On Test Generation through Programming in UDITA

EPFL Technical Report LARA-REPORT-005, September 2009

Milos Gligoric
University of Illinois
gliga@illinois.edu

Tihomir Gvero
EPFL, Switzerland
tihomir.gvero@epfl.ch

Vilas Jagannath
University of Illinois
vbangal2@illinois.edu

Sarfraz Khurshid
University of Texas, Austin
khurshid@ece.utexas.edu

Viktor Kuncak
EPFL, Switzerland
viktor.kuncak@epfl.ch

Darko Marinov
University of Illinois
marinov@illinois.edu

ABSTRACT

We present an approach for describing tests using non-deterministic *test generation programs*. To write test generation programs, we introduce UDITA, a Java-based language with non-deterministic choice operators and an interface for generating linked structures. We also describe new algorithms that generate concrete tests by efficiently exploring the space of all executions of non-deterministic UDITA programs.

We implemented our approach and incorporated it into the official, publicly available repository of Java PathFinder (JPF), a popular tool for verifying Java programs. We evaluate our technique by generating tests for data structures, refactoring engines, and JPF itself. Our experiments show that test generation using UDITA is faster and leads to test descriptions that are easier to write than in previous frameworks. Moreover, the novel execution mechanism of UDITA is essential for making test generation feasible. Using UDITA, we have discovered a number of previously unknown bugs in Eclipse, NetBeans, Sun javac, and JPF.

1. INTRODUCTION

Testing is the most widely used method for detecting software errors in industry, and the importance of testing is growing as the consequences of software errors become more severe. Testing tools such as JUnit are popular as they automate text *execution*. However, widely adopted tools offer little support for test *generation*. The developers and testers often have good intuition [18, 33] to determine what tests should be generated but must manually translate this intuition into actual tests. Such manual test generation is time-consuming and results in test suites that have poor quality and are difficult to reuse. This is especially the case for code that requires structurally complex test inputs, for example code that operates on programs (e.g., compilers, interpreters, model checkers, or refactoring engines) or on complex data structures (e.g., container libraries).

Consider, for example, testing a feature in a Java compiler. In this case, test inputs are representations of programs. Our intuition could be that we can expose bugs in the feature under test using as inputs, say, Java programs with complex *inheritance graphs* (consisting of classes and interfaces). The expected properties of the Java inheritance graphs are: there are no directed cycles, all parents of interfaces are interfaces, and each class has at most one parent class. It is tedious and error-prone to manually generate tests that cover many "corner cases" for inheritance graphs: a test with only one class, with a subclass and a superclass, with two classes not related by inheritance, with an interface, with multiple inheritance through interfaces, with multiple paths to a common ancestor, with abstract classes, and so on.

Recent automated techniques aim to handle such corner cases using *systematic* test generation based on specifications [7, 36], or on symbolic execution [13, 38] and its hybrids with concrete executions [11, 17, 21, 31, 37, 47, 49, 52]. Modern (hybrid) symbolic execution techniques can handle advanced constructs of object-oriented programs. However, feasible application of these techniques is largely limited to testing units of code much smaller than hundred thousand lines, or generating inputs values much simpler than representations of Java programs. The inherent requirement for not only building *path conditions*, albeit with partial constraints, but also determining their feasibility poses a key challenge for scaling to structurally complex inputs and entire systems. Handling programs of the *complexity of a compiler* remains challenging for current systematic approaches. Our insight is that systematic approaches can be made to scale by providing a framework that allows testers to utilize their domain knowledge.

We propose a new approach to generate a large number of complex test inputs by allowing the tester to write a *test generation program* in UDITA, a Java-based language with non-deterministic choices, including choices used to generate linked data structures. Each execution of a test generation program generates one test input. Our execution engine systematically explores all executions to generate inputs for *bounded-exhaustive testing* [42, 50] that validates the code under test for *all test inputs* within a given bound (e.g., all inheritance graphs with up to N nodes). UDITA thus enables testers to avoid manual test generation. However, our approach does not attempt to fully automatically identify tests [11, 31], because such approaches do not provide much control to the tester to encode their intuition. Instead, we provide testers with an expressive language in which they have sufficient control to define the space of desired tests.

