On Delayed Choice Execution for Falsification

EPFL IC LARA-REPORT-2008-08

Milos Gligoric[1], Tihomir Gvero[1], Sarfraz Khurshid[2], Viktor Kuncak[3], and Darko Marinov[4]

[1] University of Belgrade, Serbia
[2] University of Texas at Austin, TX, USA
[3] EPFL, Switzerland
[4] University of Illinois at Urbana-Champaign, IL, USA

Abstract. We present an approach for finding errors in programs and specifications. We formulate our approach as an execution mechanism for a non-deterministic guarded-command language. Guarded commands have already proved useful for verification-condition generation but are usually viewed as a non-executable representation. We show how to execute guarded commands using an explicit-state model checker. We illustrate the benefits of this approach in two related domains: bounded-exhaustive testing and falsification for Hoare triples.
The basis of our approach is the delayed-choice technique for improving the execution of guarded commands. Delayed choice postpones non-deterministic choice of values until they are used. Our approach also supports copy-propagation of symbolic values but avoids the cost of full-blown symbolic execution. We describe an implementation of our approach in Java PathFinder, a popular model checker for Java programs. Our experimental results show that our techniques significantly improve performance compared to the current execution strategy in Java Pathfinder.

1 Introduction

Program annotations such as assertions, preconditions, postconditions, and invariants are a fundamental mechanism for increasing software reliability. When program annotations are detailed enough, they enable scalable modular verification in which a program component is verified independently of the rest of the program. Such approaches have been used for modular verification of memory-safe imperative programs resulting in tools such as ESC/Java [17] and Spec# [4]. Significant case studies have been completed using such techniques, including JavaCard electronic purse implementation verification using KeY and KIV [38, 19] and data structures verification using Jahob [43].

Such modular verification approaches reduce to validation of Hoare triples $\{P\}s\{Q\}$, denoting that executing a statement s in a non-error state satisfying P does not cause errors and results in a state satisfying Q. Typically, such modular verification is an interactive effort in which a substantial portion of time is spent

debugging too strong or too weak annotations (in addition to correcting any errors in the implementation). Such inadequate annotations correspond to invalid Hoare triples. Unfortunately, the tools geared towards proving validity of Hoare triples often provide only a limited support for *falsification*, i.e., generation of counterexamples to Hoare triples. Our approach can also be viewed as automated test generation [7], which automatically generates tests inputs satisfying P, then tests the program $s; \mathsf{assert}(Q)$ on these inputs.

To make modular verification more practical, this paper presents new techniques for falsification of a fairly general class of Hoare triples. We consider falsification for Hoare triples $\{P\}s\{Q\}$ where P and Q are properties expressed in an expressive (undecidable) language, and s denotes a potentially large piece of imperative code, with loops and recursion.

Approaches based on verification conditions. One approach to falsify a Hoare triple is to search for counterexamples to a verification condition for the Hoare triple. Unlike verification conditions for validity, which over-approximate program code, a verification condition for falsification needs to under-approximate program code. When s contains loops, this leads to techniques such as loop unrolling and procedure inlining. Once a verification condition is generated, model finding techniques for formulas can generate counterexamples to Hoare triple [23,15]. Among the challenges in this approach is the size of generated formulas and the limitations of model finders.

Our formulation using guarded commands. Instead of casting the problem in terms of formulas in an expressive logic, we represent the Hoare triple using a guarded-command language containing standard statements s (including assignments and loops) and special statements `assume`, `havoc`, `assert`. The validity of $\{P\}s\{Q\}$ then reduces to the absence of errors in the program

$$\texttt{havoc}(x_1, \ldots, x_n);\ \texttt{assume}(P);\ s;\ \texttt{assert}(Q)$$

where $x_1, \ldots, x_n$ denote state variables appearing in P, s, Q. Such representations are the basis of verification systems such as ESC/Java [17], Spec# [4], and Jahob [43,26].

We propose in this paper the use of execution-based, explicit-state model checking techniques to find assertion violations in non-deterministic guarded command programs, which gives counterexamples to the original Hoare triples. Moreover, we do not restrict ourselves to programs of the above form; we consider arbitrary guarded programs. This perspective led us to a solution applicable to falsification of a wide range of Hoare triples (where preconditions and postconditions are themselves expressed as arbitrary programs). At the same time, this technique provides an approach to bounded-exhaustive testing [7,29].

The use of a general-purpose language results in a system with a simple programming model that allows developers to control search efficiency by smoothly combining *predicates*, which specify the conditions of `assume` and `assert` statements, and *generators*, which directly generate states that satisfy these conditions, potentially utilizing domain-specific knowledge.

This approach allowed us to focus on the core technical problem of improving the efficiency of systematic exploration of guarded-command language executions. We implemented these ideas on top of the non-deterministic choices in Java PathFinder [41], a popular model checker for Java programs. We show that our approach can be based on a general principle of *delayed choice*, which postpones non-deterministic choices of variables until the execution requires concrete values. The implementation of this principle led to significant improvements in the Java PathFinder running times.

Contributions. We summarize our contributions as follows:

1. We show how to encode both Hoare triple falsification and test case generation as systematic exploration of non-deterministic program executions. The resulting approach
   - supports a combination of *predicates* (declarative specifications) and *generators* (imperative specifications) to describe preconditions
   - provides the semantic foundation and a generalization of the notion of test input *finitization* [7, 29, 32];
2. We present techniques that improve the performance of execution of guarded commands using Java PathFinder:
   - *delayed choice* technique postpones non-deterministic choice of values until they are examined, reducing the size of the search tree
   - *copy propagation* delays the choice of values even if the values are being copied, while avoiding the cost of full-blown symbolic execution;
3. We describe an implementation of our techniques in Java PathFinder. The results show that our techniques improve the time to generate test inputs up to a given bound or the time to find the first counterexample for a Hoare triple. Overall, they suggest that the approach is useful for detecting errors in code and specifications.

