Analysis of Software Artifacts

Somesh Jha

Lecture 15

Structural testing of concurrent programs

João Pedro Sousa

We Have been looking at

- Static analysis of code
  - Tease out the structure
  - Code optimization
- Structural test
  - Exercising the system with a specific test suite
    - **What can we get from structural test?**
    - **How does it work for concurrent systems?**

Why structural testing? (1)

Structural testing **does not** give you:

- How well the **most used** functionality is tested
- How well the **most critical** functionality is tested

Structural testing:

- Assumes all the code is equally important

Why structural testing? (2)

With structural testing you get to know:

- How much (and which) of the code is being exercised by a given test suite

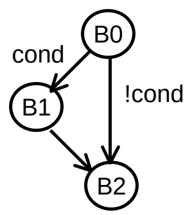
  When is that useful to know?
  - not sure about the usage patterns
  - sure about usage patterns but you want to know how much of the code is supporting them…

Computational paths

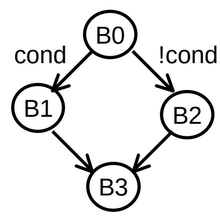
Typical patterns (1)

In order to measure how much code is exercised by a given test,
we first build a (test independent) graph view of the code to be tested

```
B0
if (cond) {
    B1
}
B2
```




```
B0
if (cond) {
    B1
} else {
    B2
}
B3
```

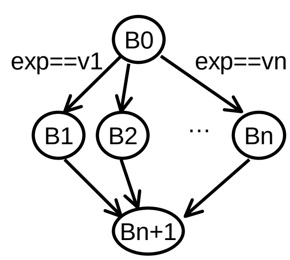


Computational paths

Typical patterns (2)

In order to measure how much code is exercised by a given test,
we first build a (test independent) graph view of the code to be tested

```
B0
case (exp) {
    v1: B1
    …
    vn: Bn
}
Bn+1
```

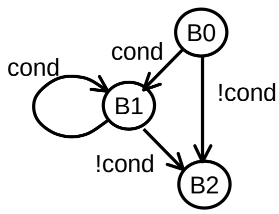


Computational paths

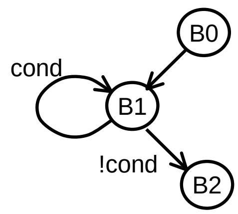
Typical patterns (3)

In order to measure how much code is exercised by a given test,
we first build a (test independent) graph view of the code to be tested

B0
while (cond) {
    B1
}
B2



B0
repeat {
    B1
} **until** (cond)
B2



How many computational paths? (1)

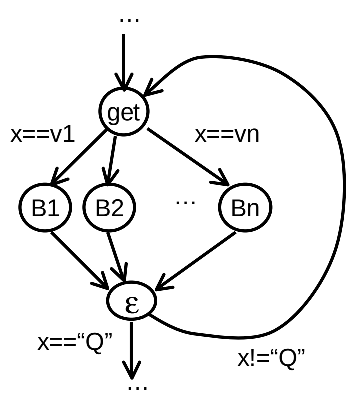
How many test cases do we need to write to check all the combinations?

Example: text editor

open file; type; replace; close file;	$\neq$	open file; replace; type; close file;	$\neq$	open file; type; close file; replace;	$\neq$	open file; replace; close file; type;

How many computational paths? (2)

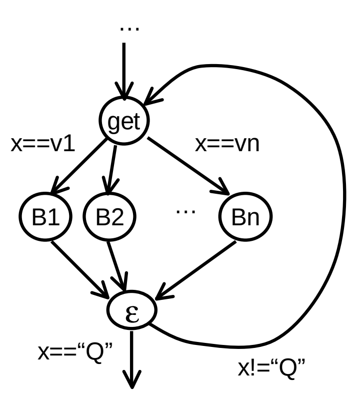
```
…
repeat {
    get (x);
    case (x) {
        v1: B1;
        …
        vn: Bn;
    }
} until (x==“Q”)
…
```



How many computational paths? (3)

