Java Pathfinder
General Overview and Current State of Affairs

Pavel Parizek

School of Computer Science,
University of Waterloo

Java Pathfinder

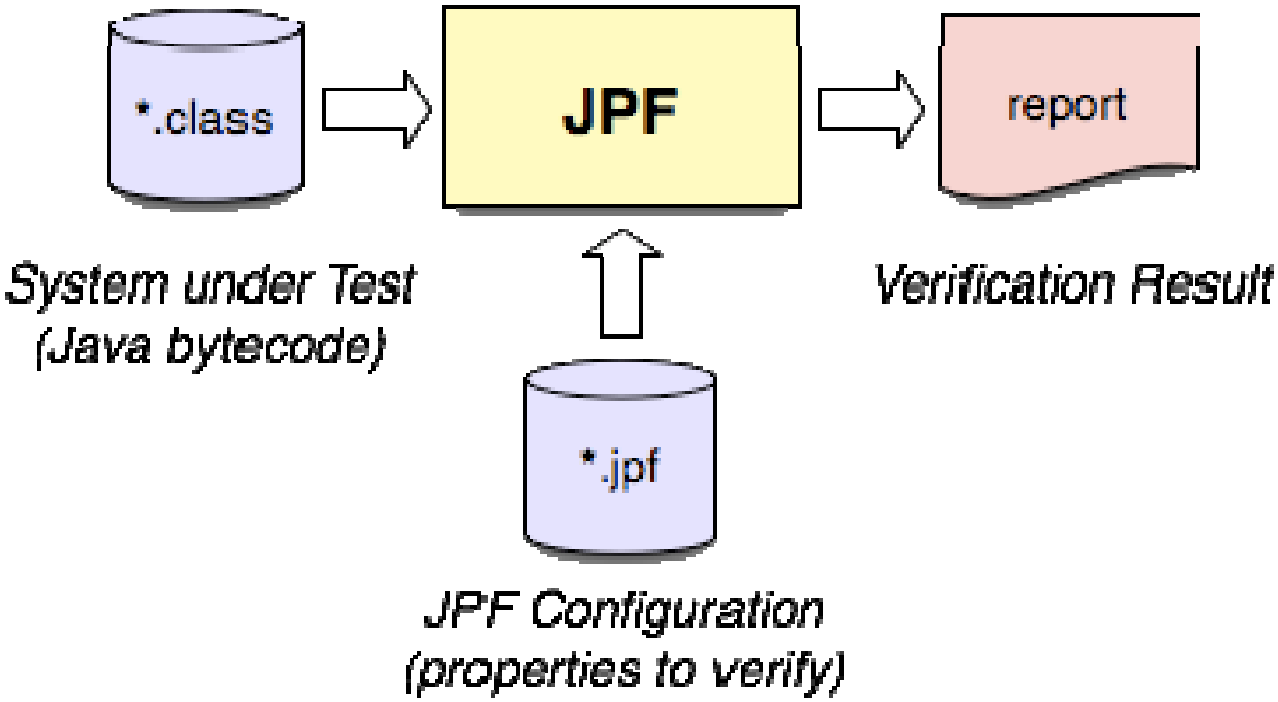
- Highly customizable and extensible platform for analysis and verification of Java programs
  - Primary mechanism: exhaustive state space traversal
    - Explicit representation of states (JVM) and transitions (Java code)
- Maintained by NASA Ames Research Center
- Web: http://babelfish.arc.nasa.gov/trac/jpf/

