Introduction to JavaPathfinder
Part 1

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Sagar Chaki
20 November 2007

Software Engineering Institute | Carnegie Mellon

Outline

Overview of JavaPathFinder

Tool Description (i.e., usage)

Example

How it works

- Architecture
- Specifying properties
- Modeling environment

Some slides borrowed from JavaPathFinder tutorial at ASE conference 2006

- http://www.visserhome.com/willem/presentations/ase06jpftut.ppt

Overview

What is JavaPathFinder (1)

Explicit state model checker for Java bytecode

Uses a customized Virtual Machine with backtracking capability to efficiently search a Java program's statespace

Focus is on finding bugs in Java programs

- concurrency related: deadlocks, (races), missed signals etc.
- Java runtime related: unhandled exceptions, heap usage, (cycle budgets)
- but also: complex application specific assertions

What is JavaPathFinder (2)

Goal is to avoid modeling effort (check the real program), or at least use a real programming language for complex models

Implies that the main challenge is scalability

JPF uses a variety of scalability enhancing mechanisms

- user extensible state abstraction & matching
- on-the-fly partial order reduction
- configurable search strategies: "find the bug before you run out of memory"
- user definable heuristics (searches, choice generators)

Key issue is configurable extensibility: overcome scalability constraints with suitable customization (using heuristics)

Key Points

Models can be infinite state

- Unbounded objects, threads,…
- Depth-first state generation (explicit-state)
- Verification requires abstraction

Handle full Java language

- ~~but only for closed systems~~
- cannot *directly* handle native code
  - no Input/output through GUIs, files, Networks, …
  - Must be modeled by java code instead

Allows Nondeterministic Environments

- JPF traps special nondeterministic methods

Checks for User-defined assertions, deadlock and user-specified properties

JPF Status

Developed at the Robust Software Engineering Group at NASA Ames Research Center

Currently in it's fourth development cycle

- v1: Spin/Promela translator - 1999
- v2: backtrackable, state matching JVM - 2000
- v3: extension infrastructure (listeners, MJI) - 2004
- v4: symbolic execution, choice generators - 4Q 2005

Open sourced since 04/2005 under NOSA 1.3 license:
<javapathfinder.sourceforge.net>

It's a first: no NASA system development hosted on public site before

11100 downloads since publication 04/2005

Tool Description: Usage

Using JavaPathfinder

Intended to be a drop-in replacement for *java,* the Java VM

- Thus, JPF accepts Java class files as input

Typically verify Java sources files in two steps:

1. Compile *.java* file(s) to *.class* file(s) using *javac*
2. Run JPF on the *.class* file(s)

Command line interface

- Can run JPF within Eclipse, but output is still textual, i.e., no GUI-based counterexample viewer like CBMC

Examples

Counter

```
int i = 0;
while (i < 2)
      i++;
assert(i == 2);
```

Counter

```
int i = 0;
while (i < 2)
       i++;
assert(i == 2);
```

i = 0

Counter

```
int i = 0;
while (i < 2)
      i++;
assert(i == 2);
```

i = 0

Counter

```
int i = 0;
while (i < 2)
      i++;
assert(i == 2);
```

i = 1

Counter

```
int i = 0;
while (i < 2)
      i++;
assert(i == 2);
```

i = 1

Counter

```
int i = 0;
while (i < 2)
       i++;
assert(i == 2);
```

i = 2

Counter

```
int i = 0;
while (i < 2)
       i++;
assert(i == 2);
```

i = 2

Choice

```
int x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

Choice

```
int x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

x =-1

y = -1

Choice

```
int x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

x =-1

y = -1

choice = false

Choice

backtrack

```
x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

x =-1

y = -1

choice = false

Choice

```
int x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

x =-1

y = -1

Choice

