JPF Tutorial - Part 1

JPF Core System

Willem Visser
Stellenbosch University

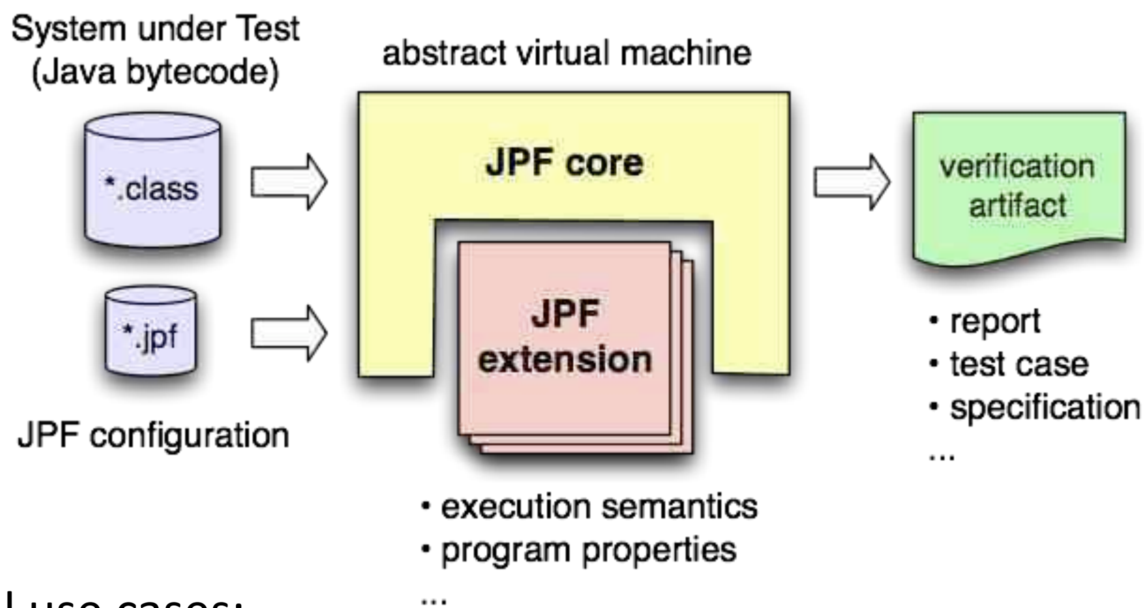
Most of the slides by Peter Mehlitz

Overview

- Examples
- What is JPF?
- Extending JPF
  - Listeners
  - Bytecode Factories
  - Model classes
- Getting started
  - Download, Install and Run (in Eclipse)
- Google Summer of Code

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



- typical use cases:
  – software model checking (deadlock & race detection)
  – deep inspection (numeric analysis, invalid access)
  – test case generation (symbolic execution)
  – ... and many more

History of JPF

- not a new project: around for 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

Users?

- 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

Awards

- widely recognized, awards for JPF in general and for related work, team and individuals
  - 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)

No Free Lunch

- you need to learn
  - JPF is not a lightweight tool
  - flexibility has its price - configuration can be intimidating
  - might require extension for your SUT (properties, libraries)
- you will encounter unimplemented/missing parts (e.g. UnsatisfiedLinkError)
  - usually easy to implement
  - exception: state-relevant native libraries (java.io, java.net)
  - can be either modeled or stubbed
- you need suitable test drivers

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

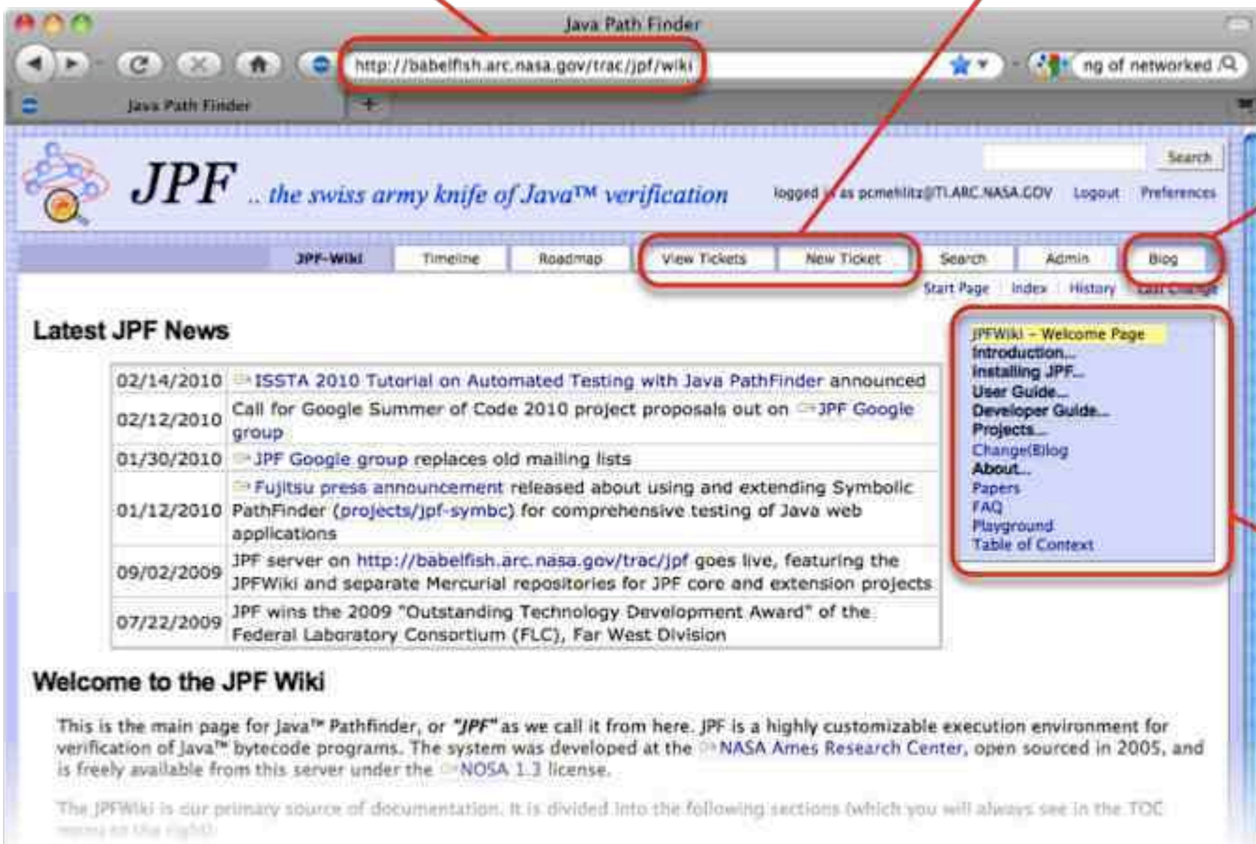
- public read access
- edit for account holders (also non-NASA)

bug tracking

- Trac ticket system

project blog

- announcements
- important changes



hierarchical navigation menu

- intro
- installation
- user docu
- developer docu
- **extension projects**

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)

Application Types

JPF unaware programs	*JPF enabled programs*	*JPF dependent programs*
runs on any JVM	"sweet spot"	runs only under JPF

annotate program
- requirements
- sequences (UML)
- contracts (PbC)
- tests

...

analyze program
- symbolic exec
  → test data
- thread safety / races

constraints

JPF unaware programs:

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

JPF dependent programs:

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

initial domain impl. costs
- domain libs can be tricky

benefits

JPF unaware programs:

non-functional properties
- unhandled exceptions
  (incl. AssertionError)
- deadlocks
- races

improved inspection
- coverage statistics
- exact object counts
- execution costs

JPF dependent programs:

low modeling costs
- statemachine w/o layout hassle,..

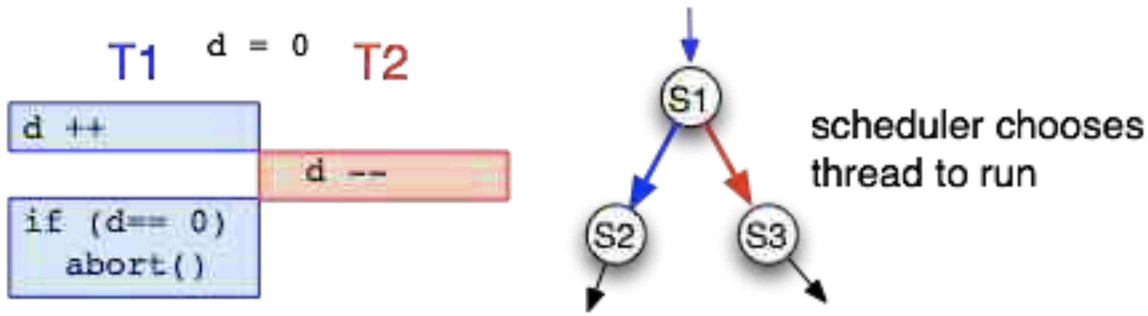
functional (domain) properties
- built-in into JPF libraries

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

Examples

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



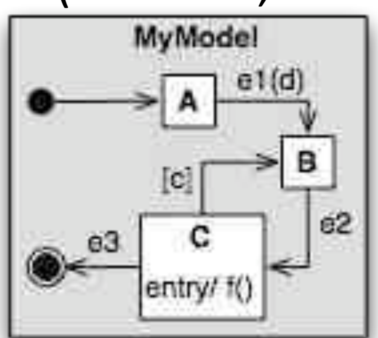
- deep inspection of production code
  - property annotations (Const, PbC,..)
  - numeric verification (overflow, cancellation)

```
@Const
int dontChangeMe() {..}
```

```
double x = (y - z) * c
```

numeric error of x?

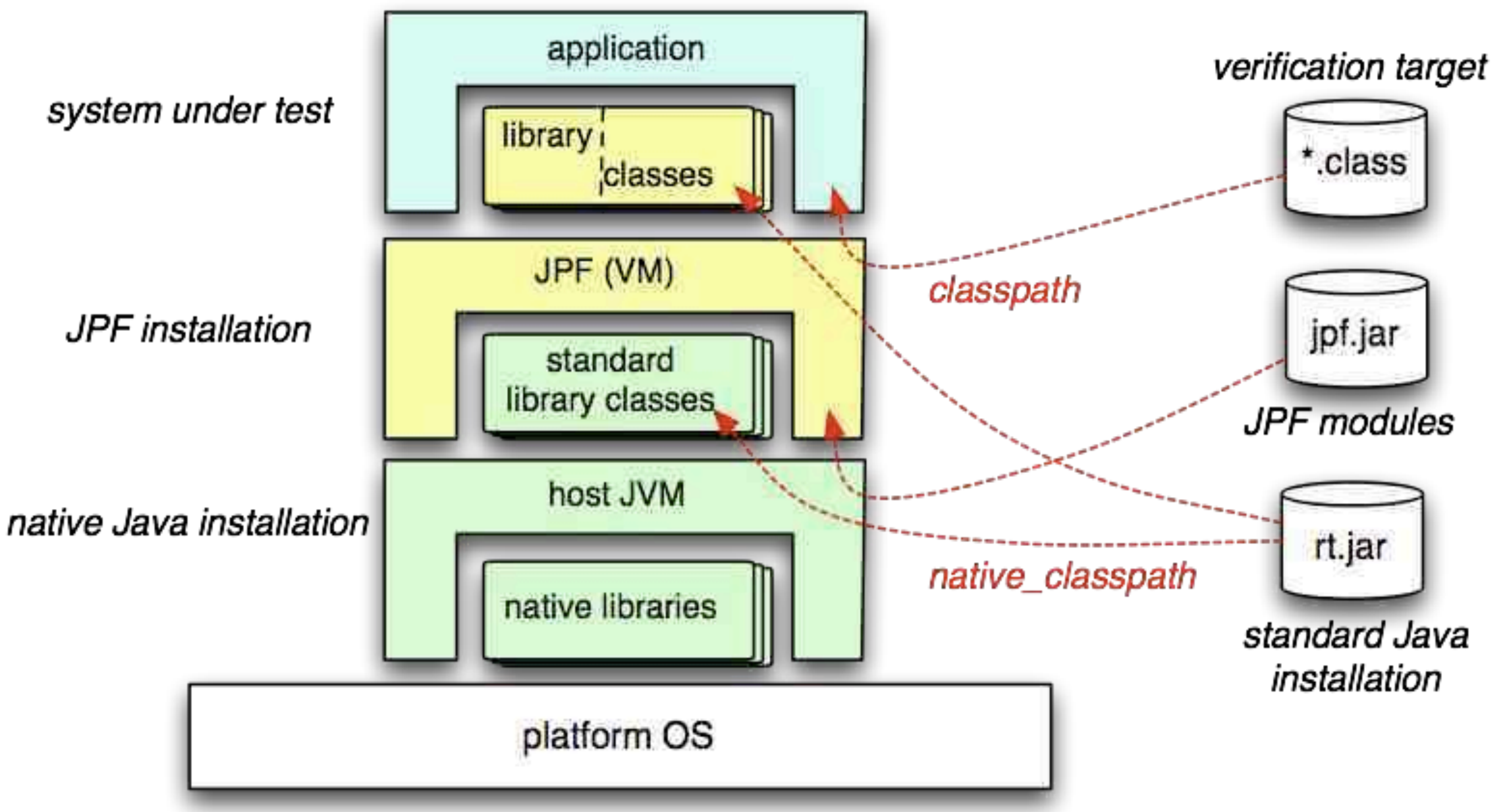
- model verification
  - UML statecharts