Outline

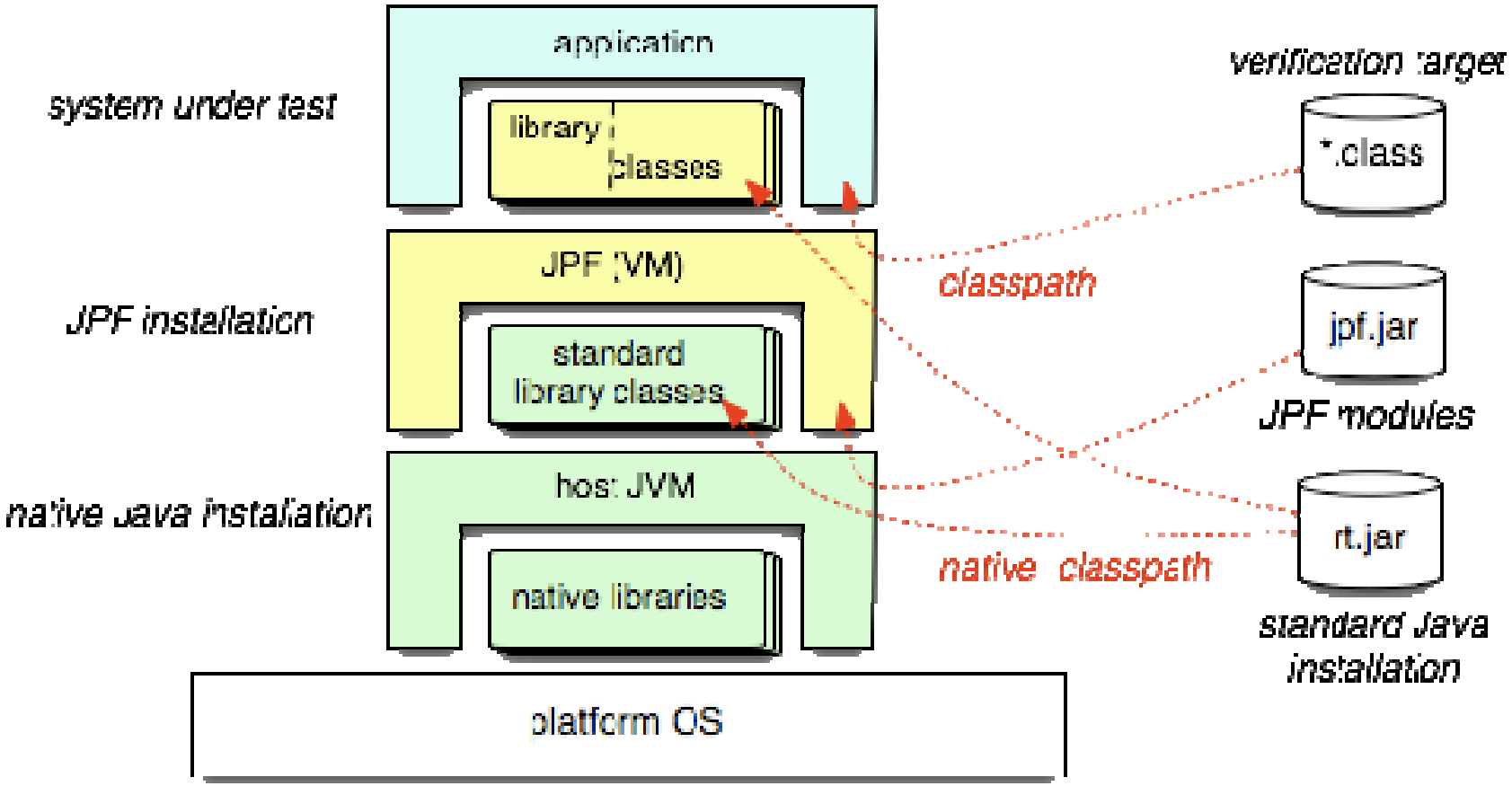
- Main features of JPF
  - Supported properties, algorithms, and optimizations
- Modular architecture of JPF
- Basic principles of internal functioning
- Extension points and JPF API
- Configuration options and running
- Major public extensions
- Current projects (GSoC)

Main features of JPF

Usage pattern



JPF stack


system under test
application
library classes
JPF installation
JPF (VM)
standard library classes
native java installation
host JVM
native libraries
platform OS
classpath
native classpath
verification target
*.class
jpf.jar
JPF modules
rt.jar
standard java installation

Supported properties

- Built-in
  - Assertion violations
  - Uncaught exceptions
  - Deadlocks
  - Race conditions
- Custom
  - Implemented using JPF's extension mechanisms

Verification process

- Exhaustive state space traversal
  - Standard DFS
- Supported optimizations
  - Partial order reduction (POR)
  - Various search heuristics
    - maximize thread interleaving, choose-free
  - State abstraction (user-defined)
  - Symmetry reductions
    - class loading, heap