```
int x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

x =-1

y = -1

choice = true

Choice

```
int x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

x =-1

y = -1

choice = true

Choice

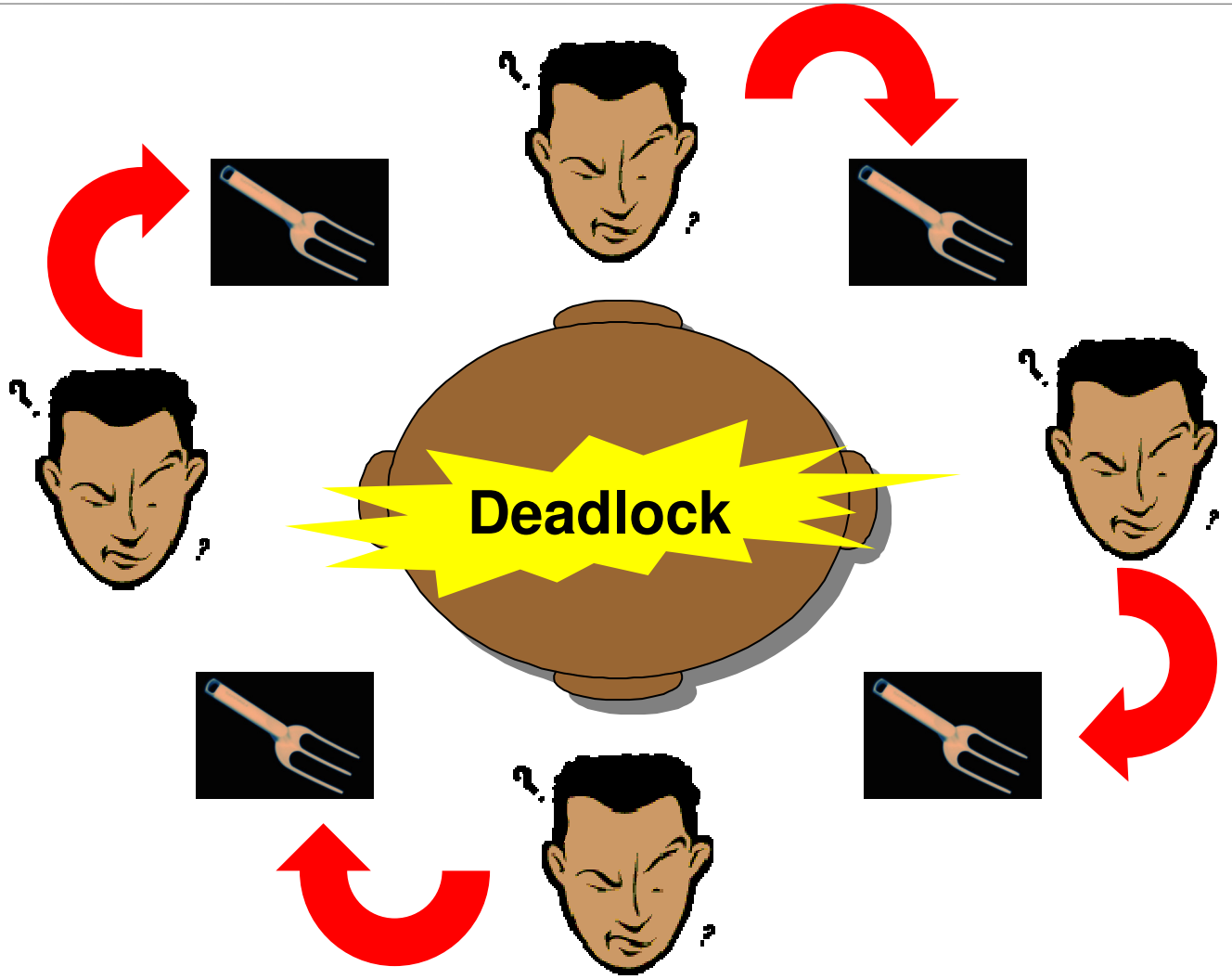
```
int x = -1,y = -1;
boolean choice = Verify.getBoolean();
if(choice)
  y = 1;
assert(x == y);
```

x =-1

y = 1

