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 ΔExecution, 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. ΔExecution exploits the fact that many execution paths in state-space exploration partially overlap. ΔExecution speeds up the state-space exploration by sharing the common parts across the executions and separately executing only the “deltas” where the executions differ. Central to ΔExecution is an efficient representation and manipulation of sets of states. ΔExecution 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 ΔExecution operates on concrete states.

ΔExecution 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.

ΔExecution operates on a ΔState, a novel representation for sets of states that include heaps. We describe efficient operations for manipulating ΔStates, which enable ΔExecution to execute programs faster than standard execution. These operations also enable ΔExecution to speed up state comparison and backtracking, two important and costly parts of state-space exploration. The key to these speed-ups in ΔExecution 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 ΔExecution 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 [11], [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
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:                } else if (temp.info > info) {
       10:                   temp = temp.right;
       11:                } else if (temp.info == info) {
       12:                     break;
       13:                } else temp = temp.left;
       14:            } else return; // no duplicates
       15:        size++;
       16:    }

       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 ∆Execution on model checkers that follow different design principles. While we found out that ∆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 ∆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 ∆Execution on ten simple subject programs and one larger case study, AODV [35]. For simple subject programs, ∆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 ∆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 ∆Execution increases time or memory usage.) For AODV, ∆Execution reduces exploration time from 0.88x to 2.04x (with median 1.72x) in JPF. For the non-exhaustive exploration, ∆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 what ∆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 sinit 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 snext. 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 V ∪ {sinit}, 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...
add(4)

trees of size 3 with values between 1 and 3. When it comes
17, 18. We say
shown in the middle results in the following trace (for progr am
execution operates on a single state at a time.

exploration of the binary search tree subject for

add(4)

BST t = new

BST t = new BST(); t.add(2)

pruned and would miss, for instance, the sequence

-----------

Both of these executions follow the same aforementioned pat
add(4)

on these five trees, standard exploration

space exploration. While this example shows the case when two
executions have identical paths, \( \Delta \text{Execution} \) 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 us-
ing \( \Delta \text{Execution} \). 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) \( \Delta \text{State} \): \( \Delta \text{Execution} \) conceptually operates on multiple in-
dividual states at the same time. More precisely, \( \Delta \text{Execution} \) oper-
ates on a single \( \Delta \text{State} \) that represents several standard
states, each corresponding to one of the individual states found
in a standard execution. The type of the root object \( \sigma_{\text{root}} \) in
delta exploration is \( \Delta \text{State} \). While standard execution invokes
add(4) separately against each standard state, \( \Delta \text{Execution} \) in-
vokes add(4) on one \( \Delta \text{State} \), effectively invoking it simulta-
neously 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 \( \Delta \text{State} \).

2) Splitting: When a method is executed on a \( \Delta \text{State} \), the
result can be more than one \( \Delta \text{State} \): \( \sigma_1, \ldots, \sigma_k \). Each of these
resulting \( k \) \( \Delta \text{States} \) represents the subset of individual states from
the original \( \Delta \text{State} \) 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 \( \Delta \text{States} \) is equal to the
number of individual states in the original \( \Delta \text{State} \) the method is
executed on, i.e., \( \sum_{i=1}^{k} |\sigma_i| = |\sigma_{\text{root}}| \).

During method execution, \( \Delta \text{Execution} \) occasionally needs to
split the \( \Delta \text{State} \). 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, \( \Delta \text{Execution} \) splits the set of states: one subset
of (two states) follows the true branch, and the other subset

Fig. 3. Executions of add(4) on a \( \Delta \text{State} \).

\[
\begin{align*}
\text{BST} & = \text{new BST(); t.add(2); t.add(3); t.remove(1) before the sequence BST t = new BST(); t.add(2). Since both sequences result in the same tree state, the latter would be pruned and would miss, for instance, the sequence BST t = new BST(); t.add(2); t.add(3).}
\end{align*}
\]

