Java PathFinder

Nadav Greenberg
Nadav Heinoch

December 8th, 2008
Outline

• Model checking
• What is JPF?
• JPF structure
• Finding our first bug
• Extensibility + Example
• JPF configurations
• The scalability problem
• Choice Generators + Example
• Lab exercise
What is Model Checking?

• Systematically verifying that a model satisfies a set of properties
  – Formal Model: UML state charts, Java programs, Promela models, ...
  – Properties: Temporal Logic (xTL), code assertions, ...

• In JPF:
  – The models are Java programs
  – The properties can be assertions, gov.nasa.jpf.Property objects, or JPF listener objects
Background: Model Checking vs. Testing/Simulation

- A test will explore a single execution path
  - *You* must identify each important execution path
  - *You* must find the inputs that will execute those paths.

A model checker can explore every execution path
- Including scheduler decisions for concurrent models

- A model checker can identify both errors and the execution paths leading to those errors
What is JPF?

• The answer used to be simple:
  "JPF is an explicit state software model checker for Java bytecode. Today, JPF is a Swiss army knife for all sort of runtime based verification purposes." from the java pathfinder homepage
• The reality is not as sweet as they’d want us to believe
• out of the box JPF is more of a debugging tool
• In recent years a strong emphasis was put on creating a well modular and extensible framework
• Many extensions now exist that make it a very powerful tool
  – Symbolic execution
  – UML state chart modeling
  – Etc.
What is JPF? – Cont.

- JPF is a Virtual Machine that runs on top of Java’s VM.
- Executes your program in all possible ways and checks for violation of pre-defined properties.
- Reports the whole execution path that led to the violation.
What Can JPF Handle?

• Pure Java up to ?? - KLOC
  – Depends on logical complexity and state size, not KLOC.
  – Programs with 1M+ lines have been analyzed
• Multi-threaded code (*Of Course!*)
  – To a certain limit, too many threads causes various problems that JPF cannot handle properly
• Can find: deadlocks, race conditions, unhandled exceptions, application-specific assertions, ...
What Can JPF Handle? – Cont.

- JPF can efficiently simulate non-determinism
  - JPF uses backtracking in order to efficiently cover all (?) possible program states.
  - JPF avoids unnecessary work by preventing re-entrance to previously visited states.
    - Too many states can crash this mechanism in unpleasant ways
What JPF is not

• JPF is not a Model Checker in it’s classic definition
  – It runs the code itself and not a symbolic model of it
  – It does not cover the whole space (for instance it needs a sub range for it’s variables)
  – It’s more of a debugger then a tool for proving correctness
JPF Diagram and Extensions

Virtual Machine

Core JPF

library abstraction
data/scheduling heuristics
VM observation
execution semantics
state-matching/-restoring

native peer
choice generator
vm listener
bytecode set
serializer, restorer

.property checker
.search listener

Search Strategy

VM driver

*e.class
*e.jar

verification target
(Java bytecode program)

system/apps
search observation

publisher, -extension

verification report

verification output
(report or GUI)

defect history

defect description

end
program trace
seen
property violation
A little history

- Developed at NASA Ames Research Center
  - Open source since April 2005

- v1: Spin/Promela translator - 1999
- v2: backtrackable, state matching JVM - 2000
- v3: extension infrastructure (listeners, MJI) - 2004
- v4: symbolic execution, choice generators - 4Q 2005
- Currently in it’s fourth development cycle
  - Still under active development (1104 revisions to version 4 alone,
    25 in the 2 weeks we have prepared this presentation…)
  - 100,000+ downloads at 2008
  - Version 5 is on it’s way
Example 1 - Racer

```java
public class Racer implements Runnable {
    int d = 42;
    public void run () {
        doSomething(1001);  // (1)
        d = 0;              // (2)
    }

    public static void main (String[] args){
        Racer racer = new Racer();
        Thread t = new Thread(racer);
        t.start();
        doSomething(1000);  // (3)
        int c = 420 / racer.d;  // (4)
        System.out.println(c);
    }

    static void doSomething (int n) {    // not very interesting..
        try { Thread.sleep(n); } catch (InterruptedException ix) {} 
    }
}
```
Racer – Cont.

