

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

Marcelo d’Amorim, Steven Lauterburg, and Darko Marinov

Abstract— We present Delta Execution, a technique that speeds up state-space exploration of object-oriented programs. 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. We exploit the fact that many execution paths in state-space exploration partially overlap. Delta Execution simultaneously operates on several states/heaps and shares the common parts across the executions, separately executing only the “deltas” where the executions differ.

We implemented Delta Execution in two model checkers: JPF, a popular general-purpose model checker for Java programs, and BOX, a specialized model checker that we developed for efficient exploration of sequential Java programs. The results for bounded-exhaustive exploration of ten basic subject programs and one larger case study show that Delta Execution reduces exploration time from 1.06x to 126.80x (with median 5.60x) in JPF and from 0.58x to 4.16x (with median 2.23x) in BOX. The results for a non-exhaustive exploration in JPF show that Delta Execution reduces exploration time from 0.92x to 6.28x (with median 4.52x).

Index Terms— Software/program verification, model checking, testing and debugging, performance, delta execution

I. INTRODUCTION

SOFTWARE testing and model checking are important approaches for improving software reliability. A core technique for model checking is *state-space exploration* [11]: 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 [16], [18], [26], [46], [48], [49]. 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 [16], [46], [48].

A key issue in state-space exploration is manipulating the program state: saving the state at non-deterministic choice 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, including Bandera [12], BogorVM [37], CHESS [32], CMC [31], JCAT [19], JNuke [4], JPF [44], SpecExplorer [43],

and Zing [3]. 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 operates *simultaneously on several states/heaps*. While such execution may be useful for several software reliability tasks—including patch validation, administrative configuration validation, testing, model checking, or replica-based fault detection and recovery [51]—this paper focuses on state-space exploration of programs with heaps. Δ Execution exploits the fact that many execution paths in state-space exploration partially overlap. Δ Execution speeds up the state-space exploration by sharing the common parts across the executions and separately executing only the “deltas” where the executions differ. Central to Δ Execution is an *efficient representation and manipulation of sets of states*. Δ Execution is thus related to shape analysis [27], [38], [50], a static program analysis that checks heap properties and operates on sets of states. However, shape analysis operates on abstract states, while Δ Execution operates on concrete states.

Δ Execution was inspired by symbolic model checking (SMC) [11], [25]. 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) [8], data structures 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 [29], [47], the heap operations—including field reads and writes, dynamic object allocation, garbage collection, and comparisons based on heap symmetry [7], [11], [23], [28], [30]—do not translate directly into efficient BDD operations.

Δ Execution operates on a Δ State, a novel representation for sets of states that include heaps. We describe efficient operations for manipulating Δ States, which enable Δ Execution to execute programs faster than standard execution. These operations also enable Δ Execution to speed up state comparison and backtracking, two important and costly parts of state-space exploration. The key to these speed-ups in Δ Execution is that various values can be constant across all states in a given set, and an operation can execute at once on a large number of states rather than executing on each of them individually.

We implemented Δ Execution in two model checkers: JPF (from Java PathFinder) and BOX (from Bounded Object Explorer). JPF is a popular, general-purpose model checker for Java programs [1], [28], [44]. BOX, in contrast, is a specialized model checker that we developed for efficient exploration of sequential Java programs. The two implementations allowed us

Marcelo is with the Universidade Federal de Pernambuco, Centro de Informática, Caixa Postal 7851, CEP 50732-970, Recife, PE, Brazil. E-mail: damorim@cin.ufpe.br. Steven and Darko are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 61801-2302, USA. E-mail: {slauter2, marinov}@cs.uiuc.edu.

```

class BST {
  Node root;
  int size;

1: void add(int info) {
2:   if (root == null)
3:     root = new Node(info);
4:   else
5:     for (Node temp = root; true; )
6:       if (temp.info < info) {
7:         if (temp.right == null) {
8:           temp.right = new Node(info);
9:           break;
10:        } else temp = temp.right;
11:       } else if (temp.info > info) {
12:         if (temp.left == null) {
13:           temp.left = new Node(info);
14:           break;
15:         } else temp = temp.left;
16:       } else return; // no duplicates
17:   size++;
18: }

  void remove(int info) { ... }
}

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

```

Fig. 1. Binary search tree implementation of a set.

to evaluate Δ Execution on model checkers that follow different design principles. While we found out that Δ Execution reduces the overall exploration time in both model checkers, the reduction is due to different reasons as discussed in Section V-A.

We evaluated Δ Execution using two types of exploration. The first type is *bounded-exhaustive exploration*, which explores all states that can result from sequences of method calls up to some bound on the length of the sequence and input values. The second type uses *abstract matching*, a recently proposed non-exhaustive state-space exploration [46] that matches states based on their shapes. For the bounded-exhaustive exploration, we evaluated Δ Execution on ten simple subject programs and one larger case study, AODV [35]. For simple subject programs, Δ Execution reduces exploration time from 1.06x to 126.80x (with median 5.60x) in JPF and from 0.58x to 4.16x (with median 2.23x) in BOX. While the main goal of Δ Execution is to reduce time, it also reduces, on average, peak memory requirement from 0.46x to 11.50x (with median 1.48x) in JPF and from 0.18x to 2.71x (with median 1.18x) in BOX. (Note that a number below 1.00x represents that Δ Execution increases time or memory usage.) For AODV, Δ Execution reduces exploration time from 0.88x to 2.04x (with median 1.72x) in JPF. For the non-exhaustive exploration, Δ Execution reduces exploration time from 0.92x to 6.28x (with median 4.52x) in JPF on the four of the ten simple subject programs used previously with abstract matching [46]. The reduction is smaller for the non-exhaustive exploration than for exhaustive exploration because abstract matching reduces the total number of states that the model checker explores.

The rest of this paper is organized as follows. Section II shows an example that illustrates the key aspects of Δ Execution and how it speeds up standard execution. Section III presents in detail the algorithms for Δ Execution. Section IV describes our two implementations. Section V presents an evaluation of Δ Execution. Section VI reviews related work, and Section VII concludes.

```

// N bounds sequence length and parameter values
exploreStandard(N)
  Next = {sinit}
  Visited = {linearize(sinit)}
  for i = 1 to N do // iterations
    Current = Next; Next = {}
    while (|Current| > 0) do
      sroot = choose a state from Current
      foreach method m in methods do
        for v = 1 to N do
          snext = execute m(v) on sroot
          l = linearize(snext)
          if (l ∉ Visited) then
            Visited = Visited ∪ {l}
            Next = Next ∪ {snext}

```

Fig. 2. Breadth-first exploration using standard execution.

II. EXAMPLE

We present an example that illustrates what Δ Execution does and how it speeds up the state-space exploration compared to standard execution that operates on a single state at a time. 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)`.

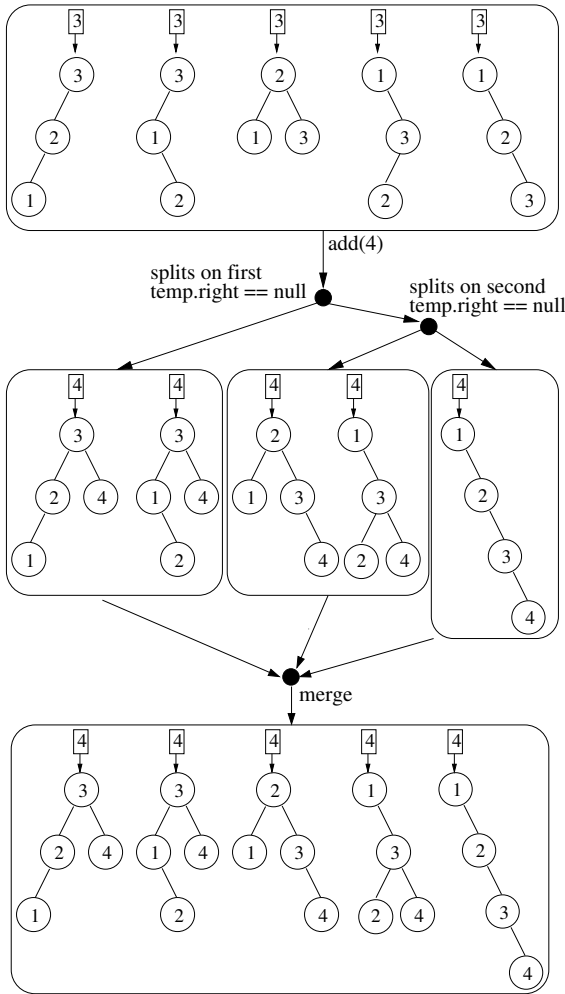
The goal of state-space exploration is to explore different sequences of method calls. A common scenario of exploration is to exhaustively explore all sequences of method calls, up to some bound [18], [46], [49]. Such exploration does not actually enumerate all sequences but instead uses state comparison to prune sequences that exercise the same states [46], [49]. Another scenario may be to generate those sequences that result in assertion violations.