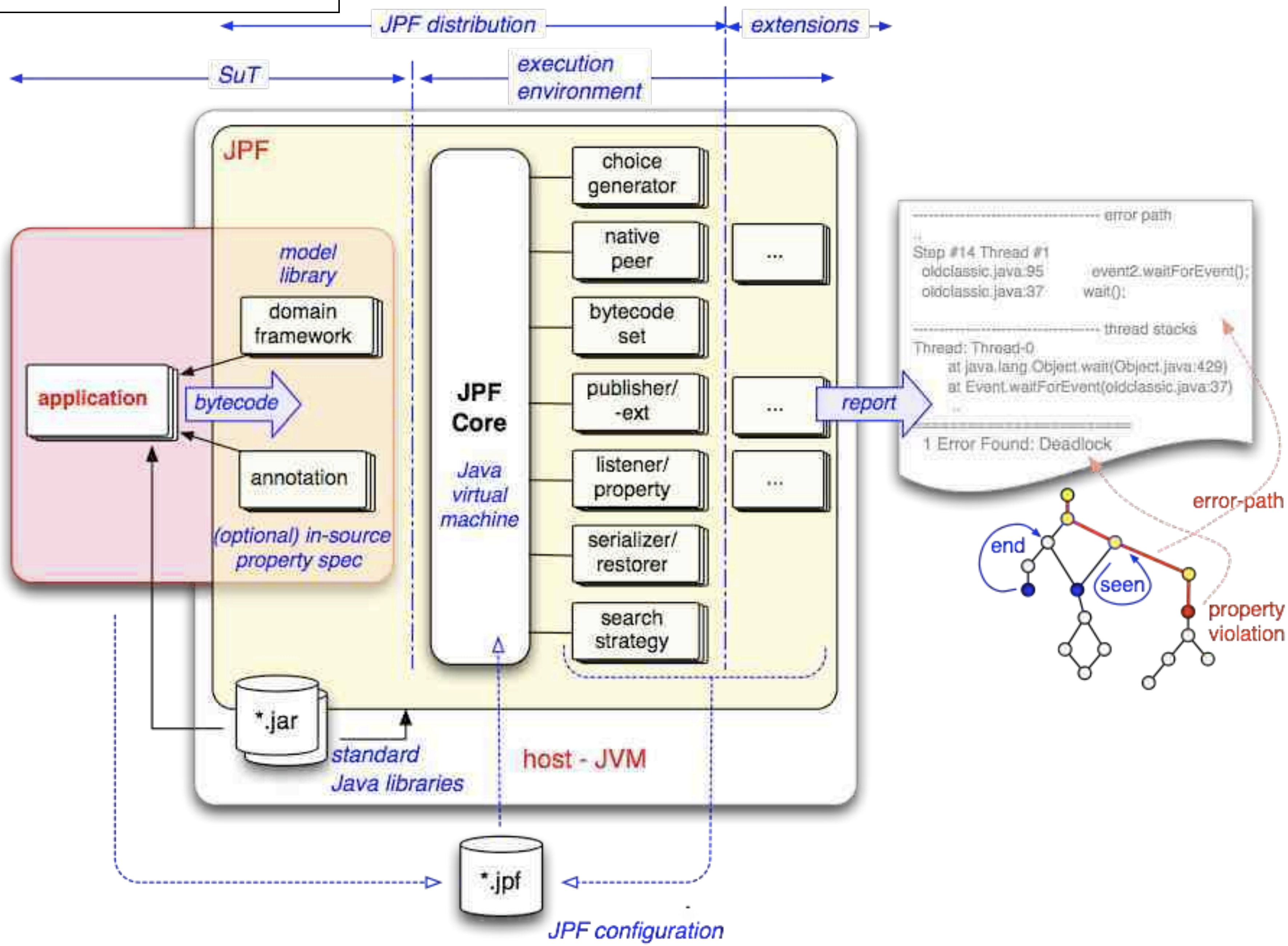
- test case generation

JPF and the Host JVM

- 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

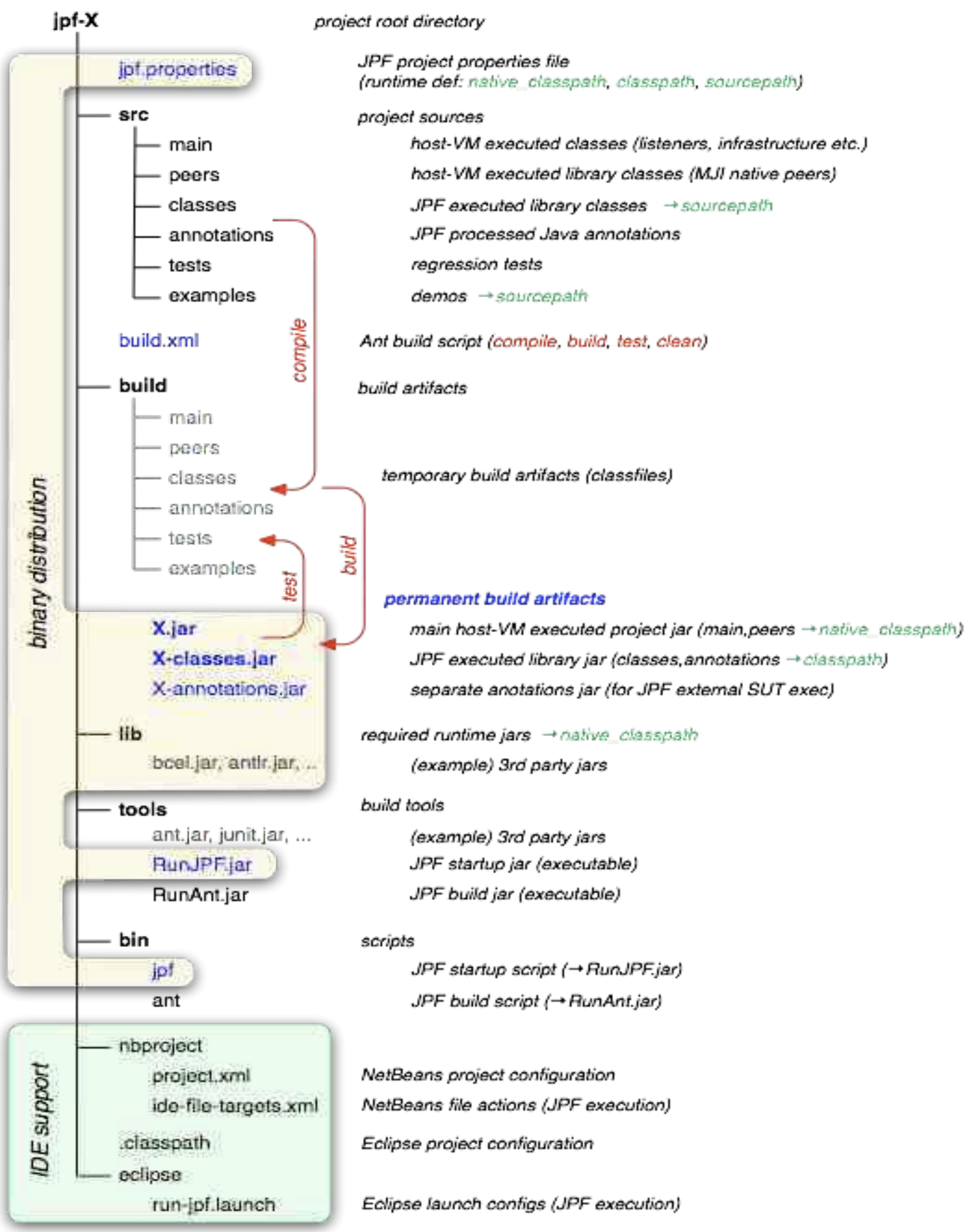


JPF Structure



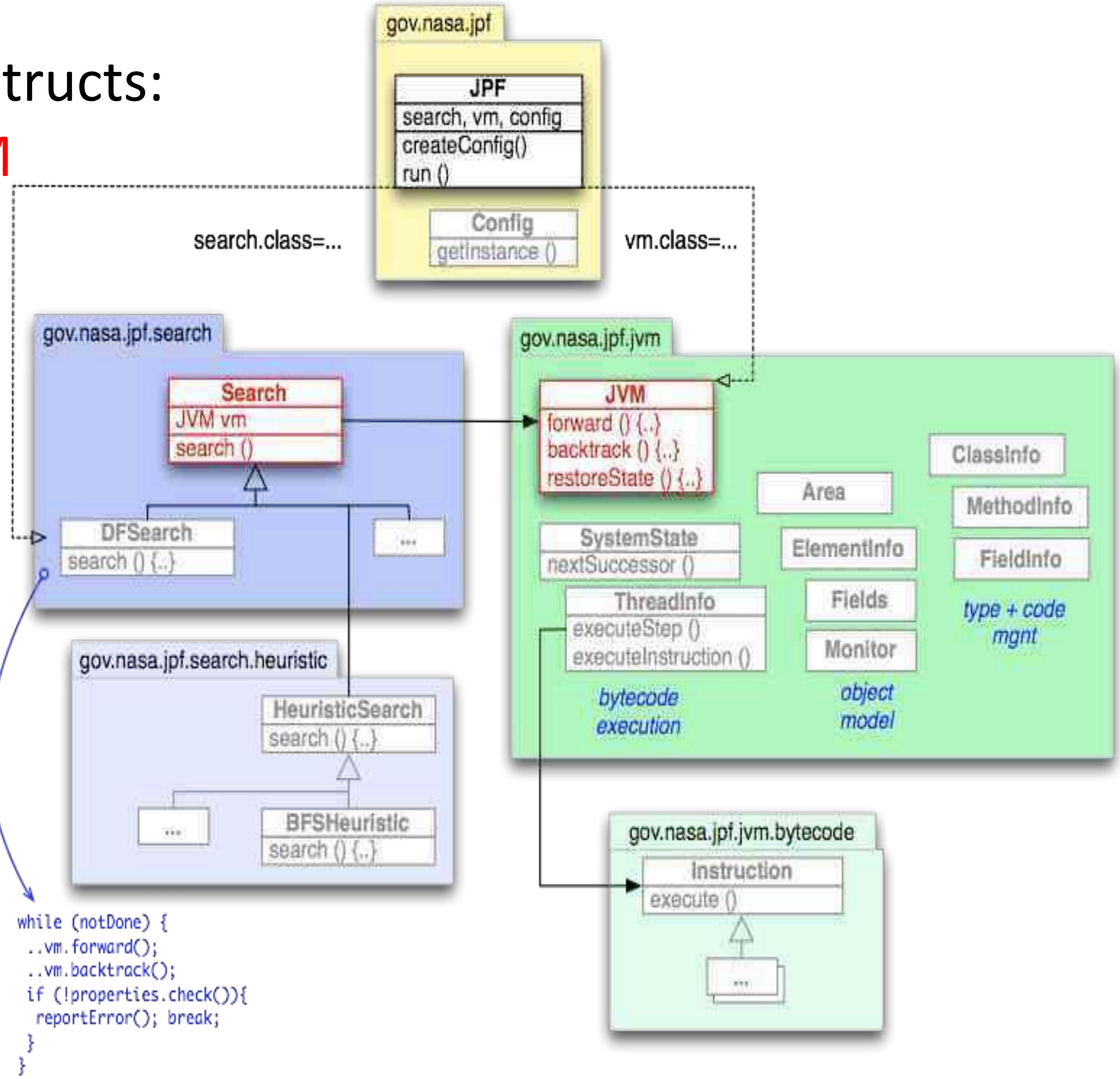
Directory Structure

- 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)



