Model Checking Programs with Java PathFinder

~

Part 2: Design

Willem Visser
<wvisser@email.arc.nasa.gov>

Peter Mehlitz
<pcmehlitz@email.arc.nasa.gov>

NASA Ames Research Center
Roadmap

✦ What is JPF?
✦ Motivating Examples
✦ How To Run It
  • invocation, configuration, distribution components
✦ Basic JPF Design
  • Search, VM
✦ Extending JPF
  • Listeners
  • Properties
  • MJI
✦ Underlying Mechanisms
  • Partial Order Reduction
✦ Future Infrastructure
  • ChoiceGenerator (yet another extension mechanism)
What is Java PathFinder (1)

✦ explicit state model checker for Java bytecode
✦ 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 JPF (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)
JPF Status

- developed at the Robust Software Engineering Group at NASA Ames Research Center
- currently in its 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
- 7500 downloads since publication 04/2005
Motivating Examples

✧ not your usual model checking examples
✧ oldclassic: concurrency (missed signal causing a deadlock)
✧ Crossing (AKA “stoned hippies”): JPF extension mechanisms
✧ K9 Rover: ‘real’ model, size (partial order reduction)
How To Run JPF

✧ generally speaking: like a VM ("java" replacement):
  > bin/jpf <jpf-options> <test-app main class>

✧ BUT: lots of configuration (classes) and parameterization (booleans, integers etc.)

✧ JPF is an open system

✧ need for flexible, extensible configuration system

✧ quite powerful, but can be somewhat confusing
JPF Configuration

class gov.nasa.jpf.Config {
  Object getInstance(key,type) throws Config.Exception
  Object getEssentialInstance(key,type) ..
  boolean getBoolean(key) ..
}

myheuristic.some_value=42
search.class=..HeuristicSearch
search.heuristic.class=MyHeuristic
vm.class=..JVM

java {-vm-arg..} gov.nasa.jpf.JPF [-c config-file] {+key=value..} [-show] main-class {app-arg..}

# section 1: general properties
log = warning
...

# section 2: Search properties
search.class = gov.nasa.jpf.search.DFSearch
...

# section 3: JVM properties
vm.class = gov.nasa.jpf.jvm.JVM
...

> java gov.nasa.jpf.JPF -c bfs.properties
+search.heuristic.class=MyHeuristic
+myheuristic.some_value=42
MyTestApp

# breadth first JPF configuration
search.class = \
  gov.nasa.jpf.search.heuristic.HeuristicSearch
search.heuristic.class = \
  gov.nasa.jpf.search.heuristic.BFSHeuristic

# default.properties in JPF root dir
- missing entry
- wrong type
- general exception

location
- command-line specified file
- or jpf.properties in JPF root dir
- or jpf.properties resource in jar
  (loaded via gov.nasa.jpf.JPF)

 ConfigException

- default.properties in JPF root dir
- or default.properties resource in jar
  (loaded via gov.nasa.jpf.JPF)
two major concepts: **Search** and **VM**

- **Search** is the VM driver and Property evaluator
- **VM** is the state generator
Under the Hood - Search

- `<Search>`
  - `search()`

- `<AbstractSearch>`
  - `vm`
  - `depth`
  - `listener`
  - `forward()`
  - `backtrack()`

- `DFSearch`
  - `search() {..}`

- `HeuristicSearch`
  - `queue`
  - `heuristic`
  - `search()`
  - `generateChildren()`
  - `expandState()`

- `<Heuristic>`
  - `int heuristicValue()`

- `BFSHeuristic`
  - `int heuristicValue() {..}`

- `end`
- `seen`
- `error`

- Sorted state queue
Under the Hood - VM

- `<VM>`
  - forward()
  - backtrack()

- JVM
  - backtrackStack
  - systemState
  - listener
  - forward()
  - backtrack()
  - isNewState()

- object model
  - Area
  - ElementInfo
  - Fields

- type model
  - ClassInfo
  - MethodInfo
  - FieldInfo

- execution model
  - ThreadList
  - ThreadInfo
  - Monitor
  - Instruction
  - AALOAD
  - ...