A. Standard exploration

Figure 2 shows pseudocode that systematically generates sequences of method calls to explore different states of a subject. This exploration operates using standard execution, so we call it a *standard exploration*. Starting with an initial state s_{init} for the subject (in our example, an empty tree), it exhaustively explores sequences (up to length N) of the subject's methods (in our example, add and remove), with values between 1 and N .

Following the execution of a subject method, a linearization is computed for the resulting state s_{next} . Linearization translates an object graph into an integer array representing the graph in a canonical form; it is a common technique used to facilitate efficient comparison of state that include heaps [11], [23], [28], [30]. If the linearization is not in the set `Visited`, it is added and also added to the set `Next` for exploration during the next iteration. Otherwise, any sequence that results in a state that has already been visited is pruned from further exploration.

Note that state comparison is performed only at the method boundaries (not during method execution). This naturally partitions an execution path into subpaths, each covering execution of one method invocation. As in other related studies [16], [46], [49], we consider a breadth-first exploration of the state space. A bounded depth-first exploration could miss parts of the state space since state comparison could prune a shorter sequence (that results in some state) because of a longer sequence (that results in the same state). For example, a depth-first exploration limited to three method calls could explore the sequence `BST t = new`

Fig. 3. Executions of `add(4)` on a Δ State.

`BST(); t.add(1); t.add(2); t.remove(1)` before the sequence `BST t = new BST(); t.add(2)`. Since both sequences result in the same tree state, the latter would be pruned and would miss, for instance, the tree `BST t = new BST(); t.add(2); t.add(3)`.

B. Overlapping execution paths

Figure 3 shows several states that arise during a state-space exploration of the binary search tree subject for $N = 4$. The five trees shown at the top of the figure are all (non-equivalent) trees of size 3 with values between 1 and 3. When it comes time to execute `add(4)` on these five trees, standard exploration separately executes `add(4)` on each pre-state, resulting in the five post-states shown at the bottom of the figure. We use the term *individual state* to emphasize that exploration using standard execution operates on a single state at a time.

We next describe how various executions *within* a method can have overlapping paths/traces. Each path is a trace of values for the program counter. We focus on sequential programs, so there is no thread interleaving, and the branching decisions determine the trace. For example, execution of `add(4)` on the balanced tree shown in the middle results in the following trace (for program counter values from Figure 1): 1, 2, 4, 5, 6, 7, 10, 5, 6, 7, 8, 9, 17, 18. We say *a state follows a path* iff the execution starting

```
// N bounds sequence length and parameter values
exploreDelta(N)
Next = {s_init};
Visited = {linearize(s_init)};
for i = 1 to N do // iterations
  ΔState σ_root = merge(Next); Next = {};
  foreach method m in methods do
    for v = 1 to N do
      {σ_1, ..., σ_k} = execute_Δ m(v) on σ_root; // splits
      foreach σ ∈ {σ_1, ..., σ_k} do
        foreach l ∈ linearize_Δ(σ) do
          if (l ∉ Visited) then
            Visited = Visited ∪ {l};
            Next = Next ∪ {state for l};
```

Fig. 4. Breadth-first exploration using Δ Execution.

with that state results in that path. For instance, the balanced tree follows the aforementioned path.

It is important to note that several states can follow the same path, i.e., each individual execution makes the same branching decisions. For example, consider the two executions of `add(4)` on the balanced tree in the middle and the tree to its right. Both of these executions follow the same aforementioned path (as they add a new node with value 4 to the right of the root's right child). Δ Execution exploits this similarity to speed up state-space exploration. While this example shows the case when two executions have identical paths, Δ Execution can also exploit similarities among paths even when they are not identical in their entirety.

C. Delta exploration

Figure 4 shows pseudocode for a state-space exploration using Δ Execution. We refer to this type of exploration as *delta exploration*. Delta exploration is similar to standard exploration: both prune the exploration based on resulting states and both use breadth-first exploration. However, delta exploration differs from the standard exploration in four important ways.

1) *ΔState*: Δ Execution conceptually operates on multiple individual states at the same time. More precisely, Δ Execution operates on a single Δ State that represents several standard states, each corresponding to one of the individual states found in a standard execution. The type of the root object (σ_{root}) in delta exploration is Δ State. While standard execution invokes `add(4)` separately against each standard state, Δ Execution invokes `add(4)` on one Δ State, effectively invoking it simultaneously against a set of standard states. The top of Figure 3 represents one set consisting of the five pre-states. Section III-B describes how to efficiently represent a Δ State.

2) *Splitting*: When a method is executed on a Δ State, the result can be more than one Δ State: $\sigma_1, \dots, \sigma_k$. Each of these resulting k Δ States represents the subset of individual states from the original Δ State that *follow the same execution path through the method*, i.e., make the same branching decisions. The total number of individual states in this set of Δ States is equal to the number of individual states in the original Δ State the method is executed on, i.e., $\sum_{i=1}^k |\sigma_i| = |\sigma_{root}|$.

During method execution, Δ Execution occasionally needs to *split* the Δ State. Consider, for example, the executions illustrated in Figure 3. For `add(4)`, the five pre-states at the top follow the same execution path until the first check of `temp.right == null`. At that point, Δ Execution 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 splitting enforces the invariant that all states in a set follow the same execution path through the method.

Each split introduces a non-deterministic choice point in the execution. For `add(4)`, the 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 execution 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. Note that Δ Execution produces the same (number of) states as standard execution (five in our example) but may result in fewer executions (these five states require only three different execution paths, i.e., $k = 3$, whereas the `while` loop from Figure 2 would be executed five times).

3) *Merging*: Since Δ Execution operates on sets of states (i.e., a Δ State), a delta exploration needs to periodically combine multiple individual states (or multiple small Δ States) together into a single Δ State. Figure 4 shows that states are combined at the beginning of each iteration, using the `merge` operation. Effectively, this operation combines all distinct states reachable with the method sequences of length i into one Δ State that the iteration $i+1$ will explore.

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. Δ Execution can, in principle, perform merging on any sets of states at any program point. For example, Δ Execution could merge all three sets of states from Figure 3 when they reach `size++`. However, as illustrated in Figure 4 our current implementation of Δ Execution considers only the program points that are method boundaries. While *splitting* occurs *during* method execution, *merging* only occurs *between* method executions. Section III-F describes how to efficiently merge states.

4) *Δ Linearization*: Delta exploration uses the `linearize Δ` operation to linearize the individual states in a Δ State σ all at once rather than one by one. This operation returns a set of linearizations and can do this faster than linearizing each state individually. Section III-E describes this optimization.

D. Performance

We next discuss how the performance of Δ Execution and standard execution compare. In our running example, Δ Execution 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 experimental results from Section V show that Δ Execution is faster than standard execution for a number of subject programs and values for the exploration bound N . For example, for the binary search tree example and $N = 10$, Δ Execution speeds up JPF 7.11x (while taking about two times more memory than standard execution) and speeds up our model checker BOX 1.67x (while taking about three times more memory than standard execution).

E. Reasons for speedup

We next discuss why Δ Execution can speed up the three major operations in state-space exploration: (i) (straightline) execution, which performs a deterministic step on the subject program (`execute` in our algorithms); (ii) backtracking, which explores *all* program paths created with non-deterministic choices (effectively corresponds to choices of methods m and values v in our algorithms); and (iii) (state) comparison, which prunes some of these paths based, for example, on isomorphism of visited states [7], [23] (`linearize` and `lookup` into `Visited` in our algorithms).

Δ Execution can reduce *execution* time because *some values are constant across all states in a state set*. For example, executing `size++` on all trees shown in Figure 3 takes constant time (instead of time linear to the number of states) because all trees have the same size. We measured the ratio of the number of accesses to constants over the total number of value accesses for binary search tree, and for $N = 10$, it is about 25%. However, the time savings depends not only on the ratio of accesses to constants but also on the number of states that a constant represents: if a set has n states, then the execution saves $n - 1$ operations when it operates on a constant and does not need to iterate over all n states. Using the number of states to adjust the ratio of accesses to constants shows that about 35% of accesses are to constants for binary search tree and $N = 10$. (More details on constants are available in d'Amorim's PhD thesis [14].)

Δ Execution can reduce the cost of *backtracking* as it reduces the number of executions. For example, for states from Figure 3, Δ Execution backtracks 2 times (for 3 executions), while standard execution backtracks 4 times (for 5 executions). Δ Execution introduces a backtrack point only when it needs to split an execution path because not all states in the current set evaluate a branching condition to the same value. Effectively, the index k in $\sigma_1, \dots, \sigma_k$ from Figure 4 is 3 in this example, while the size of `Current` from Figure 2 starts out as 5.

Δ Execution also enables optimized state *comparison* because it is possible to compute a set of state linearizations on a set of states simultaneously instead of one-by-one. In practice, this enables the linearization algorithm to internally share the prefixes of the linearization. Section III-E presents more details.

The trade-off between Δ Execution and standard execution can be summarized as follows: Δ Execution performs fewer executions (avoiding separate execution of the same path shared by multiple states) than standard execution, but each execution in Δ Execution (that operates on a set of standard states) is more expensive than in standard execution (that operates on one standard state). Whether Δ Execution 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 sharing of execution prefixes, and the ratio of constants. In particular, the presence of constants (i.e., values that are the same across a set of states) is essential for efficient operations under Δ Execution.