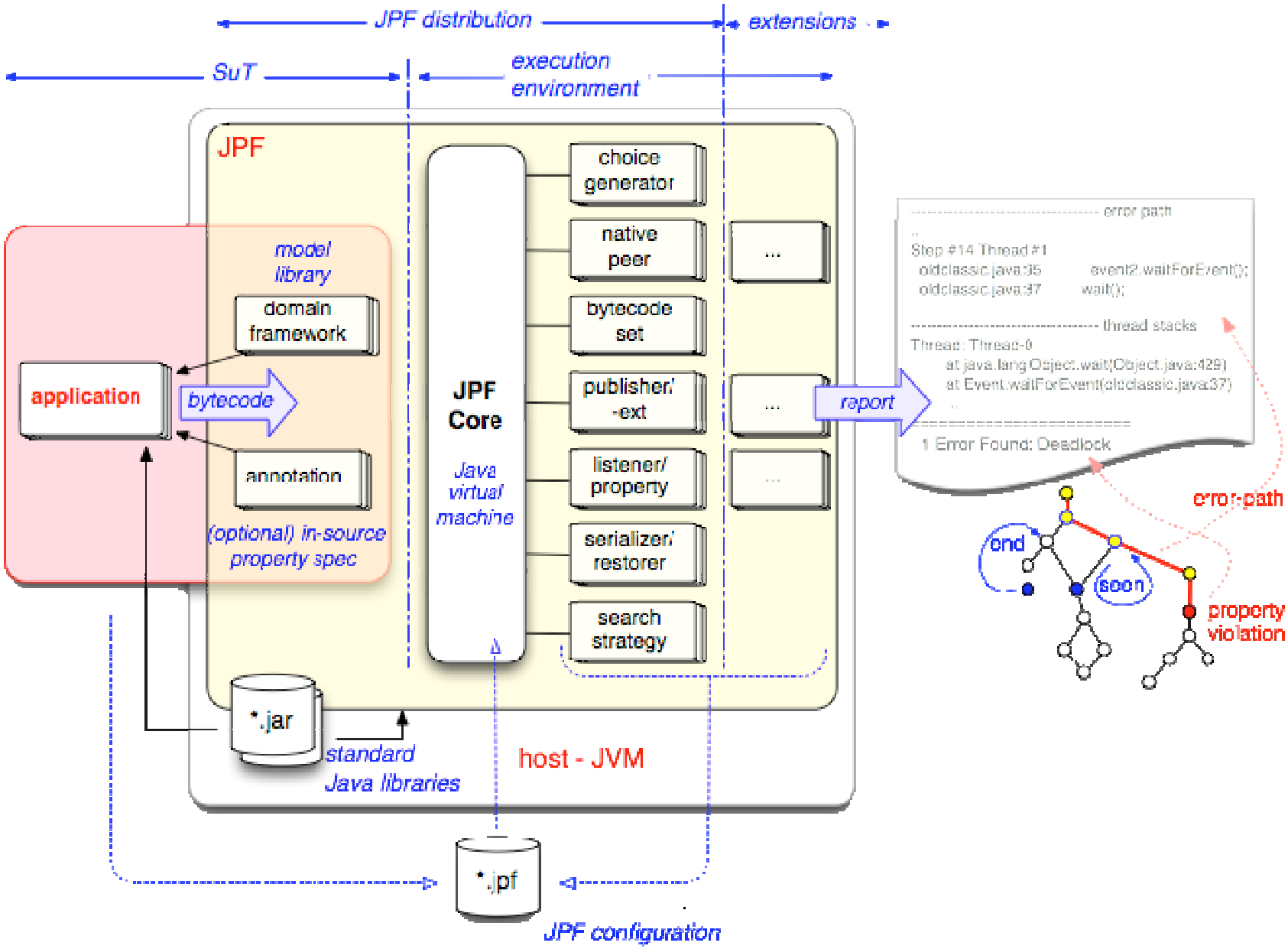
Modular architecture of JPF

JPF core

- Java Virtual Machine (JPF VM)
  - Execution choices (data non-determinism, scheduling)
  - Backtracking (restoring of previous states on a path)
  - State representation and matching
- "*Specialized JVM that explores all possible execution paths of a Java program*"

- Modular system (built from components)

Architecture of JPF core



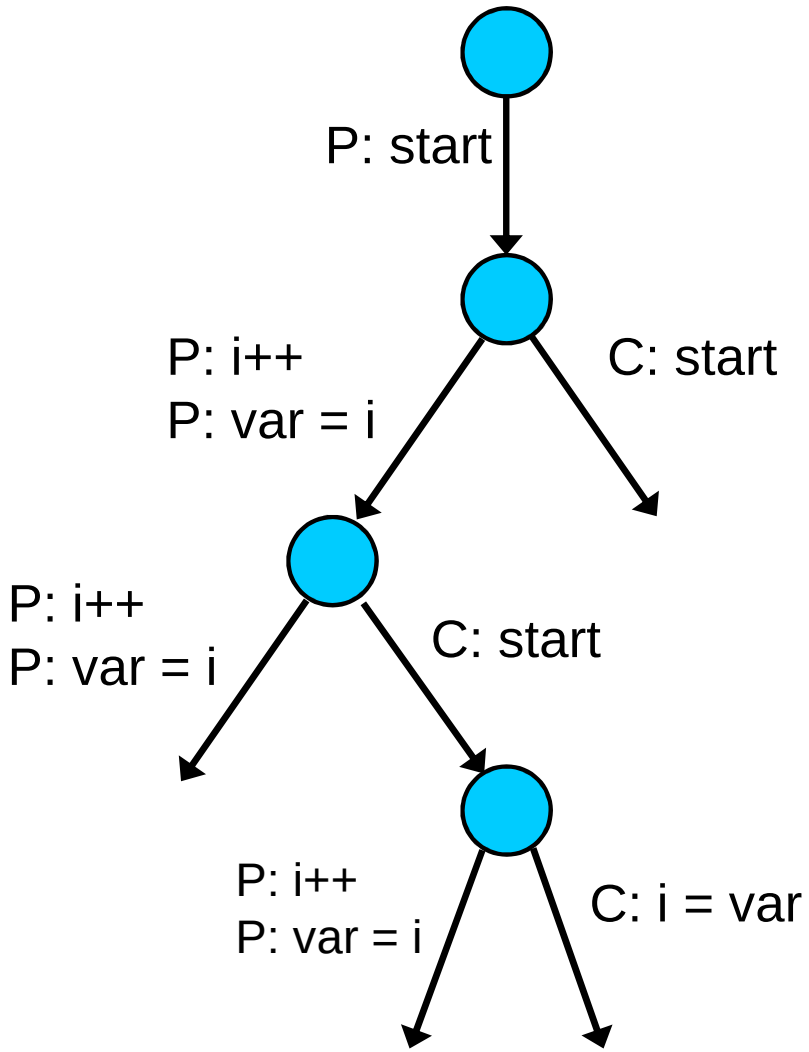
Java library classes

- Model classes: object and thread model of JPF
  - Separated from the underlying host JVM
  - Relevant core classes: `Thread`, `Object`, `Class`,
- Standard classes provided by host JVM
  - Example: `Integer`, `String`, collections (`java.util`),
  - Exception: classes with native methods