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

add(4)

---

// \( N \) bounds sequence length and parameter values
exploreDelta(N)
Next = \{s_{init}\};
Visited = \{linearize(s_{init})\};
for \( i = 1 \) to \( N \) do // iterations
\Delta\text{State } \sigma_{\text{root}} = \text{merge(Next)}; \text{ Next } = \{\};
for each method \( m \) in methods do
for \( v = 1 \) to \( N \) do
\{\sigma_1, \ldots, \sigma_k\} = \text{execute}_\Delta m(v) on \sigma_{\text{root}}; \text{ // splits}
for each \( \sigma \in \{\sigma_1, \ldots, \sigma_k\} \) do
for each \( l \in \text{linearize}_\Delta(\sigma) \) do
if \( l \notin \text{Visited} \) then
Visited = Visited \cup \{l\};
Next = Next \cup \{state for l\};

Fig. 4. Breadth-first exploration using \( \Delta\text{Execution} \).

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). \( \Delta\text{Execution} \) exploits this similarity to speed up state-
space exploration. While this example shows the case when two
executions have identical paths, \( \Delta\text{Execution} \) 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 us-
ing \( \Delta\text{Execution} \). 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) \( \Delta\text{State} \): \( \Delta\text{Execution} \) conceptually operates on multiple in-
dividual states at the same time. More precisely, \( \Delta\text{Execution} \) oper-
ates on a single \( \Delta\text{State} \) that represents several standard
states, each corresponding to one of the individual states found
in a standard execution. The type of the root object \( \sigma_{\text{root}} \) in
delta exploration is \( \Delta\text{State} \). While standard execution invokes
add(4) separately against each standard state, \( \Delta\text{Execution} \ in-
vokes add(4) on one \( \Delta\text{State} \), effectively invoking it simulta-
neously 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 \( \Delta\text{State} \).

2) Splitting: When a method is executed on a \( \Delta\text{State} \), the
result can be more than one \( \Delta\text{State} \): \( \sigma_1, \ldots, \sigma_k \). Each of these
resulting \( k \) \( \Delta\text{States} \) represents the subset of individual states from
the original \( \Delta\text{State} \) 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 \( \Delta\text{States} \) is equal to the
number of individual states in the original \( \Delta\text{State} \) the method is
executed on, i.e., \( \sum_{i=1}^{k} |\sigma_i| = |\sigma_{\text{root}}| \).

During method execution, \( \Delta\text{Execution} \) occasionally needs to
split the \( \Delta\text{State} \). 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, \( \Delta\text{Execution} \) 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 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 \(\text{Execution}\) operates on sets of states (i.e., a \(\text{DeltaState}\)), a delta exploration needs to periodically combine multiple individual states (or multiple small \(\text{DeltaStates}\)) together into a single \(\text{DeltaState}\). 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 \(\text{DeltaState}\) 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. \(\text{Execution}\) can, in principle, perform merging on any sets of states at any program point. For example, \(\text{Execution}\) could merge all three sets of states from Figure 3 when they reach size++. However, as illustrated in Figure 4 our current implementation of \(\text{Execution}\) 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) \(\text{Linearization}\): Delta exploration uses the linearizeDelta operation to linearize the individual states in a \(\text{DeltaState}\) \(\sigma\) 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 \(\text{Execution}\) and standard execution compare. In our running example, \(\text{Execution}\) 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 \(\text{Execution}\) 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 \(N = 10\), \(\text{Execution}\) 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 \(\text{Execution}\) 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).

\(\text{Execution}\) 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].)

\(\text{Execution}\) can reduce the cost of backtracking as it reduces the number of executions. For example, for states from Figure 3, \(\text{Execution}\) backtracks 2 times (for 3 executions), while standard execution backtracks 4 times (for 5 executions). \(\text{Execution}\) 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 \(\sigma_1, \ldots, \sigma_k\) from Figure 4 is 3 in this example, while the size of Current from Figure 2 starts out as 5.