2 Our Approach through an Example

We next illustrate our approach of using guarded-command language execution for falsification and test generation. We first show how it supports two existing styles of precondition specifications: predicates and generators. We then show how it allows the developers to smoothly combine these styles, resulting in natural specifications and efficient search.

As an example we use red-black trees [11], a widely used data structure underlying, for example, the TreeMap class implementation of the Java Standard library. Thanks to non-trivial data structure invariants, red-black trees often appear as a benchmark for techniques that check expressive properties [31, 24, 28, 2]. Figure 1 shows the skeleton of a red-black tree implementation.

Specification of red-black trees. The class invariant for red-black trees is a conjunction of the following properties [11], denoted isRBT:

treeness: The nodes reachable from `root` along the `left` and `right` children should form a tree (have no cycles and no two incoming pointers to any node), and the `parent` field should appropriately point to a node's immediate predecessor.

```
class Tree {
  Node root; int size;
  static boolean RED = false, BLACK = true;
  static class Node {
    Node left, right, parent;
    boolean color; int key; }
  void insert(int value) { ... }
  void remove(int value) { ... }
  boolean contains(int value) { ... }
}
```

Fig. 1. A red-black tree implementation in Java

coloring: (1) The children of a node colored red must be colored black. (2) All simple paths from the `root` to a leaf must have the same number of black nodes.

ordering: The values in the tree should be ordered for binary search, i.e., for each node `n`, the values in the left subtree should be smaller than the value in `n`, and the values in the right subtree should be larger than the value in `n`.

We next consider the problem of finding counterexamples to the Hoare triple $\{\mathsf{isRBT}\}\mathsf{remove}(v)\{\mathsf{isRBT}\}$, stating that the remove method preserves the class invariant. We reduce this problem to exploring the executions of the nondeterministic program $\mathsf{havoc}(\boldsymbol{x}); \mathsf{assume}(\mathsf{isRBT}); \mathsf{remove}(v); \mathsf{assert}(\mathsf{isRBT})$, whose transition system semantics first changes the state arbitrarily, then filters out the states that do not satisfy isRBT, executes the remove method on them, and tests that the result still satisfies isRBT.

Executable predicates. In our approach, the developer can use arbitrary executable code to specify predicates, as they would do when using Java's assert statement. Figure 2 shows the `isRBT` Java method that returns `true` exactly when the isRBT predicate holds in the current state. Among the advantages of such executable specifications is that the developers need not learn a new specification language and the compiler can use standard techniques to optimize the code. Moreover, the developers can choose to hand-optimize the checks using property-specific algorithms, such as the work-list algorithm in Figure 2 that checks **treeness**. Alternatively, the developers can choose more compact notations such as JML [10] or functional languages [37] and compile them to Java bytecodes that specify the predicates.

Assume and assert. To specify the postcondition, the developer can use the usual Java `assert` statement, writing `assert(isRBT())` after `remove` in our example to ensure that the resulting state satisfies the class invariant. In our approach, the developer can also use the dual statement, writing e.g., `assume(isRBT())`. If the currently considered state does not satisfy the invariant, the system silently ignores the current execution and moves to other executions that need to be considered. This corresponds to the relational semantics of assume in the guarded command languages and is also familiar to the users of Java PathFinder, where it corresponds to invoking, in this example, `ignoreIf(!isRBT())`.

```
boolean isRBT() {
  return treeness() && coloring() && ordering();
}
boolean treeness() {
  if (root == null) return size == 0;
  Set<Node> visited = new java.util.HashSet<Node>(); visited.add(t.root);
  List<Node> workList = new java.util.LinkedList<Node>(); workList.add(t.root);
  if (root.parent != null) return false;
  while (!workList.isEmpty()) {
    Node current = workList.removeFirst();
    Node cl = current.left;
    if (cl != null) {
      if (!visited.add(cl)) return false;
      if (cl.parent != current) return false;
      workList.add(cl); }
    Node cr = current.right;
    if (cr != null) {
      if (!visited.add(cr)) return false;
      if (cr.parent != current) return false;
      workList.add(cr); }
  }
  return size == visited.size();
}
boolean coloring() {
  // Part (1): red node must have black children
  ...
  // Part (2): number of black nodes on all paths is the same
  int numberOfBlack = −1;
  List<Pair> workList = new java.util.LinkedList<Pair>();
  workList.add(new Pair(root, 0));
  while (!workList.isEmpty()) {
    Pair p = workList.removeFirst(); Node e = p.e; int n = p.n;
    if (e != null && e.color == BLACK) n++;
    if (e == null) {
      if (numberOfBlack == −1) numberOfBlack = n;
      else if (numberOfBlack != n) return false;
    } else {
      workList.add(new Pair(e.left, n)); workList.add(new Pair(e.right, n));
    }
  }
  return true;
}
boolean ordering() { ... }
```

