞, ∞]

...)

Fixed Point after Widening

```
foo (int i) {  
  
1: int x := 5;  
2: int y := 7;  
  
3: if (i ≥ 0) {  
4:     y := y + 1;  
5:     i := i - 1;  
6:     goto 3;  
}  
  
7: assert 0 ≤ y - x  
}
```

With **widening**, our iteration now reaches a fixed point, where at label 7 we have:

$$\left(\dots \begin{array}{|c|c|c|c|} \hline \text{pc} & \text{x} & \text{y} & \text{i} \\ \hline 7 & [5, 5] & [7, \infty] & [-\infty, -1] \\ \hline \end{array} \dots \right)$$

$$P \models (0 \leq y - x) \quad \checkmark$$

$$P_{\text{Interval}} \models (0 \leq y - x) \quad \checkmark$$

interval domain **precise enough** to prove property !

Questions that should bother you

- What are we abstracting **exactly**?
- What are abstract domains **mathematically**?
- How do we discover **best/sound** abstract transformers?
What does **best** mean?
- **What is** the function that we iterate to a fixed point?
- How do we ensure **termination** of the analysis?

Modeling and Analysis of Memory Models using Alloy

Andrei Dan

Hitachi ABB Power Grids Research

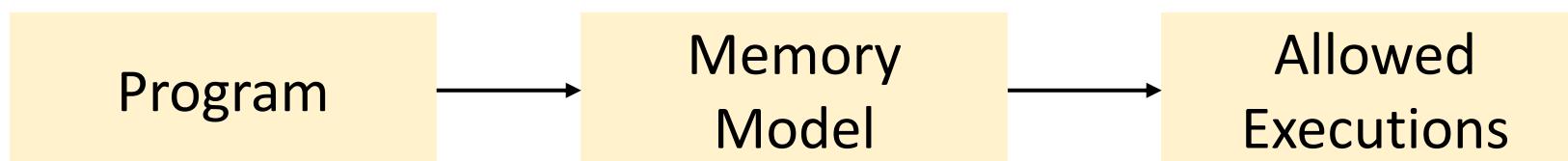
Plan

1. Introduction to [Memory Models](#) (Sequential Consistency, x86 TSO)
2. Memory Model [Axiomatic Semantics](#) (x86 TSO)
3. Axiomatic Semantics in [Alloy](#) (x86 TSO)
4. Memory Model [Evaluation Framework](#) (Remote Memory Access Model)

1. Introduction to Memory Models

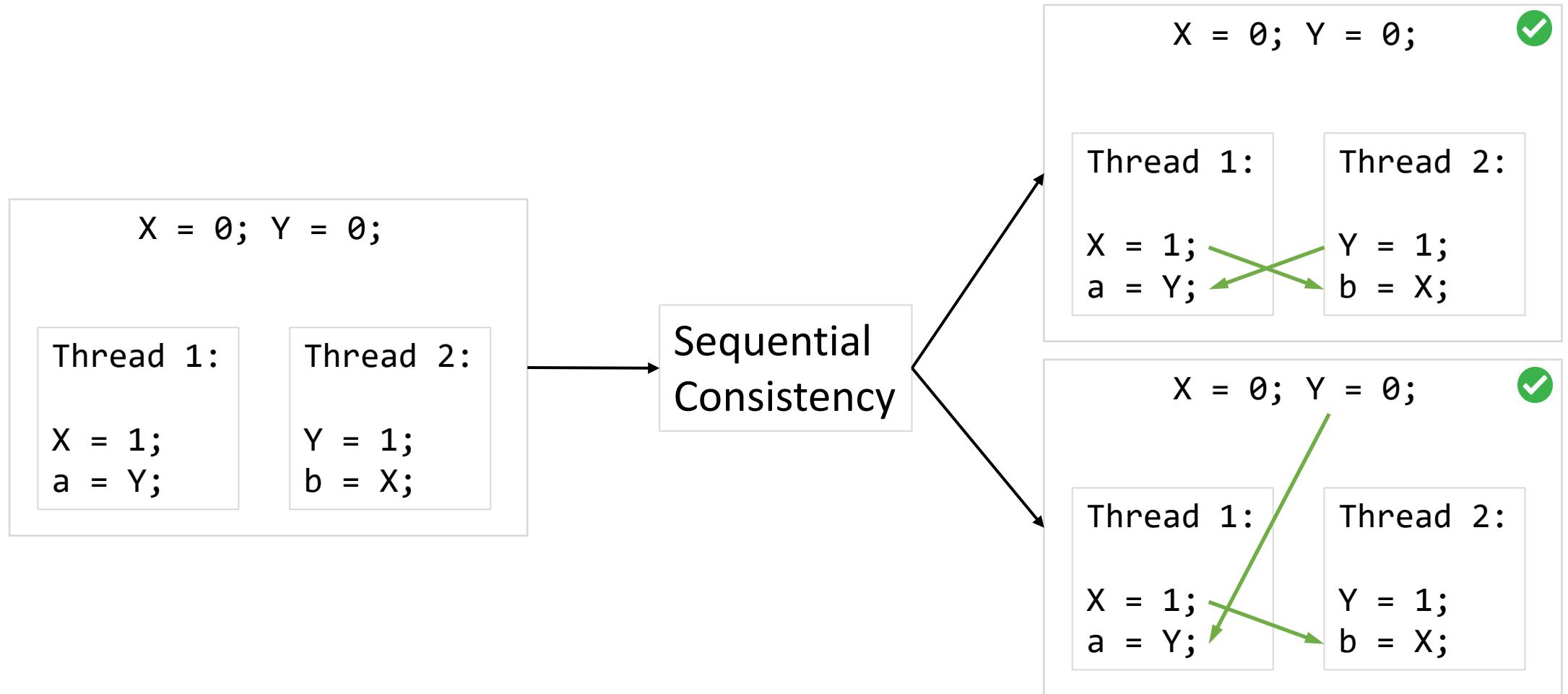
Interface between programs and their possible executions.

Determine which behaviour the memory (and thus a program) is allowed to have.

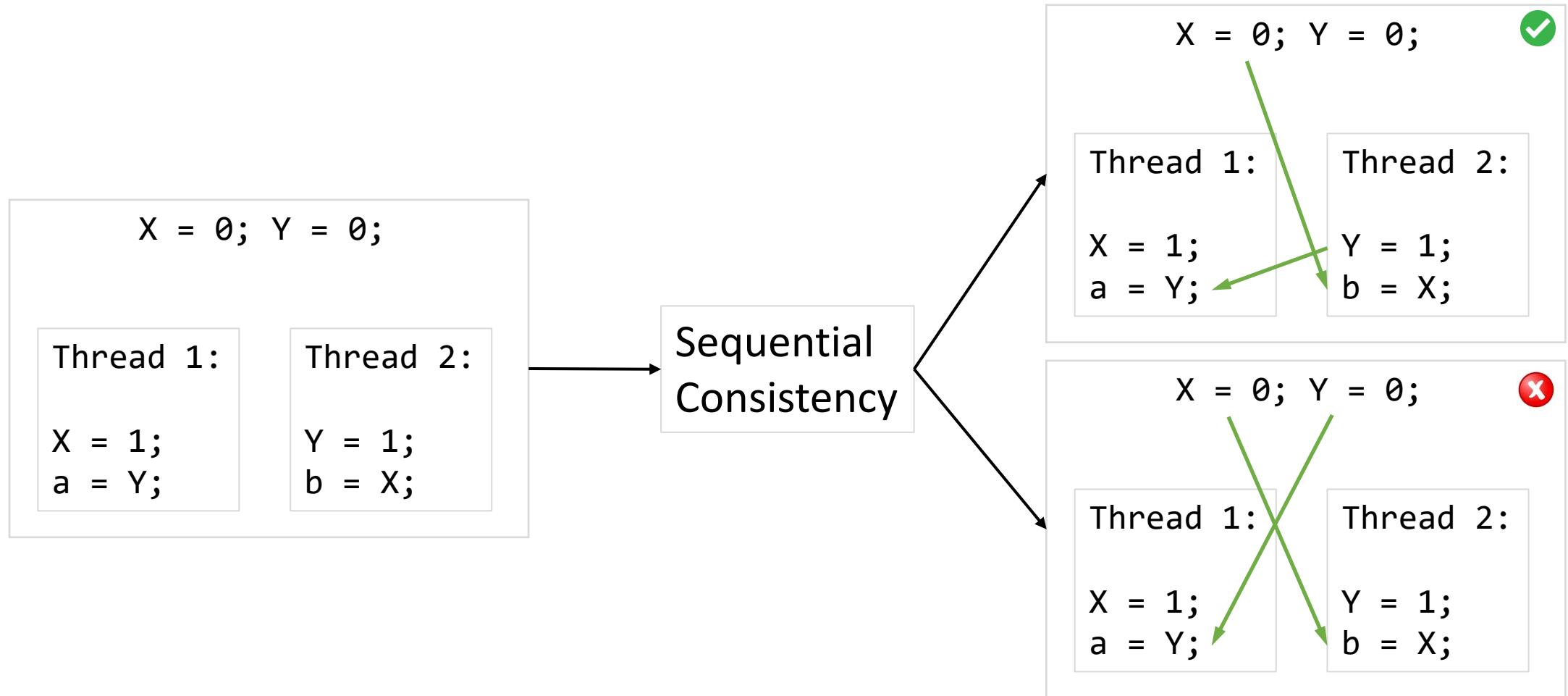


Examples: Sequential Consistency, x86 Total Store Order (TSO), Remote Memory Access (RMA), ARM, RISC-V, NVIDIA PTX

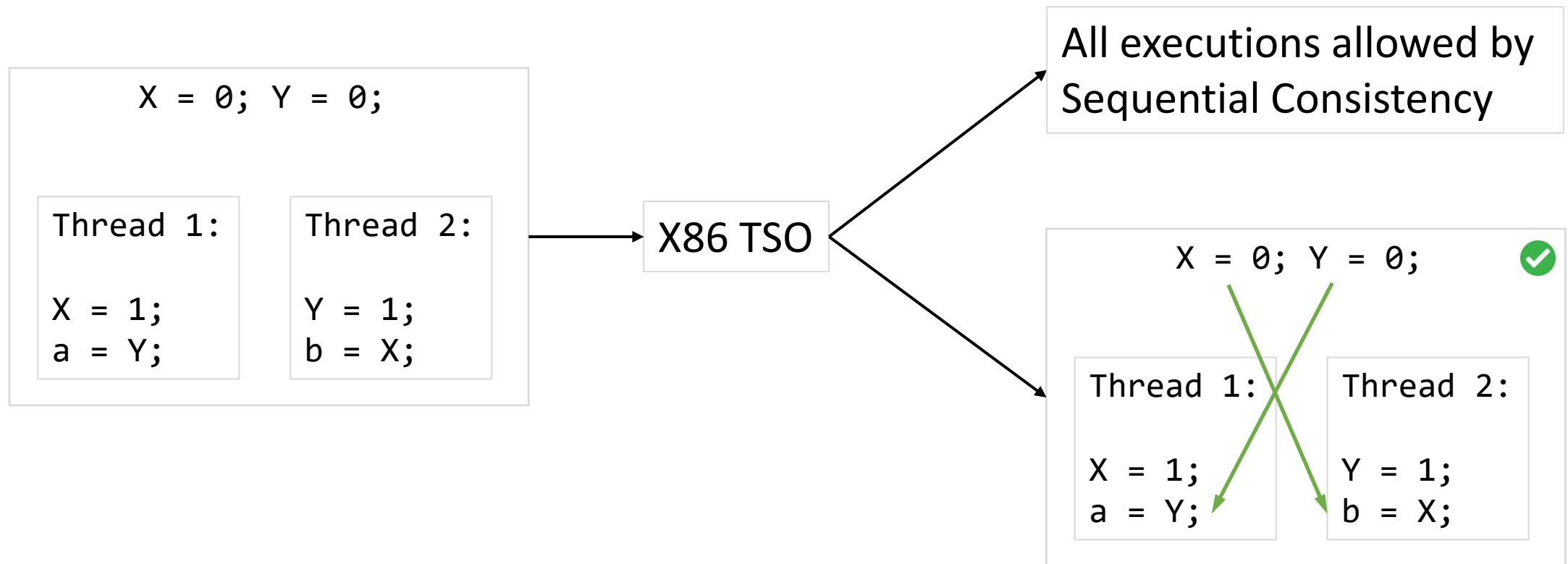
Example – Sequential Consistency



Example – Sequential Consistency



Example – x86 Total Store Order (TSO)



```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Shared variables */
int X = 0, Y = 0;

/* Function executed by thread 1 */
void *fun_thread1(void *a) {
    X = 1;
    *(int *)a = Y;
}

/* Function executed by thread 2 */
void *fun_thread2(void *b) {
    Y = 1;
    *(int *)b = X;
}

int main() {
    pthread_t thread1, thread2;
    int a, b;
    int N = 1000000;

    for (int i = 0; i < N; i++) {
        /* Set shared variables */
        X = 0; Y = 0;
        /* Start thread 1 and thread 2*/
        pthread_create(&thread1, NULL, fun_thread1, (void *)&a);
        pthread_create(&thread2, NULL, fun_thread2, (void *)&b);
        /* Wait for the completion of thread 1 and thread 2 */
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);

        if (a == 0 && b == 0) {
            printf ("i = %u: a == %u, b == %u\n", i, a, b);
        }
    }
    return 0;
}

```

Experiment - x86 TSO

Intel i5 CPU

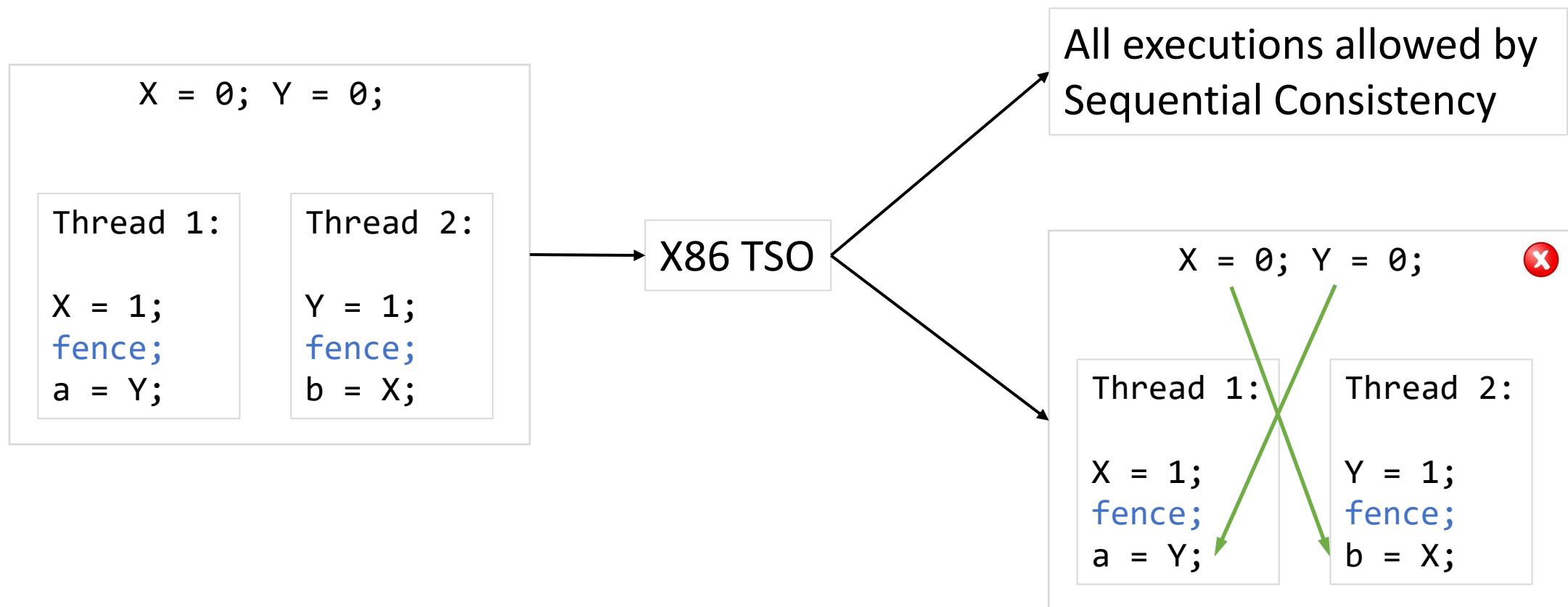
```

% ./testTSO
i = 19697: a == 0, b == 0
i = 30698: a == 0, b == 0
i = 101964: a == 0, b == 0
i = 155845: a == 0, b == 0
i = 162244: a == 0, b == 0
i = 186209: a == 0, b == 0
i = 186479: a == 0, b == 0
i = 209880: a == 0, b == 0
i = 218363: a == 0, b == 0
i = 231494: a == 0, b == 0
i = 365010: a == 0, b == 0
i = 397441: a == 0, b == 0
i = 403797: a == 0, b == 0
i = 499606: a == 0, b == 0
i = 525450: a == 0, b == 0
i = 685484: a == 0, b == 0
i = 717885: a == 0, b == 0
i = 777638: a == 0, b == 0
i = 824385: a == 0, b == 0
i = 866250: a == 0, b == 0
i = 909346: a == 0, b == 0
i = 911629: a == 0, b == 0
i = 963548: a == 0, b == 0
i = 967914: a == 0, b == 0
i = 996937: a == 0, b == 0

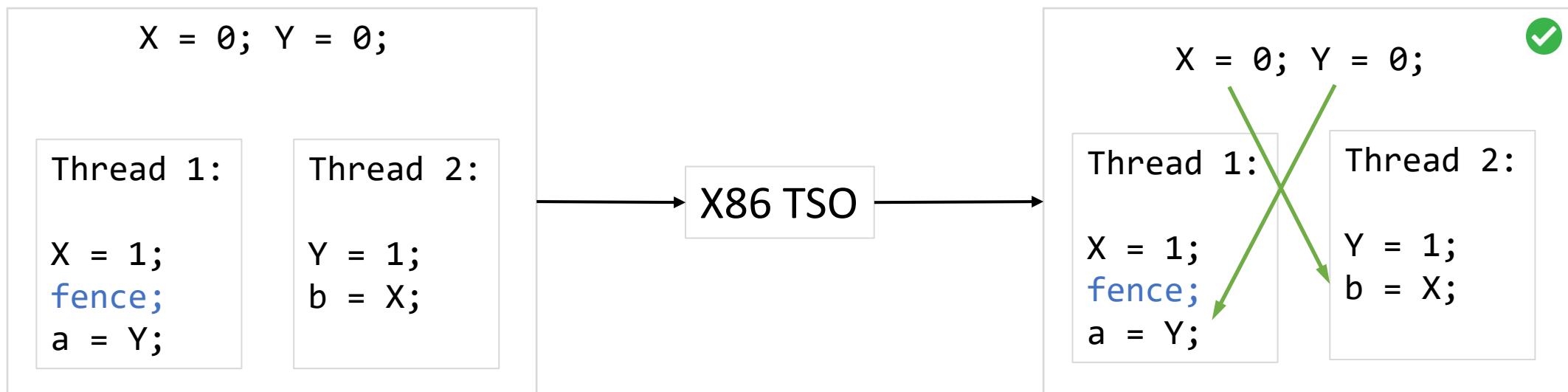
```

26/1'000'000 executions
allowed by x86 TSO and
not by Sequential Consistency

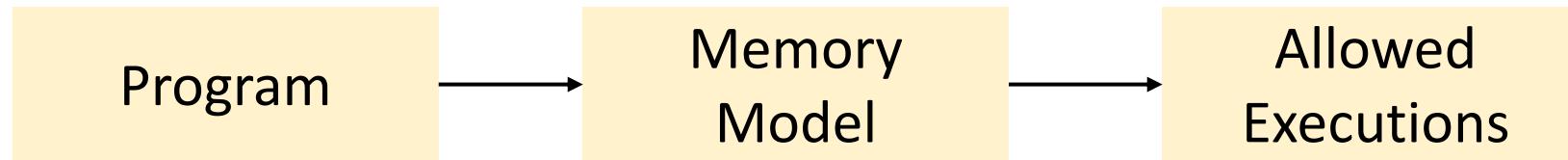
Memory Fences – Control Allowed Executions



Question – Allowed Execution?



Introduction to Memory Models - Summary



The memory model determines which executions are allowed for a program.
In particular: **from which writes can the read operations read from.**

2. Memory Model Axiomatic Semantics

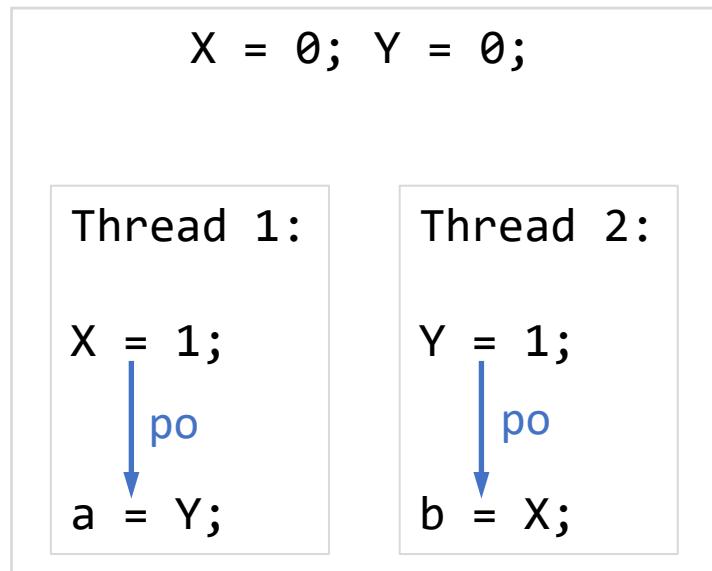
- Rigorous description of the memory model
- Axiomatic semantics defines **relations over events**
- **Events** considered for x86 TSO: read, write, fence

Relations Between Events: Program Order



Program Order – orders events in the same thread (given by program)
– transitively closed, acyclic

Example:

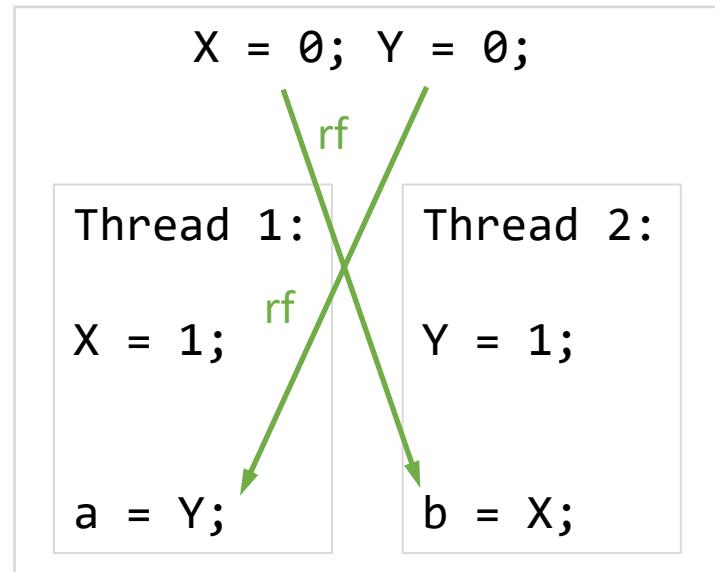


Relations Between Events: Reads From

e1 $\xrightarrow{\text{rf}}$ e2

Reads From – e2 reads the value written by e1 (execution-specific)
– each read event reads from exactly one write

Example:

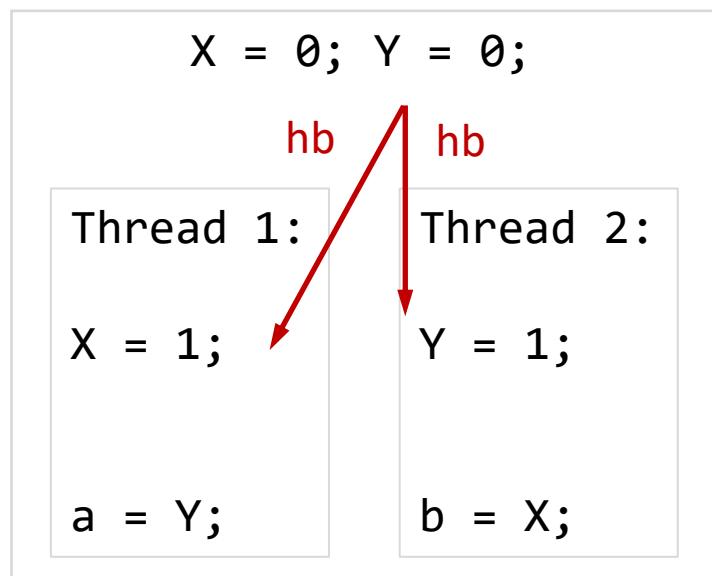


Relations Between Events: Happens Before

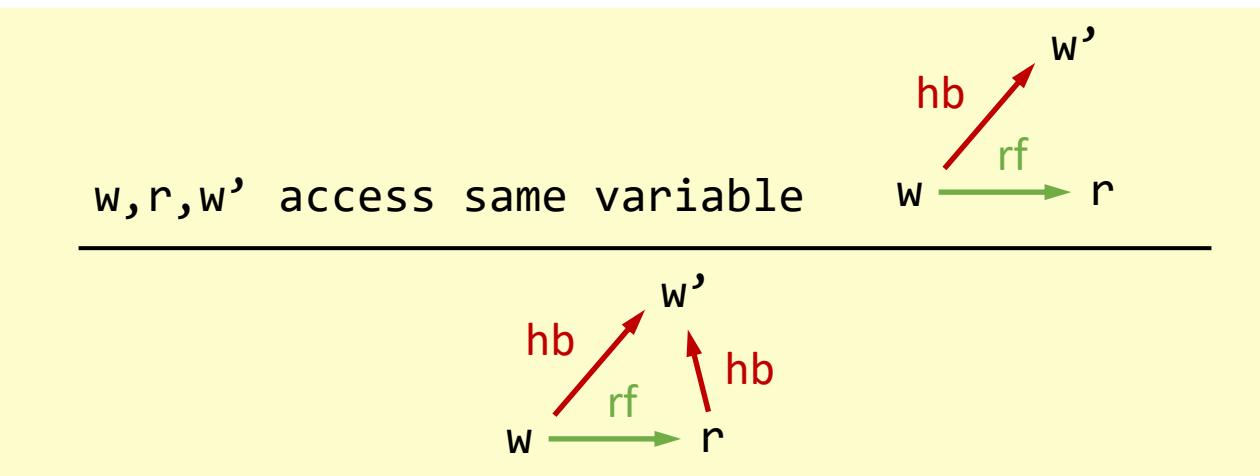
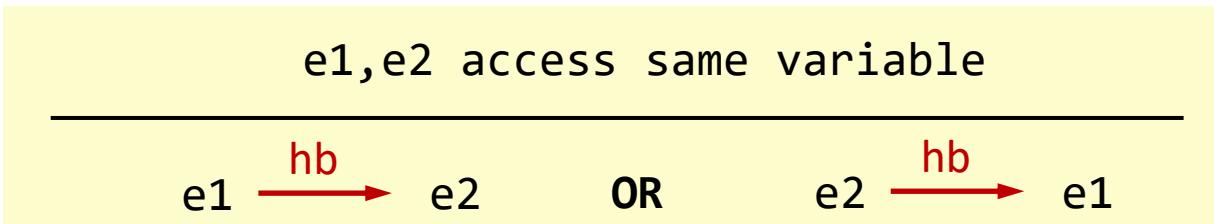
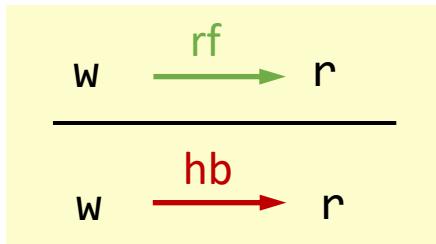


Happens Before – guarantees that the effects of $e1$ are visible to $e2$ (execution-specific)
– transitively closed, acyclic

Example:



Axiomatic Semantics Rules - x86 TSO (and SC)



Axiomatic Semantics Rules - x86 TSO specific

$$\frac{e_1 \xrightarrow{\text{po}} e_2 \quad e_1, e_2 \text{ access same variable}}{e_1 \xrightarrow{\text{hb}} e_2}$$

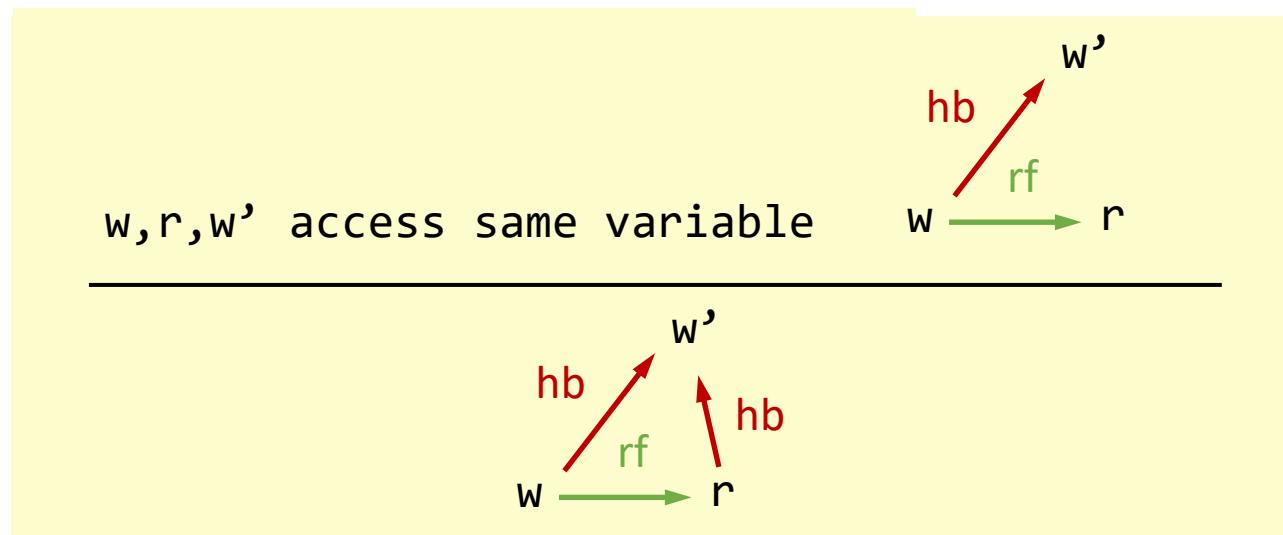
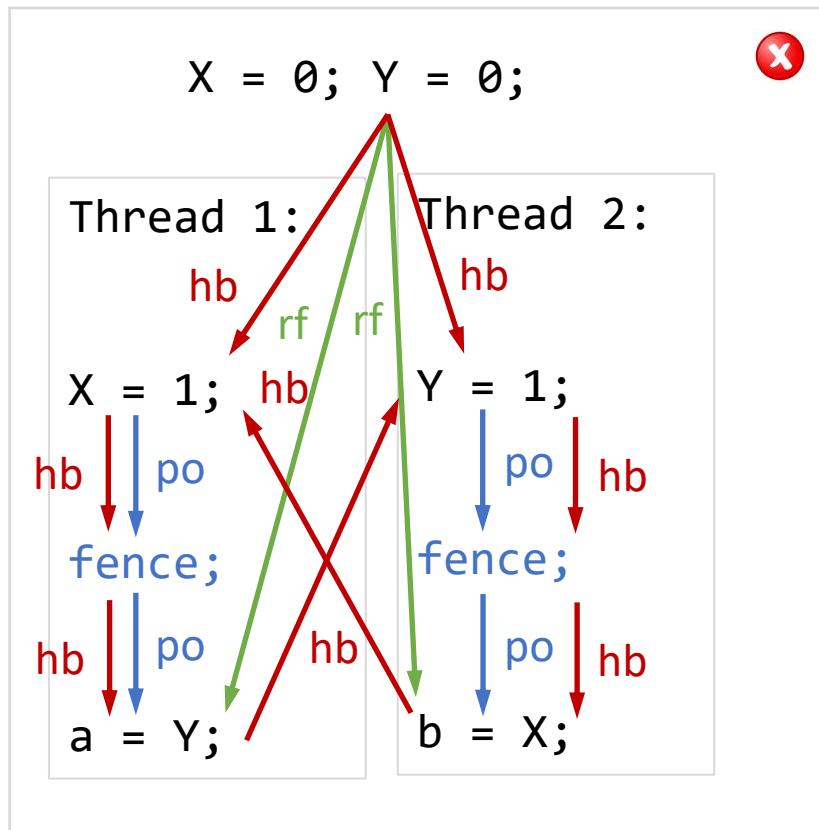
$$\frac{e_1 \xrightarrow{\text{po}} f \quad f \text{ is a fence}}{e_1 \xrightarrow{\text{hb}} f}$$

$$\frac{f \xrightarrow{\text{po}} e_1 \quad f \text{ is a fence}}{f \xrightarrow{\text{hb}} e_1}$$

How to check if an execution given by rf is allowed?

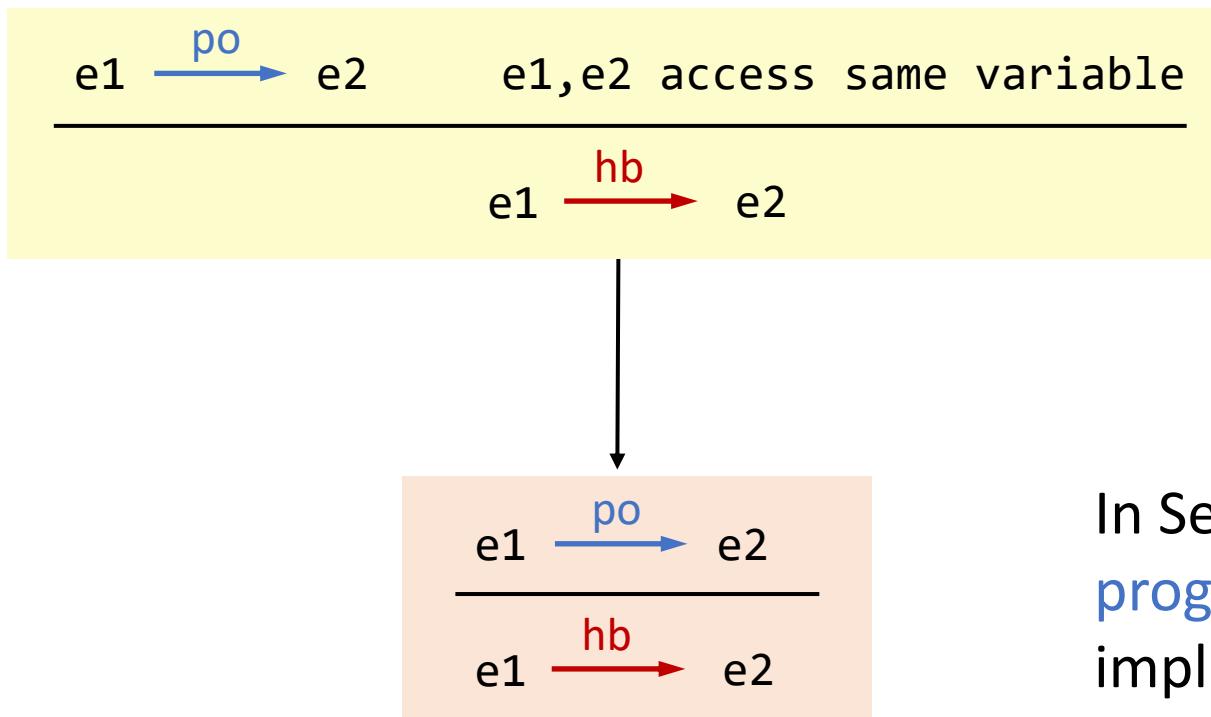
1. Apply iteratively all the rules until no new relations are added
2. Execution is allowed iff hb is acyclic

Example x86 TSO Axiomatic Semantics



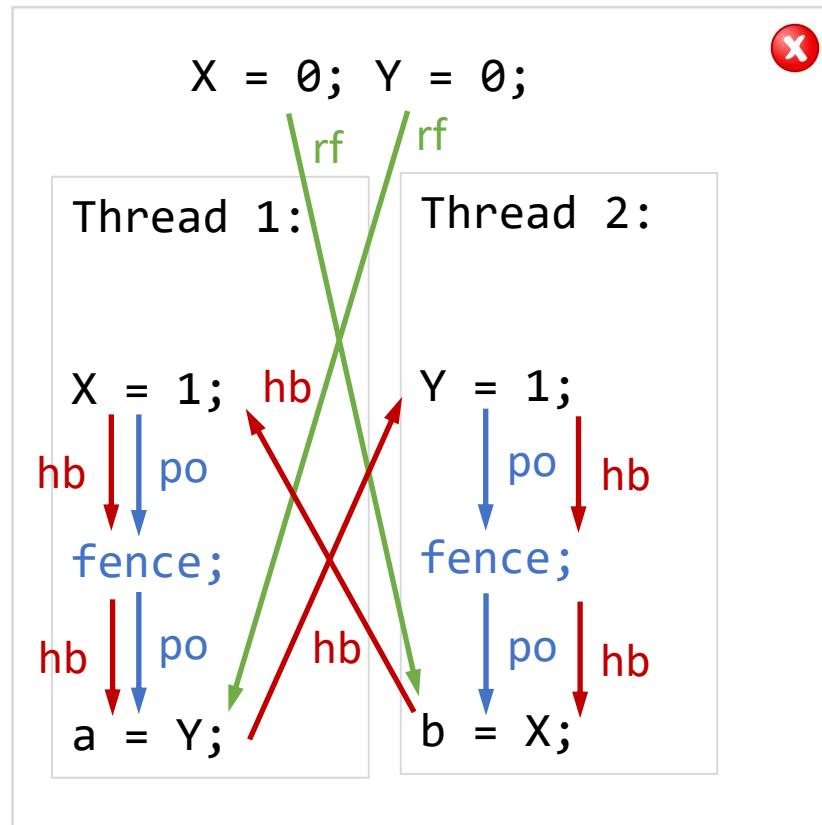
Question – Sequential Consistency Semantics

To obtain Sequential Consistency, we can modify just the x86 TSO rule below. How?



In Sequential Consistency,
program order always
implies happens before

Memory Model Semantics - Summary



Axiomatic semantics for a memory model (rules for **hb**) determines whether an execution (**rf**) is allowed for a program (events and **po**).

How to check if an execution given by **rf** is allowed?

