JAVA PATHFINDER (JPF)

TEAM

- Anusha Mannem
- Derek White
- Muhammad Yousaf

OBJECTIVE

- Basic Understanding
- Architecture
- Examples/Demos

OUTLINE

- Introduction
- Architecture
- Application Types
- Examples
- Extensions
- Installation and Configuration
- Opinion
- References

INTRODUCTION

- What is JPF
- History
- Who is using JPF
- How it works
- JPF Features
- Awards

INTRODUCTION
WHAT IS JPF

In its basic form, JPF is a Software Model Checker

•Find and explain defects

•Coverage Matrices

•Deadlocks

•Unhandled exceptions

•and many more

Depends on how you configure it

INTRODUCTION
HISTORY

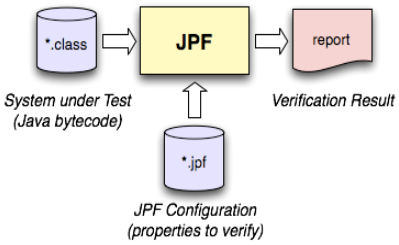
- 1999 - Project started as front end for Spin model checker
- 2000 - Reimplementation as a concrete virtual machine
- 2003 - Introduction of extension interfaces
- 2005 - Open sourced on Sourceforge
- 2008 - Participation in Google Summer Code
- 2009 - Moved to own server, hosting extension projects and Wiki
- 2010 - Participation in Google Summer Code

INTRODUCTION
WHO IS USING JPF

- NASA
- Fortune 500 companies
  - Fujitsu
- Model verification at Ames Research Center
- Collaboration > 20 universities worldwide
  - Waterloo University
  - University of Nebraska-Lincoln
- Major user group is academic research

INTRODUCTION
HOW IT WORKS

- **Virtual Machine (VM) for Java bytecode**
  - Programs to execute
  - Properties to check for as input
  - Returns a report



INTRODUCTION
JPF FEATURES

- Execution Choices
  - Identify points from where execution could proceed differently → Systematically explore them
- State Matching
  - Checks it has seen a similar program state
- Listeners
  - Lets you closely monitor all actions taken by JPF
- Trace
  - Complete execution history that leads to the bug

INTRODUCTION AWARDS

Widely recognized, awards for JPF in general and for related work, team and individuals

• 2002 - ACM Sigsoft Distinguished Paper award

• 2003 - “Turning Goals into Reality” (TGIR) Engineering Innovation

Award from the Office of AeroSpace Technology

• 2004, 2005 - Ames Contractor Council Awards

• 2007 - IBM's Haifa Verification Conference (HVC) award

• 2009 - “Outstanding Technology Development” award of the Federal Laboratory Consortium for Technology Transfer (FLC)

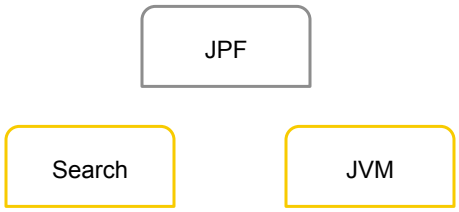
ARCHITECTURE

- **Top Level Design**
- **Choice Generator**
- **Bytecode Factory**
- **Partial Order Reduction**
- **Attributes**
- **Model Java Interface (MJI)**
- **Listener**
- **Report**

ARCHITECTURE
TOP LEVEL DESIGN

Designed around two abstractions

- Virtual Machine (JVM)
- Search



TOP LEVEL DESIGN
VIRTUAL MACHINE

- Java specific state generator
  - Matching (has the state been visited before)
  - Storing
  - Backtracking
- Three major JVM methods in the context of VM – Search Collaboration
  - Forward - generate the next state, report if the generated state has a successor. If yes, store on a backtrack stack for efficient restoration.
  - Backtrack - restore the last state on the backtrack stack
  - RestoreState - restore state

TOP LEVEL DESIGN PACKAGE STRUCTURE

JPF core is divided into the following packages:

- gov.nasa.jpf – configuration and initialization of core JPF objects (Search and JVM)
- gov.nasa.jpf.jvm – main body of the core code lives here including implementation of state generator. Also contains class management, object model, and handles bytecode execution
- gov.nasa.jpf.search – contains Search class, which is abstract base class for search policies. Also holds plain DFSSearch policy. More interesting policies contained in gov.nasa.jpf.search.heuristic