Fig. 2. Red-black tree class invariant as an executable predicate

Non-deterministic initialization as havoc. It remains to describe how our approach models the `havoc` statements, whose semantics is to non-deterministically change the values to listed variables. For this purpose, we use non-deterministic assignments. An example is the statement k=`getInt`$(0, N-1)$, a version of which is already present in Java PathFinder. Its meaning can be specified as introducing N branches in a non-deterministic execution, where in

```
void havoc(int maxSize, int maxKey) {
  size = getInt(1, maxSize);
  ObjectPool<Node> nodes = new ObjectPool<Node>(size);
  root = nodes.getAny();
  for (Node n : nodes) {
    n.left = nodes.getAny(); n.right = nodes.getAny();
    n.parent = nodes.getAny();
    n.color = getBoolean();
    n.key = getInt(1, maxKey); }
}
```

Fig. 3. Method performing non-deterministic initialization of a red-black tree

```
Tree t = new Tree(); t.havoc(N, N); assume(t.isRBT());
int v = getInt(0, N); t.remove(v); assert(t.isRBT());
```

Fig. 4. Checking the Hoare triple for `remove`

branch i (for $0 \leq i \leq N-1$) the variable k has value i. Figure 3 shows the use of this approach in our red-black tree example. The non-deterministic initialization of a red-black tree data structure proceeds in several steps: 1) pick the tree size (the number of nodes); 2) create a pool of objects of this size; 3) iterate over all objects in the pool and non-deterministically initialize their fields to point to other objects in the pool. The `getAny` picks an arbitrary object in the pool and can be, in principle, implemented using `getInt`.

Summary of writing Hoare triples in our approach. Figure 4 shows how to model the problem of checking Hoare triple $\{\mathsf{isRBT}\}\mathsf{remove}(v)\{\mathsf{isRBT}\}$ for trees up to size N and containing values between 0 and N in our system.

Eager choice execution. We can use a straightforward implementation of `getInt` and `getAny` methods, which non-deterministically picks a concrete value and immediately returns it. This allows us to easily obtain a baseline implementation on top of Java PathFinder. However, the combinatorial explosion in `havoc` causes the baseline implementation to explicitly consider $N^{O(4N)}$ possibilities for a given N. In our experiments, Java PathFinder did not complete the search over all red-black trees of size $N = 3$ within 40 minutes.

Delayed choice execution. Our approach proposes the delayed choice execution strategy for Java PathFinder. This strategy takes exactly the same description of the Hoare triple from Figure 4 and generates the red-black trees of the same size $N = 3$ in 1.2 seconds. (Section 4 contains an experimental evaluation of our approach.) The key idea of delayed execution strategy is to delay the non-deterministic choices of values to the point where the values are used for the first time. Consequently, the order in which the values are used for the first time creates a dynamic ordering of the variables in the search space of counterexamples. This approach implements the essential idea of backtracking algorithms, also present in modern SAT and SMT solvers [8, 13, 5] and constraint logic programming [1]: when a constraint fails to hold (in our case, `assume(false)` is

executed), detect the reason for the conflict and use this information to change the current assignment to variables. In particular, if the reason for conflict involves only a given set of variables, then only these variables need to be considered as candidates for a change, even if the code executed `z = getInt(...)` assignments for some other variables `z`. In our approach, a variable is involved in a conflict if its value has been read in ways other than copying it to another variable.

```
RBT generateRBT(int N) {
  RBT t = new RBT(); t.root = generateTreeBackbone(N);
  generateColoring(t); // not shown, very complex
  generateOrdering(t); // not shown, fairly simple
  t.size = numberOfNodes(t); // not shown, trivial
  return t;
}
Node generateTreeBackbone(int N) {
  if (N == 0) return null;
  Node n = new Node();
  int leftSize = getInt(0, N − 1); int rightSize = getInt(0, N − 1 − leftSize);
  n.left = generateTreeBackbone(leftSize); if (n.left != null) n.left.parent = n;
  n.right = generateTreeBackbone(rightSize); if (n.right != null) n.right.parent = n;
  return n;
}
```

Fig. 5. Code that directly generates trees

Comparison to Korat algorithms. The approach of delayed execution has already proved extremely effective in the context of the specialized Korat tool that was used by the authors for test generation [7,29,32,30] and data structure repair [16], and has been reimplemented in academic and industrial settings [40,33].

Korat accepts as input a method such as `isRBT`, implicitly performs `havoc` commands, and searches for a structure on which the method returns `true`. To specify the search space, Korat requires a set of ad-hoc parameters, called finitization, that indicate the bounds of the search space. In this work, we have shown that Korat's algorithm can be deployed in a general explicit-state model checker such as Java PathFinder using the delayed choice idea. The algorithm continues to work correctly in this generalized settings, removing many unnecessary requirements on the form of problems solved by Korat. Our new approach provides the programmer with a great flexibility in specifying the scope of finite search, because finitization is not a special construct anymore: it instead becomes simply a piece of code that contains certain non-deterministic assignments. Most importantly, the programmers are now free to mix non-deterministic assignments (`havoc`) and `assume` statements in flexible ways, combining the benefits of *predicates* and *generators*.

Using generators to establish preconditions. Instead of generating all possible graphs in Figure 3 and then filtering those that are not trees using the `treeness` method in Figure 2, a simpler and faster alternative is to directly generate trees of size N. The `generateTreeBackbone` method in Figure 5 does precisely this. The previouly shown `treeness` method in Figure 2 presents a *predicate* characterizing trees, whereas Figure 5 presents a *generator* for trees. We call the former *declarative* approach (as it only specifies *what* the trees look like, although the specification is expressed in an imperative language), and we call the latter *imperative* approach (as it specifies *how* to generate trees) [12]. Writing preconditions using generators instead of predicates can dramatically speed up the execution. The correctness of such (manual) transformation can easily be expressed in our framework as a program transformation that preserves (up to isomorphism) the set of all generated structures.

