JavaPathFinder

Radek Mařík
K333, FEE, CTU, Prague
As a selection of slides from several JavaPathFinder tutorials
2013 November 26
Outline

* What is JPF
* Usage examples
* Test case generation
* JPF architecture
* Using JPF
Outline

* What is JPF
* Usage examples
* Test case generation
* JPF architecture
* Using JPF
Introduction - What Is JPF?

- surprisingly hard to summarize - can be used for many things
- extensible virtual machine framework for Java bytecode verification: workbench to efficiently implement all kinds of verification tools

![Diagram showing JPF components: System under Test (Java bytecode), abstract virtual machine, JPF core, JPF extension, verification artifact.]

- typical use cases:
  - software model checking (deadlock & race detection)
  - deep inspection (numeric analysis, invalid access)
  - test case generation (symbolic execution)
verified Java program is executed by JPF, which is a virtual machine implemented in Java, i.e. runs on top of a host JVM

⇒ easy to get confused about who executes what
not a new project: around since 10 years and continuously developed:

- 1999 - project started as front end for Spin model checker
- 2000 - reimplementation as concrete virtual machine for software model checking (concurrency defects)
- 2003 - introduction of extension interfaces
- 2005 - open sourced on Sourceforge
- 2008 - participation in Google Summer of Code
- 2009 - moved to own server, hosting extension projects and Wiki
Who is using JPF?

- major user group is academic research - collaborations with >20 universities worldwide (uiuc.edu, unl.edu, byu.edu, umn.edu, Stellenbosch Za, Waterloo Ca, AIST Jp, Charles University Prague Cz, ..)
- companies not so outspoken (exception Fujitsu - see press releases, e.g. http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html), but used by several Fortune 500 companies
- lots of (mostly) anonymous and private users (~1000 hits/day on website, ~10 downloads/day, ~60 read transactions/day, initially 6000 downloads/month)
- many uses inside NASA, but mostly model verification at Ames Research Center
JPF’s Home

http://babelfish.arc.nasa.gov/trac/jpf

JPF’s User Forum

http://groups.google.com/group/java-pathfinder
Where to learn more - the JPF-Wiki

http://babelfish.arc.nasa.gov/trac/jpf

- public read access
- edit for account holders (also non-NASA)

bug tracking
- Trac ticket system

project blog
- announcements
- important changes

hierarchical navigation menu
- intro
- installation
- user docu
- developer docu
- extension projects
Introduction: Key Points

- JPF is research platform and production tool (basis)
- JPF is designed for extensibility
- JPF is open source
- JPF is an ongoing collaborative development project
- JPF cannot find all bugs
  - but as of today -
  - some of the most expensive bugs only JPF can find
- JPF is moderately sized system (~200ksloc core + extensions)
- JPF represents >20 man year development effort
- JPF is pure Java application (platform independent)
Outline

* What is JPF
* **Usage examples**
* Test case generation
* JPF architecture
* Using JPF
Examples

goal is to show gamut of possible applications, not detailed defect understanding

- software model checking (SMC) of production code
  - data acquisition (random, user input)
  - concurrency (deadlock, races)

- deep inspection of production code
  - property annotations (Const, PbC,..)
  - numeric verification (overflow, cancellation)

- model verification
  - UML statecharts

- test case generation

- verification of distributed application
Application Types

**JPF unaware programs**
- runs on any JVM

**JPF enabled programs**
- "sweet spot"
- \@V
- \*.java

**JPF dependent programs**
- runs only under JPF

### Constraints
- **runtime costs**
  - order of magnitude slower
  - state storage memory
- **standard library support**
  - java.net, javax.swing, ...
  - needs abstraction models
- **functional property Impl. costs**
  - listeners, MJi knowledge
- **restricted choice types**
  - scheduling sequences
  - java.util.Random

### Benefits
- **non-functional properties**
  - unhandled exceptions (incl. AssertionError)
  - deadlocks
  - races
- **Improved Inspection**
  - coverage statistics
  - exact object counts
  - execution costs
- **low modeling costs**
  - statemachine w/o layout hassle...
- **functional (domain) properties**
  - built-in into JPF libraries
- **flexible state space**
  - domain specific choices (e.g. UML "enabling events")
- **runtime costs & library support**
  - usually not a problem, domain libs can control state space

