Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs

Marcelo d'Amorim Steven Lauterburg Darko Marinov

Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{damorim, slauter2, marinov}@cs.uiuc.edu

ABSTRACT

State-space exploration is the essence of model checking and an increasingly popular approach for automating test generation. A key issue in exploration of object-oriented programs is handling the program state, in particular the heap. Previous research has focused on standard program execution that operates on one state/heap. We present Delta Execution, a technique that simultaneously operates on several states/heaps. It exploits the fact that many execution paths in state-space exploration partially overlap and speeds up the exploration by sharing the common parts across the executions and separately executing only the "deltas" where the executions differ.

We have implemented Delta Execution in JPF, a popular general-purpose model checker for Java programs, and in BOX, a specialized model checker that we have developed for efficient exploration of sequential Java programs. We have evaluated Delta Execution for (bounded) exhaustive exploration of ten basic subject programs without errors. The experimental results show that on average Delta Execution improves the exploration time 10.97x (over an order of magnitude) in JPF and 2.07x in BOX. We have also evaluated Delta Execution for one larger case study with errors, where the exploration time improved up to 1.43x.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Program Verification, D.2.5 [Software Engineering]: Testing and Debugging.

General Terms: Performance, Verification.

Keywords: Model checking, delta execution.

1. INTRODUCTION

Software testing and model checking are important approaches for improving software reliability. A core technique for model checking is *state-space exploration* [7]: it starts the program from the initial state, searches the states reachable through executions resulting from non-deterministic choices (including thread interleavings), and prunes the search when it encounters an already visited state. Stateful exploration is also increasingly used to automate test generation, in particular for unit testing of object-oriented programs [12, 14, 25, 42, 44, 45]. In this context, each test creates one or more objects and invokes on them a sequence of methods. State-space exploration can effectively search how different method sequences affect the state of objects and can generate the test sequences that satisfy certain testing criteria [12, 42, 44].

A key issue in state-space exploration is manipulating the program state: saving the state at non-deterministic branch points, modifying the state during execution, comparing states, and restoring the state for backtracking. For object-oriented programs, the main challenge is manipulating the heap, the part of the state that links dynamically allocated objects. Researchers have developed a large number of model checkers for object-oriented programs [1, 8, 16, 20, 29, 31, 35]. These model checkers have focused on efficient manipulation and representation of states/heaps for the usual program execution that operates on one state/heap. We refer to such execution as *standard execution*.

We present Delta Execution, referred to as Δ*Execution*, a technique where *program execution simultaneously operates on several states/heaps*. ΔExecution exploits the fact that many execution paths in state-space exploration partially overlap. ΔExecution speeds up the state-space exploration by sharing the common parts across the executions and separately executing only the "deltas" where the executions differ. The heart of ΔExecution is an *efficient representation and manipulation of sets of states/heaps* for object-oriented programs. ΔExecution is thus related to shape analysis [26, 36, 46], a static program analysis that checks heap properties and operates on sets of states. However, shape analysis operates on abstract states, while ΔExecution operates on concrete states.

ΔExecution is inspired by symbolic model checking (SMC) [7, 24] but considers states that include heap. SMC enabled a breakthrough in model checking as it provided a much more efficient exploration than explicit-state model checking. Conceptually, SMC executes the program on a set of states and exploits the similarity among executions. Typical implementations of SMC represent states with Binary Decision Diagrams (BDDs) [5] that support efficient operations on boolean functions. However, heap operations prevent the direct use of BDDs for object-oriented programs. Although heaps are easily translated into boolean functions [28, 43], the heap operations—including field reads and writes, dynamic object allocation, garbage collection, and comparisons based on heap symmetry [4, 7, 22, 27, 30]—do not translate directly into efficient BDD operations.

This paper makes the following contributions.

Idea: We propose the idea of sharing similar executions to speed up state-space exploration of object-oriented programs. The key insight is that many execution paths in state-space exploration partially overlap.

Technique: We describe ΔExecution, a specific technique for sharing commonalities across executions and separately executing only

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

```
public class BST {
  private Node root;
  private int size;

  public void add(int info) {
    if (root == null)
      root = new Node(info);
    else
      for (Node temp = root; true; )
        if (temp.info < info) {
          if (temp.right == null) {
            temp.right = new Node(info);
            break;
          } else temp = temp.right;
        } else if (temp.info > info) {
          if (temp.left == null) {
            temp.left = new Node(info);
            break;
          } else temp = temp.left;
        } else return; // no duplicates
    size++;
  }

  public boolean remove(int info) { ... }
}

class Node {
  Node left, right;
  int info; Node(int info) { this.info = info; }
}
```

Figure 1: Excerpt from binary search tree implementing a set.

the "delta" differences. We introduce ΔStates, a novel representation for sets of states, and present efficient operations for manipulating ΔStates.

Implementation: We have implemented ΔExecution in two model checkers, JPF [20, 29] and BOX. JPF is a general-purpose model checker for Java programs; it can explore concurrent code and can save/backtrack complete Java states, including stack and heap. We have developed BOX, a special-purpose model checker that can explore only sequential code and can save/backtrack only heap.

Evaluation: We have evaluated ΔExecution for (bounded) exhaustive exploration. The results on ten basic subject programs show that on average ΔExecution improves the exploration time 10.97x (over an order of magnitude) in JPF and 2.07x in BOX, while taking on average 1.51x less memory in JPF and roughly the same amount of memory in BOX. We have also evaluated ΔExecution for one larger case study with errors, where the exploration time improved up to 1.43x.

2. EXAMPLE

We next present an example that illustrates how ΔExecution speeds up the state-space exploration compared to standard execution. Figure 1 shows a binary search tree class that implements a set. Each `BST` object stores the size of the tree and its root node, and each `Node` object stores an integer value and references to the two children. The `BST` class has methods to add and remove tree elements. A test sequence for the binary search tree class consists of a sequence of method calls, for example `BST t = new BST(); t.add(1); t.remove(2);`.

The goal of state-space exploration is to explore different sequences of method calls. A common exploration scenario is to exhaustively explore all sequences of method calls, up to some bound [14, 42, 45]. Such exploration does not actually enumerate all sequences but instead uses state comparison to prune sequences that exercise the same states [42, 45].

Figure 2 shows an example driver program that enables a model checker to systematically explore different states of the tree. (The code as shown is for standard execution, and the commented parts

```
// N bounds sequence length and parameter values
public static void mainStandard(int N) {
/* public static void mainDelta(int N) { */
  BST bst = new BST(); // empty tree
  for (int i = 0; i < N; i++) {
    /* bst = Delta.newIteration(bst); */
    int methNum = Verify.getInt(0, 1);
    int value = Verify.getInt(1, N);
    /* Delta.newValue(); */
    switch (methNum) {
      case 0: bst.add(value); break;
      case 1: bst.remove(value); break;
    }
    Standard.stopIfVisited(bst); /* Delta.merge(bst); */
  }
}
```

Figure 2: Drivers for standard execution and ΔExecution.

are for ΔExecution.) The driver creates the initial state of the binary search tree and exhaustively explores sequences (up to length N) of the methods `add` and `remove` (with values between 1 and N). The driver selects different methods and input values using the library method `getInt(int lo, int hi)` that introduces a non-deterministic choice point to return a number between `lo` and `hi`.

The standard driver discards from further exploration any sequence that results in a state that has already been visited; the driver uses the library method `stopIfVisited(Object root)` that ignores the current execution path and forces backtracking (to a preceding choice point) if the state reachable from `root` has already been visited in the exploration. Note that the comparison of states is performed only at the method boundaries (not during method execution), which naturally partitions an execution path into subpaths that each cover execution of one method invocation. As in other related studies [12, 42, 45], we consider a breadth-first exploration of the state space. (A depth-first exploration could miss parts of the state space since state comparison could eliminate a state with a shorter sequence in favor of a state with a longer sequence.)