This paper makes the following contributions:

1) New language for describing tests: We present UDITA, a language that enhances Java with two important extensions. The first extension are *non-deterministic choice* commands, and the *assume* command that (partially) restricts these choices. These constructs are familiar to users of model checkers such as Java PathFinder (JPF) [54]. Thanks to the built-in non-determinism, writing a test generation program (from which *many* test inputs can be generated) is often as simple as writing Java code that generates *one* particular test input. The second extension is the *object pool* abstraction that allows the tester to control generation of linked structures with any desired sharing patterns, including trees but also DAGs, cyclic graphs, and domain-specific data structures. Due to its expressive power, UDITA enables testers to write test generation programs using any desirable mixture of declarative [7, 22, 29, 36, 41, 42] and imperative [19, 35] styles, whereas previous systems required the use of only one style.

2) New test generation algorithms: We present efficient techniques for test generation by systematic execution of non-deterministic programs. Our techniques build on systematic exploration performed by explicit-state model checkers to obtain the effect of bounded-exhaustive testing [42, 50].

The efficiency of our techniques is based on a general principle of *delayed choice* [44], i.e., lazy non-deterministic evaluation [23]. The basic delayed choice technique postpones the choices for each variable until it is first accessed. The more advanced *copy propagation* technique further postpones the choices even if the values are being copied. Like lazy evaluation, our techniques guarantee that each non-deterministic choice is executed at most once.

Our techniques support primitive fields, but are particularly well-suited for linked structures (Section 4.2). The techniques use a new object pool interface. We postpone the choice of object identity until object's first non-copy use, reducing the amount of search. Furthermore, we avoid isomorphic structures [34, 41] which gives another source of exponential performance improvement. Finally, to determine the feasibility of symbolic fresh-object constraints in the current path, we use a new polynomial-time algorithm (figures 11 and 12), which is in contrast to NP-hard path condition constraints of symbolic execution [38, 47].

3) Implementation: We describe an implementation of UDITA and of our optimizations on top of JPF [54], a popular model checker for Java. Our code is publicly available as an extension (called `delayed`) in the JPF repository:

http://javapathfinder.sourceforge.net

4) Evaluation: We have performed several sets of experiments to evaluate UDITA. The *first set* of experiments, on six data structures show that **our optimizations improve the time** to generate test inputs up to a given bound and the time to find the first fault.

The *second set* of experiments compared UDITA with Pex [52], a state-of-the-art testing tool based on symbolic execution. Our results found that object pools are a powerful abstraction for guiding exploration, orthogonal to the path-bounding approaches used by tools such as Pex. In particular, even a naive implementation of object pools helped Pex enumerate structures and **find bugs faster**.

The *third set* of experiments is on testing refactoring engines, which are software development tools that take as input program source code and refactor (transform) it to change its design without changing its behavior [25, 45].

```
class IG {
  Node[] nodes; int size;
  static class Node {
    Node[] supertypes;
    boolean isClass; } }
```

Fig. 1: A simple representation of inheritance graph

Modern IDEs such as Eclipse or NetBeans include refactoring engines for Java. A key challenge in testing refactoring engines is generating input programs. Figures 5 and 6 show some example programs with multiple inheritance that revealed bugs in Eclipse. To generate such programs, we need to both "generate inheritance graphs" and "add methods" in the classes and interfaces in the graphs. Our experience with UDITA's combined declarative/imperative style shows that, compared to our prior approach, ASTGen [19, 35], UDITA is **more expressive**, resulting in **shorter** (and easier to write) test generation programs, and sometimes provides **faster generation** despite executing on the significantly slower JPF virtual machine. Through these experiments, we **revealed new bugs** in Eclipse and NetBeans, and even a bug in the Sun Java compiler.

The *fourth set* of experiments, on testing parts of the UDITA implementation, revealed several **new bugs in JPF**, and one bug in our JPF extension that we subsequently corrected. These results suggest that UDITA is effective in helping detect real bugs in large code bases.

Our experimental results are publicly available at:

http://mir.cs.illinois.edu/udita

2. EXAMPLE

To illustrate UDITA and its key strengths, we consider generation of inheritance graphs for Java programs. We have found such generation effective for testing real-world applications including compilers, interpreters, model checkers, and refactoring engines (Section 5). The example illustrates the ability of UDITA to describe data structures with non-trivial invariants. Figure 1 shows a simple representation of inheritance graphs in Java. A graph has several nodes. Each node is either a class or an interface, and has zero or more supertypes that are themselves classes or interfaces.

