

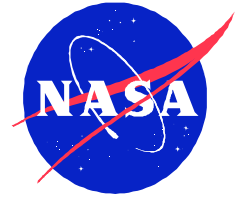
JPF for Beginners

David Bushnell

david.h.bushnell@nasa.gov

JPF Workshop 2008

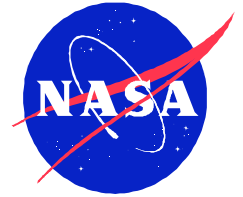
What is JPF?



- *An explicit state model checker*
 - Focus is on finding bugs in Java programs
- *A framework for runtime Java verification*
 - Model checking
 - Symbolic execution
 - UML state chart modeling
 - Numeric Verification (int overflow, fp over/underflow, ...)
 - ... *ad infinitum*

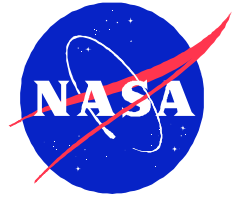


What is Model Checking?

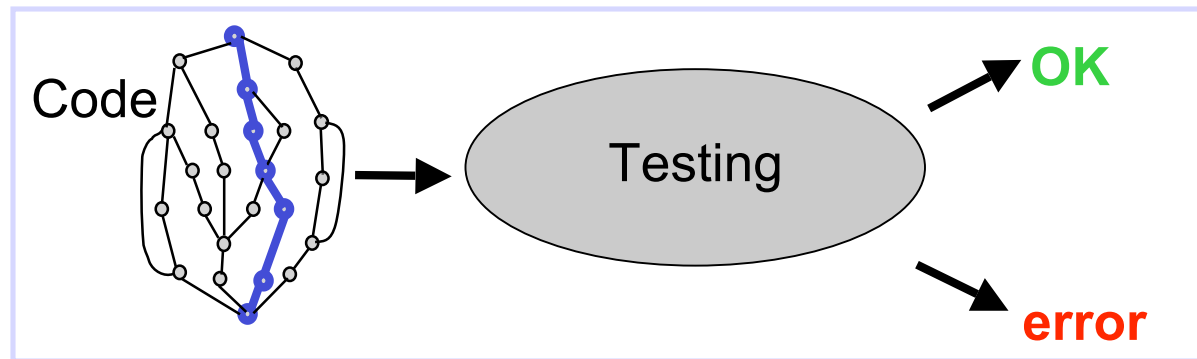


- Systematically verifying that a model satisfies a set of properties
 - Formal Model: UML state charts, Java programs, Promela models, ...
 - Properties: Temporal Logic (xTL), code assertions, ...
- In JPF:
 - The models are Java programs
 - The properties can be assertions, `gov.nasa.jpf.Property` objects, or JPF listener objects

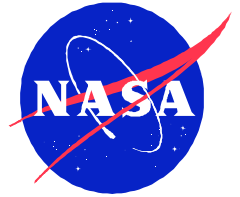
Model Checking vs Testing



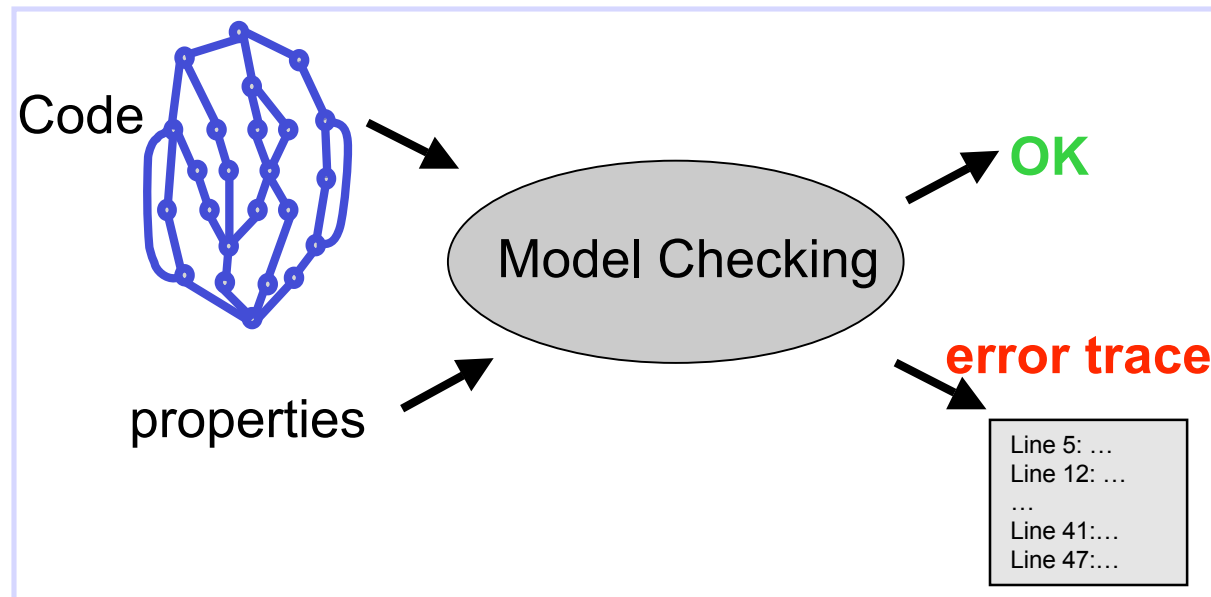
- A test will explore a single execution path
 - *You* must identify each important execution path
 - *You* must find the inputs that will execute those paths.



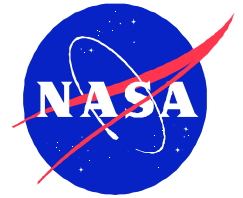
Model Checking vs Testing



- 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



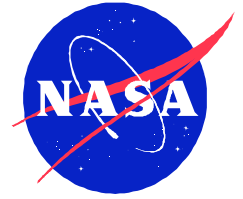
What Can JPF Handle?



- Pure Java up to ??-KLOC
 - Depends on logical complexity and state size, not KLOC.
 - Programs with 100K+ lines have been analyzed
- Multi-threaded code (*Of Course!*)
- Can find: deadlocks, race conditions, unhandled exceptions, application-specific assertions, ...

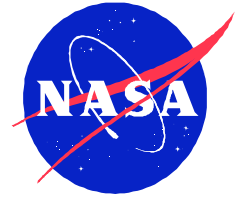


What Can't JPF Handle?