Figure 3 illustrates some states that arise in the state-space exploration corresponding to the call `mainStandard(4)`. Among other states, the exploration visits the five trees of size three shown at the top of the figure. (For simplicity, the figure does not show the `BST` object that contains size `3` and points to the root node.) The exploration executes `add(4)` on the five trees of size three. The standard driver separately executes `add(4)` on each pre-state, resulting in the five post-states shown at the bottom of the figure.

While standard execution invokes `add(4)` separately against each standard state, ΔExecution invokes `add(4)` simultaneously against *a set of standard states*. ΔExecution itself operates on one state, called a Δ*State*, which represents a set of individual standard states. We call the operation that combines standard states into a ΔState *merging*. The top of Figure 3 illustrates one set consisting of the five pre-states. (Section 3.1 describes how to efficiently represent a ΔState, and Section 3.5 describes how to efficiently merge states.)

During program execution, ΔExecution occasionally needs to *split* the ΔState. Informally, we say that *a state (or set of states) follows an execution path* if ΔExecution operates on that state as it executes that path. For `add(4)`, for example, the five pre-states follow the same execution path until the first check of `temp.right == null`. At that point, ΔExecution splits the set of states: one subset (of two states) follows the `true` branch, and the other subset (of three states) follows the `false` branch. Note that the split enforces the invariant that all states in a set follow the same path.

Each split introduces a non-deterministic choice point in the execution. For `add(4)`, one execution with two states terminates after creating a node with value `4` and assigning it to the right of the root. The figure depicts this execution with the left arrow. The other exe-

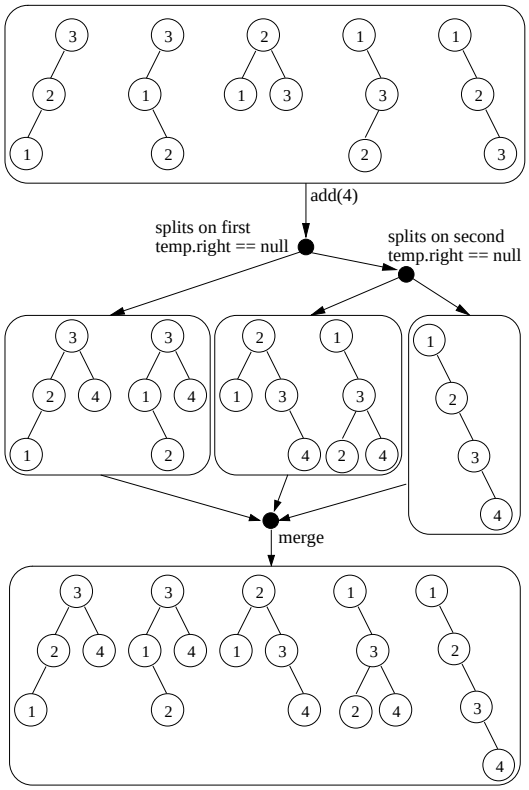

Figure 3: Executions of `add(4)` on a set of states.

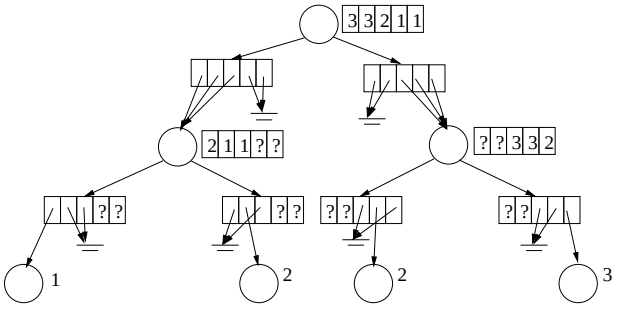

Figure 4: ΔState for the five pre-states from Figure 3.

cution with three states splits at the second check of `temp.right == null`: two (middle) states follow the `true` branch, and one (rightmost) state follows the `false` branch. These two executions terminate without further splits, appropriately adding the value `4` to the final trees.

We next describe the *merging* that ΔExecution performs to build a ΔState from individual states. Merging is a dual operation of splitting: while splitting partitions a set of states into subsets, merging combines several sets of states (or several individual states) into a larger set. In principle, merging can be performed on any sets of states whenever the executions associated with those states reach the same program point. For example, ΔExecution could merge all three sets of states from Figure 3 when they reach `size++`. However, our current implementation of ΔExecution considers only the program points that are method boundaries: it merges the states only after all of them finish the execution path for one method, since that is also where state comparison is done.

Figure 2 also shows a driver (obtained by using the commented code) that explores states using ΔExecution. The delta driver is similar to the standard driver: both use non-deterministic choices to select different methods and input values, both prune the exploration based on the state of `bst`, and both use breadth-first exploration. However, the delta driver differs from the standard driver in the way it operates on the state. First, `bst` in the delta driver is a ΔState that represents several individual trees. Second, the delta driver backtracks the state differently than the standard driver. Specifically, the method `newIteration` returns one ΔState of all individual states that should be explored in a given iteration. In the first iteration, this ΔState is a singleton that has only the initial state (with the empty tree). The method `merge` at the end of one method execution path collects those trees (from `bst`) that have not been previously visited and thus should be explored in the next loop iteration. Effectively, the driver combines all distinct states reachable with the method sequences of length `i` into one ΔState for the iteration `i+1`. The method `newValue` updates the internal state for ΔExecution as backtracking should not restore some parts of that internal state.

We next discuss how the performance of ΔExecution and standard execution compare. In our running example, ΔExecution requires only three execution paths to reach all five post-states that `add(4)` creates for the five pre-states. Additionally, these three paths share some prefixes that can be thus executed only once. In contrast, standard execution requires five executions of `add(4)`, one execution for each pre-state, to reach the five post-states. Also, each of these five separate executions needs to be executed for the entire path. The trade-off between ΔExecution and standard execution can be summarized like this: ΔExecution performs fewer executions (avoiding separate execution of the same path shared by multiple states) than standard execution, but each execution in ΔExecution (that operates on a set of standard states) is more expensive than in standard execution (that operates on one standard state). It is also important to note that the presence of constants (i.e., values that are the same across a set of states) is essential to efficient operations under ΔExecution. Whether ΔExecution is faster or slower than standard execution for some exploration depends on several factors, including the number of execution paths, the number of splits, the cost to execute one path, the number of constants, and the sharing of execution prefixes.

The experimental results from Section 5 show that ΔExecution is faster than standard execution for a number of subject programs and values for the bound N from the drivers. For example, for the binary search tree example and $N = 10$, ΔExecution speeds up JPF 4.41x and our model checker BOX 1.67x, while using over 2x more memory in JPF and 3x more memory in BOX. (On average, ΔExecution uses as much memory as standard execution.)

3. TECHNIQUE

The key idea of ΔExecution is to execute a program simultaneously on a set of standard states. We first discuss Δ*States* that represent sets of states. We describe in detail two main operations on ΔStates: *splitting*, which divides a set of states into subsets for executing different program paths, and *merging*, which combines several states together into a set. We also present how program execution works in ΔExecution and how ΔExecution facilitates an optimized comparison of states.

3.1 ΔState

ΔExecution represents a set of individual standard states as a single Δ*State*. Each ΔState encodes all the information from the orig-

inal individual states. A ΔState includes Δ*Objects* that can store multiple values (either references or primitives) that exist across the multiple individual states represented by a ΔState.

Figures 5, 6, and 7 show the classes used to represent ΔStates for the binary search tree example. We discuss here only the field declarations from those classes. (The methods from those classes implement the operations on ΔState and are explained later in the text.) Each object of the class `DeltaNode` stores a collection of references to `Node` objects, and each object of the class `DeltaInt` stores a collection of primitive integer values. The `BST` and `Node` objects are changed such that they have fields that are ΔObjects.

