Java Path Finder (JPF)

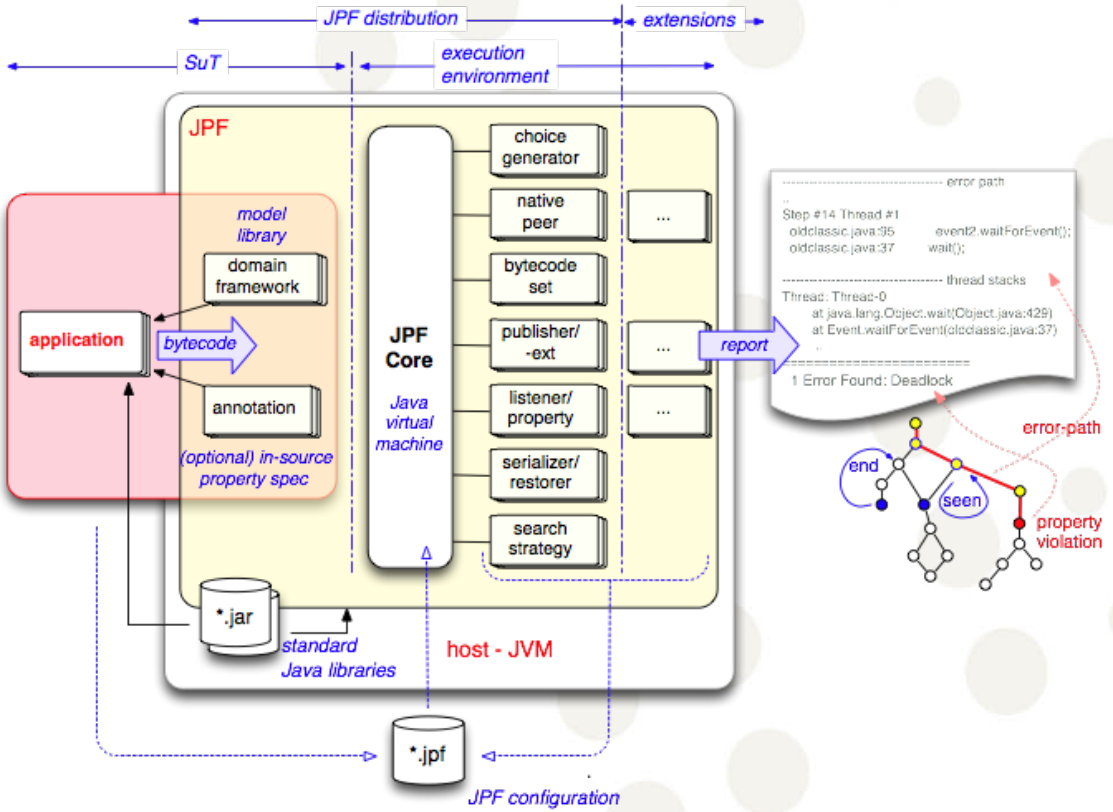
Christian Bergum Bergersen
June 1, 2015

UNIVERSITY OF OSLO

What is Java Path Finder?

- Java Path Finder is an open-source analysis system that automatically verifies/model check Java programs. Initially developed by NASA.
- **The Java code is the model for JPF**. Using a customizable Virtual Machine that supports features such as state storage, state matching and much more! Actually a VM running on top of JVM.
- Module based, the core JPF model supports checks for generic properties such as absence of unhandled exceptions, deadlocks, and race conditions.

Java Path Finder components



Example: Java code with race-condition

```
1  class Racer extends Thread {
2    static int sharedInt = 0;
3
4    public void run () {
5      System.out.printf("Thread %d started!\n", getId());
6        for(int i = 0; i < 10000; i++) {
7          sharedInt++;
8        }
9    }
10
11   public static void main (String [] a) throws Exception {
12     new Racer().start(); new Racer().start();
13     Thread.sleep(1000);
14     System.out.println("Value: " + sharedInt);
15   }
16 }
```

```
$ java Racer
Thread 8 started!
Thread 9 started!
Value: 19786

$ java Racer
Thread 8 started!
Thread 9 started!
Value: 18702
```

- Non-deterministic result due to concurrency without synchronization. **Value:** should be 20000 !
- In Java, this is easily fixed by adding a mutex (**synchronized** method or block).