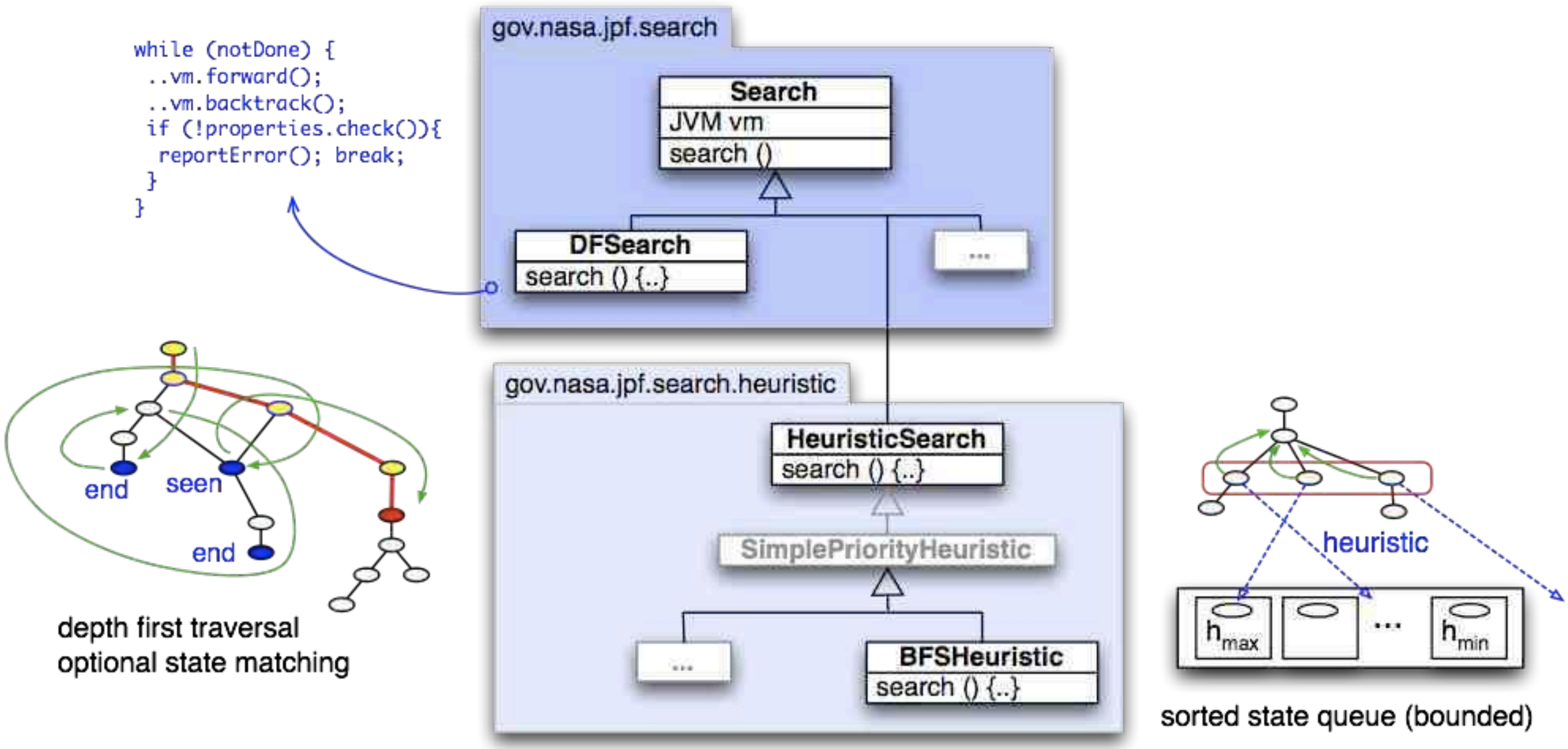
JPF Top-level Structure

- two major constructs: Search and JVM
- JVM produces program states
- Search is the JVM driver



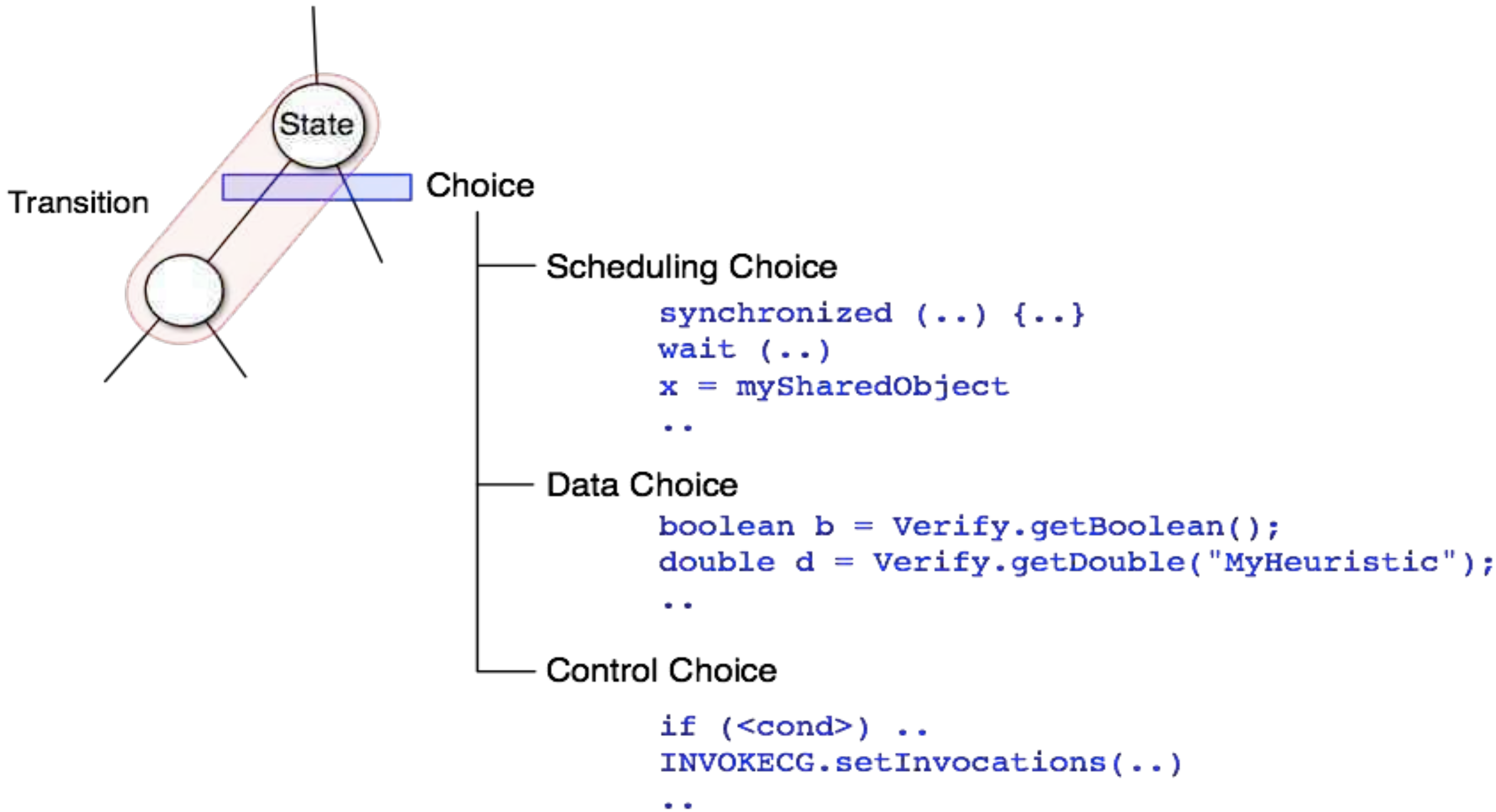
Search Policies

- 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



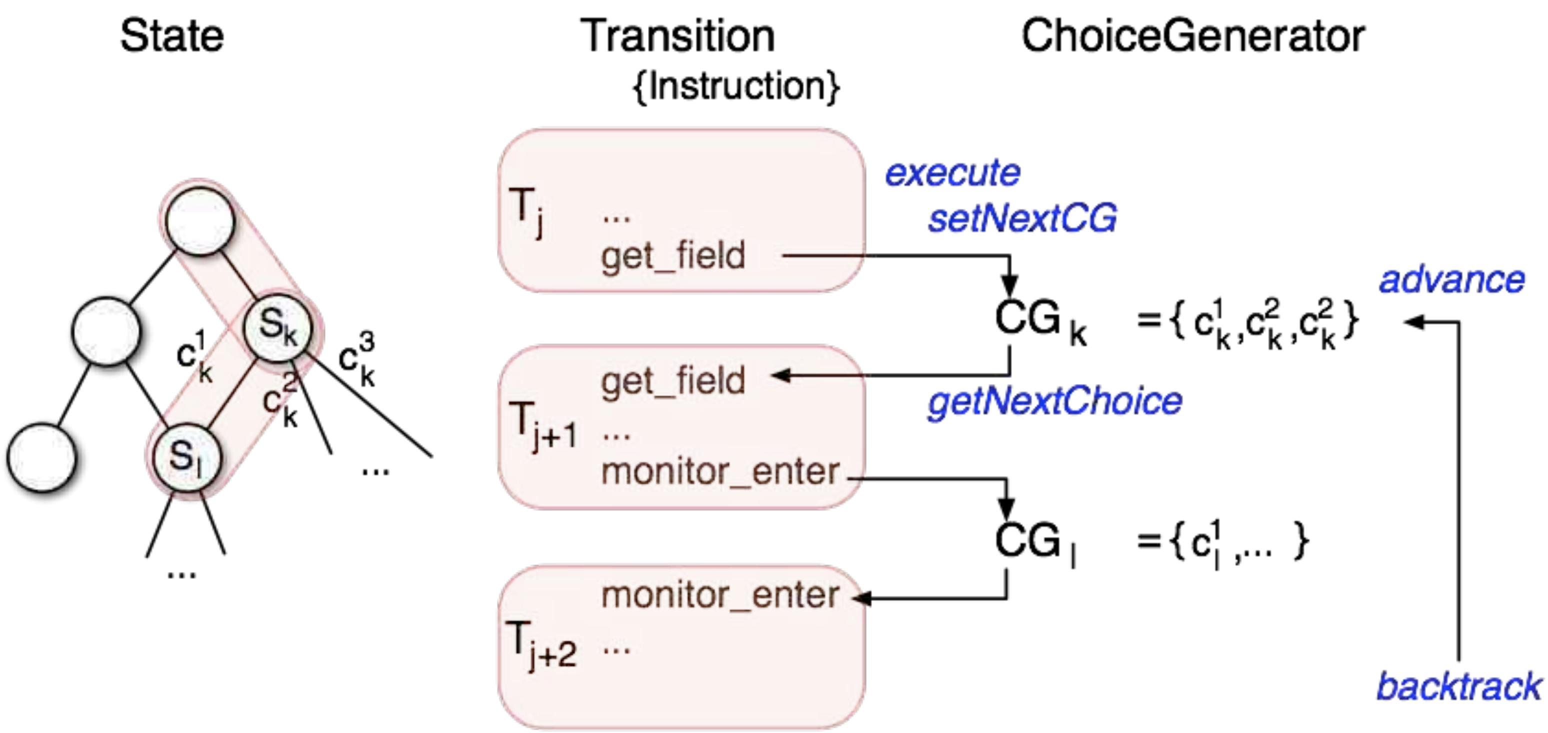
Exploring Choices

- model checker needs choices to explore state space
- there are many potential types of choices (scheduling, data, ..)
- choice types should not be hardwired in model checker