Basic principles of internal functioning

On-the-fly state space construction

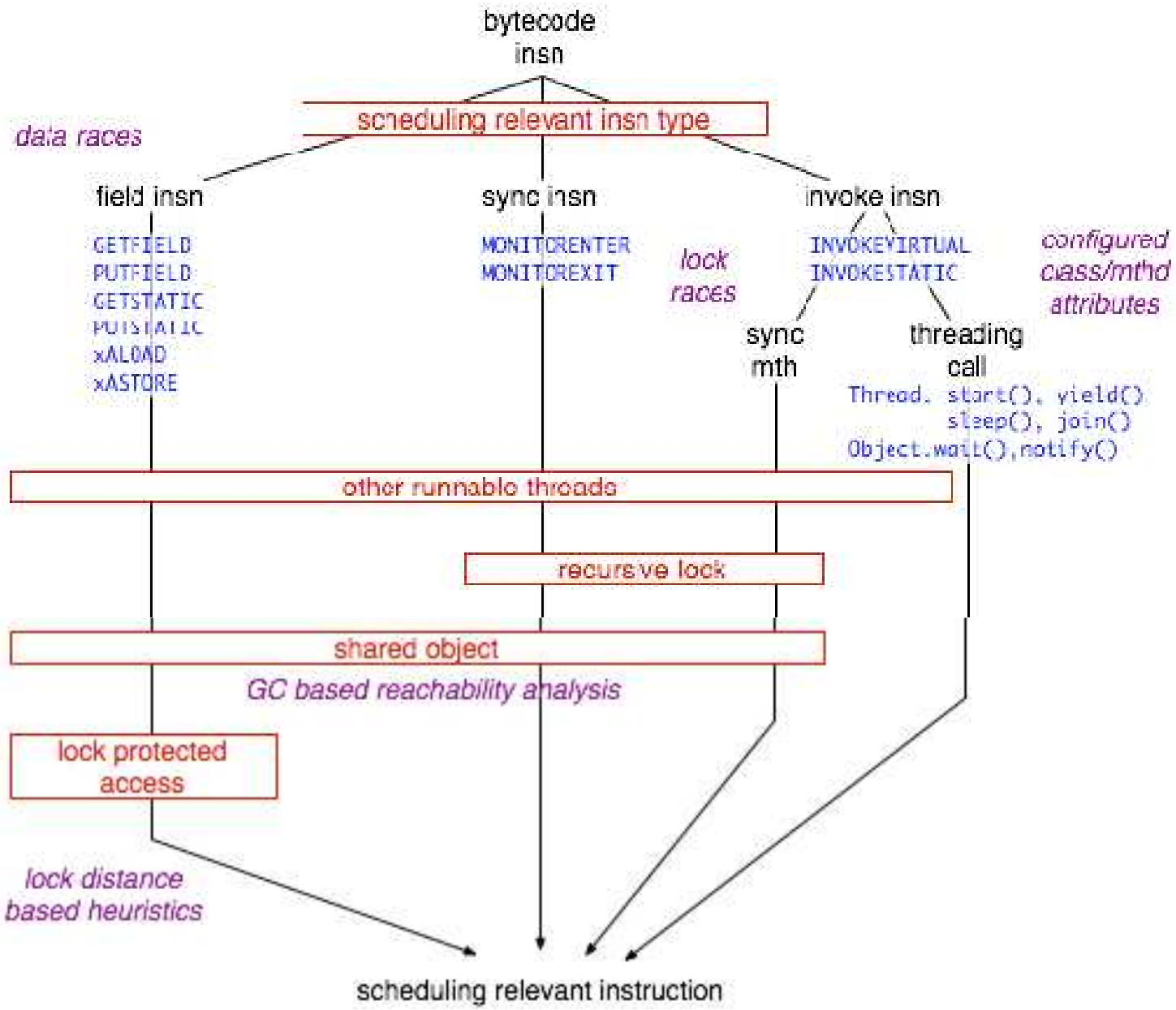

```
public Producer extends Thread {
  void run() {
    while (true) {
      Main.var = ++i;
    }
  }
}

public Consumer extends Thread {
  void run() {
    while (true) {
      i = Main.var;
      System.out.println(i);
    }
  }
}

public class Main {
  public static int var;

  public static void main(...) {
    (new Producer(var)).start();
    (new Consumer(var)).start();
  }
}
```