\(\text{Execution}\) 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 \(\text{Execution}\) and standard execution can be summarized as follows: \(\text{Execution}\) performs fewer executions (avoiding separate execution of the same path shared by multiple states) than standard execution, but each execution in \(\text{Execution}\) (that operates on a set of standard states) is more expensive than in standard execution (that operates on one standard state). Whether \(\text{Execution}\) 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 \(\text{Execution}\).

III. TECHNIQUE

The main idea of \(\text{Execution}\) is to execute a program simultaneously on a set of standard states. Figure 4 presents a high-level algorithm for \(\text{Execution}\). We first discuss some key properties of the algorithm. We then present more details of the algorithm. The central part of \(\text{Execution}\) is \(\text{DeltaState}\), 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.

\Delta 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 \Delta Execution and how \Delta 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 \Delta Execution is to speed up standard exploration; \Delta 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. \Delta State

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

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

Fig. 6 shows the \Delta State that represents the set of five pre-states from Figure 3. Each \Delta State consists of layers of “regular” objects and \Delta Objects. For this example, the circles represent Node objects, the single rectangle represents a BST object, the array-like structures represent either DeltaNode objects or DeltaInt objects, the stand-alone integers represent DeltaInt objects that are constants, and the stand-alone arrow leaving the top-most rectangle represents a DeltaNode object that is a constant. In this \Delta 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 \Delta 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 \Delta 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 \Delta 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 \Delta States uses an optimized representation for constants. When a field value is constant across all relevant states, that field is represented in the \Delta 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 \Delta State, and is important both for reducing the memory requirements of \Delta States and for improving the efficiency of operations on \Delta States.

C. Splitting

\Delta Execution operates on a \Delta State that represents a set of standard states. \Delta 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 \Delta 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 \Delta Execution is to efficiently split \Delta States. Our solution is to introduce a statemask that identifies the currently active states within a \Delta State. Each statemask is a set of state indexes. At the beginning of an execution, \Delta Execution initializes
the statemask to the set of all state indexes. For example, the execution of \( \text{add}(4) \) for the \( \Delta \text{State} \) from Figure 6 starts with the statemask being \{0, 1, 2, 3, 4\}.

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

After \( \Delta \text{Execution} \) finishes the execution path for some (sub)set of states, it \textit{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 \( \Delta \text{State} \) to exactly what it was at the split point. A model checker can implement backtracking in several ways. JPF, for instance, stores and 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 \( \Delta \text{State} \). Consider, for example, a \( \Delta \text{Object} \) 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 \textit{sparse representations} for \( \Delta \text{Objects} \): instead of using an array to map all possible state indexes into values, a sparse \( \Delta \text{Object} \) can use representations that \textit{map only the active state indexes into values}, thereby reducing the memory requirement.

D. Program execution model

We next discuss how \( \Delta \text{Execution} \) executes program operations. The key is to execute each operation simultaneously on a set of values. \( \Delta \text{Execution} \) uses a non-standard program execution that manipulates a \( \Delta \text{State} \) 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 \( \Delta \text{Execution} \).

We use parts of the instrumentation to describe the semantics of \( \Delta \text{Execution} \).

1) \textit{Classes}: The instrumentation changes the original program classes and generates new classes for \( \Delta \text{Objects} \). Figure 7 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 DeltaNode that stores and manipulates the multiple Node references that can exist across the multiple states in a \( \Delta \text{State} \). Figure 9 shows the class DeltaInt that stores and manipulates multiple int values; this class is a part of the \( \Delta \text{Execution} \) library and is not generated anew for each program.

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

2) \textit{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 \( \Delta \text{Objects} \).

