
Automatic Testcase Generation for Flight Software

**NASA Planetary Spacecraft
Fault Management Workshop
April 2008**

**David Bushnell, RIACS/NASA ARC
Corina Pasareanu, Perot Systems/NASA ARC
Ryan Mackey, NASA JPL**



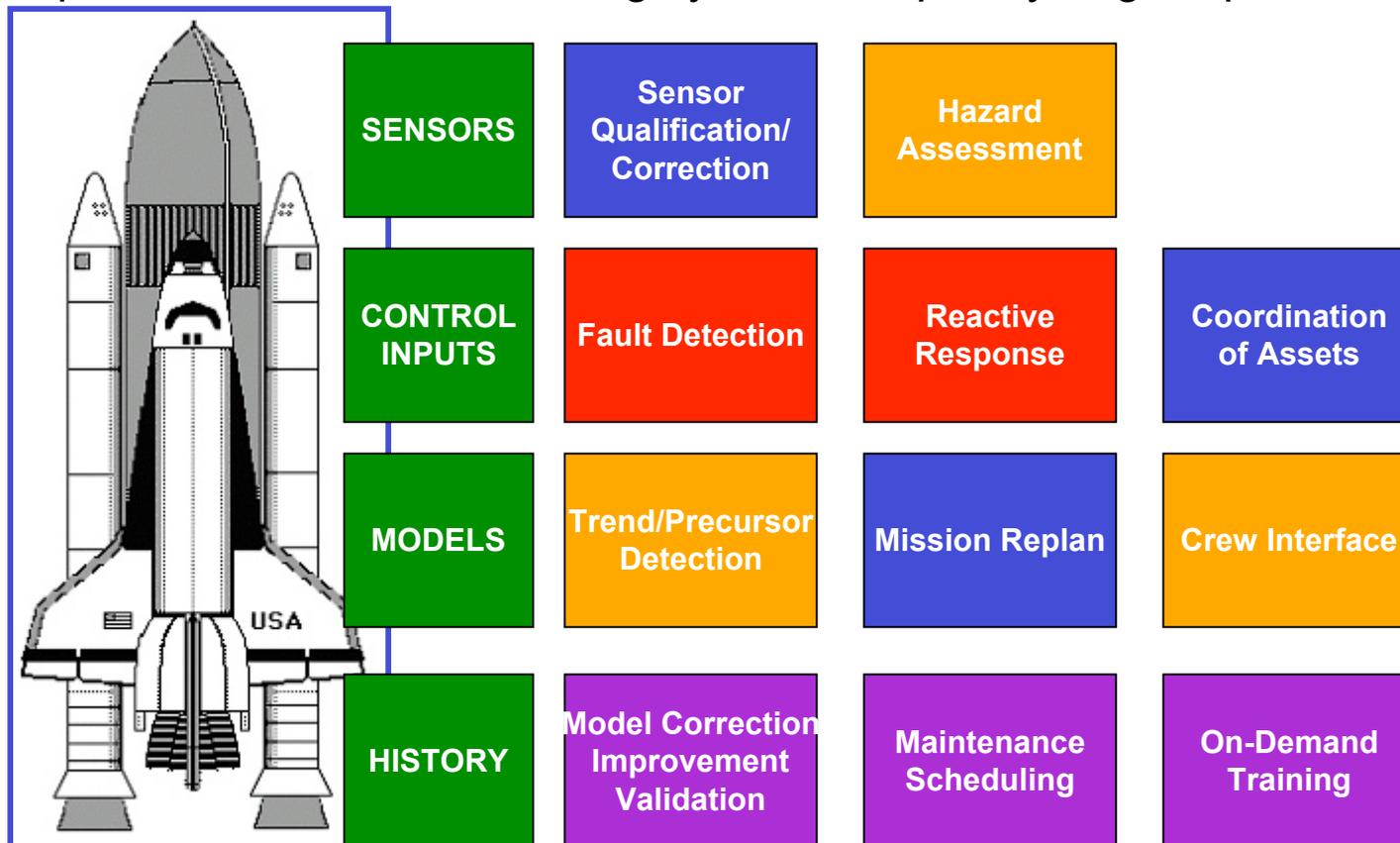
TacSat 3, ISHM, and VSM

- **Faults are a fact of life in engineered systems**
- **NASA needs better ways of handling and recovering from faults**
- **Fault management is a major driver of complexity in software**
- **NASA's TacSat 3 VSM project applies Integrated Systems Health Management (ISHM) and Vehicle Systems Management (VSM) technologies to an experimental Air Force satellite**
- **NASA will test ISHM and VSM onboard TacSat 3 over an extended period**

What is ISHM?