- state model
  - SystemState
  - KernelState
  - Scheduler
Extending JPF - Listeners

- preferred way of extending JPF: ‘Listener’ variant of the Observer pattern - keep extensions out of the core classes
- listeners can subscribe to Search and VM events
public interface SearchListener {

    /* got the next state */
    void stateAdvanced (Search search);

    /* state was backtracked one step */
    void stateBacktracked (Search search);

    /* a previously generated state was restored
     * (can be on a completely different path) */
    void stateRestored (Search search);

    /* JPF encountered a property violation */
    void propertyViolated (Search search);

    /* we get this after we enter the search loop, but BEFORE
     * the first forward */
    void searchStarted (Search search);

    /* there was some constraint hit in the search, we back out
     * could have been turned into a property, but usually is an
     * attribute of the search, not the application */
    void searchConstraintHit (Search search);

    /* we're done, either with or without a preceeding error */
    void searchFinished (Search search);
}

public interface VMLListener {

    void instructionExecuted (VM vm); // VM has executed next instruction

    void threadStarted (VM vm); // new Thread entered run() method

    void threadTerminated (VM vm); // Thread exited run() method

    void classLoaded (VM vm); // new class was loaded

    void objectCreated (VM vm); // new object was created

    void objectReleased (VM vm); // object was garbage collected

    void gcBegin (VM vm); // garbage collection mark phase started

    void gcEnd (VM vm); // garbage collection sweep phase terminated

    void exceptionThrown (VM vm); // exception was thrown

    void nextChoice (VM vm); // choice generator returned new value

}
public class HeapTracker

    extends GenericProperty implements VMLListener, SearchListener {
    class PathStat { .. int heapSize = 0; .. }    // helper to store additional state info

    PathStat stat = new PathStat();
    Stack pathStats = new Stack();

    public boolean check (VM vm, Object arg) {    // GenericProperty
        return (stat.heapSize <= maxHeapSizeLimit);
    }

    public void stateAdvanced (Search search) {    // SearchListener
        if (search.isNewState()) { ..
            pathStats.push(stat);
            stat = (PathStat)stat.clone(); ..
        }
    }

    public void stateBacktracked (Search search) {    // SearchListener
        .. if (!pathStats.isEmpty()) stat = (PathStat) pathStats.pop();
    }

    public void objectCreated (VM vm) { ..    // VMLListener
        ElementInfo ei = ((JVM)vm).getLastElementInfo();
        ..stat.heapSize += ei.getHeapSize(); ..
    }

    public void objectReleased (VM vm) {    // VMLListener
        ElementInfo ei = ((JVM)vm).getLastElementInfo();
        ..stat.heapSize -= ei.getHeapSize(); ..
    }

    ...
}
Extending JPF - Listener Configuration

- listeners are usually configured, not hard coded
- per configuration file:

```java
search.listener = MySearchListener
vm.listener = MyVMListener
jpf.listener = MyCombinedListener:MySecondListener...
```

- per command line:

```bash
jpf ... +jpf.listener=MyCombinedListener ...
```

- hard coded:

```java
MyListener listener= new MyListener(..);
.. 
Config config = JPF.createConfig( args);
JPF jpf = new JPF( config);
jpf. addSearchListener (listener);
jpf. addVMListener (listener);
jpf.run();
.. 
```
Extending JPF - Properties

- dedicated check objects that are configured into the Search (checked after each new transition)
- simple interface:

  ```java
  public interface Property extends Printable {
    boolean check (VM vm, Object arg);
    String getErrorMessage();
  }
  ```
- JPF includes a number of non-functional Property implementations:
  - NotDeadlockedProperty
  - NoUncaughtExceptionsProperty
- configured via search.properties:
  ```properties
  search.properties = gov.nasa.jpf.jvm.NotDeadlockedProperty:...:x.y.z.MyProperty
  ```
- generic way to introduce functional properties: Java assertions
  (`->AssertionError -> NoUncaoughtExceptionsProperty`)
- more complex: listeners that also implement Property (HeapTracker)
- even more complex: DynamicAssertions (“assertions with state” - check objects which are state tracked by JPF)
JPF is a state-tracking JVM, running on top of a general JVM

*Java Native Interface (JNI) analogy: “execute one level lower”*

*Model Java Interface (MJI): execute in the host VM that runs JPF itself*
MJI - Why?