Figure 4 shows the ΔState that represents the set of five pre-states from Figure 3. Each ΔState consists of layers of "regular" objects and ΔObjects. In this ΔState, each of the pre-states has a corresponding *state index* that ranges from 0 to 4. Note that we could extract each of the five pre-states by traversing the ΔState while indexing it with the appropriate state index. For example, we can extract the balanced tree using state index 2. Also note that some of the values in the example ΔState are "don't cares" (labeled with '`?`') because the corresponding object is not reachable for that state index. For example, the first node to the left of the root has '`?`' in the field `info` for the last two states (with indexes 3 and 4) because those states have the value `null` for the field `root.left`.

While each ΔObject conceptually represents a collection of values, the implementation does not always need to use collections or arrays. In particular, a value is often constant across all (relevant) states. For example, the `info` fields for all tree leaves in Figure 4 have constant values (for the relevant states). Our implementation uses an *optimized representation for constants*. The optimization is straightforward, and we do not discuss it in detail. We point out, however, that the optimization is important both for reducing the memory requirements of ΔStates and for improving the efficiency of operations on ΔStates.

3.2 Splitting

ΔExecution operates on a ΔState that represents a set of standard states. ΔExecution can perform many operations on the entire set. It needs to *split* the set only at a branch control point (e.g., an `if` statement) where some states from the set evaluate to different branch outcomes (e.g., for one subset of states, the branch condition evaluates to true, and for the other subset of states, it evaluates to false). We call such points *split points*; effectively, they introduce non-deterministic choice points as ΔExecution needs to explore both outcomes. (Note that no split is necessary even for branch control points when all states evaluate to the same branch outcome.)

One challenge in ΔExecution is to efficiently split ΔStates. Our solution is to introduce a *state mask* that identifies the currently *active states* within a ΔState. Each state mask is a set of state indexes. At the beginning of an execution, ΔExecution initializes the state mask to the set of all state indexes. For example, the execution of `add(4)` for the ΔState from Figure 4 starts with the state mask being $\{0, 1, 2, 3, 4\}$.

At the appropriate branch points, ΔExecution needs to split the set of states into two subsets. Our approach does not explicitly divide a ΔState into two ΔStates; instead, it simply changes the state mask to reflect the splitting of the set of states. Specifically, ΔExecution builds a new state mask to identify the new subset of active states in the ΔState. It also saves the state mask for the other subset that should be explored later on. The execution then proceeds with the new subset.

After ΔExecution finishes the execution path for some (sub)set of states, it *backtracks* to some unexplored split point to explore the other path using the state mask saved at the split point. Backtracking changes the state mask but restores the ΔState to exactly what it was at the split point. Backtracking can be implemented in several ways; Section 4 discusses how JPF uses state saving and restoration while BOX uses re-execution.

To illustrate how the state mask changes during the execution, consider the example from Figure 3. The state mask is initially $\{0, 1, 2, 3, 4\}$. At the first split point, the execution proceeds with the state mask being $\{0, 1\}$. After the first backtracking, the state mask is set to $\{2, 3, 4\}$. At the second split point, the execution proceeds with the state mask being $\{2, 3\}$. After the second backtracking, the state mask is set to $\{4\}$ for the final execution.

Appropriate use of a state mask can facilitate optimizations on the ΔState. Consider, for example, a ΔObject that is not a constant when all states are active. This object can temporarily be transformed into a constant if all its values are the same for some state mask occurring during the execution. For instance, in our running example, the value of `root.right` becomes the constant `null` when the state mask is $\{0, 1\}$. Additionally, the state mask allows the use of *sparse representations* for ΔObjects: instead of using an array to map all possible state indexes into values, a sparse ΔObject can use representations that *map only the active state indexes into values*, thereby reducing the memory requirement.

3.3 Program execution model

We next discuss how ΔExecution executes program operations. The key is to execute each operation simultaneously on a set of values. ΔExecution uses a non-standard program execution that manipulates a ΔState that represents a set of standard states. Such non-standard execution can be implemented in two ways: (1) instrumenting the code such that the regular execution of the instrumented code corresponds to the non-standard execution [25, 40, 45] or (2) changing the execution engine such that it interprets the operations in the non-standard semantics [12]. Our current implementation uses instrumentation: the subject code is preprocessed to support ΔExecution.

We use parts of the instrumentation to describe the semantics of ΔExecution.

Classes: The instrumentation changes the original program classes and generates new classes for ΔObjects. Figure 1 from Section 2 shows a part of the original code for the binary search tree example. Figures 5, 6, and 7 show the key parts of the instrumented code for this example. Figure 5 shows the instrumented version of the original `BST` and `Node` classes. Figure 6 shows the new class `DeltaNode` that stores and manipulates the multiple `Node` references that can exist across the multiple states in a ΔState. Figure 7 shows the class `DeltaInt` that stores and manipulates multiple `int` values; this class is a part of the ΔExecution library and is not generated anew for each program.

It is important to note that ΔObjects are immutable from the perspective of the instrumented code in the same way that regular primitive and reference values are immutable for standard execution. This allows sharing of ΔObjects. For example, this allows direct assignment of one `DeltaInt` object to another (e.g., `int x = y` simply becomes `DeltaInt x = y`). Our implementation internally mutates ΔObjects to achieve higher performance, in particular when values become constant across active states. The mutation handles the situations that involve shared ΔObjects and require a "copy-on-write" cloning.

Types: The instrumentation changes all types in the original program to their delta versions. Comparing figures 1 and 5, notice that the occurrences of `Node` and `int` have been replaced with the new `DeltaNode` class (from Figure 6) and the `DeltaInt` class (from Fig-

```
public class BST {
  private DeltaNode root = DeltaNode.NULL;
  private DeltaInt size = DeltaInt._new(0);

  public void add(DeltaInt info) {
    if (get_root().eq(DeltaNode.NULL))
      set_root(DeltaNode._new(info));
    else
      for (DeltaNode temp = get_root(); true; )
        if (temp.get_info().lt(info)) {
          if (temp.get_right().eq(DeltaNode.NULL)) {
            temp.set_right(DeltaNode._new(info));
            break;
          } else temp = temp.get_right();
        } else if (temp.get_info().gt(info)) {
          if (temp.get_left().eq(DeltaNode.NULL) {
            temp.set_left(DeltaNode._new(info));
            break;
          } else temp = temp.get_left();
        } else return; // no duplicates
    }
    set_size(get_size().add(DeltaInt._new(1)));
  }

  public DeltaBoolean remove(DeltaInt info) { ... }
}

class Node {
  DeltaNode left, right;
  DeltaInt info;
  Node(DeltaInt info) { this.info = info; }
}
```

Figure 5: Instrumented `BST` and `Node` classes.

ure 7), respectively. The instrumentation also appropriately changes all definitions and uses of fields, variables, and method parameters to use ΔObjects.

Field accesses: The instrumentation replaces standard object field reads and writes with calls to new methods that read and write fields across multiple objects. For example, all reads and writes of `Node` fields are replaced with calls to getter and setter methods in `DeltaNode`. Consider, for instance, the field read `temp.left`. In ΔExecution, `temp` is no longer a reference to a single `Node` object but a reference to a `DeltaNode` object that tracks multiple references to possibly many different `Node` objects. The `left` field of `Node` is now accessed via the `get_left` method in `DeltaNode`. This method returns a `DeltaNode` object that references (one or more) `Node` objects that correspond to the `left` fields of all `temp` objects whose states are active in the state mask. In general, this can result in an execution split when some objects in `temp` are `null`.

Operations: The instrumentation replaces (relational and arithmetic) operations on reference and primitive values with method calls to `DeltaNode` and `DeltaInt` objects. All original operations on values now operate on ΔObjects that represent sets of values. More precisely, the methods in ΔObjects do not need to operate on all values but only on those values that correspond to the active state indexes as indicated by the state mask.

For an example arithmetic operation, consider integer addition. In standard execution, the addition takes two integer values and creates a single value. In ΔExecution, it takes two `DeltaInt` objects and creates a new `DeltaInt` object. The `add` method in `DeltaInt` (Figure 7) shows how ΔExecution conceptually performs pairwise addition across all active state indexes for the two `DeltaInt` objects. Our implementation optimizes the cases when those objects are constant (to avoid the loop or state indexing).