- This example is small and trivial, but JPF can also be used to find race-conditions in much bigger and complex programs!

```
$ java -jar RunJPF.jar ../Racer.jpf

JavaPathfinder v7.0 - (C) RIACS/NASA Ames Research Center
========================================== system under test
Racer.main()
========================================== error 1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Racer.globalInt
  Thread-1 at Racer.run(Racer.java:7)
                "sharedInt++;"  : putstatic
  Thread-2 at Racer.run(Racer.java:7)
                "sharedInt++;"  : getstatic
```

Testing/Runtime verification vs Model Checking

- When writing and executing a test for a program, you only execute a single execution path! Almost impossible to identify and write a test for all execution paths!
- A model checker as JPF can identify all execution paths, execute them and show traces leading to errors.
- For n threads with m statements each, the number of possible scheduling sequences equals t.

$$t = \frac{(n * m)!}{m!^n}$$

The State Space Explosion Problem

Since concurrent actions can be executed in any arbitrary order, considering all possible interleaving's of concurrent actions can lead to a very large state space. It can be shown that the number of states increases exponentially with the number of threads.

In JPF this means limitations in the size of programs JPF manage to check. Often when model checking big programs in JPF you will see:

Too little memory to hold all states

java.lang.OutOfMemoryError: Java heap space

Possible Solution - The State Explosion Problem

It would be very tempting to give the JVM more/unlimited memory so it can hold all states in memory. However, thought JPF can hold all states in memory, the execution time needed to check all interleaving's between threads may take hours to days for quite small programs.

Good solution - The State Explosion Problem

- Reducing the size of the state space that needs to be checked.
- The challenge is to reduce the full state space into a subset without losing semantic.

Partial Order Reduction

A solution is to use a technique called **Partial Order Reduction (POR)** which basically groups all instructions in a thread, that do not have any effects outside the thread, into a single transition.

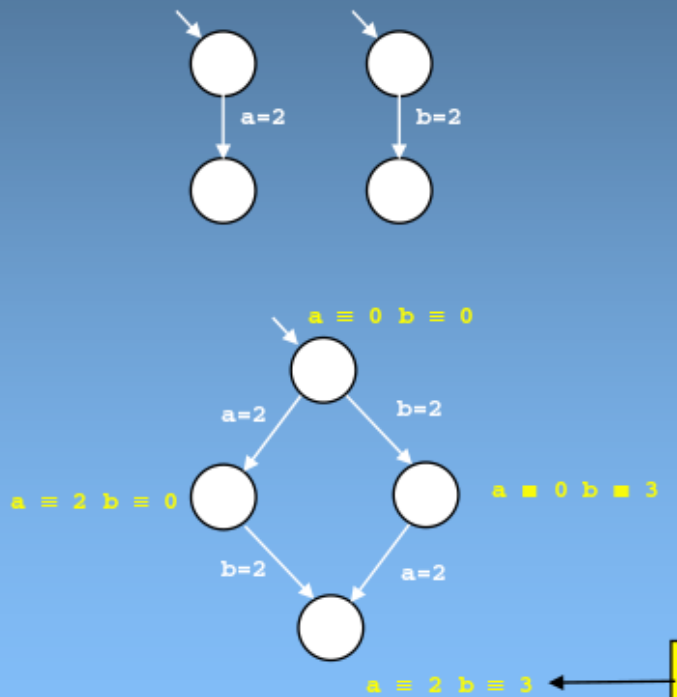
Partial Order Reduction (POR)

- JPF uses an on-the-fly partial order reduction algorithm to cut down the state space by identifying sets of concurrent actions.
- On-the-fly means that JPF under runtime executing the code inspects instructions.
- State transition is determined by the instruction type in JPF.