III. TECHNIQUE

The main idea of Δ Execution is to execute a program simultaneously on a set of standard states. Figure 4 presents a high-level algorithm for Δ Execution. We first discuss some key properties of the algorithm. We then present more details of the algorithm. The central part of Δ Execution is Δ State, a representation for a set of individual states. We describe two main operations on

```

class BST {
    DeltaNode root;
    DeltaInt size;
}
class Node {
    DeltaNode left, right;
    DeltaInt info;
}
class DeltaNode {
    // maps each state index to a Node object
    Node[] values; // conceptually
}
class DeltaInt {
    // maps each state index to an integer value
    int[] values; // conceptually
}

```

Fig. 5. Field declarations for instrumented BST and Node classes, the new DeltaNode class and the DeltaInt library class.

Δ States: *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 Δ Execution and how Δ Execution facilitates an optimized comparison of states.

A. High-level properties

Recall figures 2 and 4 which show the pseudocode for the standard exploration and delta exploration, respectively. The goal of Δ Execution is to speed up standard exploration; Δ Execution does not attempt to reduce the size of the state space but only to reduce the exploration time. More precisely, *given the same value for the bound N (and the same methods), $exploreStandard$ and $exploreDelta$ produce the same Visited set at the end of the procedure.*

Moreover, *Visited* not only contains the same values at the end of the two procedures but also contains the same values at the beginning of the main loop, i.e., for any iteration i from 1 to N , *Visited* in $exploreStandard$ has the same values as *Visited* in $exploreDelta$. This can be shown by induction, and it implies that *Visited* is equal at the end of the procedures. Similarly, *Next* is equal in both procedures for any corresponding iteration i from 1 to N .

B. Δ State

Δ Execution represents a set of individual standard states as a single Δ State. Each Δ State encodes all the information from the original individual states. A Δ State includes Δ Objects that can store multiple values (either references or primitives) that exist across the multiple individual states represented by a Δ State.

Figure 5 shows the classes used to represent Δ States 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 Δ State and are explained later in the text.) Each object of the class Δ Node stores a collection of references to Δ Node objects, and each object of the class Δ Int stores a collection of primitive integer values. The BST and Node objects are changed such that they have fields that are Δ Objects.

Figure 6 shows the Δ State that represents the set of five pre-states from Figure 3. Each Δ State consists of layers of “regular” objects and Δ Objects. For this example, the circles represent Node objects, the single rectangle represents a BST object, the array-like structures represent either Δ Node objects or

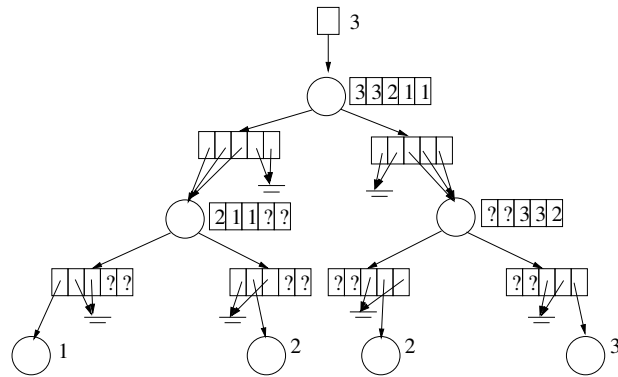


Fig. 6. Δ State for the five pre-states from Figure 3.

Δ Int objects, the stand-alone integers represent Δ Int objects that are constants, and the stand-alone arrow leaving the top-most rectangle represents a Δ Node object that is a constant. In this Δ State, 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 Δ State 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 Δ State are “don’t cares” (labeled with “?”) because the corresponding object is not reachable for that state index. For example, the first Node object 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 Δ Object 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*, i.e., the states where the value is not “don’t care”. For example, the *size* field of the BST object has value 3 for all five states, and the *info* field for each tree leaf in Figure 6 has a constant value (since there is only one relevant state).

Our implementation of Δ States uses an *optimized representation for constants*. When a field value is constant across all relevant states, that field is represented in the Δ State as a single value, as opposed to a sequence of values corresponding to different states. This optimization is applied at merge time when initially constructing a Δ State, and is important both for reducing the memory requirements of Δ States and for improving the efficiency of operations on Δ States.

C. Splitting

Δ Execution operates on a Δ State that represents a set of standard states. Δ Execution 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 Δ Execution needs to explore both outcomes. (Note that not all branch control points require a split since it is possible that all states can evaluate to the same branch outcome.)

One challenge in Δ Execution is to efficiently split Δ States. Our solution is to introduce a *statemask* that identifies the currently *active states* within a Δ State. Each statemask is a set of state indexes. At the beginning of an execution, Δ Execution initializes

the statemask to the set of all state indexes. For example, the execution of `add(4)` for the Δ State from Figure 6 starts with the statemask being $\{0, 1, 2, 3, 4\}$.

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

After Δ Execution finishes the execution path for some (sub)set of states, it *backtracks* to some unexplored split point to explore the other path using the statemask saved at the split point. Backtracking changes the statemask but restores the Δ State to exactly what it was at the split point. A model checker can implement backtracking in several ways. JPF, for instance, stores and restores state, while BOX uses program re-execution. Section IV elaborates this discussion.

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

Appropriate use of a statemask can facilitate optimizations on the Δ State. Consider, for example, a Δ Object 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 statemask occurring during the execution. For instance, in our running example, the value of `root.right` becomes the constant `null` when the statemask is $\{0, 1\}$. Additionally, the statemask allows the use of *sparse representations* for Δ Objects: instead of using an array to map all possible state indexes into values, a sparse Δ Object can use representations that *map only the active state indexes into values*, thereby reducing the memory requirement.

D. Program execution model

We next discuss how Δ Execution executes program operations. The key is to execute each operation simultaneously on a set of values. Δ Execution uses a non-standard program execution that manipulates a Δ State that represents a set of standard states. Such non-standard execution can be implemented in two ways: (i) instrumenting the code such that the regular execution of the instrumented code corresponds to the non-standard execution [26], [45], [49] or (ii) changing the execution engine such that it interprets the operations in the non-standard semantics [2], [16]. Our current implementation uses instrumentation: the subject code is pre-processed to support Δ Execution.

We use parts of the instrumentation to describe the semantics of Δ Execution.

1) *Classes*: The instrumentation changes the original program classes and generates new classes for Δ Objects. Figure 1 shows a part of the original code for the binary search tree example. Figures 7, 8, and 9 show the key parts of the instrumented code for this example. Figure 7 shows the instrumented version of the original BST and Node classes. Figure 8 shows the new

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

    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)));
    }

    void remove(DeltaInt info) { ... }
}

class Node {
    DeltaNode left = DeltaNode.NULL;
    DeltaNode right = DeltaNode.NULL;
    DeltaInt info = DeltaInt._new(0);
    Node(DeltaInt info) { this.info = info; }
}
```

Fig. 7. Instrumented BST and Node classes.

class `DeltaNode` that stores and manipulates the multiple `Node` references that can exist across the multiple states in a Δ State. Figure 9 shows the class `DeltaInt` that stores and manipulates multiple `int` values; this class is a part of the Δ Execution library and is not generated anew for each program.

It is important to note that Δ Objects 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 Δ Objects, for example directly assigning one `DeltaInt` object to another (e.g., `int x = y` simply becomes `DeltaInt x = y`). Our implementation internally mutates Δ Objects to achieve higher performance, in particular when values become constant across active states. The mutation handles the situations that involve shared Δ Objects and require a “copy-on-write” cloning.

2) *Types*: The instrumentation changes all types in the original program to their delta versions. Comparing figures 1 and 7, notice that the occurrences of `Node` and `int` have been replaced with the new `DeltaNode` class (from Figure 8) and the `DeltaInt` class (from Figure 9), respectively. The instrumentation also appropriately changes all definitions and uses of fields, variables, and method parameters to use Δ Objects.

3) *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 Δ Execution, `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

```

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

  DeltaNode(int size) { values = new Node[size]; }
  DeltaNode(Node n) { values = new Node[] { n }; }
  static DeltaNode _new(DeltaInt info) {
    return new DeltaNode(new Node(info));
  }
  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 = choose true or false /** split **/
    StateMask.setStateMask(result ? trueMask : falseMask);
    return result;
  }
  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;
  }
  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];
    }
  }
  DeltaNode get_right() { ... }
  void set_right(DeltaNode arg) { ... }
  DeltaInt get_info() { ... }
  void set_info(DeltaInt arg) { ... }
}

```

Fig. 8. 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;
  }
  ...
}

```

Fig. 9. Part of DeltaInt library class.

states are active in the statemask. In general, this can result in an execution split when some objects in `temp` are null.

4) *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 `DeltaObjects` that represent sets of values. More precisely, the methods in `DeltaObjects` do not need to operate on all values but only on those values that correspond

to the active state indexes as indicated by the statemask.