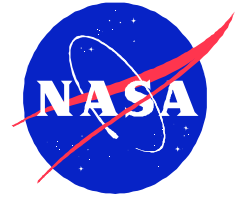
- *Unsupported* native calls (JNI)
 - Can simulate/support native calls with MJJ
- Hence: No libraries with unsupported native calls
 - Much or all of java.io, java.net, AWT, Swing, ...
- Really complex programs
 - But: it is often enough to apply JPF to a simplified version, AKA a *model*.
 - Example: apply JPF to a communications protocol used in your program

Using JPF



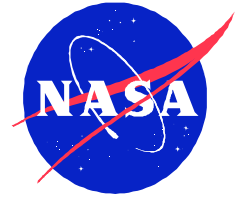
- Installing JPF
- Using JPF in an Eclipse project
- Configuring JPF
- Common Config Options
 - Example: Running JPF, detecting race conditions
- Controlling JPF Execution
 - Example: Detecting deadlock
- Extensions
- Listeners
 - Example: OpCodePrinter
- Overriding Bytecodes
 - Example: Numerics

Installing JPF for Eclipse



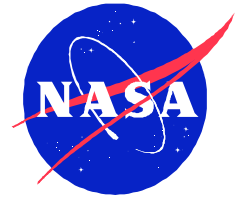
- Not covered: using JPF with other IDEs or from the command line
 - See documentation at SourceForge
- Prerequisites:
 - JDK 1.5+
 - Eclipse 3.2+ (www.eclipse.org)
 - Subclipse plugin (subclipse.tigris.org)

Installing JPF for Eclipse (2)



- Downloading JPF in Eclipse:
 - Create a new project (*not* a Java project)
 - Use the SVN ◊ “Checkout Projects from SVN” wizard
 - Repository URL:
<https://javapathfinder.svn.sourceforge.net/svnroot/javapathfinder>
 - Select “trunk” as your folder, *not* “tags” or “branches”
 - Anything for project name
 - Use defaults for everything else
 - Can have many copies of JPF, each as a different Eclipse project

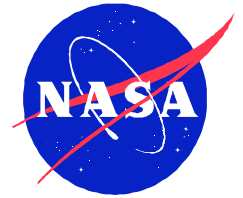
Configuring JPF



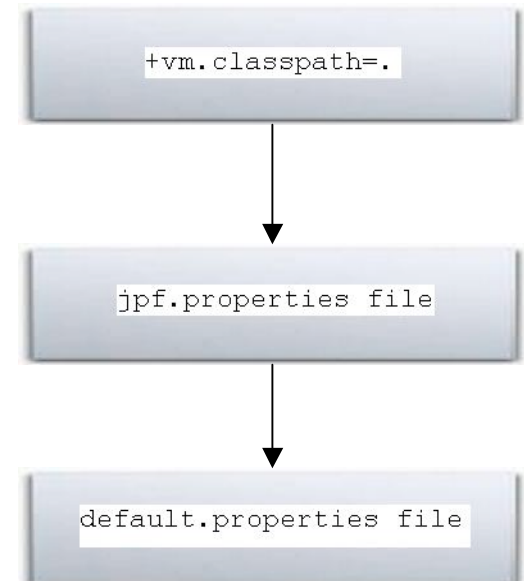
- **Bad News:** JPF has *lots* of config options
- **Good News:** The defaults are mostly ok. You seldom need to set more than 4 or 5.



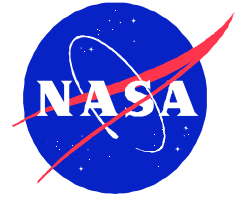
Configuring JPF



- Config hierarchy:
 - Command line args
(written as **+vm.classpath=.**)
take precedence over
 - jpf.properties values
(written as **vm.classpath=.**)
take precedence over
 - default.properties values
- Command line trick: comment out config options with **+_some.config.option=...**



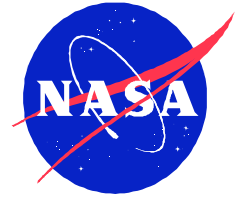
Configuring JPF (2)



- Rules:
 - *Never change* default.properties
 - jpf.properties is for values common to a project. Copy it to your project's top-level directory and change your copy (*do not change* the jpf.properties file in your JPF project)
 - Set the command line args for values specific to a single run
- In practice, in Eclipse:
 - Ignore jpf.properties.
 - Set everything with the command line using Eclipse's launch configurations
- For details on most core config properties, look in default.properties and jpf.properties

Configuring JPF

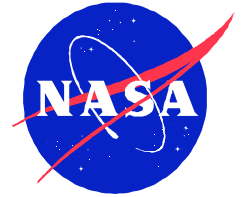
(3)



- Common config properties
 - `jpf.basedir`
 - Where JPF is located
 - `vm.classpath`
 - Where your compiled code is located, a classpath.
 - Usually set to “.”, i.e. **`vm.classpath=.`**
 - `vm.sourcepath`
 - Where your source code is located.
 - Defaults to **`vm.classpath`**, so you don’t usually need to set it
 - `search.class`
 - The search strategy to use (a class).
 - Defaults to **`gov.nasa.jpf.search.DFSearch`**
 - Look in **`src/gov/nasa/jpf/search`** for others

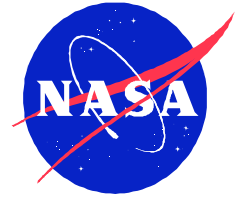
Configuring JPF

(4)

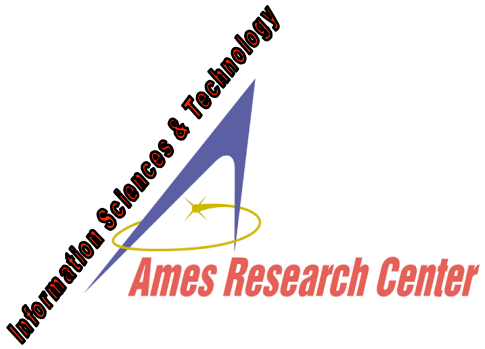


- Some other common config properties
 - `vm.storage.class`
 - Class used to hash/store states (if not set, states are not matched)
 - For small problems, can set to empty,
`vm.storage.class=`
 - `search.multiple_errors`
 - **true/false:** Quit after the first error or keep going?
 - `jpf.report.xxx`
 - Lots of options to configure the reporting subsystem
See `default.properties`
 - `jpf.report.console.finished`
 - What to report when JPF exits. Defaults to some statistics.
 - `jpf.report.console.show_code`
 - **true/false:** Show the bytecode in error traces?
 - `jpf.listener`
 - A “:” separated list of Listener classes

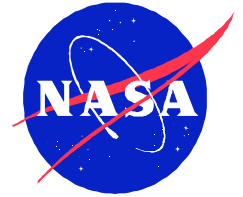
Using JPF in Eclipse



- Create an Eclipse Java project
- Write your Java code
- Create an Eclipse run configuration that:
 - Has **gov.nasa.jpj.JPJ** as its “Main class”
 - Has the right JPF config args
 - Has your JPF project in its classpath