- **ISHM: Integrated System Health Management**

- Capabilities far beyond “Fault Protection”
- Active control methods to improve safety, reliability, mission capability, sustainability, and ultimate cost
- Required in an era of increasing system complexity, e.g. Exploration



Goals of the Vehicle Systems Management Experiment

- **Conduct maturation and in-space testing of Vehicle Systems Management technologies**
 - Full-scale validation of model-based, autonomous, and ISHM software
 - Deploy and operate in space environment after launch (TRL 7)
 - Includes closed-loop experiments (full control) after end of primary mission
- **Gain integration and flight experience with TEAMS and SCL**
 - Fault detection and automation technologies TEAMS and SCL baselined for Orion spacecraft
 - Flight experiments crucial for risk-reduction of software technologies
- **Demonstrate new NASA technologies, on-ground and on-board**
 - High-level spacecraft planning
 - ISHM detection, diagnosis, reasoning technologies
 - Advanced V&V approaches to complex software, including TEAMS and SCL

Validating ISHM Components

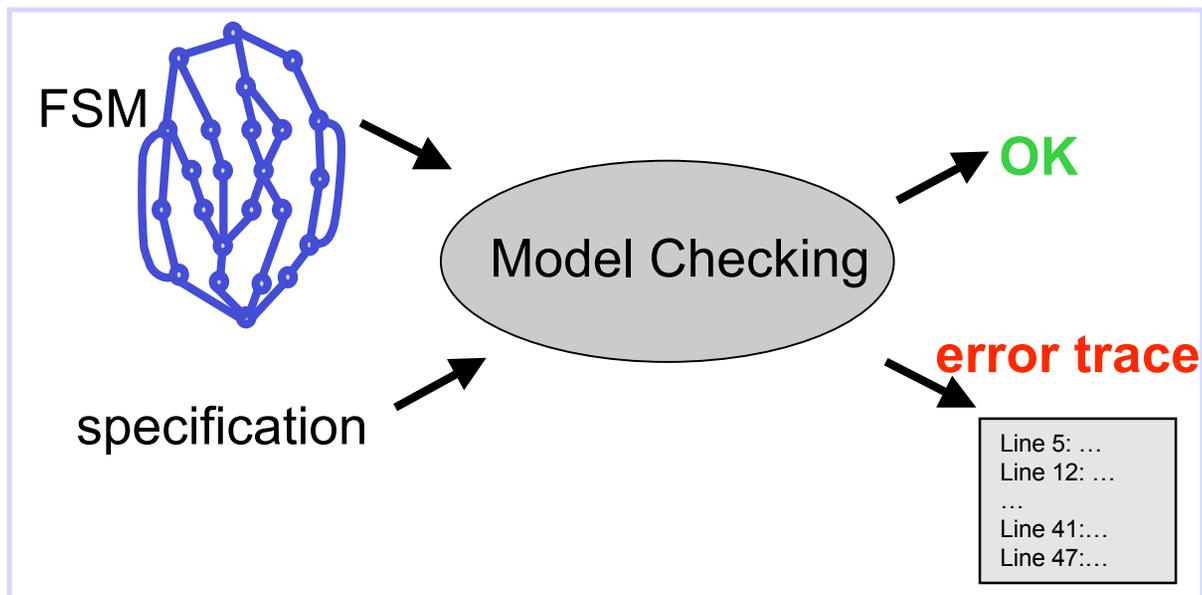
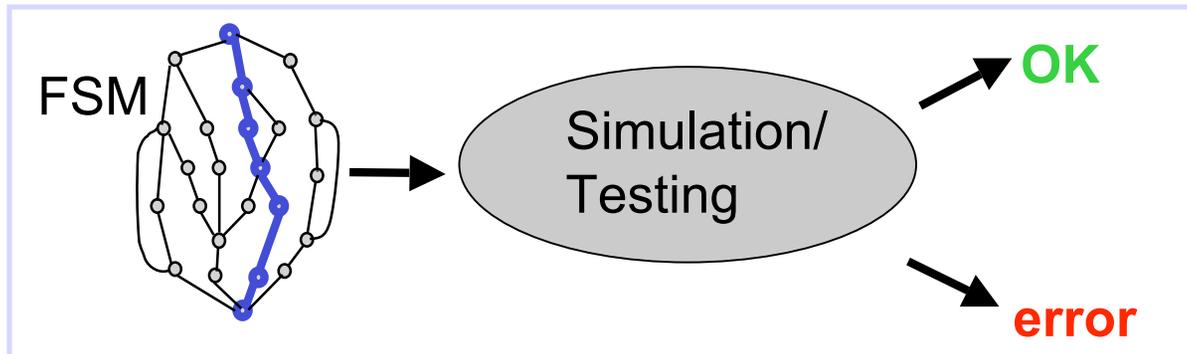
- **ISHM by definition deals with *off-nominal* conditions**
- **Have all the significant failure modes been identified?**
- **Is a given failure mode well understood?**