For an example relational operation, consider reference equality. The method `eq` in `DeltaNode` (Figure 6) performs this operation across all active state indexes. Note that this method can create a split point in the execution if the result of the operation differs

```
class DeltaNode {
  // maps each state index to a Node object
  Node[] values; // conceptually

  DeltaNode(int size) { values = new Node[size]; }
  private DeltaNode(Node n) { values = new Node[]{ n }; }
  public static DeltaNode _new(DeltaInt info) {
    return new DeltaNode(new Node(info));
  }

  public boolean eq(DeltaNode arg) {
    StateMask sm = StateMask.getStateMask();
    StateMask trueMask = new StateMask(sm.size());
    StateMask falseMask = new StateMask(sm.size());
    foreach (int index : sm)
      if (values[index] == arg.values[index])
        trueMask.enable(index);
      else
        falseMask.enable(index);
    boolean result;
    if (trueMask.isEmpty()) result = false;
    else if (falseMask.isEmpty()) result = true;
    else result = (Verify.getInt(0, 1) == 0); // split
    StateMask.setStateMask(result ? trueMask : falseMask);
    return result;
  }

  public DeltaNode get_left() {
    StateMask sm = StateMask.getStateMask();
    DeltaNode result = new DeltaNode(sm.size());
    foreach (int index : sm) {
      DeltaNode dn = values[index].left;
      result.values[index] = dn.values[index];
    }
    return result;
  }
  public void set_left(DeltaNode arg) {
    StateMask sm = StateMask.getStateMask();
    IdentitySet<Node> set = new IdentitySet<Node>();
    foreach (int index : sm) {
      Node n = values[index];
      if (set.add(n)) // true if n was added
        n.left = n.left.clone();
      n.left.values[index] = arg.values[index];
    }
  }
  public DeltaNode get_right() { ... }
  public void set_right(DeltaNode arg) { ... }
  public DeltaInt get_info() { ... }
  public void set_info(DeltaInt arg) { ... }
}
```

Figure 6: New `DeltaNode` class.

```
class DeltaInt {
  // maps each state index to an integer value
  int[] values; // conceptually

  DeltaInt add(DeltaInt arg) {
    StateMask sm = StateMask.getStateMask();
    DeltaInt result = new DeltaInt(sm.size());
    foreach (int index : sm)
      result.values[index] = values[index] + arg.values[index];
    return result;
  }
  ...
}
```

Figure 7: Part of `DeltaInt` library class.

across the states. If so, `eq` introduces a non-deterministic choice (with `getInt`) that returns a boolean `true` or `false` after appropriately setting the state mask.

Method calls: The instrumentation replaces a standard method call with a method call whose receiver is a ΔObject, which allows making the call on several objects at once. Note that each call introduces a semantic branch point (since different objects may have different dynamic types) and can result in an execution split.

```
void linearize(Object o, StateMask sm) {
  foreach (int index : sm) {
    Pair(Map _, Seq s) = linObject(o, new Map(), index);
    checkVisited(index, s);
  }
}

Pair<Map, Seq> linObject(Object o, Map ids, int index) {
  if (o == null) return Pair(ids, Seq(NULL));
  if (o in ids) return Pair(ids, Seq(ids.get(o)));
  int id = ids.size();
  return linFields(o, ids.put(o, id), Seq(id), index);
  /*return linFields(o, 0, ids.put(o, id), Seq(id), index);*/
}

Pair<Map, Seq> linFields(Object o, Map ids,
                         Seq seq, int index) {
  for (int f = 0; f < o.numberOfFields(); f++) {
    Object fo = o.getField(f).values[index];
    Pair(ids, Seq s) = linObject(fo, ids, index);
    seq = seq.append(s);
  }
  return Pair(ids, seq);
}

Pair<Map, Seq> linFields(Object o, int f, Map ids,
                         Seq seq, int index) {
  if (f < o.numberOfFields()) {
    Object fo = o.getField(f).values[index];
    Pair(Map m, Seq s) = linObject(fo, ids, index);
    return linFields(o, f + 1, m, seq.append(s), index);
  } else return Pair(ids, seq);
}
```

Figure 8: Non-optimized linearization of ΔState.

```
Stack stack; // mutable structure
void linearize(Object o, StateMask sm) {
  stack = new Stack();
  Triple(Map _, Seq s, StateMask tm) =
    linObject(o, new Map(), sm);
  checkVisited(tm, s); // all states from tm have sequence s
  while (!stack.isEmpty()) {
    Tuple(Object o, int f, Map ids,
          Seq seq, StateMask nm) = stack.pop();
    Triple(Map _, Seq s, StateMask tm) =
      linFields(o, f, ids, seq, nm);
    checkVisited(tm, s);
  }
}

Triple<Map, Seq, StateMask>
  linObject(Object o, Map ids, StateMask sm) {
  if (o == null) return Triple(ids, Seq(NULL), sm);
  if (o in ids) return Triple(ids, Seq(ids.get(o)), sm);
  int id = ids.size();
  return linFields(o, 0, ids.put(o, id), Seq(id), sm);
}

Triple<Map, Seq, StateMask>
  linFields(Object o, int f,
            Map ids, Seq seq, StateMask sm) {
  if (f < o.numberOfFields()) {
    Triple(Object fo, StateMask em, StateMask nm) =
      split(o.getField(f), sm);
    if (nm is not empty)
      stack.push(o, f, ids, seq, nm);
    Triple(StateMask om, Map m, Seq s) = linObject(fo, ids, em);
    return linFields(o, f + 1, m, seq.append(s), om);
  } else return Triple(sm, ids, seq);
}
```

Figure 9: Optimized linearization of ΔState.

3.4 Optimized state comparison

Heap symmetry [7, 22, 27, 30] is an important technique that model checkers use to alleviate the state-space explosion problem. Heap symmetry detects equivalent states: when the exploration encounters a state equivalent to some already visited, the exploration path can be pruned. In object-oriented programs, two heaps are equivalent if they are *isomorphic* (i.e., have the same structure and primitive values, while their object identities can vary) [4, 22, 30]. An efficient way to compare states for isomorphism is to use *linearization* (also known as serialization or marshalling) that translates a heap into a sequence of integers such that two heaps are isomorphic if and only if their linearizations are equal.

ΔExecution exploits the fact that different heaps in a ΔState can share prefixes of linearization. Instead of computing linearizations separately for each state in a set of states, ΔExecution *simultaneously computes a set of linearizations* for a ΔState. Sharing the computation for the prefixes not only reduces the execution time but also reduces memory requirements as it enables sharing among the sequences used for linearizations.

We next present how to transform a basic algorithm that separately linearizes each state from a ΔState into an efficient algorithm that simultaneously linearizes all states from a ΔState. Figure 8 shows a pseudo-code of a basic algorithm that iterates over each active state from the state mask and computes the linearization for the individual state. For simplicity of presentation, this algorithm assumes that the heaps contain only reference fields of only one class. Our actual implementation handles general heaps with objects of different classes, primitive fields, and arrays.

The method `linObject` produces a sequence of integers that represent linearization for the state reachable from `o`. When `o` is `null`, `linObject` returns a singleton sequence with the value that represents `null`. When `o` is a reference to a previously linearized object, `linObject` returns a singleton sequence with the identifier used for that object, which handles object aliasing. The map `ids` stores the association between objects and their ids. When `o` is an object not yet linearized, `linObject` creates a new id for it, appropriately extends the map, and linearizes all the object fields.

The method `linFields` linearizes the fields of a given object. A typical implementation is iterative, as shown in the first `linFields` method. It is important to note that the value of the expression `o.getField(f).values[index]` determines the linearizations for different states. We target this expression to be the split point in our optimized linearization algorithm. The algorithm thus needs to explore different execution paths from this point, effectively performing backtracking. We want to implement the optimized algorithm to execute on a regular JVM, so to support backtracking.