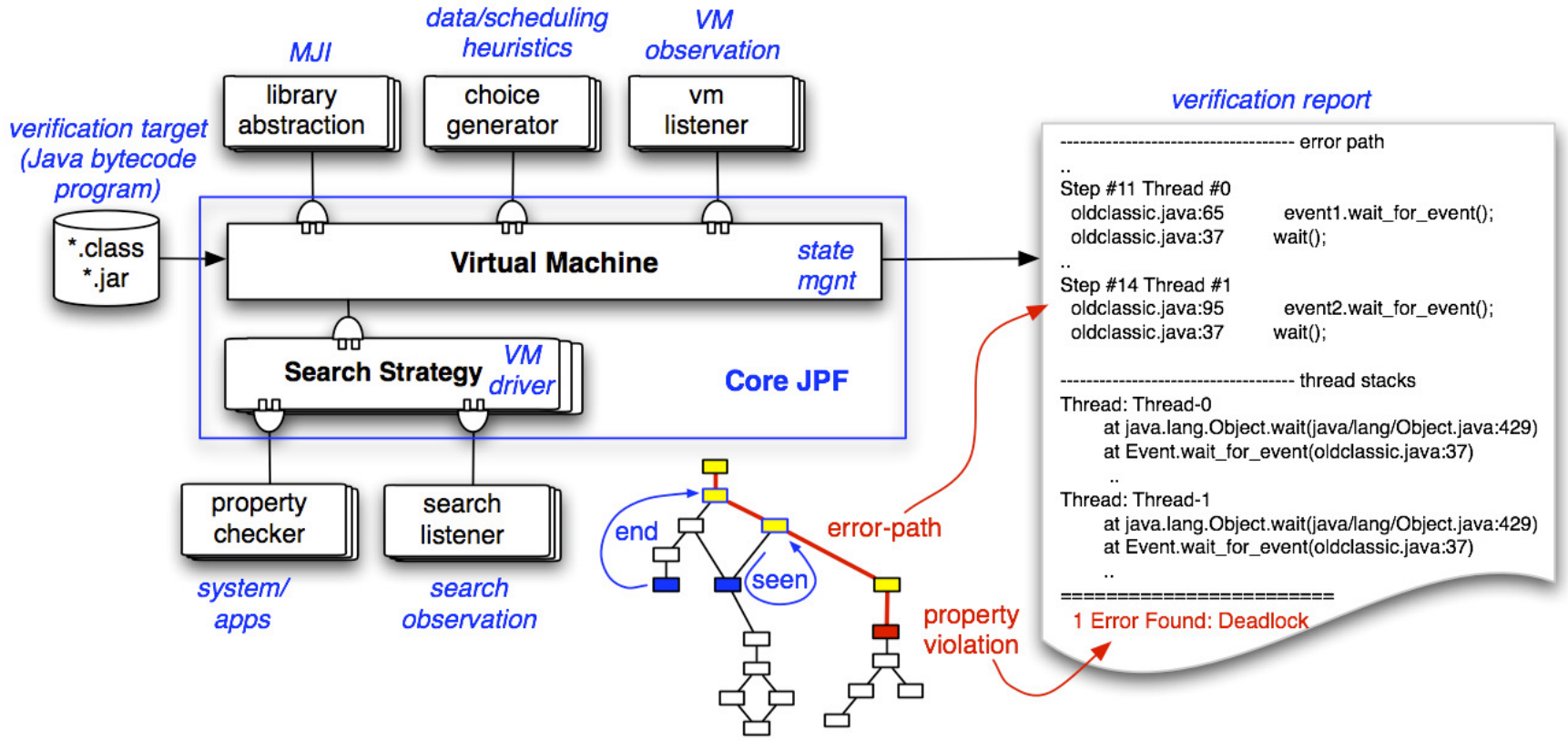
choice = true

Dining Philosophers



How it Works: Architecture

JavaPathFinder Architecture

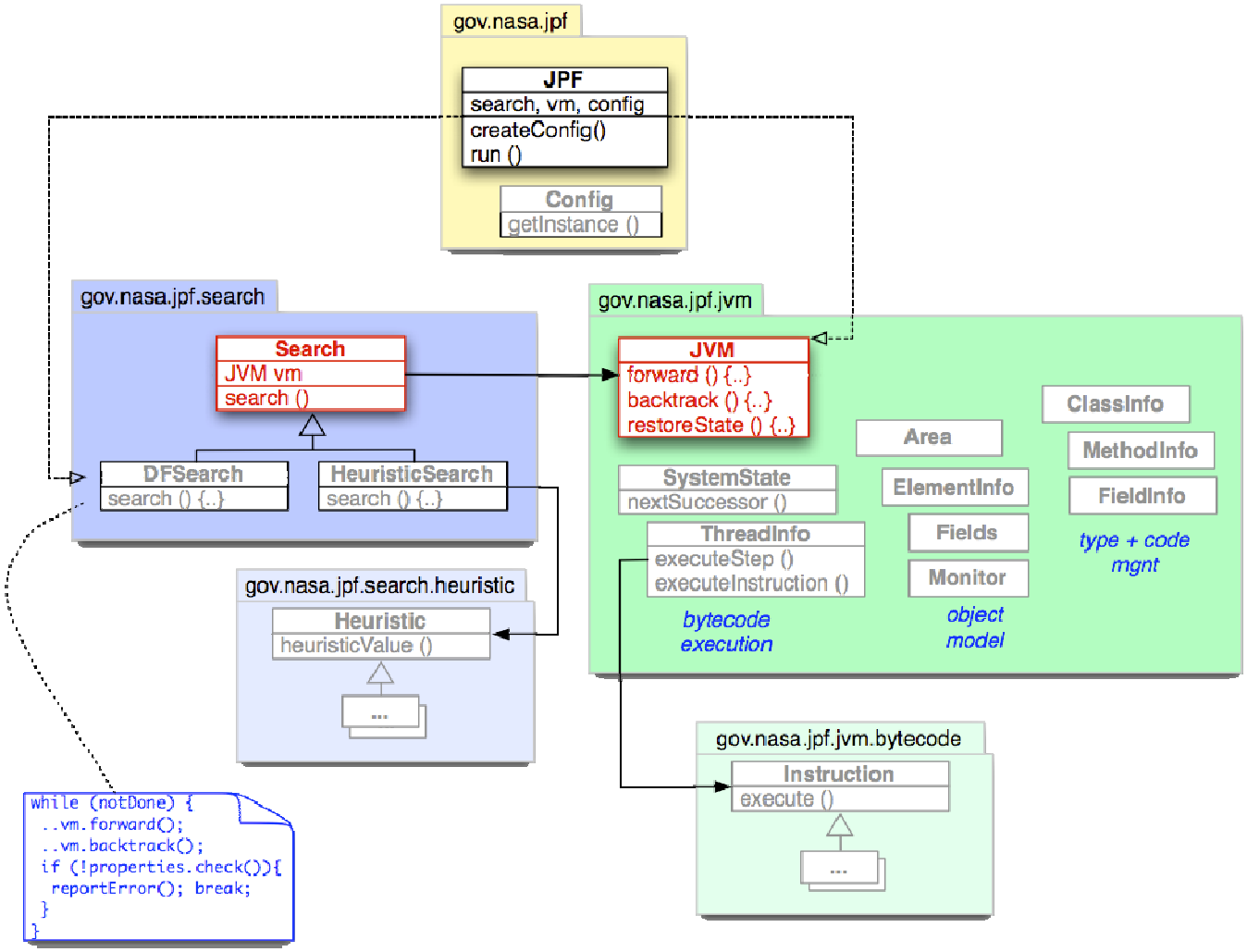


Under the Hood: Toplevel Structure

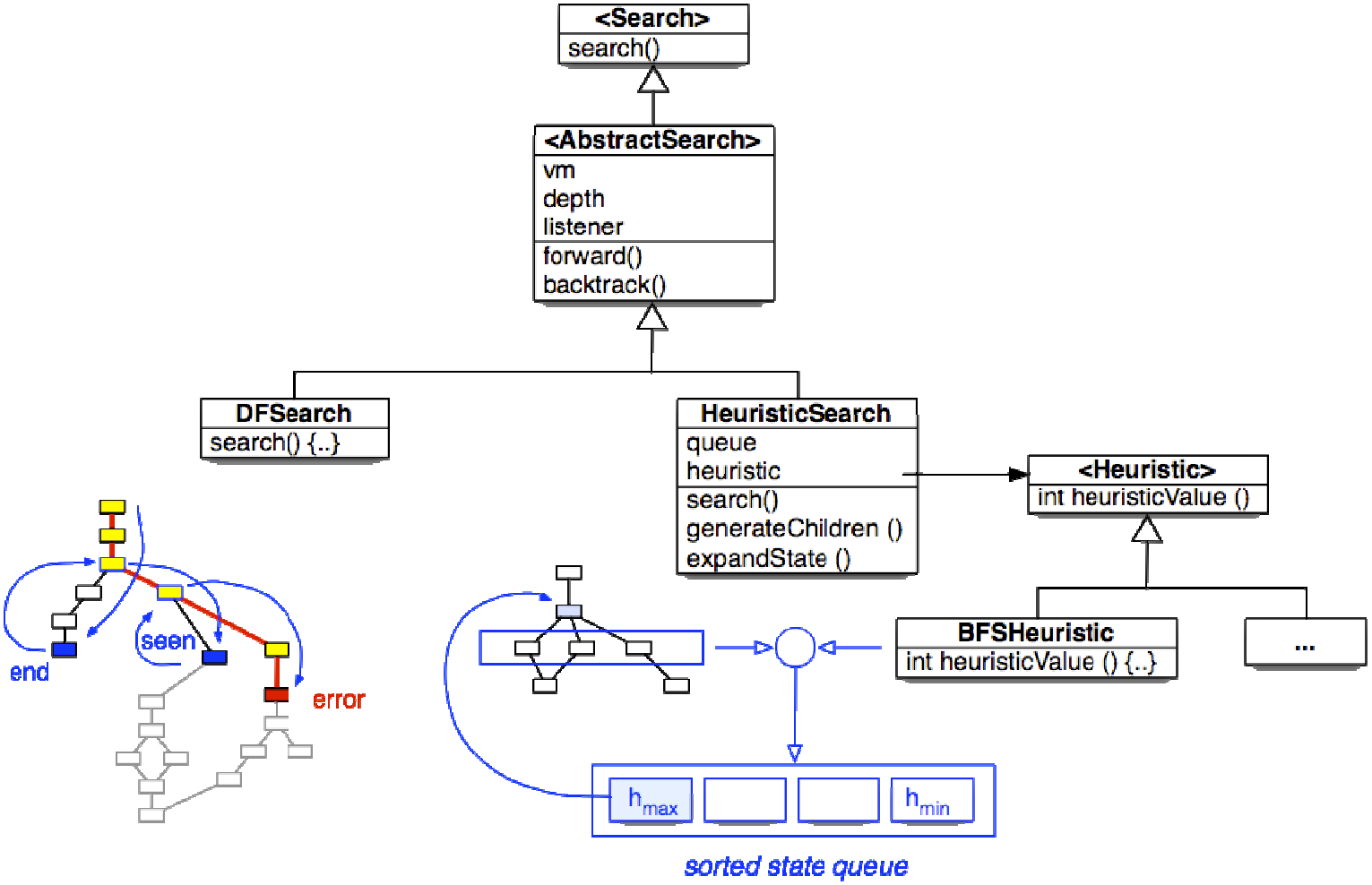
Two major concepts: Search and VM

Search is the VM driver and Property evaluator

VM is the state generator



Under the Hood: Search



Specifying Properties

Implement Properties in JPF

Java assertions

- Basic safety properties
- We have already seen examples of these

Use gov.nasa.jpf.Property

- Simple properties that can be checked based on the information remaining after the execution of a transition

Use gov.nasa.jpf.SearchListener and gov.nasa.jpf.VMListener

- More complex properties

Details at the JPF website

- javapathfinder.sourceforge.net/doc/How_to_Implement_Properties.html

Implementing the Property Interface

JPF hardcoded with three properties: *NotDeadlockedProperty*, *NoAssertionViolatedProperty*, and *NoUncaughtExceptionsProperty*

New properties can be added by implementing the *Property* interface

