Symbolic Execution and Model Checking for Testing

Corina Păsăreanu and Willem Visser

Perot Systems/NASA Ames Research Center and SEVEN Networks

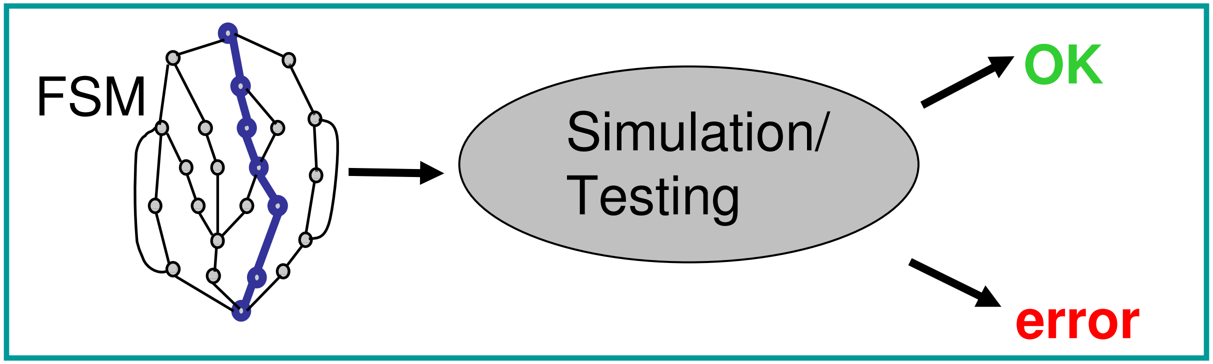
Thanks

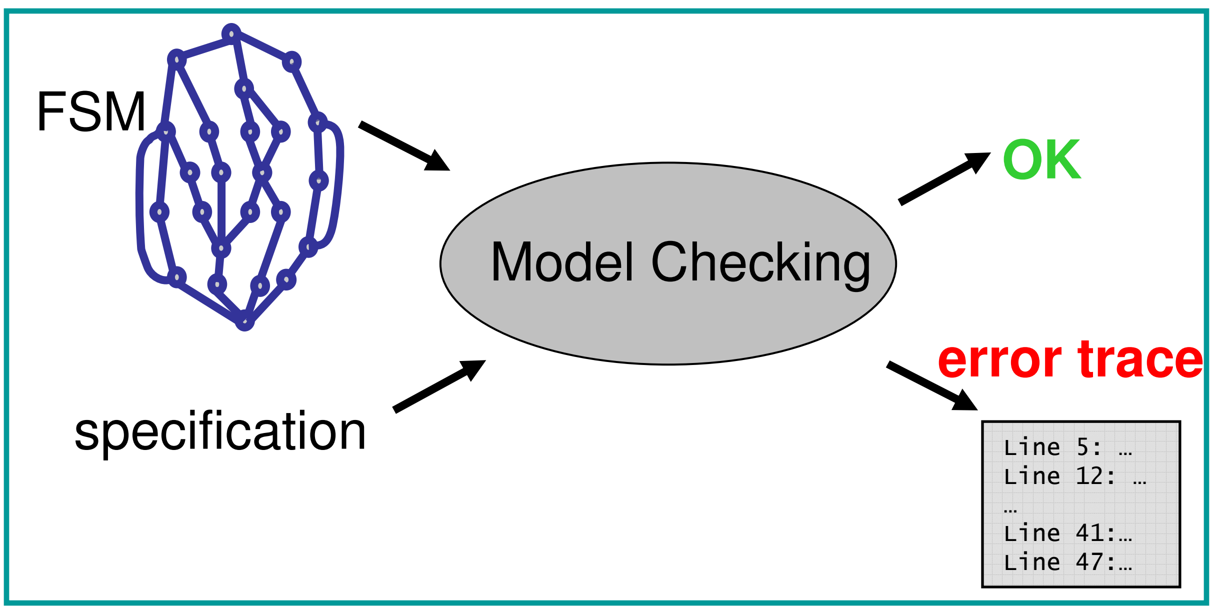
- Saswat Anand (Georgia Institute of Technology)
- Sarfraz Khurshid (University of Texas, Austin)
- Radek Pelánek (Masaryk University, Czech Republic)
- Suzette Person (University of Nebraska, Lincoln)
- Aaron Tomb (University of California, Santa Cruz)
- David Bushnell, Peter Mehlitz, Guillaume Brat (NASA Ames)