1. Apply iteratively all the rules until no new relations are added
2. Execution is allowed **iff** **hb** is acyclic

3. Axiomatic Semantics in Alloy

- The events and axiomatic rules over the **po**, **rf** and **hb** relations can be encoded in the Alloy solver
- Alloy automatically generates executions satisfying the semantics
- For a given program, Alloy enumerates all executions allowed by the memory model

```
sig Address {}

sig Thread {}

abstract sig Event {
    po: set Event,
    hb: set Event,
    thr: one Thread
}

abstract sig MemoryEvent extends Event {
    address: one Address
}

sig Write extends MemoryEvent {
    val_written: one Int,
    rf: set Read
}

sig Read extends MemoryEvent {
    val_read: one Int
}

sig InitialValue extends Write {
```

Events in Alloy

```

sig Address {}

sig Thread {}

abstract sig Event {
    po: set Event,
    hb: set Event,
    thr: one Thread
}

abstract sig MemoryEvent extends Event {
    address: one Address
}

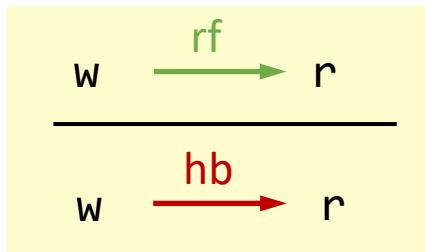
sig Write extends MemoryEvent {
    val_written: one Int,
    rf: set Read
}

sig Read extends MemoryEvent {
    val_read: one Int
}

sig initialValue extends Write {}

```

Rules in Alloy



```

fact { all w, r: Event |
    r in rf[w]
    implies
    r in hb[w]
}

```

```

sig Address {}

sig Thread {}

abstract sig Event {
    po: set Event,
    hb: set Event,
    thr: one Thread
}

abstract sig MemoryEvent extends Event {
    address: one Address
}

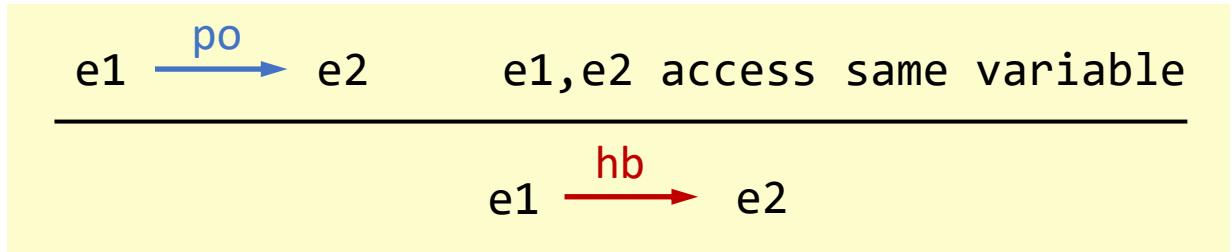
sig Write extends MemoryEvent {
    val_written: one Int,
    rf: set Read
}

sig Read extends MemoryEvent {
    val_read: one Int
}

sig initialValue extends Write {}

```

Rules in Alloy



```

fact { all disj e1, e2: MemoryEvent |
    (e2 in po[e1] and address[e1] = address[e2])
    implies
    e2 in hb[e1]
}

```

```

sig Address {}

sig Thread {}

abstract sig Event {
    po: set Event,
    hb: set Event,
    thr: one Thread
}

abstract sig MemoryEvent extends Event {
    address: one Address
}

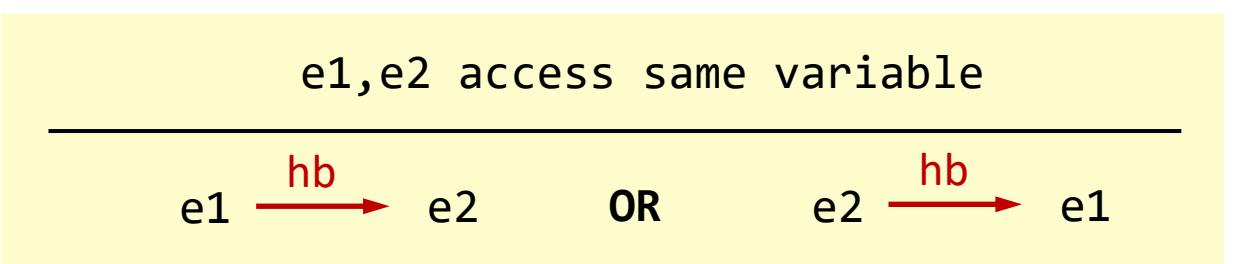
sig Write extends MemoryEvent {
    val_written: one Int,
    rf: set Read
}

sig Read extends MemoryEvent {
    val_read: one Int
}

sig initialValue extends Write {}

```

Rules in Alloy



```

fact {all disj e1, e2: MemoryEvent |
    (address[e1] = address[e2])
    implies
    ((e2 in hb[e1]) or (e1 in hb[e2])))

```

```

sig Address {}

sig Thread {}

abstract sig Event {
    po: set Event,
    hb: set Event,
    thr: one Thread
}

abstract sig MemoryEvent extends Event {
    address: one Address
}

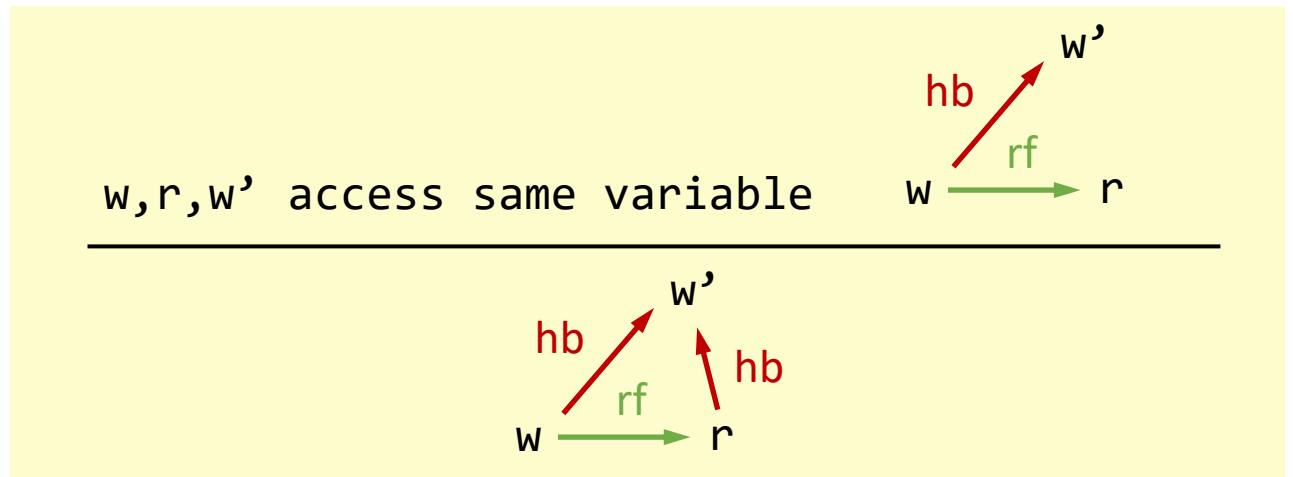
sig Write extends MemoryEvent {
    val_written: one Int,
    rf: set Read
}

sig Read extends MemoryEvent {
    val_read: one Int
}

sig initialValue extends Write {}

```

Rules in Alloy



```

fact { all r: Read, w': Write |
    ((address[r] = address[w']) and
     (w' in hb[~rf[r]]))
    .
    implies
    (w' in hb[r])
}

```

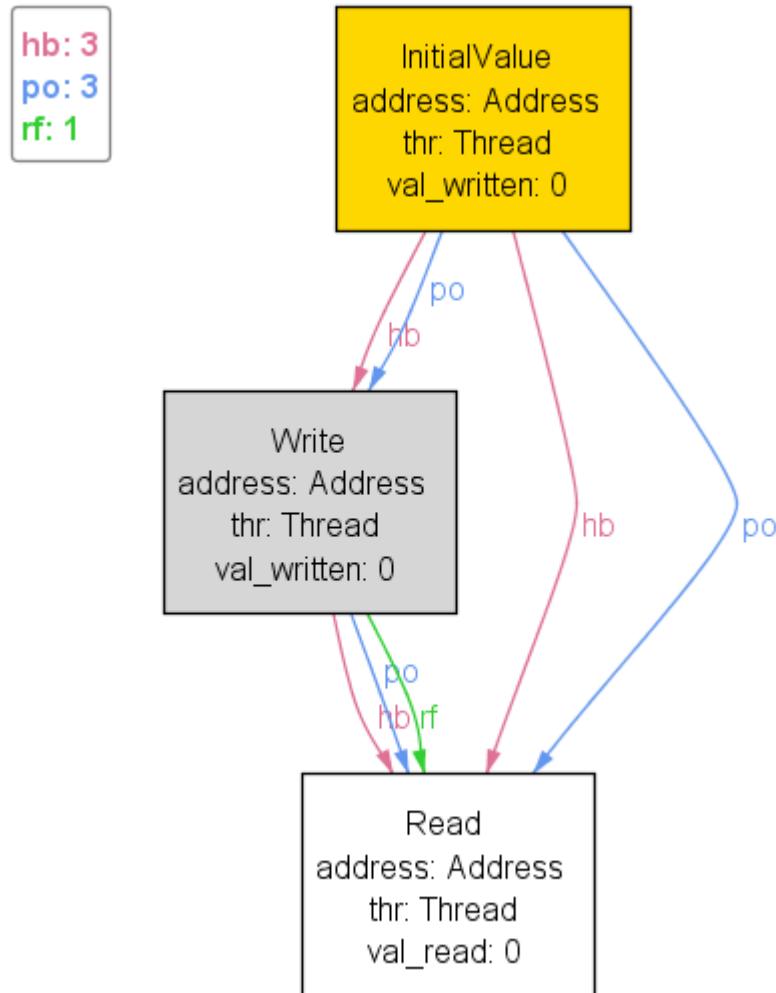
Additional Constraints

- Each read event reads from one write
- **po** and **hb** are transitively closed and acyclic
- InitialValue properties (happen before all other events)
- **po** between events of the same thread
- Events connected by **rf** have the same address and the same value

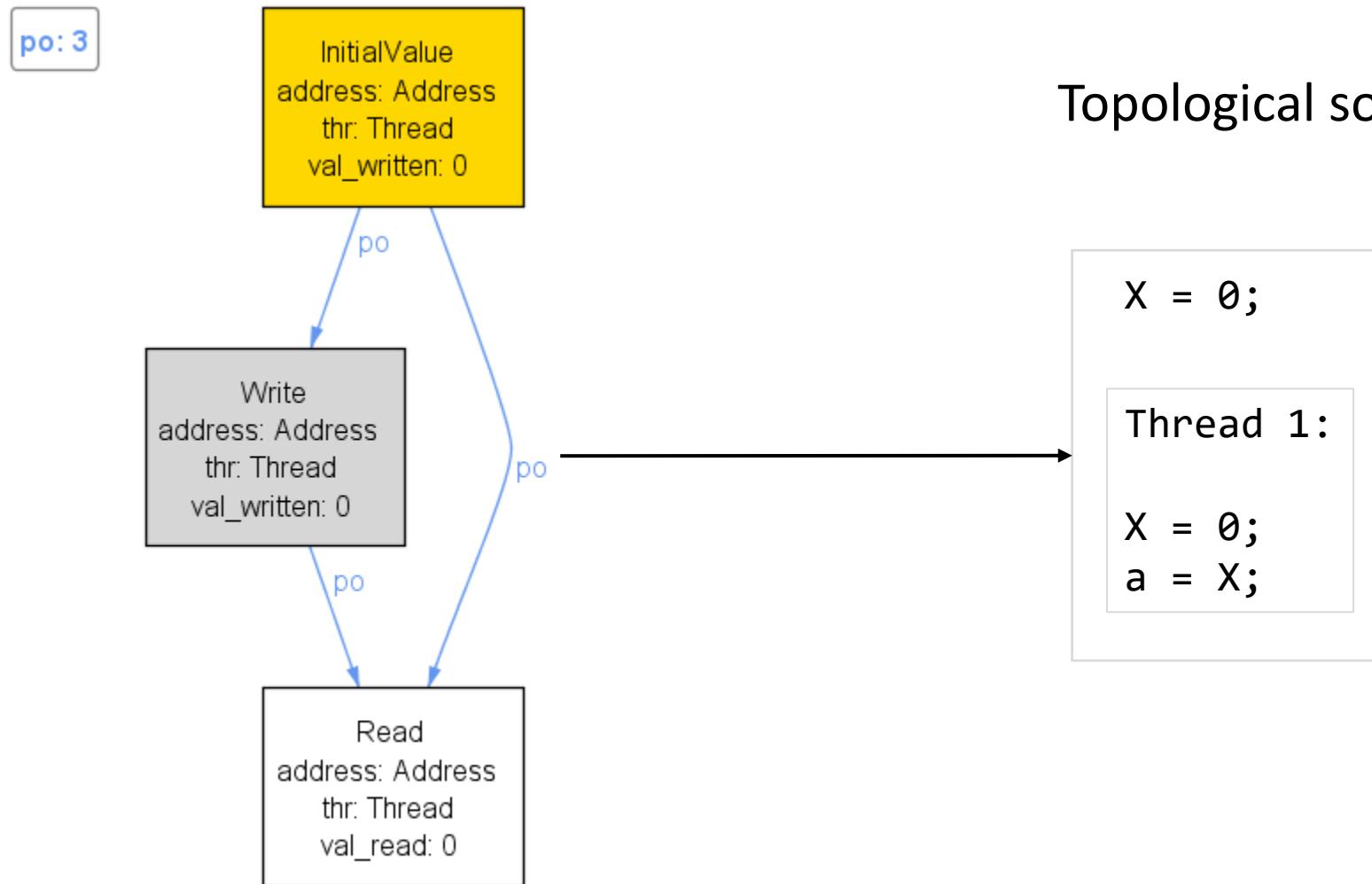
Alloy-Generated Executions

Execution Parameters:

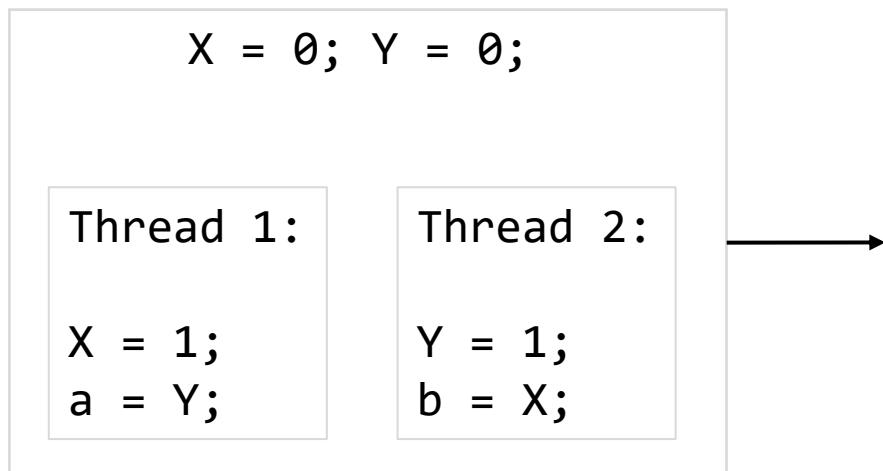
```
pred gen1 {  
    #Write = 2 and  
    #Read = 1 and  
    #Thread = 1  
}  
run gen1 for 3
```



From Executions to Programs



Analysing a Given Program



```
pred prog {
    some disj t1, t2: Thread,
    disj X, Y: Address,
    disj Xi, Yi: initialValue,
    disj wX, wY: Write,
    disj rY, rX: Read |
    /* Initial X = 0, Y = 0*/
    val_written[Xi] = 0 and address[Xi] = X and thr[Xi] = t1 and
    val_written[Yi] = 0 and address[Yi] = Y and thr[Yi] = t1 and
    /* Thread 1: X = 1; read Y */
    val_written[wX] = 1 and address[wX] = X and thr[wX] = t1 and
    address[rY] = Y and thr[rY] = t1 and rY in po[wX] and
    /* Thread 2: Y = 1; read X */
    val_written[wY] = 1 and address[wY] = Y and thr[wY] = t2 and
    address[rX] = X and thr[rX] = t2 and rX in po[wY]
}

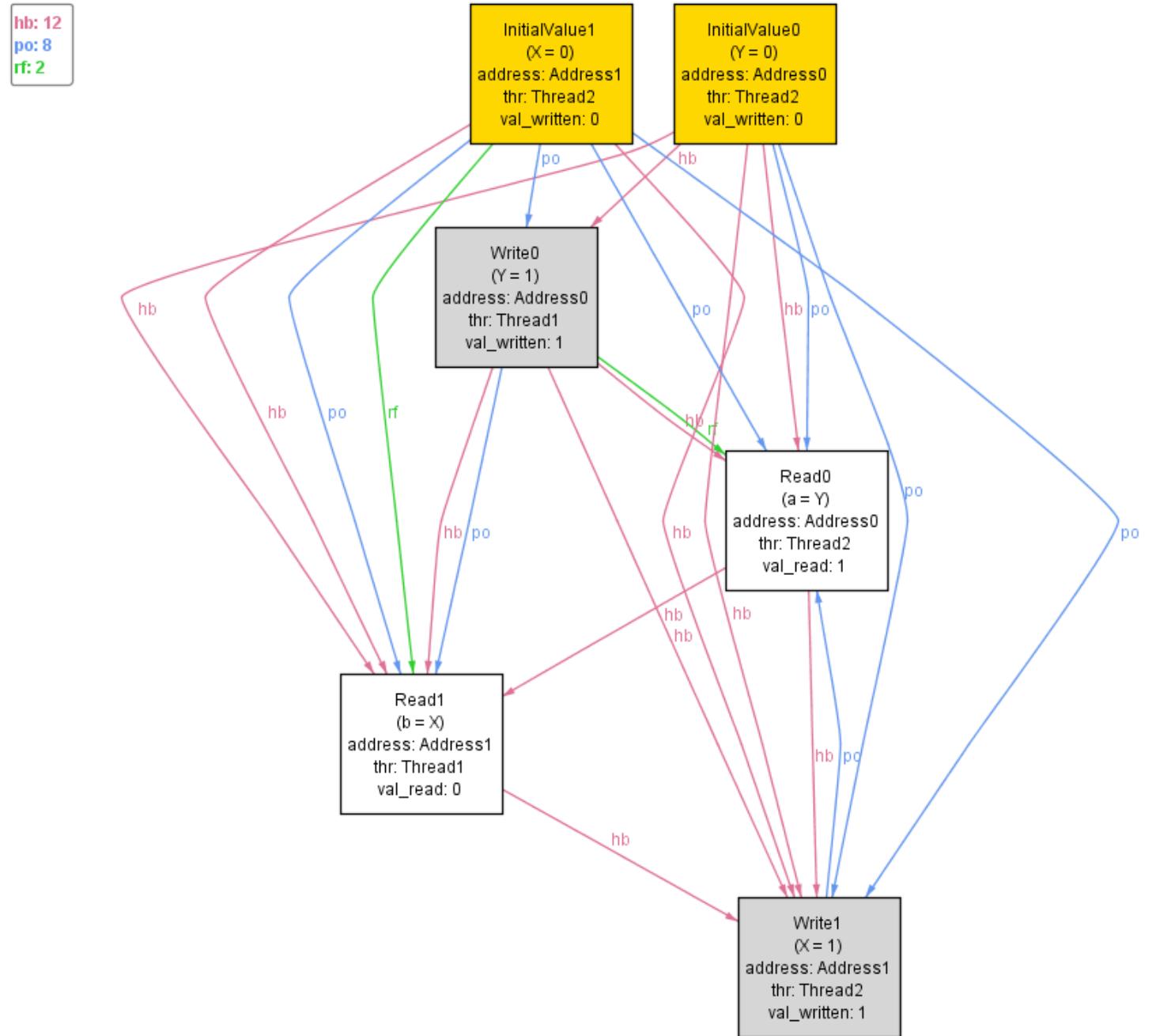
run prog for 6
```

```

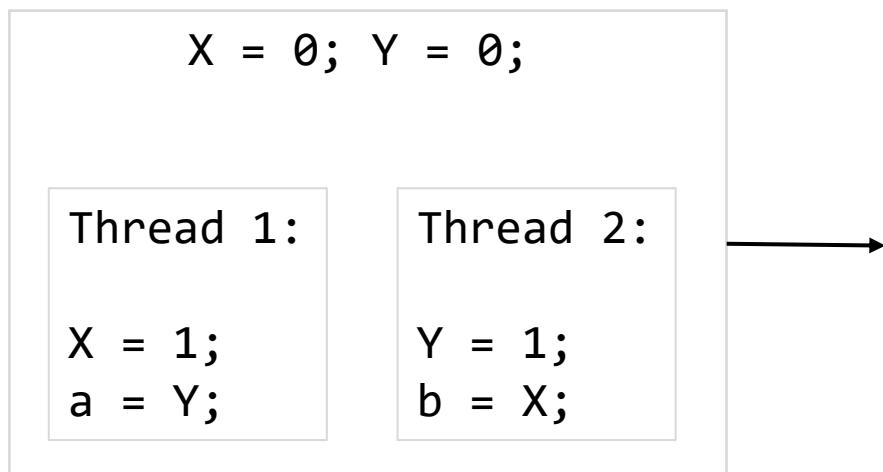
X = 0; Y = 0;

Thread 1:           Thread 2:
X = 1;
a = Y;
Y = 1;
b = X;

```

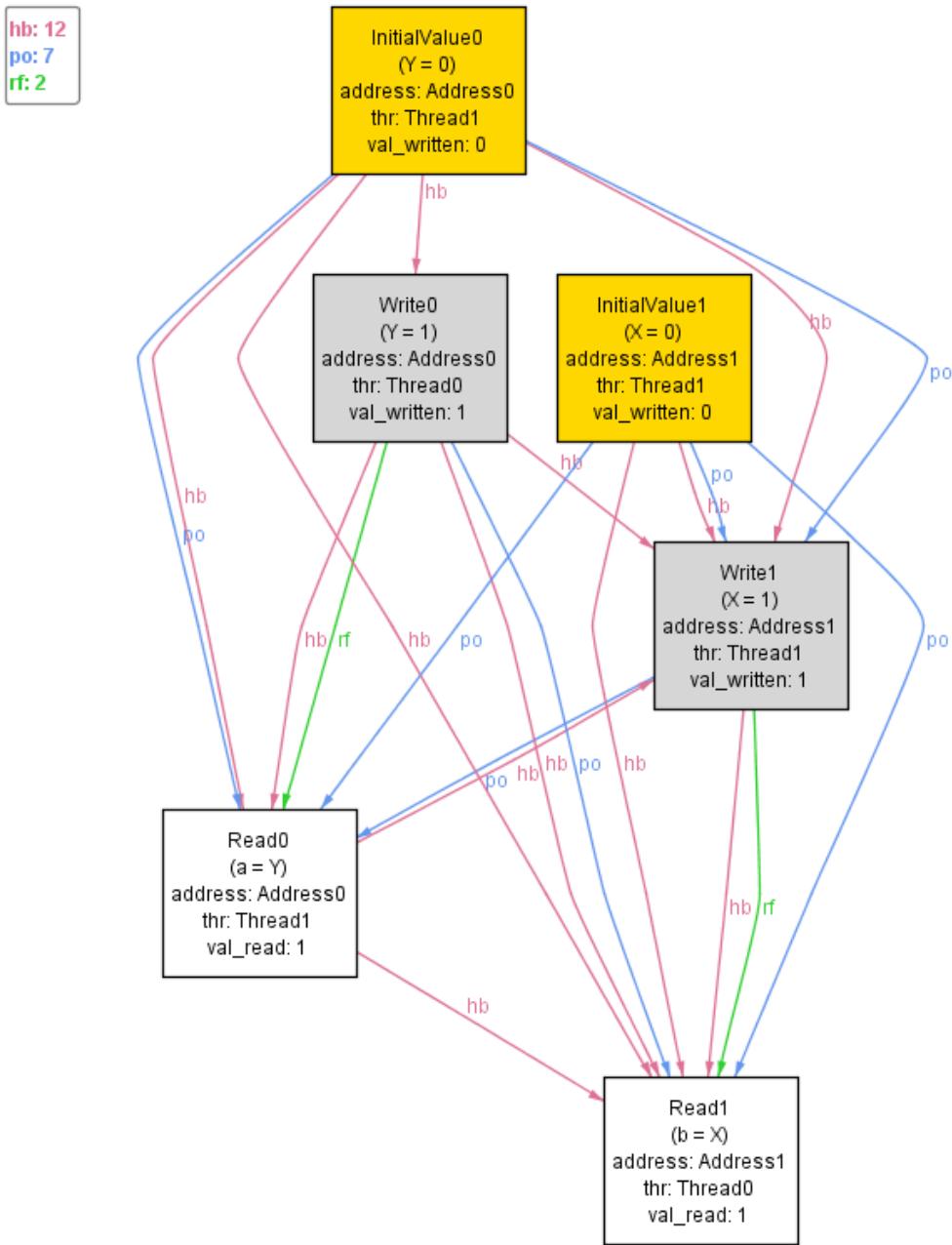
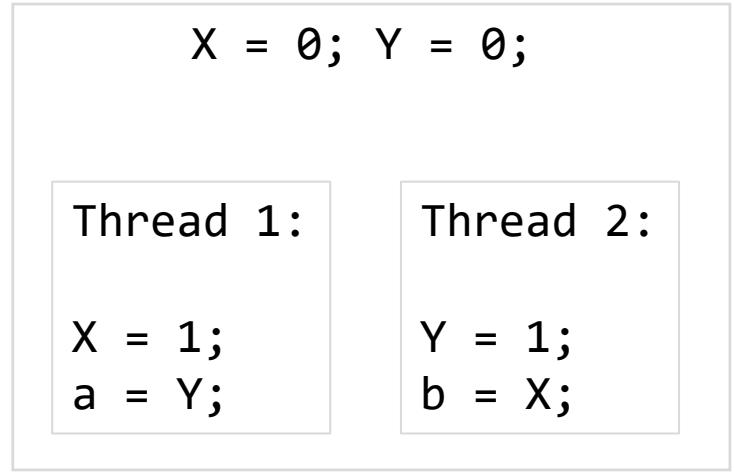


Analysing a Given Program

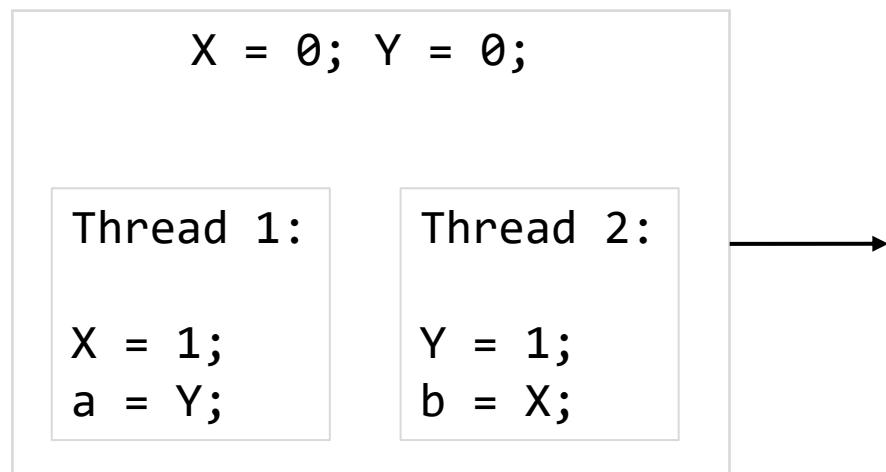


```
pred prog {
    some disj t1, t2: Thread,
    disj X, Y: Address,
    disj Xi, Yi: initialValue,
    disj wX, wY: Write,
    disj rY, rX: Read |
    /* Initial X = 0, Y = 0*/
    val_written[Xi] = 0 and address[Xi] = X and thr[Xi] = t1 and
    val_written[Yi] = 0 and address[Yi] = Y and thr[Yi] = t1 and
    /* Thread 1: X = 1; read Y */
    val_written[wX] = 1 and address[wX] = X and thr[wX] = t1 and
    address[rY] = Y and thr[rY] = t1 and rY in po[wX] and
    /* Thread 2: Y = 1; read X */
    val_written[wY] = 1 and address[wY] = Y and thr[wY] = t2 and
    address[rX] = X and thr[rX] = t2 and rX in po[wY]
    and not (val_read[rX] = 0 and val_read[rY] = 1)
}
```

run prog for 6



Analysing a Given Program



```
pred prog {
    some disj t1, t2: Thread,
        disj X, Y: Address,
        disj Xi, Yi: initialValue,
        disj wX, wY: Write,
        disj rY, rX: Read |
    /* Initial X = 0, Y = 0*/
    val_written[Xi] = 0 and address[Xi] = X and thr[Xi] = t1 and
    val_written[Yi] = 0 and address[Yi] = Y and thr[Yi] = t1 and
    /* Thread 1: X = 1; read Y */
    val_written[wX] = 1 and address[wX] = X and thr[wX] = t1 and
    address[rY] = Y and thr[rY] = t1 and rY in po[wX] and
    /* Thread 2: Y = 1; read X */
    val_written[wY] = 1 and address[wY] = Y and thr[wY] = t2 and
    address[rX] = X and thr[rX] = t2 and rX in po[wY]
    and not (val_read[rX] = 0 and val_read[rY] = 1)
    and not (val_read[rX] = 1 and val_read[rY] = 1)
    and not (val_read[rX] = 1 and val_read[rY] = 0)
    and not (val_read[rX] = 0 and val_read[rY] = 0)
}
```

run prog for 6

Question – Generating Executions

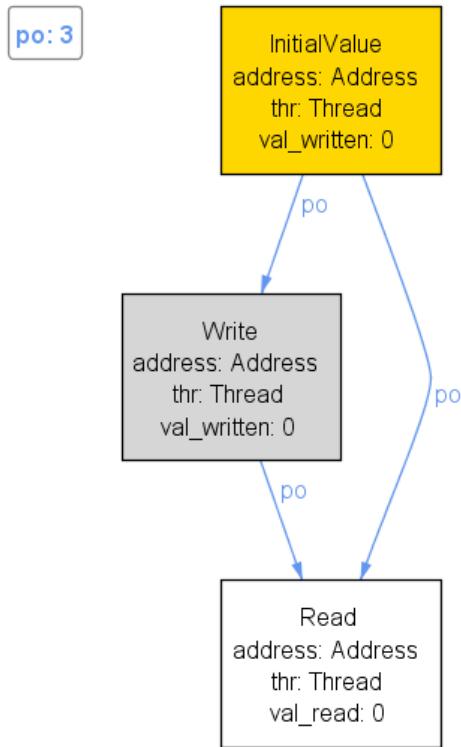
How to generate executions that are allowed by x86 TSO and not by Sequential Consistency?

In general, encode both models in Alloy and generate executions where the **hb** of Sequential Consistency has a cycle, whereas the **hb** of x86 TSO is acyclic

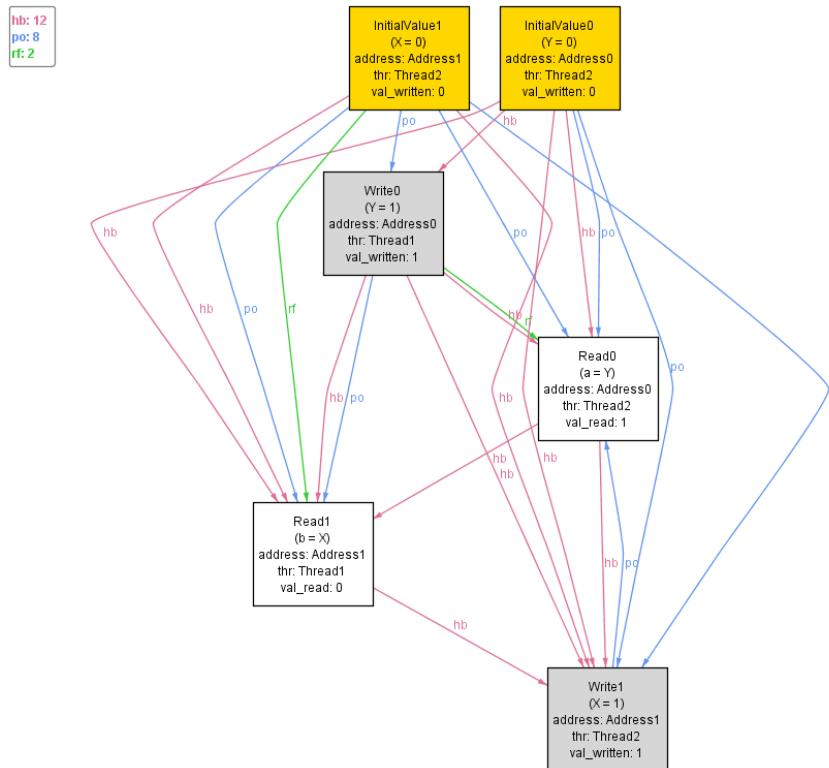
In this case, possible to generate x86 TSO executions where **po** does not imply **hb**

Axiomatic Semantics in Alloy - Summary

Alloy-Generated Executions



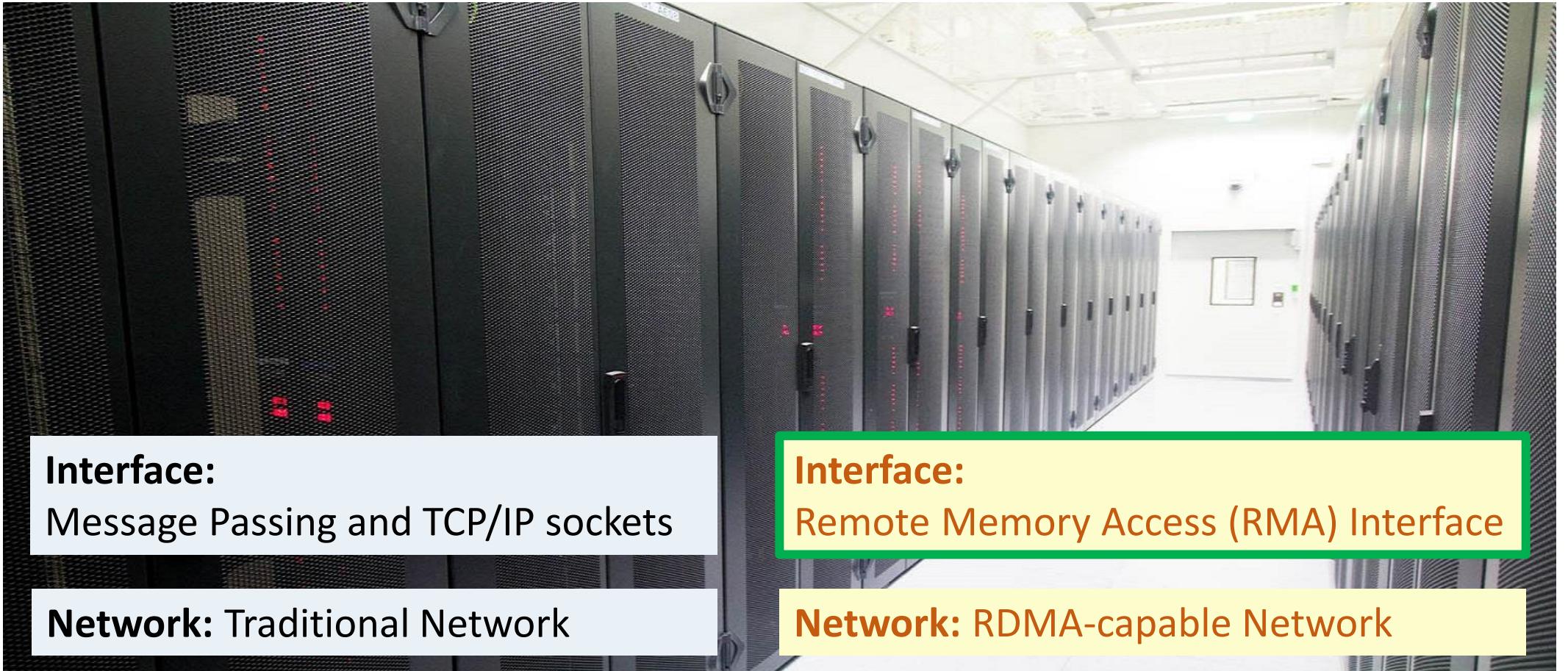
Analysing a Given Program – All Possible Read Values



4. Memory Model Evaluation Framework

- Based on (informal) documentation, build a formal Remote Memory Access (RMA) model in Alloy
- Automatically generate tests, together with their allowed executions by the formal RMA model
- Run tests on the real RMA networks and observe executions
- Compare real executions with executions allowed by the model

4. Remote Memory Access (RMA) Networks



Interface:

Message Passing and TCP/IP sockets

Network: Traditional Network

Interface:

Remote Memory Access (RMA) Interface

Network: RDMA-capable Network

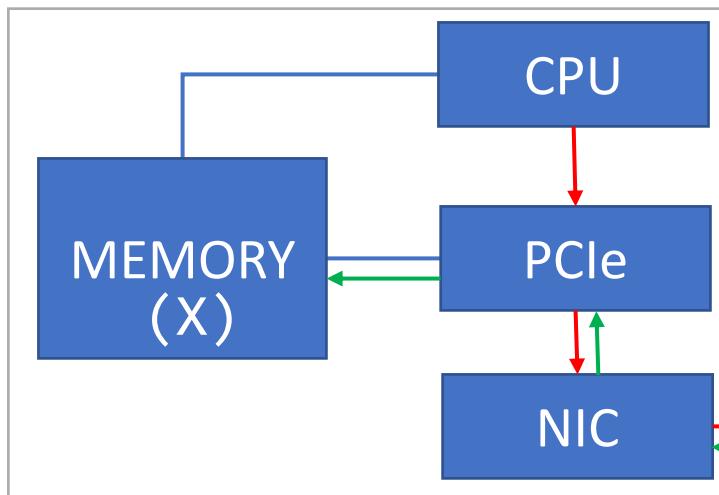
RMA Overview

`dmapp_get_nbi(X, Y, P2);`



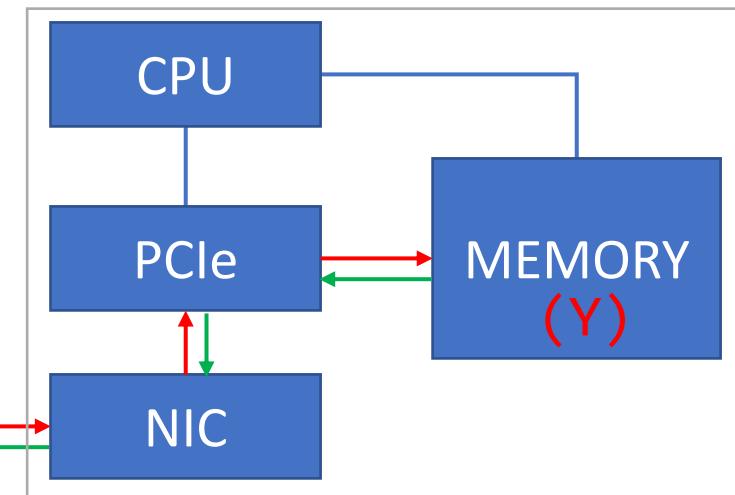
RMA Overview

`dmapp_get_nbi(X, Y, P2);`



Low latency

High bandwidth



Widely supported (**Cray Aries and Gemini, Infiniband, IBM Blue Gene, IBM Percs**)

Problem

Semantics are specific to network technologies

IBV Verbs (Infiniband), DMAPI (Cray), Portals 4 API

No standard memory model has been established yet

Informal memory semantics in the existing documentation

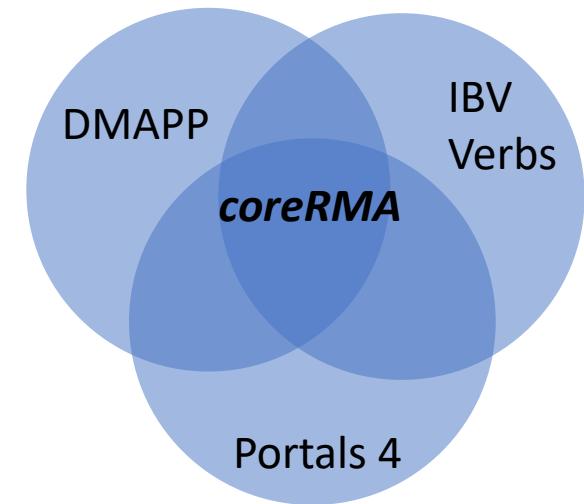
This Work

coreRMA

- Language
- Axiomatic semantics
- Formalize essential characteristics of RMA programming

Evaluation Framework

- Find contradictions between existing doc and real behavior
- How precise is *coreRMA*?



coreRMA Language

Local Read	<code>r = X;</code>
Local Write	<code>X = r;</code>
Remote Get	<code>X = get(Y, P#);</code>
Remote Put	<code>put(Y, P#, X);</code>
Remote Get-Accumulate	<code>X = rga(Y, P#, Z);</code>
Remote Compare-and-Swap	<code>X = cas(Z, P#, Y, W);</code>
Flush	<code>flush(P#);</code>

Function Mappings

<i>coreRMA</i>	<i>IBV Verbs (Infiniband)</i>	<i>DMAPP (Cray)</i>	<i>Portals 4 API</i>
put	<code>ibv_wr_rdma_write</code>	<code>dmapp_put_nbi</code>	<code>PtlPut</code>
get	<code>ibv_wr_rdma_read</code>	<code>dmapp_get_nbi</code>	<code>PtlGet</code>
flush	<code>ibv_reg_notify_cq</code>	<code>dmapp_gsync_wait</code>	<code>PtlCTWait</code>

Example RMA Program

(X = 1)

Process 1:

a = X;

(Y = 0)

Process 2:

```
put(X, P1, Y);  
Y = get(X, P1);  
flush(P1);  
b = Y;
```

Overview of RMA Semantics

Statements

`X = get(Y, P1);`

Events

`eread(Y, P1)`

`write(X);`

`put(X, P1, Y);`

`read(Y)`

`ewrite(X, P1);`

Overview of RMA Semantics

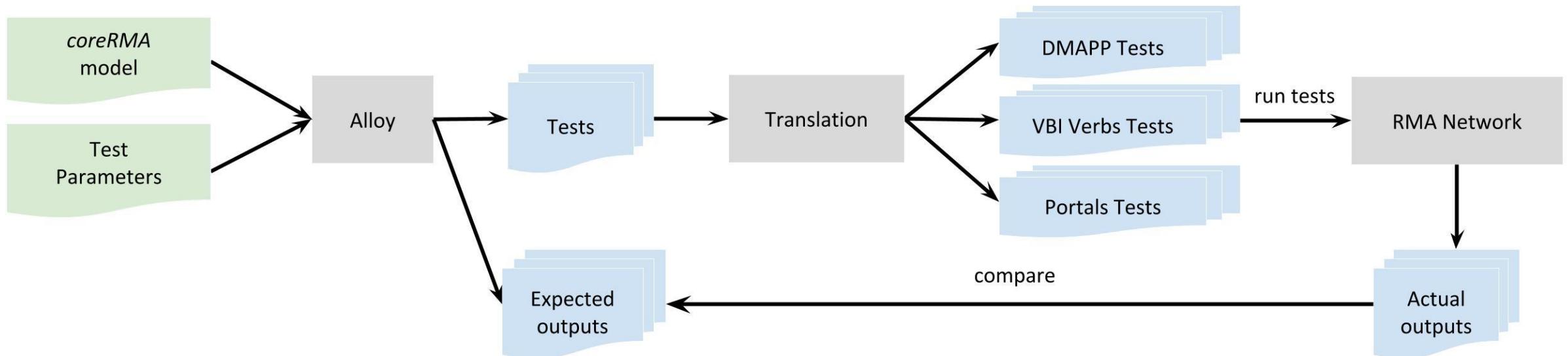
Rule for In-Order Routing

$$\frac{e1 \xrightarrow{\text{po}} e2 \quad e1, e2 - \text{same remote process}}{e1 \xrightarrow{\text{hb}} e2}$$

e1 and e2 are remote actions (**eread** for **get**, **ewrite** for **put**)

14 rules defining **hb**, based on **po** and **rf**.

coreRMA Validation Framework

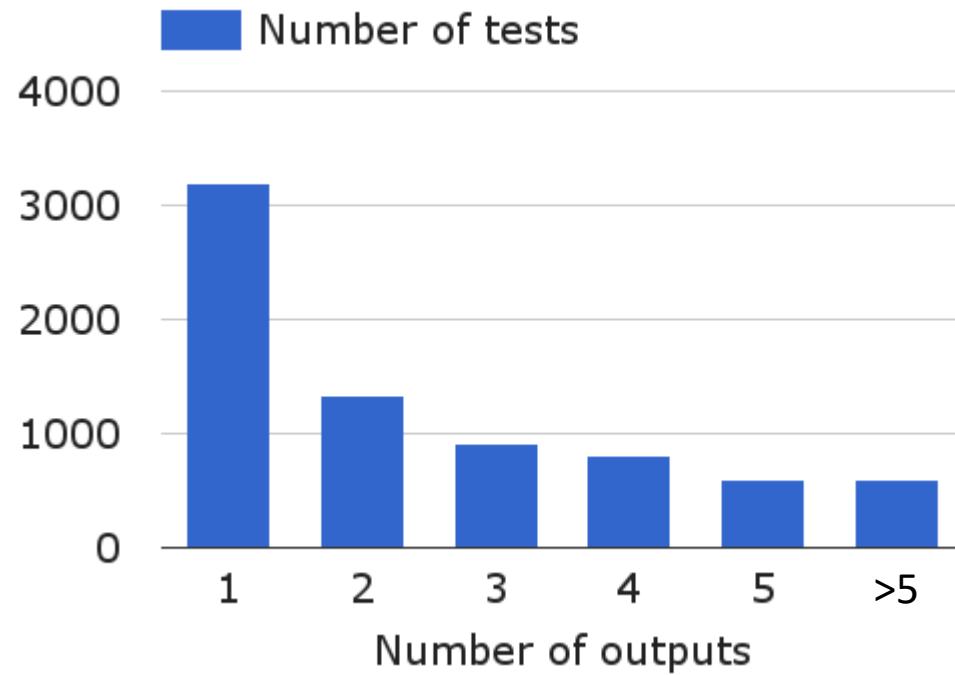


Test Generation

Automatically generated 7411 tests (1 or 2 processes).

