State Extensions for Java PathFinder

Darko Marinov
University of Illinois at Urbana-Champaign

JPF Workshop
May 1, 2008

Broader Context: Work on JPF at UIUC

- Two lines of work
  - Extend functionality
  - Improve performance
- Summary
  - Techniques: Incremental state-space exploration, Delta execution, Mixed execution, Symbolic exec.
  - Six papers: TSE 2008, ICSE 2008, ICSE Demo 2008, ISSTA 2007, ICFEM 2006, ASE 2006
  - Several contributions to JPF codebase: overflow checking, untracked fields, bug fixes

Collaborators

- Darko’s grad students
  - Marcelo d'Amorim (PhD 2007), Steven Lauterburg
- Undergrad visitors from University of Belgrade
  - Milos Gligoric, Tihomir Gvero, Aleksandar Milicevic, Sasa Misailovic
- Other researchers from UIUC
  - Ahmed Sobeih, Mahesh Viswanathan
- Other researchers from elsewhere
  - Carlos Pacheco (MIT), Michael Ernst (MIT), Sarfraz Khurshid (UT Austin), Tao Xie (NCSU)

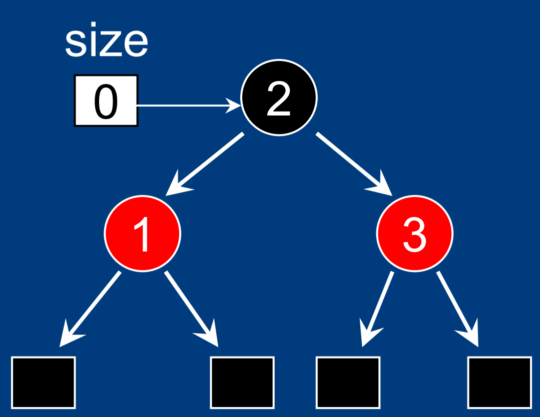
Java PathFinder

- Java PathFinder (JPF) is an explicit-state model checker for Java programs
  - Used to find bugs in programs or verify properties
- Takes as input a Java program and explores all executions that the program can have
- JPF generates as output:
  - Executions that violate given properties
  - Test inputs for the given program
  - Statistics about state-space exploration

Example: Red-black tree

Simplified class TreeMap:

```
class TreeMap {
  int size; Entry root;
  static class Entry {
    int key, value; boolean color;
    Entry left, right, parent; ...
  }
  void put(int key, int value) { ... }
  void remove(int key) { ... }
}
```

