

---

# Symbolic Execution and Model Checking for Testing

Corina Păsăreanu & Willem Visser

*Perot Systems/NASA Ames Research Center & SEVEN Networks*

Thanks:

*Saswat Anand (Georgia Institute of Technology)*

*Sarfraz Khurshid (University of Texas, Austin)*

*Radek Pelánek (Masaryk University)*

# Introduction

---



- (... still needs work)
- Framework for symbolic execution (JPF-SE)–  
built around Java PathFinder
  - Abstract subsumption checking for state matching
  - Test input generation: container classes; NASA software

# JPF – SE

---



- Explicit state model checking can not handle large/complex input data domains
- JPF – SE [TACAS'03, TACAS'07]:
  - Framework built around Java PathFinder (JPF)  
<[javapathfinder.sourceforge.net](http://javapathfinder.sourceforge.net)>
  - Symbolic execution of Java code
  - Abstract subsumption checking for state matching
    - No automated refinement
    - User-provided abstractions
- Generate tests for systems that manipulate complex data structures
- Applied to container classes; NASA software

# 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 – Explicit 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



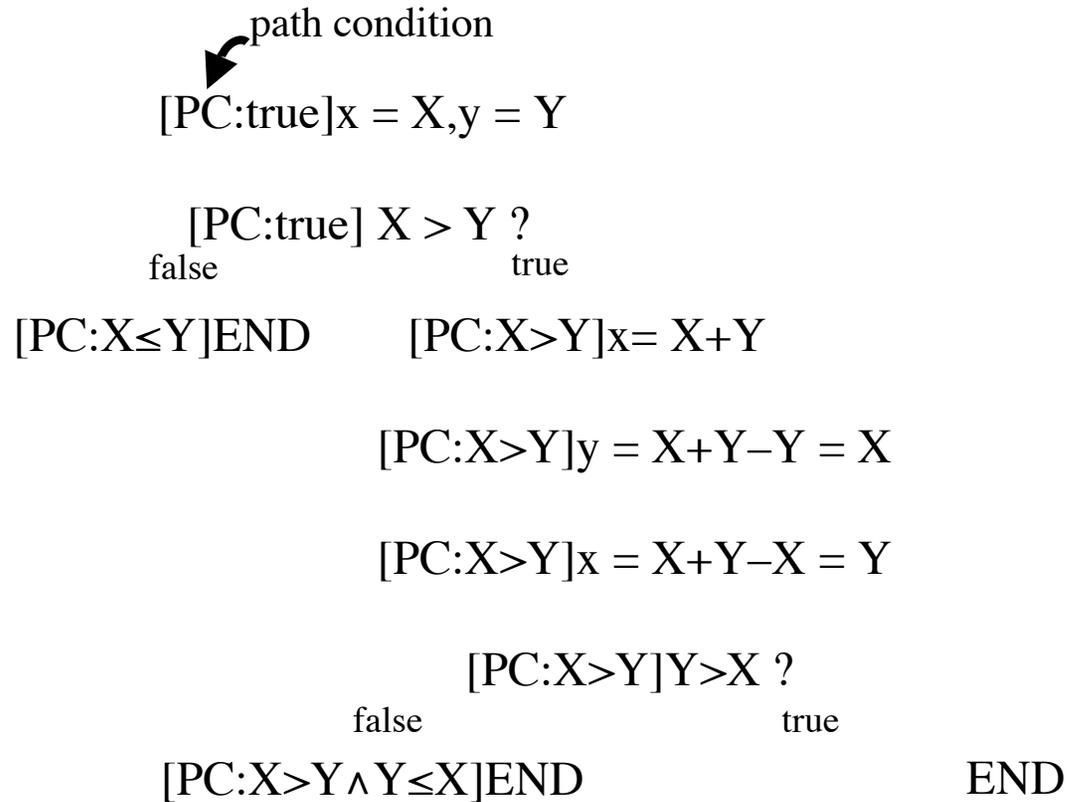