Consider integer addition as an example of arithmetic operations. In standard execution, addition takes two integer values and creates a single value. In Δ Execution, it takes two `DeltaInt` objects and creates a new `DeltaInt` object. The `add` method in `DeltaInt` (from Figure 9) shows how Δ Execution 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 `foreach` loop and state indexing).

Consider reference equality as an example of relational operations. The method `eq` in `DeltaNode` (from Figure 8) 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 comparison differs across the states. If so, `eq` introduces a non-deterministic choice that returns a boolean `true` or `false`. In all cases, `eq` appropriately sets the statemask.

5) *Method calls*: The instrumentation replaces a standard method call with a method call whose receiver is a `DeltaObject`, which allows making the call on several objects at once. Note that each call also introduces a semantic branch point due to dynamic dispatch (i.e., different objects may have different dynamic types) and can result in an execution split.

E. Optimized state comparison

Heap symmetry [11], [23], [28], [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 state, 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) [7], [23], [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 iff their linearizations are equal.

Δ Execution exploits the fact that different heaps in a Δ State can share prefixes of linearization. Instead of computing linearizations separately for each state in a set of states, Δ Execution *simultaneously computes a set of linearizations* for a Δ State. 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.

Figure 10 shows the pseudocode for an optimized algorithm that simultaneously linearizes all states from a Δ State. For simplicity of presentation, the algorithm assumes that the heap contains only reference fields and of only one class. We point out that our actual implementation handles general heaps with objects of different classes, primitive fields, and arrays. More details about the general case, as well as how to develop this algorithm from a basic one that linearizes one state at a time, are available elsewhere [14], [15].

The top-level function, $linearize_{\Delta}$, takes as input an object o , which represents the root of a Δ State, and a statemask sm , which represents the active states in that Δ State. It computes $linSet$, a set of linearizations. Each linearization is a sequence of integers l that represents one or more states marked by the statemask tm . This function uses the helper functions $linObject$ and $linFields$ described below. We first explain these functions for the simple

```

linearize $\Delta$ (Object o, StateMask sm)
  stack = empty stack
  ( $\_$ , l, tm) = linObject(o, empty Map, sm)
  linSet = {l} // all states in tm have sequence l
  while (|stack| > 0) do
    (o, f, ids, lpre, nm) = pop from stack
    ( $\_$ , l, tm) = linFields(o, f, ids, lpre, nm)
    linSet = linSet  $\cup$  {l}
  od
  return linSet;

// returns a triple of a Map, Lin, and StateMask
linObject(Object o, Map ids, StateMask sm)
  if (o = null) then return (ids, [NULL], sm)
  if (o  $\in$  ids) then return (ids, [get(ids, o)], sm)
  id = |ids|
  return linFields(o, 0, put(ids, o, id), [id], sm)

// returns a triple of Map, Lin, and StateMask
linFields(Object o, int f, Map ids, Lin l, StateMask sm)
  if (f < numberOfFields(o)) then
    (fo, em, nm) = split(getField(o, f), sm)
    if (|nm| > 0) then
      push(o, f, ids, l, nm) onto stack
      (m, lpost, om) = linObject(fo, ids, em)
      return linFields(o, f+1, m, append(l, lpost), om)
    else return (ids, l, sm)

```

Fig. 10. Optimized linearization of states in a Δ State.

case with one state, effectively considering that sm is a singleton and ignoring the variable $stack$. We then explain the general case.

The function $linObject$ takes a root object o and produces a sequence of integers that represent the linearization for the state reachable from o . When o is null, $linObject$ returns a one-element sequence with the value that represents null. When o is a reference to a previously linearized object, $linObject$ returns a one-element sequence with the integer id that the map ids associates with that object. (The map ids facilitates handling of object aliasing [23].) 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 function $linFields$ linearizes the fields of a given object, starting from the field at offset f . (Each field has an offset that ranges from 0 to one less than the number of fields in that object.) In its simplest form, this function first linearizes the state from the object fo that the field f points to (the result is called $lpost$) and then recursively linearizes the remaining fields, from $f+1$, after appending $lpost$ to the resulting sequence l . (This is effectively a tail-recursive function where l serves as the accumulator for the result.)

When there is only one state in sm , there are no splits in the execution. However, the linearizations depend on the value of the field, $getField(o, f)$, which may differ for different states. When there is such a difference, it is necessary to split the statemask into two, continue to explore one of them, and then backtrack to explore the other. This is the only source of non-determinism in the linearization. (Note that $linFields$ and $linObject$ manipulate functional objects `Map` and `Lin`, which facilitates backtracking of the state.) Effectively, all three functions maintain the invariant that the linearization prefix l that they compute up to any point is the same for all states in the statemask sm .

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 offset, the map for object identifiers, the current linearization sequence, and the statemask. While $stack$ is mutable, the other structures are immutable, which makes it easy to restore the state. The `while`

loop in $linearize\Delta$ visits each pending backtracking point until it finishes computing all linearizations.

The function $split$ in $linFields$ takes as input a Δ Object $do = getField(o, f)$ and a statemask sm . It returns a standard object $fo = do.values[index]$ for some $index$ in sm , a statemask em (which comes from “equals mask”) of $index$ values such that $do.values[index] = fo$, and a statemask nm (which comes from “non-equal mask”) of $index$ values such that $do.values[index] \neq fo$. At this point, $linFields$ first pushes onto $stack$ an entry with the backtracking information for nm and then continues the linearization of fo for the active states indicated in em .

F. Merging

The dual of splitting sets of states into subsets is *merging* several sets of states into a larger set. Recall the exploration for Δ Execution from Figure 4. It merges all non-visited states from the previous iteration into a Δ State to be used for the current iteration. More precisely, our current implementation of the $merge$ function receives as input the set of linearizations representing those non-visited states.

Our merging algorithm uses *delinearization* to construct a Δ State 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 Δ State 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: 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 Δ Objects that should be constants (with respect to the reachability of the nodes), which results in a more efficient Δ State. Such constants can occur quite often; for instance, in our experiments (see Section V), the percentage of the constant Δ Objects in the merged Δ States ranges from 33% (for `bst` and $N = 11$) to 69% (for `treemap` and $N = 12$). Second, the merging algorithm *greedily* shares the objects in the resulting Δ State: it attempts to share the same Δ Object among as many individual states as possible. For example, in Figure 6, the left node from the root is shared among three of the five states.

Figure 11 shows the pseudocode for our merging algorithm. For simplicity of presentation, the algorithm assumes that the heap contains only reference fields and of only one class. Our actual implementation handles general heaps with objects of different classes, primitive fields, and arrays. The input is an array of linearizations, and the output is a root object for a Δ State. The algorithm maintains an array of maps from object ids to actual objects (which handles aliasing and is dual of ids used in linearization in Figure 10) and an array of offsets that track progress through the different linearizations (since they do not need to go in a “lockstep”).

The function $createObject$ constructs *one* object shared for all states in the given statemask and invokes $createDeltaObject$ to construct each field of the object. Note that this sharing does not constitute aliasing in the standard semantics since only one reference is visible for any given state.


```

Object merge(Lin[] lin)
  N = |lin| // number of individual states
  offsets = array (size N) of 0's
  maps = array (size N) of empty maps // id→Object
  sm_init = {0, ..., N-1} // statemask for all states
  return createObject(sm_init)

Object createObject(StateMask sm)
  o = new Object
  foreach i in sm do
    id = lin[i][offsets[i]++]
    put(maps[i], id, o)
  od
  foreach field f in o do o.f = createDeltaObject(sm)
  return o;

DeltaObject createDeltaObject(StateMask sm)
  d = new DeltaObject
  cm = {}; // statemask if new object is needed
  foreach i in sm do
    id = lin[i][offsets[i]++]
    if (id = NULL) then d.values[i] = null
    else if (id ∈ maps[i]) then
      d.values[i] = get(maps[i], id)
    else // need to create a new object for this id
      cm = cm ∪ {i}; offsets[i]--
  od
  if (|cm| > 0) then
    co = createObject(cm)
    // greedily share new object across states
    foreach i in cm do d.values[i] = co
  if (d.values is constant with respect to sm) then
    // use constants where possible
    d = new DeltaObjectConst
  return d

```

Fig. 11. Pseudocode for the merging algorithm.

The function *createDeltaObject* examines the field values across all states in the statemask *sm*. For each state, it checks for three possible options for the field's object id: (i) it denotes the null reference, (ii) it denotes an alias, or (iii) it denotes a new object. For the first two options, the algorithm assigns the value to the Δ Object *d* as it performs the check. For the third option, it just records in the statemask object *cm* the index of the state during the check. If the statemask *cm* is not empty after the check across all states, the algorithm recursively invokes (once) *createObject* to create an object that will be shared among the states in *cm*. Lastly, the algorithm checks if the Δ Object *d* is semantically a constant, i.e., if it contains the same value across all states denoted by *sm*. If so, a special constant object is created.