TOP LEVEL DESIGN SEARCH STRATEGIES

- Responsible for selecting the state from which the JVM should proceed
- Direct the JVM to move forward to the next state or to backtrack. Search objects act as “drivers” for the JVM
- Mainly provides a single search method, which includes the main loop that iterates through the relevant state space until it has been completely explored or a property violation has been found
- Search strategy objects evaluate property objects such as NotDeadlockedProperty and NoAssertionsViolatedProperty
- Implementations include simple DFS or priority queue based search, configurable for search types by selecting interesting states from the collection of successors (HeuristicSearch)

TOP LEVEL DESIGN SEARCH STRATEGIES (2)

- DFSHeuristic
  - Heuristic prioritizer for depth for search
- BFSHeuristic
  - Heuristic prioritizer for breadth first search
- BeamSearch
  - State queue is reset for each priority level (i.e. doesn't hop between state levels when fetching the next state from the queue)
- GlobalBranchCoverage
  - Prioritizer to maximize global branch coverage

TOP LEVEL DESIGN SEARCH STRATEGIES (3)

- GlobalInstCoverage - prioritizer that favors global instruction coverage
- GlobalSwitchThread - prioritizer that tries to minimize re-scheduling
- Interleaving - Maximize thread interleavings
- MostBlocked - Tries to maximize the number of blocked states
- PreferThread - Favors certain threads specified by thread names during initialization
- And more...

ARCHITECTURE CHOICE GENERATOR

- When performing software model checking it is important to make the best choices (following the right paths) that will result in reaching system states that are interesting
- It is also necessary to make the best use of resources and work within constraints of the execution environment
- Motivation for ChoiceGenerators come from early support for “random” data acquisition in gov.nasa.jpf.jvm.Verify

ARCHITECTURE CHOICE GENERATOR (2)

- While this works well when small sets of choice values are involved, the mechanism for enumerating all possible results from a type specific interval becomes more questionable for large intervals and fails completely for infinite choice sets (such as floats)
- There must be a departure from the “consider all possible choices” world of model checking that allows for knowledge about the real world to help make choice sets finite and manageable
- But, the heuristics that would allow this are application and domain specific so they shouldn’t be hardcoded into test drivers that JPF analyzes

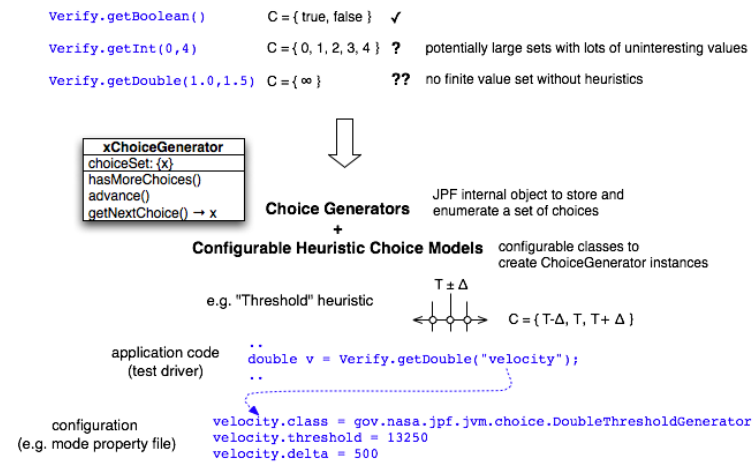
ARCHITECTURE CHOICE GENERATOR (3)

- This has lead to the following requirements of the JPF choice mechanism:
  - choice mechanisms have to be decoupled - thread choices should be independent of data choices, double choices from int choices etc.
  - choice sets and enumeration should be encapsulated in dedicated, type specific objects. The VM should only know about the most basic types, and otherwise use a generic interface to obtain choices
  - selection of classes representing (domain specific) heuristics, and parameterization of ChoiceGenerator instances should be possible at runtime via JPF's configuration mechanism

ARCHITECTURE CHOICE GENERATOR (4)

- The example given (VELOCITY) shows the use of randomly chosen velocity values of double type
- The example of a threshold model where it is of interest how the system reacts below, at, and above specific threshold values
- The infinite set of choices from the range of possible doubles is reduced to only three “interesting” values
- A symbolic name “velocity” is used rather than a reference to a ChoiceGenerator class, which JPF uses to look up the class name from configuration data
- If the heuristic needs parameterization, values can also be passed to the constructor using properties in the configuration file