However, using generators alone is highly non-trivial. Although it was easy to write code that generates *arbitrary* trees, generating only trees for which correct coloring exists is much more difficult. In fact, an entire research paper was devoted to such efficient generation of red-black trees [2]. In comparison, declarative generation is often easier, anecdotally confirmed by the fact that undergraduate students were able to write appropriate checks [30]. This trade-off justifies delayed choice execution as an optimization for predicate-based execution exploration, but also asks for approaches to combine generators and predicates.

Combining generators and predicates. Our approach makes combination of generators and predicates possible because they are both expressed in a unified framework: systematic execution of guarded commands. Consider the properties in our running example. For the **treeness** property, comparing the imperative generation (Figure 5) and declarative generation (figures 2 and 3), one could argue that it is easier to write a generator than a predicate. Our new approach allows the developer to combine, for example, a generator for **treeness** with a predicate for **coloring**. One would generate trees as in `generateTreeBackbone` method of Figure 5 and then find appropriate node colors using the **coloring** predicate in Figure 2.

Isomorphism avoidance. Our approach also supports avoiding the generation of structures that are isomorphic thanks to the abstract nature of Java references. In a naive approach to implementing **getAny** method for object pools, invoking the method N times and reading each of the invoked values would generate N^k possible k-tuples of values. A better implementation uses the fact that all fresh objects are observationally equivalent. It therefore returns either one of the previously returned objects or the first object from the pool that was not returned before. This avoids the isomorphism that follows from the fact that Java programs cannot observe numeric values of references, only their equality and the associated field values [7].

Copy propagation. We finally illustrate the copy propagation feature of our approach, which keeps non-deterministic values symbolic even if they are copied through memory locations. Consider first a version of red-black tree that, in addition to the `key` field also has a `value` field storing arbitrary objects. Because

```
void sort(int[] keys, int[] elems) {
  for (int i = 0; i < keys.length − 1; i++)
    for (int j = i+1; j < keys.length; j++)
      if (keys[i] > keys[j]) {
        int tmp = keys[i]; keys[i] = keys[j]; keys[j] = tmp;
        tmp = elems[i]; elems[i] = elems[j]; elems[j] = tmp; }
}
...
int length = getInt(0, N);
int[] keys = new int[length]; int[] values = new int[length];
for (int i = 0; i < length; i++) {
  keys[i] = getInt(0, N); values[i] = getInt(0, VAL−1); }
sort.sort(keys, values);
assert (sorted(keys));
...
```

Fig. 6. Checking code that sorts data stored in two arrays

nodes are stored according to `key` values, no `value` field in a tree will be read by `remove` operation. Therefore, even if the field is initialized with `value` values belonging to a large set, the search will terminate equally fast, proving that the initial `value` fields do not affect the correctness of the Hoare triple.

Consider, however, code shown in Figure 6 of sorting data stored in the `values` array, according to keys stored in the separate `keys` array. The correspondence between data and key is established by the position; whenever in-place sort moves keys, it also moves the corresponding values. Consequently, both `values` and `keys` entries are read by the code. The simple form of delayed execution would explore all $\texttt{VAL}^N$ possibilities for the `values` array. In contrast, our copy propagation technique keeps the values symbolic when they are copied, choosing concrete values only when the variable is involved in a non-copy operation, e.g., an arithmetic operation or field dereference. In this example, such non-copy operations do not arise for the elements of the `values` array. For $N = 5$, copy propagation therefore ensures that the exhaustive execution finishes in 4.8 seconds, regardless of the `VAL` bound. In contrast, exhaustive exploration without copy propagation times out for even moderate values of `VAL`.

3 Delayed Choice Execution Algorithm

We next make the core of our algorithm precise by presenting it as a source-to-source transformation of a simple non-deterministic language. Our formalization covers the essence of delayed execution with copy propagation.

3.1 Preliminaries

Program representation. Our language is a variant guarded-command language similar to representations such as, e.g., BoogiePL [3]. We represent the program as an abstract control-flow graph $\mathsf{CFG} \subseteq \mathsf{CS} \times \mathsf{ST} \times \mathsf{CS}$. (Structured

control statements such as 'if' and 'while' can be translated into control graphs using standard techniques.) The set CS denotes control states. For a program with a single thread and without procedure calls, CS is simply the finite set of program points. In general, CS can be an infinite set encoding program counters for all threads and the control stack. ST is the set of statements. Statements manipulate visible and invisible variables and may contain expressions.

The purpose of visible variables is to denote variables that are externally observable. In delayed execution, visible variables always contain concrete values, whereas invisible variables may also contain delayed computation. Expressions contain only visible variables and there are separate statements to load values from invisible variables and store them to invisible variables. If we included additional input/output statements into our language, their arguments would also contain only visible variables.

The statements take the following forms:

- $v = x$ (read), where v is a visible variable and x is an invisible variable;
- $x = e$ (write) where x is an invisible variable and e is an expression containing visible variables and constants;
- $x = y$ (copy) where x, y are invisible variables;
- $\mathsf{assume}(b)$, where b is a Boolean expression, behaves as no-op if b is true, and blocks execution if b is false. This statement corresponds to the 'assume' statement of the guarded command language. It also corresponds to `ignoreIf`$(\neg b)$ in Java PathFinder.
- $x{=}\mathsf{getInt}(a, b)$ is a non-deterministic write of an integer value from the set $\{a, \ldots, b\}$ to x. Here x is an invisible variable and a, b are expressions evaluated at the time $x{=}\mathsf{getInt}(a, b)$ is executed. The meaning of $x{=}\mathsf{getInt}(a, b)$ is the sequence of guarded-command language statements $\mathsf{havoc}(x)\,;\mathsf{assume}(a \leq x \leq b)$.