For states that have aliases between objects (unlike binary search tree), this greedy algorithm does not always produce a Δ State with the smallest number of nodes, and some alternative algorithms could produce smaller graphs. However, such alternative algorithms would require more time to search for appropriate sharing opportunities that result in smaller Δ States. A detailed example is available in d'Amorim's PhD thesis [14].

IV. IMPLEMENTATION

We implemented Δ Execution in two model checkers: JPF and BOX. JPF [44] is a popular model checker for Java programs; it is general-purpose and can handle multi-threaded Java programs. For the purpose of evaluating the technique under different implementations, we also implemented BOX (from *Bounded Object Explorer*), a model checker specialized for sequential programs.

A. JPF

We implemented Δ Execution by modifying JPF version 4 [1]. 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 dynamic class loading in Java). However, our experiments require only the part of the heap reachable from the root object. We therefore disabled JPF's default state comparison and instead use a specialized state comparison as done in some previous studies with JPF [16], [46], [49].

We next discuss how we implemented each component of Δ Execution in JPF. We call the resulting system Δ JPF. Δ JPF stores the Δ State as part of the JPF state, which allows the use of JPF backtracking to restore the Δ State at split points. We implemented the library operations on Δ State (such as arithmetic and relational operations, and 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.

We implemented splitting in Δ JPF 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 Δ State but uses statemasks to indicate the active states. Therefore, Δ JPF manages statemasks on the host JVM, independent of the backtracked state. We also implemented merging to execute on the host JVM and to create one Δ State as a JPF state that encodes all the non-visited states encountered in the previous iteration of the exploration. Recall that our experiments use breadth-first exploration.

Δ JPF uses instrumented code to invoke the operations that manipulate the Δ State. Section III-D describes in detail how instrumentation changes standard classes and introduces corresponding Δ Classes. Manual instrumentation is not particularly difficult but can be time-consuming and error prone. To automate instrumentation for Δ JPF, we developed a plug-in for Eclipse version 3.2 (<http://www.eclipse.org>). The plug-in takes a subject class (such as the *Node* class in our binary search tree example) and manipulates its internal AST representation to produce an instrumented class as described in Section III-D. Also, the plug-in generates Δ Classes from templates. For example, it generates the *DeltaNode* class in Figure 8 from information extracted from the original *Node* class. The plug-in takes as input the fields and constructors provided by the original class and generates accessors, mutators, a method to compare reference equality, and modified constructors. The template parameters relate mostly to method names, return types, and argument types. For example, the plug-in creates the method *get_left()* from a template by replacing a field name parameter with *left* to produce the expression *values[index].left*. The new Δ Class also provides the internal representation for the set of references to *Node* objects. In Figure 8, the class *DeltaNode* explicitly represents the set of references as an array of *Node* objects. In practice, we hide the representation in an interface so that we can experiment with different implementations such as sparse representation that only maps the active state indexes into values (Section III-C).

B. BOX

We 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,

and 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 test exploration needs to store, restore, and compare only the heap part of the state. Therefore, we used the existing ideas from state-space exploration research [3], [12], [20], [23], [31], [37], [43], [44] 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. BOX uses a *stateful* exploration (by restoring the entire state) *across iterations* and *stateless* exploration (by re-executing one method at a time) *within an 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. (This instrumentation is required even for standard, non- Δ exploration.)

We refer to the Δ Execution implementation in BOX as Δ BOX. Δ BOX needs to backtrack the Δ State in order to explore a method for various statemasks. In order to do this, Δ BOX restores the state to the beginning of the method execution by undoing any changes performed on the heap, and then *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 [20] is a well-known model checker that effectively employs re-execution.

Δ BOX implements the components of Δ Execution as presented in Section III. Δ BOX represents a Δ State as a regular Java state that contains both Δ Objects and objects of the instrumented classes. Δ BOX uses instrumented code to perform the operations on the Δ State. Instrumentation of code for Δ BOX (as well as for BOX itself) is mostly manual at this time, though it could be automated in a fashion similar to that used for Δ JPF. Like Δ JPF, Δ BOX merges states between iterations of the breadth-first exploration.

V. EVALUATION

We next present an experimental evaluation of Δ Execution. We first discuss the improvements that Δ Execution provides for an exhaustive exploration of ten basic subject programs in both JPF and BOX. We then present the results of performing a non-exhaustive exploration using Δ Execution in JPF. Finally, we present the improvements that Δ Execution provides on a larger case study, an implementation of the AODV routing protocol [35] in the J-Sim network simulator [24].

We performed all experiments on a Pentium 4 3.4GHz workstation running 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.

A. Exhaustive exploration

To evaluate the performance of Δ Execution for exhaustive exploration we used ten basic subject programs taken from a variety of sources: *binheap* is an implementation of priority queues using binomial heaps [46]; *bst* is our running example that implements a set using binary search trees [7], [49]; *deque* is our implementation of a double-ended queue using doubly

linked lists [15]; *fibheap* is an implementation of priority queues using Fibonacci heaps [46]; *filesystem* is based on the Daisy file-system code [36]; *heaparray* is an array-based implementation of priority queues [7], [49]; *queue* is an object queue implemented using two stacks [18]; *stack* is an object stack [18]; *treemap* is an implementation of maps using red-black trees based on Java Collections 1.4 [46]; *ubstack* is an array-based implementation of a stack bounded in size, storing integers without repetition [13], [34], [41]. These are small programs, ranging from 1 class (for *heaparray* and *ubstack*) to 4 classes (for *filesystem*) and from 27 (for *stack*) to 301 (for *treemap*) non-comment, non-blank lines of code.

Since the primary purpose of this portion of the evaluation is to compare the efficiency of Δ Execution and standard execution, we use correct implementations of all ten basic subjects. For instance, the original code for the Daisy *filesystem* had seeded errors, but we use a corrected version provided by Darga and Boyapati [18]. (In contrast, the AODV case study described in Section V-C uses code with errors that violate a safety property.)

For each subject described above, we wrote for both standard execution and for Δ Execution *test drivers* [45], small programs whose executions on JPF and BOX correspond to the state-space explorations shown in figures 2 and 4. The drivers exercise the main mutator methods for each subject. 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.

Table I shows the experimental results for exhaustive exploration. For each subject and several bounds (on the sequence length and parameter size, as in the pseudocode shown in figures 2 and 4), we tabulate the overall exploration time and peak memory usage with and without Δ Execution in both JPF and BOX. The cells marked "*" indicate that the experiment either ran out of 1.8GB of memory or exceeded the 1 hour time limit.

The columns labeled "std/ Δ " show the improvements that Δ Execution provides over standard execution for the ten basic subjects. Note that the numbers are ratios and not percentages; for example, for *binheap* and $N = 7$, the ratio of times is 10.82x, which corresponds to about 90% decrease. For JPF, the speedup ranges from 1.06x (for *filesystem* and $N = 6$) to 126.80x (for *heaparray* and $N = 9$), with median 5.60x. For BOX, the speedup ranges from 0.58x (for *filesystem* and $N = 3$, which actually represents almost a 2x slowdown) to 4.16x (for *queue* and $N = 7$), with median 2.23x. Note that the ratio less than 1.00x means that Δ Execution 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, Δ Execution consistently runs faster than standard execution for important cases with larger bounds.

Δ Execution provides these significant improvements because it exploits the overlap among executions in the state-space exploration. Table I also shows the information about the state spaces explored in the experiments. Note that the number of explored states is the same with and without Δ Execution. This is as expected: Δ Execution focuses on improving the exploration time and does not change the exploration itself. (We used the difference in the number of states to debug our implementations of Δ Execution.) However, the numbers of executions with and without Δ Execution do differ, and the column labeled "std/ Δ " shows the ratio of the numbers of executions. The ratio ranges from 10x

TABLE I

OVERALL TIME AND MEMORY FOR EXHAUSTIVE EXPLORATION AND CHARACTERISTICS OF THE EXPLORED STATE-SPACES; “*” INDICATES EXPERIMENTS THAT RAN OUT OF EITHER MEMORY OR TIME; “-” INDICATES UNRELIABLE MEASUREMENT OF MEMORY DUE TO SHORT RUNNING TIME.