ARCHITECTURE
CHOICE GENERATOR (5)



Source: JPF Wiki

ARCHITECTURE
CHOICE GENERATOR (6)

- The standard classes and interfaces for the ChoiceGenerator mechanism can be found in package gov.nasa.jpf.jvm, and include:
- ChoiceGenerator, BooleanChoiceGenerator, IntChoiceGenerator, DoubleChoiceGenerator, ThreadChoiceGenerator, SchedulingPoint, SchedulerFactory, DefaultSchedulerFactory
- Concrete implementations can be found in package gov.nasa.jpf.jvm.choice, and include classes like: IntIntervalGenerator, IntChoiceFromSet, DoubleChoiceFromSet, DoubleThresholdGenerator, SchedulingChoiceFromSet

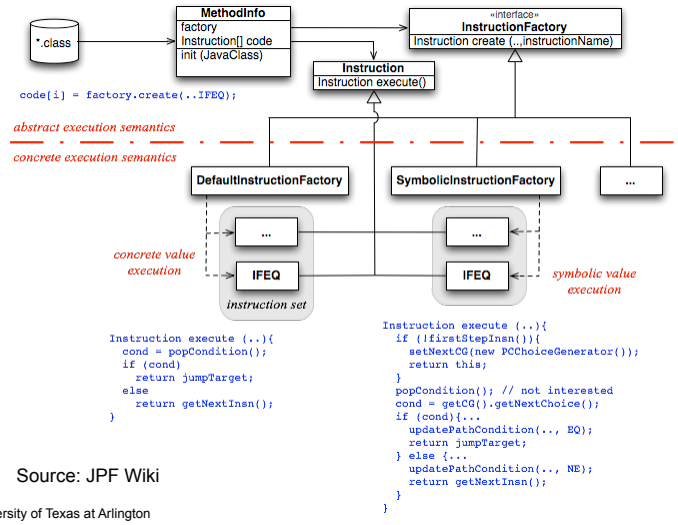
ARCHITECTURE BYTECODE FACTORIES

- JVM and its instruction set represents a multi-threaded stack machine with bytecode instructions specified in the Java Virtual Machine Specification
- JPF VM is different – associated constructs still provide all of the means to implement a Java interpreter, but JPF delegates the use of these means to instructions
- Every bytecode is executed as a representation of a corresponding Instruction object that is instantiated during class load time
- To execute the bytecode, the JPF JVM calls the execute() method of the Instruction object and the actions taken in execute() methods define execution semantics for that instruction

ARCHITECTURE BYTECODE FACTORIES (2)

- Since JPF uses a configurable factory to instantiate Instruction classes, if the user wishes to provide their own InstructionFactory and related Instruction classes, it is possible to change the semantics of Java.
- Why is it useful to extend the semantics of Java in this way?
  - To add additional checks to instructions.
  - For example, the numeric bytecode classes in jpf-numeric are extended to check for problems such as overflow and NaN propagation
  - More examples later from the jpf-symbc extension

ARCHITECTURE
BYTECODE FACTORIES (3)



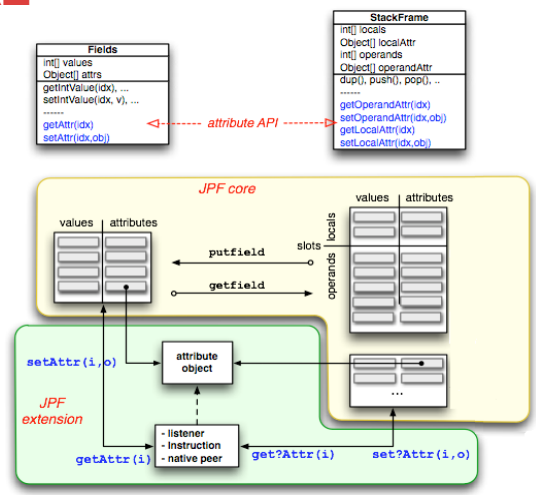
Source: JPF Wiki

ARCHITECTURE
PARTIAL ORDER REDUCTION

- If POR is enabled, VM executes all instructions in the current thread until one of the following conditions is met:
  - Next instruction is scheduling relevant
  - Next instruction yields a "nondeterministic" result
- If both conditions are detected then they are delegated to the instruction object which passes down information about the current VM execution state and threading context
- If the instruction is a transition breaker, then it creates a ChoiceGenerator and schedules itself for re-execution