```
public interface Property extends Printable {
        boolean check (Search search, VM vm);
        String getErrorMessage();
}
```

or by extending the *GenericProperty* class, i.e., overriding the *check* method

New checks can be added statically or dynamically. All registered checks are executed by JPF at the end of every state transition.

In case a *Property.check(..)* method implementation returns false, and termination has been requested, the search process is ended, and all violated properties are printed (which potentially includes error traces)

Using Listeners (1)

The gov.nasa.jpf.SearchListener and gov.nasa.jpf.VMListener instances can be used to implement more complex checks that do require more information than what is available after a transition got executed.

The rich set of callbacks enables listeners to monitor almost all JPF operations and translate them into internal state.

JPF execution control can be achieved in two ways:

1. By implementing both the appropriate listener interface and the *gov.nasa.jpf.Property* interface, then registering with *Search.addProperty(..)*, to let JPF automatically check for violated property termination between states.
2. By calling *Search.terminate()* to stop searching for new states. This can be done from anywhere within the listener, but does not automatically create error reports, which have to be done explicitly by the listener.

Using Listeners (2)

JPF includes a gov.nasa.jpf.PropertyListenerAdapter class, which can be used as base class for complex properties.

Subclasses only have to implement the interface methods they are interested in, property registration is performed automatically during the *SearchListener.searchStarted* notification.

The typical design for such a subclass is to use *VMListener* methods to determine when the property fails, and then store this condition in a field which is evaluated in the *Property.check()* method.

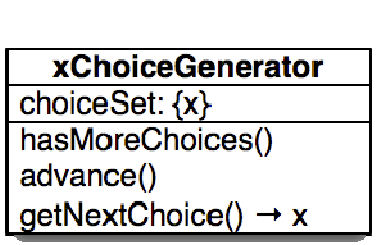
Examples of complex properties following this scheme can be found in directory *src/gov/nasa/jpf/tools* (e.g. *RaceDetector*).

