Java PathFinder

http://d3s.mff.cuni.cz

Pavel Parízek

Department of Distributed and Dependable Systems

FACULTY OF MATHEMATICS AND PHYSICS
Charles University
Java PathFinder (JPF)

- Verification framework for Java programs
  - Explicit state space traversal (with POR)
  - Highly customizable and extensible (API)

- Open source since April 2005
  - Maintainers: NASA Ames Research Center

- Available on GitHub since 2018

- WWW: https://github.com/javapathfinder/jpf-core
What JPF really is ...

- Special JVM
  - Execution choices
  - Backtracking
  - State matching

- State space exploration
  - assertions, deadlocks, races, ...
General usage pattern

![Diagram of JPF usage pattern](https://github.com/javapathfinder/jpf-core/wiki)

*Picture taken from JPF wiki (https://github.com/javapathfinder/jpf-core/wiki)*
Picture taken from JPF wiki (https://github.com/javapathfinder/jpf-core/wiki)
Program state space in JPF

- States
  - Full snapshot of JVM
    - Dynamic heap
    - Thread stacks
    - Program counters
    - Static data (classes)
    - Locks and monitors
Program state space in JPF

- Transitions
  - Non-empty sequences of bytecode instructions
  - Terminates when JPF makes a new choice
Program state space in JPF

- Choices
  - Thread scheduling
  - Data (boolean, int)
public Producer extends Thread {
    void run() {
        while (true) {
            d.buf = i;
            i++;
            d.count++;
        }
    }
}

public Consumer extends Thread {
    void run() {
        while (true) {
            k = d.buf;
            print(k);
        }
    }
}

public static void main(...) {
    Data d = new Data();
    new Producer(d).start();
    new Consumer(d).start();
}
public Producer extends Thread {
    void run() {
        while (true) {
            d.buf = i;
            ++i;
            d.count++;
        }
    }
}

public Consumer extends Thread {
    void run() {
        while (true) {
            k = d.buf;
            print(k);
        }
    }
}

public static void main(...) {
    Data d = new Data();
    new Producer(d).start();
    new Consumer(d).start();
}
public Producer extends Thread {
    void run() {
        while (true) {
            d.buf = i;
            i++;
            d.count++;
        }
    }
}

public Consumer extends Thread {
    void run() {
        while (true) {
            k = d.buf;
            print(k);
        }
    }
}

public static void main(...) {
    Data d = new Data();
    new Producer(d).start();
    new Consumer(d).start();
}
Properties

- Built-in
  - Deadlock freedom
  - Race conditions
  - Uncaught exceptions
  - Assertions
Features

- Partial order reduction
- Class loading symmetry
- Heap symmetry
- Selected heuristics
Running JPF
Running JPF

- Download JPF and unpack somewhere

- Example: Dining Philosophers
  - Command: `java -jar build\RunJPF.jar src\examples\DiningPhil.jpf`

- Output: application, error info, statistics
Error info

- Full error trace (counterexample)
- Snapshot of the error state
- Message from the property checker

Command:
```
java -jar build\RunJPF.jar +report.console.property_violation =trace,error,snapshot src\examples\DiningPhil.jpf
```
Running JPF

- Examples
  - BoundedBuffer
  - Crossing
  - oldclassic
  - Racer
JPF API
JPF API

- Listeners
  - Inspecting current program state
- Custom properties
- Search driver

- Advanced
  - Instruction factory (bytecode interpreter)
  - Scheduler (sync policy, sharedness policy)
Listeners

- Observer design pattern

- Notified about specific events
  - JVM: bytecode instruction executed, new heap object allocated, start of a new thread
  - State space traversal: new state, backtrack, finish

- Inspecting current program state
  - heap objects, local variables, thread call stacks, ...
Listeners

- SearchListener
- VMLListener
- ListenerAdapter

Examples (source code)

- JPF/src/main/gov/nasa/jpf/listener
Custom properties

- Property
- GenericProperty

- PropertyListenerAdapter
  - Common practice: decide property status based on listener notifications (and program state)

- Examples (source code)
  - JPF/src/main/gov/nasa/jpf/vm
Registering listeners and properties

\[ \text{listener} = \langle \text{class name 1} \rangle, \ldots, \langle \text{class N} \rangle \]
\[ \text{search.listener} = \ldots \]
\[ \text{search.properties} = \ldots \]
Listeners: tracking bytecode instructions