Introduction

- Goal:
  - Detect errors in complex software
  - Data structures, arrays, concurrency
- Solutions:
  - Software model checking with (predicate) abstraction
    - Automatic, exhaustive
    - Scalability issues – Explicit state model checking can not handle large, complex input domains
    - Reported errors may be spurious
  - Static analysis
    - Automatic, scalable, exhaustive
    - Reported errors may be spurious
  - Testing
    - Reported errors are real
    - May miss errors
    - Well accepted technique: state of practice for NASA projects
- Our approach:
  - Combine model checking and symbolic execution for test case generation

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

- JPF– SE [TACAS'03,'07]
  - Extension to JPF that enables automated test case generation
  - Symbolic execution with model checking and constraint solving
  - Applies to (executable) models and to code
  - Handles dynamic data structures, arrays, loops, recursion, multi-threading
  - Generates an optimized test suite that satisfy (customizable) coverage criteria
  - Reports coverage
  - During test generation process, checks for errors

RSE

Symbolic Execution

Systematic Path Exploration

Generation and Solving of Numeric Constraints

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

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

[pres = Sym_1; 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 PC_1, PC_2, PC_3 → test inputs

Applications

- NASA control software
  - Manual testing: time consuming (~1 week)
  - Guided random testing could not obtain full coverage
  - JPF-SE
    - Generated ~200 tests to obtain full coverage
    - Total execution time is < 1 min
    - Found major bug in new version
- K9 Rover Executive
  - Executive developed at NASA Ames
  - Automated plan generation based on CRL grammar + symbolic constraints
  - Generated hundreds of plans to test Exec engine
  - Combining Test Case Generation and Runtime Verification [journal TCS, 2005]
- Test input generation for Java classes:
  - Black box, white box [ISSTA'04, ISSTA'06]

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

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

path condition

[PC:true]x = X,y = Y

[PC:true] X > Y ?

false: [PC:X≤Y]END

true: [PC:X>Y]x= X+Y

[PC:X>Y]y = X+Y–Y = X

[PC:X>Y]x = X+Y–X = Y

[PC:X>Y]Y>X ?

false: [PC:X>Y∧Y≤X]END

true: [PC:X>Y∧Y>X]END

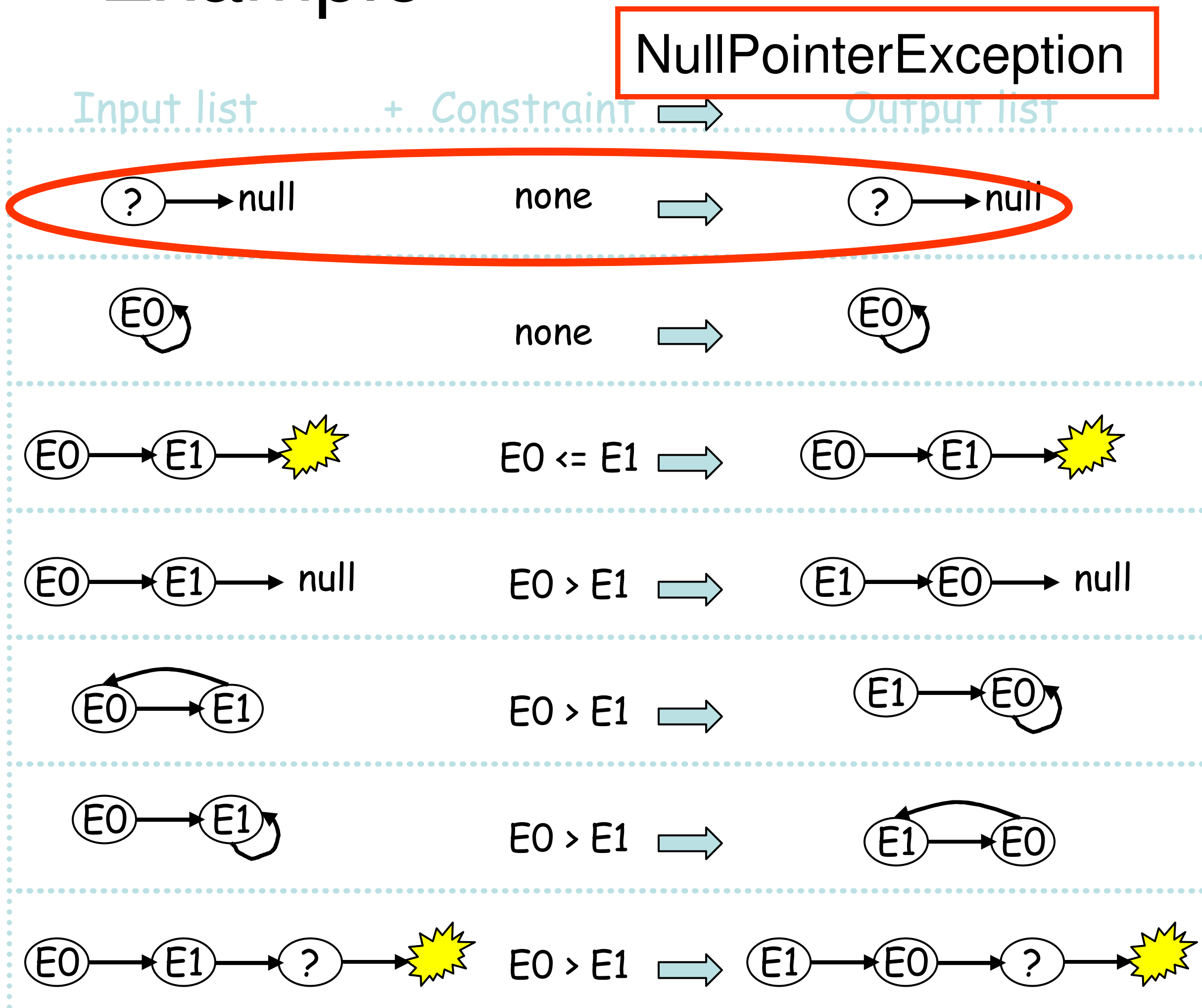
False!