- **The number of combinations of failures is overwhelming**
- **Manual test generation is expensive and time consuming**

Automatic Test Generation

- **Manual test generation for software is time consuming and error-prone**
 - Lots of tests needed for full code coverage
 - Hard to create tests that cover specific code paths
- **Manually testing concurrent code is especially difficult**
- **Model Checking offers a way out**
 - Automatically generates all paths through the code
 - When combined with Symbolic Execution, it can create a test case for each code path
 - For concurrent code, exercises not only each code path, but also each thread scheduler decision

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)

- **Developed by RSE group at NASA Ames**
- **Explicit state model checker for Java bytecode**
 - Version targeting C/C++ is under development
- **Focus is on finding bugs**
 - Concurrency related: deadlocks, (races), missed signals etc.
 - Java runtime related: unhandled exceptions, heap usage, (cycle budgets)
 - Complex application specific assertions
- **Recipient of NASA “Turning Goals into Reality” Award, 2007.**
- **Open sourced since 04/2005 under NOSA 1.3 license:**
<javapathfinder.sourceforge.net>
 - First NASA system development hosted on public site
 - ~14000 downloads since publication
 - ~25000 read transactions in 2007

Symbolic Execution

- **JPF– SE:**

- Recent extension to JPF that enables automated test case generation
- Symbolic execution with model checking and constraint solving
- Applies to (executable) **models** and to code
- Generates an optimized test suite that exercise **all the behavior** of the system under test
- Reports coverage
- Checks for errors during test generation process

Symbolic Execution

Generating and Solving Constraints

```
[pres = 460; pres_min = 640; pres_max = 960]
```

```
if( (pres < pres_min) || (pres > pres_max)) {  
    ...  
} else {  
    ...  
}
```



```
[pres = Sym1; pres_min = MIN; pres_max = MAX] [path condition PC: TRUE]
```

```
if ((pres < pres_min) ||  
    (pres > pres_max)) {
```

```
[PC1: Sym1 < MIN]
```

```
} else {
```

```
...
```

```
}
```

```
if ((pres < pres_min) ||  
    (pres > pres_max)) {
```

```
[PC2: Sym1 > MAX]
```

```
} else {
```

```
...
```

```
}
```

```
if ((pres < pres_min) ||  
    (pres > pres_max)) {
```

```
...
```

```
} else {
```

```
[PC3: Sym1 >= MIN &&  
Sym1 <= MAX]
```