ARCHITECTURE ATTRIBUTES

- Like normal VM,JPF also stores values for operands, local variables & fields.
- It features a storage extension mechanism that lets you associate arbitrary objects with such values
- These objects can then used in native peers or listeners to add state restored information that automatically follows the data flow

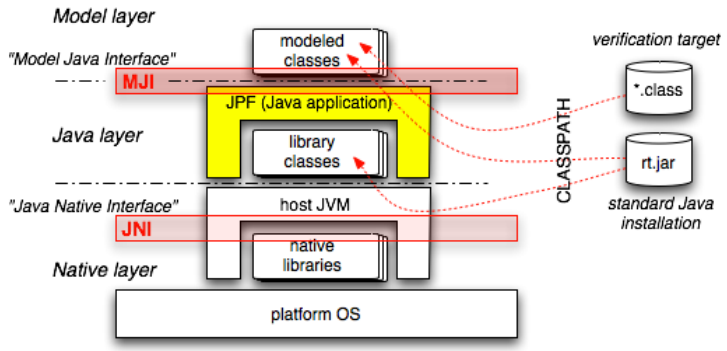


ARCHITECTURE MODEL JAVA INTERFACE

- For a normal java application, JPF can be viewed as a normal JVM.
- At different times *.class can be processed in two different ways in a JVM running JPF
  - As ordinary Java classes
  - As "modeled" classes
- JPF has its own class and object model, which is completely different and incompatible to the class and object models of the underlying host JVM executing JPF

ARCHITECTURE
MODEL JAVA INTERFACE (2)

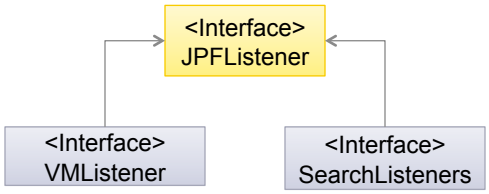
- MJI is used for lowering the execution from JPF executed code into JVM executable code



ARCHITECTURE
LISTENER

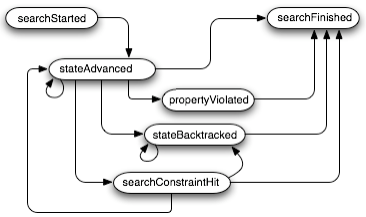
- Provides a way to observe, interact with and extend JPF execution with your own classes
- Dynamically configured at runtime
- Listeners are executed at the same level like JPF, so there is hardly any limit of what you can do with them
- Provides an Observer pattern implementation that notifies registered observer instances about certain events
  - Observer Pattern - Called the subject, maintains a list of its dependents, called observers, and observes any state changes, usually by calling one of their methods
- These notifications cover a broad spectrum of JPF operations, from low level events like *instructionExecuted* to high level events like *searchFinished*

ARCHITECTURE LISTENER (2)

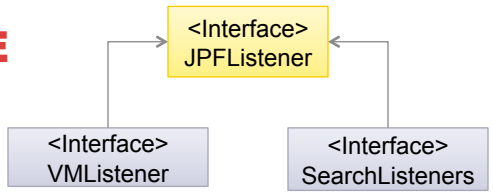


ARCHITECTURE LISTENER (2)

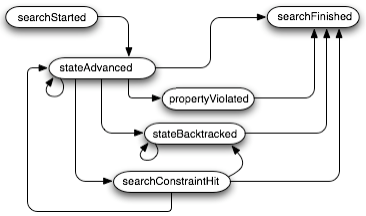
- VMListener
  - This is a fat interface, reflecting various VM operations instructionExecuted , objectCreated etc
- SearchListener
  - *SearchListener* instances are used to monitor the state space search process
    - stateAdvanced: goto next state
    - propertyViolated: encountered a property violation
    - stateBacktracked: state was backtracked one step (same path)



ARCHITECTURE LISTENER (2)