Generalized Symbolic Execution

- JPF – SE handles
  - Dynamically allocated data structures
  - Arrays
  - Numeric constraints
  - Preconditions
  - Recursion, concurrency, etc.
- Lazy initialization for arrays and structures [TACAS'03, SPIN'05]
- Java PathFinder (JPF) used
  - To generate and explore the symbolic execution tree
  - Non-determinism handles aliasing
    - Explore different heap configurations explicitly
  - Off-the-shelf decision procedures check path conditions
    - Model checker backtracks if path condition becomes infeasible
- Subsumption checking and abstraction for symbolic states

Example

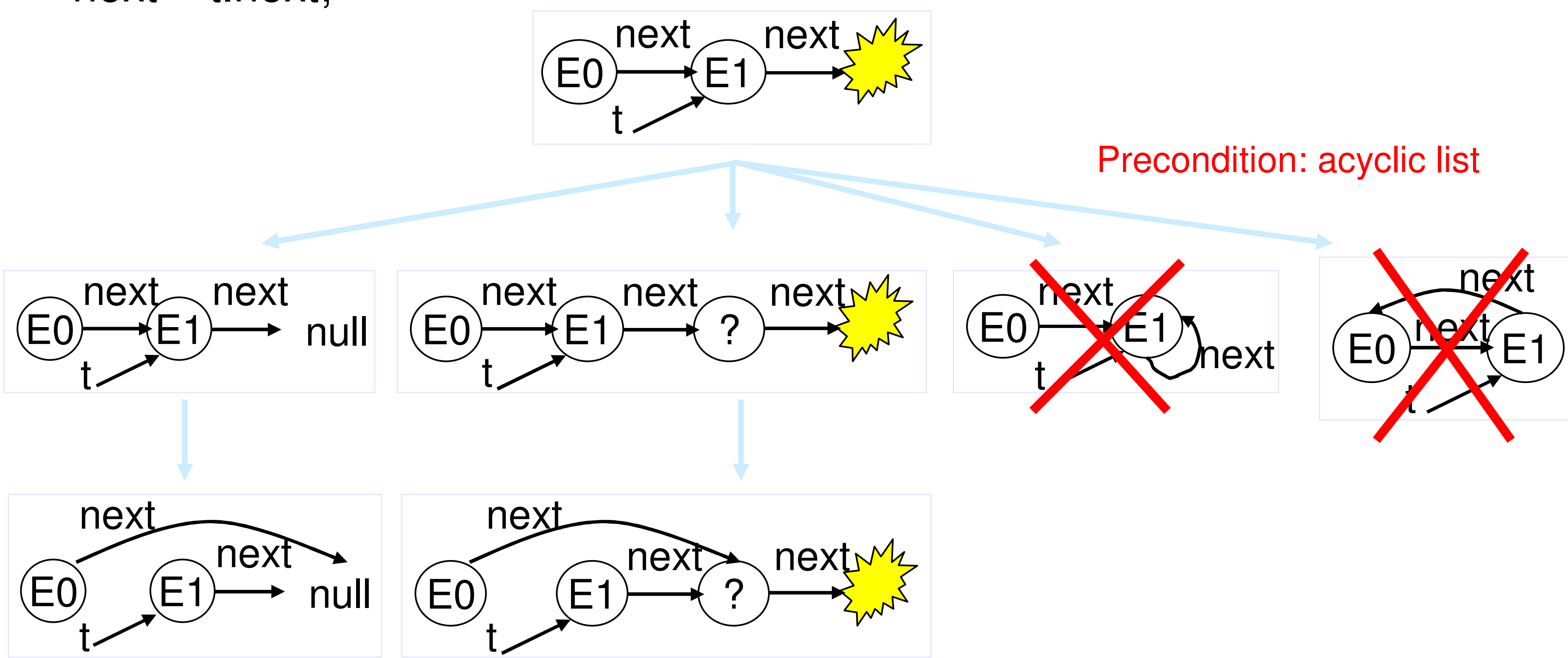
```
class Node {
    int elem;
    Node next;

    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;



Implementation

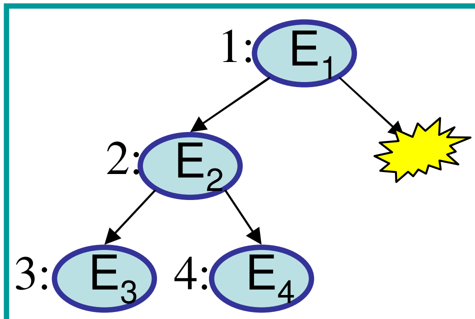
- Initial implementation
  - Done via instrumentation
  - Programs instrumented to enable JPF to perform symbolic execution
  - General: could use/leverage any model checker
- Decision procedures used to check satisfiability of path conditions
  - Omega library for integer linear constraints
  - CVCLite, STP (Stanford), Yices (SRI)

State Matching: Subsumption Checking

- Performing symbolic execution on looping programs
  - May result in an infinite execution tree
- Perform search with limited depth
- State matching – subsumption checking
  [SPIN'06, J. STTT to appear]
  - Obtained through DFS traversal of "rooted" heap configurations
    - Roots are program variables pointing to the heap
  - Unique labeling for "matched" nodes
  - Check logical implication between numeric constraints

State Matching: Subsumption Checking