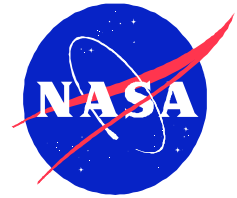
Running JPF Race Detection



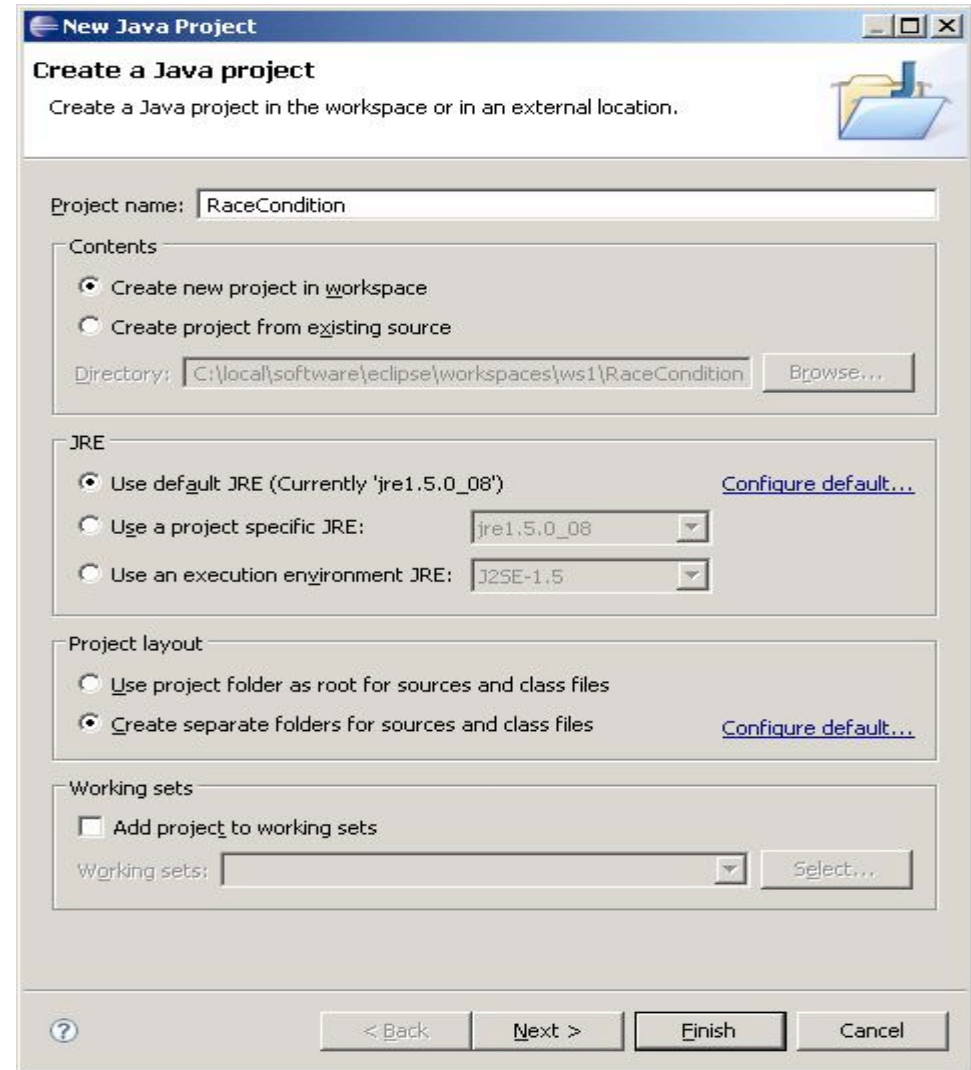
DEMO

MyRaceCondition

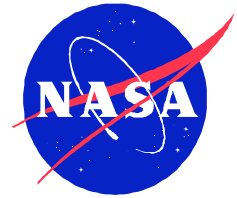
Create an Eclipse Project (1)



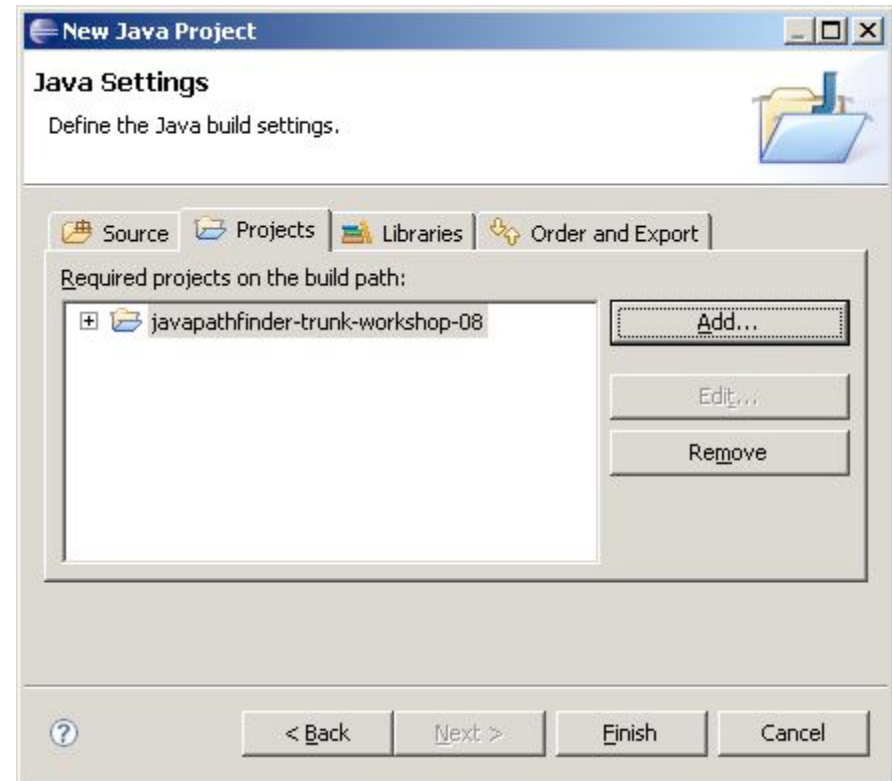
- Create a Java Eclipse project



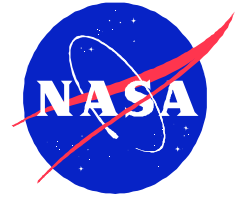
Create an Eclipse Project (2)