Program semantics. A program state (c, d) consists of the control state $c \in \mathsf{CS}$ and the data state $d \in \mathsf{DS}$. We let $\mathsf{TS} = \mathsf{CS} \times \mathsf{DS}$ denote the set of all states. A data state d is a function from variables to their values. A variable is either an invisible variable $x \in \mathsf{IV}$ or visible variable $v \in \mathsf{VV}$. Therefore, $d : (\mathsf{IV} \cup \mathsf{VV}) \to \mathsf{Vals}$.

Given a statement $s \in \mathsf{ST}$ we define its meaning $[\![s]\!] \subseteq \mathsf{DS}^2$ on program variables as follows ($f[p := q]$ is g such that $g(x) = f(x)$ for $x \neq p$ and $g(p) = q$):

- $[\![v = x]\!] = \{(d, d') \mid d'{=}d[v{:=}d(x)]\}$
- $[\![x = e]\!] = \{(d, d') \mid d'{=}d[x{:=}d(e)]\}$
- $[\![\mathsf{assume}(b)]\!] = \{(d, d) \mid d(b)\}$
- $[\![x{=}\mathsf{getInt}(a, b)]\!] = \{(d, d') \mid \exists u \in \{a, \ldots, b\}.\ d'{=}d[x{:=}u]\}$

We define transition relation $\mathsf{TR} \subseteq \mathsf{TS}^2$ in the natural way: $((c, d), (c', d')) \in \mathsf{TR}$ if and only if there exists $(c, s, c') \in E$ such that $(d, d') \in [\![s]\!]$.

We represent the meaning of a program as a computation tree $\mathsf{CT}(E, s_0) = (\mathsf{CTV}, \mathsf{CTE})$ where $\mathsf{CTV} \subseteq \mathsf{CS} \times \mathsf{DS}$ is a set of states and CTE is a set of edges on CTV that forms a tree. (In practice, the benefits of delayed execution remain

also in the presence of state comparison [42], when the computation is more appropriately represented as a graph instead of a tree.) Given an initial state $s_0 \in \mathsf{TS}$, we define the computation tree $\mathsf{CT}(E, s_0)$ as the pointwise-least pair of sets $(\mathsf{CTV}, \mathsf{CTE})$ such that

1. $s_0 \in \mathsf{CTV}$;
2. if $s \in \mathsf{CTV}$ and $(s, s') \in \mathsf{TR}$, then $(s, s') \in \mathsf{CTE}$ and $s' \in \mathsf{CTV}$.

An execution trace t is a sequence $s_0 s_1 \ldots$ of states starting from the root of the computation tree.

3.2 Delayed Execution as a Program Transformation

Our algorithm can be expressed as a program transformation on control-flow graphs that replaces certain statements (c, s, c') in the original graph with new statements $(c, \tau(s), c')$. The effect of the transformation is to postpone branching in the computation tree generated by the control-flow graph. The transformation changes the meaning of $x{=}\mathsf{getInt}(a, b)$ to store only a symbolic representation (a, b) of the possible values, which we denote by $x{=}\,\mathsf{Susp}(a, b)$. We use statement $\mathsf{force}(x)$ to denote making an actual non-deterministic choice of the stored symbolic value of x.

Figure 7 illustrates this transformation on an example of picking an ordered triple of elements x0 ≤ x1 ≤ x2 whose values are in the set $\{0, 1\}$. This small example already illustrates delayed execution. In general, when picking n elements from $\{0, 1\}$ and assuming that they are ordered, eager execution explores $2^{O(n)}$ paths, whereas delayed execution explores a polynomial number of paths.

```
x0 = Susp(0,1);
force(x0);
x1 = Susp(0,1);
force(x1);
x2 = Susp(0,1);
force(x2);
v0 = x0;
v1 = x1;
assume (v0 <= v1);
v2 = x2;
assume (v1 <= v2);
```

```
x0 = Susp(0,1);
x1 = Susp(0,1);
x2 = Susp(0,1);
force(x0);
v0 = x0;
force(x1);
v1 = x1;
assume (v0 <= v1);
force(x2);
v2 = x2;
assume (v1 <= v2);
```

Fig. 7. Eager Execution (Left) and Delayed Execution (Right) of a Program

We next describe the process of delaying the execution more precisely. We show that it preserves the projections of states onto visible variables. Our evaluation in Section 4 shows that the size of trees often decreases and sometimes dramatically so.

For each invisible variable $x \in \mathsf{IV}$, our transformation extends its domain so that it stores a pointer to a cell c where c stores either 1) a concrete value (as

before), or 2) an expression of the form $\mathsf{Susp}(a, b)$, denoting the set of values $\{x \mid a \leq x \leq b\}$ from which a concrete value may be chosen in the future. (A reference to $\mathsf{Susp}(a, b)$ corresponds to representations of delayed expressions in implementations of non-strict functional languages [21,22].) The environment d now maps not only variable names to concrete values, but also maps cells to suspended computations or concrete values. We define

$$\begin{array}{l}
[\![x{=}\,\mathsf{Susp}(a,b)]\!] = \{(d, d') \mid d' = d[x := c, c := \mathsf{Susp}(a,b)], \text{for } c \text{ cell fresh in } d\} \\
[\![v = !x]\!] = \{(d, d[v := d(d(x))]\} \\
[\![\mathsf{force}(x)]\!] = \{\, (d, d[c := u]) \mid d(x) = c \wedge \\
\qquad\qquad (d(c) = u \wedge u \in \mathsf{Vals}) \;\vee (d(c) = \mathsf{Susp}(a,b) \wedge a \leq u \leq b)\}
\end{array}$$

Note the use of shared cells to represent values of invisible variables. This ensures that the suspended computations are properly shared in the presence of copy statements for invisible variables, and preserves the set of computed results of eager non-deterministic evaluation [27].