Suppose: $\underbrace{\text{in1; in2;… “Q”}}_{\text{m-inputs}}$

Number of paths: n^m



Example:
- possible inputs = 10
- session length = 20

paths = 100,000,000,000,000,000,000

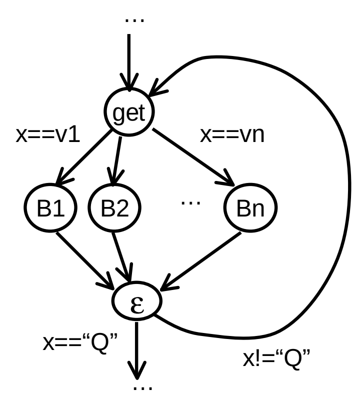
How many computational paths? (4)

Impracticable to test functionality taking into account all possible states of the software

How many code paths?

structural testing

Example:
possible inputs = 10
session length = 20
code paths = 10



How many paths?

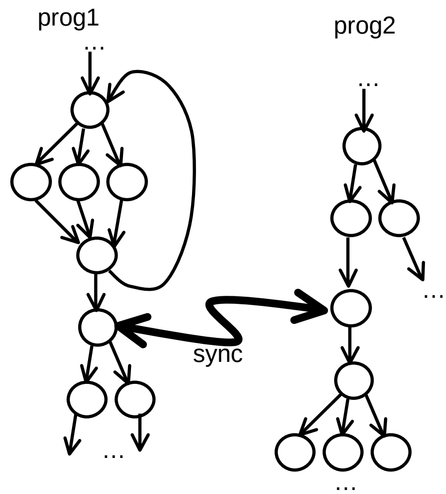
For all but trivial programs even the number of code paths can be impracticably large.

Hence the definitions of:

- All du-paths
- All edges
- All nodes…

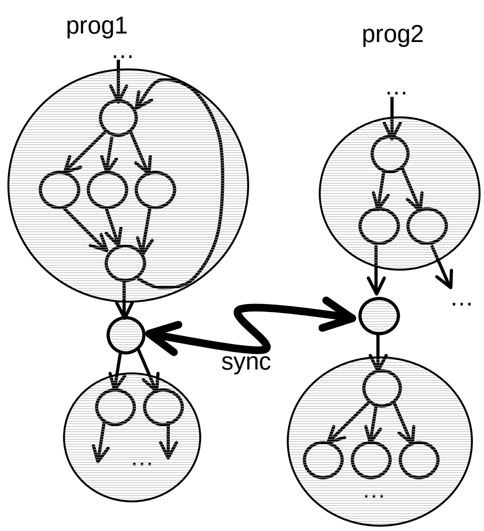
What happens when concurrency comes about?

It is impracticable to
take the product
(all possible combinations)
of the graphs for each of
the concurrent programs