Practical approach to POR

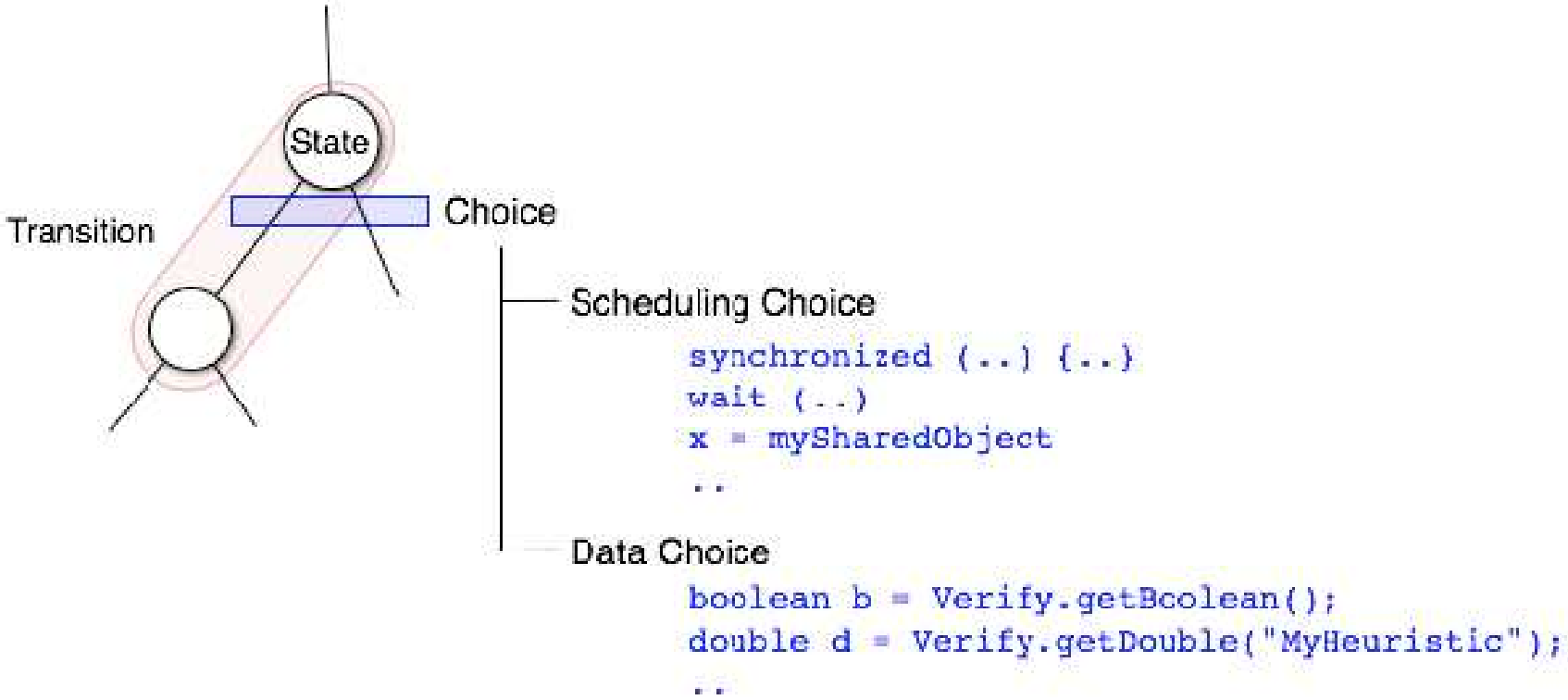
- Goal: elimination of unnecessary thread switches
  - bytecode instructions with thread-local effect
- Transition boundaries
  - Scheduling-relevant bytecode instructions (with globally visible effects)
    - Example: accessing a shared variable, start of a new thread
- Covers all interleavings of globally visible actions
- Limitations
  - Does not look ahead ➔ cannot handle diamond case

Scheduling-relevant instructions



Choice generator (CG)

- Common abstraction of execution choices of all kinds
  - Original motivation: unification of non-deterministic data selection and thread scheduling
- CG object manages the set of explored and unexplored transitions (choices) at each state