- Add your JPF project to the Java build settings



Write Your Java Code



```
public class MyRaceCondition {
    private static class Pair {
        String x = "x";
        String y = "y";

        public void update() {
            x = x + y + x;
        }
    }

    private static class RC extends Thread
    {
        Pair p;

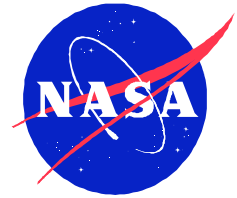
        public void run() {
            p.update();
        }
    }
}
```

```
public static void main(String[] args) {
    Pair p = new Pair();
    RC rc1 = new RC();
    RC rc2 = new RC();

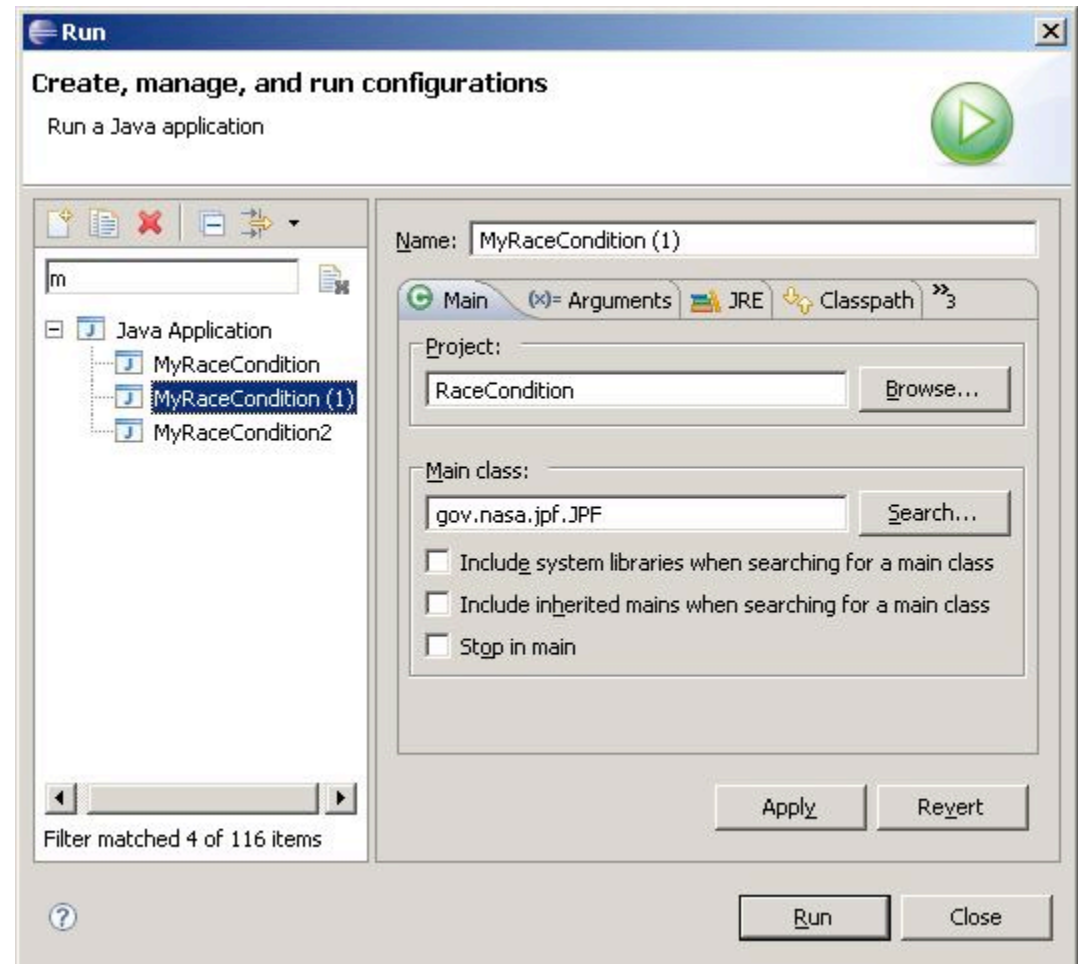
    rc1.p = p;
    rc2.p = p;

    rc1.start();
    rc2.start();
    rc1.join();
    rc2.join();
    System.out.println("x: " + p.x);
}
```

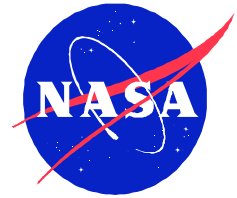
Create Eclipse Run Config (1)



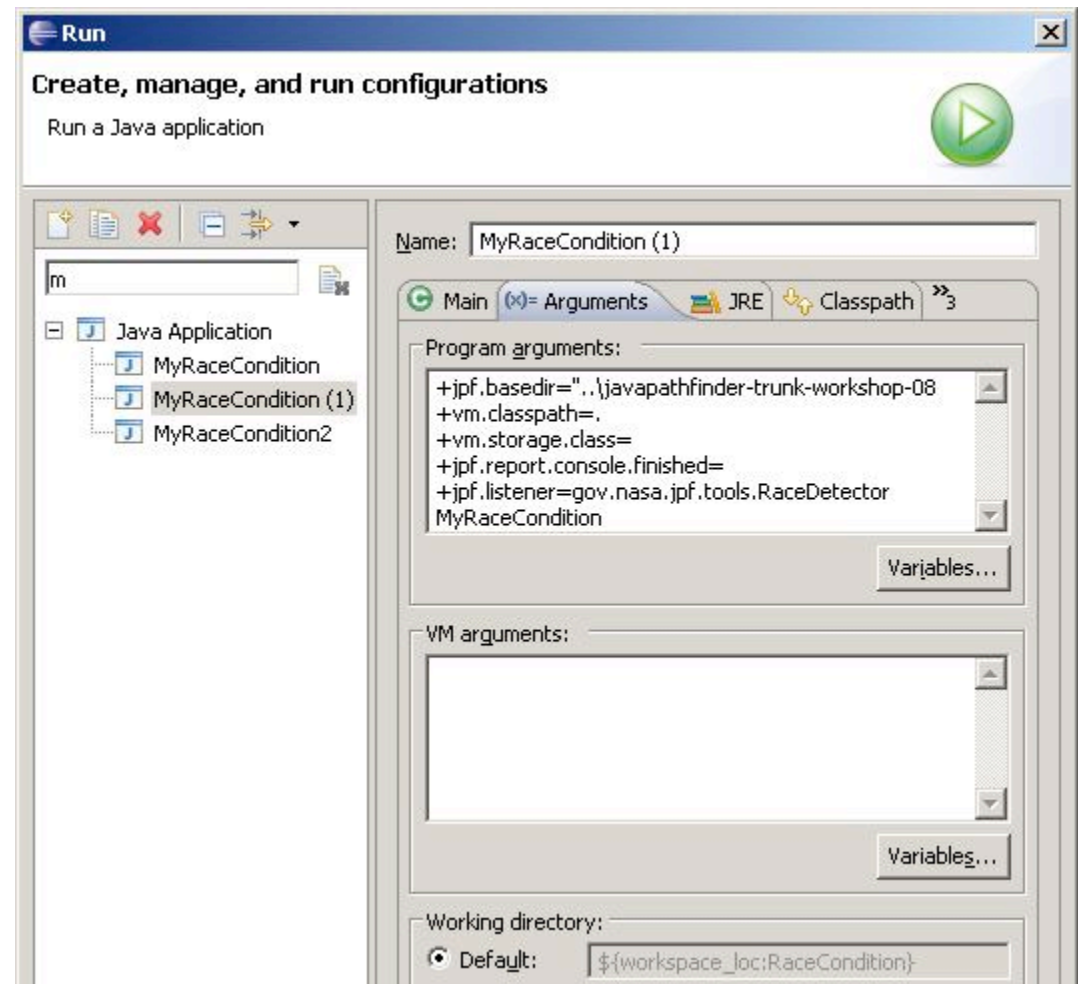
- `gov.nasa.jpj.JPF`
is the Main class



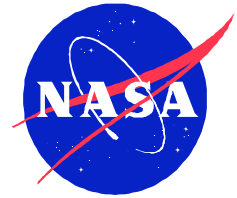
Create Eclipse Run Config (2)