experiment		JPF results				BOX results				state-space characteristics			
		time (sec)		mem.		time (sec)		mem.		# states		# executions	
subject	N	std	Δ	std/ Δ	std/ Δ	std	Δ	std/ Δ	std/ Δ	std & Δ	std	Δ	std/ Δ
binheap	7	24.87	2.30	10.82x	1.16x	0.78	0.35	2.23x	2.71x	16864	236096	401	588
	8	458.81	11.92	38.50x	1.03x	11.63	3.38	3.44x	1.08x	250083	4001328	863	4636
	9	*	*	*	*	106.54	32.74	3.25x	1.04x	1353196	24357528	1069	22785
bst	9	44.02	7.86	5.60x	0.70x	2.42	1.53	1.59x	0.77x	46960	845280	10846	77
	10	214.06	30.13	7.11x	0.46x	12.55	7.51	1.67x	0.30x	206395	4127900	22688	181
	11	*	*	*	*	67.64	49.62	1.36x	0.18x	915641	20144102	46731	431
deque	8	54.70	4.13	13.25x	1.50x	2.20	0.77	2.86x	1.54x	69281	1108496	576	1924
	9	552.11	28.84	19.14x	1.48x	22.38	7.48	2.99x	1.14x	623530	11223540	810	13856
	10	*	*	*	*	281.84	99.77	2.82x	1.18x	6235301	124706020	1100	113369
fibheap	6	3.18	1.46	2.17x	0.98x	0.22	0.16	1.40x	-	3003	21021	82	256
	7	25.09	2.82	8.90x	2.13x	1.16	0.66	1.76x	1.24x	36730	293840	130	2260
	8	400.84	21.59	18.57x	0.88x	16.77	9.75	1.72x	0.68x	544659	4901931	209	23454
filesystem	3	1.98	1.88	1.06x	0.97x	0.14	0.25	0.58x	-	58	6264	576	10
	4	17.18	3.08	5.59x	11.50x	1.18	0.71	1.67x	1.72x	1353	194832	1568	124
	5	*	*	*	*	37.43	30.04	1.25x	0.97x	64576	11623680	3940	2950
heaparray	8	104.96	3.61	29.09x	2.31x	1.21	0.88	1.37x	1.24x	97092	873828	258	3386
	9	2,724.63	21.49	126.80x	1.22x	11.92	8.91	1.34x	0.53x	804809	8048090	359	22418
	10	*	*	*	*	127.10	110.26	1.15x	0.58x	8722946	95952406	488	196623
queue	6	6.46	1.46	4.42x	2.64x	0.37	0.16	2.25x	-	10057	70399	45	1564
	7	84.42	5.08	16.63x	1.77x	3.87	0.93	4.16x	1.44x	147995	1183960	60	19732
	8	*	*	*	*	78.62	25.36	3.10x	1.00x	2578641	23207769	77	301399
stack	6	5.00	1.41	3.55x	1.01x	0.31	0.12	2.55x	-	9331	65317	42	1555
	7	59.70	4.14	14.43x	1.31x	2.92	0.71	4.09x	1.87x	137257	1098056	56	19608
	8	*	*	*	*	59.98	17.81	3.37x	1.31x	2396745	21570705	72	299593
treemap	12	274.26	53.40	5.14x	3.44x	32.88	9.12	3.61x	1.34x	96401	2313624	7774	297
	13	871.16	160.75	5.42x	3.90x	102.85	29.02	3.54x	1.48x	282532	7345832	11105	661
	14	2,860.23	562.70	5.08x	4.41x	365.54	104.09	3.51x	2.48x	844655	23650340	15178	1558
ubstack	8	61.52	4.60	13.37x	1.57x	2.26	1.28	1.77x	1.30x	109681	987129	595	1659
	9	1,502.24	32.54	46.17x	1.48x	22.60	13.52	1.67x	0.66x	991189	9911890	931	10646
	10	*	*	*	*	265.49	174.96	1.52x	0.62x	9922641	109149051	1414	77191
median	-	-	-	5.60x	1.48x	-	-	2.23x	1.18x	-	-	-	-

to 301399x. While this ratio effectively enables Δ Execution 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 Δ States, and the sharing of execution prefixes.

1) *Time*: We next discuss in more detail where state-space exploration spends time and specifically where Δ Execution reduces time. Each state-space exploration, both standard and Δ , includes three components: (i) (straightline) execution, (ii) backtracking, and (iii) (state) comparison. Δ Execution additionally includes (iv) merging. Table II shows the breakdown of the overall exploration time on these four components for JPF and BOX.

In JPF, Δ Execution significantly reduces the time for code execution and state backtracking. For example, for `binheap` and $N = 7$, Δ Execution reduces the execution time from 17.62s to 0.59s and the backtracking time from 6.71s to 1.12s. These savings are big enough and make the times for merging and state comparison irrelevant. As mentioned earlier, JPF is a general-purpose model checker that stores and restores the entire Java state and thus has a high execution and backtracking overhead.

In BOX, Δ Execution sometimes results in a higher code execution time, yet still has a smaller overall exploration time. The reason is that Δ Execution achieves significant savings in the state comparison using the optimized algorithm from Section III-E. For example, for `bst` and $N = 11$, Δ Execution increases the execution time from 3.26s to 7.96s. However, it reduces the state comparison time from 57.19s to 20.35s, 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 Δ Execution, but the optimized state comparison is only possible for Δ Execution, which uses Δ States that enable the simultaneous comparison of a set of states.

2) *Memory*: Table I also provides a comparison of memory usage. Specifically, the columns labeled “mem. std/ Δ ” show the ratio of peak memory usage for standard execution and Δ Execution. Our experimental setup uses Sun’s `jstat` [42] 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 Δ Execution for most experiments. The results show that Δ Execution reduces memory use from 0.46x to 11.50x (with median 1.48x). Note that Δ Execution occasionally uses more memory, for example for `bst`. In BOX, Δ Execution reduces memory from 0.18x to 2.71x (with median 1.18x). Note that the median of memory use in BOX has a lower value than in JPF indicating that Δ Execution consumes more memory relative to the standard execution. This is due to the fact that the Δ Execution implementation in JPF partially uses native state. This state is managed by the host JVM, which has a better memory management than the JPF JVM. In contrast, in standard

TABLE II
TIME BREAKDOWN FOR JPF AND BOX EXPERIMENTS.

experiment		standard JPF time (sec)			Δ JPF time (sec)				standard BOX time (sec)			Δ BOX time (sec)			
subject	N	exec	comp	back	exec	comp	back	merg	exec	comp	back	exec	comp	back	merg
binheap	7	17.62	0.54	6.71	0.59	0.26	1.12	0.34	0.22	0.37	0.12	0.10	0.13	0.00	0.06
	8	364.45	4.90	89.46	3.99	2.21	1.24	4.48	4.57	3.74	2.78	1.01	1.43	0.01	0.87
	9	*	*	*	*	*	*	*	21.46	67.74	15.03	4.70	21.77	0.01	6.27
bst	9	20.44	4.20	19.39	2.25	2.40	1.94	1.27	0.23	1.90	0.18	0.47	0.73	0.01	0.29
	10	103.39	21.04	89.62	7.18	12.85	3.98	6.12	0.52	10.60	0.91	1.78	3.78	0.01	1.86
	11	*	*	*	*	*	*	*	3.26	57.19	4.42	7.96	20.35	0.02	21.14
deque	8	25.50	3.45	25.75	0.72	1.08	1.18	1.14	0.32	1.50	0.33	0.15	0.39	0.00	0.19
	9	267.42	38.31	246.38	6.37	12.19	1.26	9.02	2.30	16.26	3.17	1.36	4.46	0.00	1.57
	10	*	*	*	*	*	*	*	21.95	214.30	31.48	16.48	59.01	0.01	23.87
fibheap	6	1.25	0.11	1.81	0.18	0.08	1.08	0.13	0.06	0.08	0.03	0.05	0.04	0.00	0.03
	7	14.69	0.86	9.53	0.42	0.31	1.20	0.89	0.31	0.51	0.29	0.21	0.24	0.00	0.18
	8	256.79	8.02	136.03	4.07	4.49	1.41	11.63	4.70	7.94	3.90	2.77	3.76	0.00	3.18
filesystem	3	0.24	0.05	1.69	0.20	0.15	1.46	0.07	0.02	0.09	0.01	0.04	0.06	0.00	0.03
	4	4.67	0.46	12.04	0.60	0.69	1.59	0.20	0.06	0.99	0.06	0.16	0.38	0.01	0.06
	5	*	*	*	*	*	*	*	3.30	30.35	1.70	16.74	10.45	0.02	2.59
heaparray	8	15.10	1.72	88.13	1.10	0.38	1.13	1.00	0.12	0.88	0.12	0.38	0.35	0.00	0.10
	9	160.36	17.38	2546.90	8.85	4.40	1.36	6.87	1.17	9.25	1.06	3.73	4.05	0.00	1.03
	10	*	*	*	*	*	*	*	11.47	98.01	10.46	44.85	46.36	0.01	18.52
queue	6	3.07	0.15	3.24	0.04	0.07	1.11	0.24	0.05	0.19	0.10	0.03	0.05	0.00	0.04
	7	48.30	1.52	34.60	0.18	0.70	1.10	3.09	0.80	1.85	1.09	0.07	0.45	0.00	0.39
	8	*	*	*	*	*	*	*	13.71	42.80	21.38	0.94	9.77	0.00	14.57
stack	6	1.77	0.10	3.13	0.02	0.06	1.13	0.20	0.04	0.16	0.08	0.02	0.04	0.00	0.03
	7	28.38	1.85	29.46	0.02	0.49	1.18	2.44	0.40	1.54	0.94	0.02	0.34	0.00	0.29
	8	*	*	*	*	*	*	*	7.04	34.77	16.58	0.02	7.54	0.00	10.21
treemap	12	191.51	26.06	56.70	5.05	43.52	2.00	2.83	1.44	29.60	1.26	1.55	6.74	0.02	0.66
	13	622.58	81.12	167.46	13.11	137.08	2.10	8.46	4.23	94.53	4.05	4.39	22.04	0.02	2.42
	14	2031.64	283.39	545.19	38.45	494.99	2.47	26.78	13.48	333.97	13.08	13.43	81.55	0.04	9.13
ubstack	8	31.95	2.58	26.99	1.34	0.97	1.13	1.15	0.22	1.68	0.24	0.36	0.70	0.00	0.16
	9	357.06	30.19	1114.99	14.09	9.06	1.37	8.02	2.64	16.96	1.62	3.94	7.96	0.00	1.54
	10	*	*	*	*	*	*	*	33.77	203.66	16.28	50.82	100.10	0.00	22.12

execution, only the JPF JVM handles the memory management.