Specification of inheritance graphs. Each inheritance graph needs to satisfy the following two properties:

1) **DAG** (directed acyclic graph): The nodes in the graph should have no directed cycle along the references in `supertypes`.

2) **JavaInheritance**: All supertypes of an interface are interfaces, and each class has at most one supertype class.

UDITA allows the tester to express these properties using full-fledged Java code extended with non-deterministic choices. Testers describe properties in UDITA using any desired mix of *declarative* and *imperative* style. In a purely declarative style, embodied in techniques such as TestEra [36] and Korat [7, 22, 29, 41, 42], the tester writes the *predicates*—*what* the test inputs should satisfy; then the *tool searches* for valid tests. In contrast, in purely imperative style, embodied in techniques such as ASTGen [19, 35], the tester directly writes *generators*—*how* to generate valid inputs; then *the tool executes* these generators to generate the inputs. We first present these two pure approaches, then discuss how UDITA allows freely combining them, and finally how UDITA efficiently generates inputs.

```
boolean isDAG(IG ig) {
  Set<Node> visited = new HashSet<Node>();
  Set<Node> path = new HashSet<Node>();
  if (ig.nodes == null || ig.size != ig.nodes.length) return false;
  for (Node n : ig.nodes)
    if (!visited.contains(n))
      if (!isInCycle(n, path, visited)) return false;
  return true; }
boolean isInCycle(Node node,
    Set<Node> path, Set<Node> visited) {
  if (path.contains(node)) return false;
  path.add(node); visited.add(node);
  for (int i = 0; i < node.supertypes.length; i++) {
    Node s = node.supertypes[i];
    // two supertypes cannot be the same
    for (int j = 0; j < i; j++)
      if (s == node.supertypes[j]) return false;
    // check property for every supertype of this node
    if (!isInCycle(s, path, visited)) return false;
  }
  path.remove(node);
  return true; }
boolean isJavaInheritance(IG ig) {
  for (Node n : ig.nodes) {
    boolean doesExtend = false;
    for (Node s : n.supertypes)
      if (s.isClass) {
        // interface must not extend any class
        if (!n.isClass) return false;
        if (!doesExtend) { doesExtend = true;
        // class must not extend more than one class
        } else { return false; }
      } }
  return true; }
```

Fig. 2: Declarative approach for inheritance graphs

```
IG initialize(int N) {
  IG ig = new IG(); ig.size = N;
  ObjectPool⟨Node⟩ pool = new ObjectPool⟨Node⟩(N);
  ig.nodes = new Node[N];
  for (int i = 0; i < N; i++) ig.nodes[i] = pool.getNew();
  for (Node n : ig.nodes) {
    // next 3 lines unnecessary when using generateDAGBackbone
    int num = getInt(0, N − 1);
    n.supertypes = new Node[num];
    for (int j = 0; j < num; j++) n.supertypes[j] = pool.getAny();
    // next line unnecessary when using generateJavaInheritance
    n.isClass = getBoolean(); }
  return ig; }
```

```
void mainDecl(int N) {
  IG ig = initialize(N);
  assume(isDAG(ig));
  assume(isJavaInheritance(ig));
  println(ig); }
```

```
void mainImp(int N) {
  IG ig = initialize(N);
  generateDAGBackbone(ig);
  generateJavaInheritance(ig);
  println(ig); }
```

Fig. 3: Examples of bounded-exhaustive generation

Declarative approach. Figure 2 shows Java predicates that return `true` when the above inheritance graph properties hold. To generate *all* test inputs from predicates, the tester needs to specify bounds on possible values for input elements, which in our example are the nodes, array sizes, and `isClass` fields. For this purpose, UDITA uses *non-deterministic choices*. JPF already has choices for primitive values. For example, the assignment `k=getInt(1, N)` introduces N branches in a non-deterministic execution, where in branch i (for $1 \leq i \leq N$) the variable `k` has value i. JPF can systematically explore all (combinations) of non-deterministic choices. UDITA additionally provides non-deterministic choices for pointers/objects through the notion of *object pools* (described in detail in Section 4.2). Figure 3 shows the non-deterministic initialization of an inheritance graph data structure. The method `initialize` proceeds in several steps: (1) sets the graph size (the number of nodes), (2) creates a *pool* of `Node` objects of this size, and (3) iterates over all objects in the pool to non-deterministically initialize their `supertypes` to point to other objects in the pool. The `getNew` and `getAny` methods pick a fresh object and an arbitrary object from the pool, respectively. Running `mainDecl` on UDITA generates all inheritance graphs of size N.

```
void generateDAGBackbone(IG ig) {
  for (int i = 0; i < ig.nodes.length; i++) {
    int num = getInt(0, i); // pick number of supertypes
    ig.nodes[i].supertypes = new Node[num];
    for (int j = 0, k = −1; j < num; j++) {
      k = getInt(k + 1, i − (num − j));
      // supertypes of "i" can be only those "k" generated before
      ig.nodes[i].supertypes[j] = ig.nodes[k];
    } } }
void generateJavaInheritance(IG ig) {
// not shown imperatively because it is complex:
// topologically sorts "ig" to find what nodes can be classes or interfaces
}
```