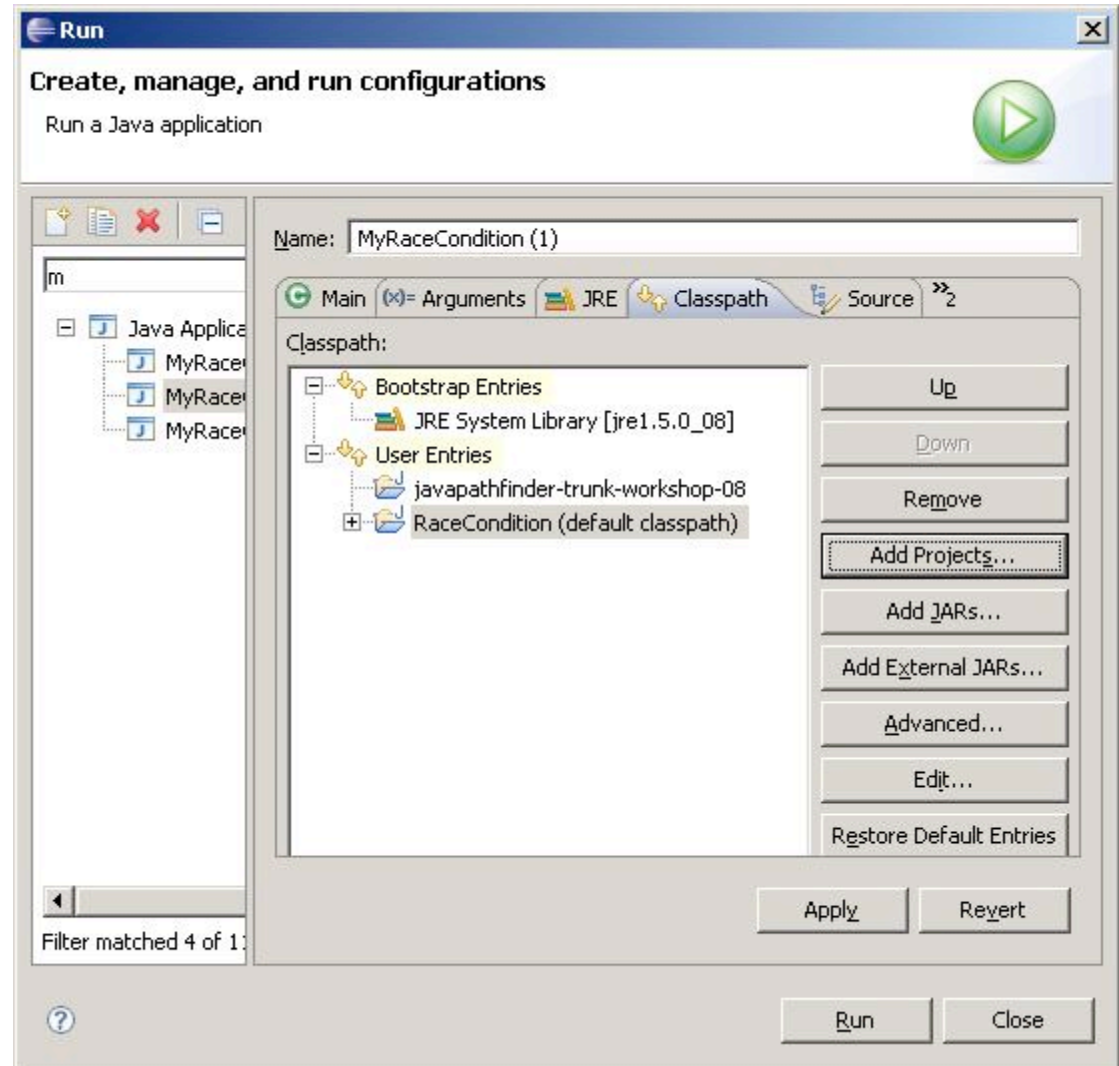
- Set your arguments
- For race conditions:
`jpfd.listener=...RaceDetector`
- Last arg should be the class you are checking



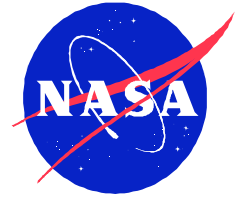
Create Eclipse Run Config (3)



- Add your JPF project in the Classpath tab
- Run



Race Detection Results

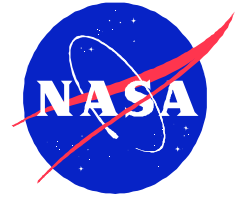


Results:

```
potential race detected: MyRaceCondition$Pair@216.x
  read from thread: "Thread-1", holding locks {} in
  MyRaceCondition$Pair.update(MyRaceCondition.java:7)
  write from thread: "Thread-0", holding locks {} in
  MyRaceCondition$Pair.update(MyRaceCondition.java:7)
```

```
===== error #1
gov.nasa.jpff.tools.RaceDetector
potential field race: MyRaceCondition$Pair@216.x
... etc ...
----- transition #9 thread: 1
gov.nasa.jpff.jvm.choice.ThreadChoiceFromSet {>Thread-0,Thread-1}
  MyRaceCondition.java:7      : x = x + y + x;
  MyRaceCondition.java:8      : }
  MyRaceCondition.java:16     : }
... etc ...
----- transition #11 thread: 2
gov.nasa.jpff.jvm.choice.ThreadChoiceFromSet {main,>Thread-1}
  MyRaceCondition.java:15     : p.update();
----- transition #12 thread: 2
gov.nasa.jpff.jvm.choice.ThreadChoiceFromSet {main,>Thread-1}
  MyRaceCondition.java:15     : p.update();
  MyRaceCondition.java:7      : x = x + y + x;
```