3) \textit{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 \( \text{temp.left} \). In \( \Delta \text{Execution} \), \( \text{temp} \) 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 \( \text{get_left} \) method in DeltaNode. This method returns a DeltaNode object that references (one or more) Node objects that correspond to the left fields of all \( \text{temp} \) objects whose

```java
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(); temp != NULL;)
                if (temp.get_right().eq(DeltaNode.NULL)) {
                    temp.set_right(DeltaNode._new(info));
                    break;
                } else if (temp.get_info().lt(info)) {
                    temp.set_left(DeltaNode._new(info));
                    break;
                } else temp = temp.get_right();
        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; }
}

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.
Consider integer addition as an example of arithmetic operations. In standard execution, addition takes two integer values and creates a single value. In ΔExecution, it takes two DeltaInt objects and creates a new DeltaInt object. The add method in DeltaInt (from Figure 9) shows how ΔExecution conceptually performs pairwise addition across all active state indexes for the two DeltaInt objects. Our implementation optimizes the cases when those objects are constant (to avoid the 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 ΔObject, 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.

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

Figure 10 shows the pseudocode for an optimized algorithm that simultaneously linearizes all states from a ΔState. 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, linearizeΔ, takes as input an object o, which represents the root of a ΔState, and a statemask sm, which represents the active states in that ΔState. 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}(\text{Object } o, \text{StateMask } sm)\\nstack = \text{empty stack}\\n\ell, tm = \text{linObject}(o, \text{empty Map}, sm)\\nlinSet = \{\ell\} // all states in tm have sequence \ell\\n\text{while } (\lbrack \text{stack} \rbrack > 0) \text{ do}\\n\quad (o, f, ids, lpre, nm) = \text{pop from stack}\\n\quad \ell, tm = \text{linFields}(o, f, ids, lpre, nm)\\n\quad \text{linSet} = \text{linSet} \cup \{\ell\}\\n\text{od}\\n\text{return } \text{linSet};\\n\\n// returns a triple of a Map, Lin, and StateMask\\n\text{linObject}(\text{Object } o, \text{Map ids, StateMask } sm)\\n\text{if } (o = \text{null}) \text{ then return } \text{null}(\text{null, null), sm})\\n\text{if } (o \in \text{ids}) \text{ then return } \text{linMap}(\text{ids, [get(ids, o)], sm})\\n\quad id = \text{ids}\[o];\\n\quad \text{return } \text{linFields}(o, 0, \text{put(ids, o, id), [id], sm});\\n\\n// returns a triple of a Map, Lin, and StateMask\\n\text{linFields}(\text{Object } o, \text{int } f, \text{Map ids, Lin } l, \text{StateMask } sm)\\n\text{if } (f < \text{numberOfFields}(o)) \text{ then}\\n\quad \{fo, em, nm\} = \text{split(getField(o, f), sm)}\\n\text{if } (\text{nm} > 0) \text{ then}\\n\quad \text{push } (o, f, ids, l, nm) \text{ onto stack}\(m, \text{lpost, om}) = \text{linObject}(fo, \text{ids, em})\\n\quad \text{return } \text{linFields}(o, f + 1, m, \text{append(l, lpost, om)});\\n\text{else return } \text{linFields}(l, \text{id, sm});\\n\\nFig. 10. Optimized linearization of states in a $\Delta$State.

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

The function split in linFields takes as input a $\Delta$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] ≠ 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 $\Delta$Execution from Figure 4. It merges all non-visited states from the previous iteration into a $\Delta$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 $\Delta$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 $\Delta$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 $\Delta$Objects that should be constants (with respect to the reachability of the nodes), which results in a more efficient $\Delta$State. Such constants can occur quite often; for instance, in our experiments (see Section V), the percentage of the constant $\Delta$Objects in the merged $\Delta$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 $\Delta$State: it attempts to share the same $\Delta$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 $\Delta$State. The algorithm maintains an array of maps from object ids to actual objects (which handles aliasing and is the 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
sminit = (0,...,N−1) // statemask for all states
return createObject(sminit)

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 a,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
c0 = createObject(cm)
   // greedily share new object across states
foreach i in cm do d.values[i] = c0
if (d.values[i] 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 DeltaObject 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 DeltaObject 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 DeltaState 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 DeltaStates. A detailed example is available in d’Amorim’s PhD thesis [14].

IV. IMPLEMENTATION