- VMListener
  - This is a fat interface, reflecting various VM operations instructionExecuted , objectCreated etc
- SearchListener
  - *SearchListener* instances are used to monitor the state space search process
  - stateAdvanced: goto next state
  - propertyViolated: encountered a property violation
  - stateBacktracked: state was backtracked one step (same path)

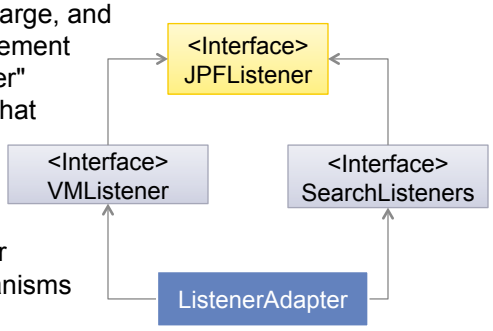


University of Texas at Arlington

35

ARCHITECTURE LISTENER (2)

- These interfaces are quite large, and listeners often need to implement both, it also provide "adapter" classes, i.e. implementors that contain all required method definitions with empty method bodies
- They also support two other interfaces/extension mechanisms

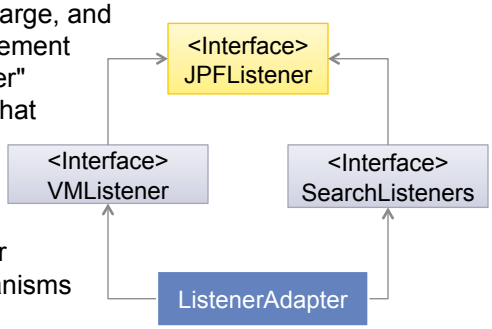


University of Texas at Arlington

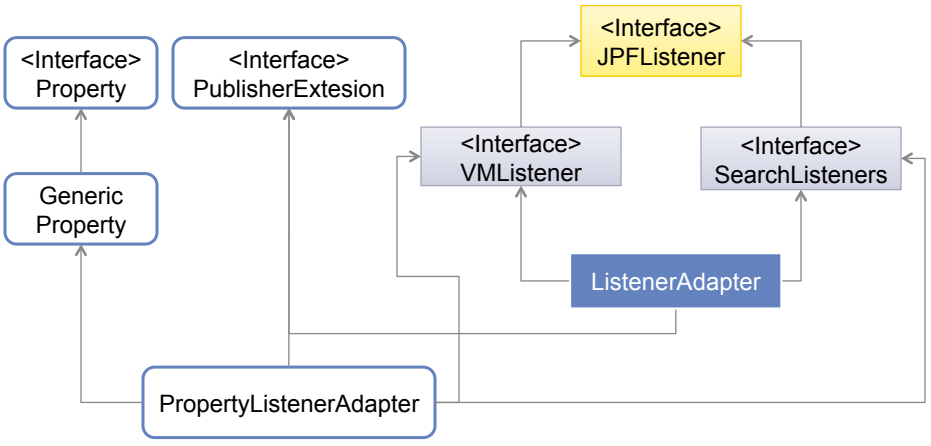
36

ARCHITECTURE LISTENER (2)

- These interfaces are quite large, and listeners often need to implement both, it also provide "adapter" classes, i.e. implementors that contain all required method definitions with empty method bodies
- They also support two other interfaces/extension mechanisms

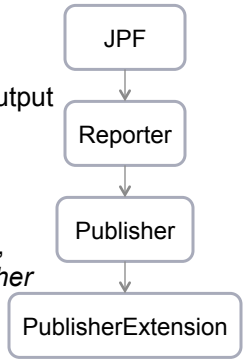


ARCHITECTURE LISTENER (2)



ARCHITECTURE REPORT

- Reporter
  - Data collector
  - Manages and notifies *Publisher* when a certain output phase is reached
- Publisher
  - Format (e.g. text, XML) specific output producers, the most prominent one being the *ConsolePublisher*
- *PublisherExtensions*
  - Registered for specific *Publishers*



APPLICATION TYPE

- **JPF Unaware**
- **JPF Dependent**
- **JPF Enabled**

APPLICATION TYPE
JPF UNAWARE PROGRAM

- Typical reason to check such an application with JPF is to look for violations of so called non-functional properties that are hard to test for, like deadlocks
- JPF is especially good at finding and explaining concurrency related defects
- Benefit: JPF can check Java program against certain properties
- Costs: JPF is much slower than a production VM, and it might not support all the Java libraries that are used in the application