## Code that swaps 2 integers

```

int x, y;

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

## Symbolic Execution Tree



# Generalized Symbolic Execution

---



- JPF – SE handles
  - Dynamically allocated data structures, arrays, preconditions
  - Recursion, concurrency, etc.
- Lazy initialization for arrays and structures [TACAS'03]
- Java PathFinder (JPF) used
  - To generate and explore the symbolic execution tree
- Implementation via instrumentation
  - Programs instrumented to enable JPF to perform symbolic execution
  - Decision procedures used to check satisfiability of path conditions
- Subsumption checking for (abstract) symbolic states

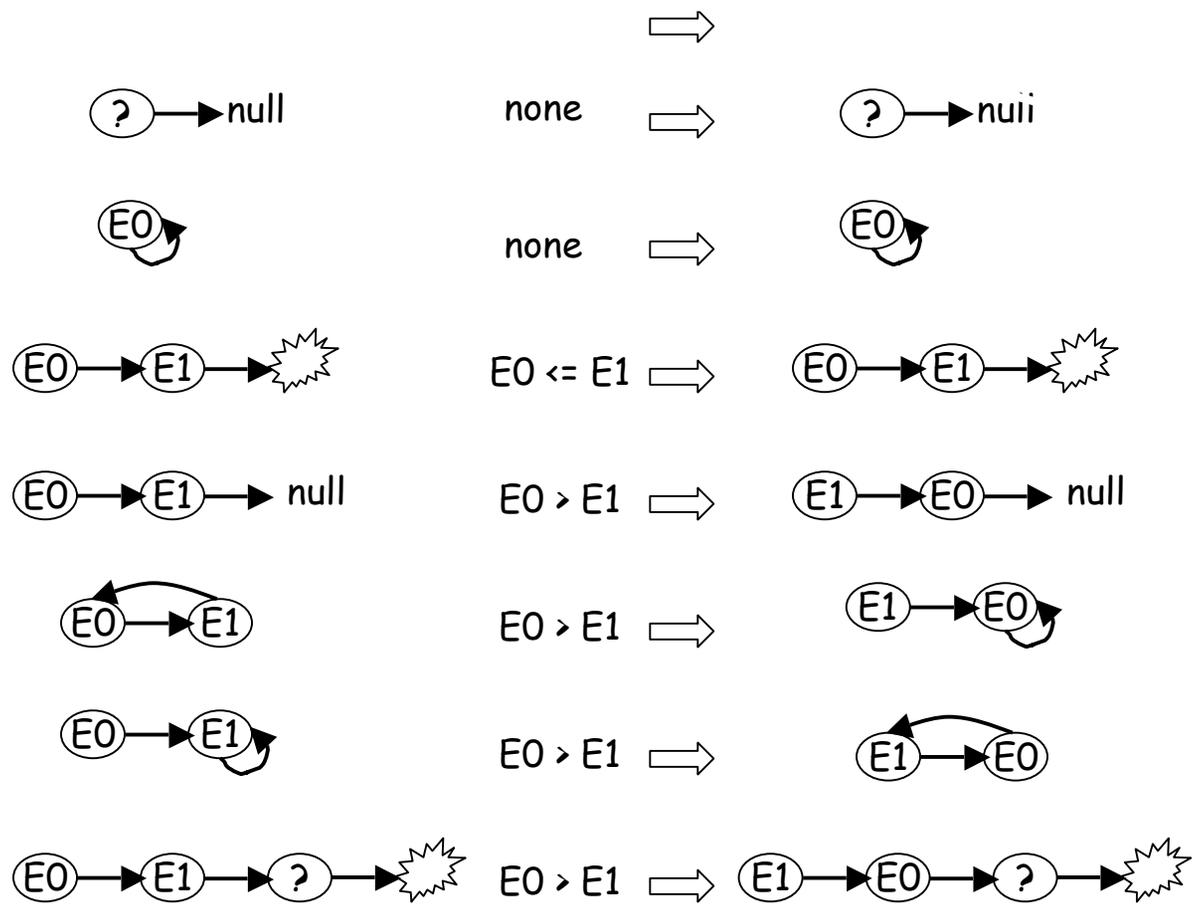
# Example



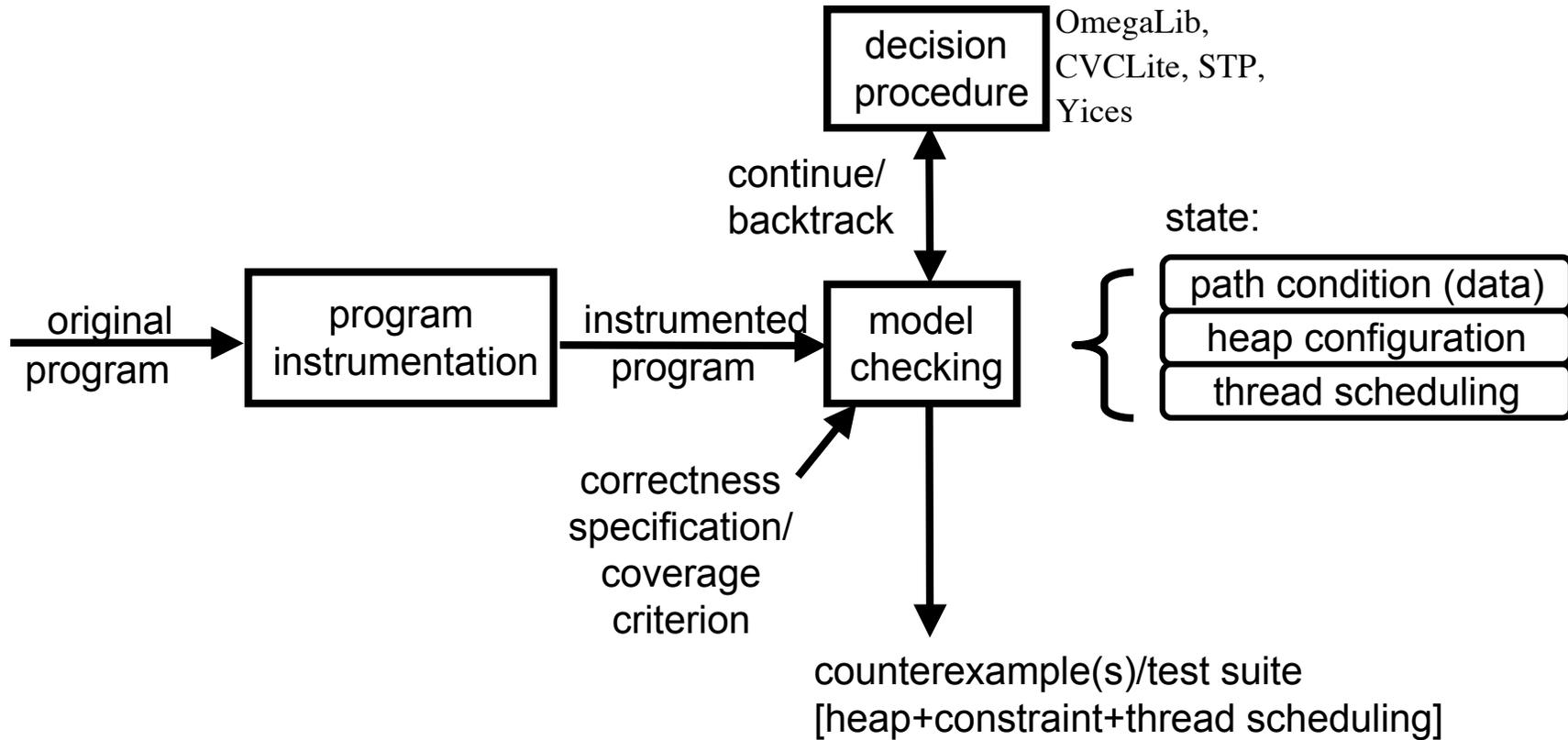
```
class Node {
  int elem;
  Node next;
```

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

## NullPointerException



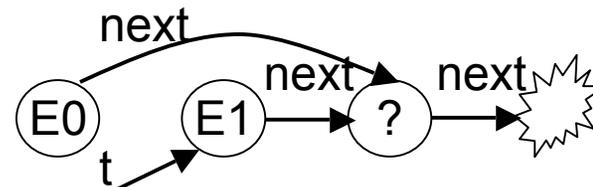
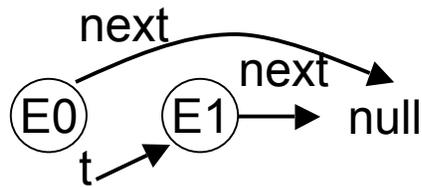
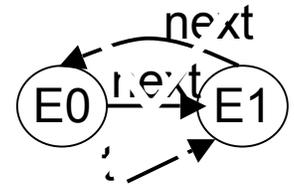
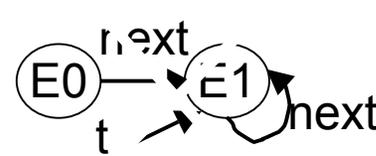
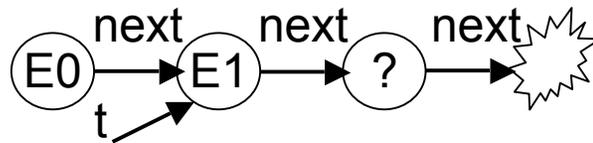
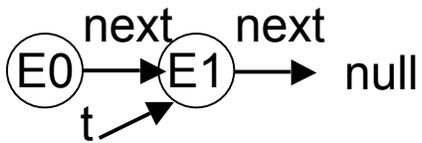
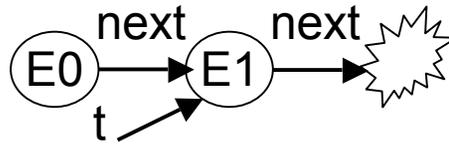