Initial tree. We represent the initial, eager, computation tree by the following transformations of statements in the original program, which obviously preserve states up to visible variables:

$$\begin{array}{rcl}
x{=}\mathsf{getInt}(a,b) & \rightsquigarrow & x{=}\,\mathsf{Susp}(a,b)\,;\mathsf{force}(x) \\
v = x & \rightsquigarrow & v = !x
\end{array}$$

An invariant of executing such computation tree is that each execution of $v = !x$ is preceded (but not necessarily immediately preceded) with an execution of $\mathsf{force}(x)$. We maintain (and further strengthen) this invariant when transforming the tree to delayed form.

Delayed tree. In the delayed tree, we do the following replacements

$$\begin{array}{rcl}
x{=}\mathsf{getInt}(a,b) & \rightsquigarrow & x{=}\,\mathsf{Susp}(a,b) \\
v = x & \rightsquigarrow & \mathsf{force}(x)\,; v = !x
\end{array}$$

In summary, the difference between two trees is that $\mathsf{force}(x)$ *immediately follows* the specification of the non-deterministic choice in the original tree, but it *immediately precedes* the dereference $v = !x$ in the delayed tree.[5]

In both of the transformed trees, the semantics of copy statements has the same form as before: it copies references to cells, without forcing the non-deterministic choices stored in these cells. In the delayed tree, this achieves the effect of copy propagation.

3.3 Preservation of Visible Variables in Delayed Execution

Define α on program states as a function that projects data state onto visible variables

$$\alpha(c, d) = (c, d|_{\mathsf{VV}})$$

[5] Informally we could say that delayed execution *postpones the decisions if possible* and *uses force only when and if needed* (to preserve the values of visible variables).

We sketch an argument showing that the initial tree and the delayed tree preserve the values $\alpha(c, d)$ for reachable states (c, d).

We show this by transforming the initial tree into the delayed tree by gradually moving $\mathsf{force}(x)$ occurrences from the root towards the leaves of the tree, by swapping their executions with executions of other statements. In the notation below, if s_1 and s_2 are two statements, we will write $s_1\,;s_2$ for the part of the tree resulting from executing s_1 and then s_2.

Statements independent of x: If s is a statement that does not read or write x, we replace

$$\mathsf{force}(x)\,;s$$

with

$$s\,;\mathsf{force}(x)$$

This includes cases where s is $y{=}\,\mathsf{Susp}(a, b)$ for y a variable distinct from x. Because

$$[\![\mathsf{force}(x)]\!] \circ [\![s]\!] = [\![s]\!] \circ [\![\mathsf{force}(x)]\!]$$

the set of states at each level of the tree is preserved by such transformation.

Assignment to x: If s_x is a statement of that assigns to x (of the form $x = \ldots$), then we replace

$$\mathsf{force}(x)\,;s_x$$

with simply s_x, removing this occurrence of $\mathsf{force}(x)$ from the tree. This clearly preserves the set of reachable states after the assignment and preserves the value of α for the set of all nodes in the tree.

Copy of x: We replace

$$\mathsf{force}(x)\,;y = x$$

with

$$y = x;\mathsf{force}(y)$$

and continue the moving of the newly introduced occurrence of $\mathsf{force}(y)$.

Double occurrence of $\mathsf{force}(x)$: We replace a double occurrence of $\mathsf{force}(x)$ with a single one. Because $\mathsf{force}(x)$ is idempotent,

$$[\![\mathsf{force}(x)]\!] \circ [\![\mathsf{force}(x)]\!] \;=\; [\![\mathsf{force}(x)]\!]$$

this transformation does not affect the set of reachable states in the tree.

Read $v =!x$: When reaching the pattern

$$\mathsf{force}(x)\,;v =!x$$

the moving stops, reaching the pattern occurring in the delayed tree.

Summary and removing leaf choices. Each step of these transformations preserves the set of states at the level n of the tree for increasing value of n, and preserves the set of $\alpha(c, d)$ in the tree for all nodes in the tree at levels $i \leq n$. In the final step, we remove all occurrences of $\mathsf{force}(x)$ that lead to leaves of

the tree. This last transformation does not preserve the set of values of x in the state, but it does preserve the set of values of visible variables.

The result of such move of statements $\mathsf{force}(x)$ is almost identical to the delayed tree. The only difference can be additional $\mathsf{force}(x)$ statements that appear in the delayed tree because our simple description of delayed tree blindly inserts $\mathsf{force}(x)$ before each read of x. However, each of our transformations preserves the invariant that, whenever $v = !x$ is executed then $d(d(x))$ already contains a concrete value. Therefore, any such additional $\mathsf{force}(x)$ statements perform no state change and can be added or removed from the tree without effect.

This completes our argument and establishes that the sets of values $\alpha(c, d)$ for all reachable states (c, d) in the eager and the delayed tree are identical.