Stored state:

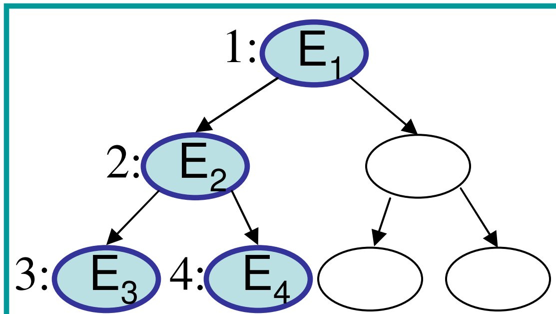


$E_1 > E_2 \land$
$E_2 > E_3 \land$
$E_2 \leq E_4 \land$
$E_1 > E_4$

Set of concrete states represented by stored state

$\supseteq$ $\Uparrow$ $\supseteq$

New state:



$E_1 > E_2 \land$
$E_2 > E_3 \land$
$E_2 < E_4 \land$
$E_1 > E_4$

Set of concrete states represented by new state

Normalized using existential quantifier elimination

Abstract Subsumption

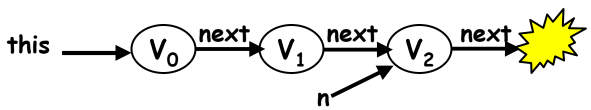
- Symbolic execution with subsumption checking
  - Not enough to ensure termination
  - An infinite number of symbolic states
- Our solution
  - Abstraction
    - Store abstract versions of explored symbolic states
    - Subsumption checking to determine if an abstract state is re-visited
    - Decide if the search should continue or backtrack
  - Enables analysis of under-approximation of program behavior
  - Preserves errors to safety properties/ useful for testing
- Automated support for two abstractions:
  - Shape abstraction for singly linked lists
  - Shape abstraction for arrays
  - Inspired by work on shape analysis (e.g. [TVLA])
- No refinement!

Abstractions for Lists and Arrays

- Shape abstraction for singly linked lists
  - Summarize contiguous list elements not pointed to by program variables into summary nodes
  - Valuation of a summary node
    - Union of valuations of summarized nodes
  - Subsumption checking between abstracted states
    - Same algorithm as subsumption checking for symbolic states
    - Treat summary node as an “ordinary” node
- Abstraction for arrays
  - Represent array as a singly linked list
  - Abstraction similar to shape abstraction for linked lists