Fig. 4: Imperative approach for inheritance graphs

Imperative approach. Instead of generating possible graphs and then filtering those that are not inheritance graphs, Figure 4 shows an alternative that directly generates DAGs of size N with the `generateDAGBackbone` method. We say that Figure 4 presents a *generator* for DAGs, which is in contrast to the *predicate* `isDAG` in Figure 2. The generator establishes *by construction* that there are no direct cycles (because `supertypes` of a node `i` can only be nodes `k` that were generated before `i`).

Writing (imperative) generators instead of (declarative) predicates can dramatically speed up the generation. However, using generators alone is highly non-trivial. Although it was relatively easy to write code that generates all *arbitrary* DAGs, it is non-trivial to eliminate isomorphic graphs (Section 4.2) or to properly label nodes as classes and interfaces (with `generateJavaInheritance`). Properties of other data structures can be even harder to express imperatively. For example, an entire research paper was devoted to efficient generation of red-black trees [5]. In comparison, declarative generation is often easier, anecdotally confirmed by the fact that undergraduate students are able to write appropriate checks [42]. This trade-off justifies the need for optimized execution for predicate-based exploration, but also asks for an approach to combine predicates and generators.

Unifying predicates and generators. UDITA makes combination of predicates and generators possible because they are both expressed in a unified framework: systematic execution of non-deterministic choices. Consider the properties in our running example. For the **DAG** property, comparing the imperative (Figure 4) and declarative generation (Figure 2), one could argue it is easier to write a generator than a predicate. However, for the **JavaInheritance** property, it is much easier to write a predicate than a generator. UDITA allows the tester to combine, for example, a generator for **DAG** with a predicate for **JavaInheritance**: write a new `main` method that invokes `generateDAGBackbone` followed by `assume(isJavaInheritance)`.

Test generation. After the tester writes some predicates and/or generators, it is necessary to execute them to gen-

```
class A implements B {
  public A m() {
    A a = null;
    return a; } }
interface B extends C {
  public B m(); }
interface C {
  public C m(); }
```

```
class A implements B {
  public C m() { // bug
    C a = null;
    return a; } }
interface B extends C {
  public B m(); }
interface C {
  public C m(); }
```

Fig. 5: IntroduceSuperType bug in Eclipse: when the refactoring is applied on `A`, the return type of `A.m` is incorrectly changed to `C` instead of displaying a warning or suggesting changing the return type to `B`

```
import java.util.List;
class A implements B, D {
  public List m(){
    List l=null;
    A a=null;
    l.add(a.m());
    return l; } }
interface D {
  public List m(); }
interface B extends C {
  public List m(); }
interface C {
  public List m(); }
```

```
import java.util.List;
class A implements B, D {
  public List<List> m() {
    List<List<List>> l=null; //bug
    A a=null;
    l.add(a.m());
    return l; } }
interface D {
  public List<List> m(); }
interface B extends C {
  public List<List> m(); }
interface C {
  public List<List> m(); }
```

Fig. 6: InferGenericType bug in Eclipse: when the refactoring is applied on the input program (left), Eclipse incorrectly infers the type of `A.m.l` as `List<List<List>>`, which does not match the return type of `A.m`

erate the tests. JPF already provides an execution engine for `getInt` and `getBoolean` non-deterministic choices, and a naive implementation of the object pool's `getNew` and `getAny` choices (whose use is shown in Figure 3) can be simply done with `getInt` (as discussed in Section 4.2). However, these default implementations, which we call *eager* as they immediately return a value, result in a combinatorial explosion, e.g., `mainDecl` from Figure 3 for $N = 4$ does not terminate in over an hour!