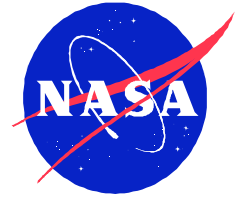

Detecting Deadlock



DEMO

MyRaceCondition2

Detecting Deadlock (1)



```
public class MyRaceCondition2 {
    private static class Pair {
        String x = "x";

        String y = "y";
        String z = "";

        public void update() {
            x = x + y + x;
        }
    }

    private static class RC1 extends Thread {
        Pair p;

        public void run() {
            synchronized (p.x) {
                synchronized (p.y) {
                    p.update();
                }
            }
        }
    }

    private static class RC2 extends Thread {
        Pair p;

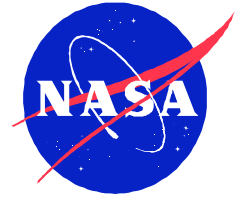
        public void run() {
            synchronized (p.y) {
                synchronized (p.x) {
                    p.update();
                }
            }
        }
    }
}
```

```
public static void main(String[] args) throws
Exception
{
    Pair p = new Pair();
    RC1 rc1 = new RC1();
    RC2 rc2 = new RC2();

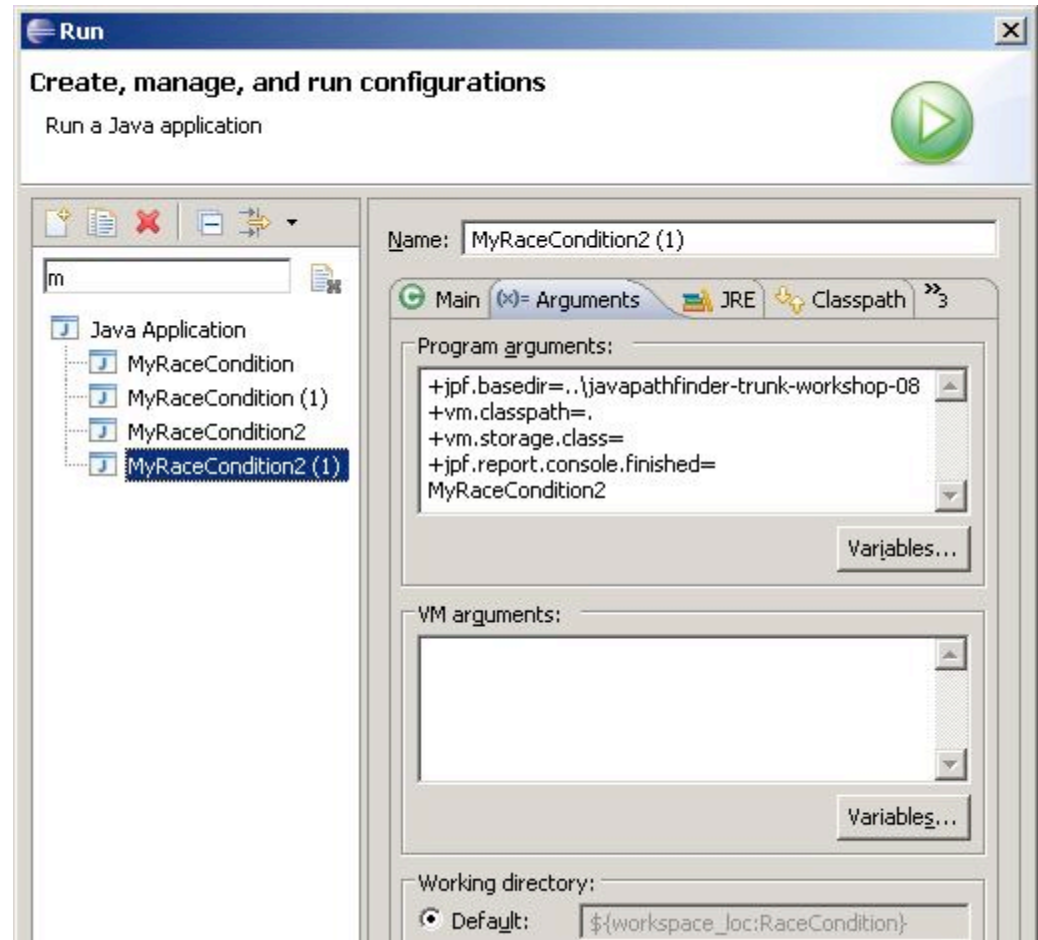
    rc1.p = p;
    rc2.p = p;

    rc1.start();
    rc2.start();
    rc1.join();
    rc2.join();
    System.out.println("x: " + p.x);
}
```

Detecting Deadlock (2)

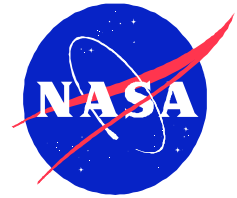


- Run changes:
 - no jpf.listener needed
 - test class: MyRaceCondition2



Detecting Deadlock

(3)



Results:

```
===== error #1
gov.nasa.jpfd.jvm.NotDeadlockedProperty
deadlock encountered:
  thread index=0,name=main,status=WAITING,this=java.lang.Thread@0,target=null,priority=5,lockCount=1
  thread index=1,name=Thread-0,status=BLOCKED,this=MyRaceCondition2$RC1@226,priority=5,lockCount=0
  thread index=2,name=Thread-1,status=BLOCKED,this=MyRaceCondition2$RC2@247,priority=5,lockCount=0
... etc ...

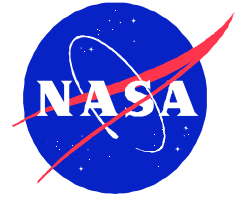
===== snapshot #1
thread index=0,name=main,status=WAITING,this=java.lang.Thread@0,target=null,priority=5,lockCount=1
waiting on: MyRaceCondition2$RC1@226
call stack:
  at java.lang.Thread.join(Thread.java:197)
  at MyRaceCondition2.main(MyRaceCondition2.java:47)

thread index=1,name=Thread-0,status=BLOCKED,this=MyRaceCondition2$RC1@226,priority=5,lockCount=0
owned locks:java.lang.String@217
blocked on: java.lang.String@219
call stack:
  at MyRaceCondition2$RC1.run(MyRaceCondition2.java:18)

thread index=2,name=Thread-1,status=BLOCKED,this=MyRaceCondition2$RC2@247,priority=5,lockCount=0
owned locks:java.lang.String@219
blocked on: java.lang.String@217
call stack:
  at MyRaceCondition2$RC2.run(MyRaceCondition2.java:30)

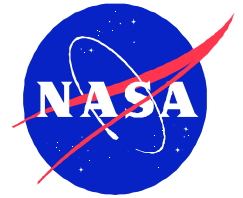
===== search finished: 4/29/08 10:38 AM
```

Verify: Controlling JPF



- The class `gov.nasa.jpjf.jvm.Verify` lets you control simple aspects of JPF
 - Calls to `Verify` methods are specially recognized and handled by JPF
 - Search and backtracking
 - Counters
 - Logging
 - Attributes

Verify: Search (1)