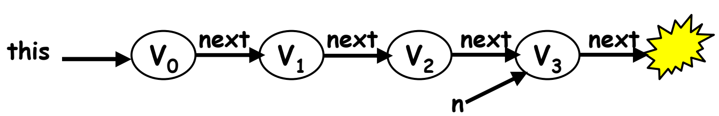
Abstraction for Lists

Symbolic states



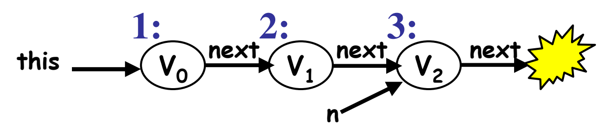
PC: $V_0 \leq v \wedge V_1 \leq v$

Unmatched!



PC: $V_0 \leq v \wedge V_1 \leq v \wedge V_2 \leq v$

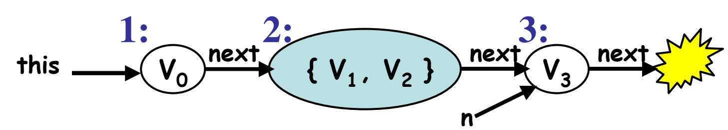
Abstracted states



$E_1 = V_0 \wedge E_2 = V_1 \wedge E_3 = V_2$

PC: $V_0 \leq v \wedge V_1 \leq v$

$\subseteq$



$E_1 = V_0 \wedge (E_2 = V_1 \vee E_2 = V_2) \wedge E_3 = V_3$

PC: $V_0 \leq v \wedge V_1 \leq v \wedge V_2 \leq v$

Applications of JPF-SE

- Test input generation for Java classes [ISSTA'04,'06]
  - Black box
    - Run symbolic execution on Java representation of class invariant
  - White box
    - Run symbolic execution on Java methods
    - Use class invariant as pre-condition
  - Test sequence generation
- Proving program correctness with generation of loop invariants [SPIN'04]
- Error detection in concurrent software
- Test input generation for NASA flight control software

Test Sequence Generation for Java Containers

- Containers – available with JPF distribution
  - Binary Tree
  - Fibonacci Heap
  - Binomial Heap
  - Tree Map
- Explore method call sequences
  - Match states between calls to avoid generation of redundant states
  - Abstract matching on the shape of the containers
- Test input – sequence of method calls

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

Testing Java Containers

- Comparison
  - Explicit State Model Checking (w/ Symmetry Reductions)
  - Symbolic Execution
  - Symbolic/Concrete Execution w/ Abstract Matching
  - Random Testing
- Testing coverage
  - Statement, Predicate
- Results
  - Symbolic execution worked better than explicit model checking