Solve path conditions PC₁, PC₂, PC₃ → test inputs

Previous Applications

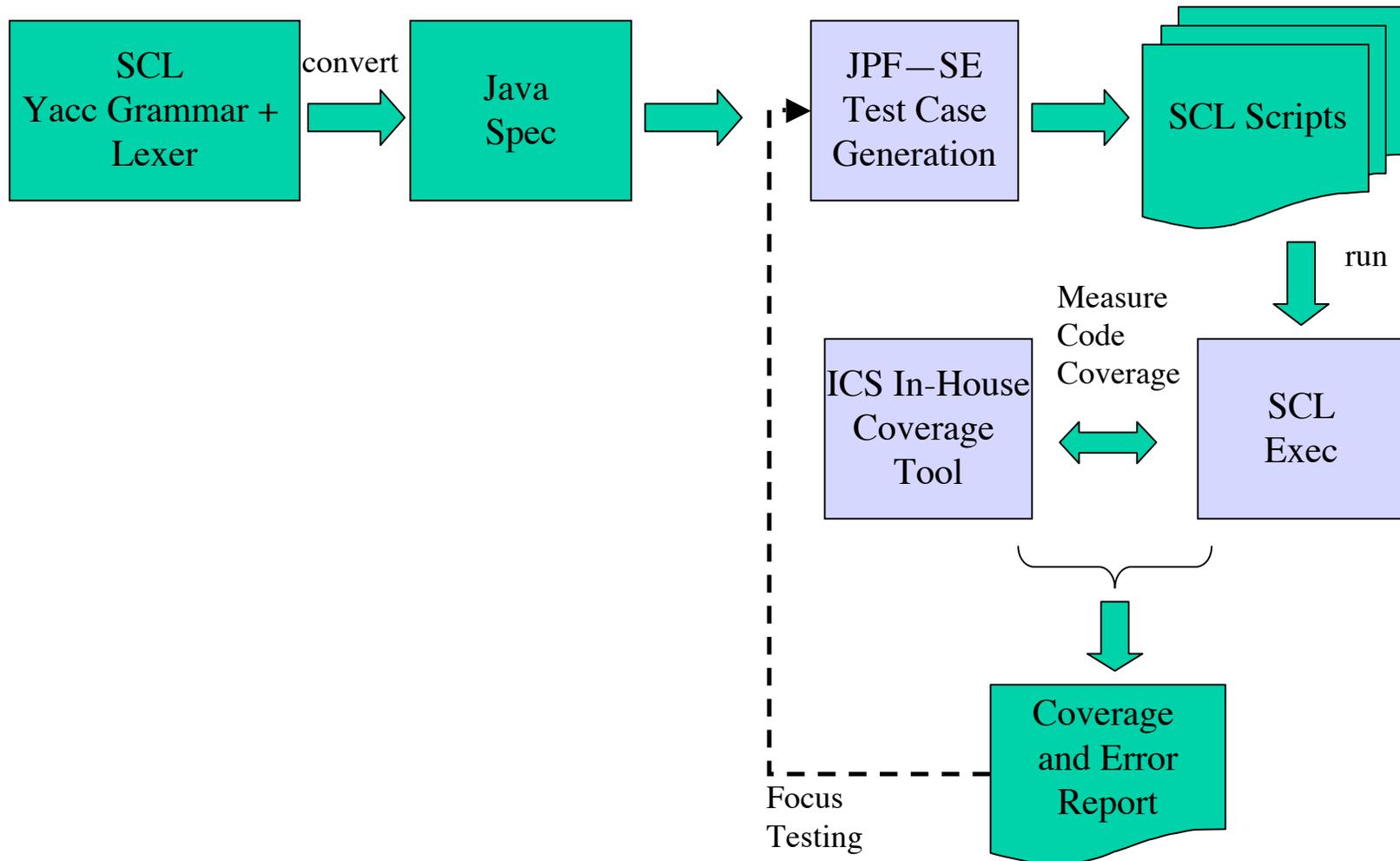
- **Onboard abort executive**
 - Prototype for CEV ascent abort handling being developed by JSC GN&C
 - Manual testing: time consuming (~1 week)
 - Guided random testing could not cover all aborts
 - JPF-SE
 - Generated 151 tests to cover all aborts and flight rules
 - Total execution time is < 1 min
 - Found major bug in new version of OAE
- **K9 Rover Executive**
 - Executive developed at NASA Ames
 - Automated plan generation based on CRL grammar
 - Generated hundreds of plans to test Exec engine