# Implementation via Instrumentation



# Lazy Initialization (illustration)



consider executing  
next = t.next;



# State Matching: Subsumption Checking

---

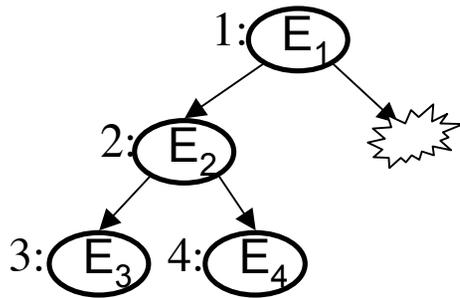


- Performing symbolic execution on looping programs
  - May result in an infinite execution tree
- Perform search with limited depth
- State matching – subsumption checking
  - [SPIN'06, J. STTT to appear]
  - Obtained through DFS traversal of “rooted” heap configurations
    - Roots are program variables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

# State Matching: Subsumption Checking



Stored state:



$$E_1 > E_2 \wedge$$

$$E_2 > E_3 \wedge$$

$$E_2 \leq E_4 \wedge$$

$$E_1 > E_4$$

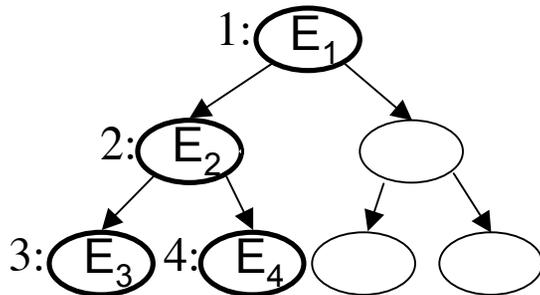
Set of concrete states represented by stored state

UI



UI

New state:



$$E_1 > E_2 \wedge$$

$$E_2 > E_3 \wedge$$

$$E_2 < E_4 \wedge$$

$$E_1 > E_4$$

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

---

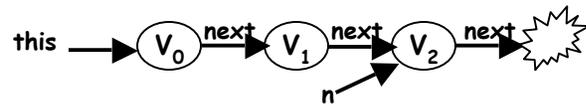


- Shape abstraction for singly linked lists
  - Summarize contiguous list elements not pointed to by program variables into summary nodes
  - Valuation of a summary node
    - Union of valuations of summarized nodes