Theorem. Let $(\mathsf{CTV}, \mathsf{CTE})$ be the eager tree and $(\mathsf{CTV}', \mathsf{CTE}')$ the delayed tree. Then $\{\alpha(p) \mid p \in \mathsf{CTV}\} = \{\alpha(p') \mid p' \in \mathsf{CTV}'\}$.

3.4 Reduction in the Number of Paths

The reasoning in the previous subsection also shows why the number of traces in the delayed tree is no larger than the number of traces in the eager tree, and can be substantially smaller. It suffices to notice that all of the above transformations that move $\mathsf{force}(x)$ downwards either preserve or reduce the number of traces (paths from root to leaf) of the tree. In practice, the most important reduction results at the point of removing $\mathsf{force}(x)$ statements that lead to the leaves of the execution tree. For example, if the transformation removes K statements of the form $\mathsf{force}(x_i)$, and each x_i points to a distinct non-deterministic choice expression $\mathsf{Susp}(0, N-1)$, then the transformation reduces the number of paths N^K times.

3.5 Generating Linked Structures

Figure 8 presents a Java-like pseudo code for an implementation of object pools with a **getAny** method that avoids isomorphic structures. The use of **getInt** inside the implementation corresponds to its *eager* version. Our system obtains benefits from delayed choice execution on linked structures by encapsulating values returned by **getAny** into suspensions, similar as for **getInt**. The suspension for **getAny** maintains a reference to the underlying ObjectPool object. An important property of such implementation (established in [29]) is that it avoids linked structures isomorphic up to reference identities [20, 7].

4 Evaluation

We implemented our delayed choice algorithm by modifying Java PathFinder using its attribute mechanism [35] to store non-deterministic values that have not been read yet and by modifying the implementation of **getInt** to generate such delayed values. The implementation of object pool is similar.

```
class ObjectPool<T> {
  T[] entries; int next; boolean full;
  ObjectPool(int size) { // precondition: 0 < size
    entries = new T[size];
    next = 0; full = false; }
  T getAny() { // class invariant: next < entries.length
    int i = getInt(0, next); // eagerly get at most one new object
    if ((i == next) && !full) {
      entries[i] = new T();
      if (next < entries.length − 1) next++;
      else full = true;
    }
    return entries[i]; }
}
```

Fig. 8. Implementation of object pools and **getAny**

			Java PathFinder Baseline		Delayed Choice	
program	size	structures	time [s]	explored	time [s]	explored
RedBlackTree	7	35	9.96	54,912	3.24	16,983
	8	64	65.67	366,080	13.85	80,470
	9	122	449.17	2,489,344	64.24	381,470
DAG	3	34	5.68	4,802	0.69	321
	4	2,352	out of mem	-	6.41	21,196
	5	769,894	-	-	1,013.75	4,997,210
HeapArray	6	13,139	16.66	160,132	4.12	27,664
	7	117,562	304.32	2,739,136	32.43	227,494
	8	1,005,075	8,166.77	54,481,005	318.59	2,325,069
NQueens	5	10	1.4	3,125	0.10	177
	6	4	7.68	46,656	0.24	746
	7	40	82.35	823,543	0.51	3,073
	8	92	out of mem	-	2.29	13,756
SearchTree	4	490	1.36	3,584	0.63	1,484
	5	5,292	15.87	131,250	3.23	21,210
	6	60,984	675.33	6,158,592	40.24	305,052
SortedList	6	924	5.94	5,5987	0.64	3,967
	7	3,432	900.67	960,800	2.38	18,026
	8	12,870	1,865.55	19,173,961	9.85	80,089

Fig. 9. Enumeration of structures satisfying their invariants

We present an evaluation of our approach using a variety of data structure implementations: `RedBlackTree` is the example introduced in Section 2; `DAG` represents directed acyclic graphs; `HeapArray` is an array-based implementation of the heap data structure; `SearchTree` is binary search tree; and `SortedList` is a doubly-linked list containing sorted elements. Additionally, `NQueens` is the traditional problem from constraint solving [1]. For each structure, we wrote its representation invariant using our combined generator/predicate approach. Our experimental setup compares standard execution of Java PathFinder with our delayed choice execution using the same invariant. We turn off state hashing of Java PathFinder in our experiments, because duplicate states rarely arise in executions of our examples. We perform two kinds of experiments: (1) enumerating all structures of a given size, and (2) finding errors in code.

		Java PathFinder Baseline		Delayed Choice	
program	size	time [s]	explored	time [s]	explored
RedBlackTree BUG1	6	7.76	3,478	1.29	1,514
	7	7.00	21,706	2.62	7,169
	8	37.31	157,834	9.38	38,457
RedBlackTree BUG2	6	4.48	5,548	2.30	2,196
	7	10.33	31,787	3.97	9,454
	8	41.54	188,384	11.75	42,997
RedBlackTree BUG3	6	2.97	3,478	1.15	1,514
	7	6.51	21,555	2.36	6,960
	8	28.85	138,853	6.95	32,667
RedBlackTree BUG4	6	3.51	3,451	1.14	1,452
	7	6.33	21,437	2.09	6,807
	8	28.75	138,863	6.76	32,667

Fig. 10. Time to first counterexample

Figure 9 shows the efficiency of our approach for structure enumeration. For each program and several bounds, we tabulate the total number of successful paths in the execution tree (the number of valid structures generated), the exploration time, and the total number of paths explored. Both techniques generate the same number of structures, but delayed choice explores fewer paths which provides significant speed-ups, from 2x up to 190x as size increases.