Each test executed **10000** times (~**20s/test** including connection setup).

Expected Outputs



Question - Generating Useful Tests

How to generate tests that have a higher chance to detect inconsistencies between the coreRMA model and the real network executions?

(X = 1)

Process 1:

a = X;

(Y = 0)

Process 2:

```
put(X, P1, Y);
Y = get(X, P1);
flush(P1);
b = Y;
```

At least one local read from each variable

Local writes should have distinct values

Tests focus on one rule of the axiomatic semantics

RQ1) Contradictions between the existing documentation and the real RMA networks?

Yes – DMAPP API (Cray Aries) 135 tests have unexpected outputs

Why? In-Order Routing guarantees not respected

We reported the concrete specification inconsistencies to Cray Inc. - **Confirmed**

Action: documentation to be updated

RQ2) How precisely does *coreRMA* model real networks?

<i>Library</i>	<i>Observed Outputs (%)</i>
DMAPP	89.9
IBV Verbs	94.5
Portals 4	89.7

RQ2) How precisely does *coreRMA* model real networks?

Non-triggerable behaviors

Introduce **random delays** after each statement.

We cannot **control the scheduler** for actions of the same remote statement (*put, get, rga, cas*).

RQ2) How precisely does *coreRMA* model real networks?

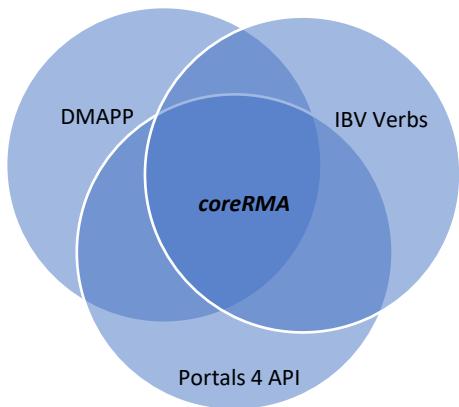
Additional network guarantees

Example

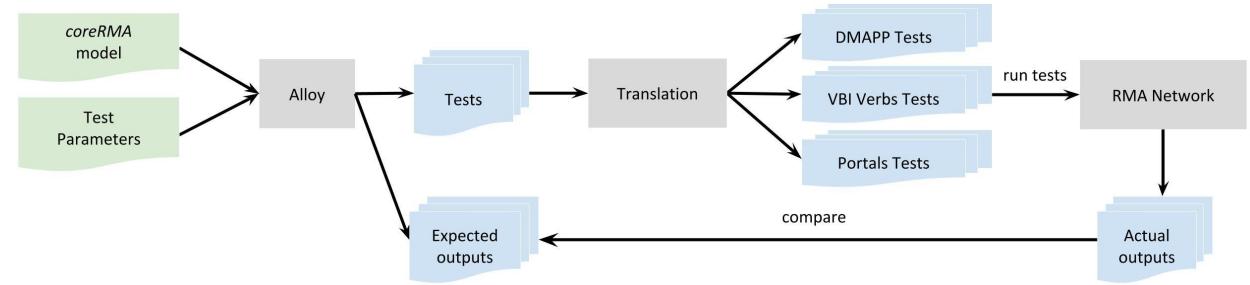
For DMAPP, for data smaller than a parametric threshold, ***put*** statements provide more guarantees.

Remote Memory Access (RMA) Networks - Summary

coreRMA Semantics



Evaluation Framework



Precise Model

<i>Library</i>	<i>Observed Outputs (%)</i>
DMAPP	89.9
IBV Verbs	94.5
Portals 4	89.7

Related Work

Automatically Comparing Memory Consistency Models. POPL '17

J. Wickerson, M. Batty, T. Sorensen, G. A. Constantinides

A Formal Analysis of the NVIDIA PTX Memory Consistency Model. ASPLOS '19

D. Lustig, S. Sahasrabuddhe, O. Giroux

RISC-V Memory Consistency Model

RISC-V Memory Model Task Group

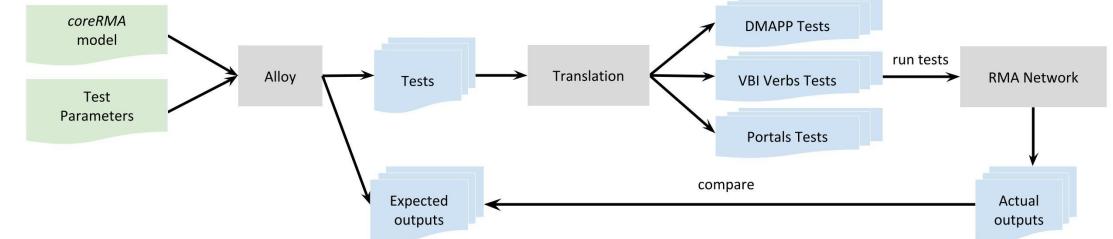
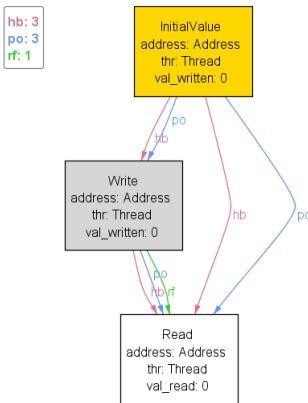
PerSeVerE: Persistency Semantics for Verification under Ext4. POPL '21

M. Kokologiannakis, I. Kaysin, A. Raad, V. Vafeiadis

Summary

Memory Model in Alloy → **Allowed Executions** → **Evaluation Framework**

```
sig Address {}  
sig Thread {}  
  
abstract sig Event {  
    po: set Event,  
    hb: set Event,  
    thr: one Thread  
}  
  
abstract sig MemoryEvent extends Event {  
    address: one Address  
}  
  
sig Write extends MemoryEvent {  
    val_written: one Int,  
    rf: set Read  
}  
  
sig Read extends MemoryEvent {  
    val_read: one Int  
}  
  
sig initialValue extends Write {  
}
```



Rigorous Software Engineering

Modelling and Specification

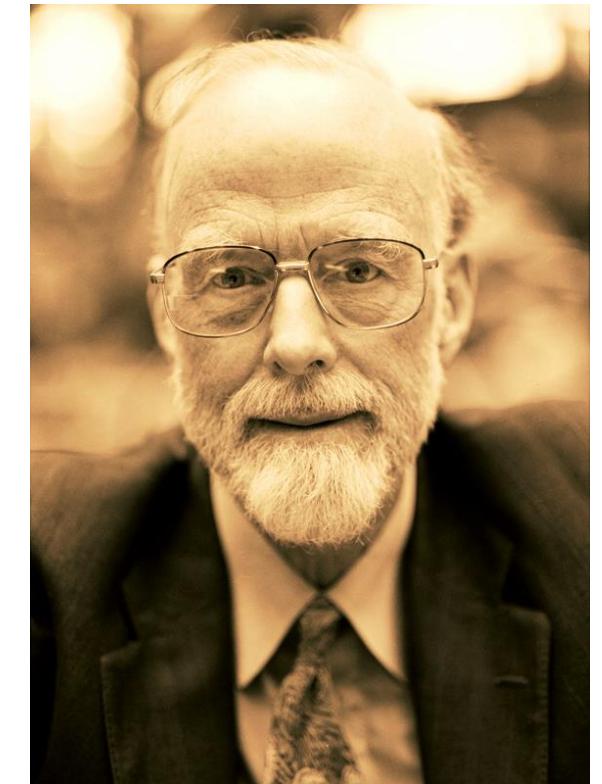
Prof. Martin Vechev

Why Declarative Design?

I conclude there are two ways of constructing a software design.

One way is to make it so simple there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

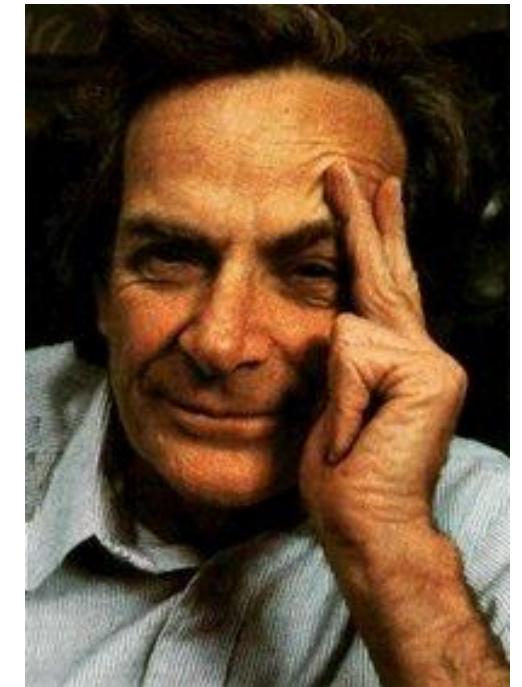
– Tony Hoare [Turing Award Lecture, 1980]



Why Automated Analysis?

The first principle is that you must not fool yourself, and you are the easiest person to fool.

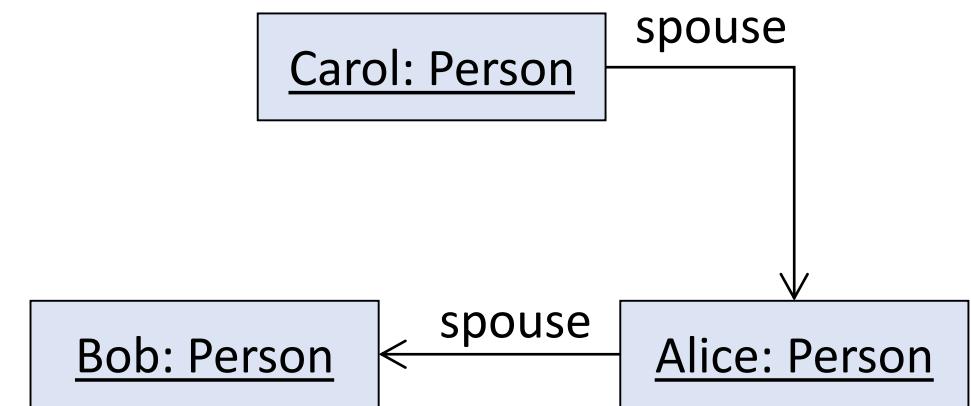
– Richard P. Feynman



Formal Modeling

- Notations and tools are based on mathematics, hence precise
- Typically used to describe some aspect of a system
- Formal models enable automatic analysis

- Finding ill-formed examples
- Checking properties



```
context SavingsAccount inv:  
self.amount >= 0
```

Alloy: language and analyzer

- Alloy is a formal modeling language based on [set theory](#)
- An Alloy model specifies a [collection of constraints](#) that describe a set of structures
- The [Alloy Analyzer](#) is a solver that takes the constraints of a model and finds structures that satisfy them
 - Generate sample structures [sampling]
 - Generate counterexamples for invalid properties
 - Visualize structures

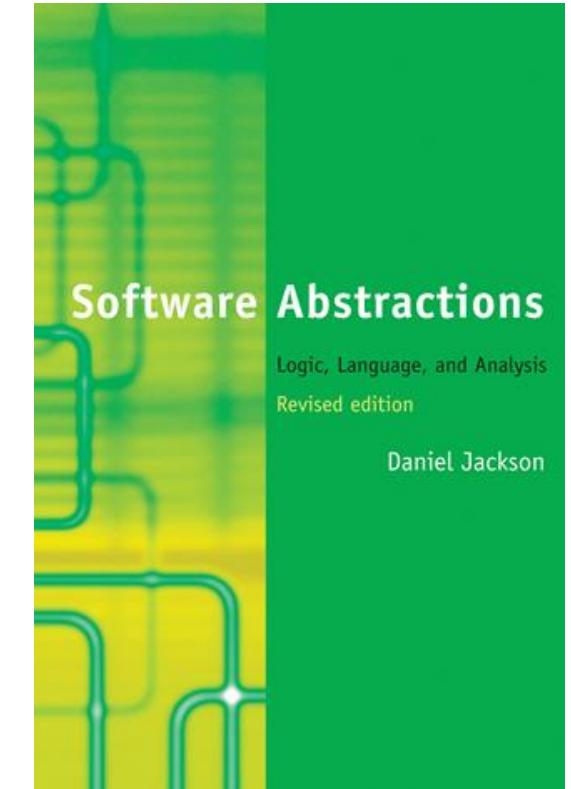
Alloy Documentation and Download

■ Documentation

- Useful tutorials available at <https://alloytools.org>
- Book by Daniel Jackson

■ Download

- Get latest version at <https://alloytools.org/download.html>
- Requires JRE

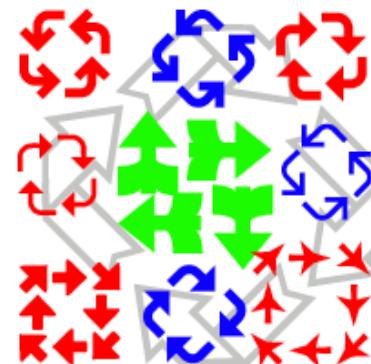


Modeling and Specification

- Logic
 - Static Models
 - Dynamic Models
 - Analyzing Models
-

Four Key Ideas . . .

1) Everything is a relation



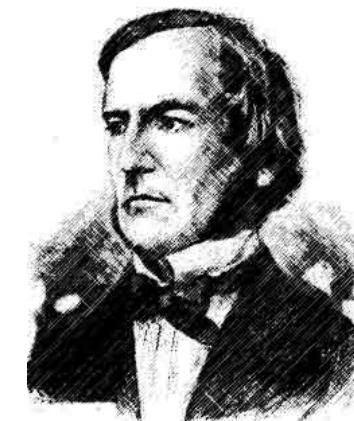
2) Non-specialized logic



3) Counterexamples & scope



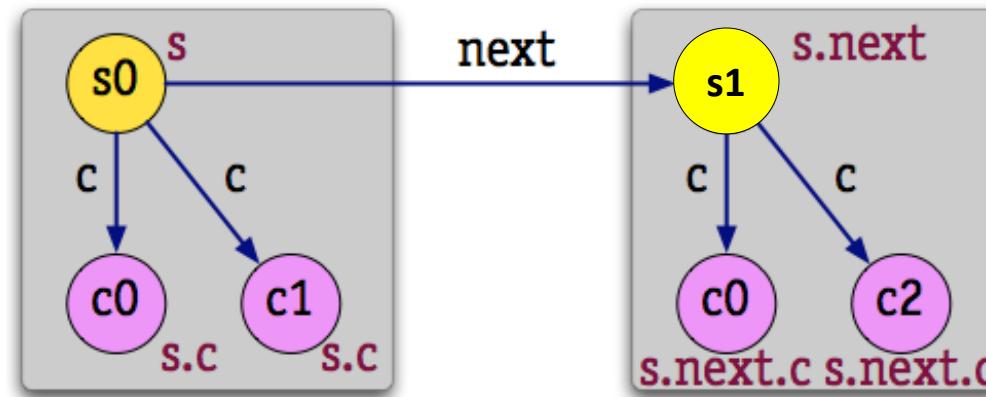
4) Analysis by SAT



George Boole

1) Everything is a Relation

- Alloy uses relations for
 - all data types – even sets, scalars, tuples
 - structures in space and time
- key operator is **dot** join
 - relational join
 - field navigation
 - function application

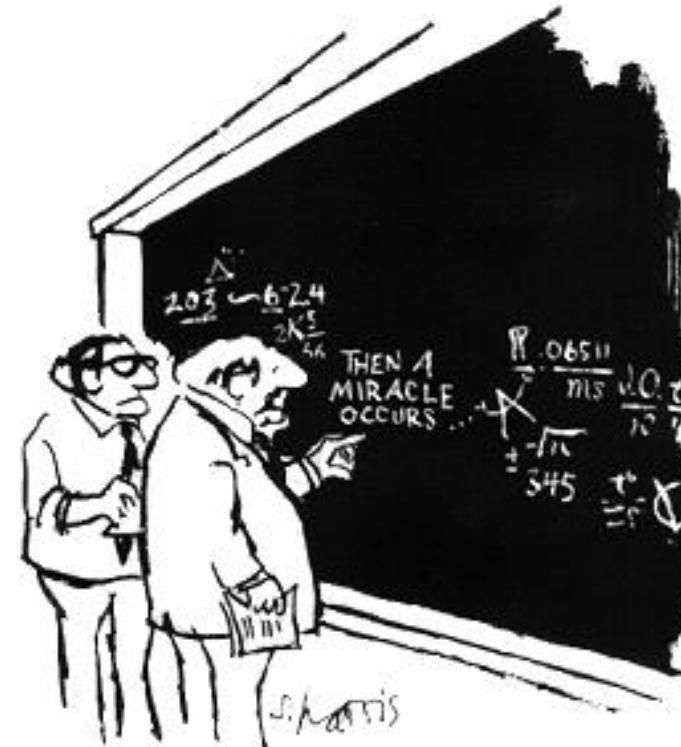


Why Relations?

- easy to understand
 - binary relation is a graph or mapping
- easy to analyze
 - first order (tractable)
- uniform

2) Non-specialized logic

- No special constructs for state machines, traces, synchronization, concurrency . . .

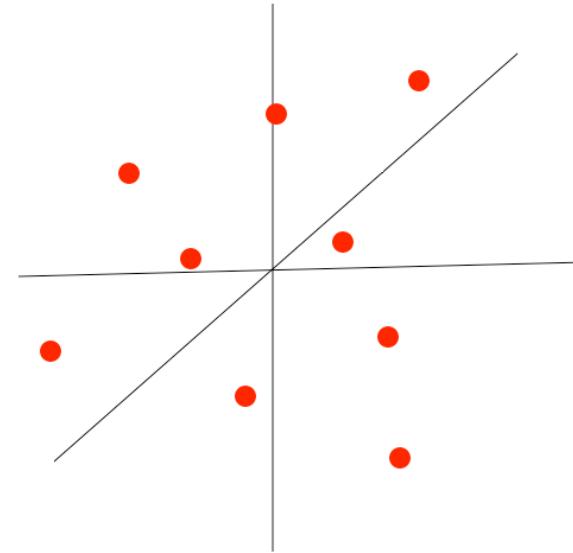


"I think you should be more explicit here in step two."

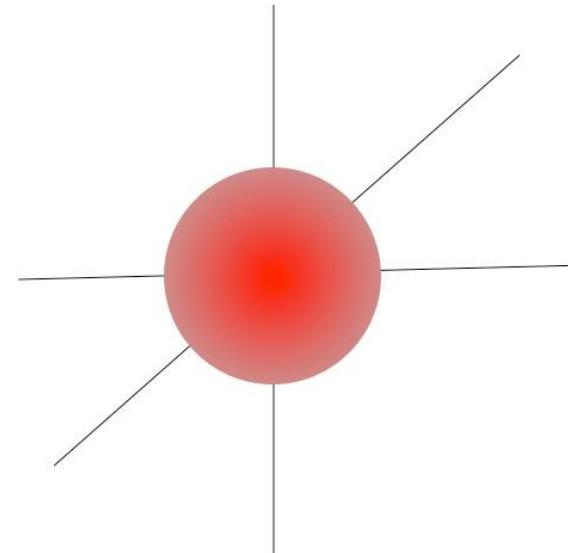
3) Counterexamples & Scope

- observations about design analysis:

- most assertions are wrong
- most flaws have small counterexamples ("Small Scope Hypothesis")



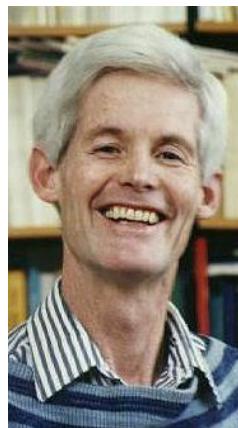
testing:
a few cases of arbitrary size



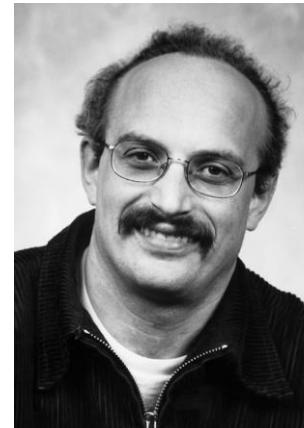
scope-complete:
all cases within a small bound

4) Analysis by SAT

- SAT, the quintessential hard problem (Cook 1971)
 - SAT is hard, so reduce SAT to your problem
- SAT, the universal constraint solver (Kautz, Selman, ... 1990's)
 - SAT is easy, so reduce your problem to SAT
 - solvers: Chaff (Malik), Berkmin (Goldberg & Novikov), ...



Stephen Cook



Eugene Goldberg



Henry Kautz



Sharad Malik



Yakov Novikov

Example: Modeling “ceilings and floors”

```
sig Platform {}
```

there are “Platform” things

```
sig Man {ceiling, floor: Platform}
```

each Man has a ceiling and a floor Platform

```
pred Above [m, n: Man] {m.floor = n.ceiling}
```

Man m is “above” Man n if m's floor is n's ceiling

```
fact {all m: Man | some n: Man | Above[n,m] }
```

“One Man's Ceiling Is Another Man's Floor”

Example: Checking “ceilings and floors”

```
assert BelowToo {  
    all m: Man | some n: Man | Above [m,n]  
}
```

"One Man's Floor Is Another Man's Ceiling"?

```
check BelowToo for 2  
    check "One Man's Floor Is Another Man's Ceiling"  
    counterexample with 2 or less platforms and men?
```

Alloy = Logic + Language + Analysis

- Logic
 - first order logic + relational calculus
- Language
 - syntax for structuring specifications in the logic
- Analysis
 - bounded exhaustive search for counterexample to a claimed property using SAT

Logic: Relations of Atoms

- Atoms are Alloy's primitive entities
 - indivisible, immutable, uninterpreted
- Relations associate atoms with one another
 - set of tuples, tuples are sequences of atoms
- Every value in Alloy logic is a relation!
 - relations, sets, scalars all the same thing

Logic: Everything's a Relation

- sets are unary (1 column) relations

```
Name = {(N0), (N1), (N2)}  
Addr = {(A0), (A1), (A2)}  
Book = {(B0), (B1)}
```

- scalars are singleton sets

```
myName = {(N1)}  
yourName = {(N2)}  
myBook = {(B0)}
```

- binary relation

```
names = {(B0, N0),  
(B0, N1),  
(B1, N2)}
```

- ternary relation

```
addrs = {(B0, N0, A0),  
(B0, N1, A1),  
(B1, N1, A2),  
(B1, N2, A2)}
```

Logic: Relations

```
addrs = {(B0, N0, A0), (B0, N1, A1),  
          (B1, N1, A2), (B1, N2, A2)}
```

B0	N0	A0
B0	N1	A1
B1	N1	A2
B1	N2	A2

size = 4

arity = 3

- rows are unordered
- columns are ordered but unnamed
- all relations are first-order
 - relations cannot contain relations, no sets of sets

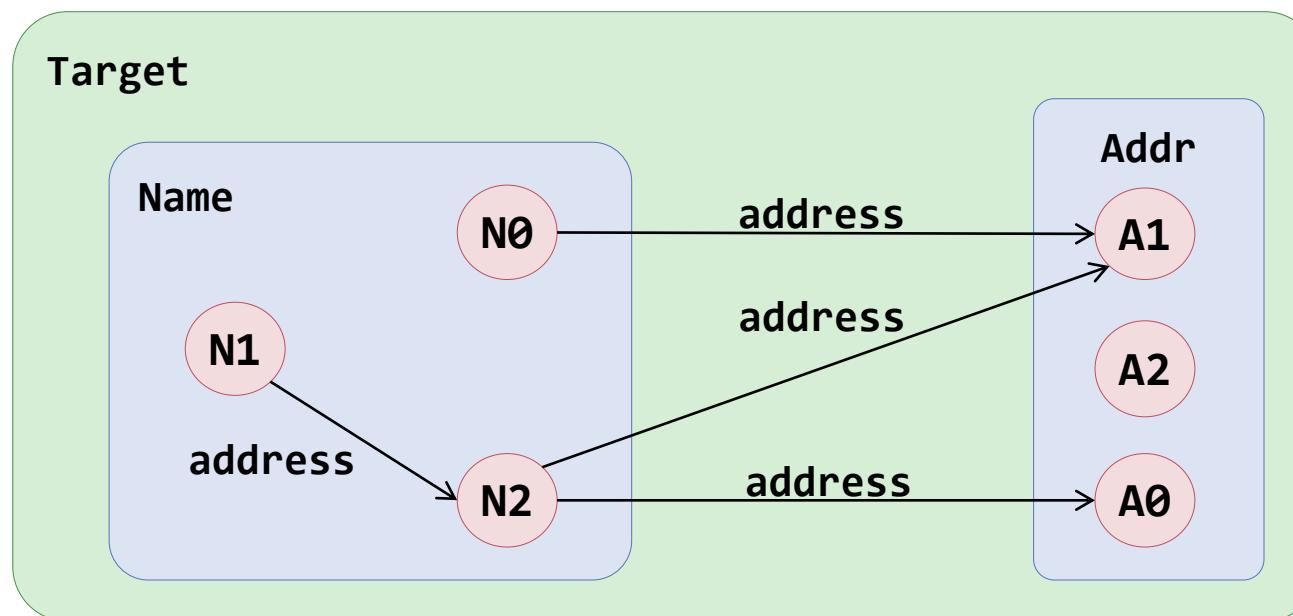
Logic: Address Book Example

Name = $\{(N_0), (N_1), (N_2)\}$

Addr = $\{(A_0), (A_1), (A_2)\}$

Target = $\{(N_0), (N_1), (N_2), (A_0), (A_1), (A_2)\}$

address = $\{(N_0, A_1), (N_1, N_2), (N_2, A_1), (N_2, A_0)\}$



Logic: Constants

none	<i>empty set</i>
univ	<i>universal set</i>
iden	<i>identity relation</i>

```
Name = {(N0), (N1), (N2)}
```

```
Addr = {(A0), (A1)}
```

```
none = {}
```

```
univ = {(N0), (N1), (N2), (A0), (A1)}
```

```
iden = {(N0, N0), (N1, N1), (N2, N2),  
        (A0, A0), (A1, A1)}
```

logic: set operators

<code>+</code>	<i>union</i>
<code>&</code>	<i>intersection</i>
<code>-</code>	<i>difference</i>
<code>in</code>	<i>subset</i>
<code>=</code>	<i>equality</i>

```
greg = {(N0)}
rob = {(N1)}
```

```
greg + rob = {(N0), (N1)}
greg = rob = false
rob in none = false
```

Name	=	{(N0), (N1), (N2)}
Alias	=	{(N1), (N2)}
Group	=	{(N0)}
RecentlyUsed	=	{(N0), (N2)}
Alias + Group	=	{(N0), (N1), (N2)}
Alias & RecentlyUsed	=	{(N2)}
Name - RecentlyUsed	=	{(N1)}
RecentlyUsed in Alias	=	false
RecentlyUsed in Name	=	true
Name = Group + Alias	=	true

```
cacheAddr = {(N0, A0), (N1, A1)}
diskAddr = {(N0, A0), (N1, A2)}
```

```
cacheAddr + diskAddr =
cacheAddr & diskAddr =
cacheAddr = diskAddr =
```

logic: product operator

->

cross product

```
Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

Name->Addr =          { (N0, A0), (N0, A1),
                           (N1, A0), (N1, A1) }

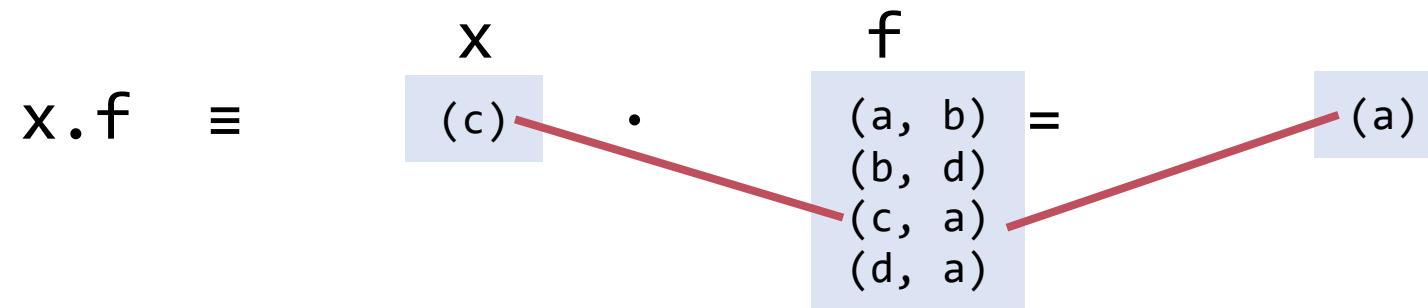
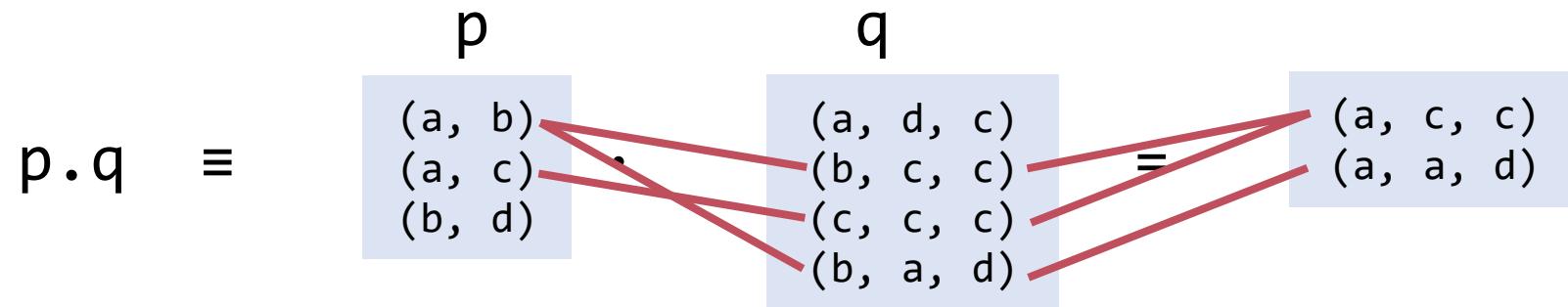
Book->Name->Addr =    { (B0, N0, A0), (B0, N0, A1),
                           (B0, N1, A0), (B0, N1, A1) }
```

```
b   = {(B0)}
b' = {(B1)}
address = {(N0, A0), (N1, A1)}
address' = {(N2, A2)}
```

b->b' =

b->address + b'->address' =

logic: relational join



logic: join operators

.
 dot join
 [] *box join*

e1[e2] = e2.e1
 a.b.c[d] = d.(a.b.c)

```

Book = {(B0)}
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
Host = {(H0), (H1)}

myName = {(N1)}
myAddr = {(A0)}

address =      {(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)}
host =         {(A0, H0), (A1, H1), (A2, H1)}

Book.address =      {(N0, A0), (N1, A0), (N2, A2)}
Book.address[myName] =  {(A0)}
Book.address.myName =  {}

host[myAddr] =
address.host =

```

logic: restriction and override

<: *domain restriction*
:> *range restriction*
++ *override*

```
p ++ q =
p - (domain[q] <: p) + q
```

```

Name      = {(N0), (N1), (N2)}
Alias     = {(N0), (N1)}
Addr      = {(A0)}
address   = {(N0, N1), (N1, N2), (N2, A0)}

address :> Addr  = {(N2, A0)}
Alias <: address = address :> Name  = {(N0, N1), (N1, N2)}
address :> Alias = {(N0, N1)}

workAddress = {(N0, N1), (N1, A0)}
address ++ workAddress = {(N0, N1), (N1, A0), (N2, A0)}

```

```
m' = m ++ (k -> v)
update map m with key-value pair (k, v)
```

logic: unary operators

\sim transpose
 $^\wedge$ transitive closure
 $*$ reflexive transitive closure
apply only to binary relations

$^r = r + r.r + r.r.r + \dots$
 $*r = \text{idem} + ^r$

```

Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
          (N1, N2), (N1, N3),
          (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
          (N1, N1), (N1, N2), (N1, N3),
          (N2, N2), (N2, N3), (N3, N3)}
    
```

```

first = {(N0)}
rest = {(N1), (N2), (N3)}

first.^next = rest
first.*next = Node
    
```

logic: boolean operators

!	not	<i>negation</i>
&&	and	<i>conjunction</i>
	or	<i>disjunction</i>
=>	implies	<i>implication</i>
	else	<i>alternative</i>
<=>	iff	<i>bi-implication</i>

four equivalent constraints:

$F \Rightarrow G \text{ else } H$

$F \text{ implies } G \text{ else } H$

$(F \And G) \Or ((\Not F) \And H)$

$(F \And G) \Or ((\Not F) \And H)$

logic: quantifiers

all *F holds for every x in e*

some *F holds for at least one x in e*

no *F holds for no x in e*

lone *F holds for at most one x in e*

one *F holds for exactly one x in e*

all $x: e \mid F$

all $x: e_1, y: e_2 \mid F$

all $x, y: e \mid F$

all disj $x, y: e \mid F$

some $n: \text{Name}, a: \text{Address} \mid a \text{ in } n.\text{address}$
some name maps to some address — address book not empty

no $n: \text{Name} \mid n \text{ in } n.^{\wedge}\text{address}$

all $n: \text{Name} \mid \text{lone } a: \text{Address} \mid a \text{ in } n.\text{address}$

all $n: \text{Name} \mid \text{no disj } a, a': \text{Address} \mid (a + a') \text{ in } n.\text{address}$

logic: quantified expressions

some e *e has at least one tuple*
no e *e has no tuples*
lone e *e has at most one tuple*
one e *e has exactly one tuple*

some Name
set of names is not empty

some address
address book is not empty – it has a tuple

no (address.Addr - Name)
nothing is mapped to addresses except names

all n: Name | **lone** n.address
every name maps to at most one address

logic: comprehensions

$$\{x_1: e_1, x_2: e_2, \dots, x_n: e_n \mid F\}$$
$$\{n: \text{Name} \mid \text{no } n.^{\wedge}\text{address} \& \text{ Addr}\}$$

set of names that don't resolve to any actual addresses

$$\{n: \text{Name}, a: \text{Address} \mid n \rightarrow a \text{ in } ^{\wedge}\text{address}\}$$

binary relation mapping names to reachable addresses

logic: if and let

```
f implies e1 else e2
let x = e | formula
let x = e | expression
```

four equivalent constraints:

```
all n: Name |
  (some n.workAddress implies n.address = n.workAddress
   else n.address = n.homeAddress)

all n: Name |
  let w = n.workAddress, a = n.address |
    (some w implies a = w else a = n.homeAddress)

all n: Name |
  let w = n.workAddress |
    n.address = (some w implies w else n.homeAddress)

all n: Name |
  n.address = (let w = n.workAddress |
    (some w implies w else n.homeAddress))
```

logic: cardinalities

#r	<i>number of tuples in r</i>
0, 1, ...	<i>integer literal</i>
+	<i>plus</i>
-	<i>minus</i>

=	<i>equals</i>
<	<i>less than</i>
>	<i>greater than</i>
=<	<i>less than or equal to</i>
>=	<i>greater than or equal to</i>

sum x: e | ie
sum of integer expression ie for all singletons x drawn from e

all b: Bag | #b.marbles =< 3
all bags have 3 or less marbles

#Marble = **sum** b: Bag | #b.marbles
the sum of the marbles across all bags
equals the total number of marbles

RSE – Project

<https://www.sri.inf.ethz.ch/teaching/rse-project-2021>

```
public final class Printer {  
    // total number of copies printed by all printers ever  
    public static int nTotalCopies = 0;  
    // upper limit on the number of copies printed in a single call to print  
    private final int nLimit;  
  
    public Printer(int nLimit) {  
        this.nLimit = nLimit;  
    }  
  
    public void print(int n) {  
        assert n >= 0;                  // check NON_NEGATIVE  
        assert n <= this.nLimit;        // check RESPECTS_LIMIT  
  
        // print copies  
        Printer.nTotalCopies += n;  
  
        // assert Printer.nTotalCopies >= nFlyers; // check ENOUGH_COPIES (only upon program termination)  
    }  
}
```

```
// expected results:
```

```
// NON_NEGATIVE SAFE
```

```
// RESPECTS_LIMIT SAFE
```

```
// ENOUGH_COPIES SAFE
```

```
public class Basic_Test_Safe {
```

```
    static final int nFlyers = 6;
```

```
    public static void m1() {
```

```
        Printer p = new Printer(3);
```

```
        p.print(3);
```

```
        p.print(3);
```

```
}
```

```
}
```

```
// expected results:
```

```
// NON_NEGATIVE UNSAFE  
// RESPECTS_LIMIT UNSAFE  
// ENOUGH_COPIES UNSAFE
```

```
public class Basic_Test_Unsafe {
```

```
    static final int nFlyers = 3;
```

```
    public void m2(int j) {
```

```
        Printer p = new Printer(2);
```

```
        if (-1 <= j && j <= 3) {
```

```
            p.print(j);
```

```
        }
```

```
}
```

```
}
```

Frameworks

- APRON (for numerical analysis)
 - Soot (for pointer analysis and parsing Java code)
 - Docker, GitLab CI/CD, JaCoCo, Maven, SLF4J
-
- Getting familiar with these is part of the assignment
 - We provide resources in the project description

Quick Demo

- Web interface to master solution
<http://rseproject21.ethz.ch/rse-project>
- Project Webpage
<https://www.sri.inf.ethz.ch/teaching/rse-project-2021>
- Skeleton in GitLab
 - GitLab CI/CD (Continuous Integration)
 - Pipeline
 - Test coverage
- Skeleton in Visual Studio Code
 - README
 - Docker
 - Maven
 - Test Coverage Report
 - Run tests inside Visual Studio Code

Rigorous Software Engineering

Testing

Martin Vechev

Slides adapted from previous years
(special thanks to Peter Müller, Felix Friedrich, Hermann Lehner)

Slides partly based on the courses
“Software Engineering I” by Prof. Bernd Brügge, TU München and
“Software Engineering” by Prof. Jan Vitek, Purdue University

Why Does Software Contain Bugs?