### Initial domain impl. costs
- domain libs can be tricky

### Restricted application models
- UML statemachines
- does not run w/o JPF libraries

---

Wednesday, February
Model Checking vs. Testing

- SMC is a rigorous formal method
- SMC does not suffer from false positives (like static analysis)
- SMC provides traces (execution history) when it finds a defect

**Testing:**
- Based on input set \{d\}
- Only one path executed at a time

**Model Checking:**
- All program states are explored until none left or defect found

Diagram details:
- Testing: Path through states
- Model Checking: Backtrack and match
Model Checking vs. Testing/Simulation

- Model individual state machines for subsystems / features
- Simulation/Testing:
  - Checks only *some* of the system executions
  - May miss errors
- Model Checking:
  - Automatically combines behavior of state machines
  - *Exhaustively* explores all executions in a systematic way
  - Handles millions of combinations – hard to perform by humans
  - Reports errors as traces and simulates them on system models
Example 1: Nondeterministic Data (2)

- Testing only covers one execution

```
1. Random random = new Random();
2. int a = random.nextInt(2);
3. int b = random.nextInt(3);
4. int c = a/(b+a -2)  // java Rand
   a=1
   b=0
   c=-1
```

- Model checking (theoretically) covers all executions

```
> jpf +cg.enumerate_random=true
Rand
   a=0
   b=0
   c=0
   b=1
   c=0
   b=2
   c=0/0
   x
   c=-1
   c=1/0
   c=1
   ERROR: ArithmeticException
```
Example 8: Programming-by-Contract

✧ property annotations can be parameterized with expressions
✧ can be a lot more complex
✧ inheritance aware
✧ expr evaluation has no SUT side-effects

```java
class Base {
    @Ensures("Result > 0")
    public int compute (int c){ return c - 1; }
    ...
    @Invariant({"d within 40 +- 5", "a > 0" })
    public class Derived extends Base {
        double d; int a; // not checked until return from ctor
        ContractViolation() { a = 42; d = 42; }
        @Requires("c > 0")
        @Ensures("Result >= 0")
        public int compute (int c){
            return c - 3;
        }
        public void doSomething (int n){
            for (int i=0; i<n; i++){
                d += 1.0;
            }
        }
    }
}
... Derived t = new Derived();
//int n = t.compute(3); // would violate strengthening base postcondition
t.doSomething(10); // violates 'd' invariant
```

jpf-aprop/src/examples/ContractViolation.jpf
Outline

- What is JPF
- Usage examples
- Test case generation
- JPF architecture
- Using JPF
Examples: Test Case Generation

- test creation is very expensive - how many tests do we need (instruction coverage, branch coverage, MCDC, path coverage ..)
- the application for symbolic execution
- “normal” verification - use data to find out which paths to explore
- problem with huge input spaces (what test data is interesting?)
- symbolic execution uses opposite direction - use program structure to deduce which data values need to be tested to reach interesting program locations (exceptions, coverage requirements)
how do we identify interesting parameter values if they are not known a priori (example 13 @PARAMS annotation)?

```c
@PARAMS(5000 | 120000 | 200000)

void abort(altitude){

    ...
    if (altitude <= 1.2e5)
        setState(stateabortLowActive)
    ...
    if (altitude >= 1.2e5)
        setState(stateabortHighActive)

```

answer: test case generation with *Symbolic Execution*
Symbolic Execution

• King [Comm. ACM 1976], Clarke [IEEE TSE 1976]
• Analysis of programs with unspecified inputs
  – Execute a program on symbolic inputs
• Symbolic states represent sets of concrete states
• For each path, build a path condition
  – Condition on inputs – for the execution to follow that path
  – Check path condition satisfiability – explore only feasible paths
• Symbolic state
  – Symbolic values/expressions for variables
  – Path condition
  – Program counter
```c
int x, y;
if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert false;
}
```

**Concrete Execution Path**

```
x = 1, y = 0
1 > 0 ? true
x = 1 + 0 = 1
y = 1 - 0 = 1
x = 1 - 1 = 0
0 > 1 ? false
```
Example – Symbolic Execution

Code that swaps 2 integers:

```java
int x, y;
if (x > y) {
    x = x + y;
y = x - y;
x = x - y;
if (x > y)
    assert false;
}
```

Symbolic Execution Tree:

- **Path condition**

```
[PC: true] x = X, y = Y
```

- **False!**

```
[PC: true] x = X, y = Y
```

```
[PC: true] X > Y?
```

```
[PC: true] X > Y?
```

```
[PC: X > Y] x = X + Y
```

```
[PC: X > Y] y = X + Y - Y = X
```

```
[PC: X > Y] x = X + Y - X = Y
```

```
[PC: X > Y] Y > X?
```

```
[PC: X > Y] Y > X?
```

```
[PC: X > Y ∧ Y ≤ X] END
```

```
[PC: X > Y ∧ Y > X] END
```

- **False!**

Solve path conditions → test inputs
Example 14: Symbolic Execution

- look for instruction that constitutes property violation and compute data that leads to it

```java
void abort(altitude, controlMotorFired){
    ...
    if (altitude <= 1.2e5)
        setState(abortLowActive)
    ...
    if (altitude >= 1.2e5)
        setState(abortHighActive)
    ...
}

symbolic.dp = choco
Symbolic Execution Mode
...
***Execute symbolic INVOKEVIRTUAL: abort(I)Z
    (altitude_1_SYMINT, controlMotorFired_CONCRETE )
...
Property Violated: PC is # = 2
altitude_1_SYMINT[120000] >= CONST_120000 &&
altitude_1_SYMINT[120000] <= CONST_120000
...
Property Violated: result is "java.lang.AssertionError: ambiguous transitions in: ascent.firstStage..."
...
========================================================================== Method Summaries
Symbolic values: altitude_1_SYMINT
abort(120000, true) -- "java.lang.AssertionError: ambiguous transitions in: ascent.firstStage..."
```
Example 15: Test Case Generation

Symbolic Execution especially suitable to generate test suite to achieve path coverage

```java
public class TestPaths {
  ...
  // what tests do we need to cover all paths?
  public static void testMe (int x, boolean b) {
    if (x <= 1200) {
      // BLOCK-1
      if (x >= 1200) {
        // BLOCK-2
      }
    }
  }
  ...
}
```

***Execute symbolic INVOKESTATIC: testMe(IZ)V ( x_1_SYMINT, b_CONCRETE )

PC # = 2

- `x_1_SYMINT[1200]` >= `CONST_1200` && `x_1_SYMINT[1200]` <= `CONST_1200`
- `x_1_SYMINT[1000000]` < `CONST_1200` && `x_1_SYMINT[-1000000]` <= `CONST_1200`

Method Summaries

- `testMe(1200, true)`  BLOCK-1, BLOCK-2
- `testMe(-1000000, true)`  BLOCK-1
- `testMe(1201, true)`  BLOCK-2

---

Wednesday, February
Symbolic PathFinder

- JPF-core’s search engine used
  - To generate and explore the symbolic execution tree
  - To also analyze thread inter-leavings and other forms of non-determinism that might be present in the code
- No state matching performed – some abstract state matching
- The symbolic search space may be infinite due to loops, recursion
  - We put a limit on the search depth
- Off-the-shelf decision procedures/constraint solvers used to check path conditions
  - Search backtracks if path condition becomes infeasible
- Generic interface for multiple decision procedures
  - Choco (for linear/non-linear integer/real constraints, mixed constraints), http://sourceforge.net/projects/choco/
  - IASolver (for interval arithmetic) http://www.cs.brandeis.edu/~tim/Applets/IASolver.html
  - Other constraint solvers: HAMPI, randomized solvers for complex Math constraints – work in progress
Implementation

• SPF implements a non-standard interpreter of byte-codes
  – To enable JPF-core to perform symbolic analysis
  – Replaces or extend standard concrete execution semantics of byte-codes with non-standard symbolic execution

• Symbolic information:
  – Stored in attributes associated with the program data
  – Propagated dynamically during symbolic execution

• Choice generators:
  – To handle non-deterministic choices in branching conditions during symbolic execution

• Listeners:
  – To print results of symbolic analysis (path conditions, test vectors or test sequences); to influence the search

• Native peers:
  – To model native libraries, e.g. capture Math library calls and send them to the constraint solver
Handling Branching Conditions

• Symbolic execution of branching conditions involves:
  – Creation of a non-deterministic choice in JPF’s search
  – Path condition associated with each choice
  – Add condition (or its negation) to the corresponding path condition
  – Check satisfiability (with Choco, IASolver, CVC3 etc.)
  – If un-satisfiable, instruct JPF to backtrack

• Created new choice generator

```java
public class PCChoiceGenerator 
    extends IntIntervalGenerator { 
    PathCondition[] PC; 
    ... 
}
```
Outline