Many factors, already mentioned for exploration time, can also influence memory usage, but a key factor is the number of constant Δ Objects in the merged state, i.e., in the Δ State. Δ Execution uses these objects to represent values that are the same across all states in a Δ State. We measured the percentage of all Δ Objects in merged states that are actually constant, across an entire exploration. For example, if we run an experiment for 2 iterations and find x_1 constants out of y_1 Δ Objects in the first iteration and x_2 out of y_2 in the second, then $(x_1 + x_2)/(y_1 + y_2)$ would be the percentage of constants. We found that there is a relatively strong positive correlation between the percentage of constant Δ Objects and the memory ratio for an experiment. For example, `bst` and $N = 11$ has a poor memory ratio, and the percentage of constant objects in Δ States is 33%, the lowest of all subjects. For `treemap` and $N = 12$, on the other hand, Δ Execution uses less memory than standard execution, and the percentage of constant objects is 69%. Note that this ratio of constants is “static” (measured during merging) and differs from the ratios discussed in Section II-E which are “dynamic” (measuring number of accesses during execution). The static ratio better reflects the memory usage.

B. Non-exhaustive exploration

We next evaluate Δ Execution for a different state-space exploration. While exhaustive exploration is the most commonly used, there are several others such as random [13], [34] or symbolic execution [2], [16], [26], [49]. Recently, Visser et al. [46] have proposed *abstract matching*, a technique for non-exhaustive state-space exploration of data structures. 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 fields. For example, all binary search trees of size one are considered equivalent. The exploration is pruned whenever it reaches a state equivalent to some previously explored state, which means that abstract matching can miss some portions of the state space.

We chose to evaluate Δ Execution for abstract matching because the JPF experiments done by Visser et al. [46] showed that abstract matching achieves better code coverage than five other exploration techniques, including exhaustive exploration, random, and symbolic execution. (The experiments did not consider whether higher code coverage results in finding more bugs.) Our evaluation uses the same four subjects used to evaluate abstract matching in JPF—`binheap`, `bst`, `fibheap`, and `treemap`—and we also ran each subject for sequence bounds up to $N = 30$ or until the experiment reached the time bound of 1 hour. We used the same test drivers as for exhaustive exploration, but randomized the order in which methods and argument values were chosen and used 10 different random seeds; Visser et al. use the same experimental setup to minimize the bias that a fixed order of method/value choices could have when combined with abstract matching.

Table III shows the results for abstract matching with and without Δ Execution. Δ Execution significantly reduces the overall exploration time for two subjects (`bst` and `treemap`) and slightly reduces or increases the time for the other two subjects (`binheap` and `fibheap`). Δ Execution provides a smaller speedup for the bounds explored for abstract matching (Table III) than for the bounds explored for exhaustive exploration (Table I). This can

TABLE III
OVERALL TIME FOR NON-EXHAUSTIVE EXPLORATION IN JPF.

experiment		standard JPF results			Δ JPF results			time
subject	N	time (sec)	#states	#exec.	time (sec)	#states	#exec.	std/ Δ
binheap	28	4.33	28	15680	4.12	28	956	1.05x
	29	4.42	29	16820	4.16	29	958	1.06x
	30	4.58	30	18000	4.27	30	1040	1.07x
bst	20	549.85	166064	10168360	90.86	150192	49645	6.05x
	21	1,237.36	381535	22466178	246.28	416946	77951	5.02x
	22	2,389.23	677848	43605496	380.42	626555	83569	6.28x
fibheap	28	18.68	881	182323	20.40	1041	7810	0.92x
	29	19.15	961	184320	20.35	1157	7269	0.94x
	30	28.68	1144	289571	28.56	1354	10981	1.00x
treemap	20	195.50	11879	1492080	43.28	11952	39131	4.52x
	21	385.33	22455	2893212	65.82	20590	48974	5.85x
	22	661.17	38126	4918100	107.33	36550	59693	6.16x

be attributed to the reduced number of states and executions in abstract matching compared to exhaustive exploration. For example, for `bst`, abstract matching for $N = 20$ explores fewer states and executions (166,064 and 10,168,360, respectively) than exhaustive exploration for $N = 11$ (915,641 and 20,144,102). In addition, there is less similarity across states and executions in abstract matching than in exhaustive exploration. Indeed, abstract matching selects the states such that they differ in shape. (The peculiarity of `binheap` is that it has only one possible shape for any given size.)

Note that abstract matching can explore a different number of states and executions with and without Δ Execution. The reason is that standard execution and Δ Execution explore the states in a different order: while standard execution explores each state index in order, Δ Execution explores at once various subsets of state indexes based on the splits during the execution. Thus, these executions can encounter in different order states that have the same shape, and only the first encountered of those states gets explored. The randomization of non-deterministic method/value choices, which is necessary for abstract matching, also minimizes the effect that different orders could introduce for Δ Execution and standard execution. As Table III shows, Δ Execution can explore more states (for example for `bst` and $N = 21$) or fewer states (for example for `bst` and $N = 20$) than standard execution, but Δ Execution speeds up exploration whenever the shapes have similarities.

C. AODV case study

We also evaluated Δ Execution on a larger application, namely the implementation of the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [35] in the J-Sim network simulator [24]. This application was previously used to evaluate a J-Sim model checker [40] and a technique that improves execution time in explicit-state model checkers [17].

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 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 [40].

The implementation of AODV, including the required J-Sim library classes, consists of 43 classes with over 3500 non-comment, non-blank lines of code. We instrumented this code using our Eclipse plug-in that automates instrumentation for Δ Execution

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 (for both standard and Δ Execution).

We used for this case study the test driver previously developed for AODV [40]. Like the drivers used for exhaustive exploration, the AODV driver invokes various methods that simulate protocol actions: sending messages, receiving messages, dropping messages, etc. Unlike those drivers, the AODV driver also (i) includes guards that ensure that an action is taken only if its preconditions are satisfied and (ii) includes a procedure that checks whether the resulting protocol state satisfies the safety property described above. In this experiment, when a violation is encountered, that state/path is pruned, but the overall exploration continues.

We ran experiments on three variations of the AODV implementation, each containing an error that leads to a violation of the safety property [40]. Table IV 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. Table IV also includes the breakdown of time for the AODV experiments. Note that most of the time in Δ Execution goes to the execution operation indicating that AODV is much more complex code than the ten basic subjects.

We implemented two optimizations in the evaluation of AODV. The first introduces a special treatment for pre- and post-conditions of methods that implement AODV actions. The second takes advantage of domain-specific knowledge about AODV: some data structures in the AODV state are semantically sets, e.g., it does not matter in which order a routing tables for an AODV node stores its entries.

1) *Pre- and post-conditions*: The evaluation of method pre- and post-conditions can split the execution in Δ Execution, effectively leading a model checker to exercise an AODV method (say, dropping a message) more than once in a given iteration, with different statemasks. This reduces the potential of Δ Execution to take advantage of the similarity across states and paths (when splitting on pre-conditions) and results in a less efficient merging (when splitting on post-conditions). However, it is unnecessary to exercise an AODV method differently for different paths of executions through pre- and post-conditions: the only result that matters is the boolean value of the conditions, not how the value is obtained. To speed up the exploration, we changed the delta exploration for AODV to merge the statemasks after evaluating the pre-conditions and before evaluation of the post-conditions.

TABLE IV
EXPLORATION OF AODV IN JPF.

experiment		standard JPF time (sec)				Δ JPF time (sec)					time	mem	# states
subject	N	total	exec	comp	back	total	exec	comp	back	merg	std/ Δ	std/ Δ	std & Δ
aodv	6	6.87	3.21	0.20	3.46	7.81	4.82	0.54	1.93	0.53	0.88x	0.53x	1061
	7	21.44	11.48	0.64	9.32	16.97	11.79	1.96	2.28	0.94	1.26x	0.56x	3796
	8	74.31	41.72	2.47	30.11	43.10	29.57	7.76	3.39	2.38	1.72x	0.52x	13195
	9	262.20	148.06	9.51	104.63	128.60	85.88	29.68	6.00	7.04	2.04x	0.58x	44735
	10	926.60	522.49	36.18	367.92	485.14	337.67	110.65	14.46	22.36	1.91x	0.51x	147805

This way, for instance, the model checker executes a method only once (in a given iteration) against all states that evaluate the precondition to true. This is a general optimization that can apply to any subject where method pre- and post-conditions are clearly identified.