  - Subsumption checking between abstracted states
    - Same algorithm as subsumption 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

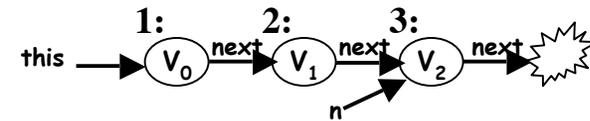


## Symbolic states



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

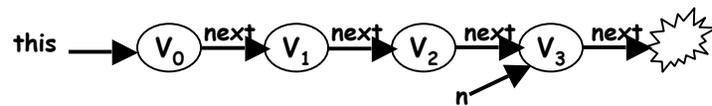
## Abstracted states



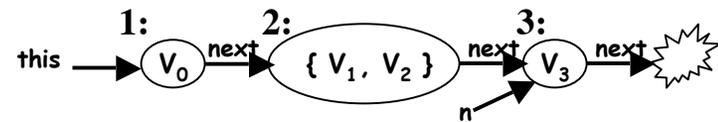
$E_1 = V_0 \wedge E_2 = V_1 \wedge E_3 = V_2$

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

U



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



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

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:
  - Black box, white box [ISSTA'04, ISSTA'06]
- Proving program correctness with generation of loop invariants [SPIN'04]
- Error detection in concurrent software
- Test input generation for NASA flight control software
- Other ...



# Testing Java Containers

---

- Containers
  - Binary Tree, Fibonacci Heap, Binomial Heap, Tree Map – available with JPF distribution
- Explore method call sequences
  - Match states between calls to avoid generation of redundant states
  - Abstract matching but no refinement
- Test input – sequence of method calls