APPLICATION TYPE
JPF DEPENDENT PROGRAM

- Their only purpose in life is to be verified by JPF (e.g. to check a certain algorithm)
- Typically, these applications are based on a domain specific framework that has been written so that JPF can handle the model, and knows what to look for
- Benefit: Model applications themselves are usually small, scale well, and do not require additional property specification
- Cost: Quite expensive to develop the underlying domain frameworks

APPLICATION TYPE
JPF ENABLED PROGRAM

- Programs that can run on any VM, but contain Java annotations that represent properties which cannot easily be expressed with standard Java language.
- Annotations
  - Ensure
  - Require
  - etc

EXAMPLES

- Non-Deterministic Data
- Data Race
- Deep Inspection
  - Const Annotation
  - Sandbox Annotation
- Overflow
- Test Case Generation – Symbolic Execution

EXAMPLE
NON-DETERMINISTIC DATA

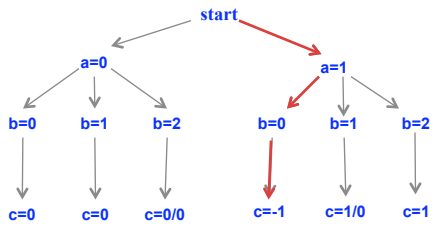
Source :

```
import java.util.Random;

public class Rand {
  public static void main (String[] args) {
    System.out.println("computing c = a/(b+a - 2)..");
    Random random = new Random(42);       // (1)

    int a = random.nextInt(2);            // (2)
    System.out.println("a=" + a);

    //... lots of code here

    int b = random.nextInt(3);            // (3)
    System.out.println("  b=" + b + "        ,a=" + a);

    int c = a/(b+a -2);                   // (4)
    System.out.println("=>  c=" + c + "      ,a=" + a + ",b=" +b);

  }
}
```


EXAMPLE
NON-DETERMINISTIC DATA (2)

Normal VM Output :

```
computing c = a/(b+a - 2)..
a=1
  b=0        ,a=1
=>  c=-1      ,a=1,b=0
```



(1) **Random random = new Random(42)**

(2) **int a = random.nextInt(2)**

(3) **int b = random.nextInt(3)**

(4) **int c = a / (b + a - 2)**

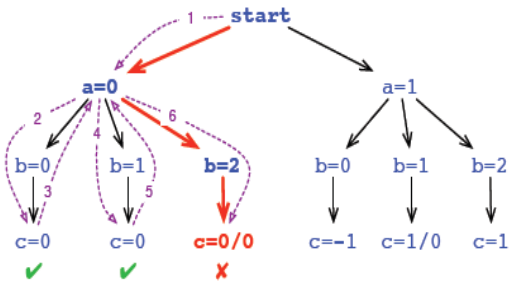
JPF EXECUTION (3)

If we start JPF as a plain 'java' replacement,the only difference is that it
(a) takes longer to complete, and
(b) tells us something about a "search"

```
Problems | Javadoc | Declaration | Console
Rand.jpf(run)
Executing command: java -jar C:\users\amul\workspace\jpf-core\build\RunJPF.jar +shell.port=4242 C:\Users\amul\workspace\jpf-core\src\examples\Rand.jp
JavaPathfinder v5.x - (C) RIACS/NASA Ames Research Center


================================================== system under test
application: Rand.java

================================================== search started: 5/3/10 11:14 AM
computing c = a/(b+a - 2)..
a=1
  b=0       ,a=1
=>  c=-1     ,a=1,b=0

================================================== results
no errors detected

================================================== statistics
elapsed time:       0:00:00
states:             new=1, visited=0, backtracked=0, end=1
search:             maxDepth=0, constraints=0
choice generators:  thread=1, data=0
heap:               gc=1, new=315, free=27
instructions:       3043
max memory:         15MB
loaded code:        classes=75, methods=1047

================================================== search finished: 5/3/10 11:14 AM
```


NON DETERMINISTIC DATA (4) JPF EXECUTION

Model checking (theoretically) covers all executions




```
>jpf +cg.enumerate_random=true
        Rand
  a=0
    b=0
      c=0
    b=1
      c=0
    b=2
ERROR: ArithmeticException
```

EXAMPLE DATA RACE

Source :

```
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) {}
    }
}
```