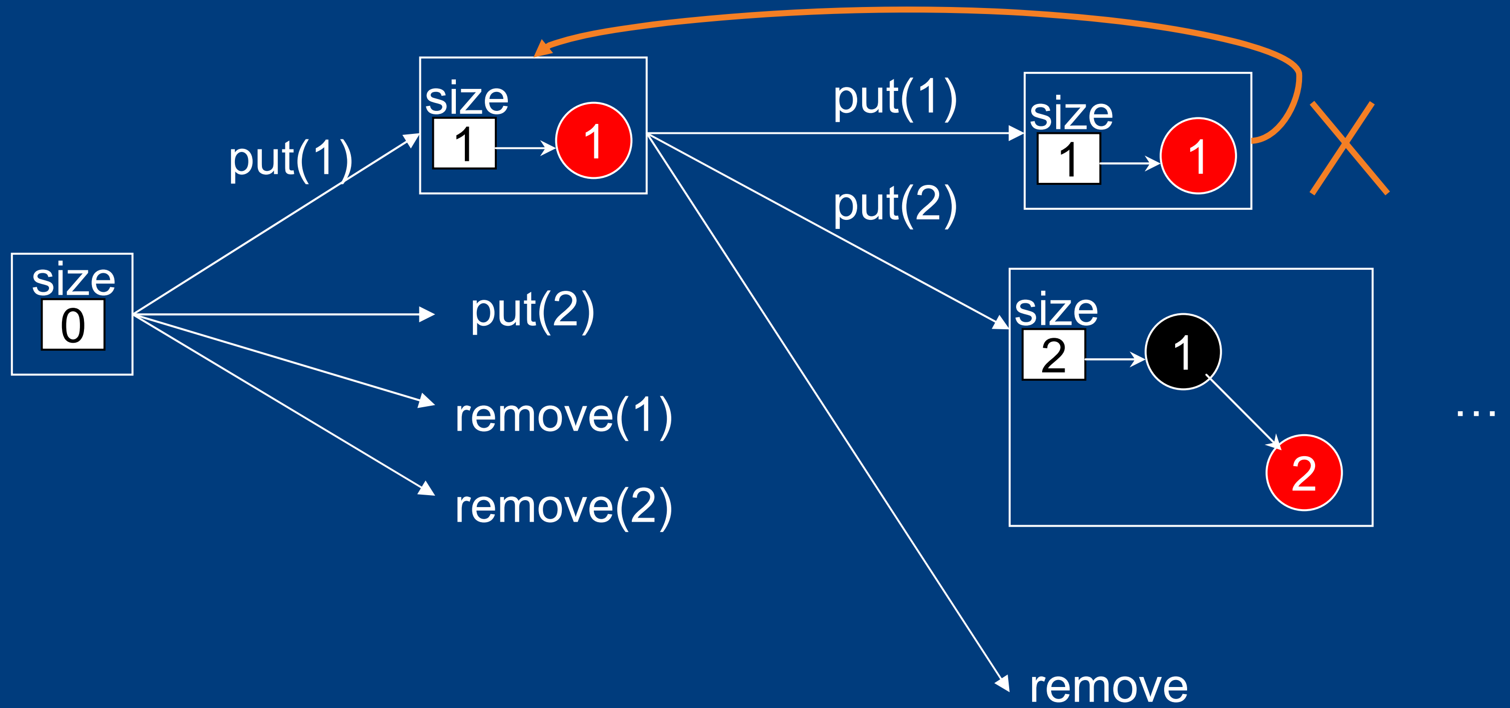


A driver for exploration of tree states:

```
// input bounds sequence length
// and range of input keys
static void main(int N) {
  // an empty tree, the root object for exploration
  TreeMap t = new TreeMap();
  for (int i = 0; i < N; i++) {
    int methodNum = Verify.getInt(0, 1);
    switch (methodNum) {
      case 0: t.put(Verify.getInt(1, N), 0); break;
      case 1: t.remove(Verify.getInt(1, N)); break;
    }
    Verify.ignoreIfPreviouslySeen(t);
    // incrementCounters(methodNum == 1);
  }
}
```

Generates method sequences, not directly object graphs (which Korat does)

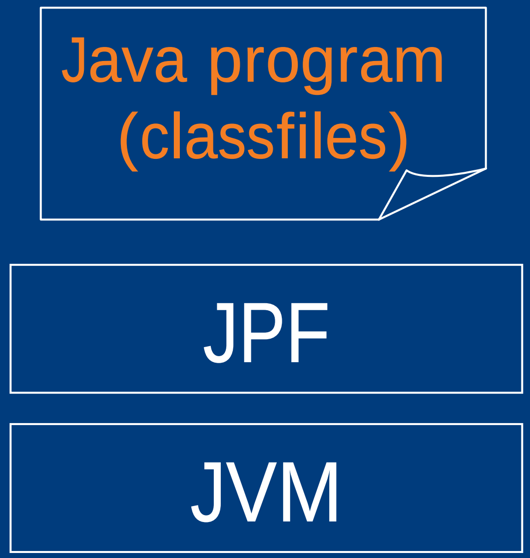
Explicit-state model checking

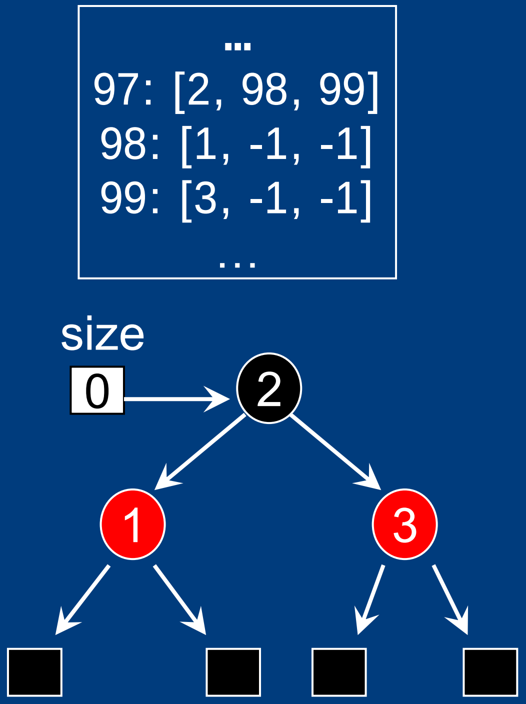


- Store state
- Take next value
- Execute operation
- Prune path (if state was seen)
- Restore state (backtrack)