  - Model checking with shape abstraction
    - Good coverage with short sequences
    - Shape abstraction provides an accurate representation of containers
  - Random testing
    - Requires longer sequences to achieve good coverage

Test Input Generation for NASA Software

- Abort logic (~600 LOC)
  - Checks flight rules, if violated issues abort
  - Symbolic execution generated 200 test cases
    - Covered all flight rules/aborts in a few seconds, discovered errors
  - Random testing covered only a few flight rules (no aborts)
  - Manual test case generation took ~20 hours
- Integration of Automated Test Generation with End-to-end Simulation
  - JPF—SE: essentially applied at unit level
  - 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

Related Approaches

- Korat: black box test generation [Boyapati et al. ISSTA'02]
- Concolic execution [Godefroid et al. PLDI'05, Sen et al. ESEC/FSE'05]
  - DART/CUTE/jCUTE/…
- Concrete model checking with abstract matching and refinement [CAV'05]
- Symstra [Xie et al. TACAS'05]
- Execution Generated Test Cases [Cadar & Engler SPIN'05]
- Testing, abstraction, theorem proving: better together! [Yorsh et al. ISSTA'06]
- SYNERGY: a new algorithm for property checking [Gulavi et al. FSE'06]
- Feedback directed random testing [Pacheco et al. ICSE'07]
- …

Variably Inter-procedural Program Analysis for Runtime Error Detection

- [ISSTA'07] Willem Visser, Aaron Tomb, and Guillaume Brat
- Dedicated tool to perform symbolic execution for Java programs
  - Does not use JPF
  - Can customize
    - Procedure call depth
    - Max size of path condition