Partial Order Reduction (POR)

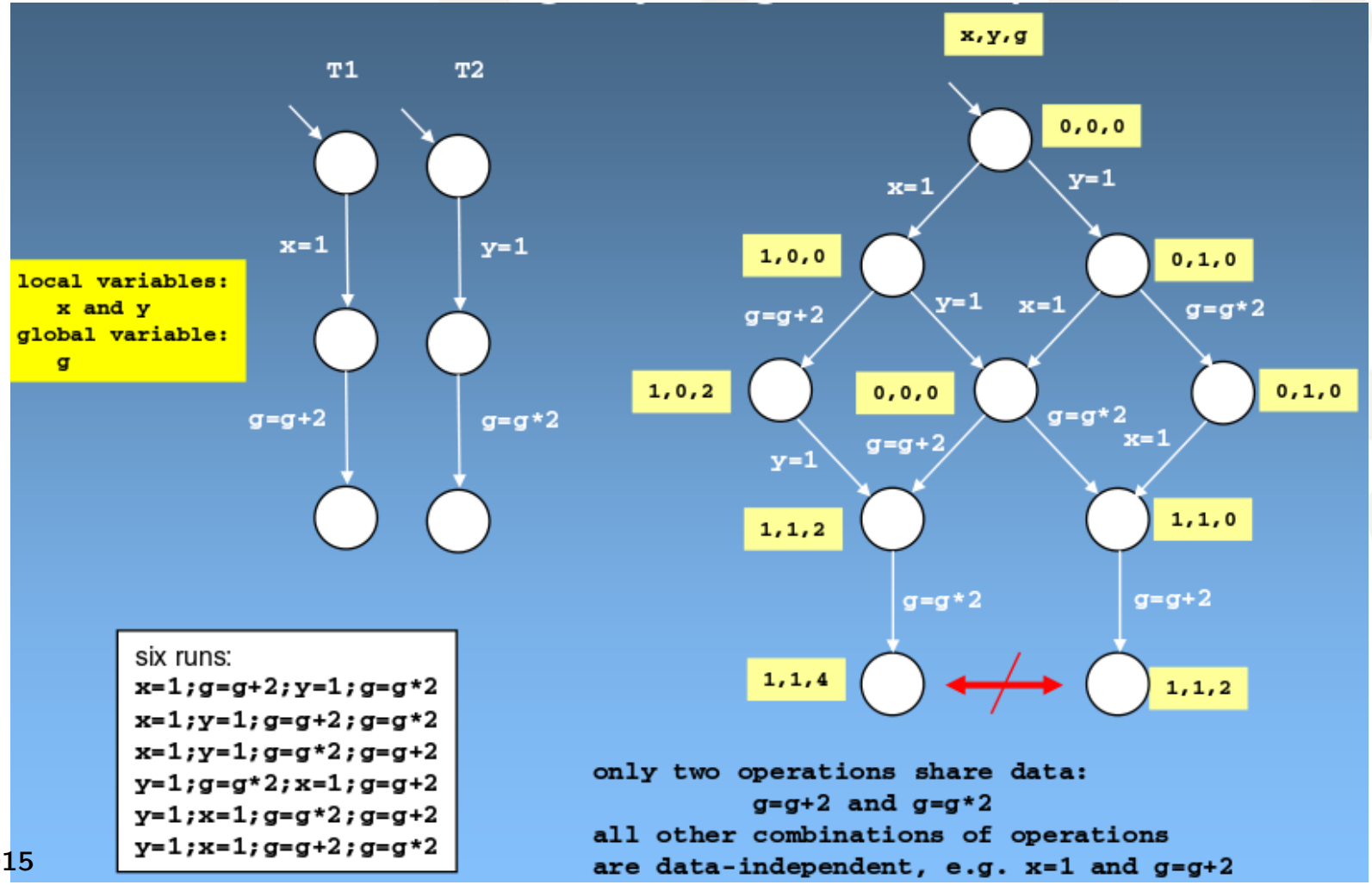
full asynchronous interleaving of process actions is sometimes redundant

```
byte a, b;

active proctype A()
{
   a = 2; 0
}

active proctype B()
{
   b = 3; 0
}
```