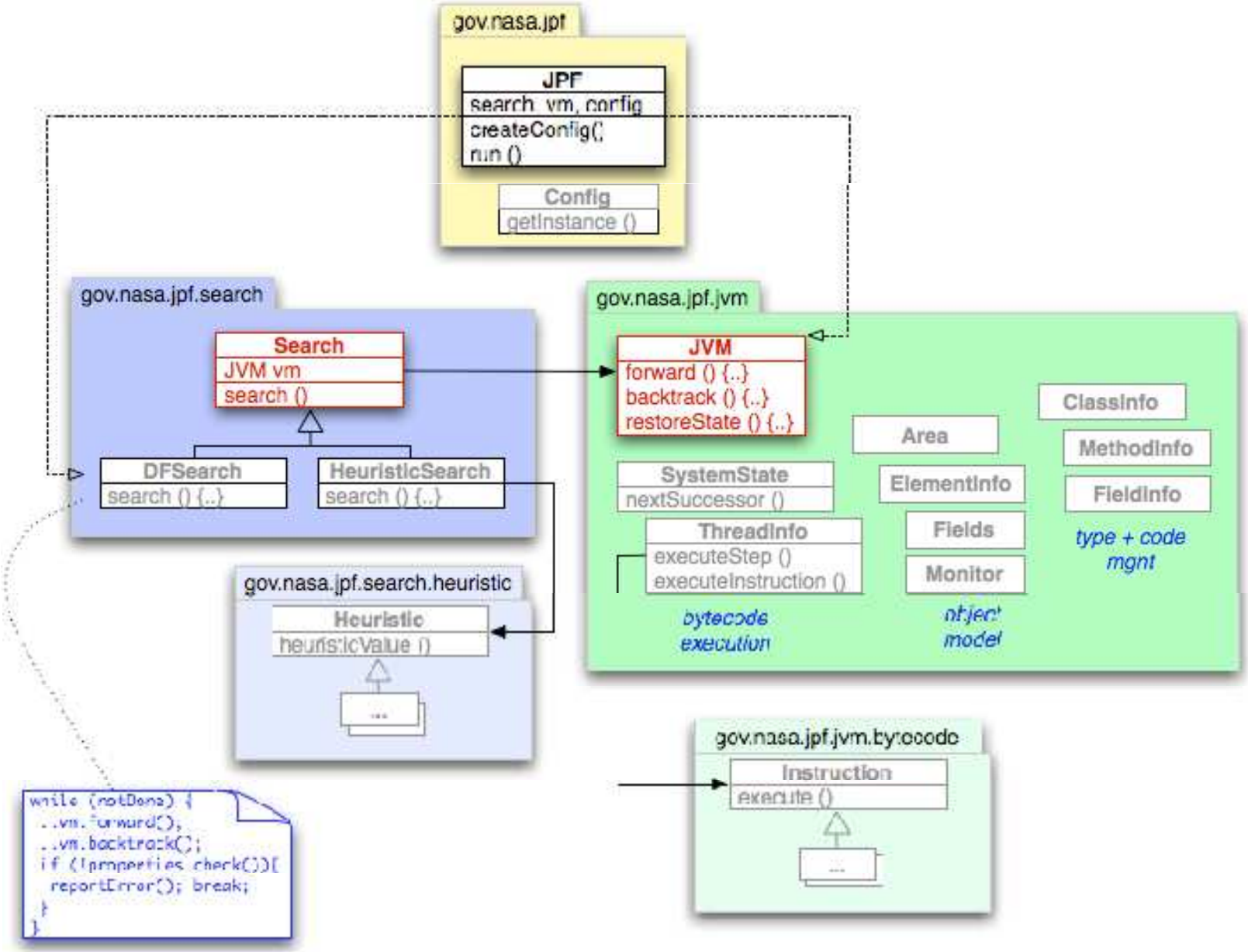
Encoding program state

- Current state
  - Program representation in JPF (data structures)
- Current path
  - Complete state of the program (JVM) represented by an integer array
    - complete image of heap, stacks of all threads, …
  - Incremental diffs are stored ➔ most transitions modify only a small part of the state
- Set of all visited states
  - Hash value ➔ for state matching

Symmetry reductions

- Class loading symmetry
  - Fixed loading order over all execution paths based on the first-time load on some execution path
- Heap symmetry
  - Canonical addresses of objects are used
    - Determined by first-time object allocation by a specific thread at a source code location