Java PathFinder – state representation

- JPF is a backtrackable Java Virtual Machine (JVM)
  - Runs on the top of the host JVM
- Uses special representation for state of model checked program





Java PathFinder – operations

- Operations on special state representation:
  - Bytecode execution: manipulates state to execute program bytecodes
  - State backtracking: stores/restores state to backtrack execution for state-space exploration
  - State comparison: detects cycles in the state space

Java PathFinder – MJI

- Model Java Interface (MJI)
  - Allows host JVM code to manipulate JPF state
  - Provides a mechanism for executing parts of application code on the host JVM
  - Similar to JNI for Java/JVM
- Quote from JPF documentation: "*For components that are not property-relevant, it makes sense to delegate the execution from the state-tracked JPF into the non-state tracked host VM.*"

Back to the example

```
// input bounds sequence length
// and range of input keys
static void main(int N) {
  // an empty tree, the root object for exploration
  TreeMap t = new TreeMap();
  for (int i = 0; i < N; i++) {
    int methodNum = Verify.getInt(0, 1);
    switch (methodNum) {
      case 0: t.put(Verify.getInt(1, N), 0); break;
      case 1: t.remove(Verify.getInt(1, N)); break;
    }
    Verify.ignoreIfPreviouslySeen(t);
    incrementCounters(methodNum == 1);
  }
}

static int totalCounter = 0, lastRemoveCounter = 0;
static void incrementCounters(boolean isLastRemove) {
  totalCounter++;
  if (isLastRemove) lastRemoveCounter++;
}
```

What happen when code changes?

Choose one by one?

Execute more efficiently?

Variables lose values after backtracking?

Some of our state extensions

```
// input bounds sequence length
// and range of input keys
static void main(int N) {
  // an empty tree, the root object for exploration
  TreeMap t = new TreeMap();
  for (int i = 0; i < N; i++) {
    int methodNum = Verify.getInt(0, 1);
    switch (methodNum) {
      case 0: t.put(Verify.getInt(1, N), 0); break;
      case 1: t.remove(Verify.getInt(1, N)); break;
    }
    Verify.ignoreIfPreviouslySeen(t);
    incrementCounters(methodNum == 1);
  }
}

static int totalCounter = 0, lastRemoveCounter = 0;
static void incrementCounters(boolean isLastRemove) {
  totalCounter++;
  if (isLastRemove) lastRemoveCounter++;
}
```

Incremental Checking: reuse for code changes

Delta Execution: execute all together

Mixed Execution: execute methods on JVM

Untracked State: not backtrack some fields

Extensions target JPF operations

	Bytecode execution	State backtracking	State comparison
Untracked State		X	
Delta Execution	X	X	X
Mixed Execution	X		
Incremental Checking	X		X

Outline

- Overview
- Untracked State
- Delta Execution
- Mixed Execution
- Incremental Checking
- Conclusions

Untracked state [Gvero et al. 2008]

- Provides a new functionality in JPF
  - By default, JPF stores and restores the entire JVM state during backtracking
  - Untracked State allows the user to mark that certain parts of the state JPF should not restore during backtracking
- Useful for collecting some information about all execution paths, e.g., counting some events or measuring coverage

Changes

- Added Java annotation: @UntrackedField
- Used to mark some fields as untracked, i.e., not to be restored during backtracking

```
@UntrackedField
static int totalCounter = 0;
@UntrackedField
static int lastRemoveCounter = 0;
static void incrementCounters(boolean isLastRemove) {
    totalCounter++;
    if (isLastRemove) lastRemoveCounter++;
}
```

Untracked state - definition

- Our implementation allows both static and non-static fields, as well as primitive and reference fields, to be marked as untracked
- An object is untracked if all its fields are untracked
- If an untracked reference points to an object, that object and all objects reachable from it are untracked
  - Gets tricky with aliasing (some tracked, some untracked references), details in paper & code doc

Our implementation

- New package gov.nasa.jpf.jvm.untracked
- Several changes to existing classes, aiming to minimally affect existing JPF code
  - Did not change the way that JPF stores the state: JPF still stores all fields of all objects, even if some are untracked
  - Only changed the way that JPF restores the state to avoid restoring untracked fields and objects
- Our code is integrated in JPF’s repository
  - Thanks to Peter for feedback

Previous solution

- Before we added @UntrackedField to JPF, one had to maintain state not backtracked by JPF using MJI or listeners
- MJI requires much more coding, for counters:
  - Mark the incrementCounters method as native
  - Provide a separate class that implements this method, keeping state on host JVM
- Listeners
  - Can intercept certain events
  - Manipulating JPF state still requires MJI