Results:
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
======================================== system under test
application: Racer.java
======================================== search started: 5/12/08 5:01 AM
======================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty java.lang.ArithmeticException:
  division by zero at Racer.main(Racer.java:16)
----------------------------------------- transition #1 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
  Racer.java:6                   : doSomething(1001);
  Racer.java:22                  : try { Thread.sleep(n); } catch (InterruptedException ix) {}
  Racer.java:22                  : try { Thread.sleep(n); } catch (InterruptedException ix) {}
  Racer.java:23                  : }
  Racer.java:7                   : d = 0;
----------------------------------------- transition #2 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
  Racer.java:7                   : d = 0;
  Racer.java:8                   : }
----------------------------------------- transition #3 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  Racer.java:16                  : int c = 420 / racer.d;
======================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty "java.lang.ArithmeticException:
  division by zero at Racer.main(Racer.java:16)
======================================== search finished: 5/12/08 5:01 AM
Extensibility

• It’s fourth generation was designed to be a very flexible, modular & future proof framework

• 4 Major extension mechanisms:
  – Search / VMListeners
  – Model Java Interface (MJI)
  – Configurable ChoiceGenerators
  – Bytecode Factories
Search- and VMListeners

- Listeners are the preferred way of extending JPF (an advanced user can create sophisticated extensions)
- Listener instances register themselves with Search/VM objects
  - get notified when their corresponding Subjects perform certain operations.
- Primary usage
  - Gather statistics
  - Monitor state exploration progress
  - Query details of states like field values
  - Etc…
VMListener

- Provide an extension mechanism that enables adding the mentioned functionality without modifying VM implementations
  - Not part of the core in order to keep it small and abstract, but usually come with JPF implementation packages.
- VMListeners are used in order to follow the detailed VM processing
- Observe the VM’s execution (bytecode execution, exceptions, thread starts, ...)
- Easily configurable (will be discussed later)
public interface VMListener {
    void instructionExecuted (JVM vm); // VM has executed next instruction
    void threadStarted (JVM vm); // new Thread entered run() method
    void threadTerminated (JVM vm); // Thread exited run() method
    void classLoaded (JVM vm); // new class was loaded
    void objectCreated (JVM vm); // new object was created
    void objectReleased (JVM vm); // object was garbage collected
    void gcBegin (JVM vm); // garbage collection mark phase started
    void gcEnd (JVM vm); // garbage collection sweep phase terminated
    void exceptionThrown (JVM vm); // exception was thrown
    void nextChoice (JVM vm); // choice generator returned new value
}
public class OpCodePrinter extends ListenerAdapter {
    String lastLoc = "";

    public void instructionExecuted(JVM vm) {
        Instruction instr = vm.getNextInstruction();
        if (instr != null) {
            String loc = instr.getFileLocation();
            if (loc != null && ! loc.startsWith("java")) {
                if (! lastLoc.equals(loc)) {
                    System.out.println(loc);
                    lastLoc = loc;
                }
                System.out.println("   " +
                    instr.getMnemonic().toUpperCase());
            }
        }
    }
}
JPF configurations

- JPF is very flexible = many attributes can be configured freely
  - Do not worry, there is a default for everything
  - For basic purposes the defaults usually suffice
  - Listeners, Search Heuristics, etc...

- JPF’s designers vision is that everything must be configurable (Eg - default.properties)
JPF configurations – Cont.

There are several methods to config. JPF

Per configuration file:

- search.listener = MySearchListener
- vm.listener = MyVMLListener
- jpf.listener = MyCombinedListener:MySecondListener...

Per command line:

- jpf ... +jpf.listener=MyCombinedListener ...