Applications to the TacSat Project

- **Test Case Generation for SCL**

- SCL from Interface and Control Systems, Inc. is a rule- and script-based runtime executive for aerospace applications
- Use JPF–SE to generate SCL scripts based on SCL Yacc grammar
- Run SCL exec engine on these scripts and measure coverage
- Focus SCL script generation on particular features of the language/engine

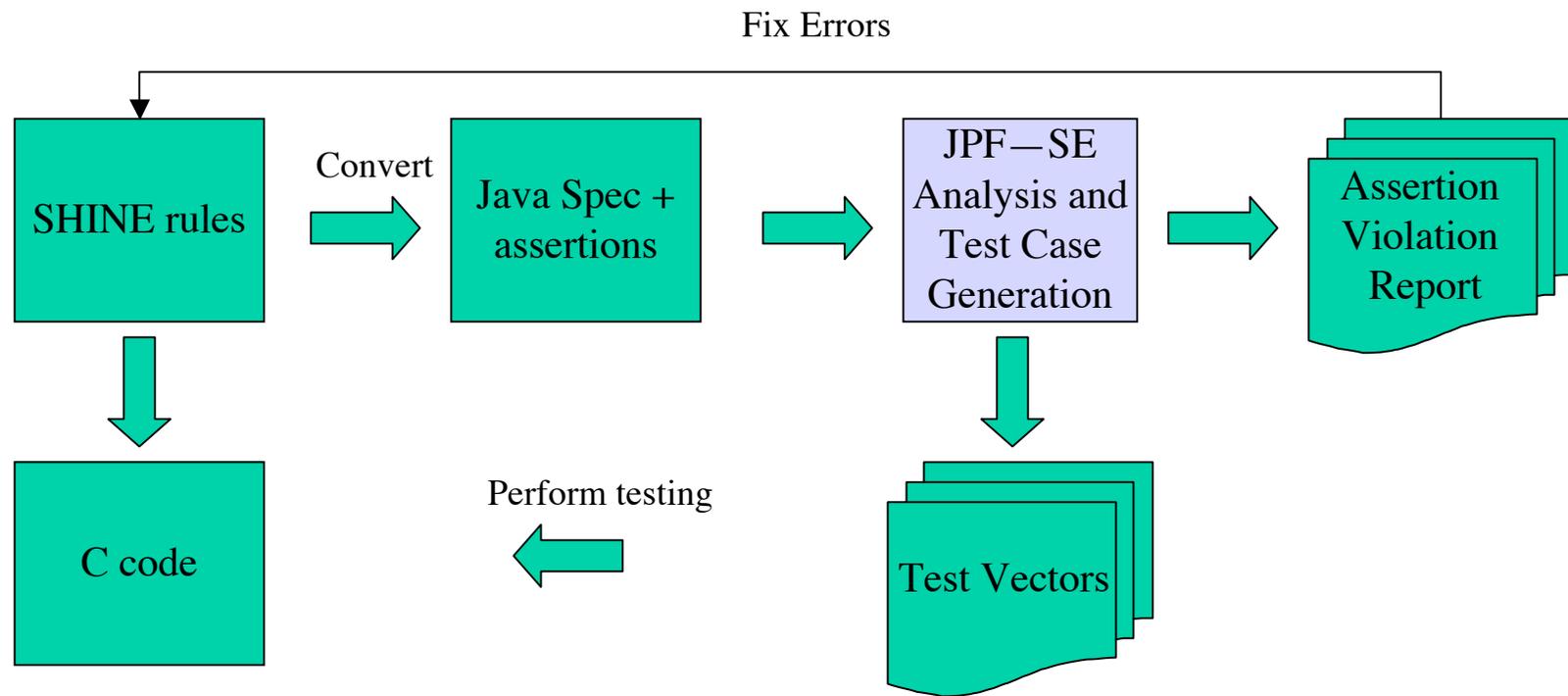
Test Case Generation for SCL



Applications to the TacSat Project

- **Test case generation for SHINE models**
 - SHINE from JPL is a very high-performance rule engine for embedded systems
 - Shine can generate C and Java code from its rule bases
 - We will apply JPF-SE to the SHINE java code to generate testcases for all paths through the rules
 - We will also apply JPF to the SHINE Java code to verify that critical safety properties hold under all possible executions