We implemented ΔExecution 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 ΔExecution 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 ΔExecution in JPF. We call the resulting system ΔJPF. ΔJPF stores the ΔState as part of the JPF state, which allows the use of JPF backtracking to restore the ΔState at split points. We implemented the library operations on ΔState (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 ΔJPF on top of the existing non-deterministic choices in JPF. It is important to point out that our implementation leverages JPF to restore the entire ΔState but uses statemasks to indicate the active states. Therefore, ΔJPF 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 ΔState 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.

ΔJPF uses instrumented code to invoke the operations that manipulate the ΔState. Section III-D describes in detail how instrumentation changes standard classes and introduces corresponding ΔClasses. Manual instrumentation is not particularly difficult but can be time-consuming and error prone. To automate instrumentation for ΔJPF, 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 ΔClasses 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 getLeft() from a template by replacing a field name parameter with left to produce the expression values[index].left. The new ΔClass 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 ΔExecution implementation in BOX as ΔBOX. ΔBOX needs to backtrack the ΔState in order to explore a method for various statemasks. In order to do this, ΔBOX 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.

ΔBOX implements the components of ΔExecution as presented in Section III. ΔBOX represents a ΔState as a regular Java state that contains both ΔObjects and objects of the instrumented classes. ΔBOX uses instrumented code to perform the operations on the ΔState. Instrumentation of code for ΔBOX (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 ΔJPF. Like ΔJPF, ΔBOX merges states between iterations of the breadth-first exploration.

V. EVALUATION

We next present an experimental evaluation of ΔExecution. We first discuss the improvements that ΔExecution 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 ΔExecution in JPF. Finally, we present the improvements that ΔExecution 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 ΔExecution 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 ΔExecution 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 ΔExecution 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 ΔExecution 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 ΔExecution 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 a 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 ΔExecution ran slower (or required more memory) than standard execution, for example for filesystem and N = 3 in BOX. While this can happen for smaller bounds, ΔExecution consistently runs faster than standard execution for important cases with larger bounds.

ΔExecution provides these significant improvements because it exploits the overlap among executions in the state-space exploration. 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 ΔExecution. This is as expected: ΔExecution 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 ΔExecution.) However, the numbers of executions with and without ΔExecution do differ, and the column labeled “std/Δ” shows the ratio of the numbers of executions. The ratio ranges from 10x
<table>
<thead>
<tr>
<th>subject</th>
<th>JPF results</th>
<th>BOX results</th>
<th>state-space characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (sec)</td>
<td>mem. time (sec)</td>
<td>mem. time (sec)</td>
</tr>
<tr>
<td></td>
<td>std</td>
<td>std/Δ</td>
<td>std</td>
</tr>
<tr>
<td>entry</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binheap</td>
<td>7</td>
<td>23.87</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>458.81</td>
<td>11.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>bst</td>
<td>9</td>
<td>44.02</td>
<td>7.86</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>214.06</td>
<td>30.13</td>
</tr>
<tr>
<td>deque</td>
<td>8</td>
<td>54.70</td>
<td>4.13</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>552.11</td>
<td>28.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>file</td>
<td>8</td>
<td>148.47</td>
<td>25.09</td>
</tr>
<tr>
<td>filesystem</td>
<td>3</td>
<td>1.98</td>
<td>1.88</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>17.18</td>
<td>3.08</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>heaparray</td>
<td>8</td>
<td>104.96</td>
<td>3.61</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>2724.63</td>
<td>21.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>queue</td>
<td>6</td>
<td>6.46</td>
<td>1.46</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>84.42</td>
<td>5.08</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>78.62</td>
<td>25.09</td>
</tr>
<tr>
<td>stack</td>
<td>7</td>
<td>5.00</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>59.70</td>
<td>4.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>treemap</td>
<td>12</td>
<td>274.26</td>
<td>53.40</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>871.16</td>
<td>160.75</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>2860.23</td>
<td>562.70</td>
</tr>
<tr>
<td>stack</td>
<td>8</td>
<td>61.52</td>