DATA RACE (2)

Configuration : listener=gov.nasa.jpf.listener.PreciseRaceDetector
report.console.property_violation=error,trace

Output :

```
...
================================ search started: 2/10/10 6:32 PM
10
10
================================ error #1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Racer@287.d
  main at Racer.main(Racer.java:16)
                "int c = 420 / racer.d;              "   : getfield
  Thread-0 at Racer.run(Racer.java:7)
                "d = 0;                              "   : putfield
================================ trace #1
------------------------------------------ transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
      [2843 insn w/o sources]
  ...
  Racer.java:16                   : int c = 420 / racer.d;
------------------------------------------ transition #4 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
  ...
  Racer.java:7                    : d = 0;
------------------------------------------ transition #5 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main,Thread-0}
  Racer.java:16                   : int c = 420 / racer.d;
```

EXAMPLE CONST ANNOTATION (1)

- Deep inspection from jpf-aprop
- Works primarily on properties that benefit or require VM level evaluation which provides execution history and details
- These checks do not modify the system under test because they are performed at the VM level
- Can be based on hints (or annotations) in the system under test
- A const annotation is a transitive property that holds for a certain dynamic scope
- These are checked by dedicated listeners under runtime monitoring
- This const annotation can check multiple levels of the stack for any changes to variables that are caused by a call in the annotated code block

EXAMPLE CONST ANNOTATION (2)

- The annotation can be used for entire classes, fields, or methods. In class scope, none of the fields are allowed to change outside static initialization or constructor invocations

```
import gov.nasa.jpf.annotation.Const;

@Const static class ConstObj {
    int d;

    ConstObj (){
      initialize();
    }

    void initialize() { // Ok to call from within ctor, error if called
  outside
      d = 42;
    }
    ...
  }
}
```

EXAMPLE
CONST ANNOTATION (4)

- If used with method scope the fields of the defining class cannot be assigned from within this method (including nested method invocations):

```
class SomeClass {
  int d;
  void set() {
    d = 42;
  }
  @Const void dontDoThis() { // 'd' not allowed to be changed
  directly or indirectly within this method
    set();
  }
}
```


EXAMPLE
SANDBOX

- Another one of the deep inspection examples
- Counterpart of @Const, @SandBox says that a class is only allowed to modify its own fields and nothing outside
- Sandbox annotation provides a *"more sophisticated, instance aware scope property"* where instances are only allowed to write to "owned" fields (which include fields of objects stored in reference fields)
- This mechanism is suitable for security related properties

EXAMPLE OVERFLOW

- Part of jpf-numeric
- Java doesn't thrown exceptions on overflows but this would be useful to find out about in a system under test

EXAMPLE TEST CASE GENERATION

- Under the symbolic execution category of JPF capabilities
- Test creation is expensive
  - Must answer the questions of how many tests are needed, what kind of tests (instruction coverage, branch coverage, path coverage).
- This is where symbolic execution can save us
- Typical verification would use data to find the paths that need to be explored
- On the other hand symbolic execution uses the program structure to deduce data values that will reach the interesting program locations

EXTENSIONS

Java Pathfinder is organized as several runtime modules that are maintained as subprojects known as extensions:

- jpf-aprop - Java annotation based properties and checkers
- jpf-awt - JPF library implementations for java.awt and javax.swing
- jpf-awt-shell - shell for model checking AWT and Swing applications
- jpf-concurrent - optimized java.util.concurrent library implementation for JPF
- jpf-delayed - postpones non-deterministic choice of values until they are used
- jpf-guided-test - Framework for guiding the search using heuristics and static analysis

EXTENSIONS (2)

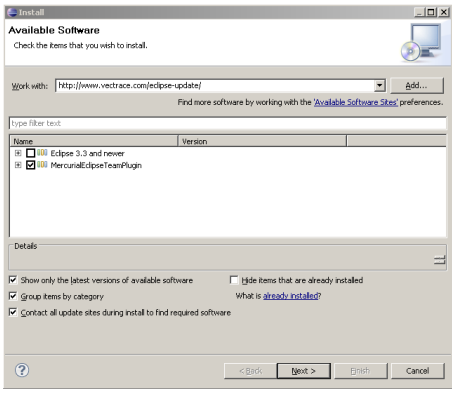
- jpf-mango - specification and proof artifact generation
- jpf-numeric - an alternative bytecode set for inspection of numeric programs
- net-iocache - I/O cache extension to handle network communication
- rtembed - verification of Java programs for real-time and embedded platforms (e.g. RTSJ)
- jpf-symbc - Symbolic PathFinder - symbolic execution for Java bytecode
- eclipse-jpf - JPF launcher plugin for Eclipse IDE
- jpf-shell - the generic user interface support for JPF
- netbeans-jpf - JPF launcher plugin for NetBeans IDE