We provide more efficient implementations, which we call *delayed* as they postpone choices of primitive values (`getInt` and `getBoolean`) and additionally optimize exploration for object pools (`getAny` and `getNew`). For example, `mainDecl` from Figure 3 for $N = 4$ terminates in just 5.5 seconds with our delayed choice. Imperative generation can be even faster than declarative search. Section 5.1 shows our experimental results for data structures. We evaluate mostly the combined declarative/imperative style, since test programs are much easier to write than for purely imperative style, and generation for purely declarative style is several orders of magnitude slower on basic JPF without delayed choice.

Section 5.3 shows our results for testing refactoring engines, where we built on the inheritance graph generator to produce Java programs as inputs to refactoring engines. Figures 5 and 6 show two example input programs, generated by UDITA, which found bugs in Eclipse, specifically in the InferGenericType and IntroduceSuperType refactorings.

3. UDITA FRAMEWORK OVERVIEW

UDITA provides a framework for building a library of generic, reusable, and composable generators. The key aspects of the UDITA framework are: (1) the basic library for primitive values and objects; (2) the ability to encapsulate UDITA generators into reusable components using interfaces; and (3) the ability to compose these components.

```
class ObjectPool⟨T⟩ {
  public ObjectPool⟨T⟩(int size, boolean includeNull) { ... }
  public T getAny() { ... }
  public T getNew() { ... } }
```

Fig. 7: Basic operations for object pools

```
interface IGenerator⟨T⟩ { T generate(); }
class IntGenerator implements IGenerator⟨Integer⟩ {
  int lo, hi;
  IntGenerator(int lo, int hi) { this.lo = lo; this.hi = hi; }
  Integer generate() { return getInt(lo, hi); } }
class IGGenerator implements IGenerator⟨IG⟩ {
  IG ig;
  IGGenerator(int N) { ig = initialize(N); }
  IG generate() {
    assume(isDAG(ig) && isJavaInheritance(ig)); return ig; } }
class PairGenerator⟨L, R⟩ implements IGenerator⟨Pair⟨L, R⟩⟩ {
  IGenerator⟨L⟩ lg; IGenerator⟨R⟩ rg;
  PairGenerator(IGenerator⟨L⟩ lg, IGenerator⟨R rg) { ... }
  Pair⟨L, R⟩ generate() {
    return new Pair⟨L, R⟩(lg.generate(), rg.generate()); } }
```

Fig. 8: UDITA interface for generators and some example generators

Basic Library. The basic library for UDITA borrows from JPF non-deterministic choices for primitive values. For example, `getInt(int lo, int hi)` returns an integer between `lo` and `hi`, inclusively; and `getBoolean()` returns a boolean value. UDITA also provides *object pools* for non-deterministic choices of objects. Figure 7 shows the interface for object pools. The constructor can create finite (if `size > 0`) or infinite (if `size < 0`) pools, which may or may not include the value `null`. The method `getAny` non-deterministically returns any value from the pool (including optionally `null`), whereas `getNew` returns an object that was not returned by previous calls (and never `null`). Section 4.2 describes the implementation of these operations.

Generator Interface. UDITA provides `IGenerator` interface for encapsulating generators, as shown in Figure 8. The only method, `generate`, produces *one* object of the generic type `T`. During the execution on JPF, this method will be *systematically* explored for all non-deterministic choices, and will generate *many* objects of the type `T`. The figure also shows an example `IntGenerator` for integer values and an example `IGGenerator` that encapsulates declarative style predicates (`isDAG` and `isJavaInheritance`).

The design of UDITA generators is influenced by ASTGen [19] (which provides Java generators for abstract syntax trees for testing refactoring engines) and QuickCheck [12] (which provides a Haskell framework for generators). UDITA provides a much simpler interface than ASTGen: instead of one method, the basic `IGenerator` for ASTGen has *five methods* [19, Sec. 3.2]. The cause of that complexity was that ASTGen runs on a regular JVM; to obtain bounded-exhaustive generation, the implementor of the interface must manually manipulate the generator state (to reset it, advance it, store/restore it). In contrast, UDITA runs on JPF, where state manipulations come for free because JPF implements state exploration. Compared to QuickCheck [12], which supports only random generation, UDITA focuses on bounded-exhaustive generation, obtaining random generation for free as one of the possible JPF exploration strategies (additional strategies include depth-first and breadth-first).