- one obvious reason: running native Java methods in JPF (otherwise we couldn’t run apps using standard libraries, which have lots of native methods)
- more general: replace methods
- specific use of native methods: interface library methods to JPF runtime system (e.g. java.lang.Thread -> ThreadInfo)
- enables usage of specialized verification API in app, interfacing to JPF functionality:
  
  ```java
  int input = gov.nasa.jpf.jvm.Verify.randomInt(10);
  ```
- but also useful for scalability reasons
  - native methods can save state space
  - native methods are executed atomically
  - native methods execute much faster
- example: java.io.RandomAccessFile
MJI - Components

- **Model** class: has native method declaration, executed by JPF
- **NativePeer** class: native method implementation, executed by JVM
- **MJIEnv**: native method calling context (to get back to JPF)

```java
package x.y.z;

class MyClass {
    ...
    native String foo (int i, String s);
}

class JPF_x_y_z_MyClass {
    public static
    int foo__ILjava_lang_String__2 (MJIEnv env, int objRef,
        int i, int sRef) {
        String s = env.getStringObject(sRef);
        ...
        int ref = env.newString(..);
        return ref;
    }
}
```

"Model" Class

"NativePeer" Class

"Model Java Interface"
package x.y.z;
class C {
  ...
  native int foo (int p);
}

class JPF_x_y_z_C {
  ...
  public static int foo__I (MJIEnv env, int thisRef, int p) {
    int d = env.getIntField(thisRef, "data");
    ...
  }
  ...
}

int a = c.foo(3);
MJI - Example

● application calls method to intercept

```java
.. 0:   getstatic #2
System.out.println("a message"); 3:   ldc #3
.. 5:   invokevirtual #4
```

● model class declares the method we want to intercept (doesn’t have to be native), is executed by JPF

```java
public class PrintStream .. {
  ..
  public void println (String s) {..} // usually native method
}
```

● native peer has the method implementation that gets executed by host VM (not simulated by JPF)

```java
class JPF_java_io_PrintStream { ..
  public static void println__Ljava_lang_String_2 (MJIEnv env,int objRef,
           int strRef) {
    env.getVM().println(env.getStringObject(strRef));
  }
}
```
Scalability - Partial Order Reduction

✦ concurrency is major contributor to state space explosion

✦ reduction of thread interleavings is paramount for scalability

✦ JPF employs on-the-fly *Partial Order Reduction* mechanism

✦ leveled approach that makes use of JVM instruction set and infrastructure (memory management)

✦ completely at runtime (on-the-fly)
POR - Scheduling Relevance

bytecode insn

scheduling relevant insn type

field insn

GETFIELD
PUTFIELD
GETSTATIC
PUTSTATIC
xALOAD
xASTORE

sync insn

MONITORENTER
MONITOREXIT

invoke insn

INVOKEVIRTUAL
INVOKESTATIC

other runnable threads

recursive lock

shared object

GC based reachability analysis

lock protected access

lock distance based heuristics

scheduling relevant instruction
to detect races, we have to identify read/write access to objects that are visible from different threads

expensive operation, BUT: can piggyback on garbage collection

two phase approach:
- mark root set with thread id (statics are shared by default)
- traverse marked objects - if another thread id is reached, mark as shared

problem: GC based on reachability, not accessibility -> need to break on certain fields (Thread.group->ThreadGroup.threads)
two types of execution path variation in JPF: scheduling sequences and nondeterministic data values

nondeterministic input (test driver) via gov.nasa.jpf.jvm.Verify API

```java
int a = Verify.random(2);  // will execute for {0,1,2}
```

what about partial enumerations (app-specific double value sets)

```java
double velocity = Verify.getDouble("v"); // {v0, vT, v1}
```

scheduling has to cover runnable set and order - will become more challenging with Realtime Java (programmable, deterministic scheduler)

current gov.nasa.jpf.jvm.Scheduler implementations: not very extensible, has structural overhead (thread/random int values, intermediate kernel states, partial value sets have to be implemented in app)

goal: unify nondeterministic data and scheduling: ChoiceGenerators
ChoiceGenerators

```java
ChoiceGenerator gen;
KernelState ks;
thread, id
nextSuccessor();
KernelState
heap, stacks
SystemState
ChoiceGenerator gen
KernelState ks
thread, id
nextSuccessor();
<ChoiceGenerator>
advance()
hasMoreChoices()
<ThreadChoice>
nextThread()
<DoubleChoice>
nextDouble()
AllRunnables
thread set
enumeration state
DoubleThreshold
low, threshold, high
enumeration state
JVM
systemState
backtrackStack
forward()
backtrack()
```

gen.advance();
if (gen.hasMoreChoices()) {
    if (gen instanceof ThreadChoice)
        thread = ((ThreadChoice
            gen).nextThread();
    ..
    thread.executeTransition();
} ..

double v = Verify.getDouble("velocity");
..
velocity.class = DoubleThreshold
velocity.threshold = 13250
velocity.delta = 500