An intermediate step in the optimization is to transform the algorithm to conceptually use the continuation-passing style [17]. In practice, the method `linFields` is transformed into a recursive implementation shown in the second `linFields` method. This version exposes the field index `f` and linearizes the fields of `o` between `f` and `o.getNumberOfFields()`. This version permits the linearization to *continue* an execution from the point it was left at in `linFields`. Note that `linFields` and `linObject` manipulate functional objects `Map` and `Seq`, which facilitates backtracking of the state.

Figure 9 shows the pseudo-code of the optimized algorithm that linearizes a ΔState in the ΔExecution mode. The new methods `linObject` and `linFields` do not take one state index but a state mask with several active state indexes to linearize. These methods now return a state mask and one linearization for all the states in that state mask. The linearization can introduce non-deterministic choices to enforce the invariant that all states in the state mask have the same linearization prefix. When the linearization completes for some state mask, it needs to backtrack to explore the remaining state masks.

The `stack` object stores the backtracking points. Each entry stores the state that needs to be restored to continue an execution

from a split point: the root object, the field index, the map for object identifiers, the current linearization sequence, and the state mask. While `stack` is mutable, the other structures are immutable, which makes it easy to restore the state. The `while` loop in `linearize` visits each pending backtracking point until it finishes computing all linearizations.

The only source of non-determinism in the linearization is the reading of fields across different states from the state mask. The method `split` takes as input a ΔObject `do = o.getField(f)` and a state mask `sm`. It returns a standard object `fo = do.values[idx]` for some `idx` from `sm`, a state mask `em` of `index` values such that `do.values[index] == fo`, and a state mask `nm` of `index` values such that `do.values[index] != fo`. At this point, `linFields` first pushes on the stack an entry with the backtracking information for `nm` and then continues the linearization of `fo` for the states in `em`.

3.5 Merging

The dual of splitting sets of states into subsets is *merging* several sets of states into a larger set. Recall the driver for ΔExecution from Figure 2. It merges all non-visited states from one iteration into a ΔState to be used at the start of the next iteration. Specifically, the `merge` method receives as the input a ΔState and (implicitly) a state mask. This method extracts the non-visited states from the ΔState and only stores their linearized representations. The method `newIteration` builds and returns a new ΔState from the stored linearized representations.

Our merging uses *delinearization* to construct a ΔState from the linearized representations of non-visited states. The standard delinearization is an inverse of linearization: given one linearized representation, delinearization builds one heap isomorphic to the heap that was originally linearized. The novelty of our merging is that it operates on a *set* of linearized representations simultaneously and, instead of building a set of standard heaps, it builds one ΔState that encodes all the heaps. It is interesting to point out that we often used in debugging our implementation the fact that linearization and delinearization are inverses; the composition of these functions gives the identity function: for any set of linearizations s, the linearization of the delinearization of s should equal s.

We highlight two important aspects of the merging algorithm. First, it identifies ΔObjects that should be constants (with respect to the reachability of the nodes), which results in a more efficient ΔState. Such constants can occur quite often; for instance, in our experiments (see Section 5), the lowest percentage of the constant ΔObjects in the merged ΔStates is 33%. Second, the merging algorithm *greedily* shares the objects in the resulting ΔState: it attempts to share the same ΔObject among as many individual states as possible. For example, in Figure 4, the left node from the root is shared among three of the five states. A more detailed discussion of the merging algorithm can be found in a technical report [11].

4. IMPLEMENTATION

We have implemented ΔExecution in two model checkers, JPF and BOX. JPF [20, 29] is a popular model checker for Java programs, but it is general-purpose and has a high overhead [13] for the subject programs considered in our study and related studies [13, 41, 42]. We have thus implemented a specialized model checker, called BOX (from *Bounded Object eXploration*), for efficient exploration of such subject programs.

4.1 JPF

We have implemented ΔExecution by modifying JPF version 4. JPF is implemented as a backtrackable Java Virtual Machine (JVM) running on top of a regular, host JVM. JPF provides operations for state-space exploration: storing states, restoring them during backtracking, and comparing them. By default, JPF compares the entire JVM state that consists of the heap, stack (for each thread), and class-info area (that is mostly static but can be modified due to the dynamic class loading in Java). However, our experiments require only the part of the heap reachable from the root object in the driver. We have therefore disabled the JPF's default state comparison and instead use a specialized state comparison as done in some previous studies with JPF [12, 42, 45].

We next discuss how we have implemented each component of ΔExecution in JPF. We call the resulting system ΔJPF. ΔJPF keeps ΔState as a part of the JPF state, which enables the use of JPF backtracking to restore ΔState at the split points. We have implemented the library operations on ΔState (such as arithmetic and relational operations or field reads and writes) to execute on the host JVM. Effectively, the library forms an extension of JPF; our goal is not to model check the library itself but the subject code that uses the library. ΔJPF uses instrumented code to invoke the operations that manipulate the ΔState.

We have implemented splitting in ΔJPF on top of the existing non-deterministic choices in JPF. It is important to point out that our implementation leverages JPF to restore the entire ΔState but uses state masks to indicate the active states. Therefore, ΔJPF manages state masks on the host JVM, outside of the backtracked state. We have implemented merging also to execute on the host JVM and to create one ΔState as a JPF state that encodes all the non-visited states encountered in the previous iteration of the exploration. Recall from Section 2 that the drivers in our experiments use breadth-first exploration. ΔJPF does not use the optimized state comparison (Section 3.4).

To automate the instrumentation of code for execution on ΔJPF, we have developed a plug-in for Eclipse version 3.2 [15]. This plug-in takes a subject program and manipulates its Eclipse internal AST representation to automate the steps described in Section 3.3.

4.2 BOX

We have developed BOX, a model checker optimized for sequential Java programs. JPF is a general-purpose model checker for Java that can handle concurrent code and can store/restore/compare the entire JVM state that consists of heap, stack, and class-info area. However, in unit testing of object-oriented programs, most code is sequential and most drivers need to store/restore/compare only the heap part of the state. Therefore, we have used the existing ideas from state-space exploration research [1, 8, 16, 18, 20, 22, 31, 35] to engineer a high-performance model checker for such cases.

BOX can store/restore/compare only a part of the program heap reachable from a given root. The root corresponds to the main object under exploration in the driver. BOX uses a *stateful* exploration (by restoring the entire state) *across iterations* and *stateless* exploration (by re-executing one method at a time) *within one iteration*. BOX needs to re-execute a method within an iteration as it does not store the state of the program stack. Instead, BOX only keeps a list of changes performed on the heap during a single method execution and restores the state by undoing those changes. For efficient manipulation of the changes, BOX requires that code under exploration be instrumented.

We refer to the ΔExecution implementation in BOX as ΔBOX. ΔBOX needs to backtrack the ΔState in order to explore a method for various state masks. ΔBOX *re-executes* the method from the beginning to reach the latest split point. While re-execution is seemingly slow, it can actually work extremely well in many situations. For example, Verisoft [18] is a well-known model checker that effectively employs re-execution.