3.1 Composing Generators

An important feature of generator frameworks such as ASTGen, QuickCheck, or UDITA is to allow reuse and composition of basic generators into more complex generators [12, 19]. UDITA again provides a substantially simpler solution than ASTGen, since UDITA runs on JPF which can systematically explore all combinations of values from various non-deterministic choices. Figure 8 shows an example generator that produces pairs of values based on generators for left and right elements of pairs. Note that the `generate` method of `PairGenerator` has only one line of code. In contrast, the corresponding ASTGen generator has *ten lines of code* [19, Sec. 3.3]. The reason is, again, that ASTGen needs to explicitly iterate over possible values to produce their combinations for bounded-exhaustive generation. QuickCheck provides composition through higher-order functional combinators [12] but is designed for the purely functional language Haskell, and does not have support for generating non-isomorphic graph structures. Neither ASTGen nor QuickCheck (nor any other framework that we are aware of) support such unification of declarative and imperative styles as UDITA.

4. TEST GENERATION IN UDITA

We next describe our test generation algorithms, which rely on the notion of delayed (lazy) execution of non-deterministic choices.

4.1 Test Generation for Primitive Values

Eager choice execution. We could, in principle, use a straightforward implementation of `getInt` that *immediately* chooses a concrete value and returns it. When the execution backtracks, the implementation picks a different value. This approach allows us to easily obtain a baseline implementation on top of JPF. Unfortunately, the combinatorial explosion in typical test generation programs (e.g., the `initialize` method in Figure 3) causes this baseline implementation to explicitly consider a large number of unnecessary possibilities. We therefore use a more efficient and more complex approach, but preserve the simple non-deterministic semantics on which testers can rely.

Delayed choice execution. UDITA provides efficient test generation by extending JPF with lazy evaluation of non-deterministic choices [23, 44]. 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.

Algorithm for getInt. Our algorithm for delayed execution of `getInt` can be expressed as a program transformation that postpones branching in the computation tree generated by the program. The transformation extends the domain of variables so that it stores a pointer to a mutable cell c where c contains 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 [23]. The transformation changes the meaning of x=$\mathsf{getInt}(a, b)$ to be lazy, storing only a symbolic representation (a, b) of possible values, which we

Source Code:

```
x0 = getInt(0,1);
x1 = getInt(0,1);
x2 = getInt(0,1);
x3 = getInt(0,1);
x4 = getInt(0,1);
y = x2;
assume(x0 ≤ x1);
assume(x1 ≤ x2);
assume(x2 ≤ x3);
assume(x3 ≤ x4);
```

Eager Execution:

```
x0 = Susp(0,1);
force(x0);

x1 = Susp(0,1);
force(x1);

x2 = Susp(0,1);
force(x2);

x3 = Susp(0,1);
force(x3);

x4 = Susp(0,1);
force(x4);

y = x2;
assume(x0 ≤ x1);
assume(x1 ≤ x2);
assume(x2 ≤ x3);
```

Delayed Execution:

```
x0 = Susp(0,1);
x1 = Susp(0,1);
x2 = Susp(0,1);
x3 = Susp(0,1);
x4 = Susp(0,1);

force(x2);
y = x2;

force(x0);
force(x1);
assume(x0 ≤ x1);

assume(x1 ≤ x2);

force(x3);
assume(x2 ≤ x3);

force(x4);
assume(x3 ≤ x4);
```

Copy Propagation:

```
x0 = Susp(0,1);
x1 = Susp(0,1);
x2 = Susp(0,1);
x3 = Susp(0,1);
x4 = Susp(0,1);
y = x2;

force(x0);
force(x1);
assume(x0 ≤ x1);

force(x2);
assume(x1 ≤ x2);

force(x3);
assume(x2 ≤ x3);

force(x4);
assume(x3 ≤ x4);
```