- Our ability to predict the behavior of our implementations is limited
 - Software is extremely complex
 - No developer can understand the whole system

- We make mistakes
 - Unclear requirements, miscommunication
 - Wrong assumptions (e.g., behavior of operating system)
 - Design errors (e.g., capacity of data structure too small)
 - Coding errors (e.g., wrong loop condition)

Increasing Software Reliability

Fault Avoidance

- Detect faults statically without executing the program
- Includes development methodologies, reviews, and program verification

Fault Detection

- Detect faults by executing the program
- Includes testing

Fault Tolerance

- Recover from faults at runtime (e.g., transactions)
- Includes adding redundancy (e.g., n-version programming)

Goal of Testing

- An error is a deviation of the observed behavior from the required (desired) behavior
 - Testing is a process of executing a program with the intent of finding an error
 - A successful test is a test that finds errors
-

Limitations of Testing

*Testing can only show the presence of bugs,
not their absence.*

[E. W. Dijkstra]

- It is impossible to completely test any nontrivial module or any system
 - Theoretical limitations: termination
 - Practical limitations: prohibitive in time and cost
-

5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.4 Structural Testing

Test Stages

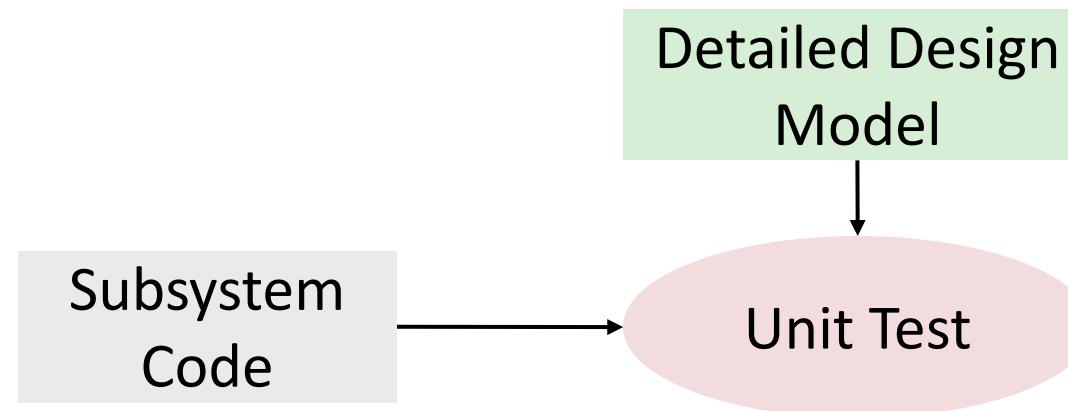
- Unit Tests

Also (not discussed here):

- Integration Tests
 - System Tests
-

Unit Testing

- Testing individual subsystems (collection of classes, or single class)



- Goal: Confirm that subsystem is correctly coded and has the intended functionality

Unit Test Example (JUnit)

```
class SavingsAccount {  
    ...  
    public void deposit( int amount ) { ... }  
    public void withdraw( int amount ) { ... }  
    public int getBalance( ) { ... }  
}  
  
@Test  
public void withdrawTest( ) {  
    SavingsAccount target = new SavingsAccount();  
    target.deposit( 300 );  
    int amount = 100;  
    target.withdraw( amount );  
    Assert.assertTrue( target.getBalance( ) == 200 );  
}
```

Implement
test driver

Create
test data

Create
test oracle

Unit Testing: Discussion

- To achieve a reasonable test coverage, one has to test each method with several inputs
 - To cover valid and invalid inputs
 - To cover different paths through the method

```
@Test  
public void withdrawTest( ) {  
    SavingsAccount target = new SavingsAccount();  
    target.deposit( 500 );  
    int amount = 0;  
    target.withdraw( amount );  
    Assert.assertTrue( target.getBalance( ) == 500 );  
}
```

Boiler-plate code
for creating test
data and writing
test oracles

Parameterized Unit Tests

- Parameterized test methods take arguments for test data
 - Decouple test driver (logic) from test data

```
[ Test ]  
public void withdrawTest( int balance, int amount ) {  
    SavingsAccount target = new SavingsAccount();  
    target.deposit( balance );  
    target.withdraw( amount );  
    Assert.IsTrue( target.getBalance( ) == balance - amount );  
}
```

- Test data can be specified as values, ranges, or random values
- Requires generic test oracles

Generic Test Oracles: Example

```
public static void bubbleSort( int[ ] a ) {  
    for( int i = 0; i < a.Length - 1; i++ ) {  
        for( int j = i + 1; j < a.Length; j++ ) {  
            if( a[ i ] > a[ j ] )  
                { int tmp = a[ i ]; a[ i ] = a[ j ]; a[ j ] = tmp; }  
        }  
    }  
}
```

```
[ Test ]  
public void bubbleSortTest( ) {  
    int[ ] a = { 7, 2, 5, 2 };  
  
    bubbleSort( a );  
  
    int[ ] expected = { 2, 2, 5, 7 };  
    Assert.AreEqual( expected, a );  
}
```

Create
test data

Create
test oracle

Generic Test Oracles: Example

```
[ Test ]  
public void bubbleSortTest( int[ ] a ) {  
    int[ ] original = ( int[ ] ) a.Clone();  
  
    bubbleSort( a );  
  
    for( int i = 0; i < a.Length - 1; i++ )  
        Assert.IsTrue( a[ i ] <= a[ i+1 ] );  
  
    bool[ ] visited = new bool[ a.Length ];  
    for( int i = 0; i < a.Length; i++ ) {  
        int j;  
        for ( j = 0; j < a.Length; j++ ) {  
            if( !visited[ j ] && a[ i ] == original[ j ] )  
                { visited[ j ] = true; break; }  
        }  
        Assert.IsFalse( j == a.Length );  
    }  
}
```

Save test data for later comparison

Check that array is sorted

Check that array is a permutation of original array

Value a[i] is not in the original array

Parameterized Unit Tests: Discussion

- Parameterized unit tests avoid boiler-plate code
- Writing generic test oracles is sometimes difficult
 - Analogous to writing strong postconditions
- Still several test methods are needed, for instance, for valid and invalid input
- Parameterized unit tests are especially useful when test data is generated automatically (see later)

5. Testing

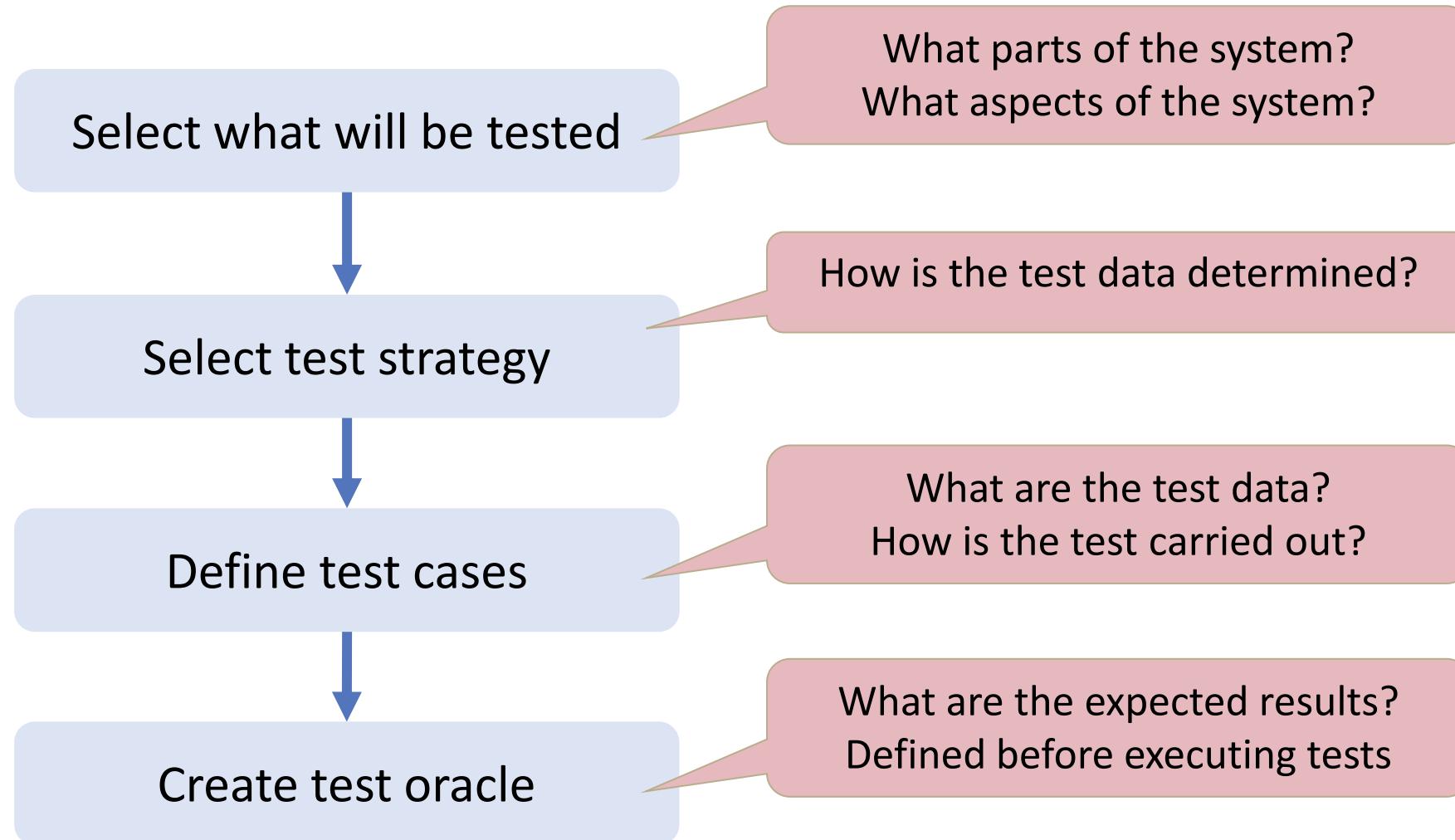
5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.4 Structural Testing

Testing Steps



Example: Solve Quadratic Equation

```
void roots( double a, double b, double c ) {  
    double q = b*b - 4*a*c;  
    if( q > 0 && a != 0 ) {  
        numRoots = 2;  
        double r = Math.sqrt( q );  
        x1 = (-b + r) / (2 * a);  
        x2 = (-b - r) / (2 * a);  
    } else if( q == 0 ) {  
        numRoots = 1;  
        x1 = -b / (2 * a);  
    } else {  
        numRoots = 0;  
    }  
}
```

Fails if $a==0$ and
 $b^2-4ac == 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Strategy 1: Exhaustive Testing

- Check UUT for all possible inputs
 - Not feasible, even for trivial programs

```
void roots( double a, double b, double c ) {  
    ...  
}
```

- Assuming that **double** represents 64-bit values, we get $(2^{64})^3 \approx 10^{58}$ possible values for a, b, c
- Programs with heap data structures have a much larger state space!

Strategy 2: Random Testing

- Select test data uniformly

```
void roots( double a, double b, double c ) {  
    double q = b*b - 4*a*c;  
    if( q > 0 && a != 0 ) {  
        ...  
    } else if( q == 0 ) {  
        numRoots = 1;  
        x1 = -b / (2 * a);  
    } else { ... }  
}
```

Fails if $a==0$ and
 $b^2-4ac == 0$

The likelihood of selecting
 $a==0$ and $b==0$
randomly is $1/10^{38}$

Random Testing: Observations

- Random testing focuses on generating test data **fully automatically**
- Advantages
 - Avoids designer/tester bias
 - Tests robustness, especially handling of invalid input and unusual actions
- Disadvantages
 - Treats all inputs as equally valuable

Strategy 3: Functional Testing

- Use requirements knowledge to determine test cases

Given three values, a , b , c ,
compute all solutions of the
equation $ax^2 + bx + c = 0$

Two solutions	One solution	No solution
$a \neq 0$ and $b^2-4ac > 0$	$a = 0$ and $b \neq 0$ or $a \neq 0$ and $b^2-4ac = 0$	$a = 0$, $b = 0$, and $c \neq 0$ or $a \neq 0$ and $b^2-4ac < 0$

Test each case of the specification

Functional Testing: Observations

- Functional testing focuses on input/output behavior
 - Goal: Cover all the requirements
- Attempts to find
 - Incorrect or missing functions
 - Interface errors
 - Performance errors
- Limitations
 - Does not effectively detect design and coding errors (e.g., buffer overflow, memory management)
 - Does not reveal errors in the specification (e.g., missing cases)

Strategy 4: Structural Testing

- Use **design knowledge** about system structure, algorithms, data structures to determine test cases that exercise a large portion of the code

```
void roots( double a, double b, double c ) {  
    double q = b*b - 4*a*c;  
    if( q > 0 && a != 0 ) {  
        ...  
    } else if( q == 0 ) {  
        ...  
    } else {  
        ...  
    }  
}
```

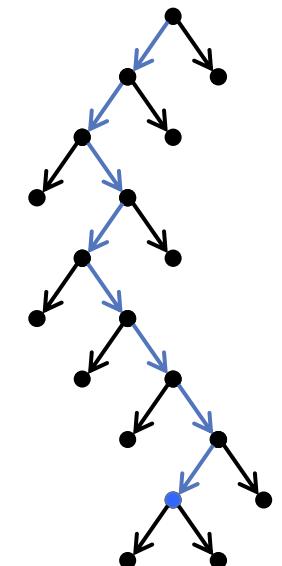
Test this
case

and this
case

and this
case

Structural Testing: Observations

- Structural testing focuses on thoroughness
 - Goal: Cover all the code
 - Not well suited for system test
 - Focuses on code rather than on requirements, for instance, does not detect missing logic
 - Requires design knowledge, which testers and clients do not have (and do not care about)
 - Thoroughness would lead to highly-redundant tests



Testing Strategies: Summary

Functional testing

- Goal: Cover all the requirements
- Black-box

Structural testing

- Goal: Cover all the code
- White-box

Random testing

- Goal: Cover corner cases
- Black-box

5. Testing

5.1 Test Stages

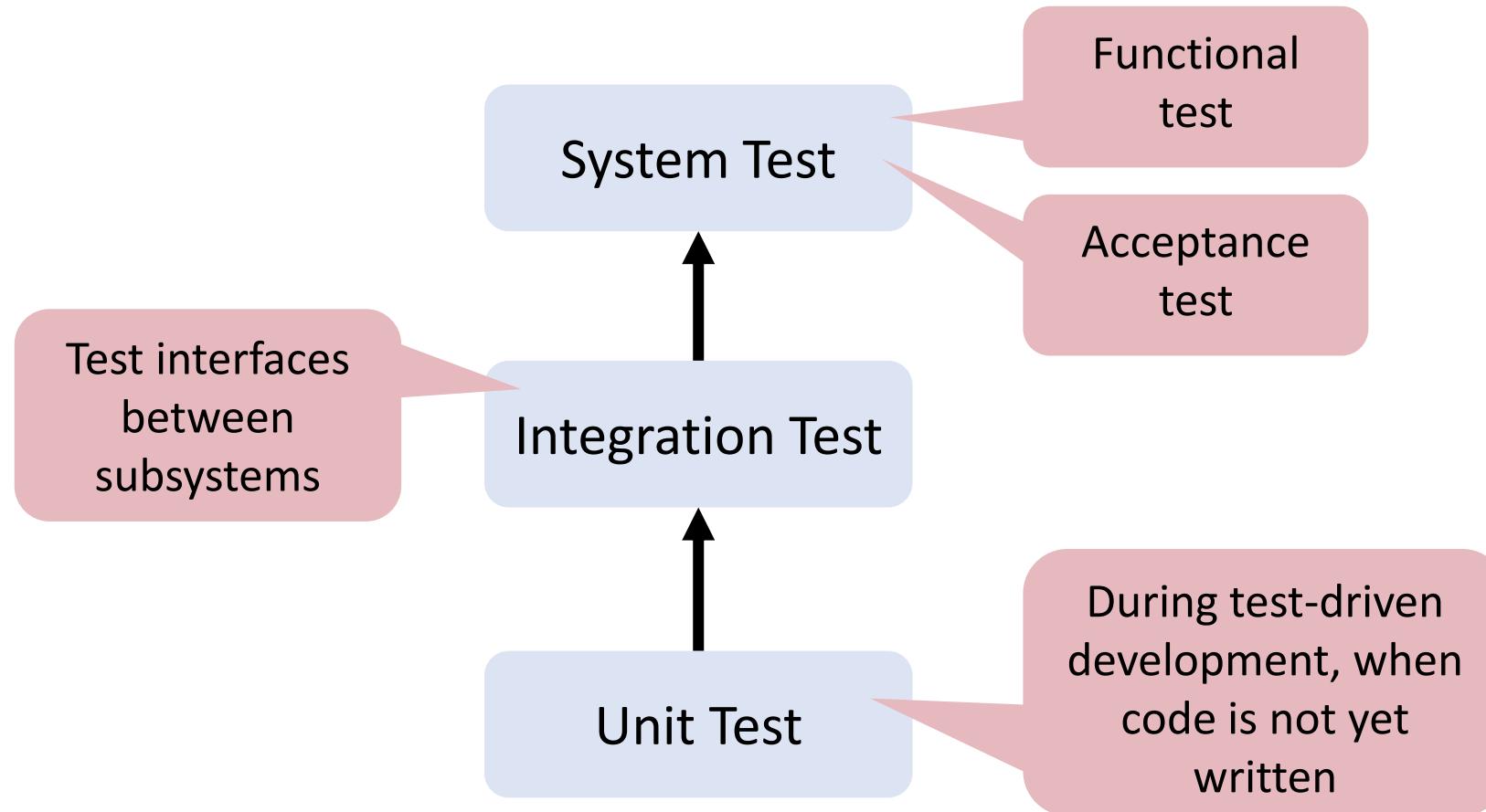
5.2 Test Strategies

5.3 Functional Testing

5.4 Structural Testing

Applications of Functional Testing

- Black-box test a unit against its requirements



5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

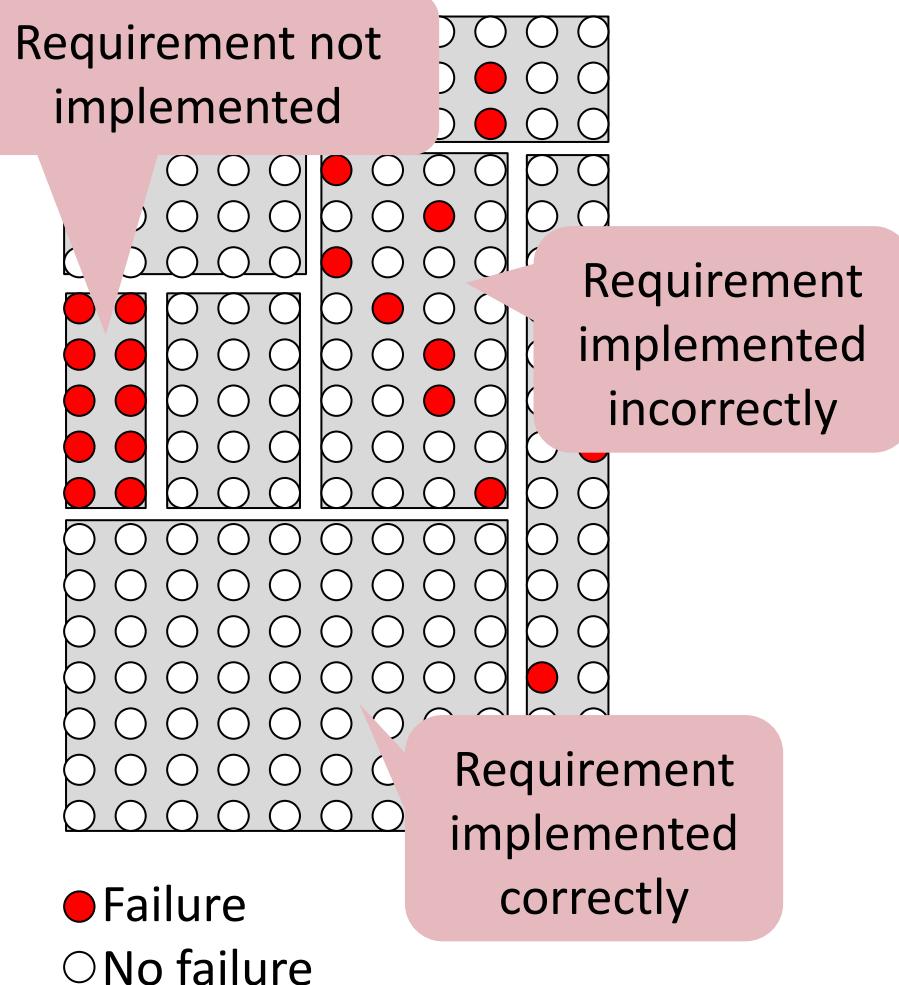
5.3.1 Partition Testing

5.3.2 Selecting Representative Values

5.3.3 Combinatorial Testing

5.4 Structural Testing

Finding Representative Inputs



- Divide inputs into equivalence classes
 - Each possible input belongs to one of the equivalence classes
 - Goal: some classes have higher density of failures
- Choose test cases for each equivalence class

Equivalence Classes: Example

Given a month (an integer in [1;12]) and a year (an integer), compute the number of days of the given month in the given year (an integer in [28;31])

month		year	
Month with 28 or 29 days	month = 2	Leap years	(year mod 4 = 0 and year mod 100 ≠ 0) or year mod 400 = 0
Months with 30 days	month ∈ {4, 6, 9, 11}	Non-leap years	year mod 4 ≠ 0 or (year mod 100 = 0 and year mod 400 ≠ 0)
Months with 31 days	month ∈ {1, 3, 5, 7, 8, 10, 12}		

Invalid inputs
missing

Equivalence Classes: Example (cont'd)

Given a month (an integer in [1;12]) and a year (an integer), compute the number of days of the given month in the given year (an integer in [28;31])

month		year	
Month with 28 or 29 days	month = 2	Leap years	(year mod 4 = 0 and year mod 100 ≠ 0) or year mod 400 = 0
Months with 30 days	month ∈ {4, 6, 9, 11}	Non-leap years	year mod 4 ≠ 0 or (year mod 100 = 0 and year mod 400 ≠ 0)
Months with 31 days	month ∈ {1, 3, 5, 7, 8, 10, 12}		
Invalid	month < 1 or month > 12		

Partitioning seems too coarse

Equivalence Classes: Example (cont'd)

Given a month (an integer in [1;12]) and a year (an integer), compute the number of days of the given month in the given year (an integer in [28;31])

month		year	
Month with 28 or 29 days	month = 2	Standard leap years	year mod 4 = 0 and year mod 100 ≠ 0
Months with 30 days	month ∈ {4, 6, 9, 11}	Standard non-leap years	year mod 4 ≠ 0
Months with 31 days	month ∈ {1, 3, 5, 7, 8, 10, 12}	Special leap years	year mod 400 = 0
Invalid	month < 1 or month > 12	Special non-leap years	year mod 100 = 0 and year mod 400 ≠ 0

5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.3.1 Partition Testing

5.3.2 Selecting Representative Values

5.3.3 Combinatorial Testing

5.4 Structural Testing

Selecting Representative Values

- Once we have partitioned the input values, we need to select **concrete values** for the test cases **for each equivalence class**
- Input from a range of valid values
 - Below, within, and above the range
 - Also applies to multiplicities on aggregations
- Input from a discrete set of valid values
 - Valid and invalid discrete value
 - Instances of each subclass

Boundary Testing

Given an integer x ,
determine the
absolute value of x

x	
Valid	all values

```
int abs( int x ) {  
    if( 0 <= x ) return x;  
    return -x;  
}
```

Negative result for
 $x==\text{Integer.MIN_VALUE}$

- A large number of errors tend to occur at **boundaries of the input domain**
 - Overflows
 - Comparisons ('<' instead of '<='), etc.)
 - Missing emptiness checks (e.g., collections)
 - Wrong number of iterations

Boundary Testing: Example

- Select elements at the “edge” of each equivalence class (in addition to values in the middle)
 - Ranges: lower and upper limit
 - Empty sets and collections

month	
Month with 28 or 29 days	month = 2
Months with 30 days	month ∈ {4, 6, 9, 11}
Months with 31 days	month ∈ {1, 3, 5, 7, 8, 10, 12}
Invalid	month < 1 or month > 12

There is only one value

Choose all values

Choose 1 and 12 plus one more

Choose
MIN_VALUE, 0, 13,
MAX_VALUE

Boundary Testing: Example (cont'd)

year	
Standard leap years	year mod 4 = 0 and year mod 100 ≠ 0
Standard non-leap years	year mod 4 ≠ 0
Special leap years	year mod 400 = 0
Special non-leap years	year mod 100 = 0 and year mod 400 ≠ 0

Choose for instance
-200.004, -4, 4, 2012,
400.008

Choose for instance
-200.003, -1, 1, 2011,
400.009

Choose for instance
-200.000, 0, 2000,
400.000

Choose for instance
-200.100, 1900, 400.100

Parameterized Unit Test for Leap Years

```
[ Test ]  
public void TestDemo29(  
    [ Values( -200004, -200000, -4, 0, 4, 2000, 2012, 400000, 400008 ) ]  
    int year )  
{  
    int d = Days( 2, year );  
    Assert.IsTrue( d == 29 );  
}
```

Only one value

All selected values for leap years and special leap years

Expected result

- Analogous test cases for February in non-leap year, months with 30 days, and months with 31 days

Parameterized Unit Test for Invalid Inputs

```
[ Test ]  
[ ExpectedException( typeof(ArgumentException) ) ]  
public void TestDemoInvalid(  
    [ Values( int.MinValue, 0, 13, int.MaxValue ) ] int month,  
    [ Values( -200100, -200004, -200003, -200000, -4, -1, 0, 1, 4, 1900,  
        2000, 2011, 2012, 400000, 400008, 400009, 400100 ) ] int year ) {  
    int d = Days( month, year );  
}
```

Expected result:
an exception

All selected
invalid values
for month

All selected
values for year

5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.3.1 Partition Testing

5.3.2 Selecting Representative Values

5.3.3 Combinatorial Testing

5.4 Structural Testing

Rigorous Software Engineering

Fuzzing and Symbolic Execution for Smart Contracts

Jingxuan He

Spring 2021

 SRILAB

<https://www.sri.inf.ethz.ch/>

 ETH zürich

Roadmap

Previous Lectures: testing, symbolic execution, and concolic execution for standard programs (e.g., C)

Plan for Today:

- I. Fuzzing for Smart Contracts
- II. Symbolic Execution for Smart Contracts
- III. Learning to Fuzz from Symbolic Execution

Testing v.s. Fuzzing

Testing

- Run programs on **well-structured** and **normal** inputs (e.g., unit tests)
- Prevent **normal users** from encountering functional errors (e.g., assertion failures)

Fuzzing

- Run programs on **abnormal** inputs (e.g., randomly generated)
- Prevent **attackers** from making exploits (e.g., buffer overflow)

```
int sum(int a, int b) {  
    int c = a + b;  
    return c;  
}
```

Given $a = 1, b = 1$, check if $c = 2$

$a = 2^{31}-100, b = 1000$ leads to overflow

Ethereum

A public decentralized blockchain platform designed to run smart contracts

- Similar to a computer that allows users to **deploy programs** (smart contracts), **maintains the state** of all deployed smart contracts, and **executes them** upon request (transactions)
- The latest block stores the **latest local states** of all smart contracts
- Transactions result in **executing code** (calling a function) in target smart contracts
- Transaction **change the state** of one or more contracts
- Ethereum smart contracts are **Turing-complete**

Ethereum transactions

From	<address>	Address of the transaction sender
To	<address>	Address of the target contract
Data	<method> <arg>, ..., <arg>	ID of the method to invoke, along with arguments
Value	<amount>	Amount of Ether sent to the target contract
Gas/gas price	<amount>/<amount>	Prevent infinite loops

Example smart contract

```
pragma solidity 0.5.8;

contract SimpleBank {

    mapping(address => uint) balances;

    function deposit(uint amount) payable public {
        balances[msg.sender] += amount;
    }

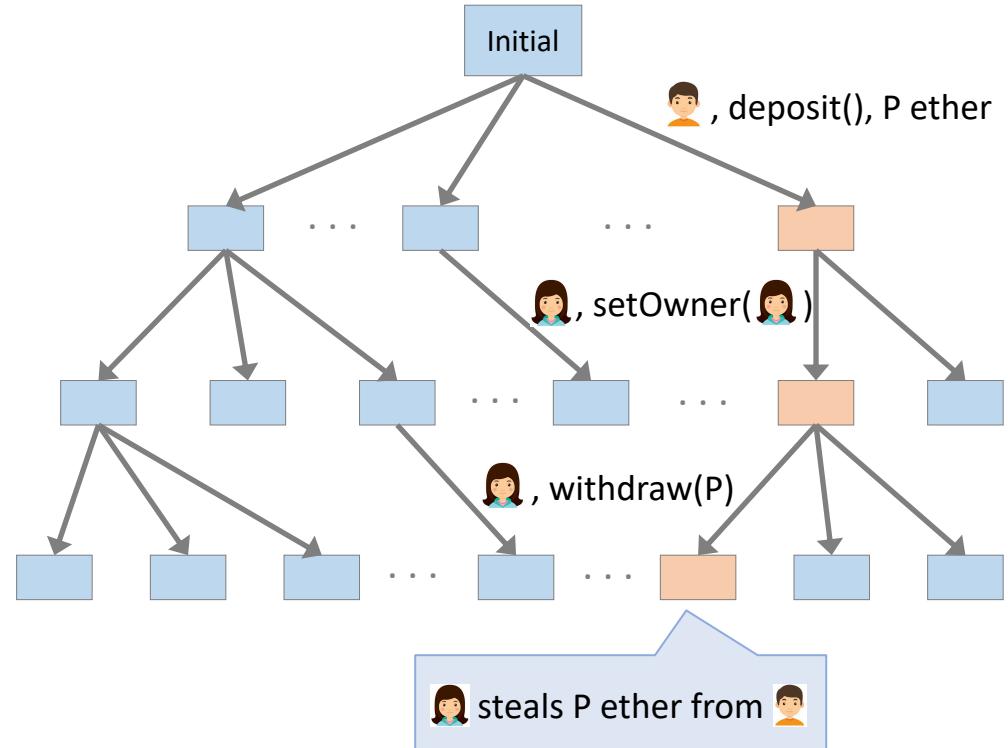
    function withdraw() public {
        msg.sender.transfer(balances[msg.sender]);
        balances[msg.sender] = 0;
    }
}
```

Smart Contract Fuzzing: Challenge

Example Contract

```
1 contract Wallet {
2     address owner;
3
4     constructor() {
5         owner = msg.sender;
6     }
7
8     function setOwner(address newOwner) {
9         // fix: require(msg.sender == owner);
10        owner = newOwner;
11    }
12
13    function deposit() payable {}
14
15    function withdraw(uint amount) {
16        require(msg.sender == owner);
17        owner.transfer(amount);
18    }
19 }
```

Exponential Number of Block States

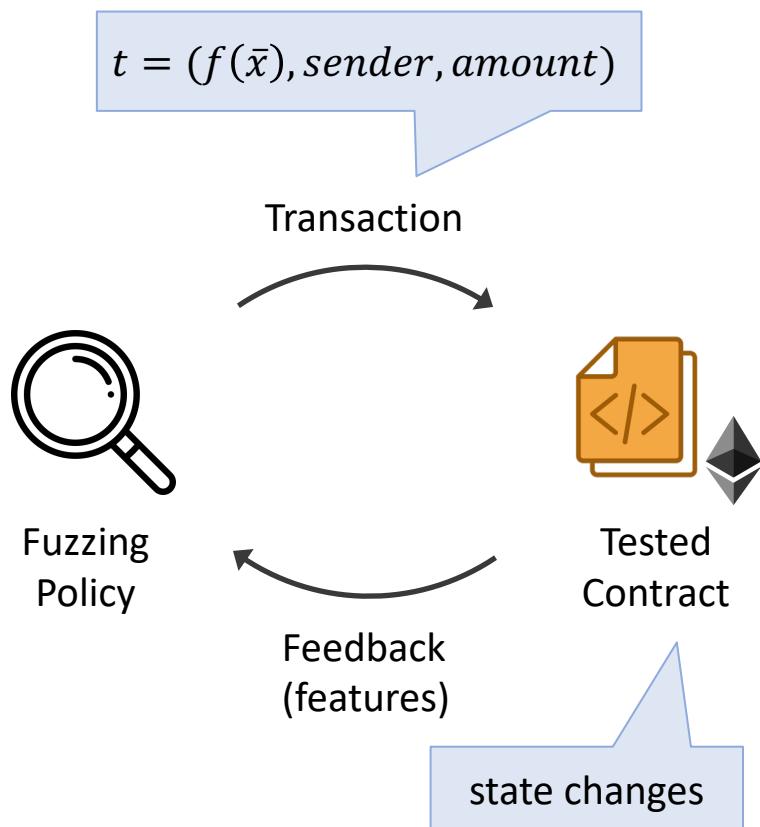


Smart Contract Fuzzing: Challenge

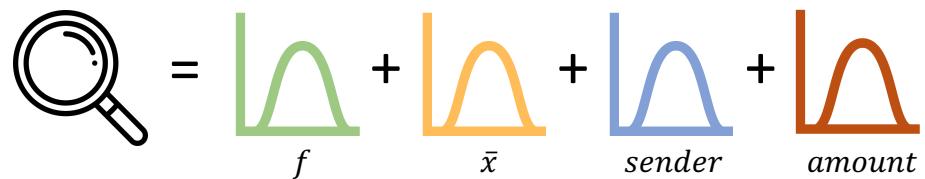
Wanted: Transaction Sequences that thoroughly explore the state space

Smart Contract Fuzzing Policy

Fuzzing Process



Policy Components



Example: a Uniformly Random Policy

$$\text{Uniform}(F) : \text{Uniform}(f)$$

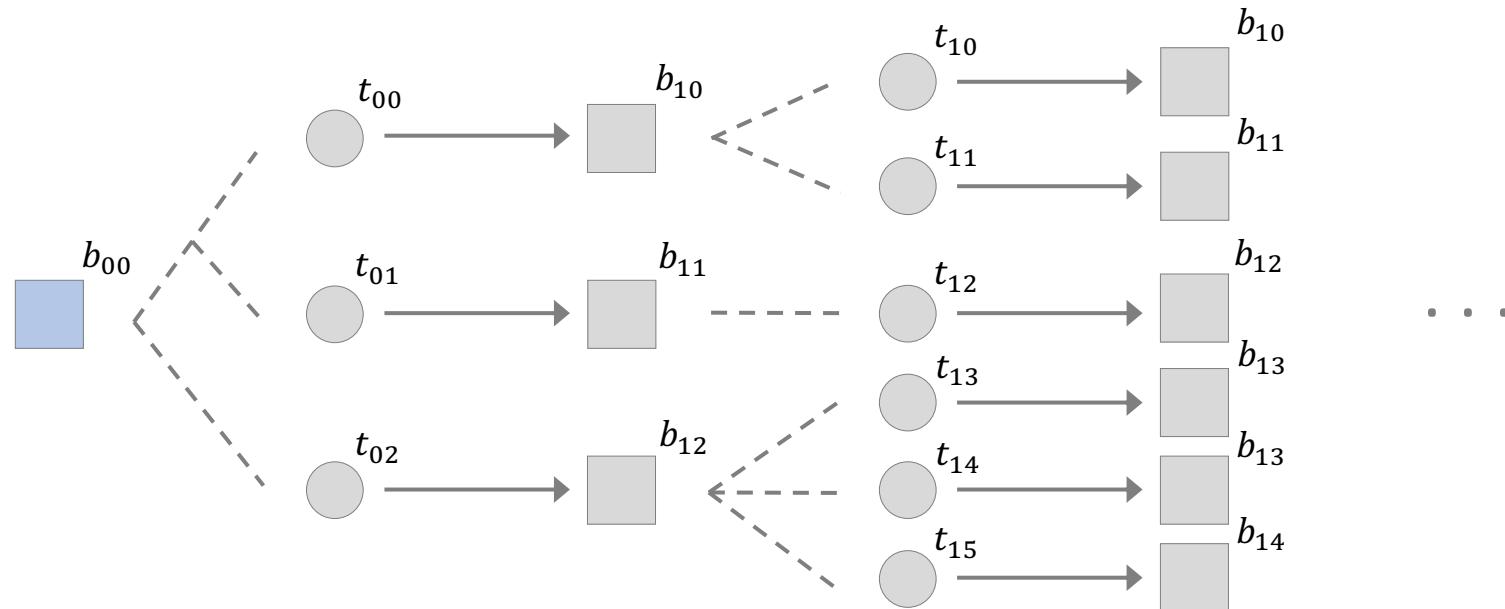
$$\text{Uniform}(\text{Signature}(f))$$

$$\text{Uniform}(\text{SENDERS})$$

$$\begin{cases} \text{Uniform}([0, MA]) & f \text{ is payable} \\ P(0) = 1 & \text{otherwise} \end{cases}$$

SE for Smart Contracts

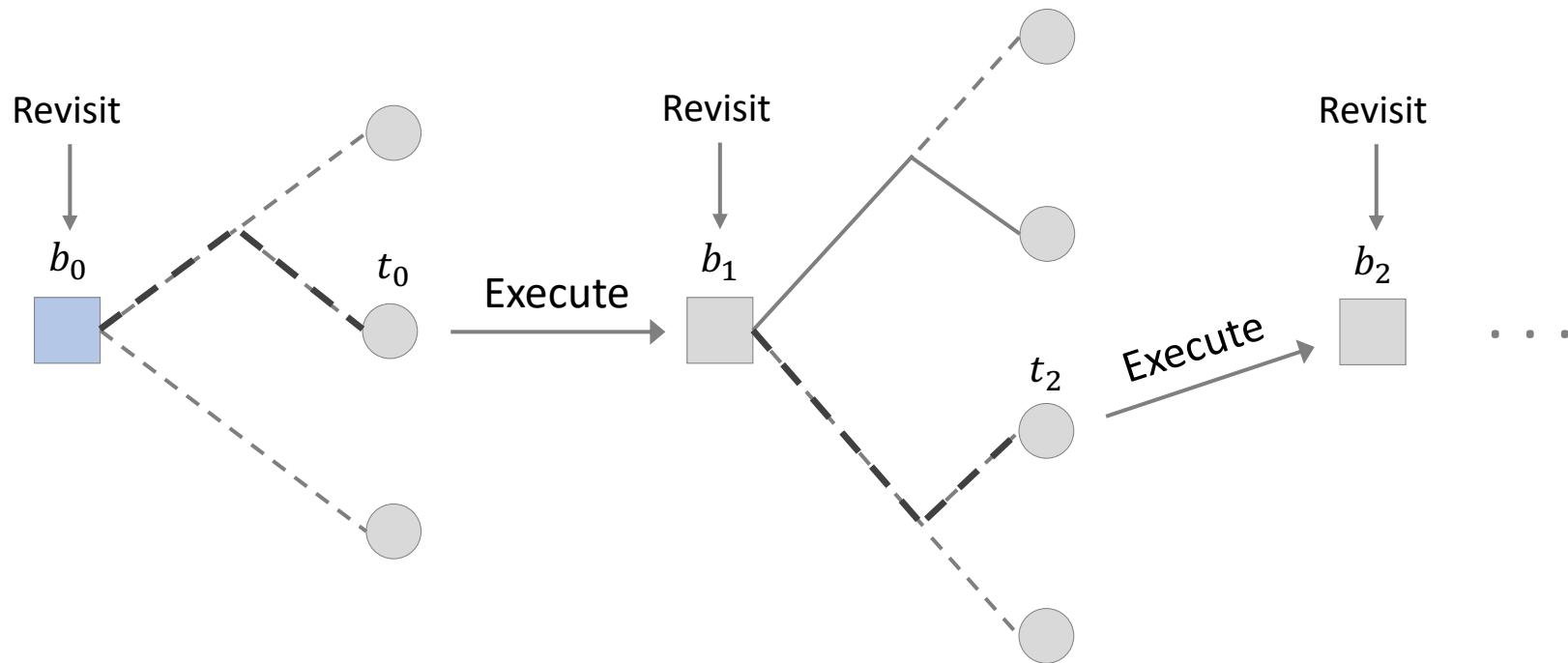
Breadth-first search:



- Covers all cases given a small depth
- Exponential number of symbolic states within a transaction
- Exponential number of symbolic transactions and block states
- Complex constraints

SE for Smart Contracts

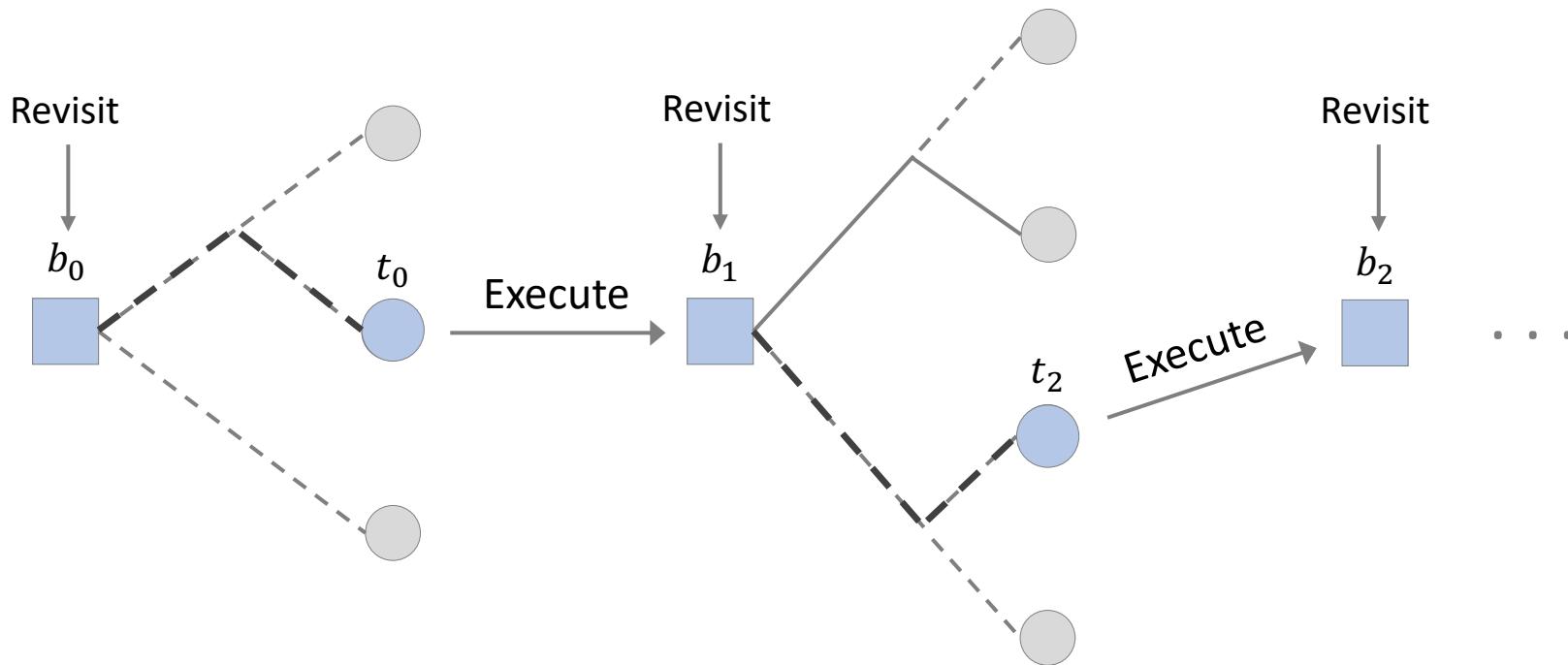
Depth-first search with revisit:



- Can handle larger depth
- Complex constraints
- Exponential number of symbolic states within a transaction

SE for Smart Contracts

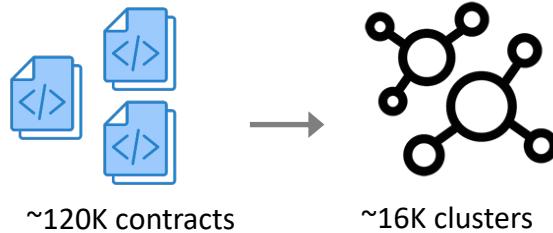
Depth-first search with revisit and concretization:



- Most practical
- Exponential number of symbolic states within a transaction

Random Fuzzing v.s. Symbolic Execution

Similarity between Contracts:

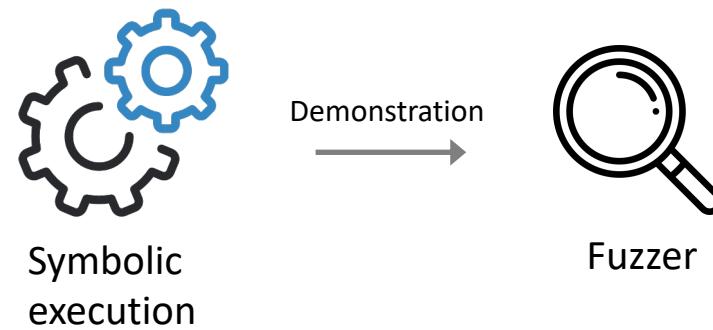
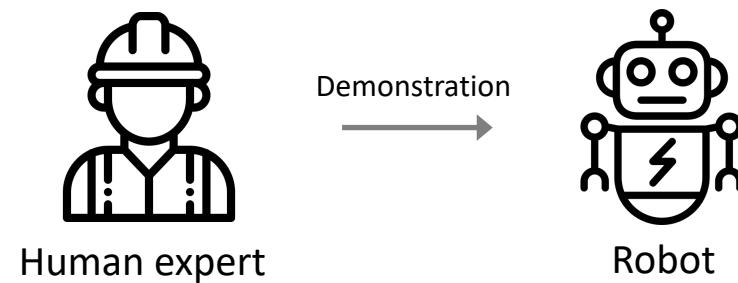