    - Max number of times a specific instruction can be revisited during the analysis
- Unsound and incomplete
  - Generated test cases are run in concrete execution mode to see if they correspond to real errors
  - "Symbolic execution drives the concrete execution"

Variably Inter-procedural Program Analysis for Runtime Error Detection

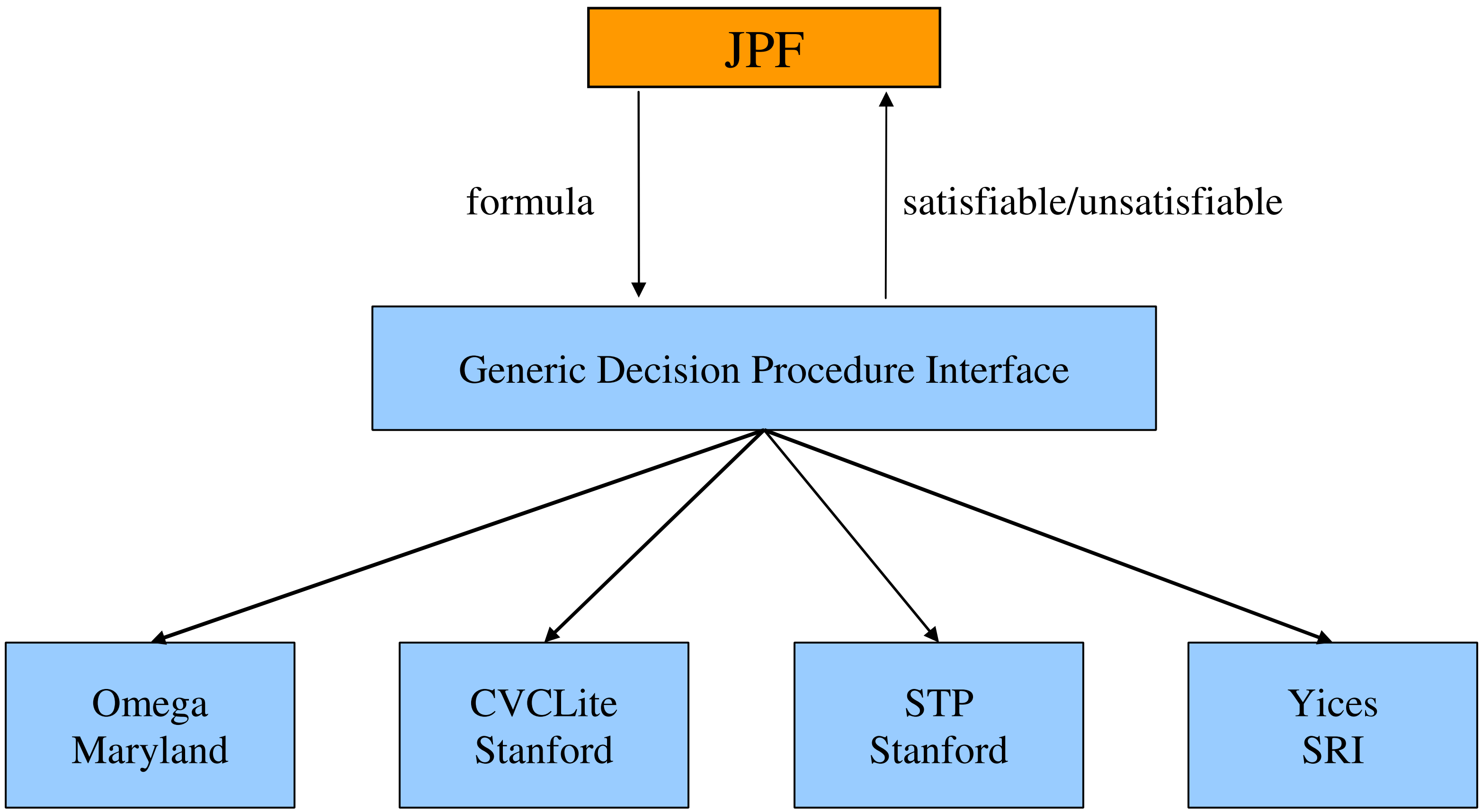
- Applied to 6 small programs and 5 larger programs (including JPF 38538 LOC, 382 Classes, 2458 Methods)
- Varied:
  - Inter-procedural depth: 0, 1 and 2
  - Path Condition size: 5, 10, 15, 20 and 25
  - Instruction revisits: 3, 5, and 10
- Results:
  - Found known bugs
  - Increasing the call depth does not necessarily expose errors, but decreases the number of false warnings
- Checking feasibility of path conditions
  - Takes a lot of time (up to 40% in some of the larger applications)
  - Greatly helps in pruning infeasible paths/eliminating false warnings
- More interesting results – see the paper

Current and Future Work

- New symbolic execution framework
  - Moved inside JPF
  - Non-standard interpretation of bytecodes
  - Symbolic information propagated via attributes associated with program variables, operands, etc.
  - Uses Choco (pure Java, from <sourceforge>) – for linear/non-linear integer/real constraints
  - Available from <javapathfinder.sourceforge.net>
- Start symbolic execution from any point in the program
- Compositional analysis
  - Use symbolic execution to compute procedure summaries
- Integration with system level simulation
  - Use system level Monte Carlo simulation to obtain ranges for inputs
- Test input generation for UML Statecharts
  - Recent JPF extension