- What is JPF
- Usage examples
- Test case generation
- JPF architecture
- Using JPF
verified Java program is executed by JPF, which is a virtual machine implemented in Java, i.e. runs on top of a host JVM
⇒ easy to get confused about who executes what
all JPF projects share uniform directory layout

binary distributions are slices of source distributions (interchangeable)

3rd party tools & libraries can be included (self-contained)

all projects have examples and regression test suites (eventually 😊)

projects have out-of-the-box IDE configuration (NB, Eclipse)
two major constructs: Search and JVM
- JVM produces program states
- Search is the JVM driver
Search Policies

- state explosion mitigation: search the interesting state space part first ("get to the bug early, before running out of memory")
- Search instances encapsulate (configurable) search policies
transitions begin with a choice and extend until the next ChoiceGenerator (CG) is set (by instruction, native peer or listener)

‘advance’ positions the CG on the next unprocessed choice (if any)

‘backtrack’ goes up to the next CG with unprocessed choices
Configurable Choice Generators - Why?

- `Verify.getBoolean()`: \( C = \{ \text{true, false} \} \)  
- `Verify.getInt(0, 4)`: \( C = \{ 0, 1, 2, 3, 4 \} \)  
- `Verify.getDouble(1.0, 1.5)`: \( C = \{ \infty \} \)

- `xChoiceGenerator`
  - `choiceSet: \{x\}`
  - `hasMoreChoices()`
  - `advance()`
  - `getNextChoice() \rightarrow x`

- **Choice Generators**
  - JPF internal object to store and enumerate a set of choices

- **Configurable Heuristic Choice Models**
  - Configurable classes to create ChoiceGenerator instances

- **Example**: "Threshold" heuristic
  - \( T \pm \Delta \)
  - \( C = \{ T-\Delta, T, T+\Delta \} \)

- Application code (test driver)
  - ```java
  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`
Listeners - the JPF plugins

- classLoaded()
- threadScheduled()
- threadNotified()
  ...
- executeInstruction()
- instructionExecuted()
- objectCreated()
  ...
- exceptionThrown()
  ...
- choiceGeneratorAdvanced()
  ...

- +listener=<listener-class>
- @JPFConfig(..)
- listener.autoload=<annotations>
- jpf.addListener(..)
  ...

- propertyViolated()
- searchContraintHit
- searchFinished()
Outline

* What is JPF
* Usage examples
* Test case generation
* JPF architecture
* Using JPF
Using JPF

- obtaining JPF
- installing, building and testing JPF
- JPF configuration
- running JPF
- JPF and NetBeans
- JPF and Eclipse
Obtaining JPF

- Mercurial repositories on 
  http://babelfish.arc.nasa.gov/hg/jpf/{jpf-core,jpf-aprop,...}

- Steps
  1. obtain and install Mercurial (http://mercurial.selenic.com)
  2. clone required JPF repositories
     
     >hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
     >hg clone .../jpf/jpf-numeric
     ...

- don’t clone whole .../hg/jpf directory!
- don’t use https://.. (only required for push)

- alternative: download binary distributions from wiki project page
  attachments (e.g. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-core) - not recommended at this time
Installing, Building and Testing JPF

- create `site.properties` file in `~/.jpf` directory
  (see http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/site-properties, required so that JPF can find jpf-core and installed projects without extensive manual classpath configuration)

  ```
  jpf-core = ~/.jpf/projects/jpf/jpf-core
  ...
  jpf-numeric = ~/.jpf/projects/jpf/jpf-numeric
  extensions+=,~/.jpf-numeric
  ...
  ```

- build & test jpf-core and required extensions
  (with included Ant builder)

  ```
  > cd jpf-core
  > bin/ant clean test
  ...
  > cd ../jpf-numeric
  > bin/ant clean test
  ...
  ```