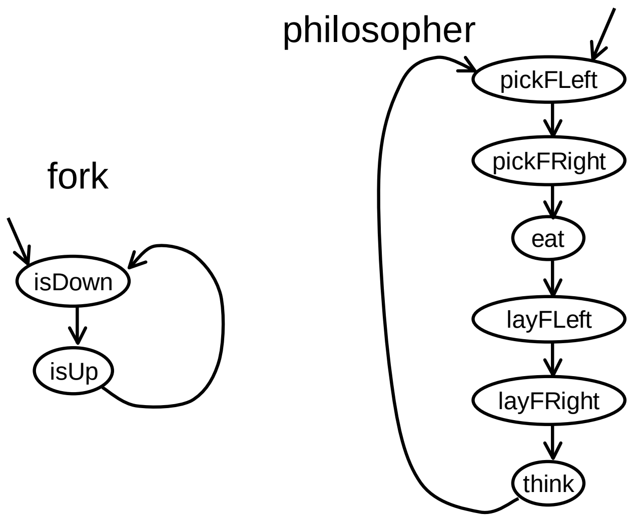


Concurrency states

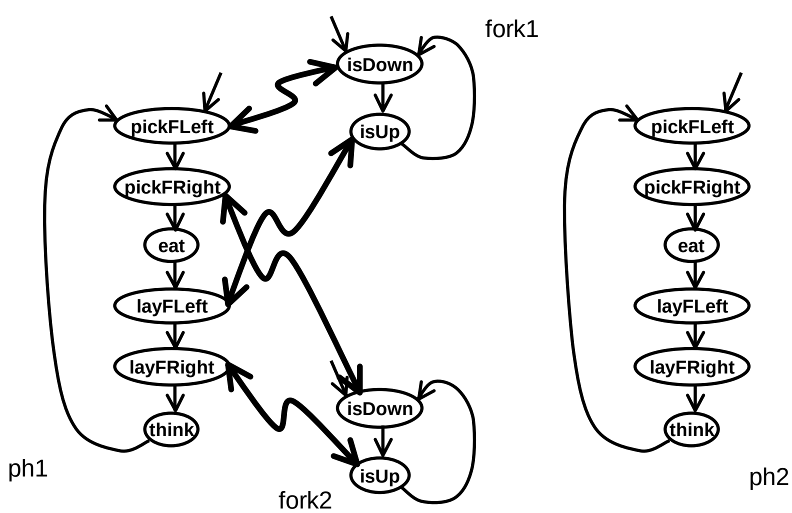
We abstract the internals
of each program,
focusing on
concurrency states



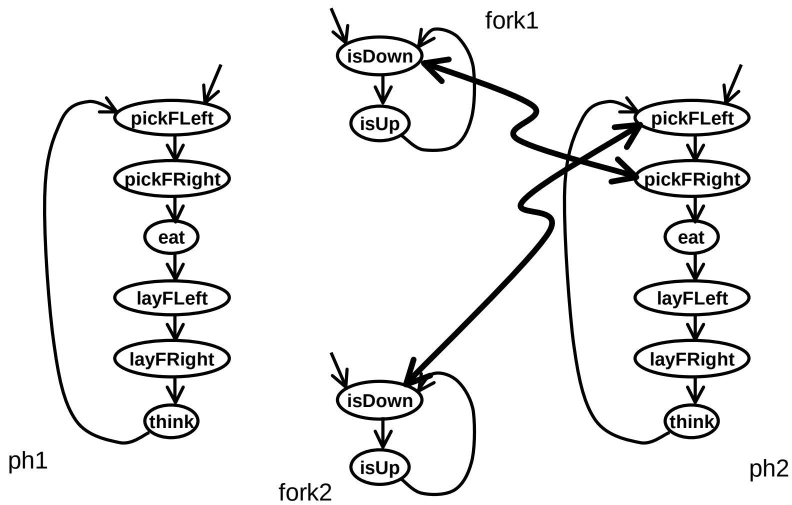
Example: dining philosophers



Simple configuration
for the dining philosophers



Simple configuration
for the dining philosophers

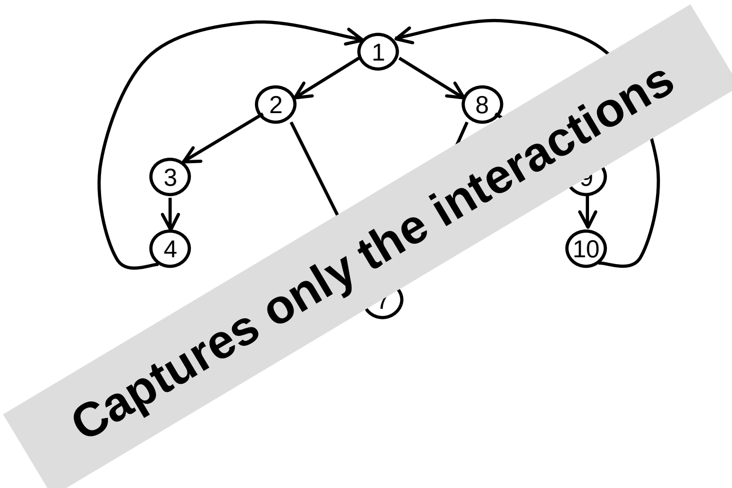


Concurrency states
for the dining philosophers (1)

	fork1	fork2	ph1	ph2
1	isDown	isDown	pickF1	pickF2
2	isUp	isDown	pickF2	pickF2
3	isUp	isUP	layF1	pickF2
4	isDown	isUp	layF2	pickF2
8	isDown	isUp	pickF1	pickF1
9	isUp	isUp	pickF1	layF2
10	isUp	isDown	pickF1	layF1
7	isUp	isUp	pickF2	pickF1