INSTALLATION AND CONFIGURATION

- Install > JDK 1.6.10
- Install Eclipse Classic 3.52
- Install Mercurial 3.5.0

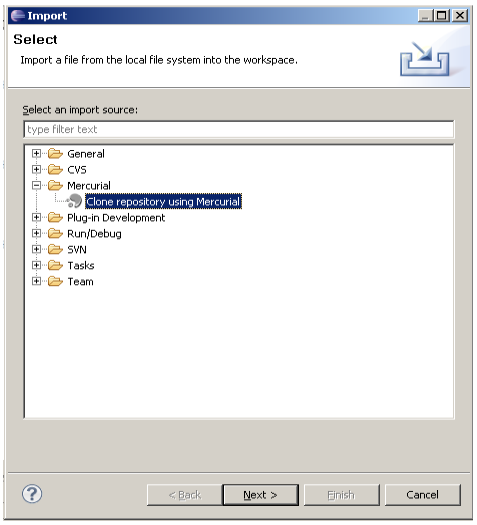
INSTALLATION AND CONFIGURATION

- Install Eclipse Mercurial Plugin
- Click help -> Install new software
- Work with: http://www.vectrace.com/eclipse-update/
- Check mark MercurialEclipse TeamPlugin
- Click next and finish



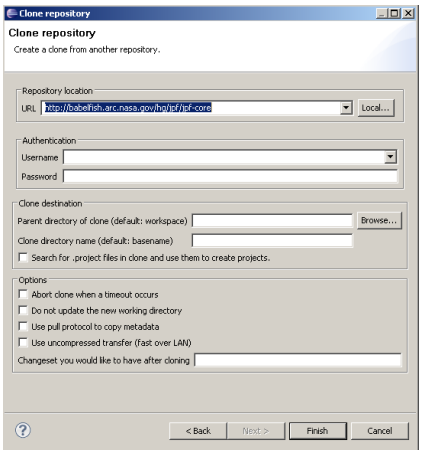
INSTALLATION AND CONFIGURATION

- In the eclipse menu:
  - File -> Import -> Mercurial -> Clone repository using Mercurial -> Next



INSTALLATION AND CONFIGURATION

- In the repository location, URL, specify http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
- Check the box for 'Search for project files in clone and use them to create projects'
- Click finish



INSTALLATION AND CONFIGURATION

- The site.properties file tells JPF at startup time where to look for installed projects, so that it can add classpaths accordingly without you having to type off your fingers
- Default location: **<user.home>/.jpf/site.properties**
- Available at: http://babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/install/site-properties/site.properties

INSTALLATION AND CONFIGURATION

- Ensure that the JAVA_HOME environment variable points to the jdk1.6xxx directory. If it is empty or points to a JRE then errors such as **javac not found** maybe seen. If you do not want the system Java settings to point to jdk1.6xxx, you can also set project specific settings in eclipse
- If you eclipse settings are set to **Build Automatically** then the project after being cloned will be built
- To build a particular project in the Project menu select **Build Project**. All the dependencies for the project will be built automatically

INSTALLATION AND CONFIGURATION

- In Eclipse go to **Project** -> **Properties**
- Select **Builders**
- Pick **Ant_Builder** -> click **Edit**
- Click on the **JRE** tab
- Select **Separate JRE** -> **Installed JREs**
- Pick JDK1.6xxx. If it is not listed under the installed JREs, click on **Add**, browse your file system to where JDK1.6xxx resides and select

OPINION

Pros

•Powerful

•Flexible

•Adaptive

Cons

•Slow

•Not well documented

•Configuration can be intimidating

REFERENCES

- http://babelfish.arc.nasa.gov/trac/jpf
- http://groups.google.com/group/java-pathfinder
- Design Patterns, Elements of Reusable Object Oriented

THANK YOU !!!