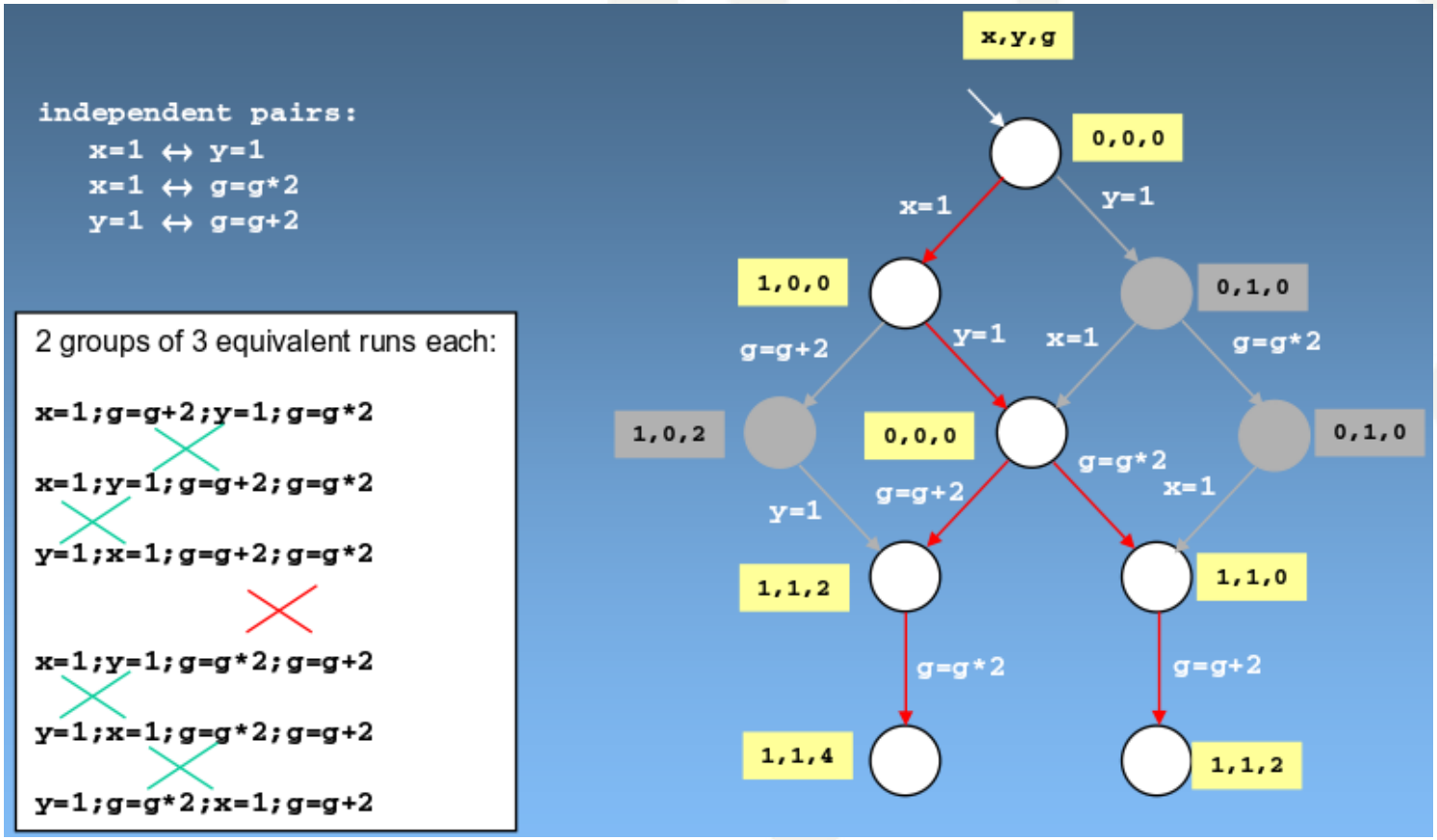


the final result is the same, no matter which path is followed

Partial Order Reduction (POR) Example



Partial Order Reduction (POR) Example



Conclusion

- We have only looked on a tiny core part of JPF, namely deadlock detection and the use of partial order reduction to collapse the state space.
- The important part is partial order reduction.
- Partial order reduction is used to construct a reduced state graph, without losing behavior.
- Reduced state graphs gives us the benefits of a reduced state space that needs to be model checked, can be applied in automatic and manual (by human) model checking.

References

- W.Visser, K.Havelund, G.Brat and S.P. "Model Checking Programs"
- Nastaran Shafiei. "Partial Order Reduction of Java Path Finder". 2010.
- Pavel Parizek. "Java Pathfinder". Slides.
- Peter C. Mehlitx. "Java Pathfinde Lecture 2: Under the hood". Slides.
- Illustrations on slide 3 and 14 is taken from Java Path finder website, visited 19.5.2015. `http://babelfish.arc.nasa.gov/trac/jpf/wiki`
- Blue slides taken from: `http://www.uio.no/studier/emner/matnat/ifi/INF5140/v09/undervisningsmateriale/10-3-Holzmann-Ch9.pdf`