JPF-Inspector

P. Jančík, P. Parízek, J. Kofroň

http://d3s.mff.cuni.cz
Motivation
print
Example02.divide() - Example02.java:33 - assert (j != 0);
Stack slots
  0 : this (Example02) = Example02@13d
  1 : j (int) = 0
  2 : i (int) = 10

INFO: SuT is stopped
  SuT (Thread=0) executes the Example02.java:32 - iinc source: j--;

INFO: SuT is stopped
  SuT (Thread=0) executes the Example02.java:30 - bipush source: int i = 10;

print j
  1 : j (int) = 1

set j = 11
Value set successfully.
print j
  1 : j (int) = 11
Non-answered questions?

How it works?

Why JPF-Inspector?
Java PathFinder

- Java Virtual Machine
  - execute bytecode
  - stores heap, stacks, threads, ...

- Explicit state model checker
  - forward/backward steps
  - models environment
JPF-Inspector

- “Debugger” on top of JPF
  - drives JPF - state space traversal
  - reads/modifies JPF state

- Concurrency errors
  - JPF → all thread scheduling

in combination
Possible usage

- Error trace analyses
  - understand
  - check similar traces
  - root causes of bugs

- State explosion
  - find causes
  - drive state space exploration

- Debugging custom extensions
JPF-Inspector features

- Single stepping
  - backward/forward
  - step in, step out, step over, step instruction

- Breakpoints
  - line, field access, exception thrown
  - conditional – program state
  - thread scheduled, object created, ...
JPF-Inspector features

- Program non-determinism
  - modify thread scheduling

- State inspection/modification
  - print/set values of variables

- Save/Re-execute commands
  - log of executed commands
  - “memento” of program state
What the screenshot does (not) show?

Too many (manual) work
→ automatize

Usability of JPF on real word programs
→ focus on unit tests
Similar tools

- **Visual Studio**
  - Log of previous states

- **GDB**
  - Record & replay → read-only view

- **VMware - Reply Debugging**
  - Snapshots of VM & reexecution
  - “Random” thread scheduling

- **Chess**
  - Explores different thread scheduling
  - Reexecutes error trace
Future

- **Usability**
  - Integration to IDE
  - User friendly commands

- **Error root causes**
  - scheduling relevant errors
  - framework for error causes detection algs.

- **Support for JPF extensions**
  - Dumping state of extension
  - Symbolic JPF
Ongoing work – student tasks

- Java Platform Debugging Architecture
  - Integrate JPF-Inspector to IDE
  - Basic debugger functions
    → usability

- State comparison (Bachelor thesis)
  - Store & compare different program states
  - Exporting state
    → roots of state explosion
Error is found

1) Analyze error trace
   - Get possible causes of error

2) Classify error cause
   - Fixed set of error types
   - Fixed set of common solutions

3) Check solution
   - Error removed

4) Report result to user
Analyze concurrency errors

1. Triggering
   - buggy thread interleaving

2. Propagation
   - data-flow
   - control-flow

3. Failure
   - common patterns

```
a = new ...
a = null;
x = a;
y = x;
z = y;
z.doSomething();
```
Thank you for your attention!

Questions?

Ideas?

Suggestions?