experiment		JPF time			JPF mem.	BOX time			BOX mem.	# states	# executions		
subject	N	std	delta	std/delta	std/delta	std	delta	std/delta	std/delta		std	delta	std/delta
binheap	7	25.40	2.66	9.55x	1.16x	0.80	0.35	2.26x	2.71x	16864	236096	401	588
	8	466.00	15.34	30.37x	1.03x	11.70	3.40	3.44x	1.08x	250083	4001328	863	4636
	9	*	*	*	*	107.14	32.91	3.26x	1.04x	1353196	24357528	1069	22785
bst	9	44.34	10.98	4.04x	0.70x	2.45	1.55	1.58x	0.77x	46960	845280	10846	77
	10	216.72	49.17	4.41x	0.46x	12.65	7.57	1.67x	0.30x	206395	4127900	22688	181
	11	*	*	*	*	68.31	49.86	1.37x	0.18x	915641	20144102	46731	431
deque	8	54.86	6.64	8.27x	1.50x	2.30	0.83	2.77x	1.54x	69281	1108496	576	1924
	9	550.57	57.72	9.54x	1.48x	22.53	7.58	2.97x	1.14x	623530	11223540	810	13856
	10	*	*	*	*	280.66	100.22	2.80x	1.18x	6235301	124706020	1100	113369
fibheap	6	3.13	1.52	2.06x	0.98x	0.22	0.16	1.34x	-	3003	21021	82	256
	7	24.88	3.13	7.94x	2.13x	1.17	0.67	1.75x	1.24x	36730	293840	130	2260
	8	398.13	28.31	14.06x	0.88x	16.89	9.80	1.72x	0.68x	544659	4901931	209	23454
filesystem	3	2.03	1.98	1.03x	0.97x	0.15	0.25	0.58x	-	58	6264	576	10
	4	17.13	3.70	4.63x	11.50x	1.20	0.72	1.67x	1.72x	1353	194832	1568	124
	5	*	*	*	*	37.84	30.01	1.26x	0.97x	64576	11623680	3940	2950
heaparray	8	104.50	4.18	24.99x	2.31x	1.24	0.89	1.39x	1.24x	97092	873828	258	3386
	9	2,718.12	26.96	100.81x	1.22x	12.02	9.00	1.33x	0.53x	804809	8048090	359	22418
	10	*	*	*	*	128.27	110.78	1.16x	0.58x	8722946	95952406	488	196623
queue	6	7.76	1.62	4.79x	2.64x	0.37	0.18	2.10x	-	10057	70399	45	1564
	7	104.41	6.37	16.38x	1.77x	3.90	0.94	4.14x	1.44x	147995	1183960	60	19732
	8	*	*	*	*	78.79	25.32	3.11x	1.00x	2578641	23207769	77	301399
stack	6	4.95	1.46	3.38x	1.01x	0.31	0.12	2.50x	-	9331	65317	42	1555
	7	59.44	5.08	11.71x	1.31x	2.93	0.68	4.27x	1.87x	137257	1098056	56	19608
	8	*	*	*	*	60.07	17.80	3.37x	1.31x	2396745	21570705	72	299593
treemap	10	579.50	7.61	76.14x	2.69x	3.29	1.25	2.63x	1.04x	13076	261520	3579	73
	11	1,754.34	19.42	90.34x	3.04x	10.80	3.26	3.32x	1.38x	35405	778910	5269	147
	12	*	*	*	*	32.81	9.14	3.59x	1.34x	96401	2313624	7774	297
ubstack	8	60.37	6.26	9.64x	1.57x	2.28	1.29	1.77x	1.30x	109681	987129	595	1659
	9	1,482.75	48.75	30.41x	1.48x	22.69	13.59	1.67x	0.66x	991189	9911890	931	10646
	10	*	*	*	*	271.56	175.61	1.55x	0.62x	9922641	109149051	1414	77191
gmean	-	-	-	**10.97x**	**1.51x**	-	-	**2.07x**	**0.97x**	-	-	-	**3040x**

Figure 10: Overall time and memory for exhaustive exploration in JPF and BOX and characteristics of the explored state spaces.

ΔBOX implements the components of ΔExecution as presented in Section 3. ΔBOX represents ΔState as a regular Java state that contains both ΔObjects and objects of the instrumented classes. Our instrumentation for ΔBOX (as well as for BOX) is partly manual at the time. ΔBOX uses instrumented code to perform the operations on the ΔState. Similarly to ΔJPF, ΔBOX merges states between iterations of the breadth-first exploration. ΔBOX employs the optimized state comparison as presented in Section 3.4.

5. EVALUATION

We present an experimental evaluation of ΔExecution. We first describe the ten basic subject programs used in the evaluation and then discuss the improvements that ΔExecution provides for an exhaustive exploration of these programs in both JPF and BOX. We then briefly mention an evaluation for a non-exhaustive exploration in JPF. We finally present the improvements that ΔExecution provides on a larger case study, an implementation of the AODV routing protocol [34].

We performed all experiments on a Pentium 4 3.4GHz workstation running under RedHat Enterprise Linux 4. We used Sun's JVM 1.5.0_07, limiting each run to 1.8GB of memory and 1 hour of elapsed time.

5.1 Basic subjects

We evaluated ΔExecution on ten subject programs taken from a variety of sources. All but one of these subjects have been previously used to evaluate testing and model-checking techniques. The following nine subjects are data structures: `binheap` is an implementation of priority queues using binomial heaps [42]; `bst` is our running example that implements a set using binary search trees [4, 45]; `deque` is our implementation of a double-ended queue using doubly-linked lists; `fibheap` is an implementation of priority queues using Fibonacci heaps [42]; `heaparray` is an array-based implementation of priority queues [4, 45]; `queue` is an object queue implemented using two stacks [14]; `stack` is an object stack [14]; `treemap` is an implementation of maps using red-black trees based on Java collection 1.4 [4, 42, 45]; `ubstack` is an array-based implementation of a stack bounded in size, storing integers without repetition [9, 33, 39, 44];The tenth subject is `filesystem`, which is based on the Daisy file-system code [10]. While the original code had seeded errors, we use a corrected version from another study [14]. The primary purpose of our evaluation is to compare the efficiency of ΔExecution and standard execution, so we use correct implementations of all basic subjects. (The AODV case study described in Section 5.4 uses code with errors that violate a safety property.)

For each subject, we wrote drivers for standard execution and for ΔExecution (similar to Figure 2). The drivers exercise the main mutator methods. For data structures, the drivers add and remove elements. For `filesystem`, the drivers create and remove directories, create and remove files, and write to and read from files.

5.2 Exhaustive exploration

Figure 10 shows the experimental results for exhaustive exploration. For each subject and several bounds (on the sequence length and parameter size, as in the driver shown in Figure 2), we tabulate the overall exploration time and peak memory usage with and without ΔExecution in both JPF and BOX, and the characteristics of the explored state spaces. The cells marked with '$*$' represent that the experiment either ran out of 1.8GB memory or exceeded the 1 hour time limit.

The columns labeled "std/delta" show the improvements that ΔExecution provides over standard execution. Note that the numbers are ratios and not percentages; for example, for `binheap` and $N = 7$, the ratio is 9.55x, which corresponds to about 90% improvement. For JPF, the speedup ranges from 0.68x (for `aodv` and $N = 6$) to 100.81x (for `heaparray` and $N = 9$), with the average of 10.97x, which is over an order of magnitude improvement.

experiment		standard execution time			ΔExecution time			
subject	N	exec.	comp.	backt.	exec.	comp.	backt.	merg.
binheap	7	18.21	0.54	6.65	0.57	0.59	1.10	0.39
	8	369.49	5.55	90.97	4.09	6.11	1.04	4.10
bst	9	20.77	3.75	19.81	2.30	5.42	2.01	1.26
	10	104.54	20.68	91.50	7.32	31.98	4.02	5.86
fibheap	6	1.24	0.07	1.82	0.21	0.13	1.06	0.12
	7	15.03	0.52	9.33	0.40	0.79	1.09	0.84
	8	257.91	8.21	132.01	3.89	11.43	1.34	11.64
treemap	10	567.16	2.65	9.69	1.39	4.34	1.48	0.41
	11	1,724.36	8.55	21.44	2.48	14.36	1.59	0.98

Figure 11: Time breakdown for JPF experiments.

experiment		standard execution time			ΔExecution time			
subject	N	exec.	comp.	backt.	exec.	comp.	backt.	merg.
binheap	7	0.23	0.34	0.14	0.09	0.15	0.00	0.06
	8	4.58	3.72	2.88	1.03	1.45	0.01	0.87
	9	21.64	68.57	15.27	4.73	21.95	0.00	6.20
bst	9	0.17	1.93	0.19	0.48	0.74	0.01	0.28
	10	0.60	10.72	0.97	1.83	3.83	0.01	1.85
	11	3.05	57.79	4.71	8.16	20.46	0.02	21.06
fibheap	6	0.06	0.08	0.04	0.04	0.05	0.00	0.02
	7	0.32	0.54	0.24	0.20	0.24	0.00	0.18
	8	4.77	7.80	3.90	2.79	3.78	0.00	3.15
treemap	10	0.20	2.86	0.20	0.34	0.78	0.01	0.07
	11	0.60	9.72	0.52	0.64	2.29	0.02	0.23
	12	1.51	29.69	1.26	1.47	6.91	0.02	0.67

Figure 12: Time breakdown for BOX experiments.