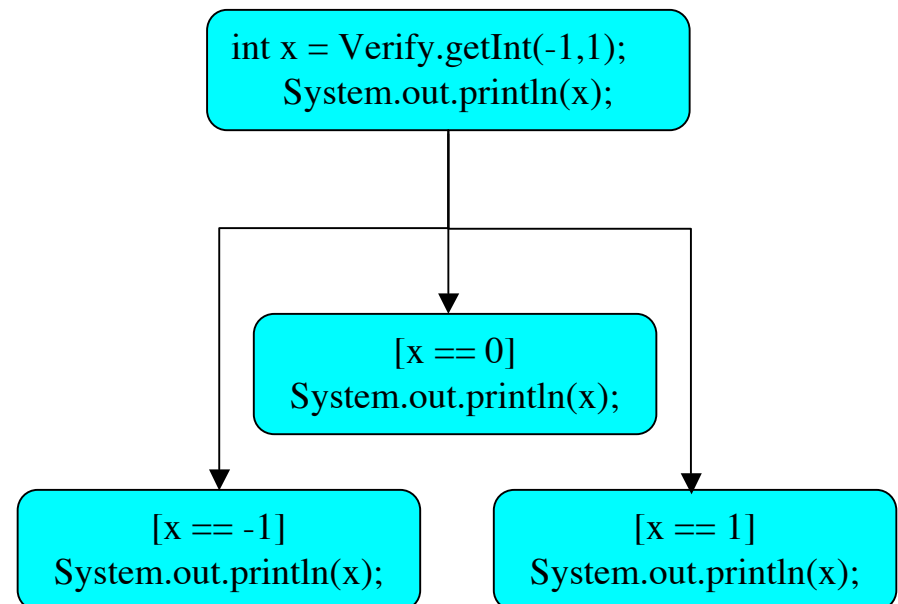
- Choice Generators

- When you need JPF to try alternatives:

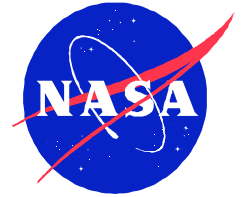
`Verify.getInt,`
`getDouble,`
`getBoolean, ...`

- When JPF hits

`Verify.getXxx()` it branches the execution tree and executes one branch for each value



Verify: Search (2)



- Choice Generator Variations

- Your code:

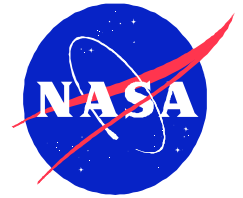
```
double y = Verify.getDouble("Tag");
```

- Your run config:

```
+Tag.class=gov.nasa.jpjf.jvm.choice.DoubleChoiceFromSet  
+Tag.values=-1.0:0.0:123.0
```

- Result: your code runs with $y == -1.0$, $y == 0.0$, and $y == 123.0$
- Other choice generators: see `gov.nasa.jpjf.jvm.choice`

Verify: Search (3)



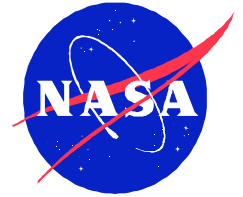
- **Verify.ignoreIf():**

Search Pruning

- Forces JPF to abandon the current execution path and backtrack to the previous choice point
- Useful when you know which parts of the execution search tree are irrelevant to you
- Can speed up search dramatically (by ignoring parts of the search tree)



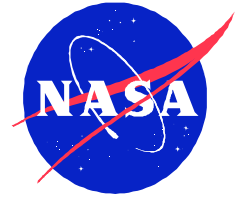
Verify: Search (4)



- Example: your method is not designed to handle cyclic graphs, but your test driver produces them

```
public void print(Graph g) {  
    Verify.ignoreIf(g.isCyclic());  
    ...  
}
```

Verify: Search

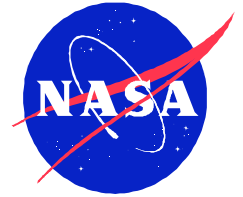


DEMO

Show Verify

Show Verify2

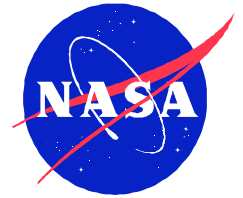
Advanced Topic: Extending JPF



- JPF is extremely flexible: many ways to extend
 - Listeners and properties (example follows)
 - Model Java Interface (MJI): Library abstractions & adding code to the core
 - Redefining bytecodes (see Symbolic Execution and Numerics extensions on SourceForge)
 - Serializer/restorer (Saving and restoring states)
 - Publisher (Collecting and printing statistics and results in different formats)

Advanced Topic: Listeners

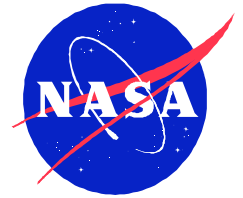
(1)



- Listeners are the preferred way of extending JPF
- You must know a bit about JPF internals to use them
- They give you access to JPF's internal execution

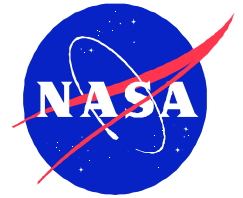


Advanced Topic: Listeners (2)



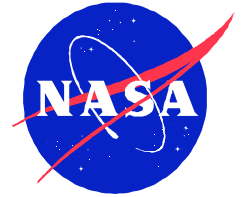
- Two flavors:
`gov.nasa.jpfc.search.SearchListener`
`gov.nasa.jpfc.jvm.VMLListener`
- Interface `SearchListener`: observe search (backtracking, states' processing, property violation, ...)
- Interface `VMLListener`: observe the VM's execution (bytecode execution, exceptions, thread starts, ...)

Advanced Topic: Listeners (3)



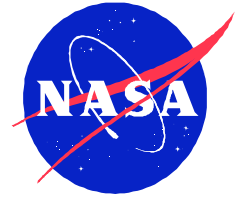
- Useful adapter class:
`gov.nasa.jpf.ListenerAdapter` implements all the methods in `SearchListener` and `VMLListener`.
- See JPF documentation (javapathfinder.sourceforge.net) and code for more details