Imitation Learning based Fuzzer

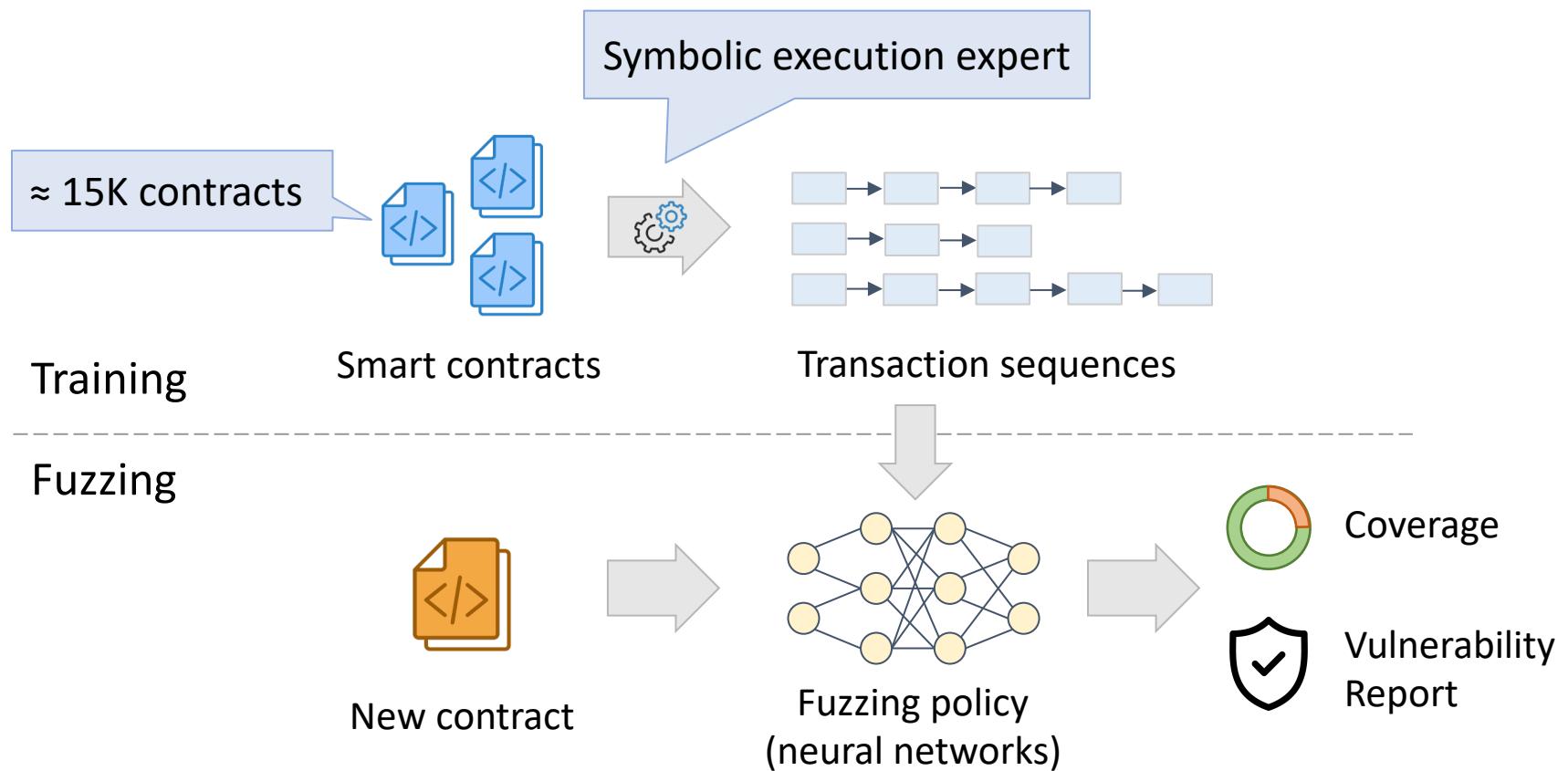
- Analyzing Ethereum's Contract Topology. Kiffer et al.. IMC '18

	Random Fuzzing	Symbolic Execution	ILF
Speed	✓ Fast	✗ Slow	✓ Fast
Inputs	✗ Ineffective	✓ Effective	✓ Effective
Coverage	✗ Low	✗ Low	✓ High

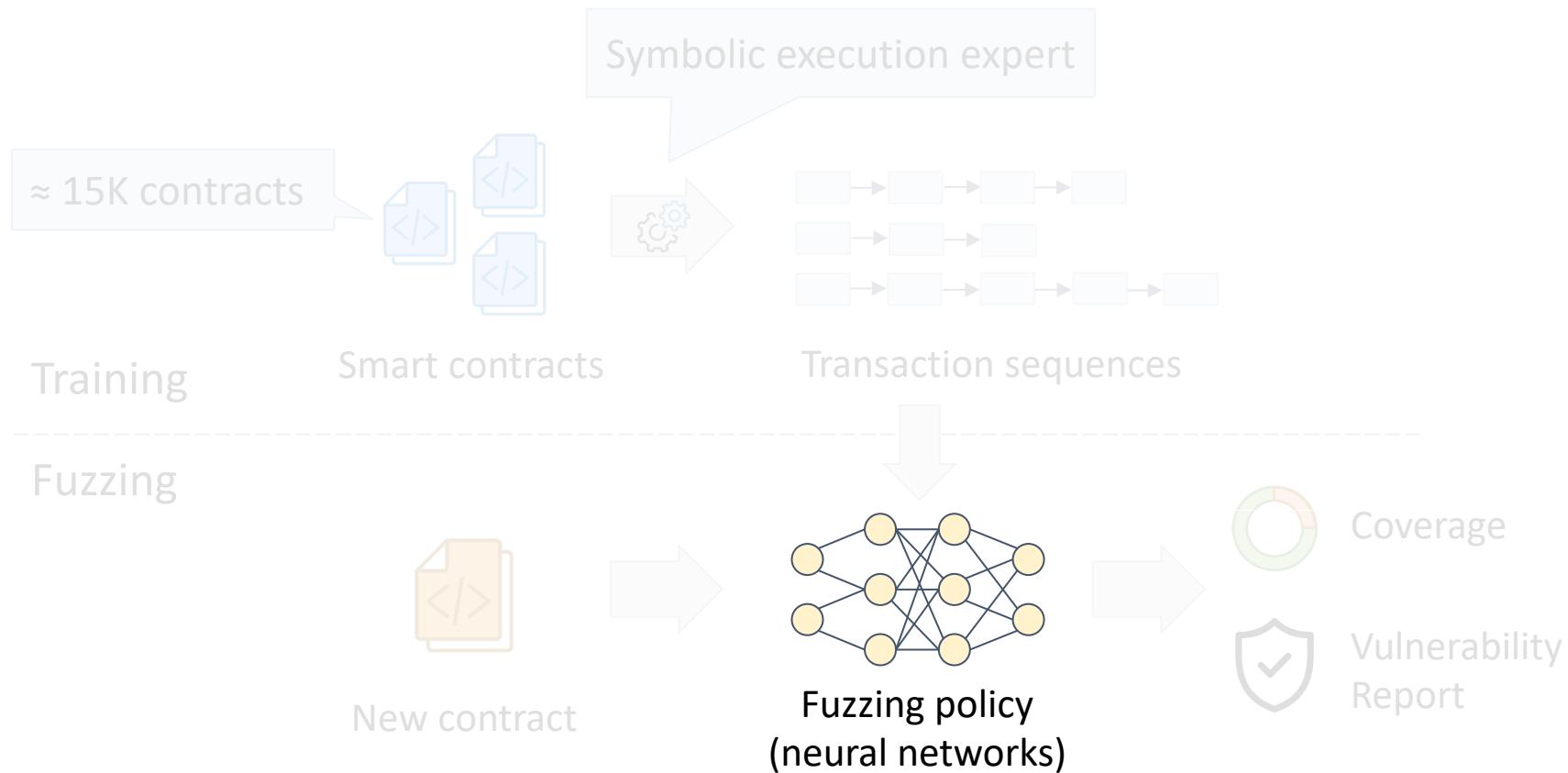
Imitation Learning



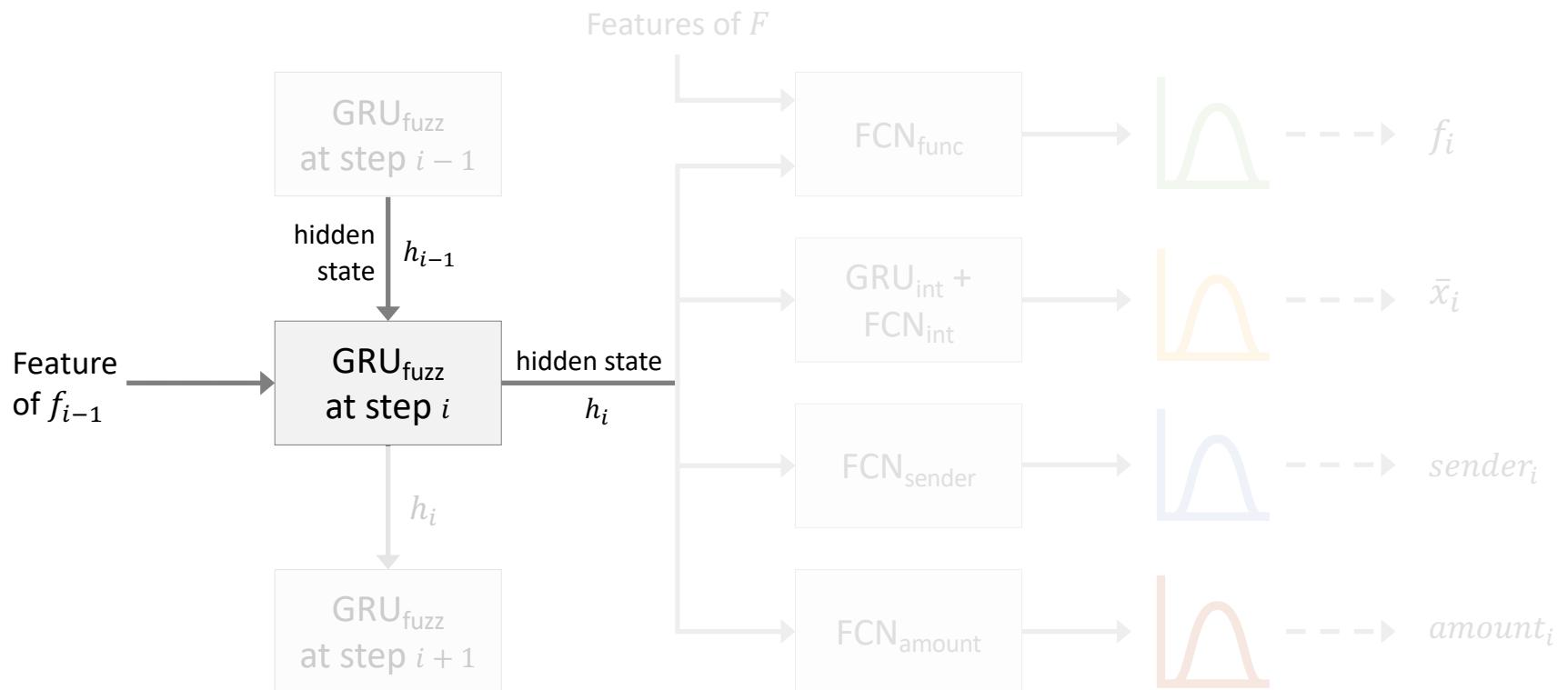
Learning to Fuzz from Symbolic Execution



Learning to Fuzz from Symbolic Execution



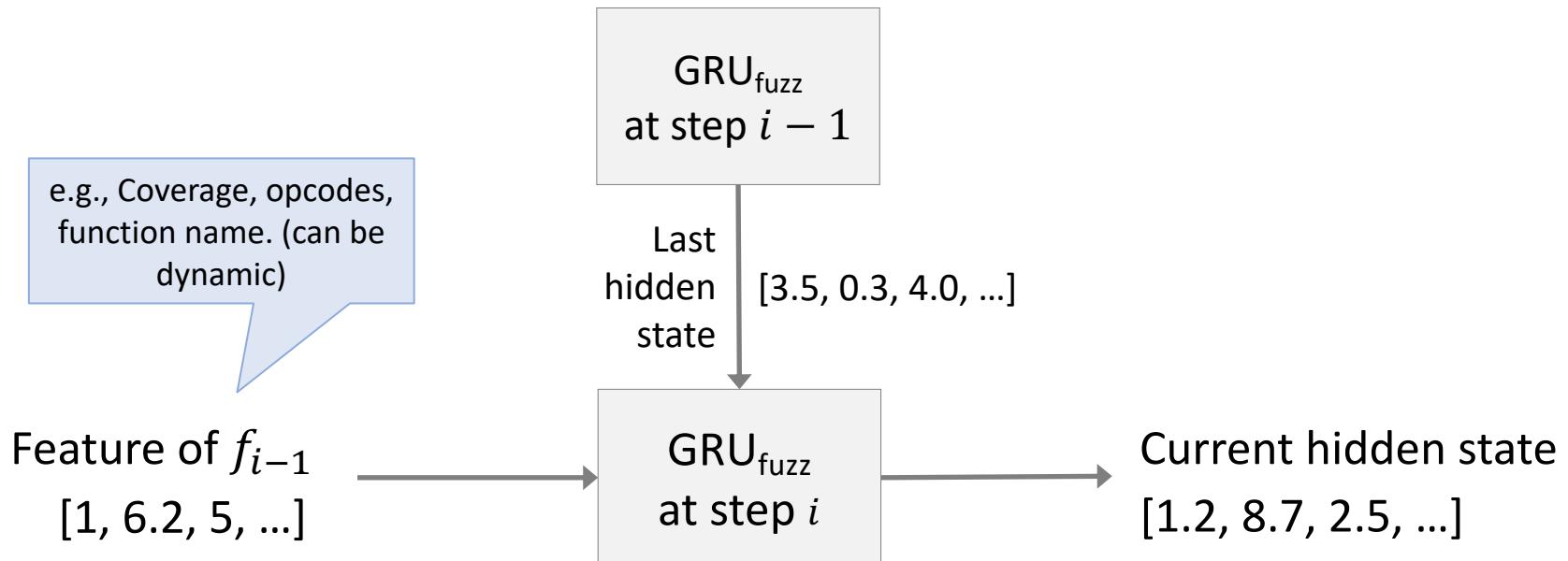
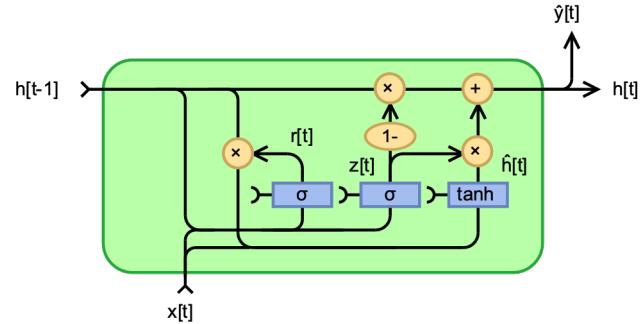
Neural Network Fuzzing Policy



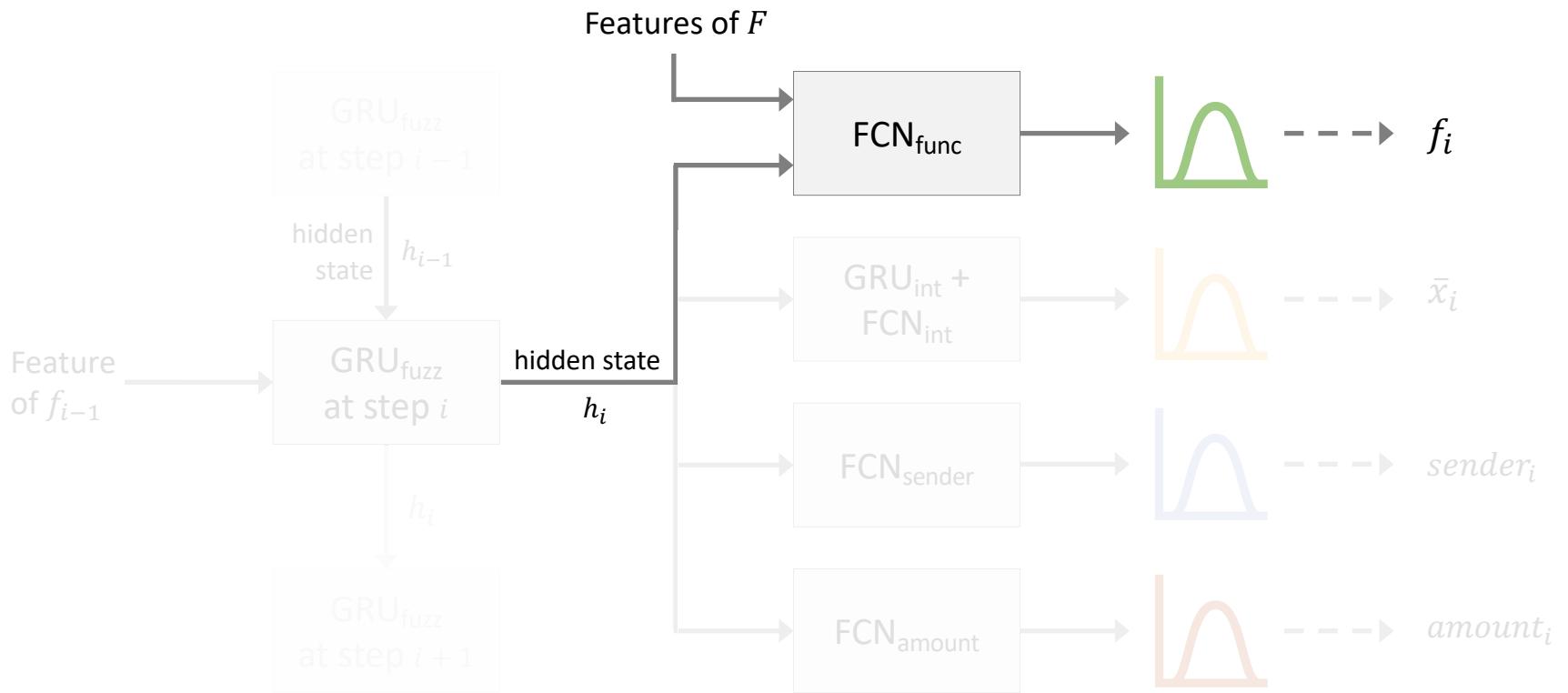
NN – Fuzzing State

Gated Recurrent Units (GRU):

- Can deal with variable length **sequential** inputs with hidden states
- A natural fit for our setting for handling **sequences of transactions**



Neural Network Fuzzing Policy



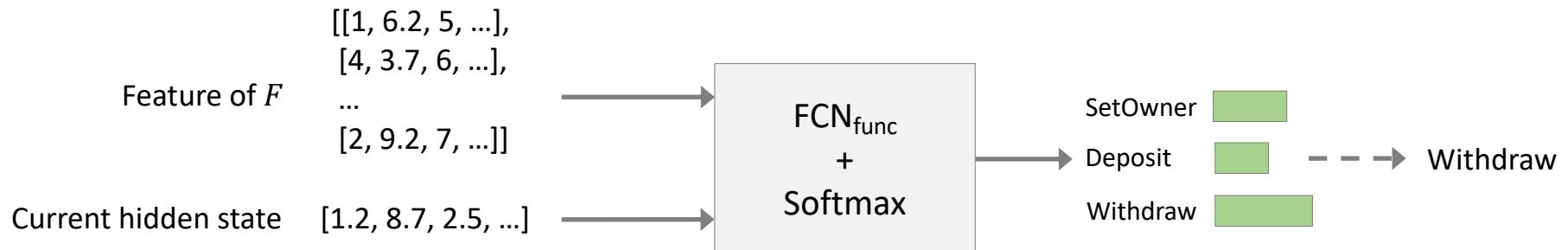
NN - Function

Fully Connected Networks (FCN):

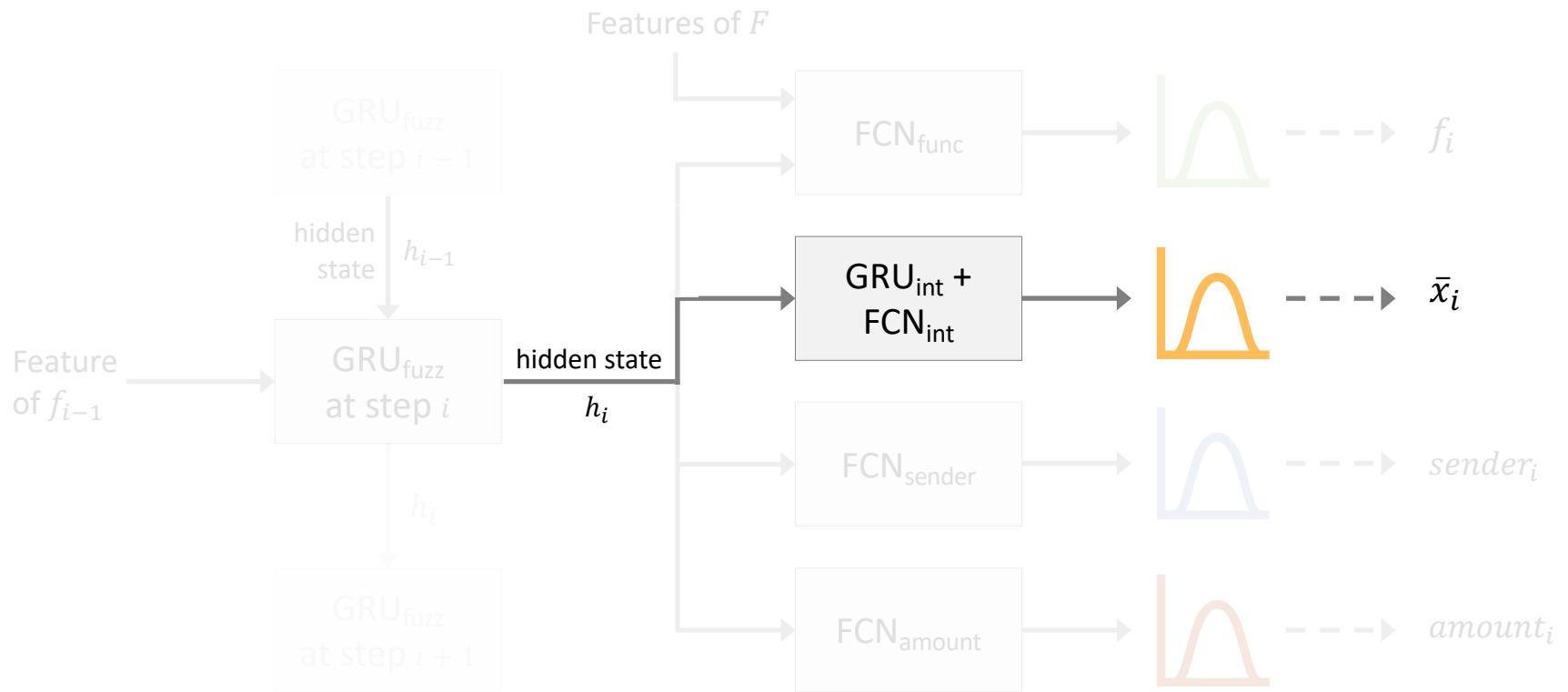
- Standard network with linear layers and ReLU activation functions

Softmax:

- Normalizes function scores (output of FCN_{func}) to a **probability distribution**



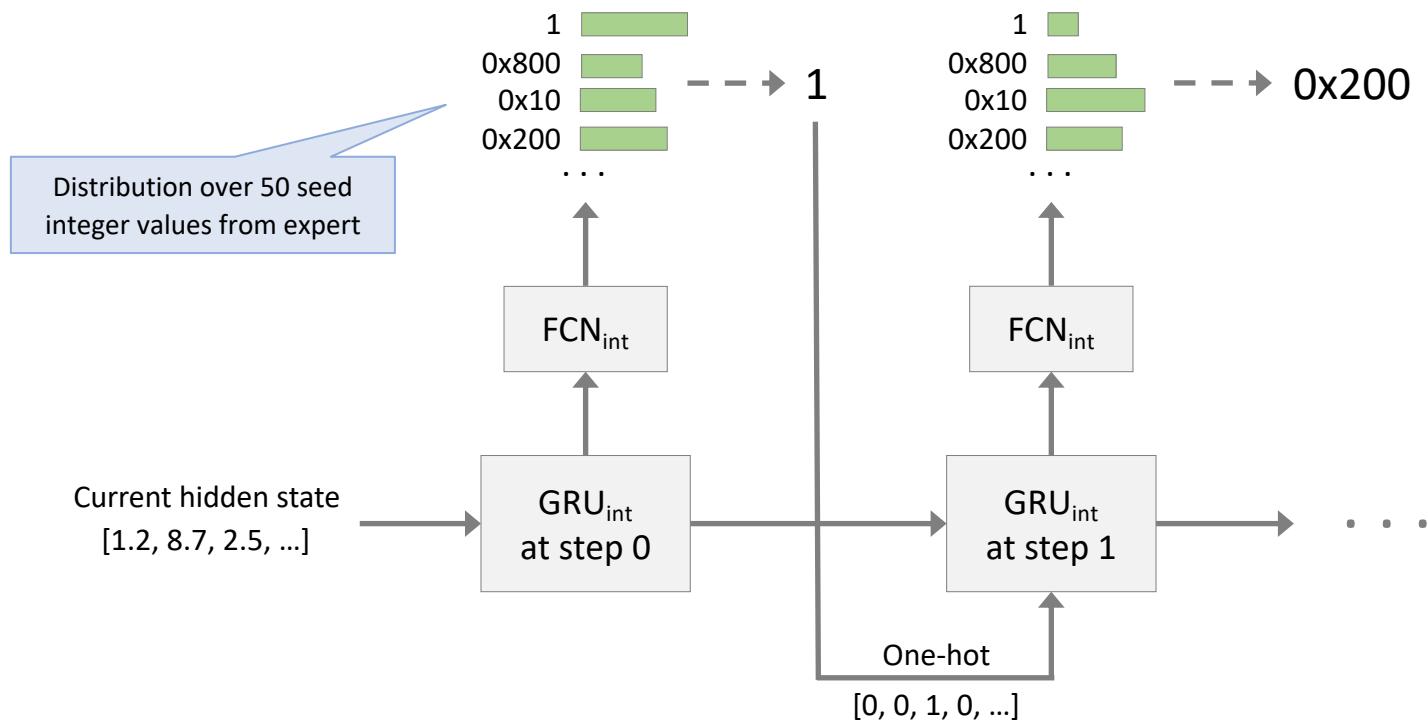
Neural Network Fuzzing Policy



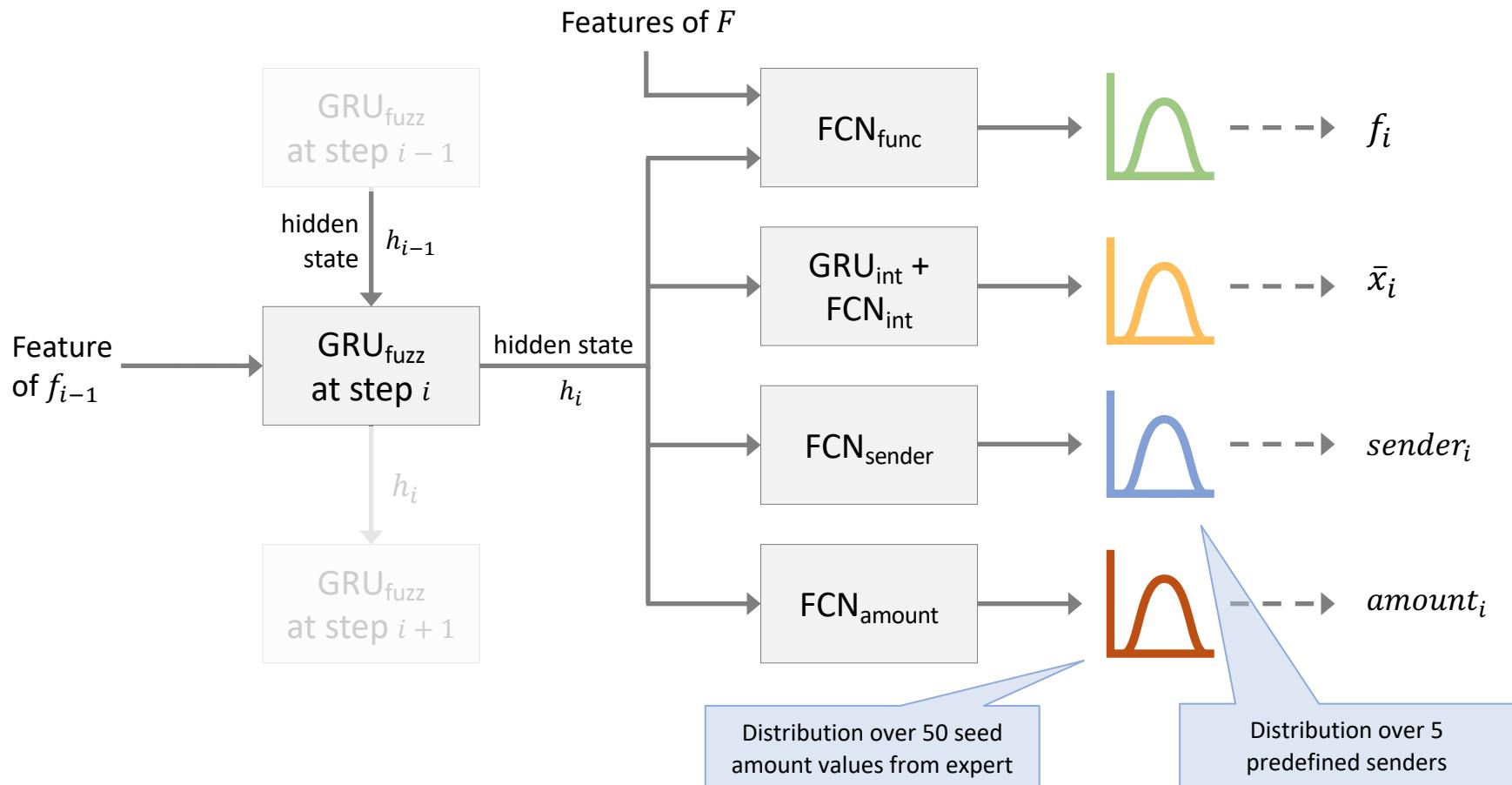
NN - Arguments

Challenges:

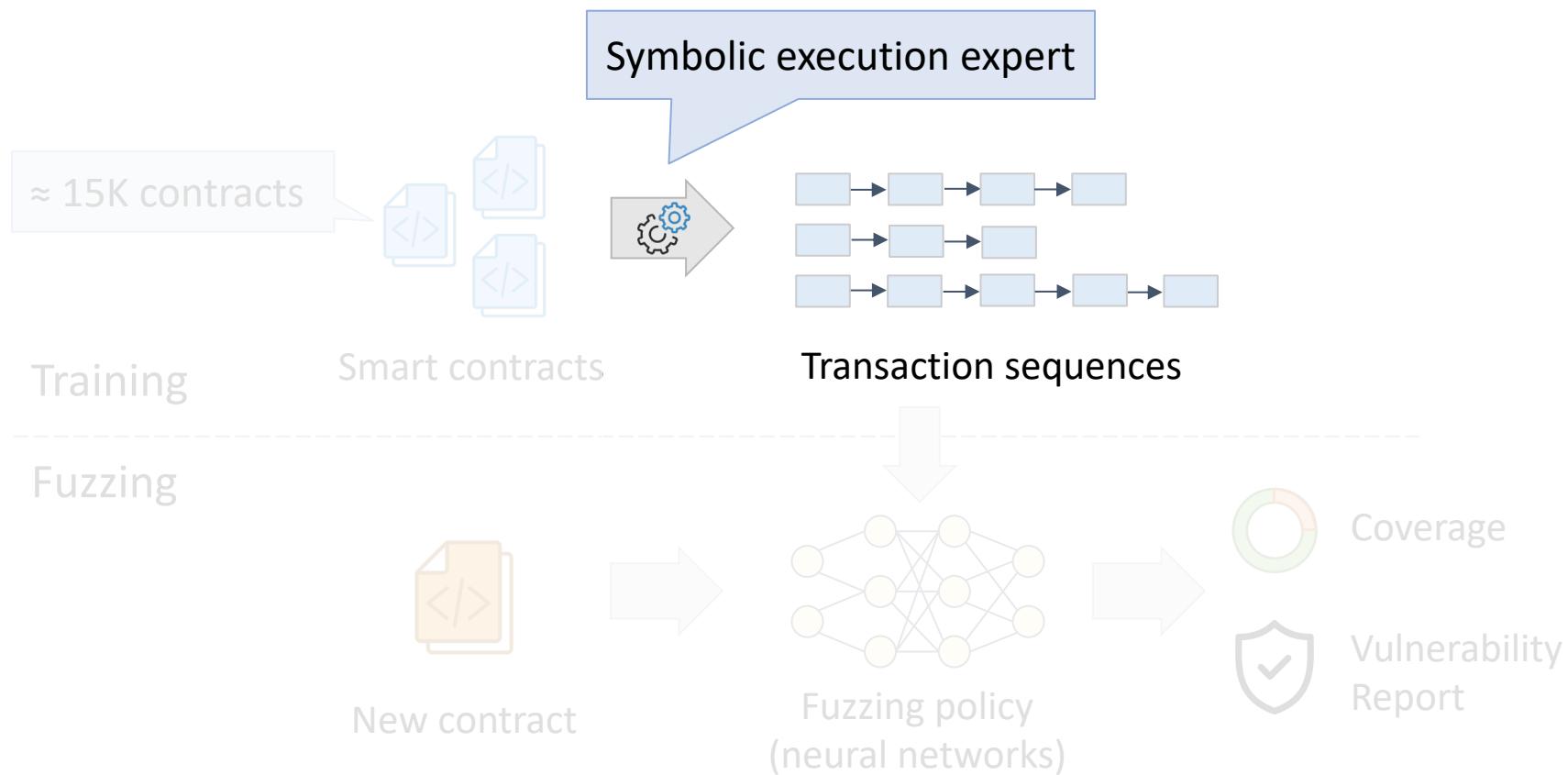
- Functions have a **variable** length of arguments
- Search space of arguments is **large** (e.g., integers of 256 bits)



Neural Network Fuzzing Policy

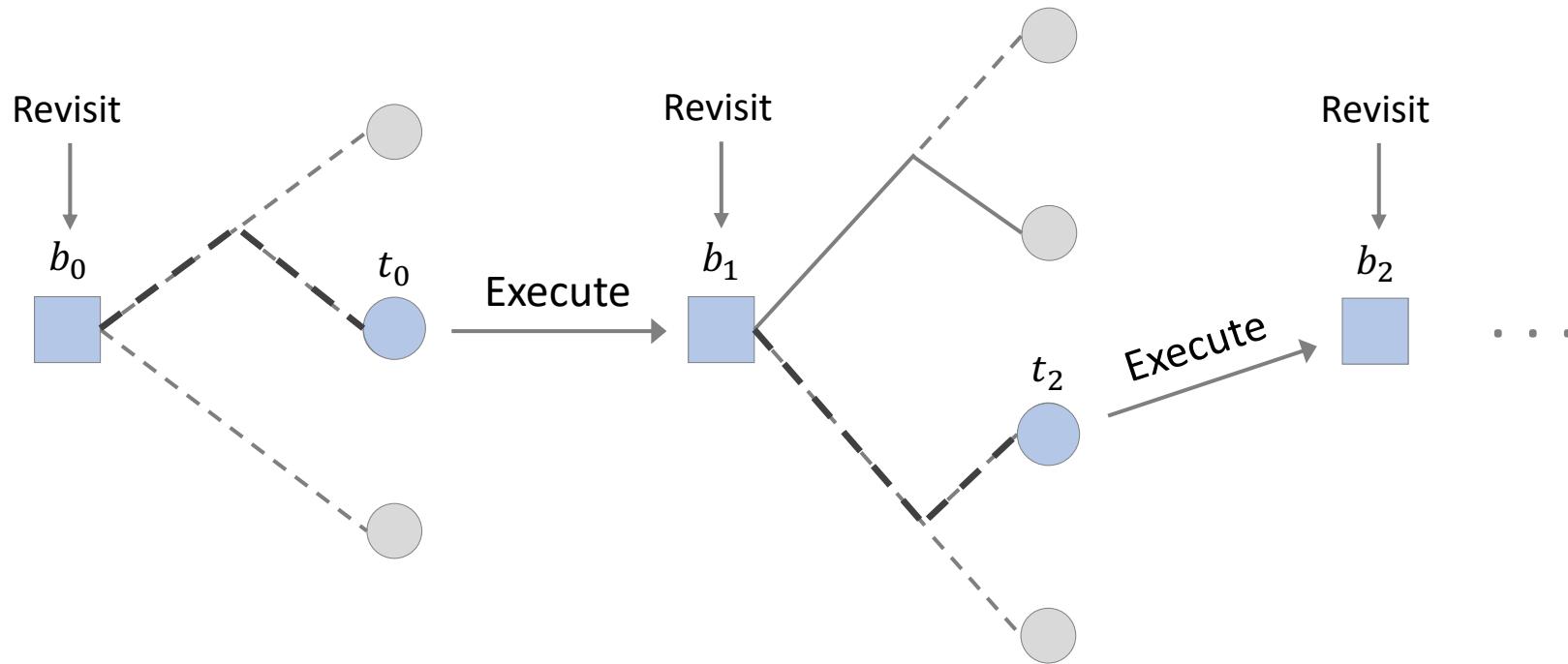


Learning to Fuzz from Symbolic Execution

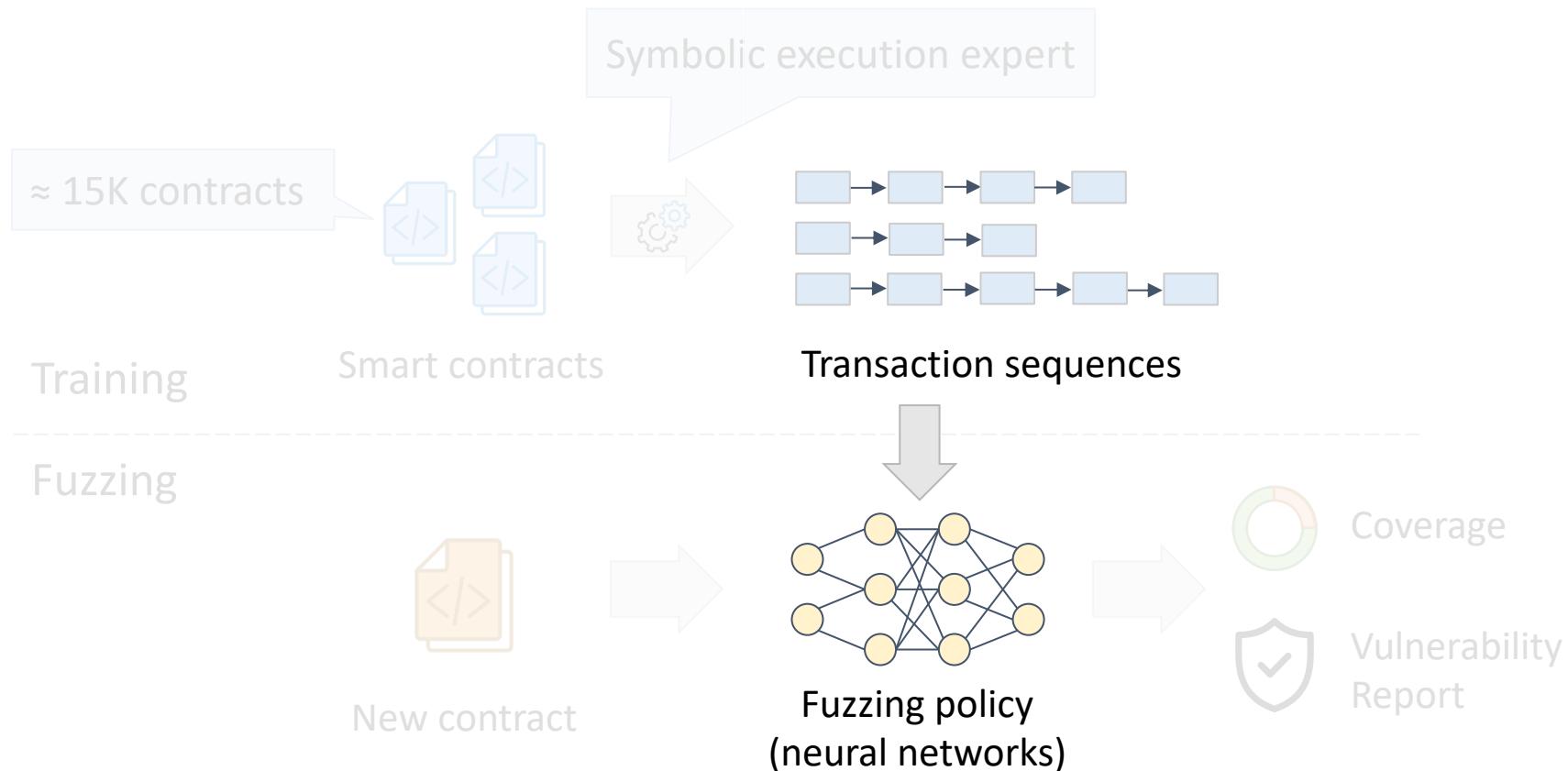


SE for Smart Contracts

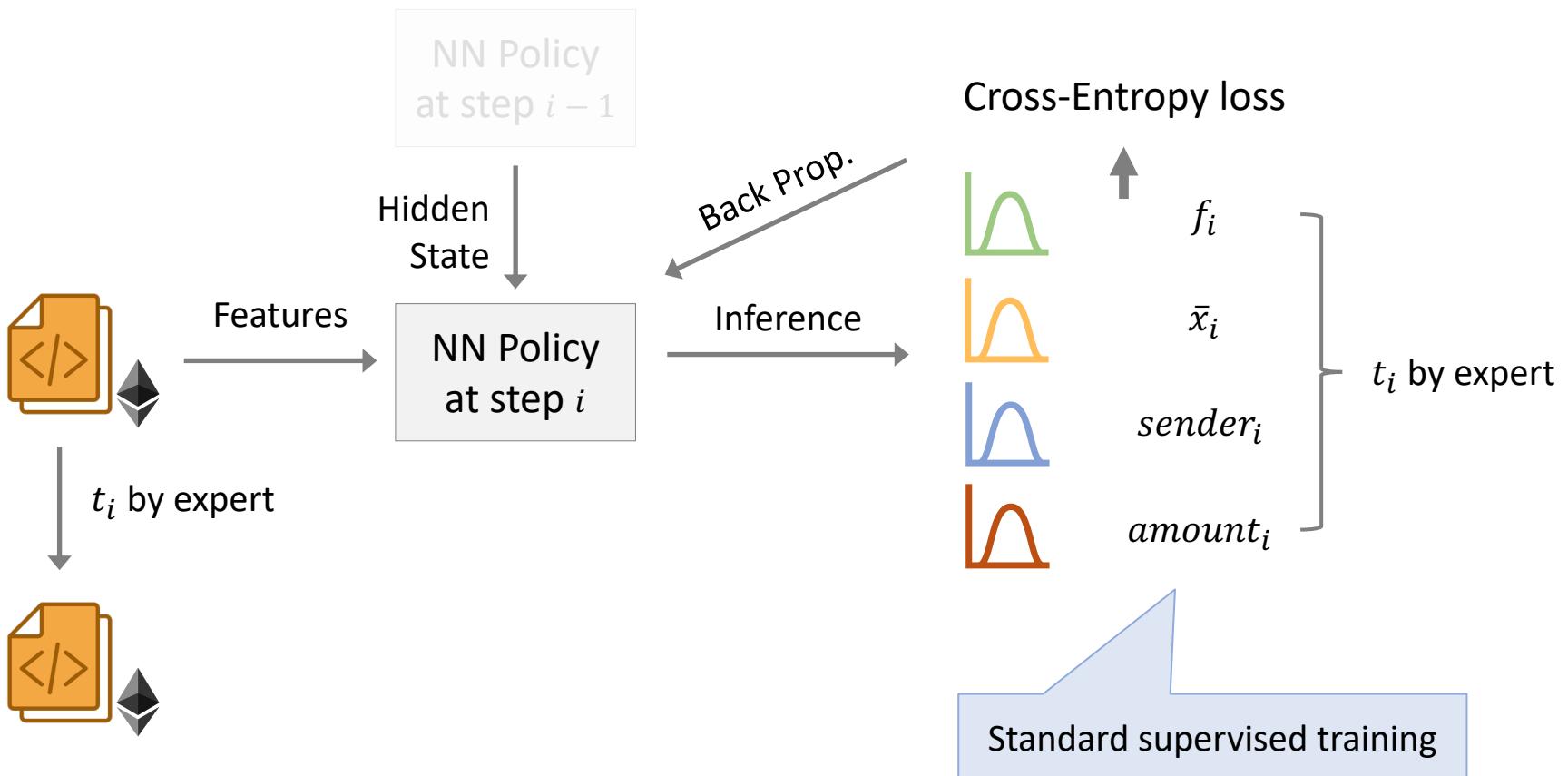
Depth-first search with revisit and concretization:



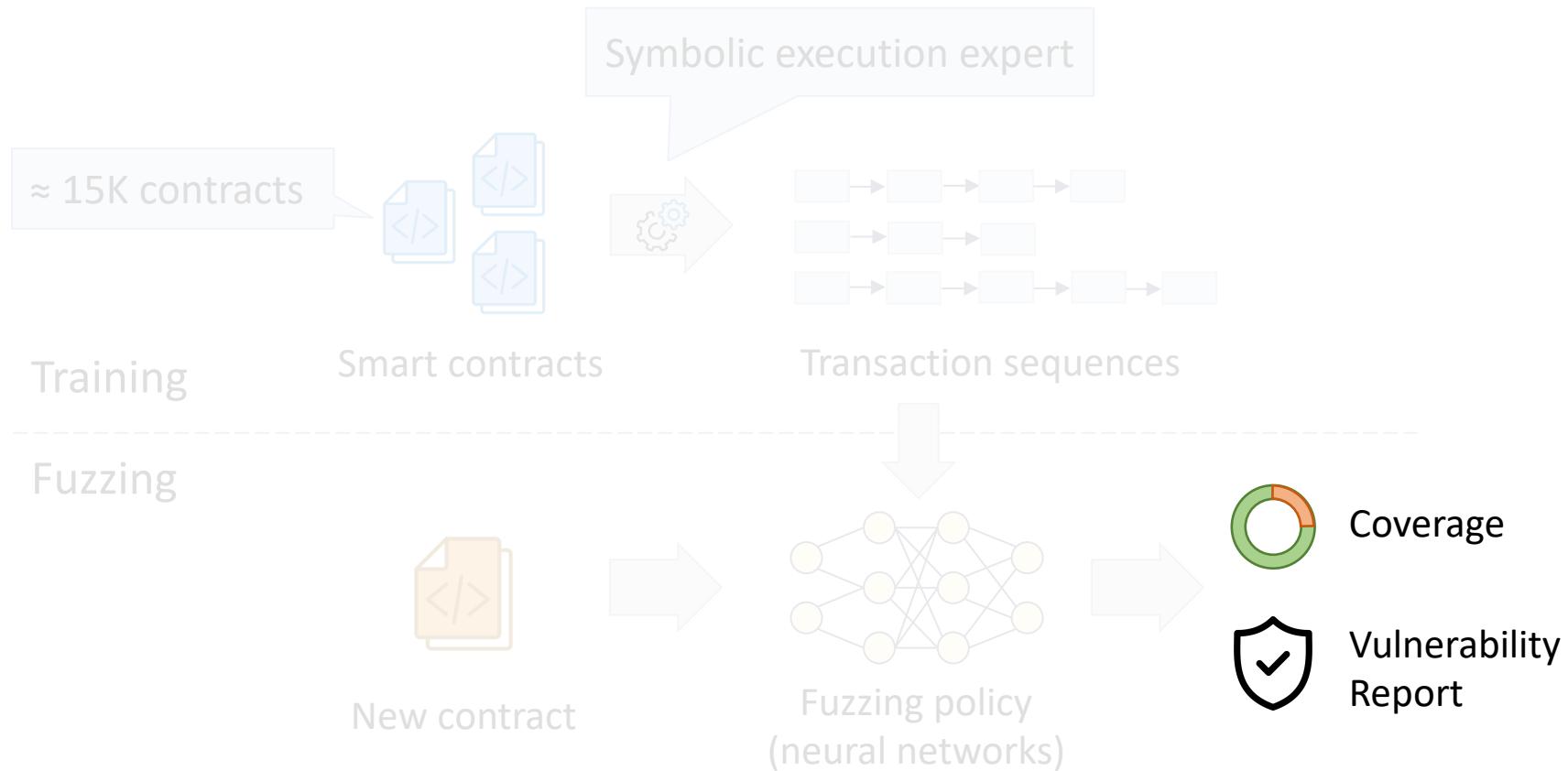
Learning to Fuzz from Symbolic Execution



Training NN Fuzzing Policy



Learning to Fuzz from Symbolic Execution



ILF System

<https://github.com/eth-sri/ilf>



- **Instruction** coverage.
- **Basic block** coverage.