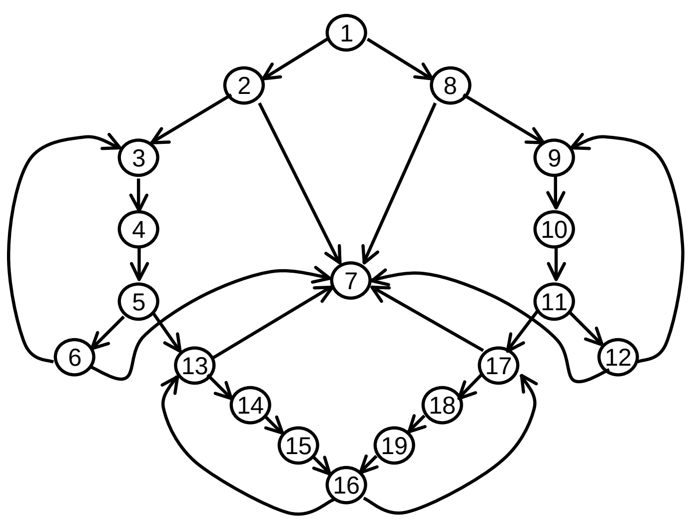
Concurrency states

for the dining philosophers (2)



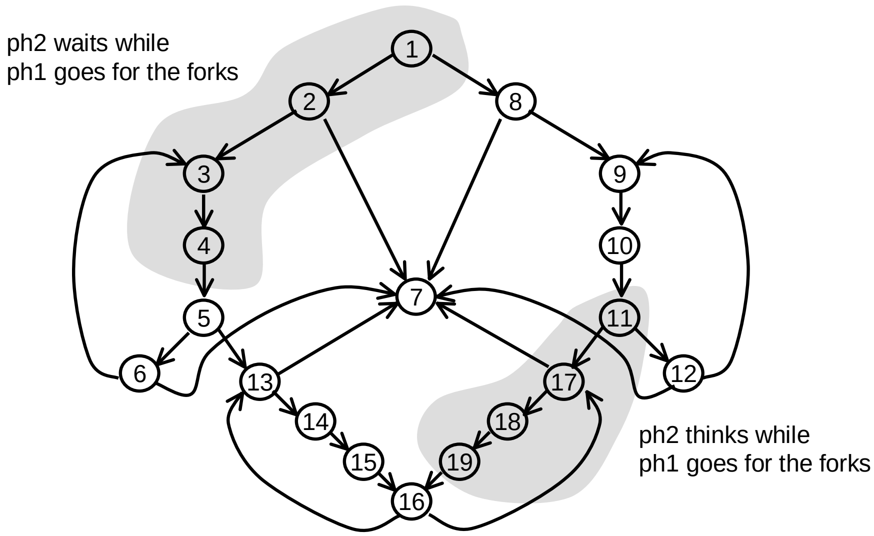
Capturing all the concurrency states

of the dining philosophers (1)



Capturing all the concurrency states

of the dining philosophers (2)



Definitions

for concurrency graphs

- **Concurrency history**
  - **is one sequence of concurrency states that can be formed by walking through a path in the concurrency graph**
- **Proper concurrency history**
  - **is one finite concurrency history where all states are distinct except, possibly, the last state in the sequence**

Coverage metrics
for concurrent programs (1)