- ExecTracker
- ObjectTracker
Listeners: inspecting program state

- CallMonitor
- ObjectTracker
Task 1

• Write your own listener
  ▪ After every field write instruction, print the field name and new value
  ▪ Before every method call (invoke), print values of all parameters supplied by the caller

• Use existing classes as a basic template
  ▪ ListenerAdapter, VMLListener, CallMonitor, ObjectTracker
  ▪ src/main/gov/nasa/jpf/listener/*
  ▪ src/main/gov/nasa/jpf/jvm/bytecode/*

• Ask questions !!
Configuration properties

- File `jpf.properties`
• Main page
  - https://github.com/javapathfinder/jpf-core/wiki

• User guide

• Internals (developer guide)
JPF source code tree

- src/main/gov/nasa/jpf
  - the “main” class (JPF), interfaces
  - vm: virtual machine, choices, built-in properties
  - jvm: Java bytecode specific, instructions, class file
  - search: search driver, heuristics
  - util: custom data structures, utility classes
  - report: reporting system (console, XML)
  - listener: various listeners
JPF and native methods

- Support for all Java bytecode instructions
  - but some library methods are native
    - file I/O, GUI, networking, ...

- Problem
  - JPF cannot determine how execution of a native method changes the program state

- Solution: Model-Java Interface (MJI)
Model-Java Interface (MJI)

- Executing native methods in the underlying JVM
- Similar mechanism to Java-Native Interface (JNI)
- Custom versions of some Java library classes
  - Object, Thread, Class, java.util.concurrent.*, ...

1. Executing native methods in the underlying JVM
2. Similar mechanism to Java-Native Interface (JNI)
3. Custom versions of some Java library classes
   - Object, Thread, Class, java.util.concurrent.*, ...

 Diagram:

- Model layer
- "Model Java Interface"
- MJI
- Java layer
- "Java Native Interface"
- JNI
- Native layer
- Host JVM
- Native libraries
- Platform OS
- JPF (Java application)
- Modeled classes
- Library classes
- Verification target
- .class
- rt.jar
- CLASSPATH
- Standard Java installation
- Systems
Environment construction
Environment construction

- Why: some programs do not contain "main"
  - libraries, components, plug-ins

- Problem: JPF accepts only complete programs

- Solution: create artificial environment
  - Program with multiple threads and data choices
    - Also called “test driver”
**Example**

- **Program:** `java.util.HashMap`

```java
public class PutTh extends Thread {
    Map m;

    public void run() {
        m.put("1", "abc");
        m.put("2", "def");
    }
}

public class GetTh extends Thread {
    Map m;

    public void run() {
        m.get("1");
        m.get("0");
    }
}

public static void main(...) {
    Map m = new HashMap();
    Thread th1 = new PutTh(m);
    Thread th2 = new GetTh(m);
    th1.start();
    th2.start();
    th1.join();
    th2.join();
}
```
Environment construction – challenges

- **Coverage**
  - Should trigger all (most) execution paths, thread interleavings, and error states

- **Approach**
  - Different method call sequences
  - Many combinations of parameter values
  - Several concurrent threads

- **State explosion**
  - Use the least possible number of concurrent threads (2)
  - Reasonable number of parameter values (domain size)
Using the Verify class

- JPF-aware test drivers (environments)
  - Checking program behavior for different inputs

- Data choice
  
  ```java
  import gov.nasa.jpf.vm.Verify
  if (Verify.getBoolean())
  int x = Verify.getInt(0, 10)
  ```

- Search pruning
  
  ```java
  Verify.ignoreIf(cond)
  ```
Task 2

- Write reasonable environment for
  - `java.util.LinkedList`
  - `java.util.concurrent.Semaphore`

- Run JPF on the complete program
  - Enable search for data race conditions
    - Use: `gov.nasa.jpf.listener.PreciseRaceDetector`

- Try different workloads (threads, input data)
Time for questions about JPF

- Architecture
- Implementation
- How something works
- Public API
- Output

- Play with JPF
  - look into source code & try examples

- Explore wiki
  - https://github.com/javapathfinder/jpf-core/wiki

- Ask questions