```
BinTree t = new BinTree(); t.add(1); t.add(2); t.remove(1);
```
- Compared
  - Traditional Model Checking, Symmetry Reductions, Symbolic Execution
  - Symbolic/Concrete Execution using Abstract Matching on the shape of the containers,
  - 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

## Conclusion (II)

---



- Symbolic execution with subsumption checking
  - Explores only feasible program behavior
  - Handles heap structures and arrays
- Abstractions for lists and arrays
  - Explore an under-approximation of feasible behavior
  - Complementary to over-approximation based abstraction
- Future work:
  - Investigate other shape abstractions
  - Combine with predicate abstraction
  - Automatic abstraction refinement
  - Compositional analysis
  - Combine Monte Carlo simulations and symbolic execution for system level testing
- Future – hybrid approaches:
  - Concrete/symbolic analysis, over-/under- approximations
  - DART/CUTE, SYNERGY [FSE'06] ...

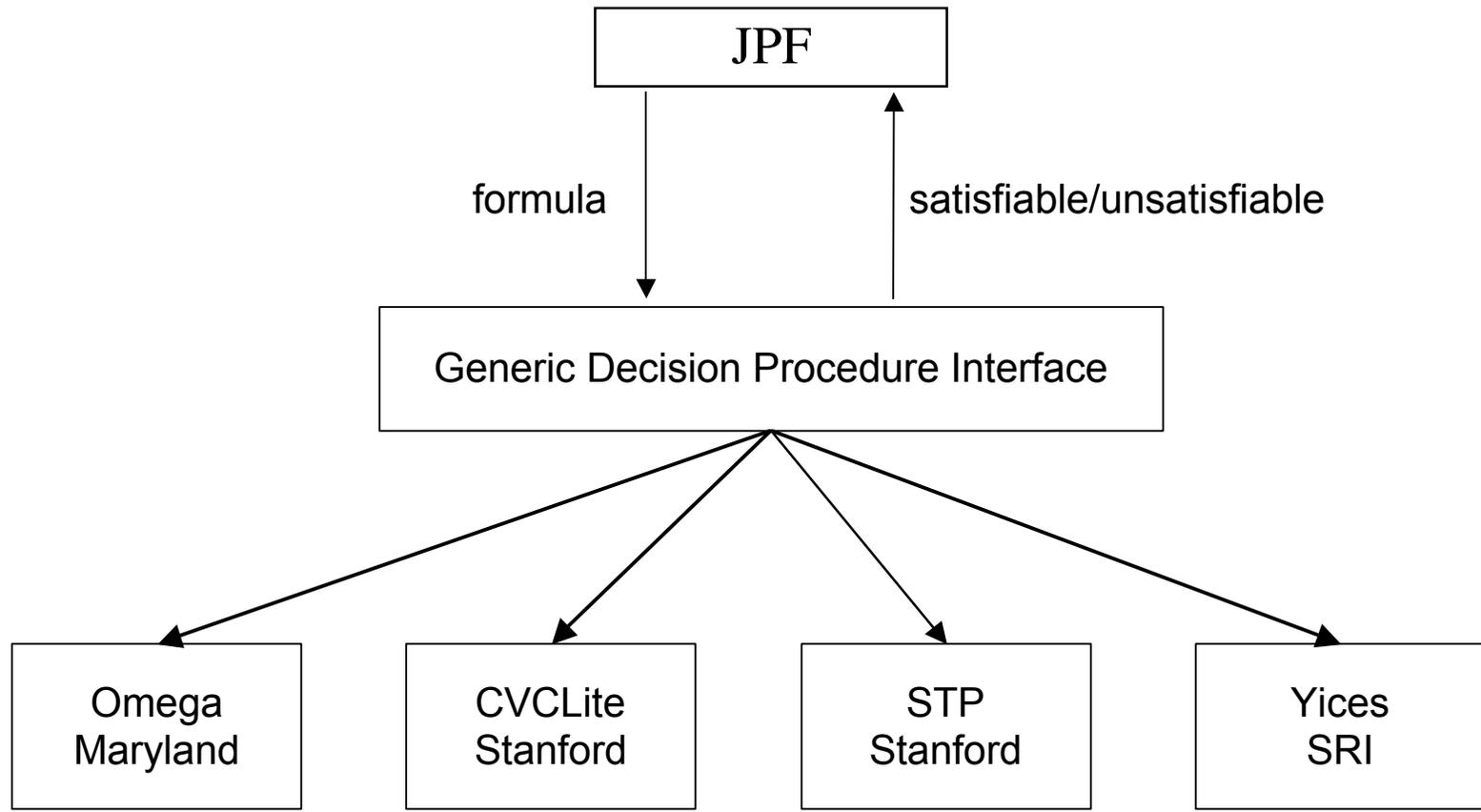
# More Information

---



- Predicate Abstraction with Underapproximation Refinement,  
*Corina S. Pasareanu, Radek Pelanek, Willem Visser,*  
in Logical Methods in Computer Science, Volume 3, Issue 1.
- JPF – SE: A Symbolic Execution Extension to Java PathFinder (tool description),  
*Saswat Anand, Corina S. Pasareanu, Willem Visser,*  
in Proceedings of TACAS'07.
- Symbolic Execution with Abstraction,  
*Saswat Anand, Corina S. Pasareanu, Willem Visser,*  
in STTT 2007 (to appear).
- <http://ase.arc.nasa.gov/people/pcorina/>

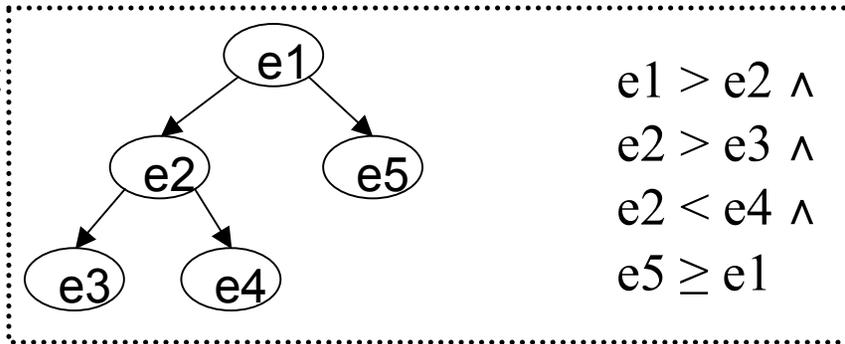