(The averages are geometric means over all the experiments.) For BOX, the speedup ranges from 0.58x (for `filesystem` and $N = 3$) to 4.27x (for `stack` and $N = 7$), with the average of 2.07x. Note that the ratio less than 1.00 means that ΔExecution ran slower (or required more memory) than standard execution, for example for `filesystem` and $N = 3$ in BOX. While this can happen for smaller bounds, ΔExecution consistently runs faster than standard execution for important cases with larger bounds.

ΔExecution provides these significant improvements because it exploits the overlap among executions in the state-space exploration. Figure 10 shows the information about the state spaces explored in the experiments. Note that the number of explored states is the same with and without ΔExecution. This is as expected: ΔExecution focuses on improving the exploration time and does not change the exploration itself. (We have used the difference in the number of states to debug our implementation of ΔExecution.) However, the numbers of executions with and without ΔExecution do differ, and the column labeled "std/delta" shows the ratio of the numbers of executions. The ratio ranges from 10x to 301399x. While this ratio effectively enables ΔExecution to provide the speedup, there is no strict correlation between the ratio and the speedup. The overall exploration time depends on several factors, including the number of execution paths, the number of splits, the cost to execute one path, the frequency of constants in ΔStates, and the sharing of execution prefixes.

We next discuss in more detail where state-space exploration spends time and where ΔExecution reduces the time. Each state-space exploration includes three components—(1) code execution, (2) state comparison, and (3) state backtracking—and ΔExecution additionally includes merging. Figures 11 and 12 show the breakdown of the overall exploration time on these four components for JPF and BOX. We show the numbers for only some of the experiments; the conclusions are the same for the other experiments.

In JPF, ΔExecution significantly reduces the time for code execution and state backtracking. For example, for `binheap` and $N = 7$, ΔExecution reduces the execution time from 18.21s to 0.57s and the backtracking time from 6.65s to 1.10s. These savings are big enough and make the times for merging and state comparison irrelevant. (ΔJPF does not even use the optimized state comparison for this exploration.) As mentioned earlier, JPF is a general-purpose model checker that stores and restores the entire Java states and thus has a high execution and backtracking overhead.

In BOX, ΔExecution sometimes results in a higher code execution time, yet has a smaller overall exploration time. The reason is that ΔExecution achieves significant savings in the state comparison using the optimized algorithm from Section 3.4. For example, for `bst` and $N = 11$, ΔExecution increases the execution time from 3.05s to 8.16s. However, it reduces the state comparison time from 57.79s to 20.46s, which more than makes up for the longer execution time. Note that the number of states and state comparisons is the same in both standard execution and ΔExecution, but the optimized state comparison is only possible for ΔExecution. Indeed, it is the execution on ΔStates that enables the simultaneous comparison of a set of states.

Figure 10 also provides a comparison of memory usage. Specifically, the columns labeled "mem. std/delta" show the ratio of peak memory usage for standard execution versus ΔExecution. Our setup uses the Sun's `jstat` monitoring tool to record the peak usage of garbage-collected heap in the JVM running an experiment. Although this particular measurement does not include the entire memory used by the JVM process, it does represent the most relevant amount used by a model checker. (The cells marked '-' represent experiments where the running time is so short that `jstat` does not provide accurate memory usage.)

For JPF, standard execution uses more memory than ΔExecution for most experiments and uses 1.51x more memory on average. However, ΔExecution occasionally uses more memory, for example for `bst`. For BOX, ΔExecution and standard execution on average use about the same amount of memory.

Many factors, already mentioned for exploration time, can influence the memory usage, but an important factor seems to be the number of constant ΔObjects. ΔExecution uses these objects to represent values that are the same across all states in a ΔState. There is a relatively strong positive correlation between the percentage of constant ΔObjects and the memory ratio for an experiment. For example, `bst` and $N = 11$ has a poor memory ratio, and the average percentage of constant objects in ΔStates is 33%, the lowest of all subjects. For `treemap` and $N = 12$, on the other hand, ΔExecution uses less memory than standard execution, and the average percentage of constant objects is 69%.

5.3 Non-exhaustive exploration

We have also evaluated ΔExecution for a different state-space exploration. Visser et al. [42] recently proposed and implemented in JPF several non-exhaustive explorations. Their results on four subject programs—`binheap`, `bst`, `fibheap`, and `treemap`—showed that *abstract matching* achieved the best structural code coverage. The main idea of abstract matching is to compare states based on their *shape abstraction*: two states that have the same shape are considered equivalent even if they have different values in nodes. In summary, our evaluation of ΔExecution for abstract matching on the same four subjects shows that ΔExecution improves the exploration time for various bounds between 0.93x and 21.81x, with an average of 3.37x. For lack of space, we cannot include the details of the evaluation, but they can be found in a technical report [11].

5.4 AODV case study

We also evaluated ΔExecution on a larger application, namely the implementation of the Ad-Hoc On-Demand Distance Vector

experiment		JPF time			JPF mem.	# states
subject	N	std	delta	std/delta	std/delta	
aodv	8	81.95	73.18	1.12x	0.52	14741
	9	296.33	226.39	1.31x	0.58	51488
	10	1,057.65	739.80	1.43x	0.51	173468

Figure 13: Exploration of AODV in JPF.

(AODV) routing protocol [34] in the J-Sim network simulator [23]. This application was previously used to evaluate a J-Sim model checker [38] and a technique for optimizing the execution of deterministic code blocks in JPF [13].

AODV is a routing protocol for ad-hoc wireless networks. Each of the nodes in the network contains a routing table that describes where a message should be delivered next, depending on the target. The safety property we check in this study expresses that all routes from a source to a destination should be free of cycles, i.e., not have the same node appear more than once in the route [38].

The implementation of AODV, including the J-Sim library classes that it depends on, consists of 43 classes with over 3500 non-blank, non-comment lines of code. We instrumented this code using the Eclipse plug-in that automates instrumentation for ΔExecution on JPF. The resulting instrumented code consisted of 143 classes with over 9500 lines of code. We did not try this case study in BOX since it currently requires much more manual work for instrumentation.

We used for this case study the driver previously developed for AODV [38]. Like the `bst` driver shown in Figure 2, the AODV driver invokes various methods that simulate protocol actions (sending messages, receiving messages, dropping messages etc.). Unlike the `bst` driver, the AODV driver (1) includes guards that ensure that an action is taken only if its preconditions are satisfied and (2) includes a procedure that checks whether the resulting protocol state satisfies the safety property described above. In our experiments, when a violation is encountered, the driver prunes that state/path but continues the exploration.

We ran experiments on three variations of the AODV implementation, each containing an error that leads to a violation of the safety property [38]. Figure 13 shows the results of experiments on one variation. Since the property was first violated in the ninth iteration for all three variations, the results for the other two variations were similar, and we do not present them here.

For AODV, ΔExecution improves the overall exploration time for up to 1.43x, while taking about twice as much peak memory as standard execution. We believe that it would be possible to improve these results by using a specialized *merging at the abstract state* level. Namely, the default merging in ΔExecution works at the concrete state level, and AODV operates on complex states, including for example routing tables. Even when two routing tables represent the same abstract state (say a set $\{\langle N_1, N_0\rangle, \langle N_2, N_0\rangle\}$), they could have different concrete states (say lists $[\langle N_1, N_0\rangle, \langle N_2, N_0\rangle]$ and $[\langle N_2, N_0\rangle, \langle N_1, N_0\rangle]$). While such differences of concrete states would disallow the default merging, it should be possible to merge those states because they represent the same abstract state.