  ```
  $ bin/ant clean test
  Buildfile: .../jpf-core/build.xml
  clean:
    [delete] Deleting directory .../jpf-core/build
  -init:
    [mkdir] Created dir: .../jpf-core/build
  ...
  build:
    [jar] Building jar: .../jpf-core/build/jpf.jar
  test:
    [junit] Running NameTests
    [junit] Tests run: 1, Failures: 0, Errors: 0...
    ...
  BUILD SUCCESSFUL
  Total time: 1 minute 30 seconds
```
almost nothing in JPF is hardwired ⇒ great flexibility but config can be intimidating

all of JPFs configuration is done through Java properties (but with some extended property file format)

- keyword expansion $jpf-root = ${user.home}/jpf
  - previously defined properties
  - system properties
- append $extensions+=,jpf-aprop
- prepend +peer_packages=jpf-symbc/build/peers,
- directives
  - dependencies $requires jpf-awt
  - recursive loading @include ..*/jpf-symbc/jpf.properties

hierarchical process

- system defaults (from jpf.jar)
- site.properties
- project properties from all site configured projects (<project-dir>/jpf.properties)
- current project properties (/jpf.properties)
- selected application properties file (*.jpf)
- command line args (e.g. bin/jpf +listener=.listeners.ExecTracker ...)
JPF Configuration (2)

Config object:

```java
.. myheuristic.some_value=42
search.class=../BFSHeuristic
vm.class=../JVM
..
```

```
java -jar ..RunJPF.jar {+key=value..} [-show] [-log] <application>.jpf
```

command line properties

application properties

project properties

site properties

default properties

> java -jar ..RunJPF.jar

```java
+myheuristic.some_value=42
MyTestApp.jpf
```

target = MyTestApp
target_args = ..
search.class = \
  gov.nasa.jpf.search.heuristic.BFSHeuristic

jpf-awt.nativo_classpath = build/jpf-awt.jar..
jpf-awt.classpath = build/jpf-awt-classes.jar..
jpf-awt.sourcepath = src/classes..
...

jpf-core = ${user.home}/projects/jpf/jpf-core

```
jpf-awt = ${user.home}/projects/jpf/jpf-awt
  extensions+=,${jpf-awt}
...
```

vm.class = gov.nasa.jpf.jvm.JVM
search.class = gov.nasa.jpf.search.BFSsearch
..
Running JPF

✦ for purists (tedious, do only if you have to)
  • setting up classpaths  
    >>> export CLASSPATH=.../jpf-core/build/jpf.jar...
  • invoking JVM  
    >>> java.gov.nasa.jpf.JPF +listener=... x.y.MySUT

✦ using site config and starter jars (much easier and portable)
  • explicitly  
    >>> java -jar tools/RunJPF.jar MySUT-verify.jpf
  • using scripts  
    >>> bin/jpf MySUT-verify.jpf

✦ running JPF from within JUnit
✦ running JPF from your program (tools using JPF)

✦ using NetBeans or Eclipse plugins
  • “Verify..” context menu item for selected * . jpf application property file
  • using provided launch configs (Eclipse) or run targets (NetBeans)
Running JPF from NetBeans

✦ use project provided run/debug tasks (requires nbproject/ide-file-targets.xml in project)
  • select *.jpf file in projects view
  • invoke Run→Run File from menubar (not in context menu)
  • results in Output view

✦ use NetBeans JPF plugin from http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/netbeans-jpf
  • download & install attached *.nbm if you don’t want to build
  • optionally install jpf-shell extension if you want JPF to run in own window
  • launch JPF by selecting *.jpf file and invoking “Verify.” context menu item
Running JPF from Eclipse

- use project provided launch configuration (requires `eclipse/run-JPF.launch` in project)
  - select *.jpf file in projects view
  - invoke Run As → Run Configurations → run-JPF from context menu
  - results in Output view

- use Eclipse JPF plugin from [http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/eclipse-jpf](http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/eclipse-jpf)
  - install from update site if you don’t want to rebuild [http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/install/eclipse-plugin/update/](http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/install/eclipse-plugin/update/)
  - optionally install jpf-shell extension if you want JPF to run in own window
  - launch JPF by selecting *.jpf file and invoking “Verify…” context menu item
Using JPF Shell

- sometimes JPF reports too long for IDE Output view
- some applications / properties might require specific visualization
- solution: configured JPF shell
- specify shell class and views in your application property file (*.jpf)
- launch normally through "Verify" context menu
- JPF shell shows as separate window
- sources can be shown in IDE editors by clicking links in JPF shell window

```
# JPF application property file using generic JPF shell
target=oldclassic

shell=.shell.basicshell.BasicShell
shell.panels+=,searchgraph
```

Wednesday, February
Why Using JPF Shell?

- run SUT (test)
- run JPF (verify)
- browse complex reports
- browse report topics
- configurable view tabs
- clickable source links
- inspect JPF run
- application/domain specific views

```
.. shell.panels+=,searchgraph
```

```
shell=.shell.awt.UTShell
shell.panels+=,component,trace,script
```