Outline

- Overview
- Untracked State
- Delta Execution
- Mixed Execution
- Incremental Checking
- Conclusions

Delta execution [d'Amorim et al. 2007]

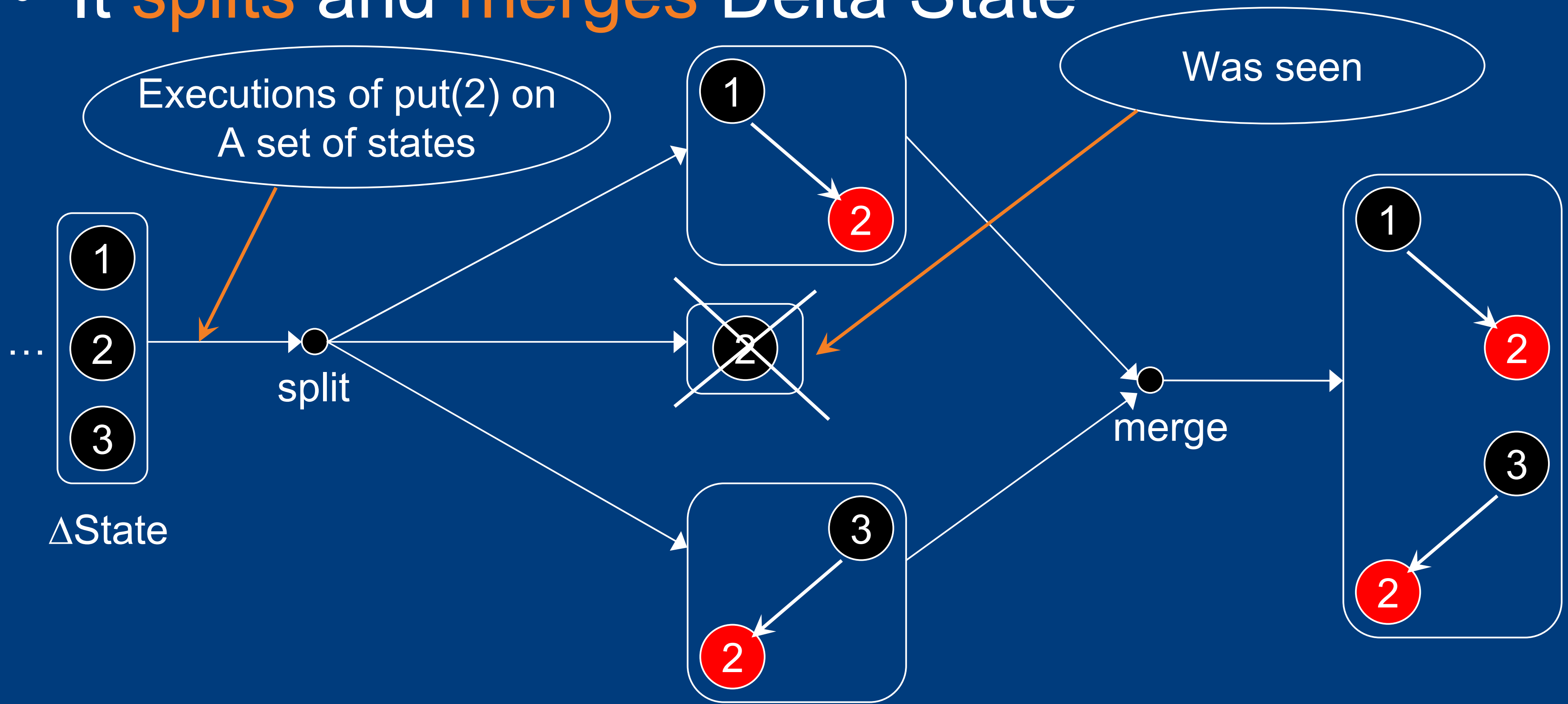
- Goal is to speed up state-space exploration
- Exploits the fact that many execution paths overlap during exploration
- Key idea: share overlapping parts of multiple executions and separately execute only those parts that differ

Our approach

- Manipulate several states at once
  - A novel representation for a set of concrete states (called Delta State)
  - Efficient operations for that representation
- Targets all three major JPF state operations
  - Bytecode execution operates on Delta State
  - State backtracking restores Delta State
  - State comparison handles many states at once

Brief illustration

- Executes a method/value combination at once against multiple TreeMap states, combined into a single Delta State
- It splits and merges Delta State