6. RELATED WORK

Handling state is the central issue in explicit-state model checkers [21, 22, 27, 30]. For example, JPF [29] implements techniques such as efficient encoding of Java program state and symmetry reductions to help reduce the state-space size [27]. ΔExecution uses the same state comparison, based on Iosif's depth-first heap linearization [22]. However, ΔExecution leverages the fact that ΔStates can be explored simultaneously to produce a set of linearizations. Musuvathi and Dill proposed an algorithm for incremental state hashing based on a breadth-first heap linearization [30]. We plan to implement this algorithm in JPF and to use ΔExecution to optimize it.

Darga and Boyapati proposed glass-box model checking [14] for pruning search. They proposed a static analysis that can reduce state space without sacrificing coverage. Glass-box exploration represents the search space as a BDD and identifies, without execution, parts of the state space that would not lead to more coverage. However, glass-box exploration requires the definition of executable invariants in order to guarantee soundness. In contrast, ΔExecution does not require any additional annotation on the code.

Symbolic execution [25, 40, 45] is a special kind of execution that operates on symbolic values. In symbolic execution, the state includes symbolic variables (that can represent a set of concrete values) and a path-condition that encodes constraints on the symbolic variables. Symbolic execution has recently gained popularity with the availability of fast constraint solvers and has been applied to test-input generation of object-oriented programs [25, 40, 45]. In the general case, constraints generated during symbolic execution are undecidable. The recent techniques combining symbolic execution and random execution show good promise in handling some of these problems [6, 19, 37]. Conceptually, both symbolic execution and ΔExecution operate on a set of states. While symbolic execution can represent an unbounded number of states, ΔExecution uses an efficient representation for a bounded set of concrete states. The use of concrete states allows ΔExecution to overcome the problems that symbolic execution has. Moreover, we plan to investigate how to apply ΔExecution to speed up symbolic execution by sharing symbolic states.

Shape analysis [26, 36, 46] is a static program analysis that verifies programs that manipulate dynamically allocated data structures. Shape analysis uses abstraction to represent infinite sets of concrete heaps and performs operations on these sets, including operations similar to splitting and merging in ΔExecution. Shape analysis computes overapproximations of the reachable sets of states and loses precision to obtain tractability. In contrast, ΔExecution operates precisely on sets of concrete states but can explore only bounded executions.

Offutt et al. [32] proposed DDR, a technique for test-input generation where the values of variables are ranges of concrete values. DDR uses symbolic execution (on ranges) to generate inputs. Intuitively, DDR can be efficiently implemented as the ranges are split (using a technique called *domain splitting*) when constraints are added to the system. DDR requires inputs to be given as ranges, implements a lossy abstraction (to reduce the size of the state space in favor of more efficient decision procedures), and does not support object graphs. ΔExecution focuses on object graphs and does not require inputs to be ranges, but the use of ranges as a special representation in ΔStates could likely improve ΔExecution even more, and we plan to investigate this in the future.

In the introduction, we have discussed the relationship between symbolic model checking [7, 24] and ΔExecution. ΔExecution is inspired by symbolic model checking and conceptually performs the same exploration but handles states that involve heaps. BDDs are typically used as an implementation tool for symbolic model checking. Predicate abstraction in model checking [2, 3] reduces the checking of general programs into boolean programs that are efficiently handled by BDDs. While predicate abstraction has shown great results in many applications, it does not handle well complex data structures and heaps. BDDs have been also used for efficient program analysis [28, 43] to represent analysis information as sets and relations. These techniques employ either data [28] or control abstraction [43] to reduce the domains of problems and make them

tractable. It remains to investigate if it is possible to leverage on a symbolic representation, such as BDDs, to represent sets of concrete heaps to efficiently execute programs in ΔExecution mode.

We previously proposed a technique, called Mixed Execution, for speeding up straightline execution in JPF [13]. Mixed Execution considers only one state and uses an existing JPF mechanism to execute code parts outside of the JPF backtracked state, improving the exploration time up to 37%. ΔExecution considers multiple states and improves the exploration time by an order of magnitude.

7. CONCLUSIONS

We have presented ΔExecution, a novel technique that significantly speeds up state-space exploration of object-oriented programs. State-space exploration is an important component of model checking and automated test generation. ΔExecution executes the program simultaneously on a set of standard states, sharing the common parts across the executions and separately executing only the "deltas" where the executions differ. The key to efficiency of ΔExecution is ΔState, a representation of a set of states that permits efficient operations on the set. The experiments on two model checkers show that ΔExecution can reduce the time for state-space exploration from two times to over an order of magnitude.

In the future, we plan to apply the ideas from ΔExecution in more domains. First, we plan to manually transform some important algorithms to work in the "delta mode", as we did for the optimized comparison of states. For instance, we plan to transform merging of ΔStates, which would further improve the results of ΔExecution. Second, we plan to evaluate automatic ΔExecution outside of state-space exploration. For example, in regression testing the old and the new versions of a program can be run in the "delta mode", which would allow a detailed comparison of the states from two versions. We believe that ΔExecution can also provide significant benefits in these new domains.

Acknowledgments We thank Corina Pasareanu and Willem Visser for helping us with JPF, Chandra Boyapati and Paul Darga for providing us with the subjects from their study [14], Ahmed Sobeih for helping us with the AODV case study, and Brett Daniel, Kely Garcia, and Traian Serbanuta for their comments on an earlier draft of this paper. We also thank Ryan Lefever, William Sanders, Joe Tucek, Yuanyuan Zhou, and Craig Zilles—our collaborators on the larger Delta Execution project [47]—for their comments on this work. This work was partially supported by NSF CNS-0615372 grant. We also acknowledge support from Microsoft Research.

8. REFERENCES

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV*, 2004.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, 2000.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, 2006.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, 2000.

[9] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software Practice and Experience*, 34, 2004.

[10] Daisy File System. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software.

[11] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. Technical Report UIUCDCS-R-2007-2844, University of Illinois, Urbana, IL, April 2007.

[12] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE*, 2006.

[13] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *ICFEM*, 2006.

[14] P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *OOPSLA*, 2006.

[15] Eclipse foundation. `http://www.eclipse.org/`.

[16] Foundations of Software Engineering at Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/AsmL`.

[17] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 2001.

[18] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL*, 1997.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[20] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), 1997.

[22] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, 2001.

[23] J-Sim. `http://www.j-sim.org/`.

[24] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, 1990.

[25] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, April 2003.

[26] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *POPL*, Jan. 2002.

[27] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN '01*, 2001.

[28] O. Lhotak and L. Hendren. Jedd: A BDD-based relational extension of Java. In *PLDI*, 2004.

[29] P. C. Mehlitz, W. Visser, and J. Penix. The JPF runtime verification system. `http://javapathfinder.sourceforge.net/JPF.pdf`.

[30] M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In *SPIN*, 2005.

[31] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*, 2002.

[32] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software Practice and Experience*, 29(2), 1999.

[33] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, 2005.

[34] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Proc. IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100. IEEE Computer Society Press, 1999.

[35] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *ESEC/FSE*, 2003.

[36] R. Rugina. Quantitative shape analysis. In *11th International Static Analysis Symposium (SAS'04)*, Aug. 2004.

[37] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, September 2005.

[38] A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In *ICFEM*, volume 3785 of *LNCS*, 2005.

[39] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, 2002.

[40] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, 2004.

[41] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red-black trees using abstraction. In *ASE*, 2005.

[42] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *ISSTA*, 2006.

[43] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[44] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE*, 2004.

[45] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, 2005.

[46] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.

[47] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d'Amorim, S. Lauterburg, R. M. Lefever, and J. Tucek. Delta execution for software reliability. To appear in HotDep, 2007.