# JPF - SE



# State Matching: Subsumption Checking



Stored state:



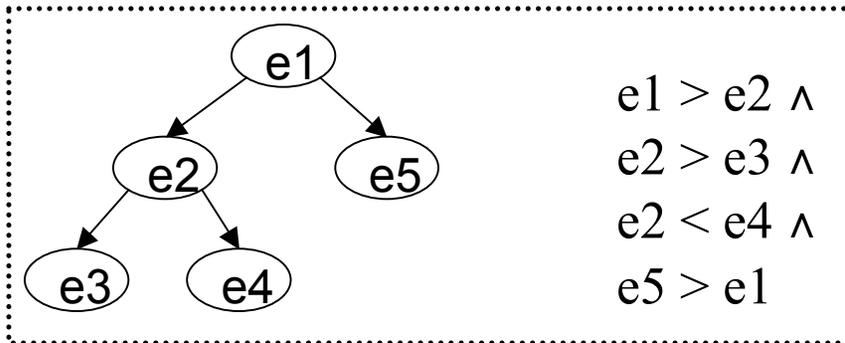
Set of concrete states represented by stored state

Same shape

↑ Matched

UI

New state:



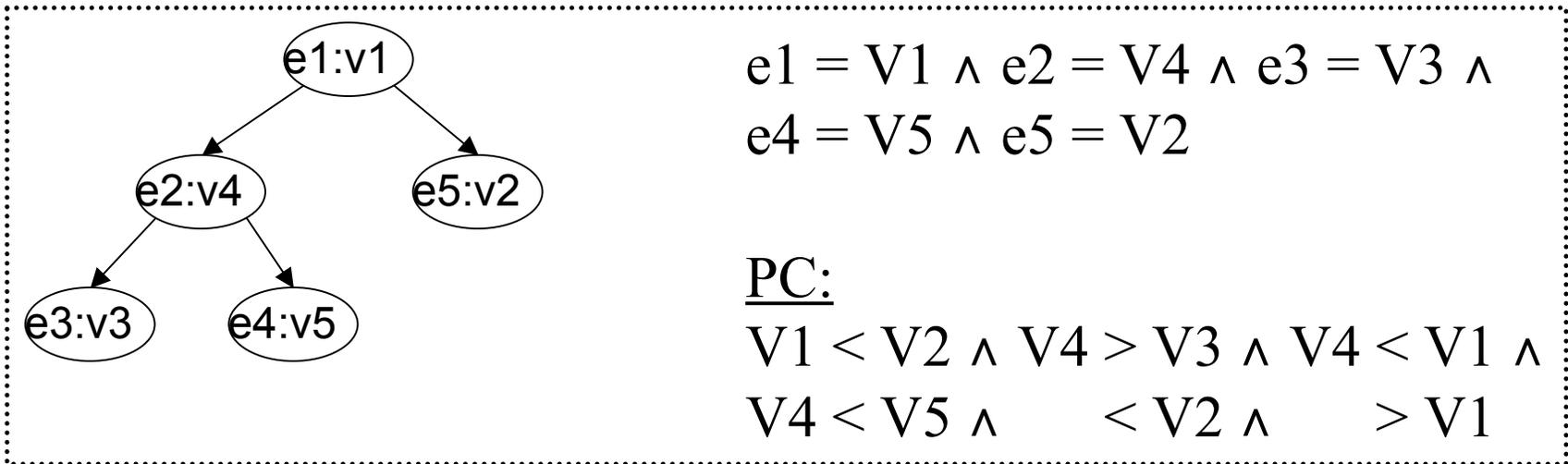
Set of concrete states represented by new state

Normalized using existential quantifier elimination

# Subsumption Checking



## Existential Quantifier Elimination



$\exists V1, V2, V3, V4, V5, V7:$

$$e1 = V1 \wedge e2 = V4 \wedge e3 = V3 \wedge e4 = V5 \wedge e5 = V2 \wedge PC$$

simplifies to

$$e1 > e2 \wedge e2 > e3 \wedge e2 < e4 \wedge e5 > e1$$

# 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

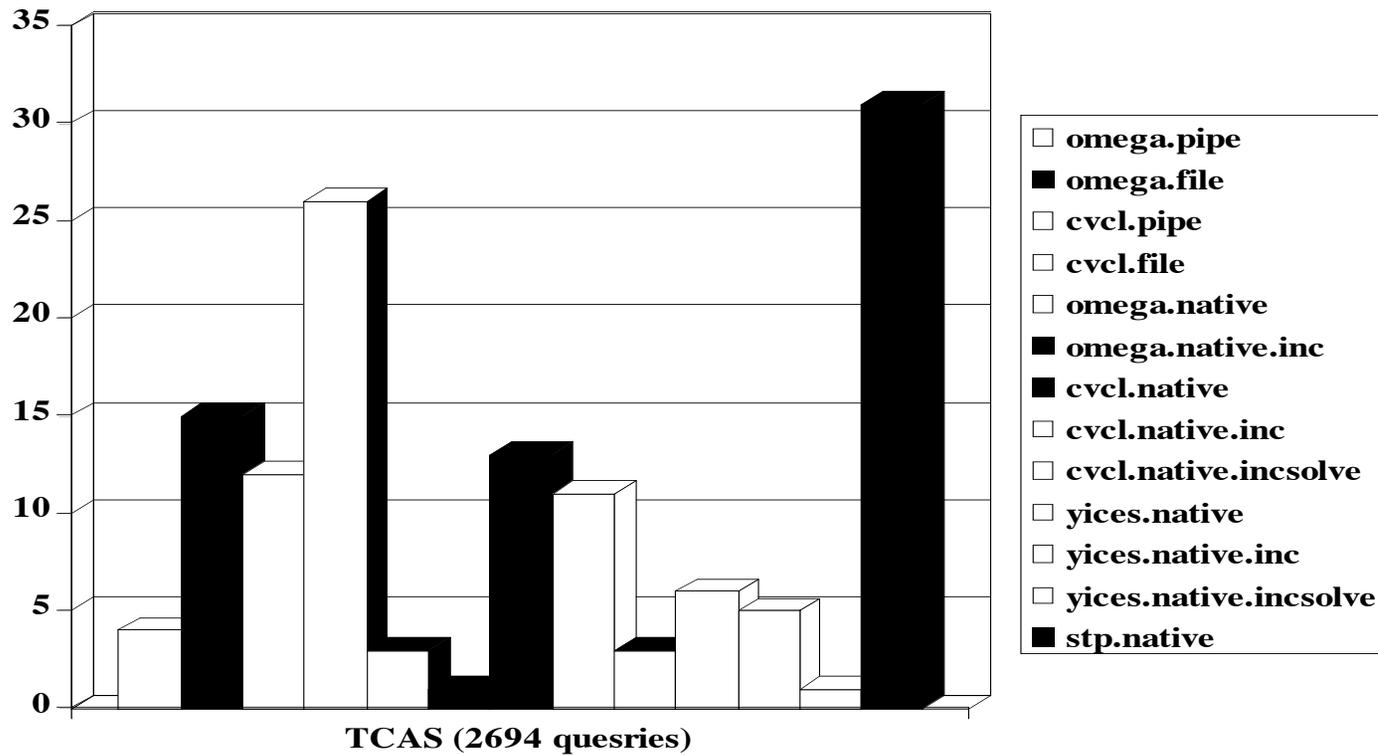
# Decision Procedure Options

---



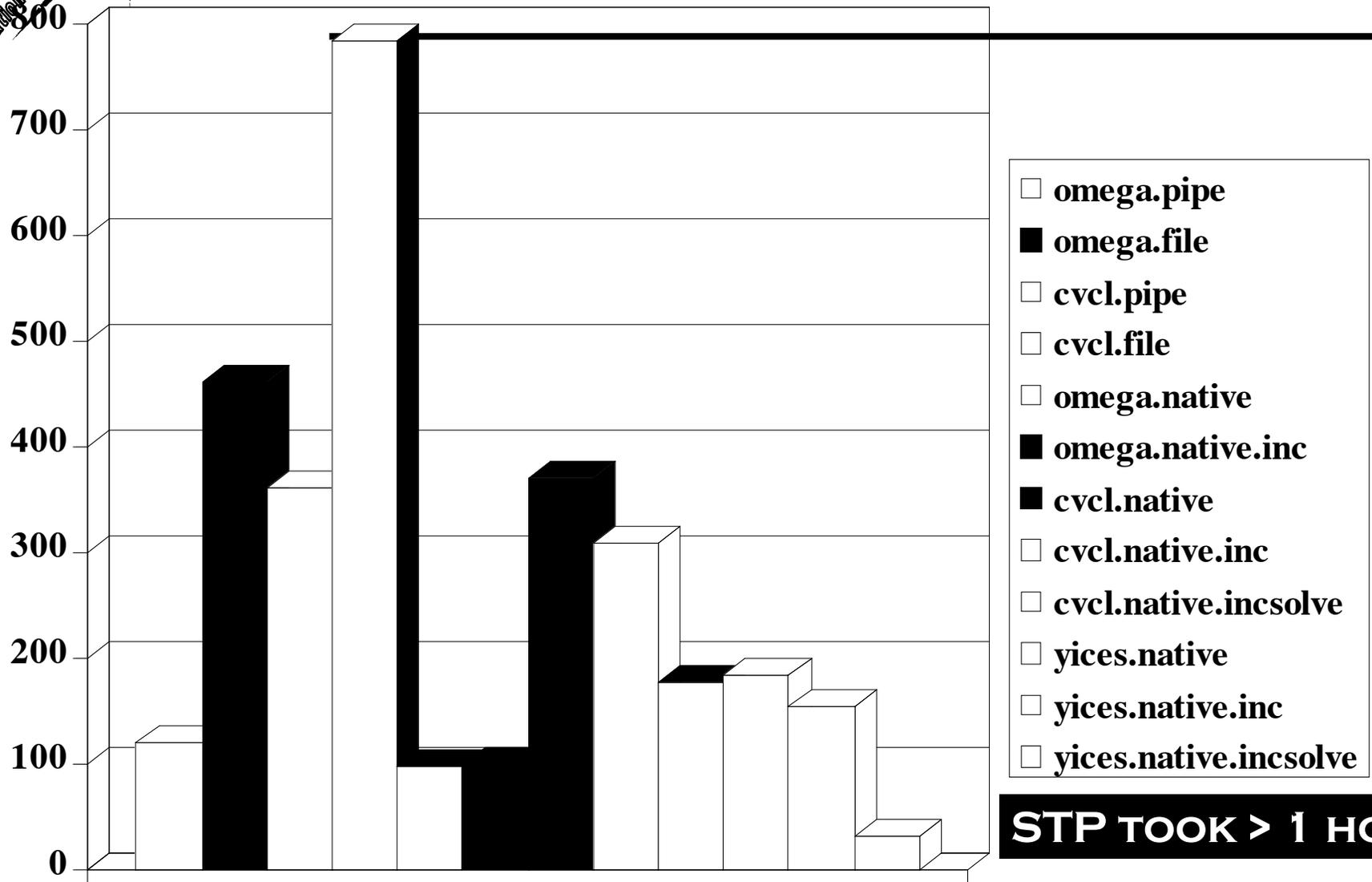
- +symbolic.dp=
  - omega.file
  - omega.pipe
  - omega.native
  - omega.native.inc
    - ...inc - with table optimization
  - yices.native
  - yices.native.inc
  - yices.native.incsolve
    - ...incsolve - Table optimization and incremental solving
  - cvcl.file
  - cvcl.pipe
  - cvcl.native
  - cvcl.native.inc
  - 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



TreeMap size 6 (83592 queries)

**STP TOOK > 1 HOUR**

# Applications

---



- Test input generation for NASA flight control software: abort logic (400 LOC)
  - Symbolic execution generated 150 test cases in ~30 seconds
  - Covered all flight rules/aborts in a few seconds, discovered errors
  - Random testing covers only a few flight rules (no aborts)
  - Manual test case generation took ~20 hours