<td>4.60</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1502.24</td>
<td>32.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>median</td>
<td>-</td>
<td>-</td>
<td>5.60x</td>
</tr>
</tbody>
</table>
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 Objects in the merged state, i.e., in the ∆State. ∆Execution uses these objects to represent values that are the same across all states in a ∆State. We measured the percentage of all ∆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 \( y_1 \) ∆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 ∆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 ∆States is 33%, the lowest of all subjects. For treemap and \( N = 12 \), on the other hand, ∆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 ∆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 ∆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 ∆Execution. ∆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). ∆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
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 \( \Delta \)Execution. The reason is that standard execution and \( \Delta \)Execution explore the states in a different order: while standard execution explores each state index in order, \( \Delta \)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 \( \Delta \)Execution and standard execution. As Table III shows, \( \Delta \)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 \( \Delta \)Execution speeds up exploration whenever the shapes have similarities.

### C. AODV case study

We also evaluated \( \Delta \)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 \( \Delta \)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 \( \Delta \)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 \( \Delta \)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 \( \Delta \)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 \( \Delta \)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.
This way, for instance, the model checker executes a method only once (in a given iteration) against all states that evaluate the pre-condition 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 ∆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 ∆Execution uses the same state comparison, based on Iosif’s depth-first heap linearization [23]. However, ∆Execution leverages the fact that ∆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 ∆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, ∆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 ∆Execution operate on a set of states. While symbolic execution can represent an unbounded number of states, ∆Execution uses an efficient representation for a bounded set of concrete states. The use of concrete states allows ∆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 ∆Execution. Shape analysis computes overapproximations of the reachable sets of states and loses precision to obtain tractability. In contrast, ∆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. ∆Execution focuses on object graphs and does not require inputs to be ranges. However, the use of ranges as a special representation in ∆States could likely improve ∆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 ∆Execution. ∆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 ∆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%. ∆Execution

### TABLE IV

<table>
<thead>
<tr>
<th>experiment</th>
<th>standard JPF time (sec)</th>
<th>∆JPF time (sec)</th>
<th>time</th>
<th>mem</th>
<th># states</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject</td>
<td>N</td>
<td>total</td>
<td>exec</td>
<td>comp</td>
<td>back</td>
</tr>
<tr>
<td>aodv</td>
<td>6</td>
<td>6.87</td>
<td>3.21</td>
<td>0.20</td>
<td>3.46</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>21.44</td>
<td>11.48</td>
<td>0.64</td>
<td>9.32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>74.31</td>
<td>41.72</td>
<td>2.47</td>
<td>30.11</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>262.20</td>
<td>148.06</td>
<td>9.51</td>
<td>104.63</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>926.60</td>
<td>522.49</td>
<td>36.18</td>
<td>367.92</td>
</tr>
</tbody>
</table>

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

VII. CONCLUSIONS

We presented \textit{DEExecution}, 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. \textit{DEExecution} 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 \textit{DEExecution} is \textit{DSSTATE}, 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 \textit{DEExecution} 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 \textit{DEExecution} 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 \textit{DSSTATEs} would further improve the speedup of \textit{DEExecution}. Second, we plan to explore the applicability of \textit{DEExecution} for multi-threaded programs. For instance, it may be possible to efficiently execute code sections for multiple thread interleavings at the same time using \textit{DEExecution}. Third, we plan to evaluate automatic \textit{DEExecution} 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 \textit{DEExecution} 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


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://cien.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/~slaeter2.

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.