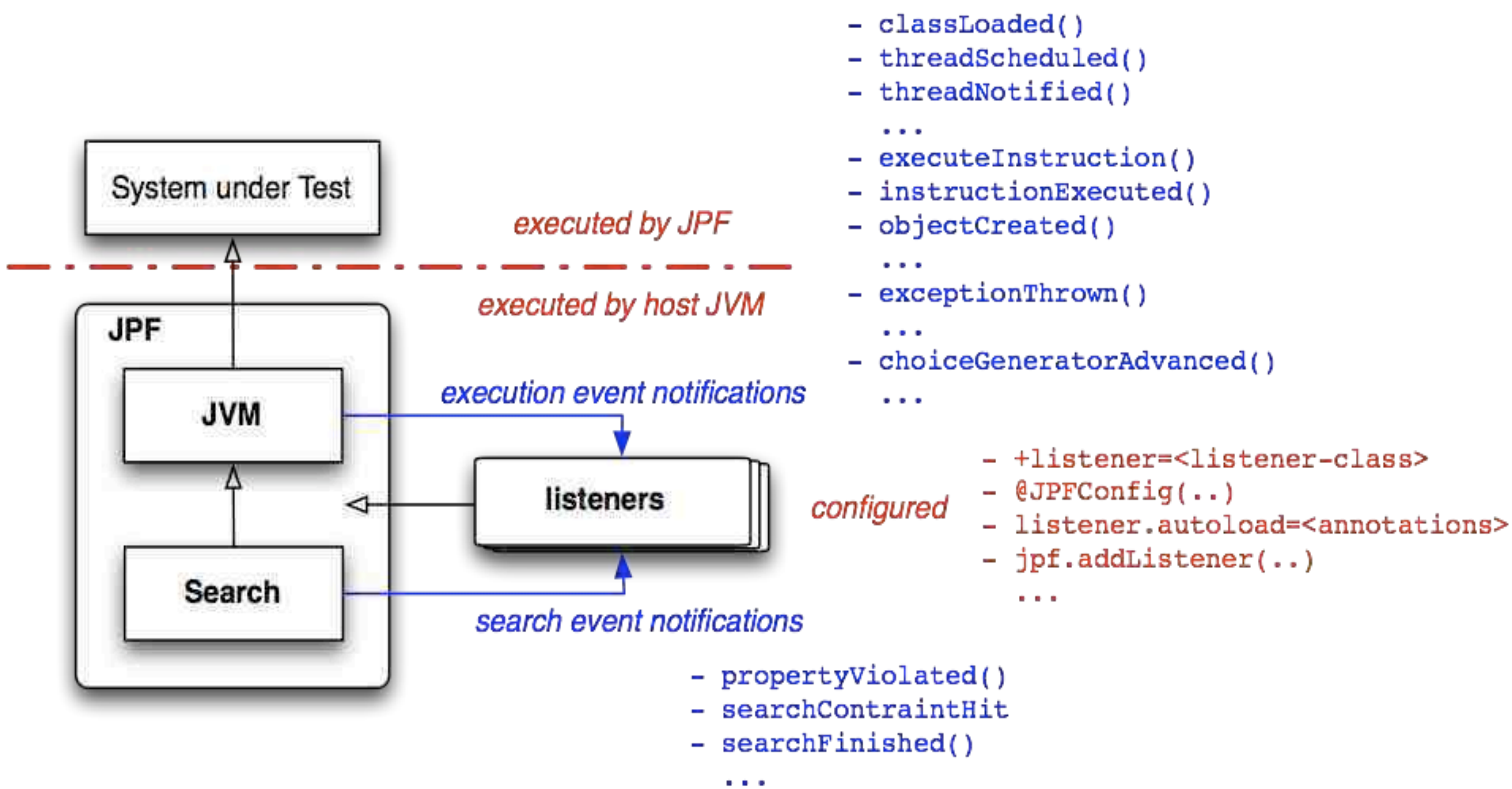
Choice Generators

- 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

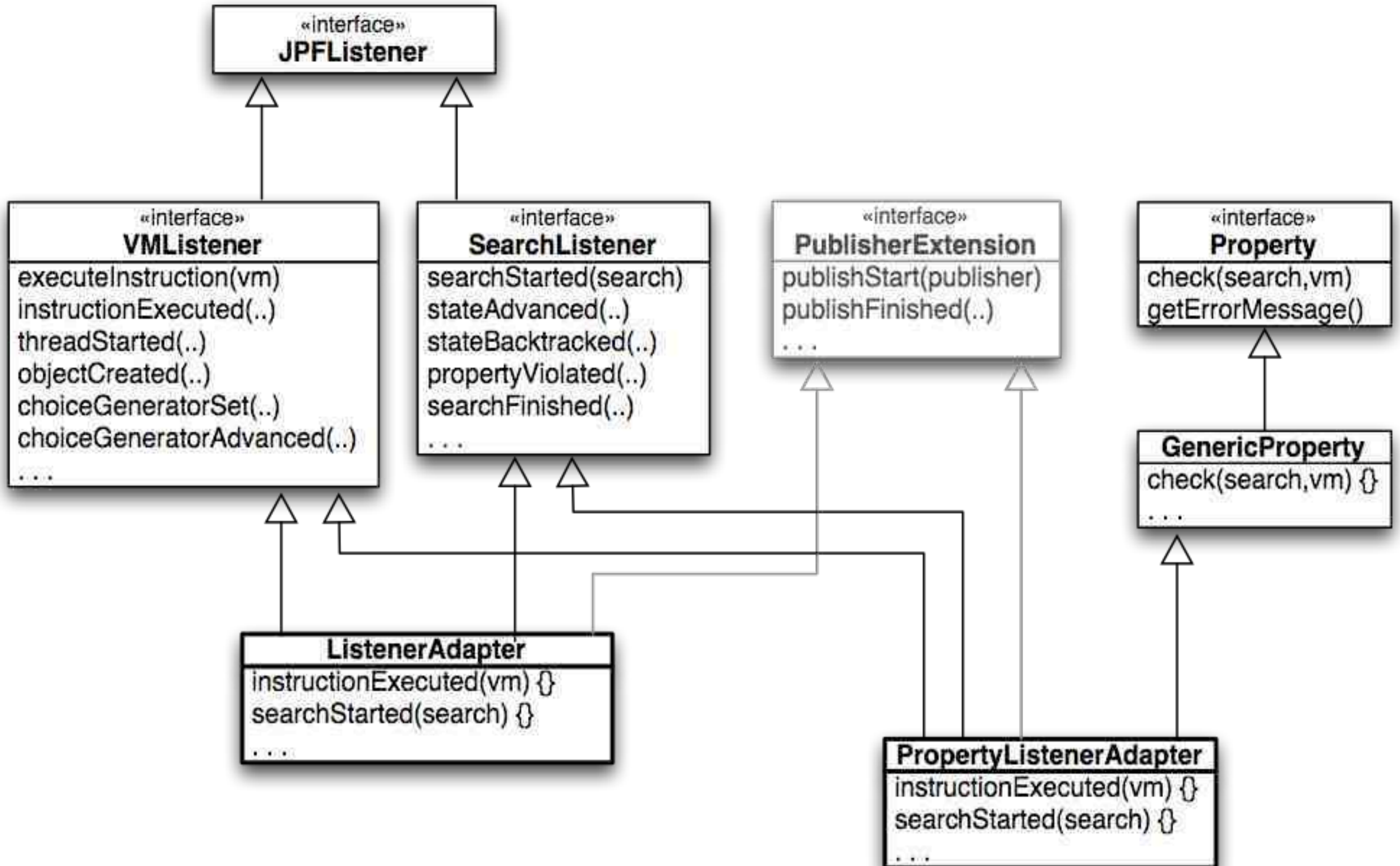


- Choice Generators are configurable as well, i.e. create your own

Listeners, the JPF Plugins



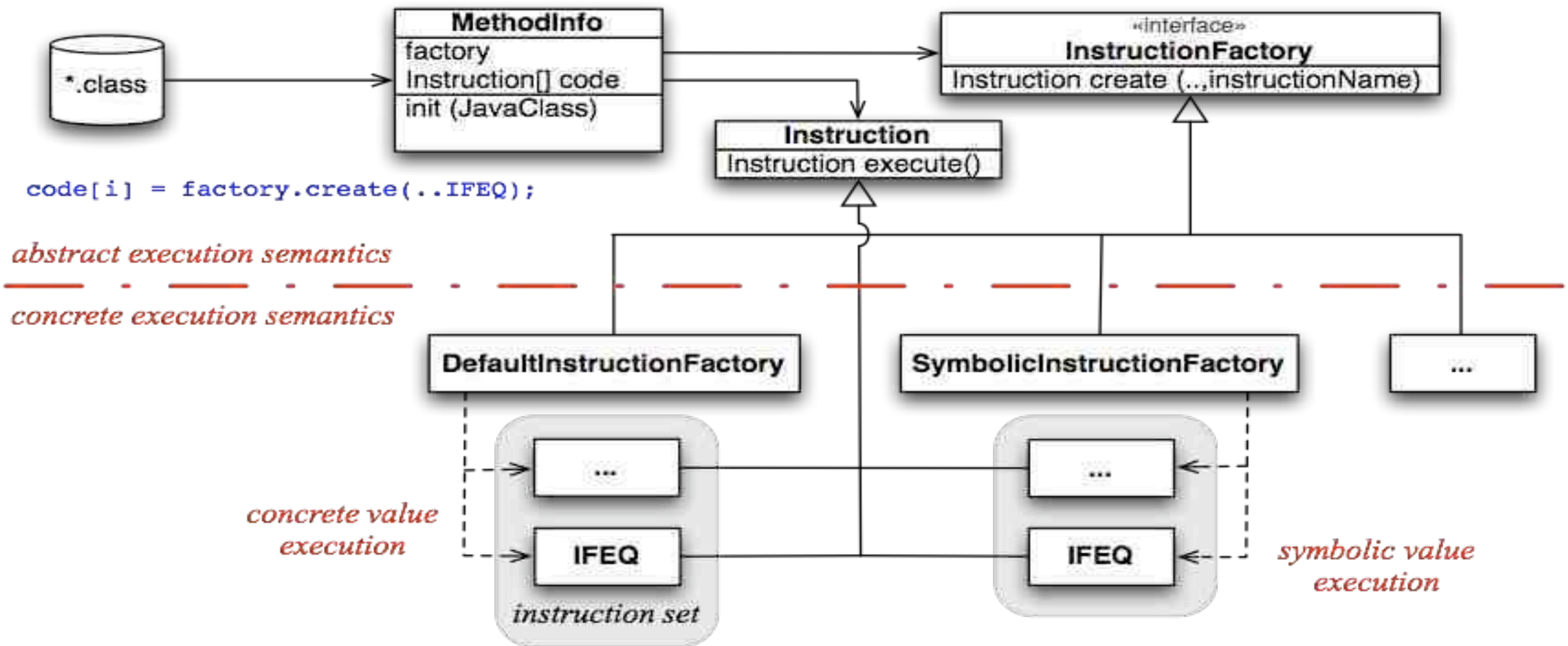
Listeners Implementation



Example Listener
Checking NonNull Annotation on Return

```
public class NonnullChecker extends ListenerAdapter {
...
public void executeInstruction (JVM vm) {
  Instruction insn = vm.getLastInstruction();
  ThreadInfo ti = vm.getLastThreadInfo();

  if (insn instanceof ARETURN) { // check @NonNull method returns
    ARETURN areturn = (ARETURN)insn;
    MethodInfo mi = insn.getMethodInfo();
    if (areturn.getReturnValue(ti) == null) {
      if (mi.getAnnotation("java.annotation.Nonnull") != null) {
        Instruction nextPc = ti.createAndThrowException(
                              "java.lang.AssertionError",
                              "null return from @Nonnull method: " +
                              mi.getCompleteName());
        ti.setNextPC(nextPC);
        return;
      }
    }
...
```

Bytecode Instruction Factories




```
Instruction execute (..){
  cond = popCondition();
  if (cond)
    return jumpTarget;
  else
    return getNextInsn();
}
```

```
Instruction execute (..){
  if (!firstStepInsn()){
    setNextCG(new PCChoiceGenerator());
    return this;
  }
  popCondition(); // not interested
  cond = getCG().getNextChoice();
  if (cond){...
    updatePathCondition(.., EQ);
    return jumpTarget;
  } else {...
    updatePathCondition(.., NE);
    return getNextInsn();
  }
}
```

Example – Bytecode Factory

- provide alternative Instruction classes for relevant bytecodes
- create & configure InstructionFactory that instantiates them

compiler

```
void notSoObvious(int x){
int a = x*50;
int b = 19437583;
int c = a;
for (int k=0; k<100; k++){
  c += b;
  System.out.println(c);
}}
...
notSoObvious(21474836);
```

```
...
[20] iinc
[21] goto 10
[10] iload_4
[11] bipush
[12] if_icmpge 22
[13] iload_3
[14] iload_2
[15] iadd
```

JPF configuration

```
vm.insn_factory.class =
 .numeric.NumericInstructionFactory
```