Figure 10 shows the effectiveness of our approach for finding errors in code. Bugs of omission were manually inserted into the implementation of `RedBlackTree` by a student not familiar with our work. The "explored" column shows the number of candidate structures explored until the bug is hit. Delayed choice execution once again outperforms standard JPF execution, by 3x-4x.

5 Related Work

Techniques similar to delayed choice execution are common in constraint solving—both for constraints written in imperative languages and for constraints written in declarative languages. For example, Korat [7] implicitly uses delayed choice by monitoring field accesses and using them in field initializations for the new candidates it explores. Generalized symbolic execution [24] follows Korat and uses "lazy initialization" to make non-deterministic field assignments on first-access. Deng et al.'s [14] "lazier initialization" builds on generalized symbolic execution and makes non-deterministic field assignments on first-use. A key difference between previous work and this paper is that we provide a generic framework that supports delayed choice execution for arbitrary guarded commands and provides fully automatic execution using the Java PathFinder model checker [41].

Our unification of generators and predicates generalizes Korat's finitization, which has a very specific purpose: to define a bound on the input space of the predicate that represents the constraint to solve. While the users may still write standard Korat finitizations, the unification enables them to seamlessly combine dedicated generators with imperative predicates, which enables easier formula-

tion of constraints, faster generation of solutions, as well as focused generation of a desired subset of solutions.

Delayed choice execution is a lightweight form of symbolic execution [25]. Delayed choice execution does not explicitly build path conditions or examine them for feasibility using decision procedures. Several recent approaches to systematic testing use symbolic execution in some form, often by combining it with concrete executions [18, 39, 9, 34]. Delayed choice execution, which explores only concrete executions but delays the assignment of values to fields until they are used, is complementary to these approaches and can be used to optimize them.

Concrete executions have also been used with predicate abstraction [36] as well as generation of proofs [6]. In contrast, our approach focuses on efficiently covering a specified set of concrete executions, without attempting to perform approximation or generating proofs. Note that a natural consequence of delayed choice execution is that if the code does not "force" the assignment of a concrete value to a variable (the value remains suspended), the exploration proves that the execution would produce the same result for all values of that variable (even for values outside the range specified by the user).

Researchers have identified non-deterministic call-by-need lambda calculus as a useful programming model [27]. Our delayed choice execution employed call-by-need execution for side-effect-free choice expressions, thereby incorporating non-deterministic call-by-need execution into an imperative language.

The Eclipse constraint solver [1] provides a constraint logic programming (CLP) interface for writing declarative constraints. Eclipse provides *suspensions* that delay testing of predicates until more information is available. We believe adapting techniques from CLP holds much promise for imperative constraint solving, as witnessed by our encouraging results in implementing a form of suspensions in Java PathFinder. Moreover, we believe that the non-deterministic imperative programming paradigm that we advocate is among the most likely vehicles to incorporate advanced declarative constructs into modern languages.

6 Conclusions

We have shown that Hoare triple falsification and test generation can be naturally expressed as exploration of guarded command language executions. Moreover, we have shown how to use the concept of delayed execution to turn explicit-state model checker into an engine that efficiently explores such executions. We found the resulting system to be extremely effective in detecting bugs in specifications and code. We expect our experience to encourage further study of connections between systematic executions of imperative programs and declarative constraint solving.

Acknowledgements. We would like to thank Igor Andjelkovic for creating faulty versions of the Red Black Tree example.

References

1. K. Apt and M. G. Wallace. *Constraint Logic Programming using Eclipse*. CUP, 2006.

2. Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee J. White. State generation and automated class testing. *STVR*, 10(3), 2000.
3. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.
4. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
5. Clark Barrett and Cesare Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, 2007.
6. Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *ISSTA*, 2008.
7. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.
8. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4SMT solver. In *CAV*, 2008.
9. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.
10. Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003.
11. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.
12. Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC/FSE*, 2007.
13. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
14. Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE*, 2006.
15. Greg Dennis, Felix Chang, and Daniel Jackson. Modular verification of code with SAT. In *ISSTA*, 2006.
16. Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *ASE*, 2007.
17. Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *PLDI*, 2002.
18. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, 2005.
19. Dominik Haneberg, Gerhard Schellhorn, Holger Grandy, and Wolfgang Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Asp. Comput.*, 20(1):41–59, 2008.
20. Radu Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of the SPIN Workshop on Software Model Checking (SPIN)*, volume 2318 of *LNCS*, pages 22–41, July 2002.
21. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
22. Simon L Peyton Jones and David R Lester. Implementing functional languages: A tutorial, 2000.
23. Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.
24. Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.

25. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
26. Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
27. Arne Kutzner and Manfred Schmidt-Schauß. A non-deterministic call-by-need lambda calculus. In *ICFP*, 1998.
28. Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, 2007.
29. Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005.
30. Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.
31. Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.
32. A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *ICSE*, 2007.
33. S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with Korat. In *ESEC/FSE*, 2007.
34. Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, pages 15–26, 2008.
35. C.S. Pasareanu, P.C. Mehlitz, D.H. Bushnell, K. Gundy-Burlet, M.R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.
36. C.S. Pasareanu, R. Pelánek, and W. Visser. Predicate abstraction with under-approximation refinement. *Logical Methods in Comp. Sci.*, 3(1), 2007.
37. The Scala programing language. http://www.scala-lang.org. Last accessed October 2008.
38. P.H. Schmitt and I. Tonin. Verifying the Mondex case study. In *SEFM*, 2007.
39. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, 2005.
40. Keith Stobie. Model based testing in practice at Microsoft. *Electr. Notes Theor. Comput. Sci.*, 111:5–12, 2005.
41. Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
42. Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.
43. Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.