- Use symbolic execution to aid regression testing
- Apply to NASA software …

Thank you!

JPF – SE


JPF
formula
satisfiable/unsatisfiable
Generic Decision Procedure Interface
Omega
Maryland
CVCLite
Stanford
STP
Stanford
Yices
SRI

Communication Methods

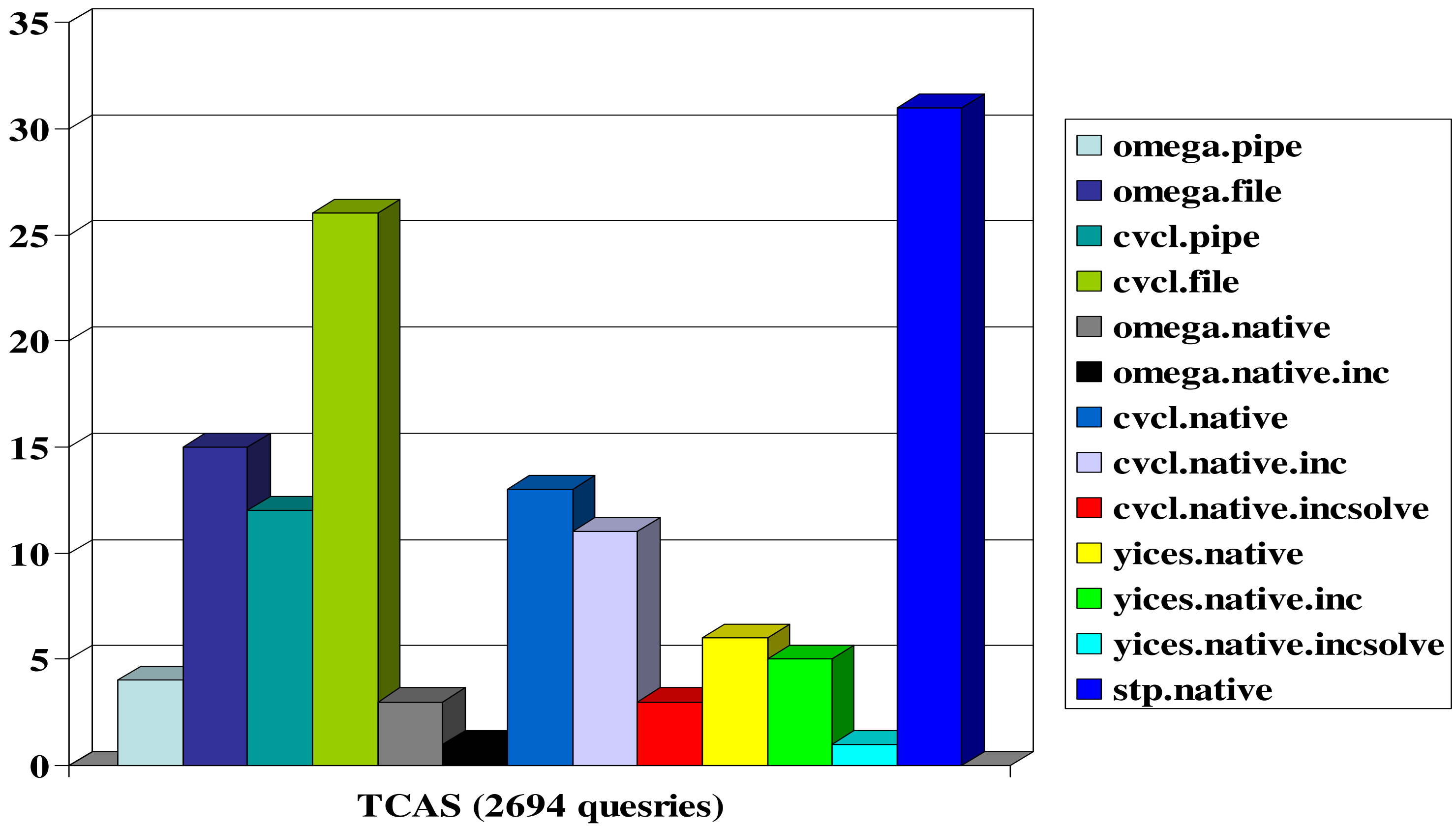
- JPF and the Interface code is in Java
  - Decision procedures are not in Java, mainly C/C++ code
- Various different ways of communication
  - Native: using JNI to call the code directly
  - Pipe: start a process and pipe the formulas and results back and forth
  - Files: same as Pipe but now use files as communication method
- Optimizations:
  - Some decision procedures support running in a incremental mode where you do not have to send the whole formula at a time but just what was added and/or removed.
  - CVCLite, Yices

RSE

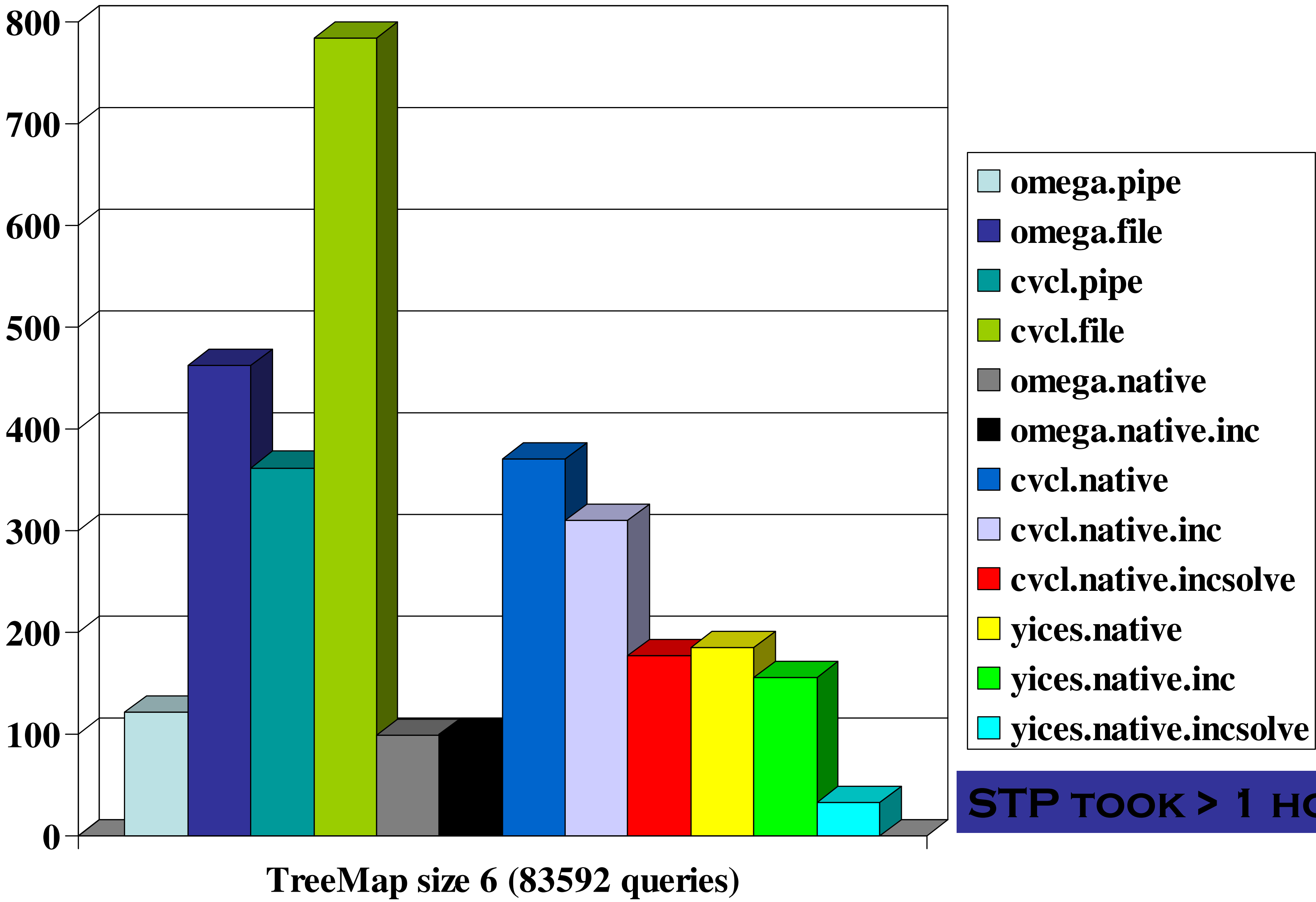
Decision Procedure Options

- +symbolic.dp=
  - omega.file
  - omega.pipe
  - omega.native
  - omega.native.inc
    - …inc - with table optimization
  - yices.native
  - yices.native.inc
  - yices.native.incsolve
    - …incsolve - Table optimization and incremental solving
  - cvcl.file
  - cvcl.pipe
  - cvcl.native
  - cvcl.native.inc
  - cvcl.native.incsolve
  - stp.native
- If using File or Pipe one must also set
  - Symbolic.<name>.exe to the executable binary for the DP
- For the rest one must set LD_LIBRARY_PATH to where the DP libraries are stored
  - Extensions/symbolic/CSRC
- Currently everything works under Linux and only CVCLite under Windows
  - Symbolic.cvclite.exe = cvclite.exe must be set with CVClite.exe in the Path

Results TCAS



Results TreeMap



STP TOOK > 1 HOUR