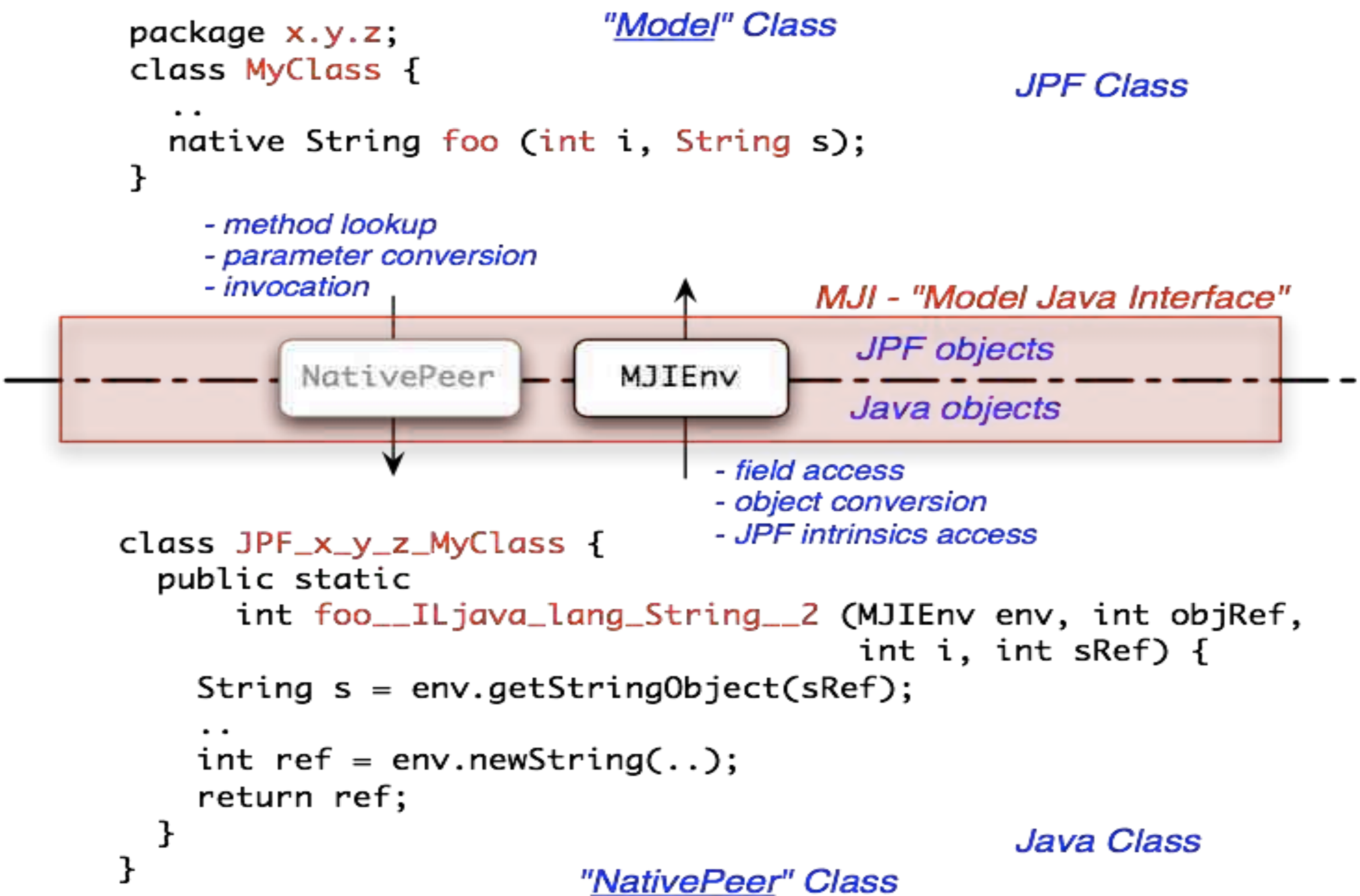
class loading

code execution
(by JPF)

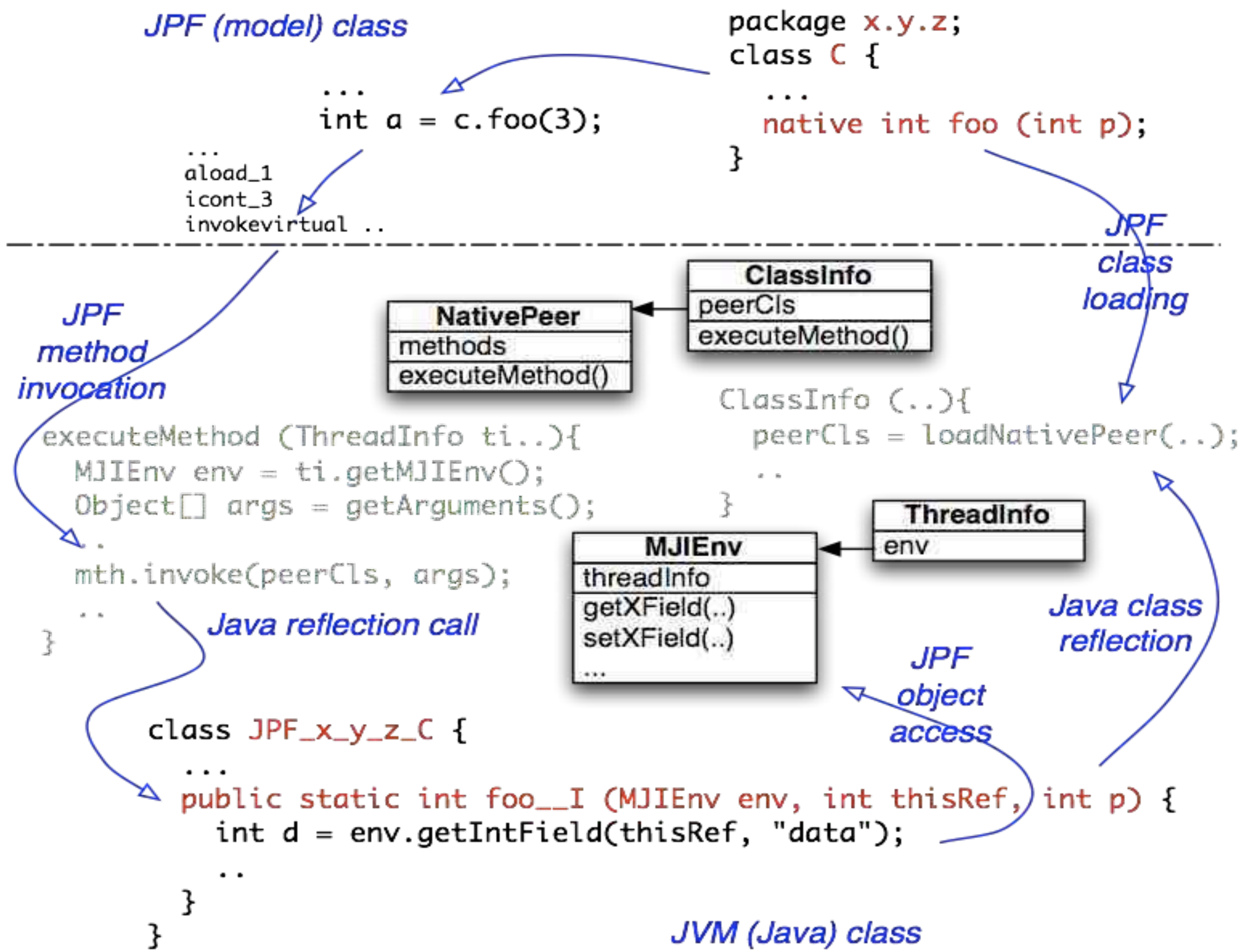
```
class IADD extends Instruction {
 Instruction execute (.., ThreadInfo ti) {
  int v1 = ti.pop();
  int v2 = ti.pop();
  int res = v1 + v2;
  if ((v1>0 && v2>0 && res<=0) …throw ArithmeticException..
```

MJI - Model-Java-Interface

- execution lowering from JPF executed code into JVM executed code



MJI - Implementation



MJI - Example

```
public class JPF_java_lang_String {
...
public static int indexOf__I__I (MJIEnv env, int objref, int c) {
   int vref = env.getReferenceField(objref, "value");
   int off = env.getIntField(objref, "offset");
   int len = env.getIntField(objref, "count");
   for (int i=0, j=off; i<len; i++, j++){
       if ((int)env.getCharArrayElement(vref, j) == c)
         return i;
   }
   return -1;
}
public static int toCharArray_____3C (MJIEnv env, int objref){
   ...
   int cref = env.newCharArray(len);
   for (int i=0, j=off; i<len; i++, j++){
     env.setCharArrayElement(cref, i, env.getCharArrayElement(vref, j));
    }
    return cref;
}
public static boolean matches__Ljava_lang_String_2__Z(MJIEnv env,int objRef, int regexRef) {
   String s = env.getStringObject(objRef);
   String r = env.getStringObject(regexRef);
    return s.matches(r);
}
```

Obtaining JPF

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

  (1) Get Mercurial

    (1) Eclipse Update site: http://cbes.javaforge.com/update

  (2) Get jpf-core

    (1) File – Import – Mercurial - Clone repository using Mercurial - Next

    (2) Specify http://babelfish.arc.nasa.gov/hg/jpf/jpf-core

    (3) Check the box for 'Search for .project files in clone and use them to create projects'

    (4) Finish

  (3) Build

    (1) Project – Properties - Select Builders - Ant Builder - Click Edit

    (2) Click JRE tab - Separate JREs - Installed JREs

    (3) Pick a JDK 1.6xxx. JRE will not find javac

Running JPF (1)

- Create site.properties in $(user.home)/.jpf
  - One line is enough for now:
  - $(user.home)/My Documents/workspace/jpf-core
- Install Eclipse Plugin (from the website description)
  - Ensure that you are running Eclipse >= 3.5 (Galileo)
  - In Eclipse go to Help -> Install New Software
  - In the new window selected "Add"
  - The name is up to you but, set "Location" to http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/install/eclipse-plugin/update/
  - From the "Work with:" drop down menu select the update site that you just entered from the previous step
  - Check the "Eclipse-JPF" check box, select "Next" and go through the install process.

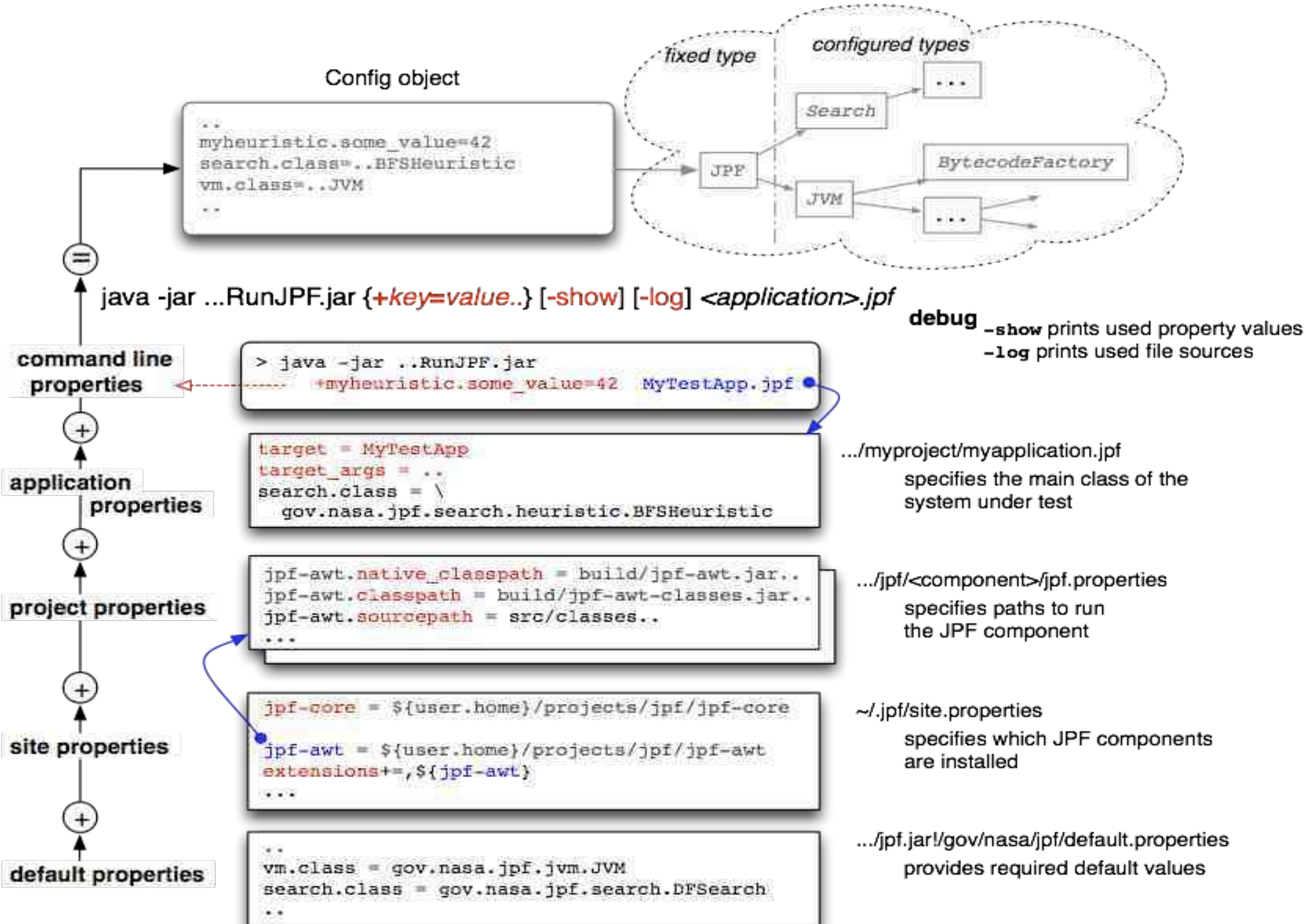
Running JPF (2)

- Right click on *.jpf file and pick “Verify”
  - Go to src/examples and right click on oldclassic.jpf
  - Should see a deadlock!

Configuring JPF

- 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` no space between key and '+' !
  - 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 ...`)

Configuration cont.