Hard coded:

- MyListener listener = new MyListener(..);
- Config config = JPF.createConfig( args);
- JPF jpf = new JPF( config);
- jpf. addSearchListener (listener);
- jpf. addVMLListener (listener);
- jpf.run();
State space explosion – The scalability problem

• Configurable search strategies
  – What is the best search heuristic?
  – Can be configured ad hoc

• Reducing the number of states
  – Partial Order Reduction (coming up soon…)
  – Heuristic Choice Generators (a bit late…)
  – Host VM Execution
  – State Abstraction

• Reducing state storage costs
POR - Partial Order Reduction

- Concurrent programs force us to encounter all possible interleaving – generates large models
- The number of scheduling induced states can be significantly reduced without inflicting on the program’s semantics
- JPF’s POR does not rely on user instrumentation nor static analysis
- JPF automatically determines at runtime which instructions have to be treated as state transition boundaries
• VM executes all instructions in the current thread until one of the following conditions is met:
  – the next instruction is scheduling relevant
  – the next instruction yields a "nondeterministic" result (i.e. simulates random value data acquisition)

• Each bytecode instruction type corresponds to a subclass that determines scheduling relevance based on the following factors:
• **Instruction Type**  
  – Only about 10% of the Java bytecode instructions are scheduling relevant, i.e. can have effects across thread boundaries  
  For example: Thread [start(), sleep(), yield(), join()] , wait(), notify().

• **Object Reachability**  
  – field access is the major type of interaction between threads. However, most of them don’t really have an affect on scheduling.  
  – This is an expensive operation, but most of the work is already done by the JVM’s GC.

• **Thread and Lock Information**  
  – Even if the instruction type and the object reachability suggest scheduling relevance, there is no need to break the current transition in case there is no other runnable thread.
JPF mechanisms that can inflict potential thread interleaving:

- **Attributor**
  - A configurable mechanism to classify objects, methods and field attributes into groups according to their relevance to scheduling.

- **VMLListener**
  - A listener can explicitly request a reschedule by calling ThreadInfo.yield() in response to a instruction execution notification.

- **Verify**
  - beginAtomic(), endAtomic() functions to control thread interleaving.
Choice Generators

- When you need JPF to try alternatives:
  Verify.getInt, getDouble, getBoolean, ...
- When JPF hits Verify.getXxx() it branches the execution tree and executes one branch for each value

```java
int x = Verify.getInt(-1, 1);
System.out.println(x);
[x == 0]
System.out.println(x);
[x == -1]
System.out.println(x);
[x == 1]
System.out.println(x);
```
Choice Generators config.

- Each CG has a symbolic name (e.g. “items”)
  - Identification of the CG’s configuration in a properties file
- Selection of integer numbers from a set (example)
  - Configuration:
    - items.class = IntChoiceFromSet
    - items.values = {-6, 1, 2, 5, 503, 250000}
  - Application in a test driver:
    - Int i = Verify.getInt("items");
  - User can change the values (or a CG class) without the need to recompile test drivers
- Heuristic selection of a floating-point number that is below, at, or above a specific threshold
  - speed.class= DoubleThresholdGenerator (extends DoubleChoiceGenerator)
    - speed.threshold= 0.01
    - speed.low= -2.5
    - speed.high= 6.0
- Application-specific heuristics (CG classes) can be used
import gov.nasa.jpf.jvm.Verify;

public class example2 {
    example2(){}
    int dodiv(int x, int y){
        int z;
        z=x/y;
        return z;
    }
    public static void main(String[] args) {
        int res;
        example2 ex = new example2();
        int arg2 = Verify.getInt(-10,10);
        res = ex.dodiv(10,arg2);
        System.out.println("result: "+res);
    }
}

**NOTE:**
This set-up causes JPF to go through all values of arg2 from -10 to 10. Should hit a div-by-0 exception somewhere...
Choice Generators – Example

.gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
   at example2.dodiv(example2.java:9)
   at example2.main(example2.java:24)

.gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
   [483 insn w/o sources]
   example2.java:19               : int arg2 = Verify.getInt(-10,10);

.gov.nasa.jpf.jvm.choice.IntIntervalGenerator[-10..10,delta=+1,cur=0]
   example2.java:19               : int arg2 = Verify.getInt(-10,10);
   example2.java:21               : example2 ex = new example2();
   example2.java:5                : example2()
   [1 insn w/o sources]
   example2.java:5                : example2()
   example2.java:21               : example2 ex = new example2();
   example2.java:24               : res = ex.dodiv(10,arg2);
   example2.java:9                : z=x/y;
Th-th-th-that's all folks!