2) *Special data structures*: Some data structures that the AODV implementation uses are sets implemented with lists. As a result of comparing states at the implementation level, the model checker can explore more states than necessary. For instance, two states can differ in the order of the elements in the lists although they represent the same set. The routing table is a key data structure in AODV, so we changed the implementation to keep the routing tables as sorted lists. This change comes with the cost of sorting the table when it is updated. However, it results in fewer explored states—because the model checker finds more states equivalent—in both standard and Δ Execution.

VI. RELATED WORK

Handling state is the central issue in explicit-state model checkers [22], [23], [28], [30]. For example, JPF [44] implements techniques such as efficient encoding of Java program state and symmetry reductions to help reduce the state-space size [28]. Our Δ Execution uses the same state comparison, based on Iosif’s depth-first heap linearization [23]. However, Δ Execution leverages the fact that Δ States 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 Δ Execution to optimize it.

Darga and Boyapati proposed glass-box model checking [18] for pruning search. They use a static analysis that can reduce state space without sacrificing coverage. Glass-box exploration represents the search space as a BDD and identifies 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, Δ Execution does not require any additional annotation on the code.

Symbolic execution [26], [45], [49] is a special kind of execution that operates on symbolic values. The state includes symbolic variables (which 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 [2], [26], [45], [49]. Common problems in symbolic execution include the treatment of arrays, object graphs, loops (and recursion), domains of unbounded size, libraries, and native code. CBMC [10] addresses these problems using paths of bounded length and finite input domains. The recent techniques combining symbolic execution and random execution show good promise in addressing some

of these problems [9], [21], [39]. Conceptually, both symbolic execution and Δ Execution operate on a set of states. While symbolic execution can represent an unbounded number of states, Δ Execution uses an efficient representation for a bounded set of concrete states. The use of concrete states allows Δ Execution to overcome some of the problems that symbolic execution has with representing dynamically allocated data (heap).

Shape analysis [27], [38], [50] is a static program analysis that verifies programs that manipulate dynamically allocated data. 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 Δ Execution. Shape analysis computes overapproximations of the reachable sets of states and loses precision to obtain tractability. In contrast, Δ Execution operates precisely on sets of concrete states but can explore only bounded executions.

Offutt et al. [33] 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 since it splits the ranges when it adds constraints 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. Δ Execution focuses on object graphs and does not require inputs to be ranges. However, the use of ranges as a special representation in Δ States could likely improve Δ Execution even more, so we plan to investigate this in the future.

In the introduction, we discussed the relationship between symbolic model checking [11], [25] and Δ Execution. Δ Execution 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 [5], [6] 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 [29], [47] to represent analysis information as sets and relations. These techniques employ either data [29] or control abstraction [47] 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 Δ Execution mode.

We previously proposed a technique, called Mixed Execution, for speeding up straightline execution in JPF [17]. 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%. Δ Execution

considers multiple states, improving the exploration time up to two orders of magnitude.

VII. CONCLUSIONS

We presented Δ Execution, a novel technique that significantly speeds up state-space exploration of object-oriented programs. State-space exploration is an important element of model checking and automated test generation. Δ Execution 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 Δ Execution is Δ State, a representation of a set of states that permits efficient operations on the set. The experiments done on two model checkers, JPF and BOX, and with two different kinds of exploration show that Δ Execution can reduce the time for state-space exploration from two times to over an order of magnitude, while taking on average less memory in JPF and roughly the same amount of memory in BOX.

In the future, we plan to apply the ideas from Δ Execution 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, doing so for the merging of Δ States would further improve the speedup of Δ Execution. Second, we plan to explore the applicability of Δ Execution for multi-threaded programs. For instance, it may be possible to efficiently execute code sections for multiple thread interleavings at the same time using Δ Execution. Third, we plan to evaluate automatic Δ Execution outside of state-space exploration. In regression testing, for example, the old and the new versions of a program can run in the “delta mode” which would allow a detailed comparison of the states from the two versions. We believe that Δ Execution can also provide significant benefits in these new domains.

ACKNOWLEDGMENT

We would like to 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 [18], 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 [51]—for their comments on this work. This work was partially supported by NSF grants CNS 0613665 and CNS 0615372 and by a CAPES fellowship under grant #15021917. We also acknowledge support from Microsoft Research. We also thank the anonymous reviewers of our ISSTA 2007 paper [15] and this journal article for their comments that helped us improve the presentation.

REFERENCES

- [1] JPF webpage. <http://javapathfinder.sourceforge.net>.
- [2] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 134–138, 2007.
- [3] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 484–487, 2004.
- [4] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient dynamic analysis for Java. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 462–465, 2004.
- [5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
- [6] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the International SPIN Workshop on Model Checking of Software (SPIN)*, pages 113–130, 2000.
- [7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.
- [8] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
- [9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.
- [10] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCN*, pages 168–176, 2004.
- [11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
- [12] 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 *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.
- [13] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software - Practice and Experience*, 34:1025–1050, 2004.
- [14] M. d’Amorim. *Efficient Explicit-State Model Checking of Programs with Dynamically Allocated Data*. Ph.D., University of Illinois at Urbana-Champaign, Urbana, IL, Oct. 2007.
- [15] M. d’Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 50–60, New York, NY, USA, 2007. ACM Press.
- [16] 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 *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, pages 59–68, 2006.
- [17] M. d’Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *Proceedings of International Conference on Formal Methods and Software Engineering (ICFEM)*, volume 4260, pages 549–567, 2006.
- [18] P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 363–382, 2006.
- [19] C. DeMartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.
- [20] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 174–186, 1997.
- [21] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 40, pages 213–223, New York, NY, USA, 2005. ACM Press.
- [22] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [23] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, page 254, Washington, DC, USA, 2001. IEEE Computer Society.
- [24] J-Sim. <http://www.j-sim.org/>.
- [25] 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 *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
- [26] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the*

- International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, April 2003.
- [27] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 17–32, 2002.
- [28] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proceedings of the international SPIN workshop on Model checking of software (SPIN)*, pages 80–102, Toronto, Canada, 2001.
- [29] O. Lhotak and L. Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation (PLDI)*, pages 158–169, New York, NY, USA, 2004. ACM Press.
- [30] M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In *Proceedings of the International SPIN Workshop on Model Checking of Software (SPIN)*, pages 28–42, 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 *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, December 2002.
- [32] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 446–455, New York, NY, USA, 2007. ACM Press.
- [33] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software - Practice and Experience*, 29(2):167–193, 1999.
- [34] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 504–527, Glasgow, Scotland, July 2005.
- [35] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100. IEEE Computer Society Press, 1999.
- [36] S. Qadeer. Daisy File System. Joint CAV/ISSTA Special Event on Specification, Verification, and Testing of Concurrent Software. 2004.
- [37] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proceedings of the European Software Engineering Conference and SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 267–276, 2003.
- [38] R. Rugina. Shape analysis quantitative shape analysis. In *Proceedings of the Static Analysis Symposium (SAS)*, pages 228–245, 2004.
- [39] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, Sept. 2005.
- [40] A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*, pages 235–250, 2005.
- [41] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proceedings of the XP/Agile Universe Conference*, pages 131–143, 2002.
- [42] Sun Microsystems. *jstat: Java Virtual Machine Statistics Monitoring Tool*. <http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html>.
- [43] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 273–282, New York, NY, 2005. ACM Press.
- [44] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.
- [45] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107, 2004.
- [46] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.
- [47] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.
- [48] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 196–205, Sept. 2004.
- [49] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 365–381, Apr. 2005.
- [50] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–545, 2004.
- [51] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d’Amorim, S. Lauterburg, R. M. Lefever, and J. Tucek. Delta execution for software reliability. In *Workshop on Hot Topics in System Dependability (HotDep)*, Edinburgh, UK, June 2007.



Marcelo d’Amorim received his PhD degree from the University of Illinois at Urbana-Champaign, USA, in October 2007. He is currently a post-doctorate research fellow in the Software Productivity Group at the Universidade Federal de Pernambuco, Brazil. His research interest is on productivity in software engineering, focusing on the study of automated techniques for testing and debugging. More information is available at <http://cin.ufpe.br/~damorim>.



Steven Lauterburg received his BS degree in Computer Science in 1985 from the University of Illinois at Urbana-Champaign and his MS degree in Computer Science in 2004 from DePaul University in Chicago. He has extensive software industry and process improvement experience, accumulated while working at Accenture for over 17 years. Steven is currently a PhD student at the University of Illinois at Urbana-Champaign. His research interests include software testing, model checking, and program analysis. More information is available at <http://mir.cs.uiuc.edu/~slauter2>.



Darko Marinov is an Assistant Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He received his PhD degree in 2005 and SM degree in 2000, both in Computer Science from MIT. His main research interests are in software engineering, with an emphasis on improving software reliability, using software testing and model checking. More information is available at <http://www-faculty.cs.uiuc.edu/~marinov>.