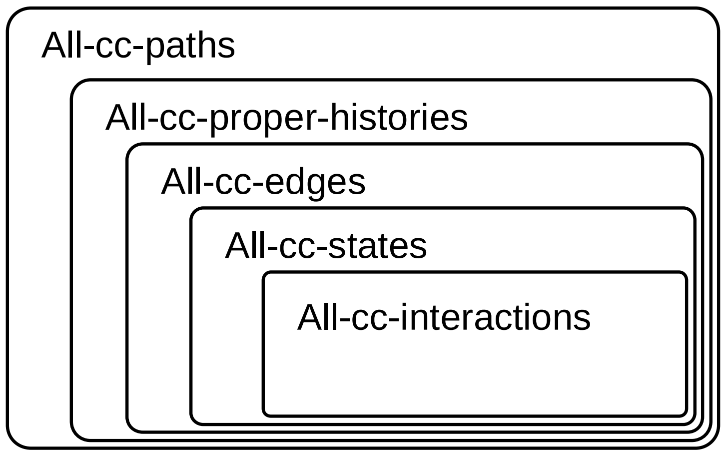
A program S and a set P of paths covered by a given test suite can be judged against the following **criteria**:

- **All-cc-paths (S,P)**
  - P is the set of all paths through the concurrency graph
- **All-cc-proper-histories (S,P)**
  - P contains the set of all finite proper concurrency histories

Coverage metrics
for concurrent programs (2)

- **All-cc-edges (S,P)**
  - for every edge E in the concurrency graph, there is at least one path in P that contains E
- **All-cc-states (S,P)**
  - for every concurrency state C in the concurrency graph, there is at least one path in P where C occurs
- **All-cc-interactions (S,P)**
  - for every concurrency state C that involves an interaction between tasks, there is at least one path in P where C occurs

Coverage metrics
subsumption hierarchy



Coverage metrics
in practice

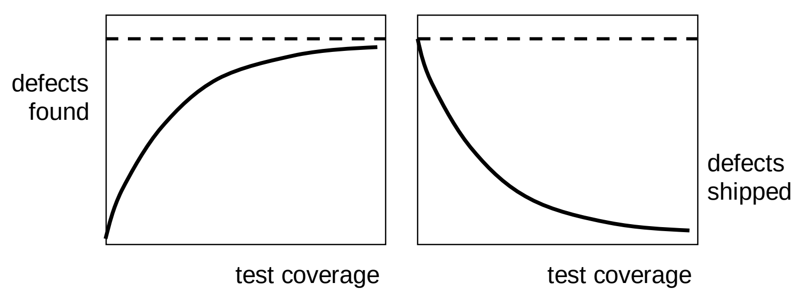
- **As hinted by the complete concurrency graph for the simple dinning philosophers example, non-trivial systems will have very large concurrency graphs.**
- **Rather than explicitly building the concurrency graph for the whole system, the number of concurrency nodes and edges can be estimated based on the graphs for the individual components.**
  - **see Structural Testing of Concurrent Programs, Richard Taylor et al, IEEE Transactions on Software Engineering, Vol. 18, No. 3, 1992.**

Coverage metrics
how to obtain them?

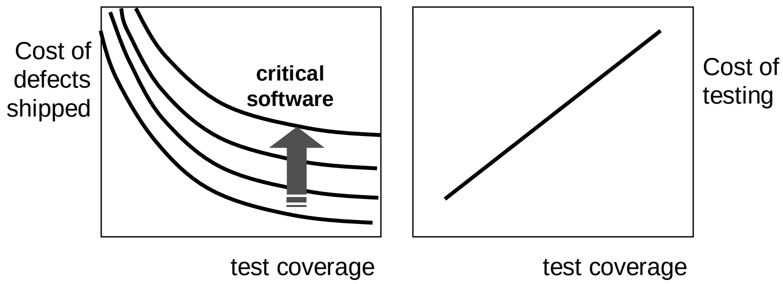
Tools give us a percentage of coverage relative to a criteria.
There are two approaches for that:

- **Apply a (automatic) code transformation, introducing instrumentation in concurrency states**
- **Modify the OS scheduler so to gather information related with concurrency states**

Cost-benefit of testing (1)



Cost-benefit of testing (2)



Cost-benefit of testing (3)