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)

JPF and JUnit

- derive your test cases from `gov.nasa.jpf.util.test.TestJPF`
- run normally under JUnit or from Ant `<junit ..>` task
- be aware of that test case is run by JVM *and* JPF

```
public class ConstTest extends TestJPF {
  static final String[] JPF_ARGS = { "+listener=.aprop.listener.ConstChecker" };

  //--- standard driver to execute single test methods
  public static void main(String[] args) {
    runTestsOfThisClass(args);
  }

  //--- the test methods
 @Test
  public void testStaticConstOk () {
    if (verifyNoPropertyViolation(JPF_ARGS)){
      ConstTest.checkThis();
  } }
 ...
```

Verification goal

code checked by JPF

Summer Projects

- 9 Google Summer of Code Projects
- 5 Ames internships
- 1 Fujitsu internship

http://babelfish.arc.nasa.gov/trac/jpf/wiki/summer-projects/start

Conclusions

- JPF is a highly extensible tool suite
- It is now 10 years old and has been open source for half that time
- So please, use it, change it…
- Contact for more information
  - Peter Mehlitz (peter.c.mehlitz@nasa.gov)
  - Neha Rungta (neha.s.rungta@nasa.gov)
  - Corina Pasareanu (Corina.S.Pasareanu@nasa.gov)
  - Willem Visser (willem@gmail.com)

JPF Tutorial – Part 2

Symbolic PathFinder – Symbolic Execution of Java Byte-code

Corina Pãsãreanu

Carnegie Mellon University/NASA Ames Research

Symbolic PathFinder (SPF)

- Combines symbolic execution with model checking and constraint solving to perform symbolic execution
- Used mainly for automated test-case generation
- Applies to executable models (e.g. Stateflow, UML state-machines) and to code
- Generates an optimized test suite that exercise all the behavior of the system under test
- Reports coverage (e.g. MC/DC)
- During test generation process, checks for errors
- Uses JPF's search engine
- Applications:
  – NASA (JSC's Onboard Abort Executive, PadAbort-1, Ames K9 Rover Executive, Aero, TacSat -- SCL script generation, testing of fault tolerant systems)
  – Fujitsu (testing of web applications, 60 000 LOC)
  – Academia (MIT, U. Minnesota, U. Nebraska, UT Austin, Politecnico di Milano, etc.)

Features

SPF handles:

- Inputs and operations on booleans, integers, reals
- Complex data structures (with polymorphism)
- Complex **`Math`** functions
- Pre-conditions, multi-threading
- Preliminary support for: String, bit-vector, and array operations

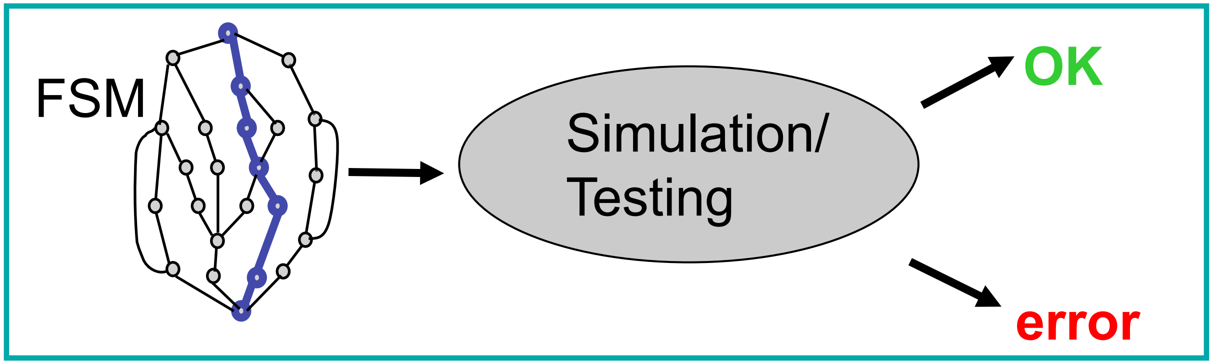
Allows for mixed concrete and symbolic execution

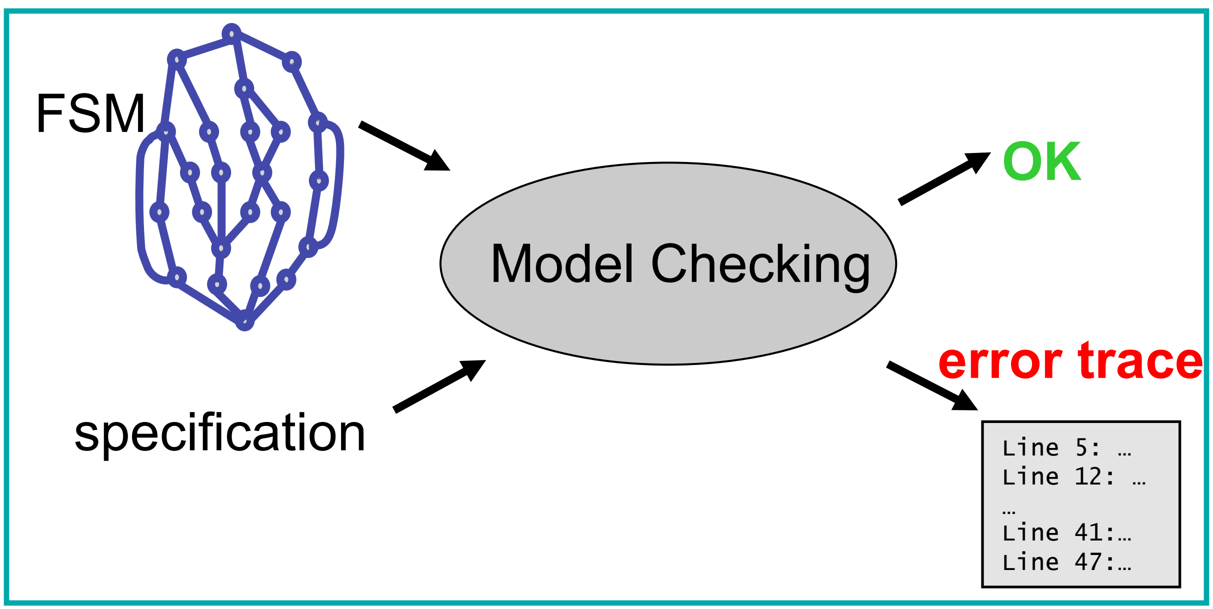
- Start symbolic execution at any point in the program and at any time during execution

Can be used:

- As customizable test case generator
  - User specifies coverage criterion, e.g..MC/DC
  - Search strategy, e.g. BFS or DFS
  - Output format, e.g. HTML tables or JUnit tests
- To generate counter-examples to safety properties in concurrent programs
- To prove light-weight properties of software
- For differential analysis between program versions [FSE'08]

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

Java PathFinder (JPF)

- Explicit state model checker for Java bytecode
  - Built on top of custom made Java virtual machine
- Focus is on finding bugs
  - Concurrency related: deadlocks, (races), missed signals etc.
  - Java runtime related: unhandled exceptions, heap usage, (cycle budgets)
  - Application specific assertions
- JPF uses a variety of scalability enhancing mechanisms
  - user extensible state abstraction & matching
  - on-the-fly partial order reduction
  - configurable search strategies
  - user definable heuristics (searches, choice generators)
- Recipient of NASA “Turning Goals into Reality” Award, 2003.
- Open sourced:
  - <javapathfinder.sourceforge.net>
  - ~14000 downloads since publication
- Largest application:
  - Fujitsu (one million lines of code)

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

Example – Standard Execution

Code that swaps 2 integers

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

Concrete Execution Path

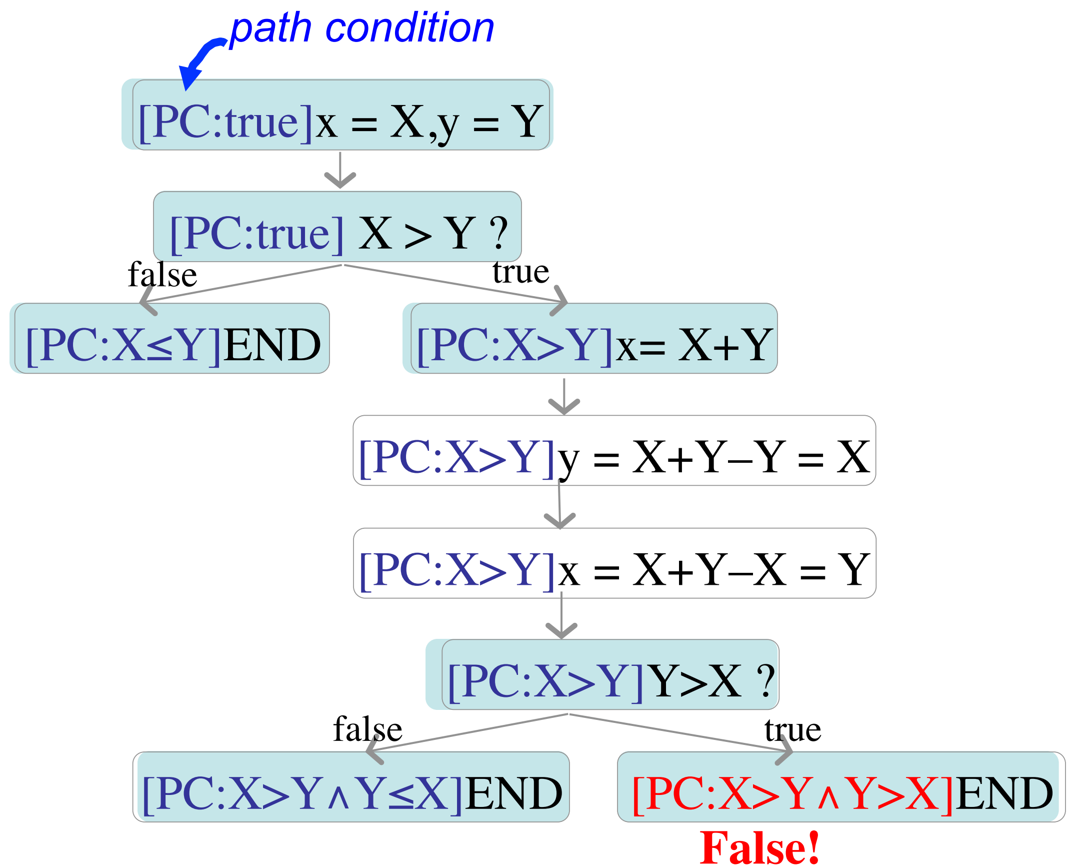
x = 1, y = 0
↓
1 > 0 ? true
↓
x = 1 + 0 = 1
↓
y = 1 – 0 = 1
↓
x = 1 – 1 = 0
↓
0 > 1 ? false

Example – Symbolic Execution

Code that swaps 2 integers:

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

Symbolic Execution Tree:



Solve path conditions → test inputs

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
  - **CVC3** http://www.cs.nyu.edu/acsys/cvc3/
  - 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

An Instruction Factory for Symbolic Execution of Byte-codes

- JPF core:
  - Implements concrete execution semantics based on stack machine model
  - For each method that is executed, maintains a set of **`Instruction`** objects created from the method byte-codes
- We created **`SymbolicInstructionFactory`**
  - Contains instructions for the symbolic interpretation of byte-codes
  - New **`Instruction`** classes derived from JPF's core
  - Conditionally add new functionality; otherwise delegate to super-classes
  - Approach enables simultaneous concrete/symbolic execution

Attributes for Storing Symbolic Information

- Program state:
  - A call stack/thread:
    - Stack frames/executed methods
    - Stack frame: locals & operands
  - The heap (values of fields)
  - Scheduling information

- We used previous experimental JPF extension of **slot attributes**
  - Additional, state-stored info associated with locals & operands on stack frame
- Generalized this mechanism to include **field attributes**
- Attributes are used to store symbolic values and expressions created during symbolic execution
- Attribute manipulation done mainly inside JPF core
  - We only needed to override instruction classes that create/modify symbolic information
  - E.g. numeric, compare-and-branch, type conversion operations
- Sufficiently general to allow arbitrary value and variable attributes
  - Could be used for implementing other analyses
  - E.g. keep track of physical dimensions and numeric error bounds or perform DART-like execution ("concolic")

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

```
public class PCChoiceGenerator
     extends IntIntervalGenerator {
     PathCondition[] PC;
     …
}
```

Example: IADD

Concrete execution of IADD byte-code:

```
public class IADD extends
   Instruction { …
 public Instruction execute(…
   ThreadInfo th){
   int v1 = th.pop();
   int v2 = th.pop();
   th.push(v1+v2,…);
   return getNext(th);
 }
}
```

Symbolic execution of IADD byte-code:

```
public class IADD extends
   ….bytecode.IADD { …
 public Instruction execute(…
   ThreadInfo th){
   Expression sym_v1 = ….getOperandAttr(0);
   Expression sym_v2 = ….getOperandAttr(1);
   if (sym_v1 == null && sym_v2 == null)
     // both values are concrete
     return super.execute(… th);
   else {
     int v1 = th.pop();
     int v2 = th.pop();
     th.push(0,…); // don't care
     …
     ….setOperandAttr(Expression._plus(
         sym_v1,sym_v2));
     return getNext(th);
   }
 }
}
```

Example: IFGE

Concrete execution of IFGE byte-code:

```
public class IFGE extends
    Instruction { …
 public Instruction execute(…
    ThreadInfo th){
    cond = (th.pop() >=0);
    if (cond)
     next = getTarget();
    else
     next = getNext(th);
    return next;
 }
}
```