JPF might still execute instructions after the property failure was detected, since the *check()* method is only called after the transition is completed.

Environment Modeling

Choice Generator Motivation

`Verify.getBoolean()`	C = { true, false }	✓	
`Verify.getInt(0,4)`	C = { 0, 1, 2, 3, 4 }	?	potentially large sets with lots of uninteresting values
`Verify.getDouble(1.0,1.5)`	C = { ∞ }	??	no finite value set without heuristics



Choice Generators

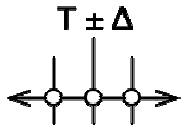
JPF internal object to store and enumerate a set of choices

+

Configurable Heuristic Choice Models

configurable classes to create ChoiceGenerator instances

e.g. "Threshold" heuristic



C = { T-Δ, T, T+ Δ }

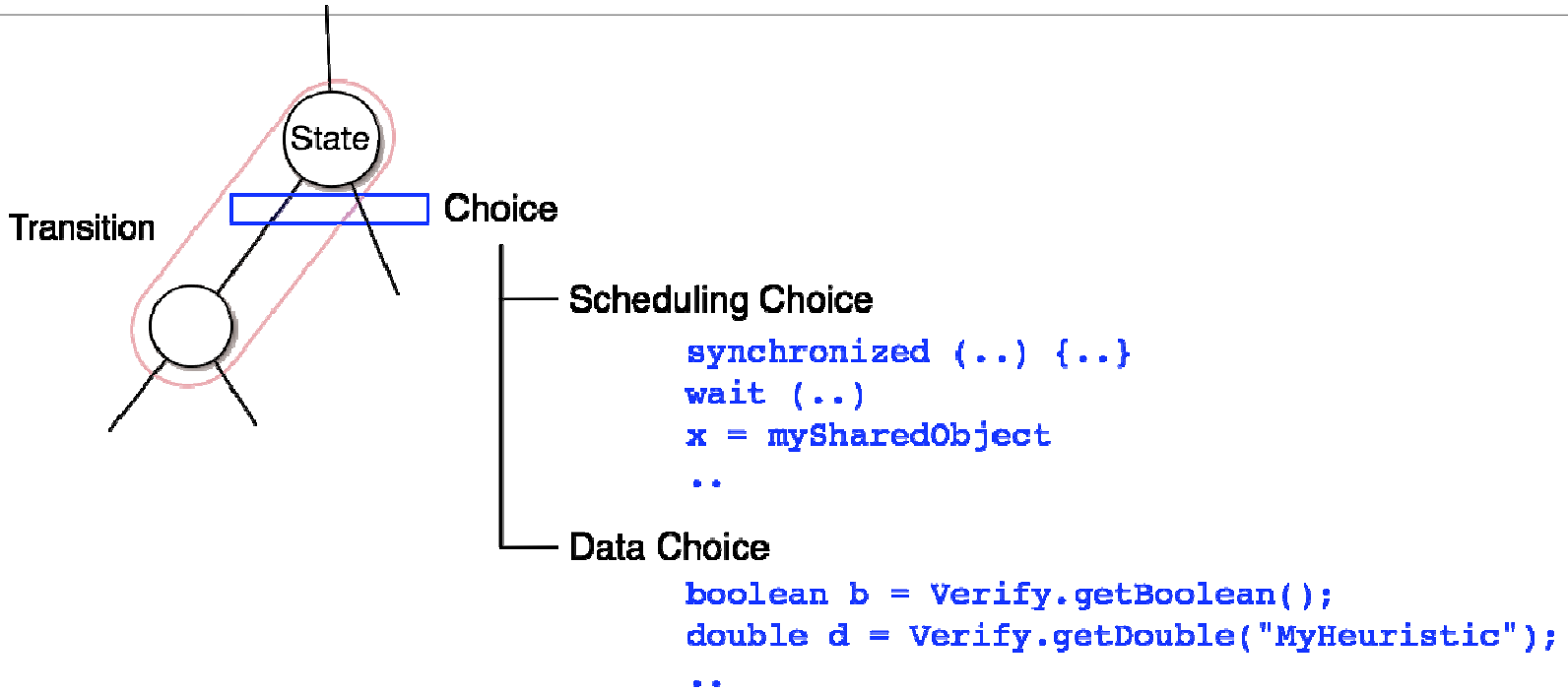
application code (test driver)

```
..
double v = Verify.getDouble("velocity");
..
```

configuration (e.g. mode property file)

```
velocity.class = gov.nasa.jpf.jvm.choice.DoubleThresholdGenerator
velocity.threshold = 13250
velocity.delta = 500
```

JPF Perspective



State consists of 2 main components, the state of the JVM and the current and next choice Generator (i.e. the objects encapsulating the choice enumeration that produces new transitions)

Transition is the sequence of instructions that leads from one state. There is no context within a transition, it's all in the same thread. There can be multiple transitions leading out of one state

Choice is what starts a new transition. This can be a different thread, i.e. scheduling choice, or different “random” data value.

Role of Choices

In other words, possible existence of Choices is what terminates the last Transition, and selection of a Choice value precedes the next Transition.

The first condition corresponds to creating a new ChoiceGenerator, and letting the SystemState know about it.

The second condition means to query the next choice value from this ChoiceGenerator (either internally within the JVM, or in an instruction or native method).

Software Engineering Institute | Carnegie Mellon