VMListener Example



DEMO
OpCodePrinter

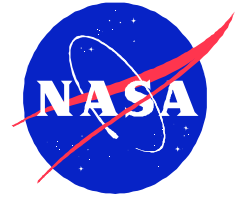
VMListener Example



```
public class OpCodePrinter extends ListenerAdapter {
    String lastLoc = "";

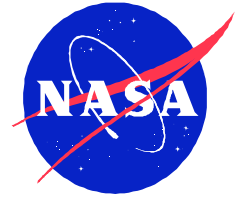
    public void executeInstruction(JVM vm) {
        Instruction instr = vm.getNextInstruction();
        if (instr != null) {
            String loc = instr.getFileLocation();
            if (loc != null && ! loc.startsWith("java")) {
                if (! lastLoc.equals(loc)) {
                    System.out.println(loc);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


JPF Extensions



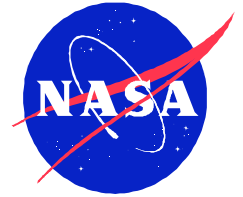
- Extensions found under `extensions/` in the JPF distribution
- Developed independently of JPF core
 - JPF core code should never refer to extension code
 - vice versa ok (of course!)
- Symbolic Execution (`extensions/symbc`)
- UML StateCharts (`extensions/statechart`)
- UI Model Checking (`extensions/ui`)
- Compositional Verification (`extensions/cv`)
- Numeric Properties (`extensions/numeric`)

Extension Example

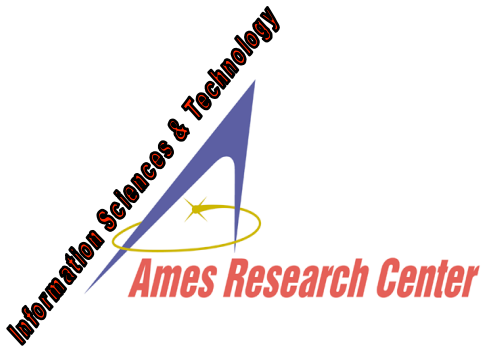


- Numerics Extension
 - Finds “bad” numerical behavior
 - integer and long overflow
 - floating point compares (NaN, infinity)
 - floating point inexact propagation (NaN, infinity)
 - floating point cancellation (lost precision)
 - How?
 - Write new bytecode implementations
 - Write bytecode factory

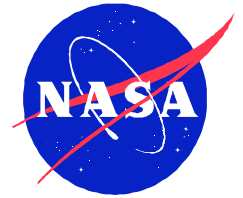
Numerics Extension



- Write InstructionFactory:
 - Tells JPF which bytecodes are being overridden and which classes to use for them
- Override numeric bytecodes:
 - Floating point comparisons: DCMPG, DCMPL, FCMPG, FCMPL
 - Floating point arithmetic: DADD, DSUB, DMUL, DDIV, FADD, FSUB, FMUL, FDIV
 - Int/long arithmetic: IADD, ISUB, IMUL, IDIV, IINC, LADD, LSUB, LMUL, LDIV
- Set config options



Numerics Extension (InstructionFactory)



```
public class NumericInstructionFactory extends GenericInstructionFactory {

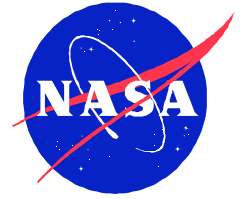
    // which bytecodes do we replace
    static final String[] BC_NAMES = {
        "DCMPG", "DCMPL", "DADD", "DSUB", "DMUL", "DDIV",
        "FCMPG", "FCMPL", "FADD", "FSUB", "FMUL", "FDIV",
        "IADD", "ISUB", "IMUL", "IDIV", "IINC",
        "LADD", "LSUB", "LMUL", "LDIV"};

    // where do they reside
    protected static final String BC_PREFIX = "gov.nasa.jpfn.numeric.bytecode.";

    // what classes should use them
    protected static final String[] DEFAULT_EXCLUDES = { "java.*", "javax.*" };

    public NumericInstructionFactory (Config conf){
        super(BC_PREFIX, BC_NAMES, null, DEFAULT_EXCLUDES);
        NumericUtils.init(conf);
    }
}
```

Numerics Extension (Original IMUL)



```
public class IMUL extends Instruction {
    public void setPeer (org.apache.bcel.generic.Instruction i,
                        ConstantPool cp) {
    }

    public Instruction execute (SystemState ss, KernelState ks,
                               ThreadInfo th) {

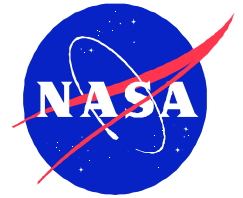
        int v1 = th.pop();
        int v2 = th.pop();

        th.push(v1 * v2, false);

        return getNext(th);
    }

    public int getByteCode () {
        return 0x68;
    }
}
```

Numerics Extension (Overridden IMUL)



```
public class IMUL extends gov.nasa.jpf.jvm.bytecode.IMUL {

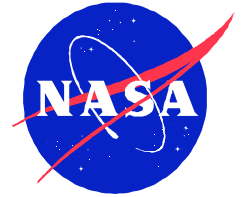
    @Override
    public Instruction execute (SystemState ss, KernelState ks, ThreadInfo th) {
        int v1 = th.pop();
        int v2 = th.pop();

        // check for overflow
        if ((long)v1 * (long)v2 != v1 * v2){
            return th.createAndThrowException("java.lang.ArithmeticException",
                "integer overflow: " + v2 + "*" +
                v1 + "=" + v1*v2);
        }

        th.push(v1 * v2, false);

        return getNext(th);
    }
}
```

Numerics Extension

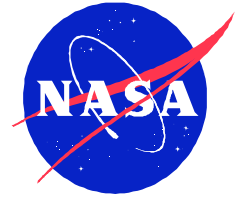


DEMO

NumericsExample



Numerics Extension (Config Options/Running)



Recall:

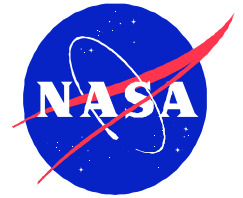
- Main class: `gov.nasa.jpf.JPF`
- Classpath: include your JPF project

Config Program Arguments:

```
+jpf.basedir=..\javapathfinder-trunk-workshop-08  
+vm.classpath=.  
+vm.storage.class=  
+jpf.report.console.finished=  
+vm.insn_factory.class=gov.nasa.jpf.numeric.NumericInstructionFactory  
NumericsExample
```




Numerics Extension (Output)



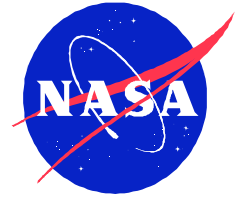
```
===== error #1
gov.nasa.jpjf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: integer overflow: 43046721*43046721=-501334399
    at NumericsExample.main(NumericsExample.java:6)

===== trace #1

----- transition #0 thread: 0
gov.nasa.jpjf.jvm.choice.ThreadChoiceFromSet {>main}
    [1864 insn w/o sources]
    NumericsExample.java:4      : int i = 3;
    NumericsExample.java:5      : for (int j=0; j<10; j++) {
    NumericsExample.java:6      : i = i * i;
    ... etc...
    NumericsExample.java:6      : i = i * i;
    NumericsExample.java:5      : for (int j=0; j<10; j++) {
    NumericsExample.java:6      : i = i * i;

===== snapshot #1
```

References



- JPF: <http://javapathfinder.sourceforge.net>
- Eclipse: <http://www.eclipse.org>
- Subversion Plugin: <http://subclipse.tigris.org>
- Older but more advanced tutorial:
<http://www.visserhome.com/willem/presentations/ase06jpftut.ppt>
- Survey of recent additions: <http://dsrg.mff.cuni.cz/teaching/seminars/2007-06-26-Parizek-JPF4.pdf>
- NASA Ames RSE group publications (including JPF):
<http://ti.arc.nasa.gov/tech/rse/publications/vandvpub.php#model>
- Book on Model Checking theory (not JPF):
Systems and Software Verification, by B. Berard, M. Bidoit, *et. al.*
- Model checking tutorials:
Google for “model checking tutorial”