Some experimental results

Subject-Bound	Exploration Time (sec)			# States	# Executions	
	Standard	Δ Exec	Std / Δ		Std	Δ Exec
binheap-8	458.81	11.91	38.50x	250083	4001328	863
bst-10	214.06	30.13	7.11x	206395	4127900	22688
deque-9	552.11	28.84	19.14x	623530	11223540	810
fibheap-8	400.84	21.59	18.57x	544659	4901931	209
filesystem-4	17.18	3.08	5.59x	1353	194832	1568
heaparray-9	2724.63	21.49	126.80x	804809	8048090	359
queue-7	84.42	5.08	16.63x	147995	1183960	60
stack-7	59.70	4.14	14.43x	137257	1098056	56
treemap-11	90.80	9.43	9.63x	35405	778910	5269
ubstack-9	1502.24	32.54	46.17x	991189	9911890	931
GMEAN			10.79x			

Outline

- Overview
- Untracked State
- Delta Execution
- Mixed Execution
- Incremental Checking
- Conclusions

Mixed execution [d'Amorim et al. 2006]

- Goal is to speed up execution/exploration
- Key idea: execute some parts of the program being checked not on JPF but directly on the host JVM
- Executes on the host JVM deterministic blocks that have no:
  - thread interleavings
  - non-deterministic choices
- This extension targets only bytecode execution

Mixed execution – translation

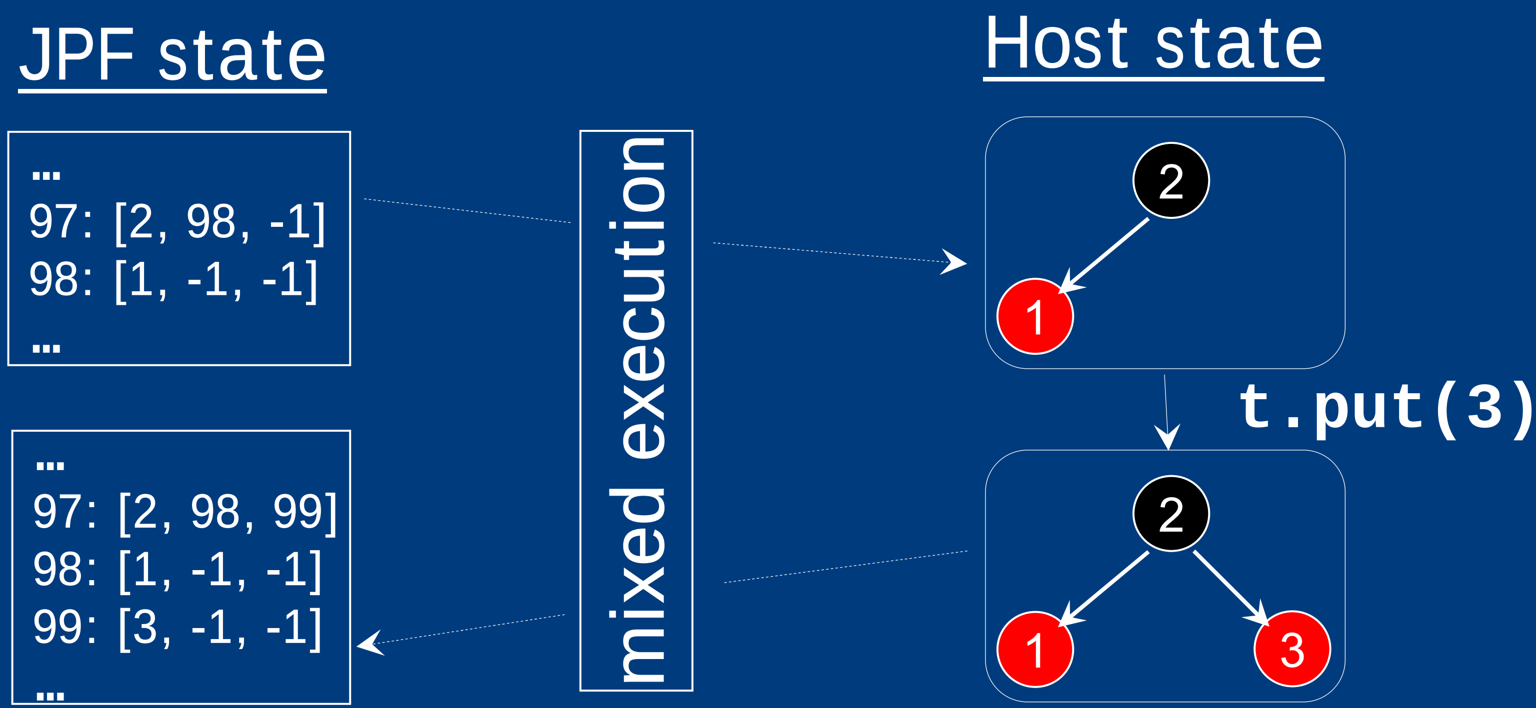
- Translates the state between JPF and JVM:
  - From JPF to JVM at the beginning of a block
  - From JVM to JPF at the end of a block
- Lazy translation
  - Optimization that speeds up Mixed Execution
  - Translates only the parts of the state that an execution accesses (not entire reachable states)