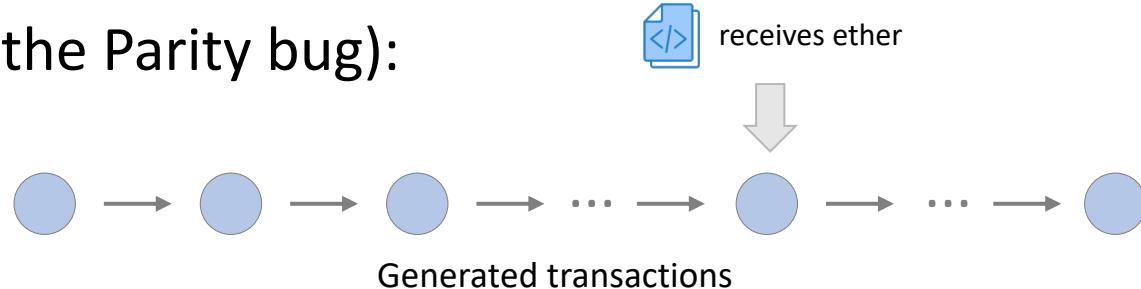
- **Locking:** The contract cannot send out but can receive ether.
- **Leaking:** An attacker can steal ether from the contract.
- **Suicidal:** An attacker can deconstruct the contract.
- **Block Dependency:** Ether transfer depends on block state variables.
- **Unhandled Exception:** Root call does not catch exceptions from child calls.
- **Controlled Delegatecall:** Transaction parameters explicitly flow into arguments of a *delegatecall* instruction.

Vulnerability Checking

- Locking (e.g., the Parity bug):

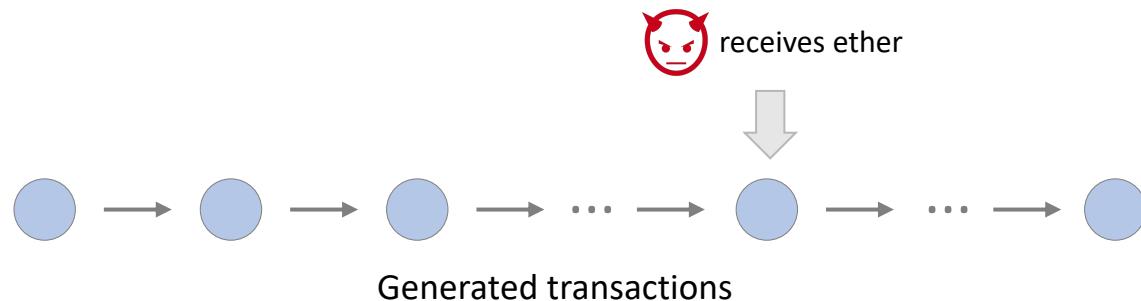
Statically:

 cannot send out ether

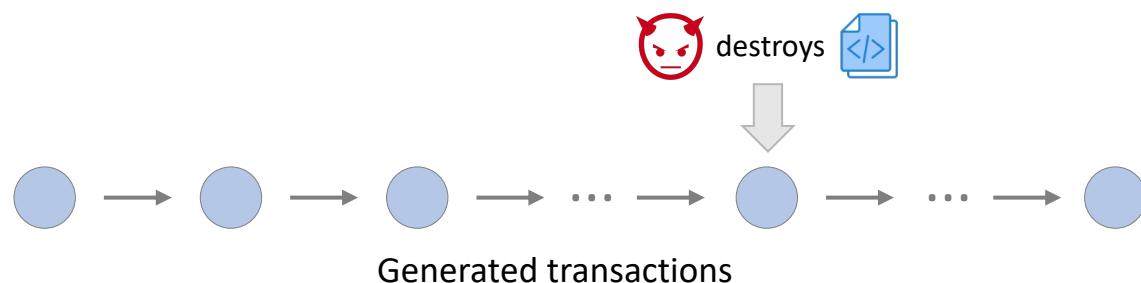


- Leaking:

$\text{balance}[\text{red devil icon}] = 0$



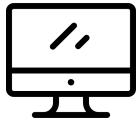
- Suicidal:



Evaluation



- 18,496 Contracts (5,013 Large & 13,483 Small)
- 5-fold Cross Validation

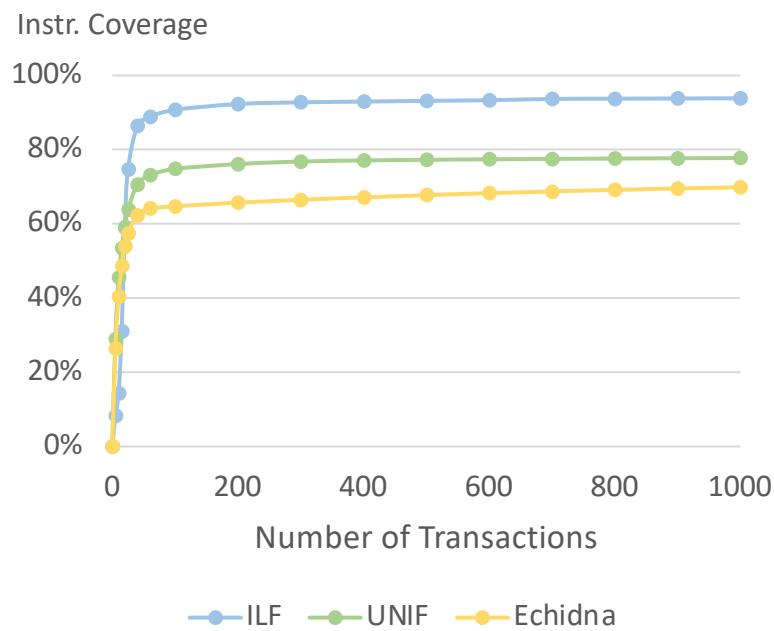


- UNIF
- EXPERT
- Echidna
- MAIAN
- ContractFuzzer

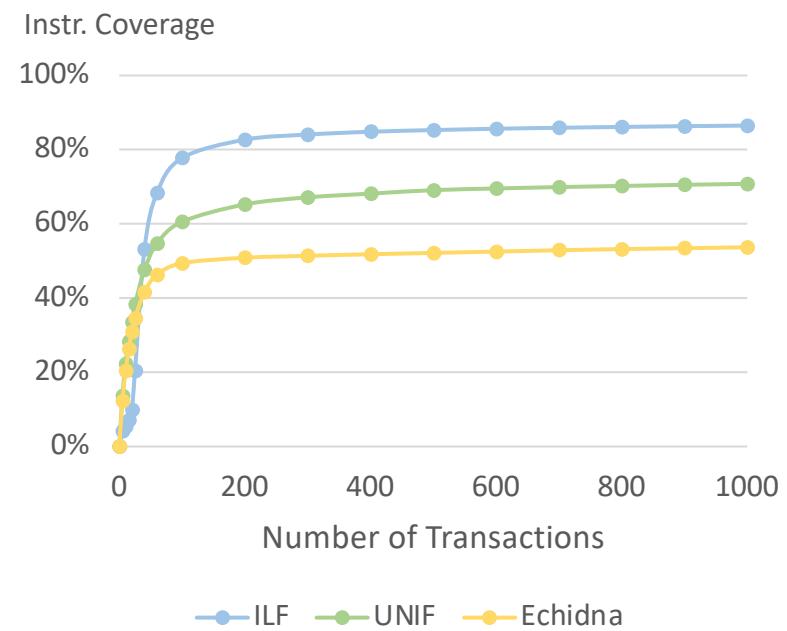


- Coverage & Speed
- Fuzzing Components
- Vulnerability Detection
- Case Study

Coverage: ILF vs. Fuzzers

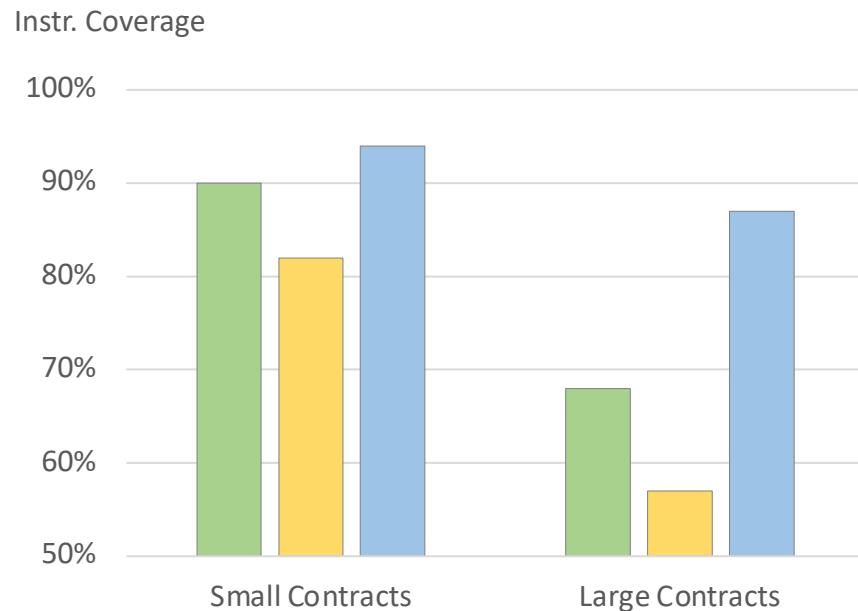


Small contracts



Large contracts

Coverage: ILF vs. Symbolic Expert

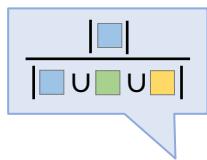


Small: 30 txs, 547s
Large: 49 txs, 2,580s

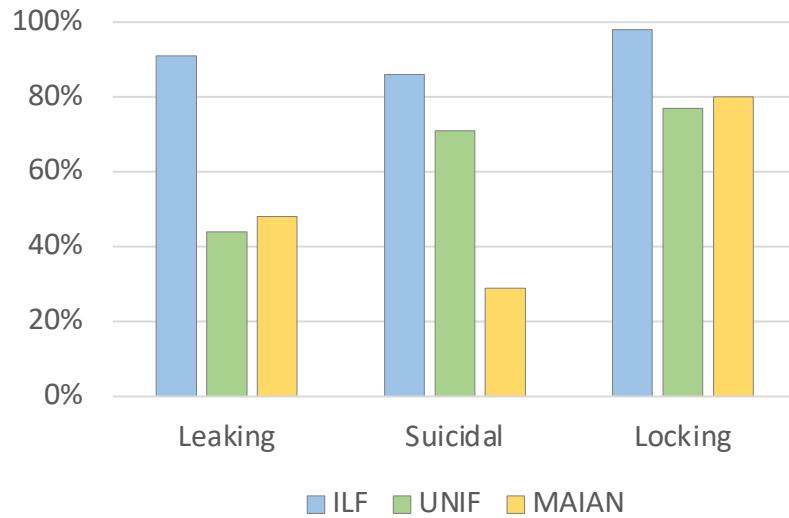
Small: 13s
Large: 17s
148 txs/s

- EXPERT
- ILF (#tx same as EXPERT)
- ILF (2k txs)

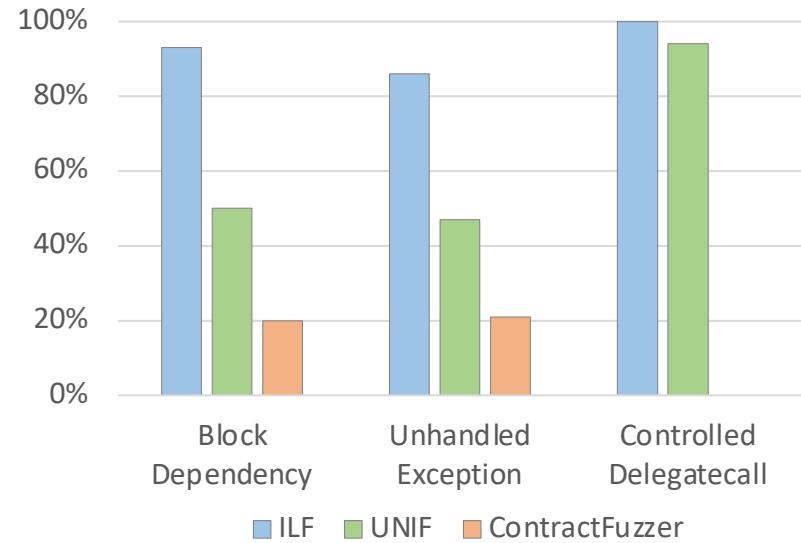
Vulnerability Detection



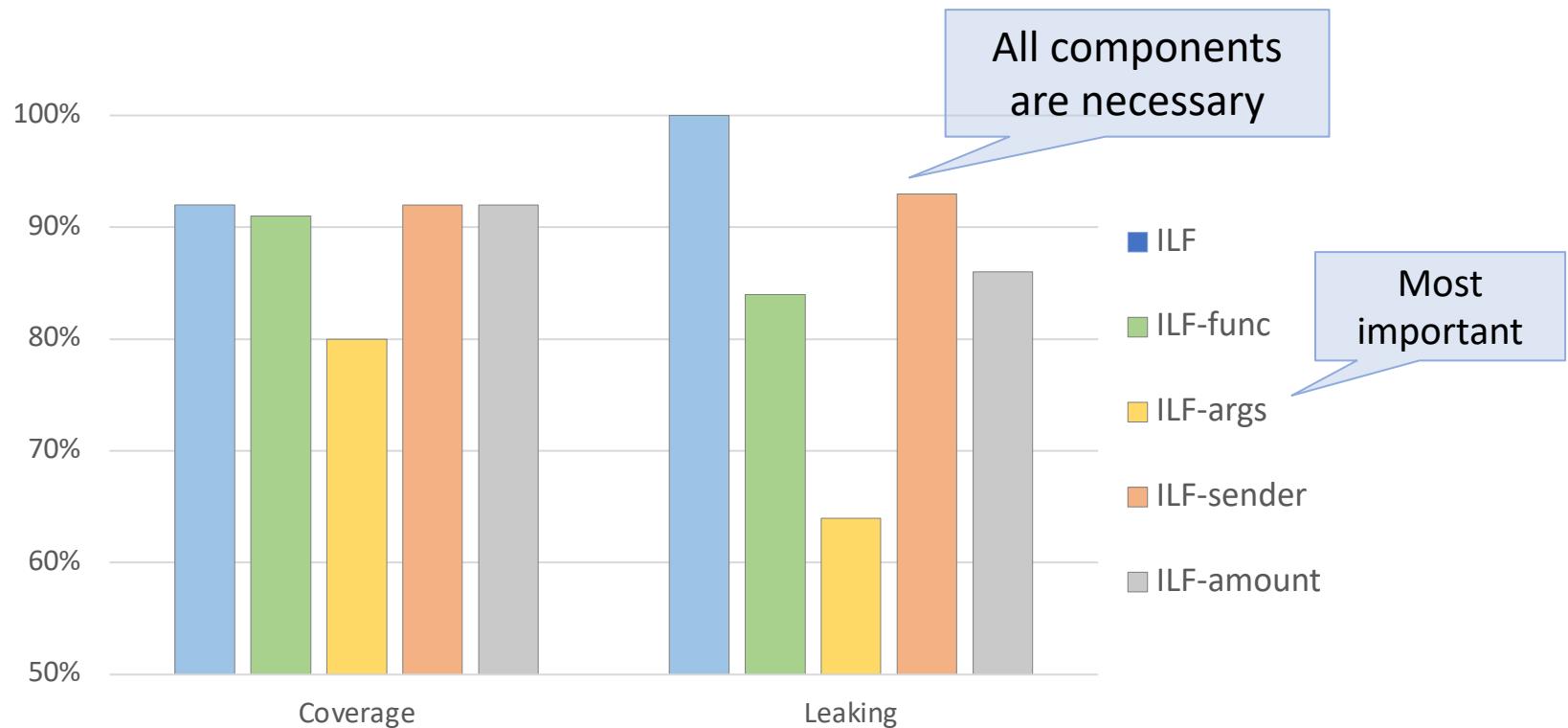
% of True Vulnerabilities



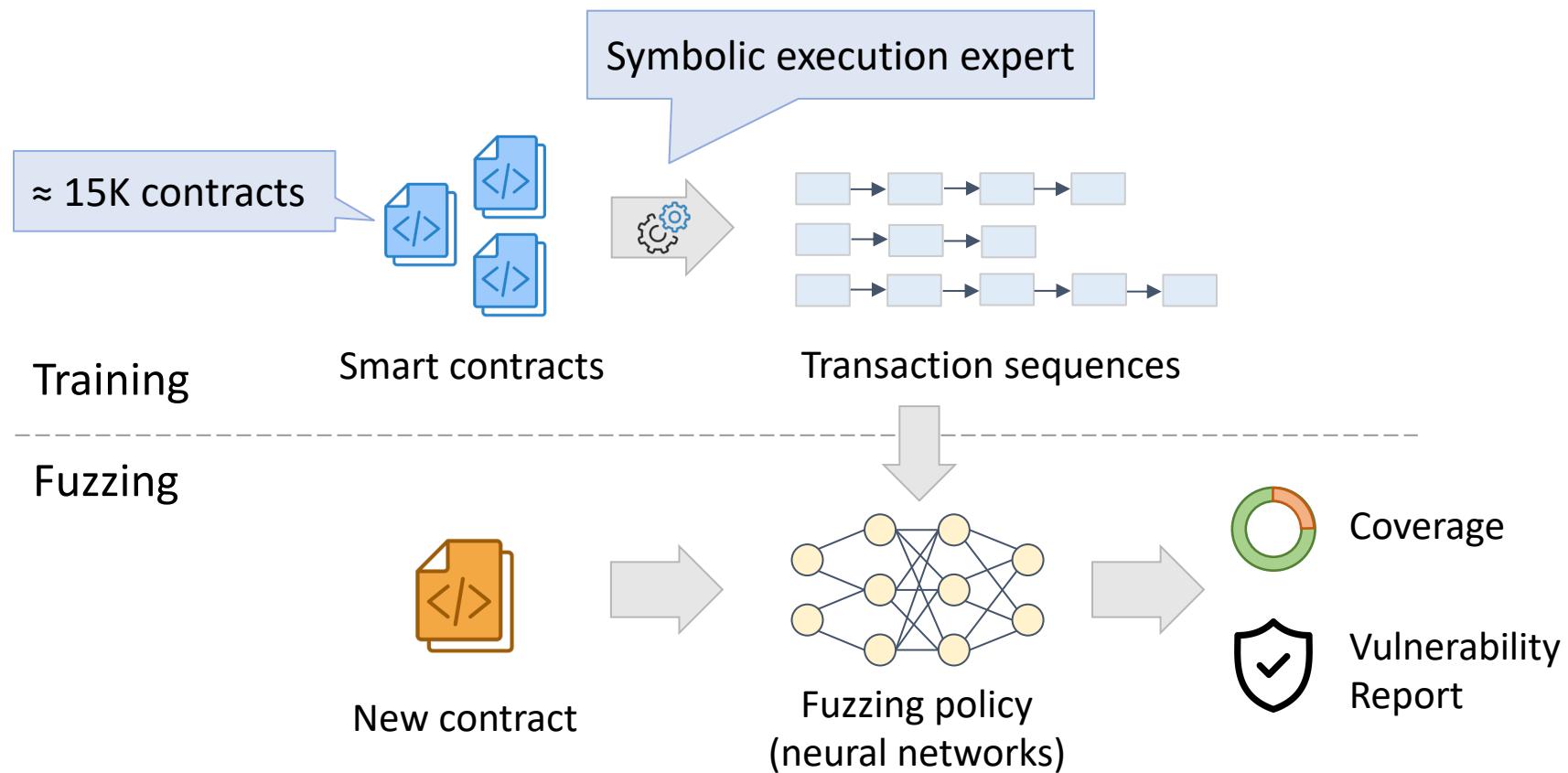
% of True Vulnerabilities



Importance of Policy Components



Summary



Rigorous Software Engineering

Applications of Analysis: Static Concurrency Checker

Prof. Martin Vechev

So far, we studied the theory behind program analysis together with few examples of numerical domains (e.g., intervals)

In this lecture, we will see an example of applying program analysis techniques for solving a cool problem in concurrency. This a rather general problem which arises in various practical settings, for instance, HPC, GPU programming and others.

Motivation

Determinism especially wanted in applications which use structured parallel languages, where there is renewed interest

- FJ (Java), Cilk (MIT), X10 (IBM), DPJ (UIUC), TPL (Microsoft), CUDA (NVIDIA) (GPUs)

Increasing number of Parallel Algorithms

- Scientific Computing, Signal Processing, Encryption, Sorting, Searching, String Indexing
- Meant to be **deterministic**: for the same input, produce the same output

Motivating Example

```
void update (double[][] G, int start, int last, double c1, double c2, int nm, int ps) {  
    for (tid = start ; tid < last ; tid += 1) {  
        fork {  
            int i = 2 * tid - ps;  
            for (int j=1; j < nm; j++)  
                G[i][j] = c1 * (G[i-1][j] + G[i+1][j] +  
                                 G[i][j-1] + G[i][j+1]) +  
                                 c2 * G[i][j];  
        }  
    }  
    join;  
}
```

Uses parallel computation to apply the method of successive over-relaxation for solving a system of linear equations.

Is this program deterministic? If no, find a counter-example. If yes, prove it.

Property we want: Determinism

Any pair of terminating executions starting with
equivalent input states, end in **equivalent output** states

Proving arbitrary programs deterministic is hard, instead,
prove a **stronger** property which implies determinism

Sources of Unboundedness

- Unbounded Heap
- Unbounded range of array indices
- Unbounded number of threads
 - Spawned in a loop

Dealing with Unboundedness

Use Abstraction

Bounded representation of an infinite number of states

Dealing with Unboundedness

- Unbounded Heap
 - Compute finite set of abstract locations
 - Using flow-insensitive **points-to analysis**
- Unbounded range of array indices
 - Compute symbolic index constraints
 - Using **numerical abstractions**
- Unbounded number of threads
 - We will not discuss this

Check (Abstract) Conflict-Freedom

Step 1: Compute all reachable **abstract** states

Step 2: Check if abstract states are **conflict-free**

Step 1: Compute Abstract States

Lets **compute invariants** for each thread. This computation is done via **sequential analysis** and is based on running an abstract interpreter using numerical analysis (e.g., octagons) and pointer analysis.

Compute Abstract States: Example

```
void update (double[][] G, int start, int last, double c1, double c2, int nm, int ps) {  
    for (tid = start ; tid < last ; tid += 1) {  
        fork {  
            int i = 2 * tid - ps;  
            1: double[] Gi = G [i];  
            2: double[] Gim = G [i - 1];  
            3: double[] Gip = G [i + 1];  
            for (int j=1; j<nm; j++) {  
                4: double tmp1 = Gim[j]  
                5: double tmp2 = Gip[j]  
                6: double tmp3 = Gi[j-1]  
                7: double tmp4 = Gi[j+1]  
                8: double tmp5 = Gi[j];  
                9: Gi[j] = c1 * (tmp1 + tmp2 + tmp3 + tmp4) + c2 * tmp5  
            } }  
        join; }  
}
```

Compute Abstract States: Example

```
void update (double[][] G, int start, int last, double c1, double c2, int nm, int ps) {  
    for (tid = start ; tid < last ; tid += 1) {  
        fork {  
            int i = 2 * tid - ps;  
            1: double[] Gi = G [i];  
            2: double[] Gim = G [i - 1];  
            3: double[] Gip = G [i + 1];  
            for (int j=1; j<nm; j++) {  
                4: double tmp1 = Gim[j]  
                5: double tmp2 = Gip[j]  
                6: double tmp3 = Gi[j-1]  
                7: double tmp4 = Gi[j+1]  
                8: double tmp5 = Gi[j];  
                9: Gi[j] = c1 * (tmp1 + tmp2 + tmp3 + tmp4) + c2 * tmp5  
            } }  
        join; }  
    } }  
}
```

$r_a = (G \rightarrow \{A_G\},$
 $Gim \rightarrow T,$
 $Gip \rightarrow T,$
 $Gi \rightarrow T)$

$\sigma_1 = \{pc=1, idx = 2*tid - ps\}$
 $\sigma_2 = \{pc=2, idx = 2*tid - ps - 1\}$
 $\sigma_3 = \{pc=3, idx = 2*tid - ps + 1\}$
 $\sigma_4 = \{pc=4, 1 \leq idx < nm\}$
 $\sigma_5 = \{pc=5, 1 \leq idx < nm\}$
 $\sigma_6 = \{pc=6, 0 \leq idx < nm-1\}$
 $\sigma_7 = \{pc=7, 2 \leq idx < nm+1\}$
 $\sigma_8 = \{pc=8, 1 \leq idx < nm\}$
 $\sigma_9 = \{pc=9, 1 \leq idx < nm\}$

Meaning of abstract states

These abstract states were computed by analyzing the program sequentially.

But our programs are parallel/concurrent. They can have behaviors not present in the sequential programs.

So how should we think of these computed abstract states in our concurrent setting? How do we know they really approximate the behaviors of the concurrent program?

Abstract States

$\sigma_1 = \{pc = 1, idx = 2*tid - ps\}$

$\sigma_2 = \{pc = 2, idx = 2*tid - ps - 1\}$

$\sigma_3 = \{pc = 3, idx = 2*tid - ps + 1\}$

$\sigma_4 = \{pc = 4, 1 \leq idx < nm\}$

$\sigma_5 = \{pc = 5, 1 \leq idx < nm\}$

$\sigma_6 = \{pc = 6, 0 \leq idx < nm-1\}$

$\sigma_7 = \{pc = 7, 2 \leq idx < nm+1\}$

$\sigma_8 = \{pc = 8, 1 \leq idx < nm\}$

$\sigma_9 = \{pc = 9, 1 \leq idx < nm\}$

Cartesian States

tid = 1

pc₁ = 1, ps = 0, idx₁ = 2

pc₁ = 1, ps = 1, idx₁ = 1

pc₁ = 1, ps = 2, idx₁ = 0

tid = 4

pc₄ = 1, ps = 0, idx₄ = 8

pc₄ = 1, ps = 3, idx₄ = 5

pc₄ = 1, ps = 7, idx₄ = 1

tid = 1

pc₁ = 5, nm = 5, idx₁ = 4

pc₁ = 5, nm = 9, idx₁ = 8

pc₁ = 5, nm = 3, idx₁ = 2

tid = 4

pc₄ = 5, nm = 3, idx₄ = 2

pc₄ = 5, nm = 1, idx₄ = 0

pc₄ = 5, nm = 9, idx₄ = 7

$$r_a = (G \rightarrow \{A_G\}, \\ Gim \rightarrow T, \\ Gip \rightarrow T, \\ Gi \rightarrow T)$$

Abstract States

$\sigma_1 = \{pc = 1, idx = 2*tid - ps\}$

$\sigma_2 = \{pc = 2, idx = 2*tid - ps - 1\}$

$\sigma_3 = \{pc = 3, idx = 2*tid - ps + 1\}$

$\sigma_4 = \{pc = 4, 1 \leq idx < nm\}$

$\sigma_5 = \{pc = 5, 1 \leq idx < nm\}$

$\sigma_6 = \{pc = 6, 0 \leq idx < nm-1\}$

$\sigma_7 = \{pc = 7, 2 \leq idx < nm+1\}$

$\sigma_8 = \{pc = 8, 1 \leq idx < nm\}$

$\sigma_9 = \{pc = 9, 1 \leq idx < nm\}$

Cartesian States

tid = 1

pc₁ = 1, ps = 0, idx₁ = 2

pc₁ = 1, ps = 1, idx₁ = 1

pc₁ = 1, ps = 2, idx₁ = 0

tid = 4

pc₄ = 1, ps = 0, idx₄ = 8

pc₄ = 1, ps = 3, idx₄ = 5

pc₄ = 1, ps = 7, idx₄ = 1

tid = 1

pc₁ = 5, nm = 5, idx₁ = 4

pc₁ = 5, nm = 9, idx₁ = 8

pc₁ = 5, nm = 3, idx₁ = 2

tid = 4

pc₄ = 5, nm = 3, idx₄ = 2

pc₄ = 5, nm = 1, idx₄ = 0

pc₄ = 5, nm = 9, idx₄ = 7

$$r_a = (G \rightarrow \{A_G\}, \\ Gim \rightarrow T, \\ Gip \rightarrow T, \\ Gi \rightarrow T)$$

Cartesian States

tid = 1

$pc_1 = 1, ps = 0, idx_1 = 2$

$pc_1 = 1, ps = 1, idx_1 = 1$

$pc_1 = 1, ps = 2, idx_1 = 0$

tid = 4

$pc_4 = 1, ps = 0, idx_4 = 8$

$pc_4 = 1, ps = 3, idx_4 = 5$

$pc_4 = 1, ps = 7, idx_4 = 1$

tid = 1

$pc_1 = 5, nm = 5, idx_1 = 4$

$pc_1 = 5, nm = 9, idx_1 = 8$

$pc_1 = 5, nm = 3, idx_1 = 2$

tid = 4

$pc_4 = 5, nm = 3, idx_4 = 2$

$pc_4 = 5, nm = 1, idx_4 = 0$

$pc_4 = 5, nm = 9, idx_4 = 7$

Cartesian States

tid = 1

$pc_1 = 1, ps = 0, idx_1 = 2$

$pc_1 = 1, ps = 1, idx_1 = 1$

$pc_1 = 1, ps = 2, idx_1 = 0$

tid = 4

$pc_4 = 1, ps = 0, idx_4 = 8$

$pc_4 = 1, ps = 3, idx_4 = 5$

$pc_4 = 1, ps = 7, idx_4 = 1$

Concrete Program States

tid = 1

$pc_1 = 5, nm = 5, idx_1 = 4$

$pc_1 = 5, nm = 9, idx_1 = 8$

$pc_1 = 5, nm = 3, idx_1 = 2$

tid = 4

$pc_4 = 5, nm = 3, idx_4 = 2$

$pc_4 = 5, nm = 1, idx_4 = 0$

$pc_4 = 5, nm = 9, idx_4 = 7$

Cartesian States

tid = 1

$pc_1 = 1, ps = 0, idx_1 = 2$

$pc_1 = 1, ps = 1, idx_1 = 1$

$pc_1 = 1, ps = 2, idx_1 = 0$

tid = 4

$pc_4 = 1, ps = 0, idx_4 = 8$

$pc_4 = 1, ps = 3, idx_4 = 5$

$pc_4 = 1, ps = 7, idx_4 = 1$

Concrete Program States

tid = 1

$pc_1 = 5, nm = 5, idx_1 = 4$

$pc_1 = 5, nm = 9, idx_1 = 8$

$pc_1 = 5, nm = 3, idx_1 = 2$

tid = 4

$pc_4 = 5, nm = 3, idx_4 = 2$

$pc_4 = 5, nm = 1, idx_4 = 0$

$pc_4 = 5, nm = 9, idx_4 = 7$

Cartesian States

Concrete Program States

tid = 1

$pc_1 = 1, ps = 0, idx_1 = 2$

$pc_1 = 1, ps = 1, idx_1 = 1$

$pc_1 = 1, ps = 2, idx_1 = 0$

tid = 4

$pc_4 = 1, ps = 0, idx_4 = 8$

$pc_4 = 1, ps = 3, idx_4 = 5$

$pc_4 = 1, ps = 7, idx_4 = 1$

$pc_1 = 1, idx_1 = 2, pc_4 = 1, idx_4 = 8, ps = 0$

$pc_1 = 1, idx_1 = 2, pc_4 = 1, idx_4 = 8, ps = 0$

tid = 1

$pc_1 = 5, nm = 5, idx_1 = 4$

$pc_1 = 5, nm = 9, idx_1 = 8$

$pc_1 = 5, nm = 3, idx_1 = 2$

tid = 4

$pc_4 = 5, nm = 3, idx_4 = 2$

$pc_4 = 5, nm = 1, idx_4 = 0$

$pc_4 = 5, nm = 9, idx_4 = 7$

Is this state
possible ?

Cartesian States

tid = 1

$pc_1 = 1, ps = 0, idx_1 = 2$

$pc_1 = 1, ps = 1, idx_1 = 1$

$pc_1 = 1, ps = 2, idx_1 = 0$

tid = 4

$pc_4 = 1, ps = 0, idx_4 = 8$

$pc_4 = 1, ps = 3, idx_4 = 5$

$pc_4 = 1, ps = 7, idx_4 = 1$

Concrete Program States

$pc_1 = 1, idx_1 = 2, pc_4 = 1, idx_4 = 8, ps = 0$

$pc_1 = 5, idx_1 = 2, pc_4 = 1, idx_4 = 8, ps = 0, nm = 3$

tid = 1

$pc_1 = 5, nm = 5, idx_1 = 4$

$pc_1 = 5, nm = 9, idx_1 = 8$

$pc_1 = 5, nm = 3, idx_1 = 2$

tid = 4

$pc_4 = 5, nm = 3, idx_4 = 2$

$pc_4 = 5, nm = 1, idx_4 = 0$

$pc_4 = 5, nm = 9, idx_4 = 7$

Cartesian States

tid = 1

$pc_1 = 1, ps = 0, idx_1 = 2$

$pc_1 = 1, ps = 1, idx_1 = 1$

$pc_1 = 1, ps = 2, idx_1 = 0$

tid = 4

$pc_4 = 1, ps = 0, idx_4 = 8$

$pc_4 = 1, ps = 3, idx_4 = 5$

$pc_4 = 1, ps = 7, idx_4 = 1$

Concrete Program States

$pc_1 = 1, idx_1 = 2, pc_4 = 1, idx_4 = 8, ps = 0$

$pc_1 = 5, idx_1 = 2, pc_4 = 1, idx_4 = 8, ps = 0, nm=3$

tid = 1

$pc_1 = 5, nm = 5, idx_1 = 4$

$pc_1 = 5, nm = 9, idx_1 = 8$

$pc_1 = 5, nm = 3, idx_1 = 2$

tid = 4

$pc_4 = 5, nm = 3, idx_4 = 2$

$pc_4 = 5, nm = 1, idx_4 = 0$

$pc_4 = 5, nm = 9, idx_4 = 7$

$pc_1 = 5, idx_1 = 2, pc_4 = 5, idx_4 = 2, nm=3$

Step 2: Check the property

Now that we've computed the abstract states, we can go ahead and actually check the property we are interested in proving on these abstract states

Abstract Conflict-Free Checker

For every pair of **abstract** states σ_i, σ_k :

if st_i in σ_i and st_k in σ_k **may happen** in parallel

$A = \text{combine } \sigma_i, \sigma_k \text{ and } (tid_i \neq tid_k)$

if $ALoc(st_i)$ and $ALoc(st_k)$ **may equal** in A

exit (“Program may be **non-deterministic**”)

exit (“Program is **deterministic**”)

temporal check

spatial check

- Combine means meet in the domain
- Note: thread constraint ($tid_i \neq tid_k$) not expressible in polyhedra

Abstract Conflict-Free Checker

For every pair of **abstract** states σ_i, σ_k :

if st_i in σ_i and st_k in σ_k **may happen** in parallel

$A = \text{combine } \sigma_i, \sigma_k$ and $(tid_i \neq tid_k)$

if $ALoc(st_i)$ and $ALoc(st_k)$ meet in A

exit (“Program is **deterministic**”)

Can be answered with a
may-happen analysis
(many works on this)

exit (“Program is **deterministic**”)

- Combine means meet in the domain
- Note: thread constraint ($tid_i \neq tid_k$) not expressible in polyhedra

May Equal (Arrays)

ALoc(st) := (pointer variable, integer variable)

Example: ALoc (G[i] = 128) := (G, i)

May ALoc(st₁) := (a, i) equal ALoc(st₂) := (b, k) in abstract state A ?

*For presentation, assume conflicts can occur only on arrays not variables.

May Equal (Arrays)

ALoc(st) := (pointer variable, integer variable)

Example: ALoc (G[i] = 128) := (G, i)

May ALoc(st₁) := (a, i) equal ALoc(st₂) := (b, k) in abstract state A ?

if (a _____ b) and (i _____ k)

output: Yes

else

output: No

*For presentation, assume conflicts can occur only on arrays not variables.

May Equal (Arrays)

ALoc(st) := (pointer variable, integer variable)

Example: ALoc (G[i] = 128) := (G, i)

May ALoc(st₁) := (a, i) equal ALoc(st₂) := (b, k) in abstract state A ?

if (a may alias b) and (i may overlap k)

output: Yes

else

output: No

*For presentation, assume conflicts can occur only on arrays not variables.