Test Case Generation for SHINE models



Early Results

- **SCL Results**

- We have part of the SCL Yacc grammar translated to Java and have generated test scripts covering that part of the grammar
- After coverage feedback from ICS, we will extend the translation and focus it on uncovered paths

- **SHINE Results**

- The SHINE-to-Java translator working and we are beginning to generate test cases for simple rule sets
- We have tested sample safety properties with our rule sets and generated both test cases and code traces showing the paths to the violations

Handout

Automatic Testcase Generation for Flight Software

- **Contacts:**

- David Bushnell: david.h.bushnell@nasa.gov
- Corina Pasareanu: corina.s.pasareanu@nasa.gov
- Ryan Mackey: ryan.m.mackey@nasa.gov

Sample SCL Grammar and Java Code

SCL Grammar

```
complex_expression  
: function  
| expression NE expression  
| expression LT expression  
...etc...  
;
```

Java Code

```
public String complex_expression() {  
    int selector = Verify.random(1);  
    switch (selector) {  
        case 0:  
            return expression() + "!=" + expression();  
        case 1:  
            return expression() + "<" + expression();  
        ...etc...  
        default:  
            throw new GrammarException();  
    }  
}
```

Java Pathfinder Output for SCL Grammar

... etc...

```
-- Script Set:  
SCRIPT TestScript070  
  MESSAGE "a message"  
  battvolts = battvolts  
  MESSAGE "a message"  
  battvolts = 1  
END TestScript070
```

... etc...

```
-- Script Set:  
SCRIPT TestScript17007  
  if battvolts then  
    EXECUTE TestScript07006 , PRIORITY = 4  
  end if  
END TestScript17007
```

```
SCRIPT TestScript07006  
  MESSAGE "a message"  
  battvolts = battvolts + battvolts  
END TestScript07006
```

Sample SHINE Rules and Java Code

SHINE Rules

```
(Def_Rule JR01
:Order 1
:If (= J01 1)
:Then (:Set J02 2))
```

```
(Def_Rule JR02
:Order 1
:If (= J02 2)
:Then (:Set J03 3))
```

```
(Def_Rule JR03
:Order 1
:If (= J03 3)
:Then (printf "Done\n"))
```

Java Code

```
private void sr_JR01() {
    sa_J02 = 2;
    ep_RFC_Flag = true;
    ep_DS_RFC_S[ep_DS_RFC_PSO].rdfv_JR02_Flag
        = true;
}

private void sr_JR02() {
    sa_J03 = 3;
    ep_RFC_Flag = true;
    ep_DS_RFC_S[ep_DS_RFC_PSO].rdfv_JR03_Flag
        = true;
}

private void sr_JR03() {
    System.out.printf("Done\n");
}
```

Java Pathfinder Output for SHINE Rules

Symbolic Execution Mode

JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center

Execute symbolic INVOKESPECIAL: doTEST(III)V (J01_1_SYMINT, J02_2_SYMINT, J03_3_SYMINT)

Done

Done

Done

doTEST: # = 3

J03_3_SYMINT[3] == CONST_3 && J02_2_SYMINT[2] == CONST_2 && J01_1_SYMINT[1] == CONST_1

Done

Done

doTEST: # = 3

J03_3_SYMINT[-10000] != CONST_3 && J02_2_SYMINT[2] == CONST_2 && J01_1_SYMINT[1] == CONST_1

...etc...

doTEST: # = 3

J03_3_SYMINT[3] == CONST_3 && J02_2_SYMINT[-10000] != CONST_2

&& J01_1_SYMINT[-10000] != CONST_1

doTEST: # = 3

J03_3_SYMINT[-10000] != CONST_3 && J02_2_SYMINT[-10000] != CONST_2

&& J01_1_SYMINT[-10000] != CONST_1

References

- **Java Pathfinder source code and documentation:**
<http://javapathfinder.sourceforge.net/>
- **Java Pathfinder and Symbolic Execution:**
[JPF--SE: A Symbolic Execution Extension to Java PathFinder](http://ti.arc.nasa.gov/people/pcorina/papers/jpfseTACAS07.pdf)
<http://ti.arc.nasa.gov/people/pcorina/papers/jpfseTACAS07.pdf>