Symbolic execution of IFGE byte-code:

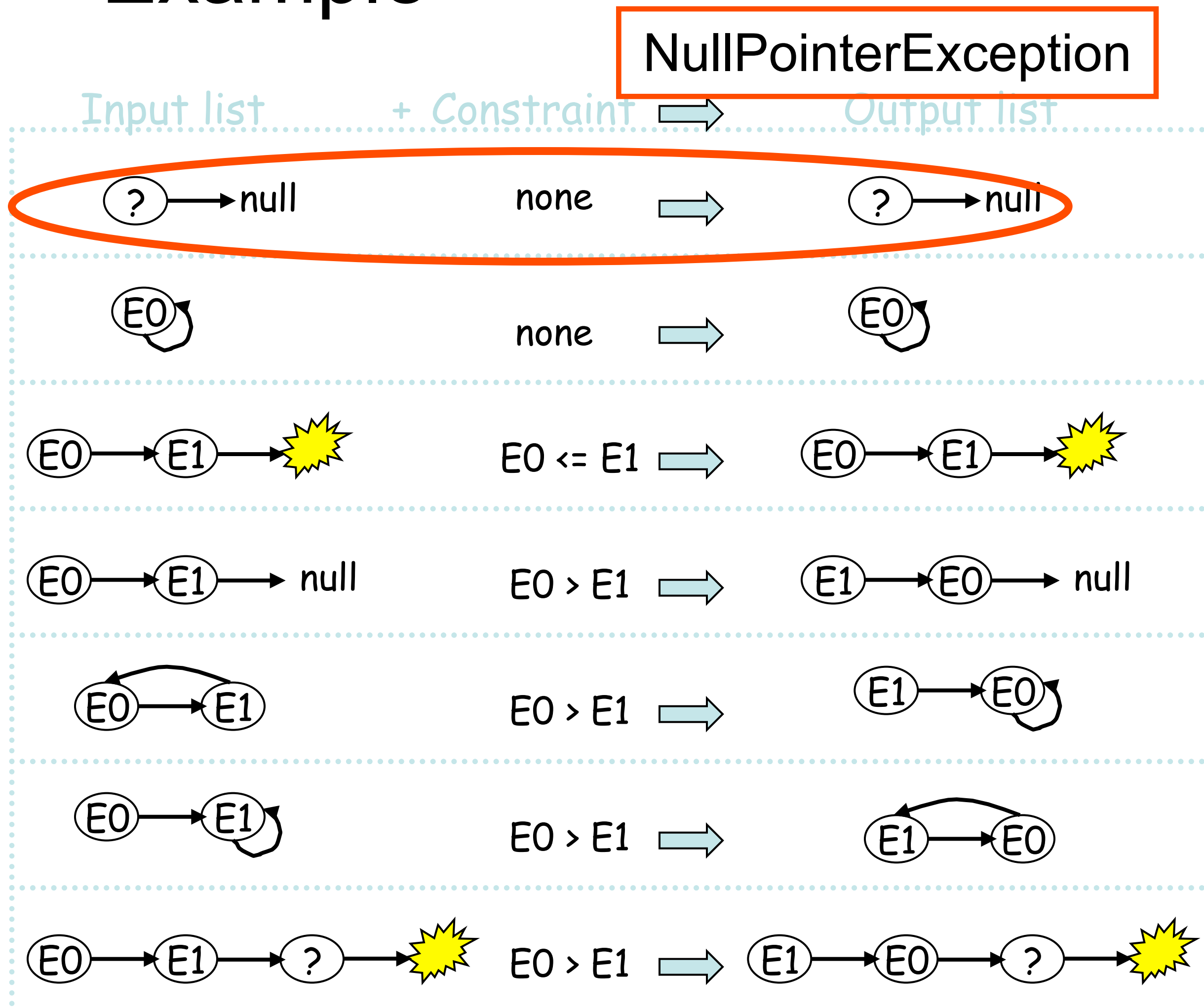
```
public class IFGE extends
   ….bytecode.IFGE { …
 public Instruction execute(…
   ThreadInfo th){
   Expression sym_v = ….getOperandAttr();
   if (sym_v == null)
     // the condition is concrete
     return super.execute(… th);
   else {
     PCChoiceGen cg = new PCChoiceGen(2);…
     cond = cg.getNextChoice()==0?false:true;
     if (cond) {
        pc._add_GE(sym_v,0);
        next = getTarget();
     }
     else {
        pc._add_LT(sym_v,0);
        next = getNext(th);
     }
     if (!pc.satisfiable()) … // JPF backtrack
     else cg.setPC(pc);
     return next;
   } } }
```

Handling Input Data Structures

- Lazy initialization for recursive data structures [TACAS'03] and arrays [SPIN'05]
- JPF-core used
  - To generate and explore the symbolic execution tree
  - Non-determinism handles aliasing
    - Explore different heap configurations explicitly
- Implementation:
  - Lazy initialization via modification of GETFIELD, GETSTATIC bytecode instructions
  - Listener to print input heap constraints and method effects (outputs)

Example

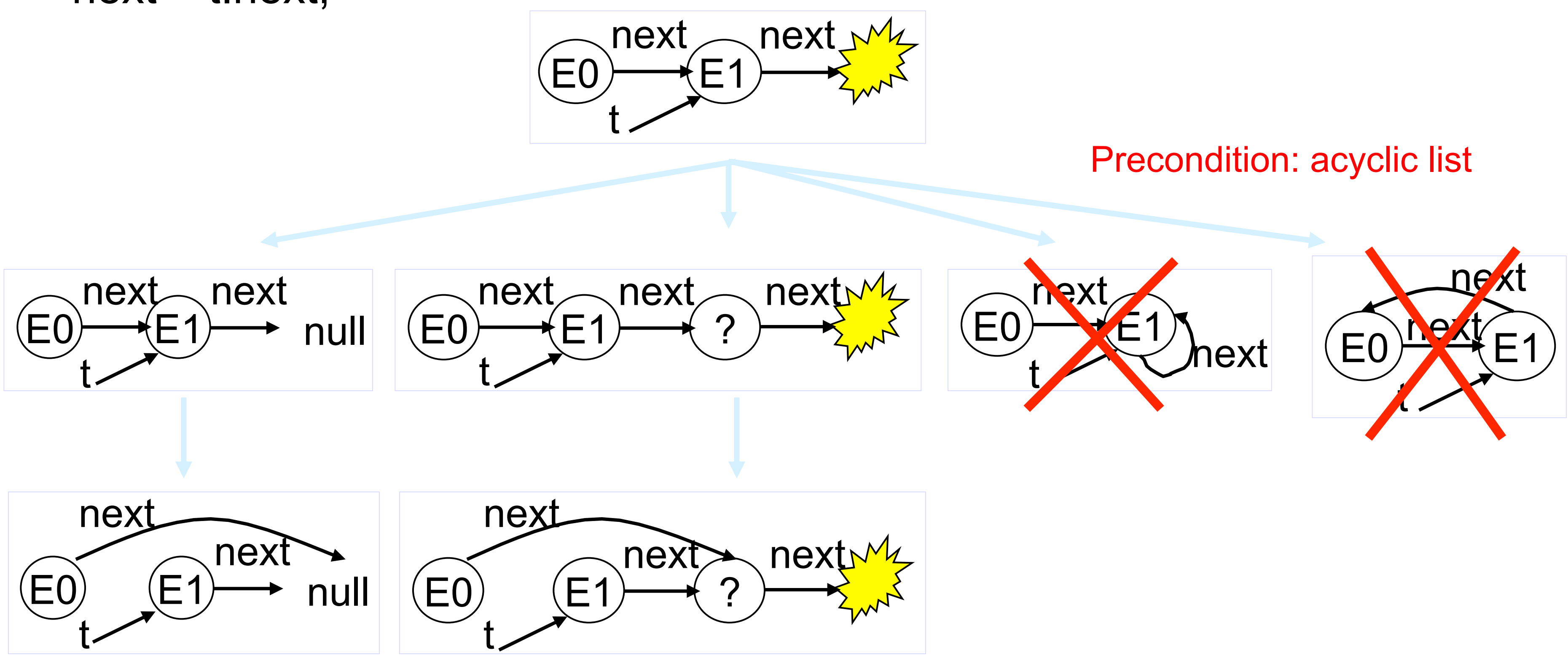
```
class Node {
    int elem;
    Node next;

    Node swapNode() {
      if (next != null)
        if (elem > next.elem) {
          Node t = next;
          next = t.next;
          t.next = this;
          return t;
        }
      return this;
    }
}
```



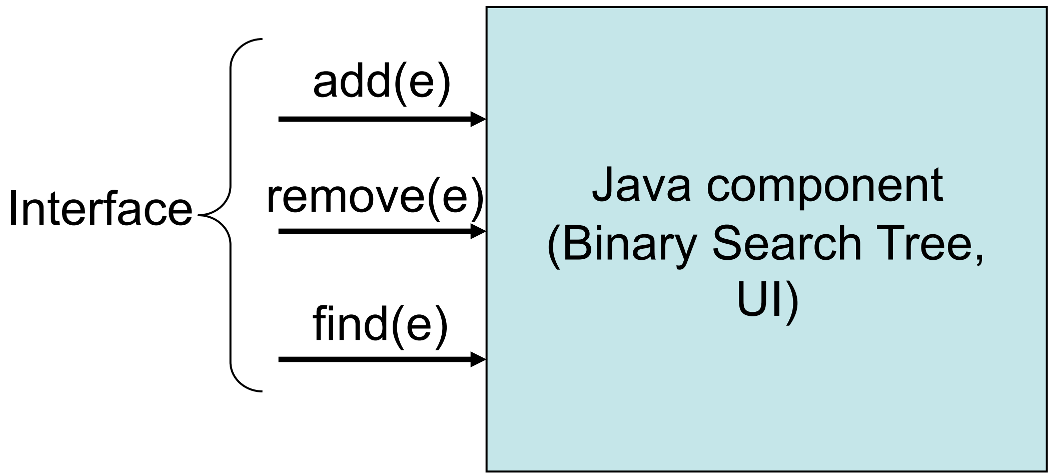
Lazy Initialization (illustration)

consider executing

next = t.next;



Generating Test Sequences with Symbolic PathFinder



Generated test sequence:

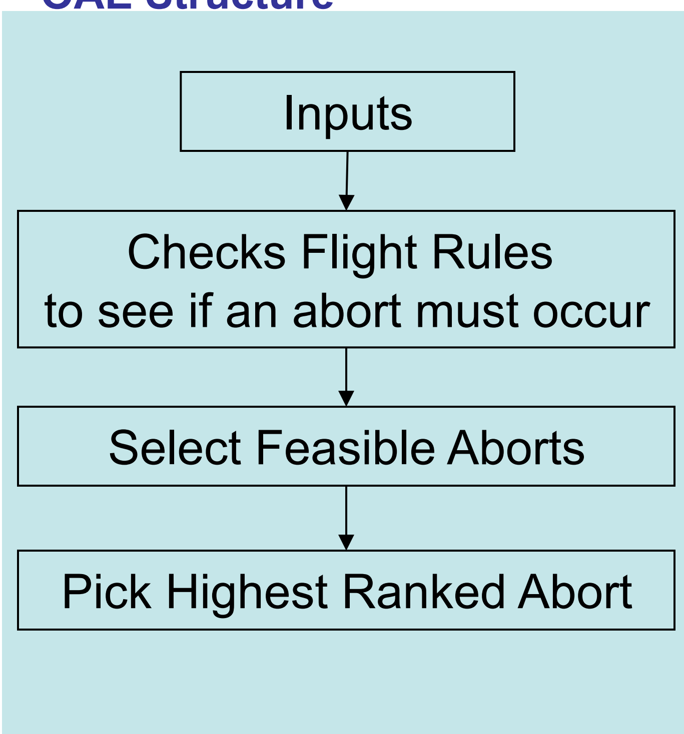
```
BinTree t = new BinTree
();
 t.add(1);
 t.add(2);
 t.remove(1);
```

- Listener SymbolicSequenceListener used to generate JUnit tests:
  - method sequences (up to user-specified depth)
  - method parameters
- JUnit tests can be run directly by the developers
- Measure coverage
- Support for abstract state matching
- Extract specifications

Application: Onboard Abort Executive (OAE)

Prototype for CEV ascent abort handling being developed by JSC GN&C

OAE Structure



Paper at ISSTA conference 2008

Results

- Baseline
  - Manual testing: time consuming (~1 week)
  - Guided random testing could not cover all aborts
- Symbolic PathFinder
  - Generates tests to cover all aborts and flight rules
  - Total execution time is < 1 min
  - Test cases: 151 (some combinations infeasible)
  - Errors: 1 (flight rules broken but no abort picked)
  - Found major bug in new version of OAE
  - Flight Rules: 27 / 27 covered
  - Aborts: 7 / 7 covered
  - Size of input data: 27 values per test case
- Integration with End-to-end Simulation
  - Input data is constrained by environment/physical laws
    Example: inertial velocity can not be 24000 ft/s when the geodetic altitude is 0 ft
  - Need to encode these constraints explicitly
  - Solution: Use simulation runs to get data correlations -- as a result, we eliminated some test cases that were impossible