May Equal (Arrays)

$\text{ALoc(st)} := (\text{pointer variable}, \text{ integer variable})$

Example: $\text{ALoc}(G[i] = 128) := (G, i)$

May $\text{ALoc(st}_1\text{)} := (a, i)$ equal $\text{ALoc(st}_2\text{)} := (b, k)$ in abstract state A ?

if (a may alias b) and (i may overlap k)

output: Yes

else

output: No

*For presentation, assume co

If in A_H : $\text{points-to}(a) \cap \text{points-to}(b) \neq \emptyset$

May Equal (Arrays)

$\text{ALoc(st)} := (\text{pointer variable}, \text{ integer variable})$

Example: $\text{ALoc}(G[i] = 128) := (G, i)$

May $\text{ALoc(st}_1\text{)} := (a, i)$ equal $\text{ALoc(st}_2\text{)} := (b, k)$ in abstract state A ?

if (a may alias b) and (i may overlap k)

output: Yes

else

output: No

If:

$A_N \sqcap (i = k) \neq \perp$

*For presentation, assume co

If in A_H : $\text{points-to}(a) \cap \text{points-to}(b) \neq \emptyset$

1. Compute Abstract States: Example

```
void update (double[][] G, int start, int last, double c1, double c2, int nm, int ps) {  
    for (tid = start ; tid < last ; tid += 1) {  
        fork {  
            int i = 2 * tid - ps;  
            1: double[] Gi = G [i];  
            2: double[] Gim = G [i - 1];  
            3: double[] Gip = G [i + 1];  
            for (int j=1; j<nm; j++) {  
                4: double tmp1 = Gim[j]  
                5: double tmp2 = Gip[j]  
                6: double tmp3 = Gi[j-1]  
                7: double tmp4 = Gi[j+1]  
                8: double tmp5 = Gi[j];  
                9: Gi[j] = c1 * (tmp1 + tmp2 + tmp3 + tmp4) + c2 * tmp5  
            } }  
        join; }
```

$$r_a = (G \rightarrow \{A_G\}, Gim \rightarrow T, Gip \rightarrow T, Gi \rightarrow T)$$
$$\sigma_1 = \{pc=1, idx = 2*tid - ps\}$$
$$\sigma_2 = \{pc=2, idx = 2*tid - ps - 1\}$$
$$\sigma_3 = \{pc=3, idx = 2*tid - ps + 1\}$$
$$\sigma_4 = \{pc=4, 1 \leq idx < nm\}$$
$$\sigma_5 = \{pc=5, 1 \leq idx < nm\}$$
$$\sigma_6 = \{pc=6, 0 \leq idx < nm-1\}$$
$$\sigma_7 = \{pc=7, 2 \leq idx < nm+1\}$$
$$\sigma_8 = \{pc=8, 1 \leq idx < nm\}$$
$$\sigma_9 = \{pc=9, 1 \leq idx < nm\}$$

2. Pick a pair of abstract states

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

3. Temporal check: may happen ?

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

Statements:

6: $tmp3 = Gi_1[idx_1]$

vs.

9: $Gi_2[idx_2] = ...$

3. Temporal check: may happen ?

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

Statements:

6: $tmp3 = Gi_1[idx_1]$ vs. 9: $Gi_2[idx_2] = ...$

May Happen in Parallel ? Yes

4. Spatial check: may equal ?

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

Statements:

6: $tmp3 = Gi_1[idx_1]$ vs. 9: $Gi_2[idx_2] = ...$

Abstract Locations:

$ALoc_6 := (Gi_1, idx_1)$ vs. $ALoc_9 := (Gi_2, idx_2)$

4. Spatial check: may equal ?

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

Statements:

6: $tmp3 = Gi_1[idx_1]$ vs. 9: $Gi_2[idx_2] = ...$

Abstract Locations:

$ALoc_6 := (Gi_1, idx_1)$ vs. $ALoc_9 := (Gi_2, idx_2)$

may Gi_1 alias Gi_2 ? Yes

may idx_1 overlap idx_2 ? Yes

4. Spatial check: may equal ?

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

Statements:

6: $tmp3 = Gi_1[idx_1]$

vs.

9: $Gi_2[idx_2] = ...$

Program may be non-deterministic

Abstract Locations:

$ALoc_6 := (Gi_1, idx_1)$

vs.

$ALoc_9 := (Gi_2, idx_2)$

may Gi_1 alias Gi_2 ? Yes

may idx_1 overlap idx_2 ? Yes

Heap abstraction imprecise

Aliasing information **inside** reference arrays lost!

Example reference arrays: Object A[], double G[][]

Points-to information not enough

Property Wanted: references inside array are **distinct**

$$\forall i.j.A. \ i \neq j \Rightarrow A[i] \neq A[j]$$

Very difficult to prove in general !

Domain-specific solution

- In most numerical HPC/GPU programs
 - Reference arrays are initialized **only** with fresh objects
 - Example: `A[i] = new Object();`
 - Never updated after (parallel threads do not modify A)
- Easy to prove as a global invariant
 - Initialization in a single procedure
 - Simple analysis

Spatial check revisited: may equal ?

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

Statements:

6: $tmp3 = Gi_1[idx_1]$ vs. 9: $Gi_2[idx_2] = ...$

Abstract Locations:

$ALoc_6 := (Gi_1, idx_1)$ vs. $ALoc_9 := (Gi_2, idx_2)$

may Gi_1 alias Gi_2 ?

Let us look where Gi_1 and Gi_2 come from

Compute Abstract States: Example

```
void update (double[][] G, int start, int last, double c1, double c2, int nm, int ps) {  
    for (tid = start ; tid < last ; tid += 1) {  
        fork {  
            int i = 2 * tid - ps;  
            1: double[] Gi = G [i];  
            2: double[] Gim = G [i - 1];  
            3: double[] Gip = G [i + 1];  
            for (int j=1; j<nm; j++) {  
                4: double tmp1 = Gim[j]  
                5: double tmp2 = Gip[j]  
                6: double tmp3 = Gi[j-1]  
                7: double tmp4 = Gi[j+1]  
                8: double tmp5 = Gi[j];  
                9: Gi[j] = c1 * (tmp1 + tmp2 + tmp3 + tmp4) + c2 * tmp5  
            } }  
        join; }
```

$$r_a = (G \rightarrow \{A_G\}, Gim \rightarrow T, Gip \rightarrow T, Gi \rightarrow T)$$
$$\sigma_1 = \{pc=1, idx = 2*tid - ps\}$$
$$\sigma_2 = \{pc=2, idx = 2*tid - ps - 1\}$$
$$\sigma_3 = \{pc=3, idx = 2*tid - ps + 1\}$$
$$\sigma_4 = \{pc=4, 1 \leq idx < nm\}$$
$$\sigma_5 = \{pc=5, 1 \leq idx < nm\}$$
$$\sigma_6 = \{pc=6, 0 \leq idx < nm-1\}$$
$$\sigma_7 = \{pc=7, 2 \leq idx < nm+1\}$$
$$\sigma_8 = \{pc=8, 1 \leq idx < nm\}$$
$$\sigma_9 = \{pc=9, 1 \leq idx < nm\}$$

Spatial check revisited: may equal ?

$\sigma_1 = (pc_1 = 1, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), idx_1 = 2*tid_1-ps)$

$\sigma_1 = (pc_2 = 1, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), idx_2 = 2*tid_2-ps)$

Statements:

1: $Gi_1 = G[idx_1]$

vs.

1: $Gi_2 = G[idx_2]$

may Gi_1 alias Gi_2 ? No

(using $\forall i, j . i \neq j \Rightarrow A_G[i] \neq A_G[j]$)

* Note that both accesses above are read-only

Coming back to the main case

Spatial check revisited: may equal ?

$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$

$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$

Statements:

6: $tmp3 = Gi_1[idx_1]$ vs. 9: $Gi_2[idx_2] = ...$

Abstract Locations:

$ALoc_6 := (Gi_1, idx_1)$ vs. $ALoc_9 := (Gi_2, idx_2)$

may Gi_1 alias Gi_2 ? No

Spatial check revisited: may equal ?

$$\sigma_6 = (pc_1 = 6, (G_1 \rightarrow \{A_G\}, Gim_1 \rightarrow T, Gip_1 \rightarrow T, Gi_1 \rightarrow T), 0 \leq idx_1 < nm - 1)$$
$$\sigma_9 = (pc_2 = 9, (G_2 \rightarrow \{A_G\}, Gim_2 \rightarrow T, Gip_2 \rightarrow T, Gi_2 \rightarrow T), 1 \leq idx_2 < nm)$$

Statements:

6: tmp3 = Gi₁[idx₁]

vs.

9: Gi₂[idx₂] = ...

Program is deterministic

Abstract Locations:

ALoc₆ := (Gi₁, idx₁)

vs.

ALoc₉ := (Gi₂, idx₂)

may Gi₁ alias Gi₂ ? No

Implementation

- Soot
 - Analysis works on Jimple intermediate representation
 - Can generate a lot of variables
 - Need to identify loops (to know where to widen)
- Apron library
 - For computing numerical invariants
- Benchmarks
 - Used adapted Java JGF benchmarks

Limitations

- Intra-Procedural
 - Had to inline some functions
 - Failed to inline in one benchmark
- Cannot handle specific kinds of non-linear constraints
 - $A[x*N + z] = c$, where N never changes
 - Cool problem
- Atomic sections (commute)
 - `atomic {x = x + 1; }`
- Accesses from nested primitive arrays
 - $A[B[i]] = 5$, where $B[i]$'s are distinct

Lecture Recap

- Application of Analysis to handling HPC/GPU programs
- To automatically prove determinism, prove a stronger property: conflict-freedom
- Limitations:
 - certain linear constraints, interference, better array content analysis

Rigorous Software Engineering

Mathematical Foundations for Analysis

Prof. Martin Vechev



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Sonatype Acquires MuseDev



Acquisition Pairs Developer-Friendly Source Code Analysis with Full-Spectrum Software Supply Chain Management

Fulton, MD – Tuesday, March 16, 2021 — **Sonatype**, the leader in developer-friendly tools for software supply chain management and security, today announced the acquisition of **MuseDev**, an innovative code analysis platform. MuseDev's core offering automatically analyzes and provides uniquely accurate feedback on each developer pull request, making it easy to find and fix critical security, performance, and reliability bugs during code review.

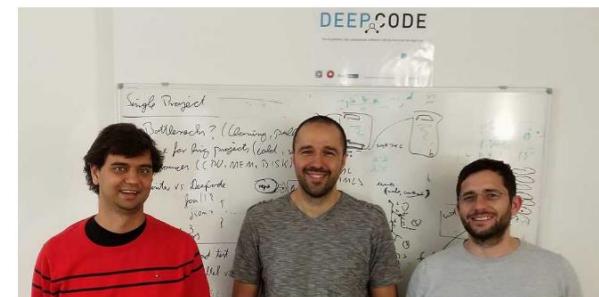
With the addition of Muse, the **Sonatype Nexus platform** now offers customers full-spectrum control of the cloud-native software development lifecycle including: first-party source code, third-party open source code, infrastructure as code, and containerized code.

"Beginning today, with the acquisition of MuseDev, we are further expanding our platform to help customers automatically control the quality of code their developers write," said Wayne Jackson, CEO of Sonatype. "Coupled with our recently launched Nexus Container and Infrastructure as Code

ETH AI spin-off DeepCode acquired by a unicorn in cybersecurity

23.09.2020

DeepCode, a spin-off of the Secure, Reliable and Intelligent Systems Lab, is to be acquired by Snyk, a world leader in developer-first security code analysis.



The DeepCode Founders (from left to right): Veselin Raychev (CTO), Boris Paskalev (CEO) and Prof. Martin Vechev, Head of the Secure, Reliable, and Intelligent Systems Lab. (Photo: ETH Zurich)

The ETH spin-off DeepCode was founded in 2016 von Veselin Raychev, Boris Paskalev und Prof. Martin Vechev. It has developed the first AI system that can learn from billions of program codes quickly, enabling AI-based detection of security and reliability code issues. Veselin Raychev has received prestigious awards for the underlying research.

Mathematical Concepts

- Structures: posets, lattices
- Functions: monotone, fixed points
- Approximating functions

Structures: Motivation

Structures are important as they define
the concrete and abstract domains

Partially Ordered Sets (posets)

A **partial order** is a binary relation $\sqsubseteq \subseteq L \times L$ on a set L with these properties:

- Reflexive: $\forall p \in L: p \sqsubseteq p$
- Transitive: $\forall p,q,r \in L: (p \sqsubseteq q \wedge q \sqsubseteq r) \Rightarrow p \sqsubseteq r$
- Anti-symmetric: $\forall p,q \in L: (p \sqsubseteq q \wedge q \sqsubseteq p) \Rightarrow p = q$

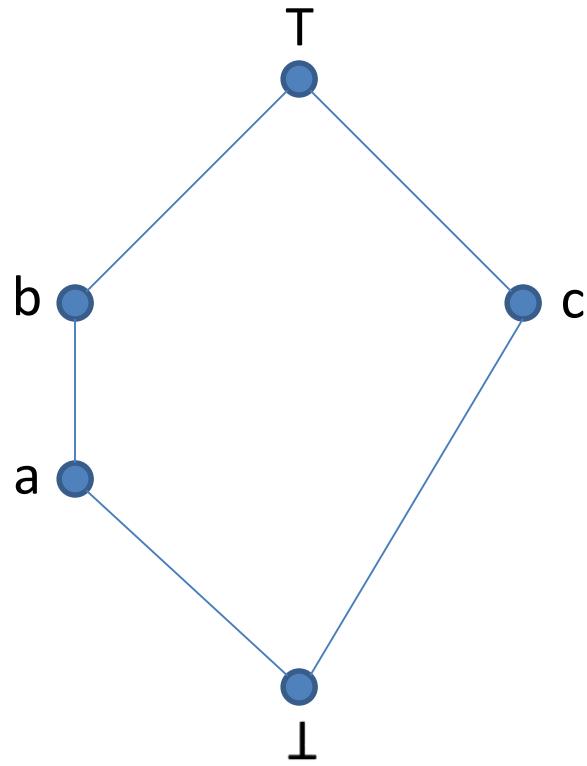
A poset (L, \sqsubseteq) is a set L equipped with a partial ordering \sqsubseteq

- For example: $(\wp(L), \subseteq)$ is a poset, where \wp denotes powerset

Intuition: captures implication between facts

- $p \sqsubseteq q$ intuitively means that $p \Rightarrow q$
- Later, we will say that if $p \sqsubseteq q$, then p is “more precise” than q (that is, p represents fewer concrete states than q)

Posets shown as Hasse Diagrams



given the set { a,b,c, T, \perp }

the Hasse diagram shows the order:

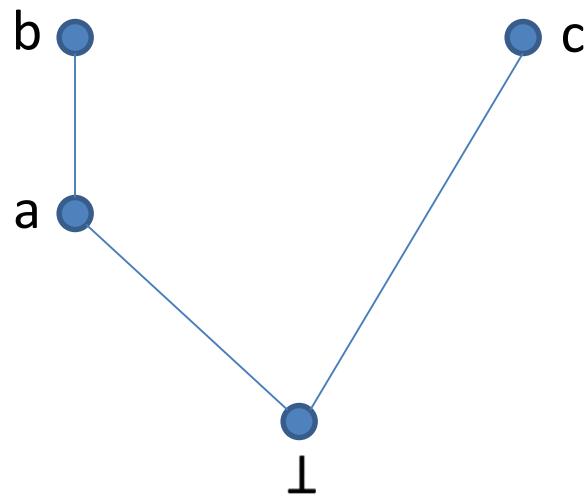
$$\{ (\perp, a), (\perp, c), (a, b), (b, T), (c, T), (a, a), (b, b), (c, c), (T, T), (\perp, \perp), (\perp, b), (\perp, T), (a, T) \}$$

Least / Greatest in Posets

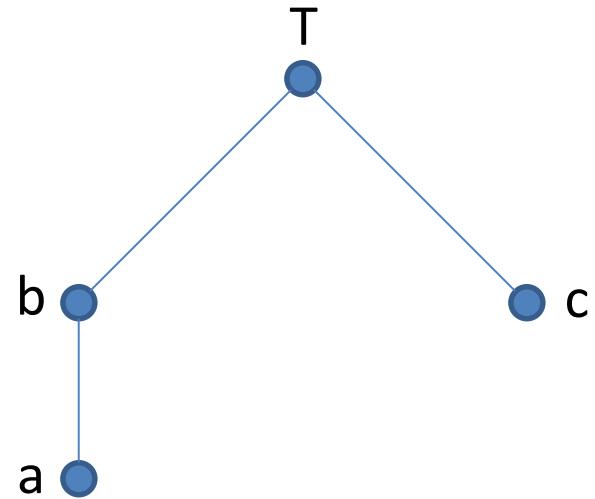
Given a poset (L, \sqsubseteq) , an element $\perp \in L$ is called the least element if it is smaller than all other elements of the poset: $\forall p \in L: \perp \sqsubseteq p$. The greatest element is an element T if $\forall p \in L: p \sqsubseteq T$.

The least and greatest elements may not exist, but if they do they are unique.

Least / Greatest: example



No greatest element



No least element

Example where both do not exist ?

Bounds in Posets

Given a poset (L, \sqsubseteq) and $Y \subseteq L$:

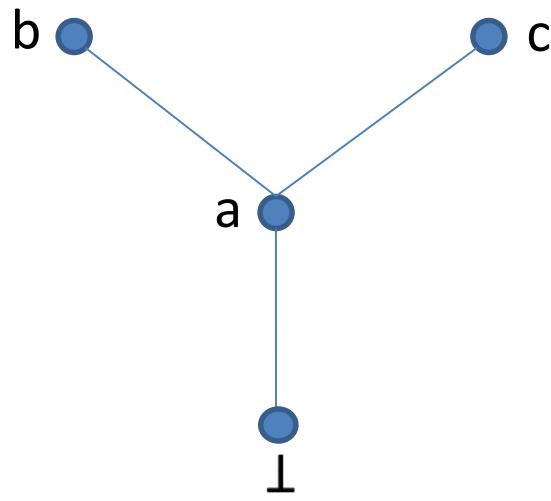
$u \in L$ is an **upper bound** of Y if $\forall p \in Y: p \sqsubseteq u$

$l \in L$ is a **lower bound** of Y if $\forall p \in Y: p \sqsupseteq l$

note that the bounds for Y **may not exist**

- $\sqcup Y \in L$ is a **least upper bound** of Y if $\sqcup Y$ is an upper bound of Y and $\sqcup Y \sqsubseteq u$ whenever u is another upper bound of Y .
 - $\sqcap Y \in L$ is **greatest lower bound** of Y if $\sqcap Y$ is a lower bound of Y and $\sqcap Y \sqsupseteq l$ whenever l is another lower bound of Y
-
- Note that $\sqcup Y$ and $\sqcap Y$ need not be in Y .
 - We often write $p \sqcup q$ for $\sqcup\{p, q\}$ and $p \sqcap q$ for $\sqcap\{p, q\}$

Bounds: example



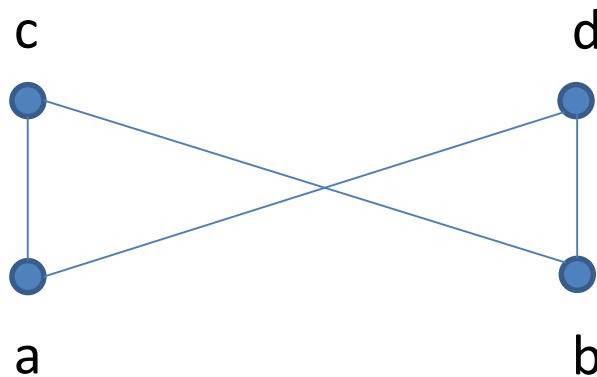
$\{b, c\}$ has no upper bound

$\{b, c\}$ has 2 lower bounds: a and \perp

where $\sqcap \{b, c\} = a$

No T element

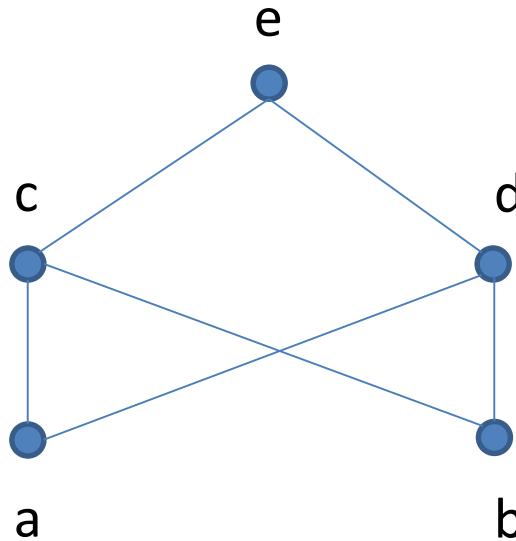
Bounds: example



is there a \sqcup for 'a' and 'b' ?

is there a \sqcap for 'c' and 'd' ?

Bounds: example



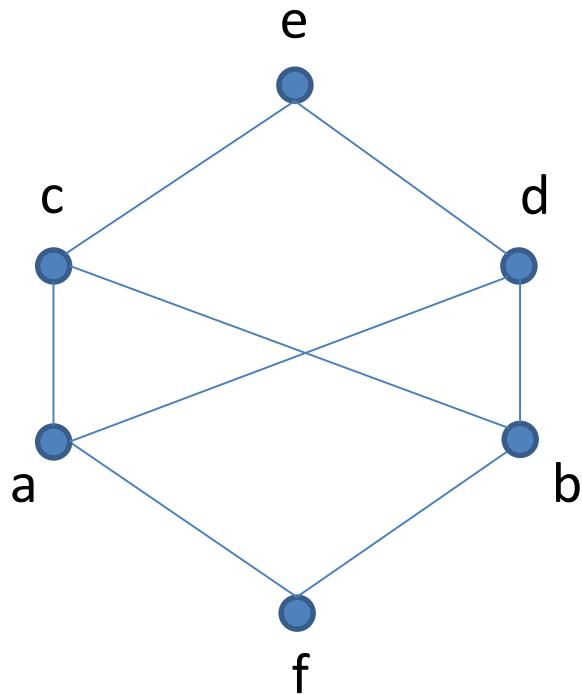
is there a \sqcup for 'a' and 'b' ?

Complete Lattices

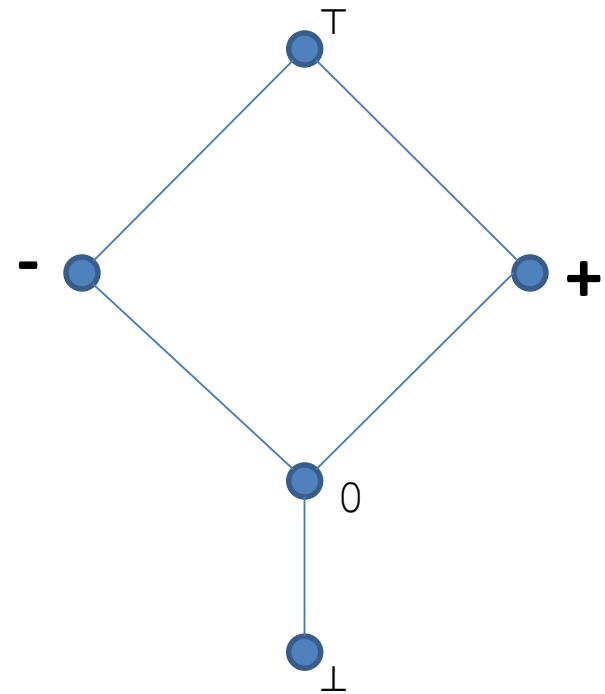
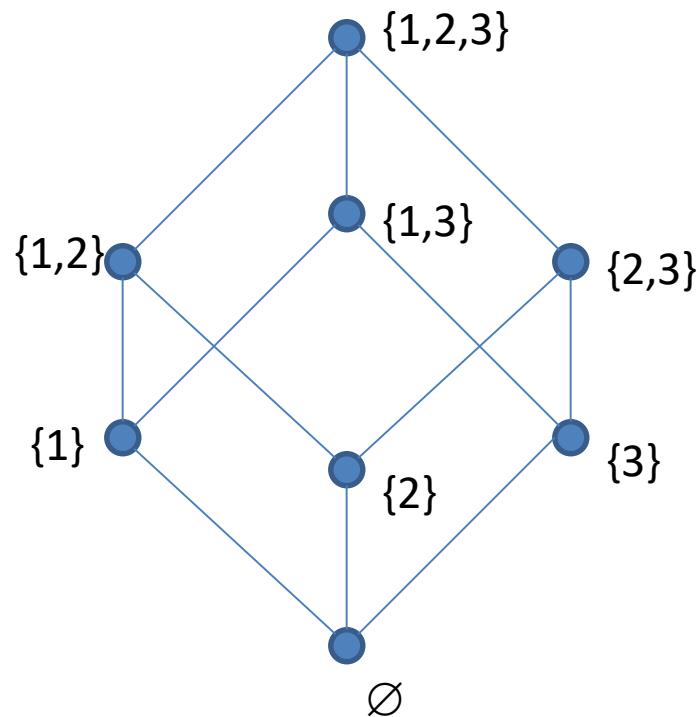
A **complete lattice** (L, \sqsubseteq, \sqcup) is a poset where $\sqcup Y$ and $\sqcap Y$ exist for any $Y \subseteq L$.

For example, for a set L , $(\mathcal{P}(L), \subseteq, \cup, \cap)$ is a complete lattice.

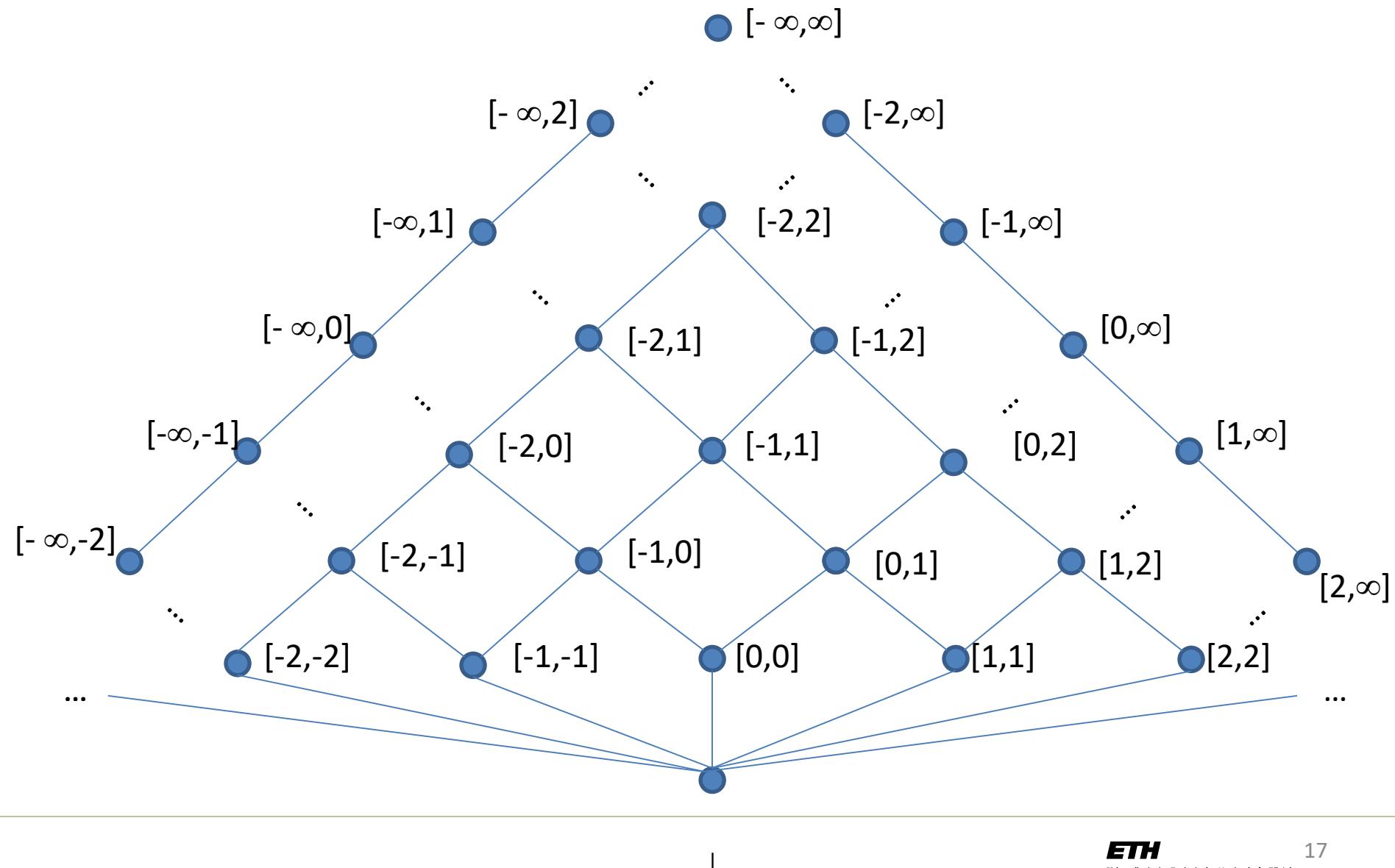
Is this a complete lattice ?



Complete Lattices: Examples



Complete Lattices: Examples



Later we will see that the set of traces
 $\llbracket P \rrbracket$ also belongs to a complete lattice

Concepts

- Structures: posets, lattices
- Functions: monotone, fixed points
- Approximating functions

Functions

A function $f: A \rightarrow B$ between two posets (A, \sqsubseteq) and (B, \leq) is increasing (monotone): $\forall a, b \in A: a \sqsubseteq b \Rightarrow f(a) \leq f(b)$

Often, we use the special case where the function is between elements in the same poset. That is, $f: A \rightarrow A$. Then a monotone function is: $\forall a, b \in A: a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$

Fixed Points

For a poset (L, \sqsubseteq) , function $f: L \rightarrow L$, and element $x \in L$:

- x is a **fixed point** iff $f(x) = x$
- x is a **post-fixedpoint** iff $f(x) \sqsubseteq x$

$\text{Fix}(f)$ denotes the set of all fixed points

$\text{Red}(f)$ = set of all post-fixedpoints

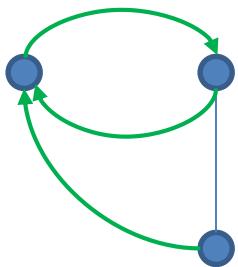
Least Fixed Points

For a poset (L, \sqsubseteq) and a function $f: L \rightarrow L$, we say that $\text{lfp}^{\sqsubseteq} f \in L$ is a **least fixed point** of f if:

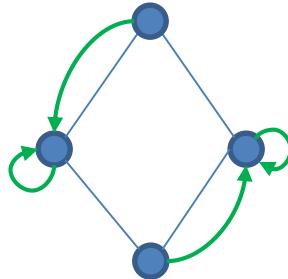
- $\text{lfp}^{\sqsubseteq} f$ is a fixed point
- It is the least fixed point: $\forall a \in L: a = f(a) \Rightarrow \text{lfp}^{\sqsubseteq} f \sqsubseteq a$

Note that the least fixed point **may not exist**.

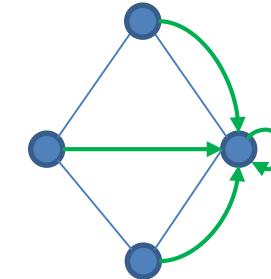
Fixed Points: Examples



- monotone function
- with no fixed point



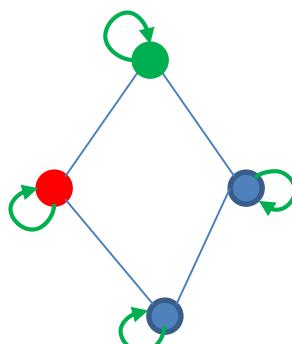
- not monotone function
- with 2 fixed points
- no least fixed point



- monotone function
- with one fixed point
- has a least fixed point



- monotone function
- with 2 fixed points
- no least fixed point



there exists a **post-fixedpoint** that is less than **some fixed point**

- monotone function
- 4 fixed points
- least fixed point

Tarski's fixed point theorem (part of it)

If $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ is a **complete lattice** and $f: L \rightarrow L$ is a **monotone function**, then

$\text{lfp}^{\sqsubseteq} f$ exists, and

$$\text{lfp}^{\sqsubseteq} f = \sqcap \text{Red}(f) \in \text{Fix}(f)$$

Note: the complete lattice can be of infinite height

Tarski's theorem tells us that a fixed point exists, but does not actually suggest an algorithm for computing it.

Next: we look at ways to compute a fixed point

Example of Abstraction

```
α ( { ⟨ 5, _ , {p↔o1, q↔o2} , {o1.k↔o3, o2.v↔o6} ⟩ ,  
      ⟨ 5, _ , {p↔o2, q↔o3} , {o1.k↔o3, o2.v↔o3} ⟩  
} )
```

Here, by $_$ we mean that the program has no integer variables.

Suppose that: object o_1 is allocated at site a_3 (program label 3)
object o_2 is allocated at site a_4 (program label 4)
object o_3 is allocated at site a_9 (program label 9)
object o_6 is allocated at site a_{31} (program label 31)

```
5 → ({p ↦ {a3, a4}, q ↦ {a4, a9} }, {a3.k↔{a9}, a4.v↔{a31, a9} })
```


Step 3: Define Abstract Transformers

We now need to define the effect of program statements manipulating pointers on the abstract domain. That is, creation of objects, pointer assignment and conditionals. It can be summarized as:

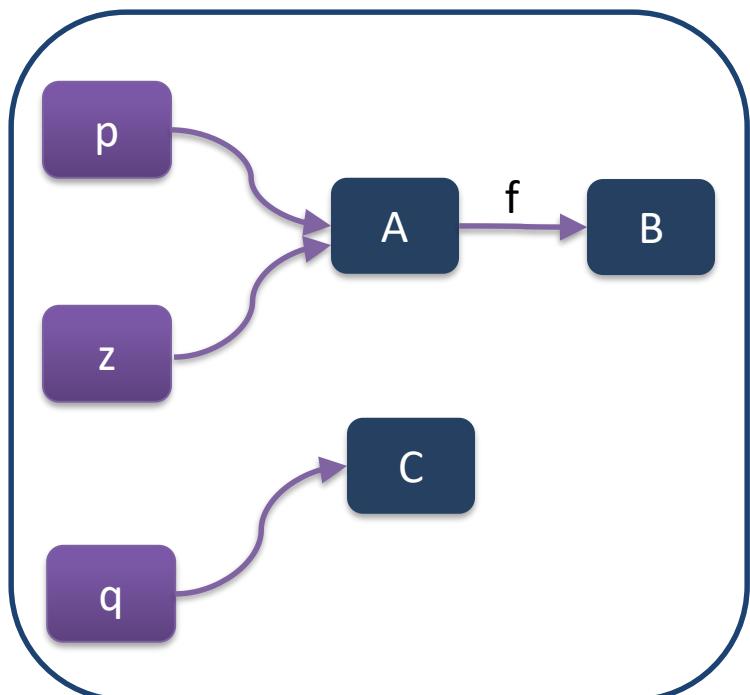
Statement/Expression	Description
$p = q$	compare two pointers
$p := \text{alloc}^\ell$	create new object
$p := q^\ell$	assign pointers
$p.f := q^\ell$	pointer heap store
$p := q.f^\ell$	pointer heap load

Lets us take a look at the most tricky one (pointer heap store). The rest are just direct assignments. The formal definitions are left as an **exercise**.

What about $p.f := q$?

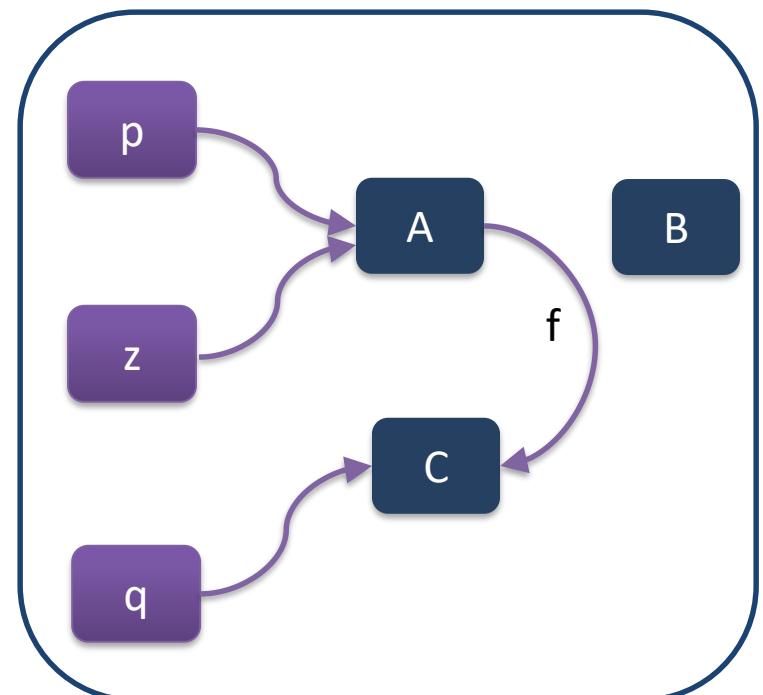
Say $p \mapsto \{A\}$, where $A.f \mapsto \{B\}$, and $q \mapsto \{C\}$. Can we have $A.f \mapsto \{C\}$ as a result?

Abstract Element AE1



$$p.f := q \quad \rightarrow$$

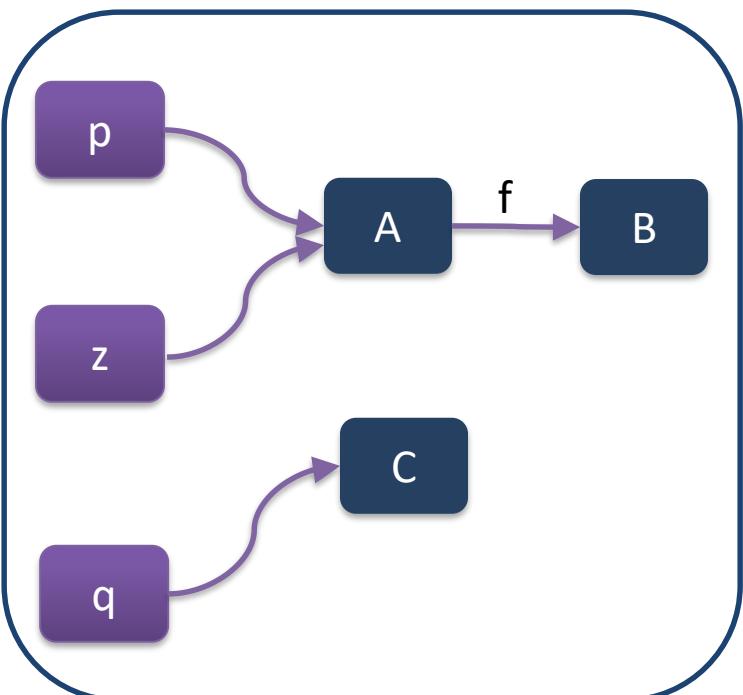
Abstract Element AE2
Is this result correct ?



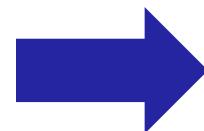
What about $p.f := q$?

To see why this is **not correct**, we need to think what the left side means in the **concrete** and what the right side means in the **concrete**.