How JPF implements state space traversal

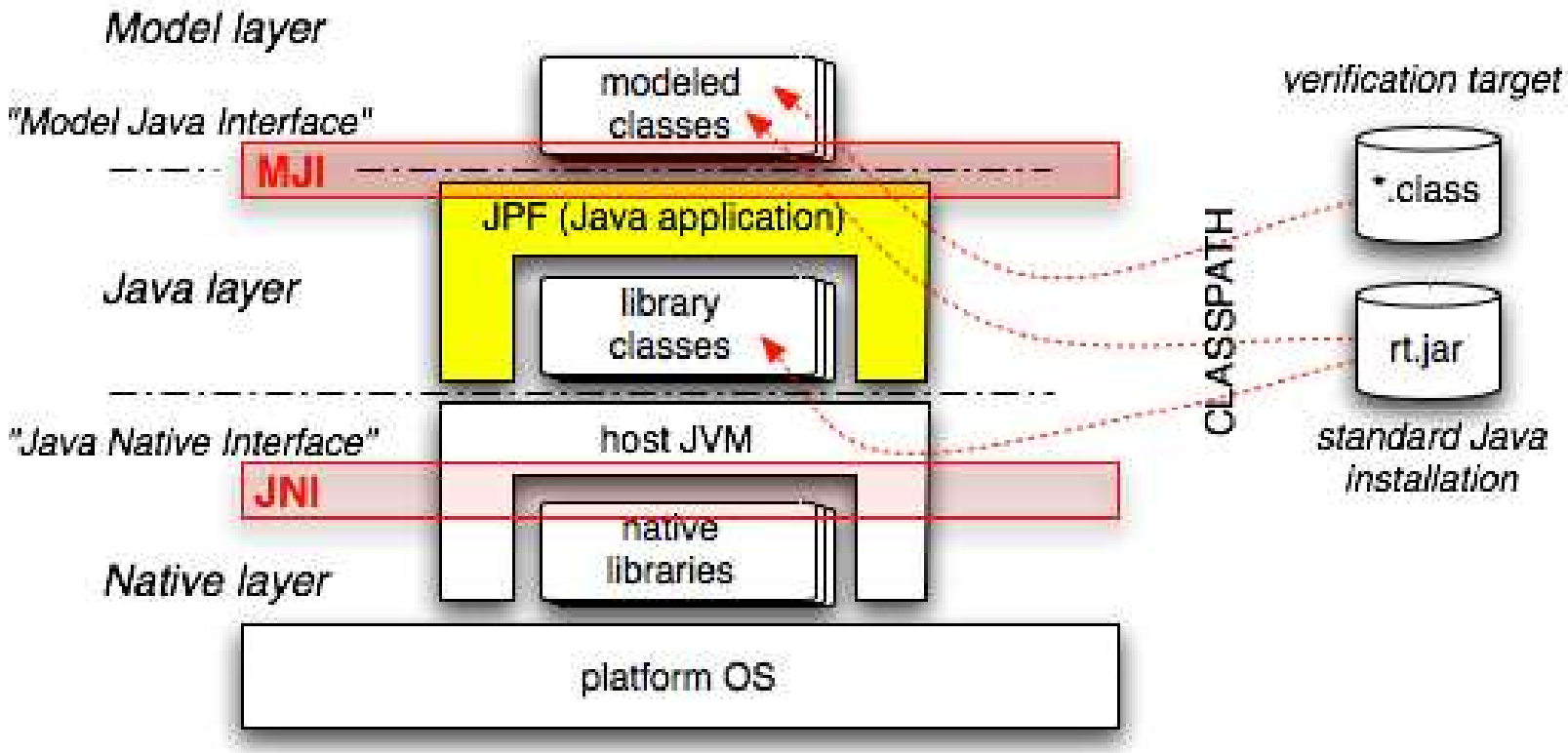


Native methods

- JPF supports all bytecode instructions
  - Custom implementation: for tracking state changes caused by their execution
- Problem: native methods
  - State changes cannot be tracked by JPF
  - Relevant features: file I/O, GUI, networking, …
- Remedy options
  - Custom implementation in Java ➔ model classes
  - Delegation to host JVM via the Model-Java Interface

Model-Java Interface (MJI)

- Allows execution of native methods in the host JVM
- Similar mechanism to Java-Native interface (JNI)



Remedy for native methods

- Model layer: model classes
  - Fully observable and backtrackable replacement of standard Java class
  - Part of JPF object model
- Java layer: native peers
  - Containers for pure Java methods executed instead of the intercepted real native methods
    - Changing program state (model) is possible via internal API
  - Executed directly by the host JVM (part of JPF code)

Library abstractions via MJI

- Currently supported features
  - File I/O + streams (java.io) – partially done
  - Network communication (java.net) – in progress
  - User interface (java.awt, java.swing) – in progress

Extension points and JPF API

Extension points and JPF API

- Listeners (search, VM)
- Custom properties
- The Verify API
- Bytecode instructions
- State management

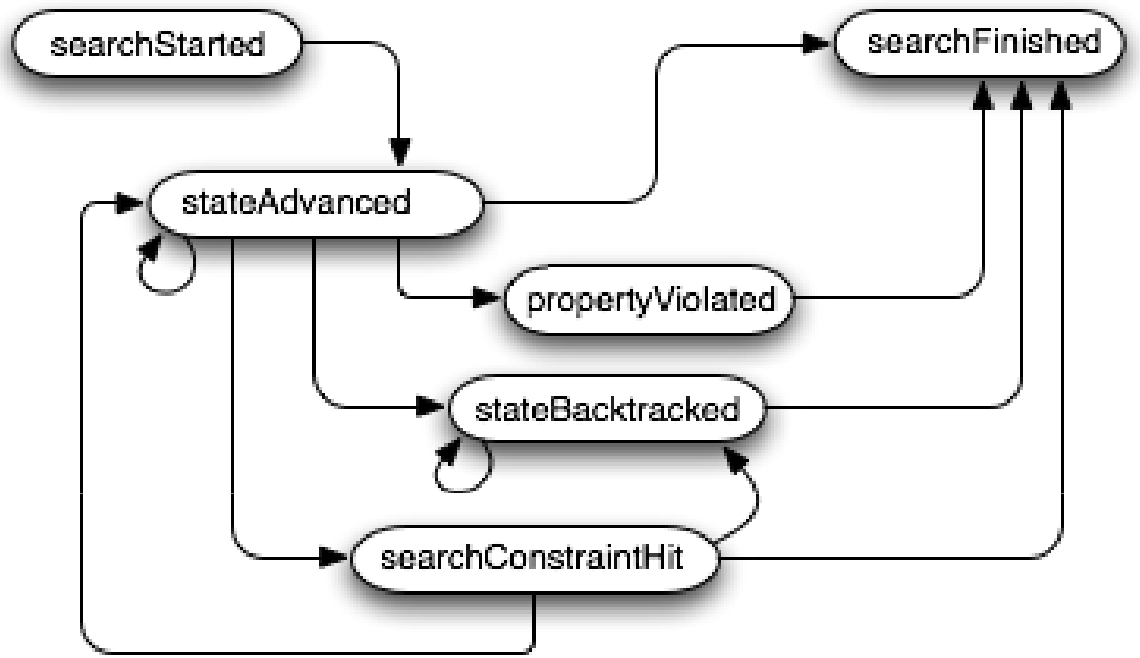
- Search algorithms
  - DFS (default), BFS, Simulation, Heuristic search
  - Several heuristics available in JPF distribution
    - Examples: maximize thread interleavings, choose free, …