RSE

Generated Test Cases and Constraints

Test cases:

// Covers Rule: FR A_2_A_2_B_1: Low Pressure Oxodizer Turbopump speed limit exceeded
// **Output: Abort:IBB**
CaseNum 1;
CaseLine in.stage_speed=3621.0;
CaseTime 57.0-102.0;

// Covers Rule: FR A_2_A_2_A: Fuel injector pressure limit exceeded
// **Output: Abort:IBB**
CaseNum 3;
CaseLine in.stage_pres=4301.0;
CaseTime 57.0-102.0;
…

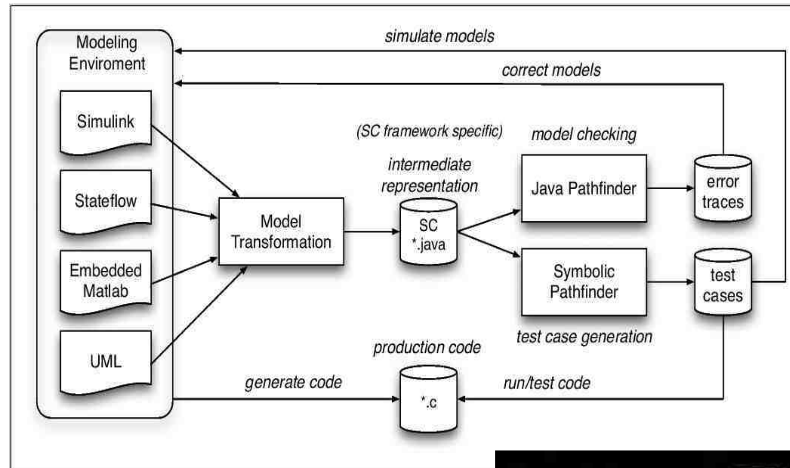
Constraints:

//Rule: FR A_2_A_1_A: stage1 engine chamber pressure limit exceeded Abort:IA
PC (~60 constraints):
in.geod_alt(9000) < 120000 && in.geod_alt(9000) < 38000 && in.geod_alt(9000) < 10000 &&
in.pres_rate(-2) >= -2 && in.pres_rate(-2) >= -15 &&
in.roll_rate(40) <= 50 && in.yaw_rate(31) <= 41 && in.pitch_rate(70) <= 100 && …

Test-Case Generation for UML and Simulink/Stateflow Models

Generic Framework:

- Enables:
  - Analysis for UML and Simulink/Stateflow models;
  - Test case generation to achieve high degree of coverage (state, transition, path, MC/DC)
  - Pluggable semantics: implements both Stateflow and UML state-chart semantics
  - Study of integration and interoperability issues between **heterogeneous models**
- Technologies:
  - Model transformation (Vanderbilt U. collaborators)
  - Model analysis (Java Pathfinder model checker)
  - Test-case generation (Symbolic Pathfinder)
- JPF/SPF seamlessly integrated in Matlab environment
- Demonstrated on:
  - Orion's Pad Abort--1; Ares-Orion communication
- Could handle features not supported currently by commercial tools (MathWorks Design Verifier, T-VEC)



Orion orbits the moon (Image Credit: Lockheed Martin).

Shown: Framework for model-based analysis and test case-generation; test cases used to test the generated code and to discover un-wanted discrepancies between models and code.

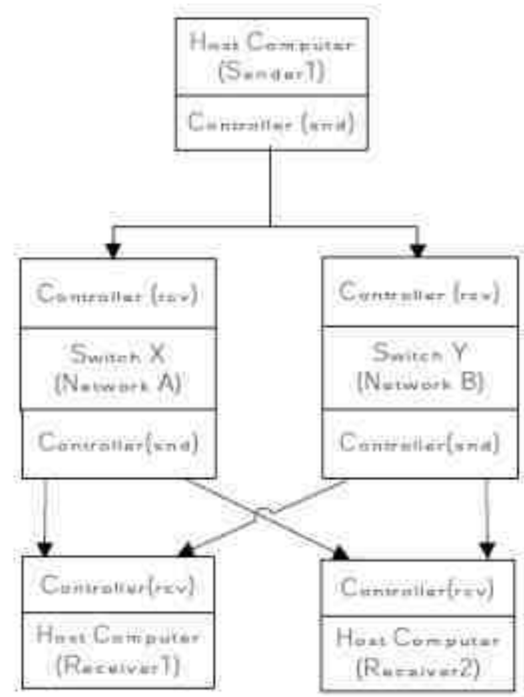
Application: Test Generation for the TTEthernet Protocol

• TTEthernet is a fault tolerant version of the Ethernet protocol that is/will be used by NASA in upcoming space networks to assure reliable network communications.

• We have modeled parts of TTEthernet for our work on automated test case generation for fault tolerant protocols.

• Test automation can reduce software costs and also increase software reliability by enabling more thorough testing.

• We implemented a PVS model of a basic version of the TTEthernet protocol (in collaboration with Langley)
• We provided a framework for translating models into input language of verification tools; it allows:
  -- the filtering of test cases to satisfy the various fault hypothesis and
  -- the verification of fault-tolerant properties
• We demonstrated test case generation for TTEthernet's Single Fault Hypothesis



<u>Shown</u>: Minimal configuration for testing agreement in TTEthernet

Tool Information

- SPF is available from
  http://babelfish.arc.nasa.gov/trac/jpf
- You will need both jpf-core and jpf-symbc
- Tool documentation can be found at:
  http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc

- File .jpf/site.properties must contain the following lines:

```
# modify to point to the location of jpf-symbc on your computer
jpf-symbc = ${user.home}/workspace/jpf-symbc
extensions+=,${jpf-symbc}
```

Example.java

```
package examples;

public class Example {
   public static void main (String[] args) {
        Example ex= new Example();
        ex.foo(2, 1);
   }
   public int foo(int x, int y){
        if (x>y) {
                System.out.println("First");
                return x;
        }
        else {
                System.out.println("Second");
                        return y;
        }
   }
}
```

Example.jpf

```
target=examples.Example

# here write your own classpath and un-comment
# classpath=/home/user_name/example-project/bin

symbolic.method= examples.Example.foo(sym#con)

# listener to print information: PCs, test cases
listener = gov.nasa.jpf.symbc.SymbolicListener

# The following JPF options are usually used for SPF as well:

# no state matching
vm.storage.class=nil
# do not stop at first error
search.multiple_errors=true
```

Results

```
Running Symbolic PathFinder ...
symbolic.dp=choco
symbolic.minint=-100
symbolic.maxint=100
symbolic.minreal=-1000.0
symbolic.maxreal=1000.0
symbolic.undefined=0
JavaPathfinder v5.x - (C) RIACS/NASA Ames Research Center
====================================================== system under test
application: examples/Example.java
====================================================== search started: 7/9/10 8:23 AM
First
PC # = 1
x_1_SYMINT[2] > CONST_1
SPC # = 0
****************************
Second
PC # = 1
x_1_SYMINT[-100] <= CONST_1
SPC # = 0
****************************
====================================================== Method Summaries
Symbolic values: x_1_SYMINT
foo(2,2)  --> Return Value: x_1_SYMINT
foo(-100,-100)  --> Return Value: 1
====================================================== Method Summaries (HTML)
<h1>Test Cases Generated by Symbolic Java Path Finder for foo (Path Coverage) </h1>
<table border=1>
<tr><td>x_1_SYMINT</td></tr>
<tr><td>2</td><td>2</td><td>Return Value: x_1_SYMINT</td></tr>
<tr><td>-100</td><td>-100</td><td>Return Value: 1</td></tr>
</table>
====================================================== results
no errors detected
====================================================== statistics
elapsed time:       0:00:02
states:             new=4, visited=0, backtracked=4, end=2
search:             maxDepth=3, constraints=0
choice generators:  thread=1, data=2
heap:               gc=3, new=271, free=22
instructions:       2875
max memory:         81MB
```

Options

- Specify the search strategy (default is DFS)

search.class = .search.heuristic.BFSHeuristic

- Limit the search depth (number of choices along the path)

search.depth_limit = 10

- You can specify multiple methods to be executed symbolically as follows:

symbolic.method=<list of methods to be executed symbolically separated by ",">

- You can pick which decision procedure to choose (if unspecified, choco is used as default):

symbolic.dp=choco
symbolic.dp=iasolver
symbolic.dp=cvc3
symbolic.dp=cvc3bitvec
symbolic.dp=no_solver (explores an over-approximation of program paths; similar to a CFG traversal)

- A new option was added to implement lazy initialization (see [TACAS'03] paper)

symbolic.lazy=on
(default is off) -- for now it is incompatible with Strings

- New options have been added, to specify min/max values for symbolic variables and also to give the default for don't care values.

symbolic.minint=-100
symbolic.maxint=100
symbolic.minreal=-1000.0
symbolic.maxreal=1000.0
symbolic.undefined=0

- Globals (i.e. fields) can also be specified to be symbolic, via special annotations; annotations are also used to specify preconditions (see src/tests/ExSymExePrecondAndMath.java).
- See also other examples in src/tests and src/examples.

Comparison with Our Previous Work

JPF– SE [TACAS'03,TACAS'07]:

- http://javapathfinder.sourceforge.net (symbolic extension)
- Worked by code instrumentation (partially automated)
- Quite general but may result in sub-optimal execution
  - For each instrumented byte-code, JPF needed to check a set of byte-codes representing the symbolic counterpart
- Required an approximate static type propagation to determine which byte-code to instrument [Anand et al.TACAS'07]
  - No longer needed in the new framework, since symbolic information is propagated dynamically
  - Symbolic JPF always maintains the most precise information about the symbolic nature of the data
- [data from Fujitsu: Symbolic JPF is 10 times faster than JPF--SE]

Related Work

- Model checking for test input generation [Gargantini & Heitmeyer ESEC/FSE'99, Heimdahl et al. FATES'03, Hong et al. TACAS'02]
  - BLAST, SLAM
- Extended Static Checker [Flanagan et al. PLDI'02]
  - Checks light-weight properties of Java
- Symstra [Xie et al. TACAS'05]
  - Dedicated symbolic execution tool for test sequence generation
  - Performs sub-sumption checking for symbolic states
- Symclat [d'Amorim et al. ASE'06]
  - Context of an empirical comparative study
  - Experimental implementation of symbolic execution in JPF via changing all the byte-codes
  - Did not use attributes, instruction factory; handled only integer symbolic inputs
- Bogor/Kiasan [ASE'06]
  - Similar to JPF—SE, uses "lazier" approach
  - Does not separate between concrete and symbolic data and doesn't handle Math constraints
- DART/CUTE/PEX [Godefroid et al. PLDI'05, Sen et al. ESEC/FSE'05]
  - Do not handle multi-threading; performs symbolic execution along concrete execution
  - We use concrete execution to set-up symbolic execution
- Execution Generated Test Cases [Cadar & Engler SPIN'05]
- Other hybrid approaches:
  - Testing, abstraction, theorem proving: better together! [Yorsh et al. ISSTA'06]
  - SYNERGY: a new algorithm for property checking [Gulavi et al. FSE'06]
- Etc.

Selected Bibliography

[ASE'10] "Symbolic PathFnder: Symbolic Execution for Java Bytecode" – tool paper, *C. Pasareanu and N. Rungta*

[ISSTA'08] "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software", *C. Pãsãreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry*, *S. Person*, *M. Pape*

[FSE'08] "Differential Symbolic Execution", *S. Person, M. Dwyer, S. Elbaum, C. Pãsãreanu*

[TACAS'07] "JPF—SE: A Symbolic Execution Extenion to Java PathFinder", *S. Anand, C. Pãsãreanu, W. Visser*

[SPIN'04] "Verification of Java Programs using Symbolic Execution and Invariant Generation", *C. Pãsãreanu, W. Visser*

[TACAS'03] "Generalized Symbolic Execution for Model Checking and Testing", *S. Khurshid, C. Pãsãreanu, W. Visser*

Summary

- Symbolic PathFinder
  - Non-standard interpretation of byte-codes
  - Symbolic information propagated via attributes associated with program variables, operands, etc.
  - Available from http://babelfish.arc.nasa.gov/trac/jpf (jpf-symbc)
- Applications at NASA, industry, academia
- Some current work:
  - Parallel Symbolic Execution [ISSTA'10]
  - String Analysis – with contributions from Fujitsu
  - Load Testing
  - Concolic execution (JPF's concolic extension)

    Contributed by MIT: David Harvison & Adam Kiezun

    http://people.csail.mit.edu/dharv/jfuzz

Questions?