Mixed execution – example

- In the TreeMap driver, executions of the put and remove methods manipulate the tree
- Mixed Execution executes these methods on the host JVM in three steps

Brief illustration

1. translates the objects from the JPF representation into the host JVM representation



2. invokes the method on the translated state

3. translates the state back

Some experimental results

- Evaluated Mixed Execution and lazy translation on six subject programs that use JPF to generate tests for data structures
  - Mixed Execution can improve the overall time for state exploration up to 1.73x
  - Improves the time for execution of deterministic blocks up to 3.05x
- Also evaluated Mixed Execution on a fairly complex routing protocol, AODV, and the results show a speedup of up to 1.41x
- Lazy translation can improve the eager Mixed Execution up to 1.35x

Outline

- Overview
- Untracked State
- Delta Execution
- Mixed Execution
- Incremental Checking
- Conclusions

Incremental checking [Lauterburg+ '08]

- Considers evolving code, basic scenario:
  - Explore state space for one version of code
  - Code changes (bug fix, optimization…)
  - How to explore new version faster?
- Previous work on incremental model checking focuses on control-intensive properties
  - Dynamically allocated data not handled well
- Our goal: speed up JPF for evolving code with dynamically allocated data

Key idea

- Reuse state space graphs from previous exploration to speed up next exploration
- In addition to performing exploration and producing usual output (tests, violations…), produce a state-space graph
  - Nodes in graphs are hashes of states (requires no data layout changes between versions)
  - Edges are transitions (method/value pairs)
- While exploring current version, check if results are known from previous version

Potential savings

- Bytecode execution
  - No need to execute an unchanged transition on a state found in previous exploration (except to build new states for exploration)
- State comparison costs
  - No need to compute hash code of a state if it is found in previous exploration
  - No need to verify correctness property of a state if it is found in previous exploration

Some experimental results

Subject & Bound	Ver.	Time (sec)		
		Non-Inc	ISSE	Savings
aodv 9	1	302.24	302.46	- 0.07%
	2	302.85	113.68	62.46%
	3	302.54	113.64	62.44%
binheap 8	1	416.90	428.02	- 2.67%
	2	404.78	249.13	38.45%
bst 11	1	1782.46	2238.98	- 25.61%
	2	1140.94	807.23	29.25%
filesystem 5	1	1083.80	1085.16	- 0.13%
	2	1064.53	419.03	60.64%
	3	1040.02	409.41	60.63%
filesystem 5	1	1053.24	1064.40	- 1.06%
	2	1045.59	446.91	57.26%
heaparray 8	1	67.36	70.69	- 4.94%
	2	131.73	137.93	- 4.71%

Time savings for non-initial explorations:

-4.71% to 62.46%

(median **56.99%**)

Outline

- Overview
- Untracked State
- Delta Execution
- Mixed Execution
- Incremental Checking
- Conclusions

Conclusions

- Developed several state extensions for JPF
  - Extending functionality
    - Untracked state for (no) backtracking
    - Overflow checking for arithmetic (not in this talk)
  - Improving performance
    - Delta execution: speedup 0.88x-126.80x
    - Mixed execution: speedup up to 1.73x
    - Incremental checking: speedup 0.96x-2.66x
- Contributed some code to the JPF codebase
  - State extensions + bug fixes

Ongoing and future work

- Ongoing work: optimized generation of object graphs (Sarfraz's talk)
  - Several optimizations to get over 10x speedup
  - Undo Backtracking contributed to JPF
- Future work
  - Contribute more code to JPF (this summer: two GSoC mentees and two undergrad visitors)
  - Integrate various extensions (synergistic speedup)
  - Speedup: Replace JPF interpreter with compiler??

http://mir.cs.uiuc.edu/jpf