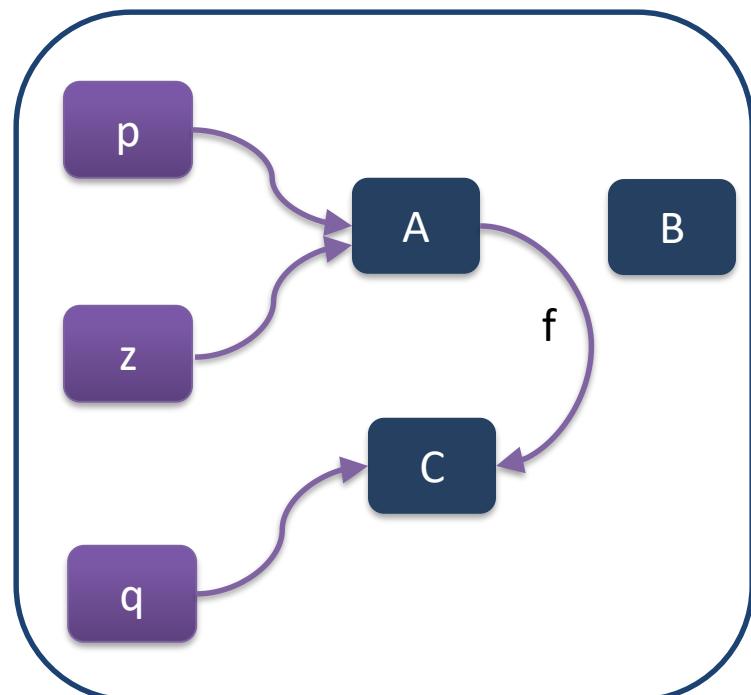
Abstract Element AE1



$p.f := q$

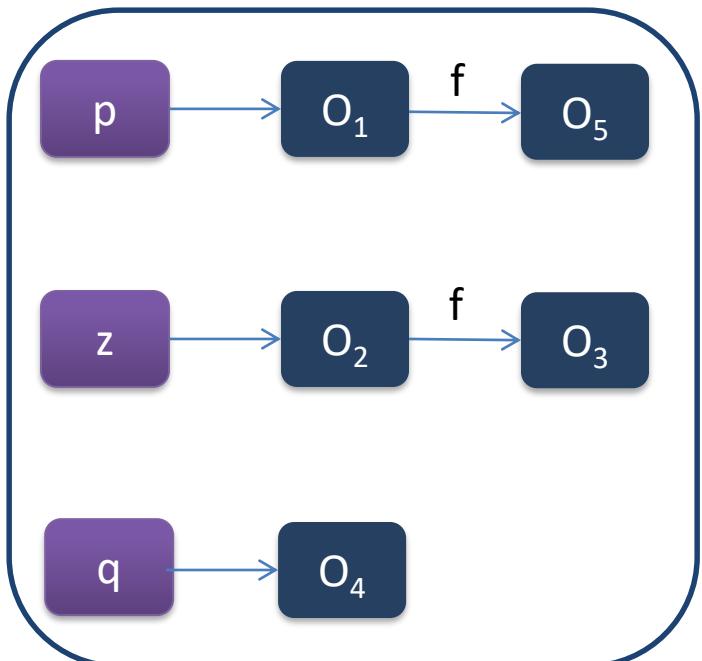


Abstract Element AE2



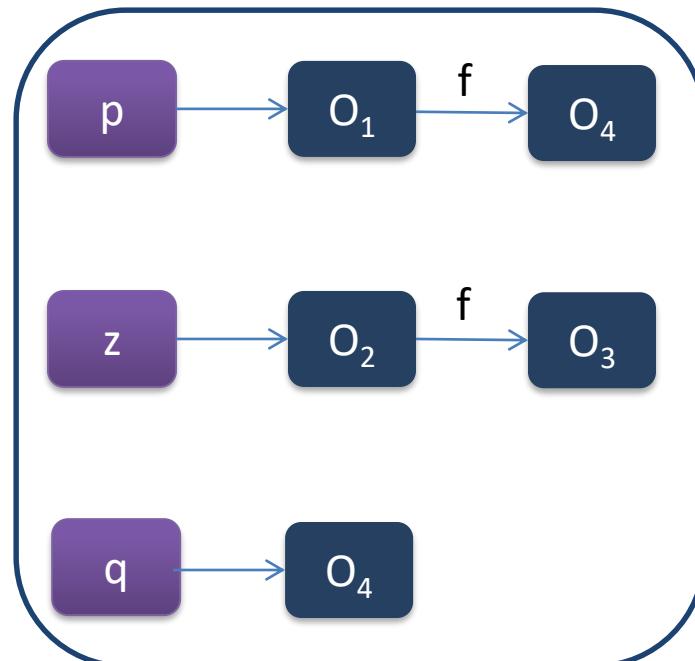
A Counter-Example in the Concrete

Possible Concrete Structure CE of AE1



$$p.f := q \quad \rightarrow$$

Possible Concrete Structure **not captured** by Abstract Element AE2

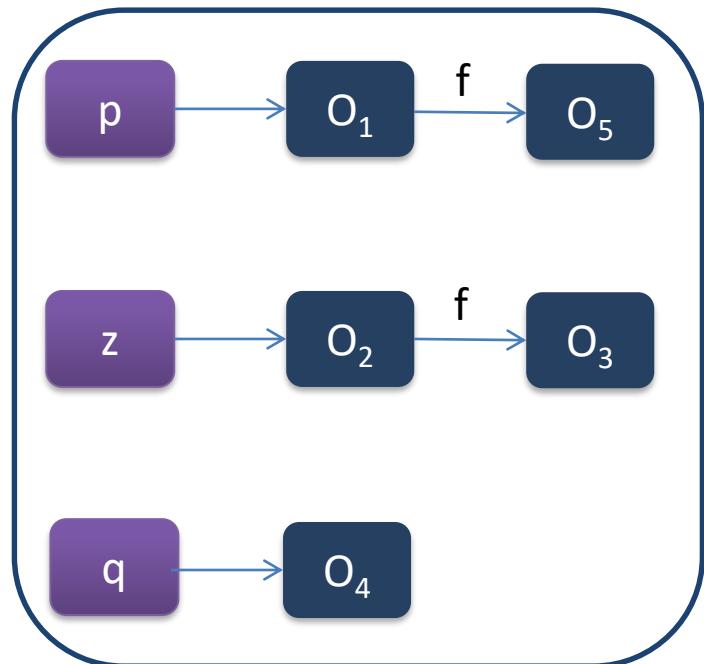


Concrete objects O_1 and O_2 allocated at site A
Concrete objects O_3 and O_5 allocated at site B
Concrete object O_4 allocated at site C

The reason this structure is not captured by AE2 is because in AE2 we can never reach an object allocated at site B via pointer z, while here, this is possible

A program which produces structure CE

Possible Concrete Structure CE of AE1

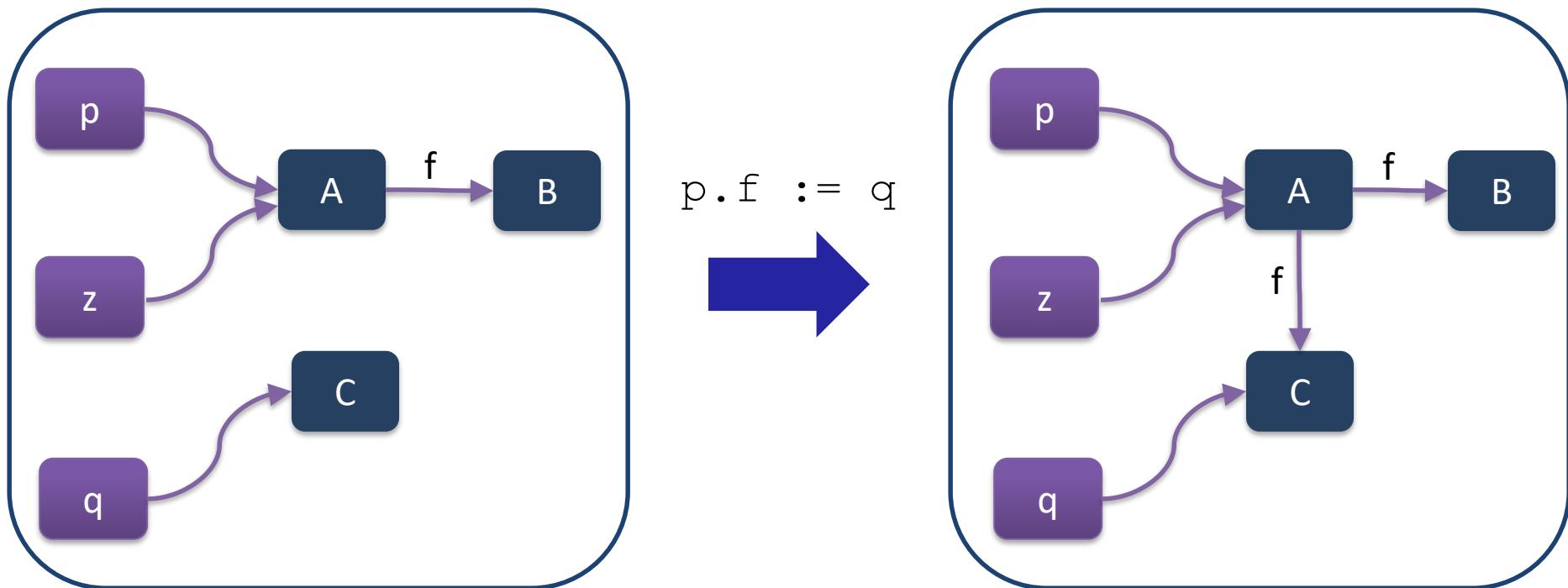


Concrete objects O_1 and O_2 allocated at site A
Concrete objects O_3 and O_5 allocated at site B
Concrete object O_4 allocated at site C

```
for (i = 0; i < 2; i++) {  
    // allocate  $O_1$ ,  $O_2$   
    A: x := alloc;  
    if (i == 0)  
        p := x;  
    else  
        z := x;  
}  
for (i = 0; i < 2; i++) {  
    // allocate  $O_3$ ,  $O_5$   
    B: x := alloc;  
    if (i == 0)  
        z.f := x;  
    else  
        p.f := x;  
}  
// allocate  $O_4$   
C: q := alloc;  
x := null;
```

What about $p.f := q$?

A **correct solution** is to apply union on the contents of $A.f$ and q , thereby obtaining that $A.f \mapsto \{B, C\}$. This is called **weak updates**. There are techniques to perform strong updates, but we will not study them in this course.



Notes on pointer analysis

The pointer analysis simply applies the transformers of the pointer manipulating statements from slide 19 on the **control-flow graph**. The function is the same shape as the one in the Interval domain, except applied to pointer relevant statements:

$$F^{\text{pointer}}: (\text{Lab} \rightarrow A) \rightarrow (\text{Lab} \rightarrow A)$$

Here, $\text{Lab} \rightarrow A$ denotes the pointer analysis domain from slide 14.

$$F^{\text{pointer}}(m)^\ell = \begin{cases} T \quad (\text{here, } T = \text{AbsObj}) & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell', \text{action}, \ell)} [\![\text{action}]\!](m(\ell')) & \text{otherwise} \end{cases}$$

Example

```
p := alloc1; // A1
q := alloc2; // A2
if p=q3 then
    z:=p4
else
    z:=q5
```

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Result of Pointer Analysis

p := alloc¹; // A1

q := alloc²; // A2

if p=q³ then

z:=p⁴

else

z:=q⁵

p ↦ {A1, A2, null},
q ↦ {A1, A2, null},
z ↦ {A1, A2, null}

p ↦ {A1}, q ↦ {A1, A2, null},
, z ↦ {A1, A2, null}

p ↦ {A1}, q ↦ {A2},
z ↦ {A1, A2, null}

p ↦ ∅, q ↦ ∅, z ↦ {A1, A2, null}

p ↦ ∅, q ↦ ∅, z ↦ ∅

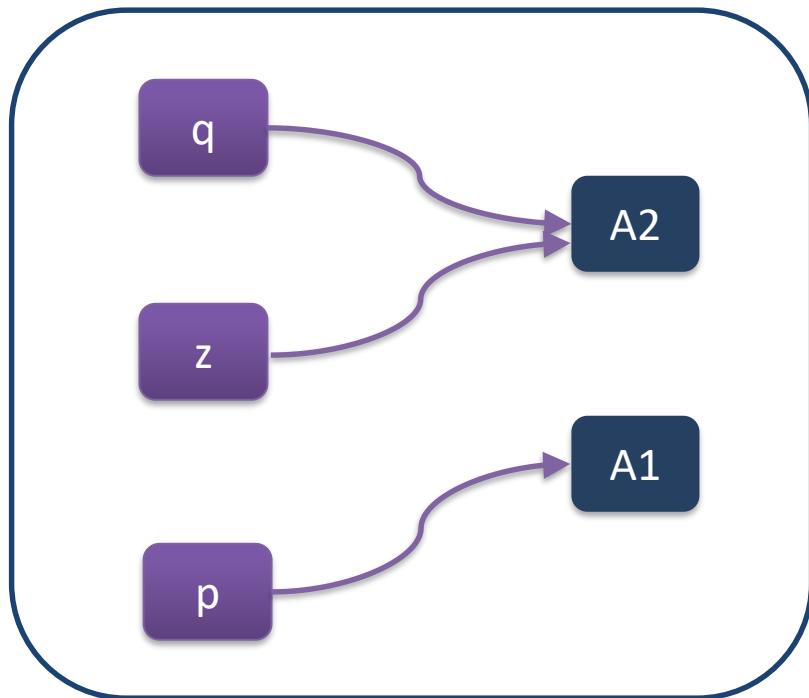
p ↦ {A1}, q ↦ {A2},
z ↦ {A1, A2, null}

p ↦ {A1}, q ↦ {A2}, z ↦ {A2}

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Flow-Sensitive: Output

Showing results **at the end of the program:**



3 points-to pairs

z and p do not alias
z and q alias

Pointer Analysis: two kinds

- Lets now take a look at the **flow insensitive** analysis.
 - Scalable points-to analysis is typically **flow-insensitive**
- Soot implements a few **flow-insensitive** analyses

Flow Insensitive Abstract Domain

$$\begin{aligned} (\text{PtrVar} \rightarrow \wp(\text{AbsObj})) \times \\ (\text{AbsObj} \times \text{Field} \rightarrow \wp(\text{AbsObj})) \end{aligned}$$

This abstract domain does not keep information per label, essentially **ignoring the control flow** of the program.

Flow-Insensitive Analysis

```
p := alloc1; // A1
q := alloc2; // A2
if p=q3 then
    z:=p4
else
    z:=q5
```

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Flow-Insensitive Analysis

p := alloc¹; // A1

q := alloc²; // A2

z := p⁴

z := q⁵

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Flow-Insensitive Analysis

```
p := alloc1; // A1
```

```
q := alloc2; // A2
```

```
z := p4
```

```
z := q5
```

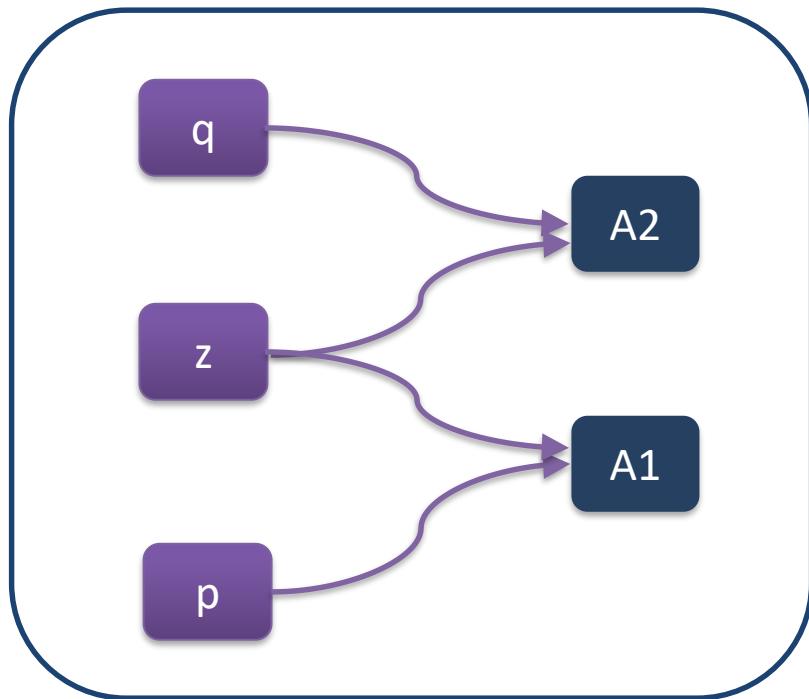
Output of Analysis:

```
p ↳ {A1}, q ↳ {A2}, z ↳ {A1, A2}
```

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)

Flow-Insensitive Output

At any program point we have:



4 points-to pairs

z and q alias

z and p alias

Alias Analysis

(this is a particular client of the pointer analysis)

- Once we have performed the pointer analysis, it is trivial to compute alias analysis
 - but not vice versa
- A function **points-to (p)** returns the set of all abstract objects that a pointer **p** can point to
 - Practically, frameworks like Soot contain similar call to points-to, where one can obtain the abstract objects a pointer points to.
- Two pointers p and q **may alias** if:
 - $\text{points-to (a)} \cap \text{points-to(b)} \neq \emptyset$

Static Analysis

In our study of static analysis, we have studied and seen how to work with both **numerical domains** as well as heap domains (like **pointer analysis**). Both of these are popular when designing real world analyzers.

This concludes our study of static analysis and over-approximation.

Next time we will look at a concrete application of using numerical and pointer analysis to make HPC/GPU programs more reliable and safe.

Rigorous Software Engineering

Applications of Analysis: Heap and Pointers

Prof. Martin Vechev

Pointer & Alias Analysis

Pointer and Alias Analysis is **fundamental** to reasoning about heap manipulating programs (pretty much all programs today). Virtually all practical static analysis tools (bug finding, verification, etc...) contain some form of pointer analysis.

Due to its importance, the topic has received much attention from the research and developer communities. In our lecture today, we will study the **core concepts** of such pointer analyses and illustrate them on examples. This will enable us to use them (like in the course project) or to build/extend such analyzers.

Let us define the concrete store

- Objs : set of all possible objects
- $\text{PtrVal} = \text{Objs} \cup \{\text{null}\}$
- $\rho \in \text{PrimEnv} : \text{Var} \rightarrow \mathbb{Z}$
- $r \in \text{PtrEnv} : \text{PtrVar} \rightarrow \text{PtrVal}$
- $h \in \text{Heap} : \text{Objs} \rightarrow (\text{Field} \rightarrow \{\text{PtrVal} \cup \mathbb{Z}\})$

A store is now: $\sigma = \langle \rho, r, h \rangle \in \text{Store} = \text{PrimEnv} \times \text{PtrEnv} \times \text{Heap}$
(before the store was only ρ)

As before $\Sigma = \text{Lab} \times \text{Store}$

Some Common Terms

- Aliases
 - Two pointers p and q are aliases if they point to the same object
- Points-to pair
 - (p, A) means p holds the address of object A
- Points-to pairs and aliases
 - if (p, A) and (r, A) then p and r are **aliases**

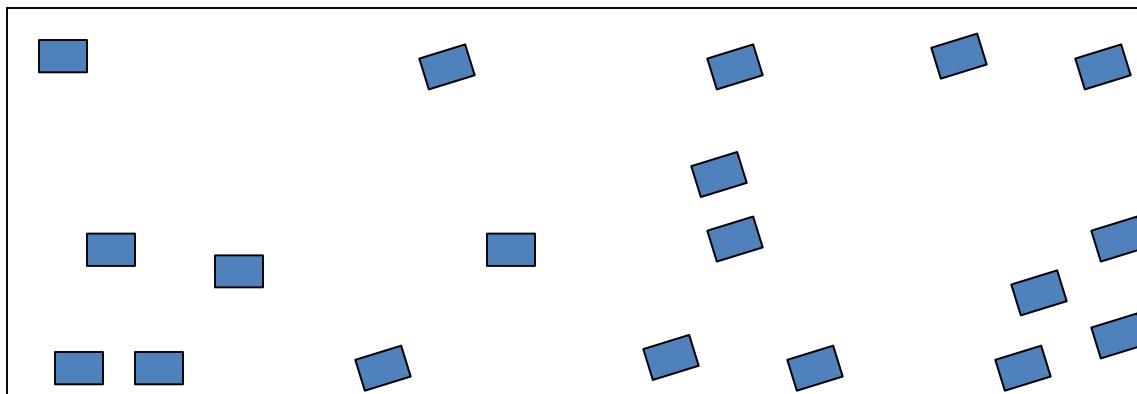
(May) Points-to Analysis

What to do with allocation of new objects? A program can create **an unbounded number** of objects.

We need to again use **abstraction**. That is, we need some **static naming scheme** for dynamically allocated objects

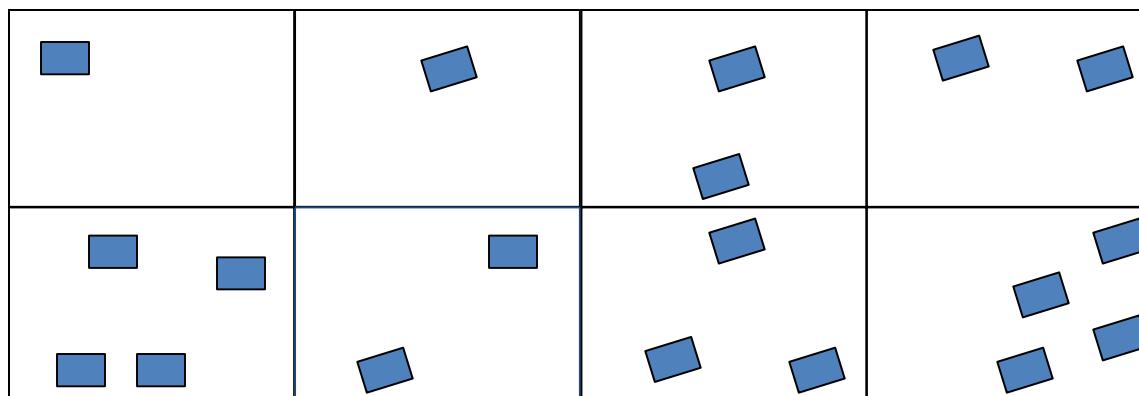
Abstraction: Allocation Sites

- Divide heap into a **fixed partition** based on **allocation site** (the statement label)
- All objects **allocated at the same program point (label)** get represented by a single “abstract object”



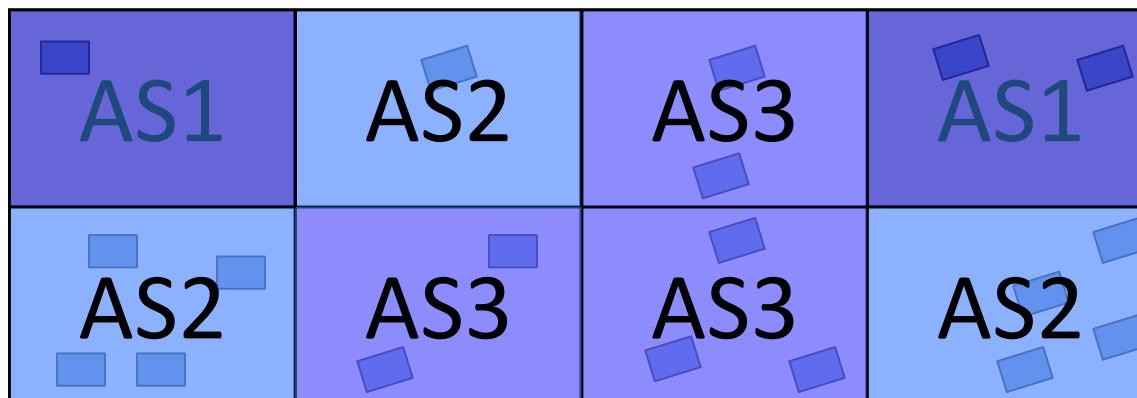
Abstraction: Allocation Sites

- Divide heap into a **fixed partition** based on **allocation site** (the statement label)
- All objects **allocated at the same program point (label)** get represented by a single “abstract object”



Abstraction: Allocation Sites

- Divide heap into a **fixed partition** based on **allocation site** (the statement label)
- All objects **allocated at the same program point (label)** get represented by a single “abstract object”



Abstract Objects

The (static) abstract objects can be just the allocation sites (labels of statements in our simple language) of the program. If this is too imprecise, we can also use the calling context. This is for instance common in library frameworks where the allocation site inside the library is not useful as we need to know where the library was called from. Naturally, bigger calling context will lead to a greater number of abstract objects.

If we use **allocation sites** (labels), we can now define the abstract objects as

$$\text{AbsObj} = \{\ell \mid \text{statement is } p := \text{alloc}^\ell\}$$

That is, this is just those labels/program counters in the program where allocation of an object occurs. Here alloc^ℓ is just the name of the allocation instruction (there can be other names, e.g., `newobject`, etc).

Pointer Analysis: two kinds

- **Flow sensitive:** respects the program control flow
 - a separate set of points-to pairs for every program point
 - the set at a program point represents possible may-aliases on some path from entry to the program point
- **Flow insensitive:** assume **all** execution orders are possible, abstracts away order between statements
 - good for concurrency (if not too imprecise)

Let us first take a look at the **flow sensitive analysis**

Abstract Interpretation: step-by-step

1. select/define an abstract domain
 - selected based on the type of **properties** you want to prove
2. define abstract semantics **for the language** w.r.t. to the domain
 - prove **sound** w.r.t **concrete semantics**
 - involves defining abstract transformers
 - that is, effect of statement / expression on the abstract domain
3. iterate abstract transformers over the abstract domain
 - until we reach a **fixed point**

The **fixed point** is the **over-approximation** of the program

Step 1: Define Domain

The abstract domain is a **complete lattice**:

$$\text{Labs} \rightarrow ((\text{PtrVar} \rightarrow \wp(\text{AbsObj})) \times (\text{AbsObj} \times \text{Field} \rightarrow \wp(\text{AbsObj})))$$

That is, the abstract domain keeps two maps at every program label. The first map contains a mapping from a pointer variable to a set of abstract objects. The second map contains a mapping from the fields of abstract objects to the set of abstract objects they point to.

Note that this lattice is of **finite height**. We have a finite number of abstract objects (i.e. `AbsObj`), finite number of field names (i.e. `Field`), and a finite number of pointer variables (i.e. `PtrVar`), and labels (i.e. `Lab`). Therefore, **we will not need widening here**.

Step 1: Define Domain

The abstract domain is a **complete lattice**:

$$\text{Labs} \rightarrow (\text{PtrVar} \rightarrow \wp(\text{AbsObj})) \times (\text{AbsObj} \times \text{Field} \rightarrow \wp(\text{AbsObj}))$$

Example of an element in the domain:

```
1 → ( p → {a5, a10}, a5.f → {a6, a9} )  
...  
43 → ...
```

We read this as follows: at program label 1, pointer p points to 2 abstract objects a₅ and a₁₀. Field f of abstract object a₅ points to two abstract objects a₆ and a₉. In this element, we have other program labels (43 of them), where there are many such pointer maps, but we did not write them explicitly here.

Step 1: Define Domain

The abstract domain is a **complete lattice**:

$$\text{Labs} \rightarrow (\text{PtrVar} \rightarrow \wp(\text{AbsObj})) \times (\text{AbsObj} \times \text{Field} \rightarrow \wp(\text{AbsObj}))$$

What are \sqsubseteq , \sqcup , \sqcap , \perp , \top ?

Example: $1 \rightarrow (p \rightarrow \{a_5, a_{10}\}, a_5.f \rightarrow \{a_6, a_9\})$

\sqsubseteq

$1 \rightarrow (p \rightarrow \{a_5, a_{10}, a_{15}\}, a_5.f \rightarrow \{a_6, a_9, a_{52}\})$

Essentially, everything is based on \sqsubseteq , \sqcup , \sqcap , lifted appropriately.
It is a **good exercise** to define them formally.

Step 2: Define Abstraction

$$\alpha: \wp(\Sigma) \rightarrow (\text{Labs} \rightarrow ((\text{PtrVar} \rightarrow \wp(\text{AbsObj})) \times (\text{AbsObj} \times \text{Field} \rightarrow \wp(\text{AbsObj}))))$$
$$\gamma: (\text{Labs} \rightarrow ((\text{PtrVar} \rightarrow \wp(\text{AbsObj})) \times (\text{AbsObj} \times \text{Field} \rightarrow \wp(\text{AbsObj})))) \rightarrow \wp(\Sigma)$$

Using α , we abstract a **set of states** into the two kinds of maps.
Similarly, using γ , we concretize the pointer maps to a set of **states**.

The formal definition of α and γ is left as an exercise.

Let us consider an example to give an intuition.

Example of Abstraction

```
α ( { ⟨ 5, _ , { p ↦ o1, q ↦ o2 } , { o1.k ↦ o3, o2.v ↦ o6 } ⟩ ,  
⟨ 5, _ , { p ↦ o2, q ↦ o3 } , { o1.k ↦ o3, o2.v ↦ o3 } ⟩  
} )
```

Here, by `_` we mean that the program has no integer variables.

Suppose that: object o_1 is allocated at site a_3 (program label 3)
object o_2 is allocated at site a_4 (program label 4)
object o_3 is allocated at site a_9 (program label 9)
object o_6 is allocated at site a_{31} (program label 31)

What is the result ?

Rigorous Software Engineering

Modelling and Specification

Prof. Martin Vechev

- Logic
 - Static Models
 - Dynamic Models
 - Analyzing Models
-

Signatures

- A signature declares a set of atoms
 - Think of signatures as classes
 - Think of atoms as immutable objects
 - Different signatures declare disjoint sets
- Extends-clauses declare subsets relations
 - File and Dir are disjoint subsets of FSObject

```
sig FSObject { }
```

```
sig File extends FSObject { }
sig Dir extends FSObject { }
```

Operations on Sets

■ Standard set operators

- + (union)
- & (intersection)
- - (difference)
- **in** (subset)
- = (equality)
- # (cardinality)
- **none** (empty set)
- **univ** (universal set)

■ Comprehensions

```
sig File extends FSOObject { }  
sig Dir extends FSOObject { }
```

```
#{ f: FSOObject | f in File + Dir } >= #Dir
```

More on Signatures

- Signature can be abstract

- Like abstract classes
- **Closed world assumption:** the declared set contains exactly the elements of the declared subsets

```
abstract sig FSObject {}  
sig File extends FSObject {}  
sig Dir extends FSObject {}
```

```
FSObject = File + Dir
```

- Signatures may constrain the cardinalities of the declared sets

- **one:** singleton set
- **lone:** singleton or empty set
- **some:** non-empty set

```
one sig Root extends Dir {}
```

Fields

- A field declares a relation on atoms
 - f is a binary relation with domain A and range given by expression e
 - Think of fields as associations
- Range expressions may denote multiplicities
 - **one**: singleton set (default)
 - **lone**: singleton or empty set
 - **some**: non-empty set
 - **set**: any set

```
sig A {
  f: e
}
```

```
abstract sig FSObject {
  parent: lone Dir
}
```

```
sig Dir extends FSObject {
  contents: set FSObject
}
```

Operations on Relations

- Standard operators

- \rightarrow (cross product)
 - \cdot (relational join)
 - \sim (transposition)
 - $^\wedge$ (transitive closure)
 - $*$ (reflexive, transitive closure)
 - $<:$ (domain restriction)
 - $>:$ (range restriction)
 - $++$ (override)
 - **iden** (identity relation)
 - $[]$ (box join: $e1[e2] = e2.e1$)
precedence: $e0.e1[e2] = e2.(e0.e1)$
- only for binary relations

```
abstract sig FSObject {
    parent: lone Dir
}
```

```
sig Dir extends FSObject {
    contents: set FSObject
}
```

```
one sig Root extends Dir { }
```

```
FSObject in Root.*contents
```

All file system objects
are contained in the root
directory

Relational Join: Example

- Consider a structure with four FSObject atoms

- r: Root, d1, d2: Dir, f: File

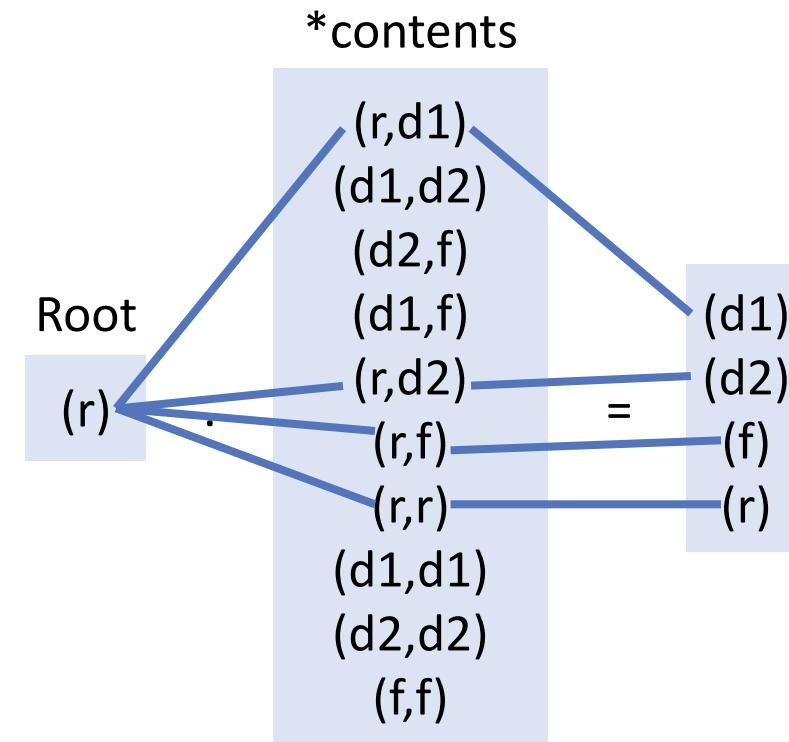
and contents relation

```
(r,d1) (d1,d2) (d2,f)
```

- The reflexive, transitive closure *contents is

```
(r,d1) (d1,d2) (d2,f)
(d1,f) (r,d2) (r,f)
(r,r) (d1,d1) (d2,d2) (f,f)
```

- The relational join Root.*contents is



FSObject in Root.*contents

More on Fields

- Fields may range over relations
- Relation declarations may include multiplicities on both sides
 - **one, lone, some, set** (default)

```
sig University {  
    enrollment: Student set -> one Program  
}
```

- Range expressions may depend on other fields

```
sig University {  
    students: set Student,  
    enrollment: students set -> one Program  
}
```

Constraints

- Boolean operators

- `!` or **not** (negation)
- `&&` or **and** (conjunction)
- `||` or **or** (disjunction)
- `=>` or **implies** (implication)
- **else** (alternative)
- `<=>` or **iff** (equivalence)

- Four equivalent constraints

`F => G else H`

`F implies G else H`

`(F && G) || ((!F) && H)`

`(F and G) or ((not F) and H)`

- Quantified expressions

- **some** e
e has at least one tuple
- **no** e
e has no tuples
- **lone** e
e has at most one tuple
- **one** e
e has exactly one tuple

`no Root.parent`

Quantification

- Quantifiers may have the following forms
 - **all** $x: e \mid F$
 - **all** $x: e_1, y: e_2 \mid F$
 - **all** $x, y: e \mid F$
 - **all disj** $x, y: e \mid F$
- Alloy supports five different quantifiers
 - **all** $x: e \mid F$
F holds for every x in e
 - **some** $x: e \mid F$
F holds for at least one x in e
 - **no** $x: e \mid F$
F holds for no x in e
 - **lone** $x: e \mid F$
F holds for at most one x in e
 - **one** $x: e \mid F$
F holds for exactly one x in e

no $d: \text{Dir} \mid d \text{ in } d.^{\wedge}\text{contents}$

contents-relation is acyclic

Predicates and Functions

- Predicates are named, parameterized formulas

```
pred p[ x1: e1, ..., xn: en ] { F }
```

```
pred isLeaf[ f: FSObject ] {  
    f in File || no f.contents  
}
```

- Functions are named, parameterized expressions

```
fun f[ x1: e1, ..., xn: en ]: e { E }
```

```
fun leaves[ f: FSObject ]: set FSObject {  
    { x: f.*contents | isLeaf[ x ] }  
}
```

Exploring the Model

- The Alloy Analyzer can search for structures that satisfy the constraints M in a model
 - Find instance of a predicate
 - A solution to
M &&
some x₁: e₁, ..., x_n: e_n | F
 - Find instance of a function
 - A solution to
M &&
some x₁: e₁, ..., x_n: e_n,
res: e | res = E
- `pred p[x1: e1, ..., xn: en] { F }`
- `run p`
- `fun f[x1: e1, ..., xn: en]: e { E }`
- `run f`

Exploring the Model: Scopes

- The existence of a structure that satisfies the constraints in a model is in general **undecidable**
- The Alloy Analyzer searches exhaustively for structures **up to a given size**
 - The problem becomes **finite** and, thus, **decidable**

```
run isLeaf
run isLeaf for 5
run isLeaf for 5 Dir, 2 File
run isLeaf for exactly 5 Dir
run isLeaf for 5 but 3 Dir
run isLeaf for 5 but exactly 3 Dir
```

Exploring the Model: Example

```
abstract sig FSObject {
    parent: lone Dir
}
```

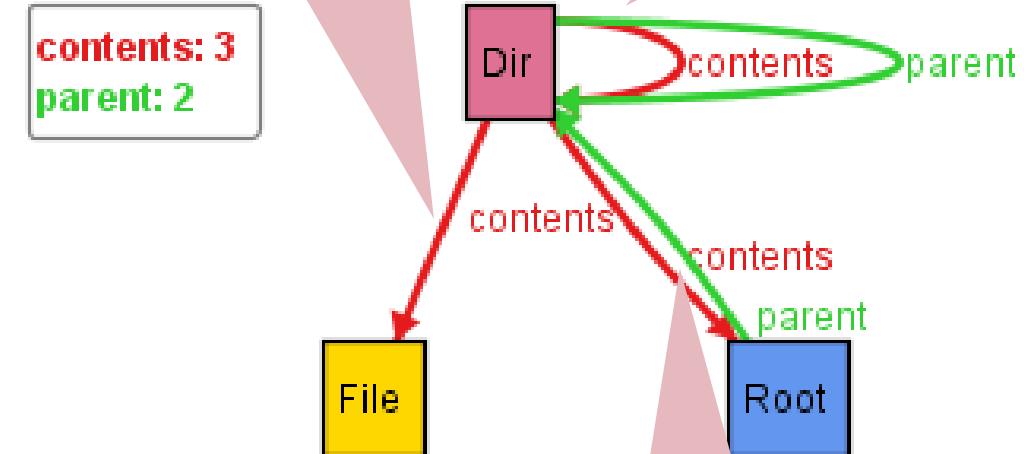
```
sig Dir extends FSObject {
    contents: set FSObject
}
```

```
one sig Root extends Dir { }
```

contents and parent
should be inverse
relations

A directory
should not
contain itself

Root should not
have a parent



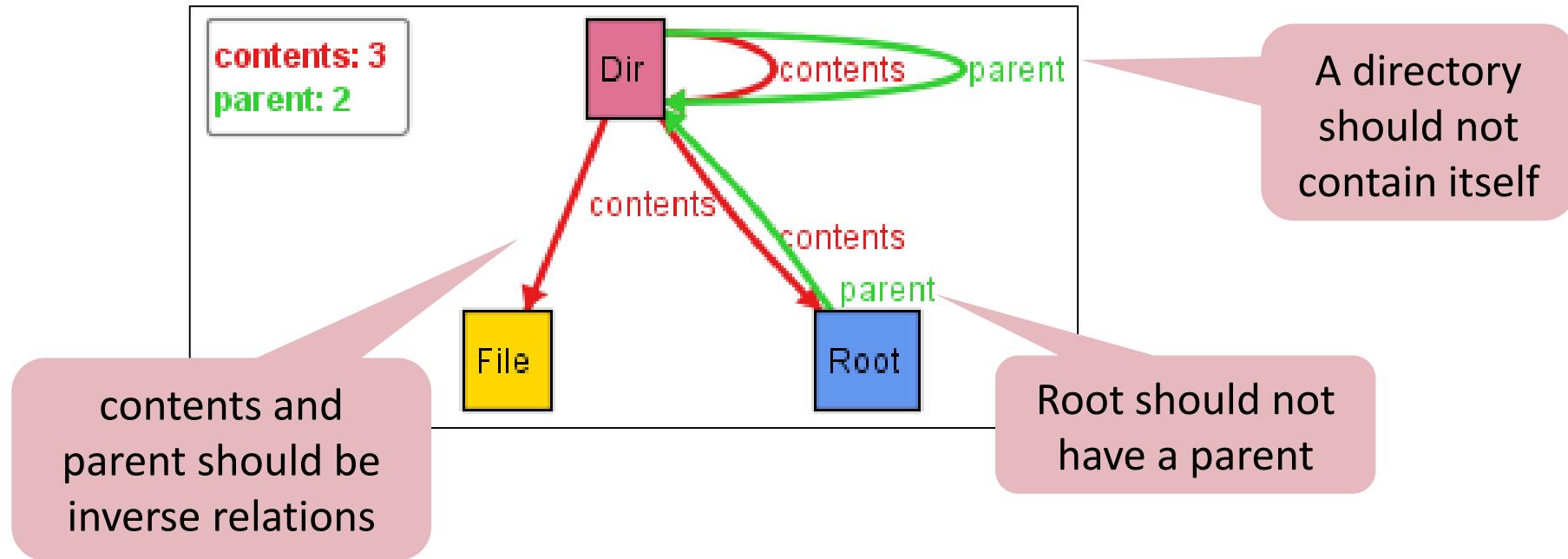
Adding Constraints

- Facts add constraints that always hold
 - **run** searches for solutions that satisfy all constraints

```
fact { F }
fact f { F }
sig S { ... } { F }
```

- Facts express value and structural invariants of the model

Adding Constraints: Example



```
fact { no Root.parent }
```

```
fact { all d: Dir, o: d.contents | o.parent = d }
```

```
fact { no d: Dir | d in d.^contents }
```

Checking the Model

- Exploring models by manually inspecting instances is cumbersome for non-trivial models
- The Alloy Analyzer can search for structures that violate a given property
 - Counterexample to an assertion
 - The search is complete for the given scope
- For a model with constraints
M, find a solution to M && !F

`assert a { F }`

`check a scope`

Checking the Model: Example

- Finding a counterexample

```
pred isLeaf[ f: FSObject ] {
    f in File || no f.contents
}
```

Root

```
assert nonEmptyRoot { !isLeaf[ Root ] }
check nonEmptyRoot for 3
```

- Proving a property

```
assert acyclic { no d: Dir | d in d.^contents }
check acyclic for 5
```

Validity is checked
(only) within the given scope

Executing "Check acyclic for 5"
 Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
 1047 vars. 63 primary vars. 1758 clauses. 50ms.
 No counterexample found. Assertion may be valid. 33ms.

Under and Over-Constrained Models

- Missing or weak facts **under-constrain** the model
 - They permit undesired structures
 - Under-constrained models are typically **easy to detect** during model exploration (using **run**) and assertion checking (using **check**)
- Unnecessary facts **over-constrain** the model
 - They exclude desired structures
- **Inconsistencies** are an extreme case of over-constraining
 - They preclude the existence of any structure
 - **All assertion checks will succeed!**

```
fact acyclic {  
    no d: Dir | d in d.*contents  
}
```

```
assert nonSense { 0 = 1 }  
  
check nonSense ✓
```

Guidelines to Avoid Over-Constraining

- Simulate model to check consistency
 - Use **run** to ensure that structures exist
 - Create predicates with desired configurations and use **run** to ensure they exist

```
fact acyclic { no d: Dir | d in d.*contents }
```

```
pred show { }  
run show
```

Executing "Run show"
Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
0 vars. 0 primary vars. 0 clauses. 3ms.
No instance found. Predicate may be inconsistent. 0ms.

- Prefer assertions over facts
 - When in doubt, check whether current model already ensures a desired property before adding it as a fact

Reference Counting List: Alloy Model (1)

Encode generic type parameter

A fact guaranteed by the Java semantics

elems is not shared (inv2)

open util/boolean

sig E { }

sig Array {

length: Int,

data: { i: Int | $0 \leq i \&& i < \text{length}$ } -> **lone E**

}

{ $0 \leq \text{length}$ }

sig ListRep {

elems: Array,

shared: Bool

}

Use library mode for booleans

Introduce array signature to model potential sharing

Array elements may be null

elems is non-null (inv1)

fact inv2 { all disj lr1, lr2: ListRep | lr1.elems != lr2.elems }

Reference Counting List: Alloy Model (2)

```
sig List {  
    rep: ListRep,  
    len: Int  
}
```

rep is non-null
(inv6)

shared conservatively
tracks sharing (inv4)

```
fact inv4 { all lr: ListRep | isFalse[ lr.shared ] => lone l: List | l.rep = lr }
```

```
fact inv7 { all l: List | 0 <= l.len && l.len <= l.rep.elems.length }
```

len is between zero and
array size (inv7)

Example: Underspecification

```
class University {  
    Set<Student> students;  
    ...  
}
```

```
class Student {  
    Program major;  
    ...  
}
```

```
sig Student {}  
sig Program {}  
sig University {}  
sig State {  
    enrollment: University -> Student -> one Program  
}
```

```
class University {  
    Map<Student, Program> enrollment;  
    ...  
}
```

```
class Student {  
    ...  
}
```

- The Alloy model leaves the choice of data structure unspecified