Symbolic Execution of Java Byte-code

Corina Păsăreanu
Perot Systems/NASA Ames Research
ISSTA’08 paper:

“Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software”

Corina Păsăreanu, Peter Mehlitz, David Bushnell, Karen Gundy-Burlet, Michael Lowry (NASA Ames)
Suzette Person (University of Nebraska, Lincoln)
Mark Pape (NASA JSC)
Automatic Test Input Generation

• Objective:
  – Develop automated techniques for error detection in complex, flight control software for manned space missions

• Solutions:
  – Model checking – automatic, exhaustive; suffers from scalability issues
  – Static analysis – automatic, scalable, exhaustive; reported errors may be spurious
  – Testing – reported errors are real; may miss errors; widely used

• Our solution: Symbolic Java PathFinder (Symbolic JPF)
  – Symbolic execution with model checking and constraint solving for automatic test input generation
  – Generates test suites that obtain high coverage for flexible (user-definable) coverage metrics
  – During test generation process, checks for errors
  – Uses the analysis engine of the Ames JPF tool
  – Freely available at: http://javapathfinder.sourceforge.net (symbc extension)
Symbolic JPF

- Implements a non-standard interpreter of byte-codes
  - To enable JPF to perform symbolic analysis
- Symbolic information:
  - Stored in attributes associated with the program data
  - Propagated dynamically during symbolic execution
- Handles:
  - Mixed integer/real constraints
  - Complex Math functions
  - Pre-conditions, multithreading
- Allows for mixed concrete and symbolic execution
  - Start symbolic execution at any point in the program and at any time during execution
  - Dynamic modification of execution semantics
  - Changing mid-stream from concrete to symbolic execution
- Application:
  - Testing a prototype NASA flight software component
  - Found serious bug that resulted in design changes to the software
Background: 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
Background: 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)
- Open sourced:
  - <javapathfinder.sourceforge.net>
  - ~14000 downloads since publication
- Largest application:
  - Fujitsu (one million lines of code)
Background: Symbolic Execution

- King [Comm. ACM 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
```java
int x, y;
if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert false;
}
```
Example – Symbolic Execution

Code that swaps 2 integers:

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

Symbolic Execution Tree:

- **Path Condition**

  - **Path Condition:** \( x \geq y \) → **END**
  - **Path Condition:** \( x > y \) → \( y = x + y - y = x \)
  - **Path Condition:** \( x > y \) → \( x = x + y - x = y \)

- **Path Condition:** \( y > x \) → **END**

- **Path Condition:** \( x > y \) ∧ \( y \leq x \) → **END**

- **Path Condition:** \( x > y \) ∧ \( y > x \) → **END**

False!

*Solve path conditions → test inputs*
Symbolic JPF

- **JPF search engine used**
  - To generate and explore the symbolic execution tree
  - Also used to analyze thread inter-leavings and other forms of non-determinism that might be present in the code
  - No state matching performed
    - In general, un-decidable
  - To limit the (possibly) infinite symbolic search state space resulting from loops, we put a limit on
    - The model checker’s search depth or
    - The number of constraints in the path condition

- **Off-the-shelf decision procedures/constraint solvers used to check path conditions**
  - Model checker backtracks if path condition becomes infeasible
  - Generic interface for multiple decision procedures
    - Choco (for linear/non-linear integer/real constraints, mixed constraints),
    - IASolver (for interval arithmetic)
      [http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html](http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html)
Implementation

- **Key mechanisms:**
  - JPF’s bytecode instruction factory
    - Replace or extend standard concrete execution semantics of byte-codes with non-standard symbolic execution
  - Attributes associated w/ program state
    - Stack operands, fields, local variables
    - Store symbolic information
    - Propagated as needed during symbolic execution

- **Other mechanisms:**
  - Choice generators:
    - For handling branching conditions during symbolic execution
  - Listeners:
    - For printing results of symbolic analysis (method summaries)
    - For enabling dynamic change of execution semantics (from concrete to symbolic)
  - Native peers:
    - For modeling native libraries, e.g. capture `Math` library calls and send them to the constraint solver
An Instruction Factory for Symbolic Execution of Byte-codes

We created \textbf{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

\textbf{JPF core:}

- Implements concrete execution semantics based on stack machine model
- For each method that is executed, maintains a set of \textbf{Instruction} objects created from the method byte-codes
- Uses abstract factory design pattern to instantiate \textbf{Instruction} objects
Attributes for Storing Symbolic Information

- 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 concolic execution

Program state:
- A call stack/thread:
  - Stack frames/executed methods
  - Stack frame: locals & operands
- The heap (values of fields)
- Scheduling information
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 or IASolver)
  – If un-satisfiable, instruct JPF to backtrack

• Created new choice generator

```java
public class PCChoiceGenerator
    extends IntIntervalGenerator {
    PathCondition[] PC;
    ...
}
```
Example: IADD

Concrete execution of IADD byte-code:

```java
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:

```java
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:

```java
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:

```java
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;
        }
    }
}
```
How to Execute a Method Symbolically

JPF run configuration:

+vm.insn_factory.class=gov.nasa.jpf.symbc.SymbolicInstructionFactory
+vm.peer_packages=gov.nasa.jpf.symbc:gov.nasa.jpf.jvm
+jpf.listener=gov.nasa.jpf.symbc.SymbolicListener
+symbolic.dp=iasolver
+symbolic.method=UnitUnderTest(sym#sym#con)

Main application class containing method under test

Instruct JPF to use symbolic byte-code set
Print PCs and method summaries
Use symbolic peer package for Math library
Use IASolver as a decision procedure
Method to be executed symbolically (3rd parameter left concrete)

Symbolic input globals (fields) and method pre-conditions can be specified via user annotations
“Any Time” Symbolic Execution

- Symbolic execution
  - Can start at any point in the program
  - Can use mixed symbolic and concrete inputs
  - No special test driver needed – sufficient to have an executable program that uses the method/code under test

- Any time symbolic execution
  - Use specialized listener to monitor concrete execution and trigger symbolic execution based on certain conditions

- Unit level analysis in realistic contexts
  - Use concrete system-level execution to set-up environment for unit-level symbolic analysis

- Applications:
  - Exercise deep system executions
  - Extend/modify existing tests: e.g. test sequence generation for Java containers
Case Study:
Onboard Abort Executive (OAE)

- Prototype for CEV ascent abort handling being developed by JSC GN&C
- Currently test generation is done by hand by JSC engineers
- JSC GN&C requires different kinds of requirement and code coverage for its test suite:
  - Abort coverage, flight rule coverage
  - Combinations of aborts and flight rules coverage
  - Branch coverage
  - Multiple/single failures
OAE Structure

- Inputs
  - Checks Flight Rules to see if an abort must occur
  - Select Feasible Aborts
  - Pick Highest Ranked Abort
Results for OAE

• Baseline
  – Manual testing: time consuming (~1 week)
  – Guided random testing could not cover all aborts

• Symbolic JPF
  – 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

• Flexibility
  – Initially generated “minimal” set of test cases violating multiple flight rules
  – OAE currently designed to handle single flight rule violations
  – Modified algorithms to generate such test cases
Generated Test Cases and Constraints

Test cases:

// Covers Rule: FR A_2_A_2_B_1: Low Pressure Oxidizer 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 && …
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

• Use simulation runs to get data correlations
  – As a result, we eliminated some test cases that were impossible due to physical laws, for example

• Simulation environment: ANTARES
  – Advanced NASA Technology ARchitecture for Exploration Studies
  – Used for spacecraft design assessment, performance analysis, requirements validation, Hardware in the loop and Human in the loop testing

• Integration
  – System level simulations with ANTARES with
  – Unit level symbolic analysis
Using **System** Simulations to Determine **Unit** Pre-Conditions

- System simulation with ANTARES:
  - Set-up input file
  - Specify log file with variables to be logged during the run
  - Monte Carlo simulations
    - File with designated input variables
    - Their probability distributions
    - No. of cases to run while sampling from probability distributions

- Correlation analysis:
  - Determine ranges for unit inputs
  - Treatment learner [Menzies & Hu, 2003]
  - Daikon invariant detector
Comparison with Our Previous Work

- **JPF–SE [TACAS’07]:**
  - [http://javapathfinder.sourceforge.net](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
  - Generalized symbolic execution/lazy initialization [TACAS’03, SPIN’04]
    - Handles input data structures, arrays
    - Plan to move it into Symbolic JPF this summer
  - Interfaced with multiple decision procedures (Omega, CVC3/CVCLite, STP, Yices) via generic interface
    - Created generic interface in Symbolic JPF
    - Plan to add multiple decision procedures soon
  - Plan to add functionality of JPF—SE to Symbolic JPF
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 subsumption 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
  - Integer symbolic inputs (used CVCLite)
- Bogor/Kiasan [ASE’06]
  - Similar to JPF—SE, uses “lazier” approach
- Concolic execution [Godefroid et al. PLDI’05, Sen et al. ESEC/FSE’05]
  - DART/CUTE/jCUTE…
  - Can 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]
Conclusion and Future Plans

- **Symbolic JPF**
  - Non-standard interpretation of byte-codes
  - Symbolic information propagated via attributes associated with program variables, operands, etc.
  - Available from <javapathfinder.sourceforge.net>, symbc extension

- **Any-time symbolic execution**

- **Integration with system level simulation**
  - Use system level Monte Carlo simulation to obtain ranges for inputs

- **Application to prototype flight component**
  - Found major bug

- **Current/Future work:**
  - Test input generation for UML Statecharts; for Simulink/Stateflow/Embedded Matlab
  - Apply to NASA software
  - Tighter integration with system level simulation
  - More decision procedures
  - Use symbolic execution for differential analysis
  - Compositional analysis
    - Use symbolic execution to compute procedure summaries
  - Parallel symbolic execution

- **JPF in Google summer of code**
  - Generalized symbolic execution
  - Generate/extend test sequences
Questions?