Listeners

- Basic usage: monitoring
  - Registered observers are notified about specific events
    - State advanced (transition ended)
    - Bytecode instruction executed
    - New thread just started
    - New object was allocated
    - (and many other)
- Other ways of usage
  - Extending program state with custom information
  - Forcing termination of transitions at specific points
  - Custom properties based on monitoring results

SearchListener

- Subject: `Search` object
- Notification model (for DFS)



- Usage
  - Monitoring state space traversal
    - e.g. construction of the state space graph (visualization)

VMListener

- Subject: `JVM` object
  - Provides access to complete state of the Java program

- Usage
  - Monitoring execution of Java bytecode instructions
    - e.g. monitoring GETFIELD and PUTFIELD instructions for races
  - Inspecting the current state of the Java program
    - e.g. acquiring values of heap object fields, …
  - Tracing other JPF VM operations
    - Examples: thread start, new object allocated, choice selected
  - Altering program state and JPF VM behavior
    - e.g. modifying set of choices for CG and selecting the next one

The Verify API

- Non-deterministic data choice

```
import gov.nasa.jpf.jvm.Verify;
...
boolean b = Verify.getBoolean();
if (b) { ... }
else { ... }
...
int x = Verify.getInt(0, 10);
// do something with 'x'
```

  - Supported data types: boolean, int, double

- Search pruning

```
...
Verify.ignoreIf(cond);
... // search is pruned if the condition is true
```

Custom bytecode instructions

- Different semantics (non-standard)
  - Additional runtime checks
  - Registering custom CGs
  - Attaching some attributes to data (heap objects)
- Some major extensions use custom instructions
  - Example: symbolic execution

State management

- Several components
  - Serializer: full JVM state ➔ integer array
  - Backtracker: maintains current path (stack of states)
  - State set: storage of program states
    - Hash value (default) X lossless storage (full state)
- Custom implementations supported
  - Example: storage of states in a file
- Abstraction: filtering state information
  - Specific fields and classes (by name)

Configuration options and running

How to configure and run JPF

- Running: several options
  - Command-line (Ant)
  - IDE plug-ins (Eclipse, Netbeans)
  - Embedded in another Java program

- Configuration
  - Properties (example: `search.class = ...`)
  - Custom modules can introduce their own properties
  - Several levels
    - system, module (core, extension), application, specific JPF run

Output


```
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
========================================= system under test
application: /Users/pcmehlitz/tmp/Racer.java
========================================= search started: 5/24/07 12:32 AM
<application output>
====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException:
   division by zero at Racer.main(Racer.java:20)
====================================================== trace #1
…
------------------------------------------------- transition #1 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
Racer.java:17 : t.start();
Racer.java:19 : doSomething(1000); // (3)
Racer.java:6 : try { Thread.sleep(n); } catch (InterruptedException ix) {} [2 insn w/o sources]
Racer.java:6 : try { Thread.sleep(n); } catch (InterruptedException ix) {}
Racer.java:7 : }
Racer.java:20 : int c = 420 / racer.d; // (4)
…
====================================================== statistics
<running time, memory needed, total number of states, …>
```

Current state of affairs

Major extensions

- Symbolic execution (JPF-SE)
- GUI framework (JPF-Shell)
- RTSJ and SCJ programs (RTEmbed)
- Network applications (Net-IOCache)
- UML statecharts model checking
- … (and many others)

- IDE support (Eclipse, NetBeans)

Current projects

- Additional library abstractions
  - `java.util.concurrent`, `java.awt`
- Combination of JPF and Junit

- Google Summer of Code (GSoC)
  - JPF-Inspector, LTL checking, symbolic string analysis, Java memory model, native collections, …
  - Support for more (Java-based) platforms: Android, X10

- Most active institutions
  - Nasa AMES, …, UIUC, …, AIST (JP), CUNI/UW, NCSU

The End