Fig. 9: Program and its Execution: Eager, Delayed, and Copy Propagation

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. The algorithm inserts $\mathsf{force}(x)$ before the first non-copy use of the variable x, treating all variable uses other than copying as strict operations. The delayed execution can be viewed as moving down $\mathsf{force}(x)$ from the point of definition of x to the point of (strict) use of x. Figure 9 illustrates this transformation on an example that picks an ordered 5-tuple of elements $\mathrm{x0} \leq \mathrm{x1} \leq \mathrm{x2} \leq \mathrm{x3} \leq \mathrm{x4}$ whose values are in the set $\{0, 1\}$. Generalizing from the 5 variables x0,x1,x2,x3,x4, when picking an array of n sorted elements from $\{0, 1\}$, eager execution fully explores a search tree of size $2^{O(n)}$. In contrast, delayed execution in this case explores a tree of size $O(n^2)$, as can be easily proved by induction. The delayed execution is more efficient because it does not pick the value of a variable until this variable is used in an assume statement, at which point the value is immediately forced to be greater or equal than previous values. The proof in [28, Sec. 3] shows the correctness and properties of delayed choice execution.

Our example also includes a statement that copies `x2` into `y`. Such copy statement causes the basic delayed execution to force (pick) the value of `y`. The enhanced *copy propagation* instead copies the reference to the lazy value, postponing the choice of concrete value even further. Appendix A.2 shows an example where copy propagation gives an exponential performance improvement over the basic delayed execution.

4.2 Test Generation for Linked Structures

Eager implementation. Figure 10 presents a Java-like pseudo code for an eager implementation of object pools.

```
class ObjectPool⟨T⟩ {
  ArrayList⟨T⟩ allocated; int maxSize;

  ObjectPool⟨T⟩(int maximumSize) {
    allocated = new ArrayList⟨T⟩();
    maxSize = maximumSize; }

  T getNew() {
    assume(allocated.size() < maxSize);
    T res = new T(); allocated.add(res);
    return res; }

  T getAny() {
    int i = getInt(0, allocated.size());
    if (i < allocated.size()) return allocated.get(i);
    else return getNew(); } }
```

Fig. 10: Eager choice implementation of object pools

We focus here on implementation of object pools of finite size that return non-`null` objects only. Our implementation also handles the (straightforward) extensions with infinite object pools, and possibly-`null` objects.

Isomorphism avoidance. An important issue in generating object graphs is to avoid structures that are isomorphic due to the abstract nature of Java references [7, 34]. In a purely imperative approach, the control of isomorphism is up to the tester and not UDITA. (Indeed, the code in Figure 4 generates isomorphic DAGs.) In a declarative approach that uses the `getAny` method from object pools, UDITA automatically avoids isomorphic structures, like Korat [7]. The implementation in Figure 10 avoids isomorphism by returning the first fresh object (as opposed to considering different possibilities for a fresh object).

Delayed choice execution implementation. The eager implementation in Figure 10 serves as a reference for our delayed choice implementation. The delayed choice implementation results in exploring the equivalent set of states as the reference implementation but does so much more efficiently. The high-level idea of delayed execution is the same as for `getInt`, but the implementation for object pools is more complex because `getNew` is a stateful command. As a result, simply creating a suspension around the methods from Figure 10 would *not* preserve the semantics because the side effects on the `allocated` set would occur in a different order. Consider the example

```
x1 = p.getAny(); x2 = p.getNew(); use(x2); use(x1);
```

The eager version results only in an execution where `x1!=x2`, whereas the version with suspended methods would generate both a state where `x1==x2` and a state where `x1!=x2`.

To preserve the set of reachable states of the eager reference implementation, our implementation introduces symbolic values at each call to `getNew` or `getAny` and also accumulates the constraints imposed by the requirement that `getNew` returns objects *distinct* from previously returned objects. When the program uses symbolic objects (doing a `force` of the value), UDITA assigns a concrete object to the symbolic object, ensuring that the accumulated constraints on distinct objects are satisfied. UDITA also ensures that it will be possible to instantiate the remaining symbolic objects while satisfying all the constraints. In the terminology of symbolic execution [38], UDITA maintains an efficient representation of the path condition, which expresses that certain symbolic objects are distinct, and ensures that the path condition is always satisfiable. To see the non-triviality

```
class Sym⟨T⟩ { // symbolic variable
  T chosen; int level; boolean isGetNew;
  Sym⟨T⟩(int level, boolean isGetNew) { ... }
}
class ObjectPool⟨T⟩ {
  List⟨T⟩ allChosen;
  List⟨List⟨Sym⟨T⟩⟩⟩ levels;
  int maxSize, lastLevel, minModelSize;

  ObjectPool(int maximumSize) {
    allChosen = new List⟨T⟩(); levels = new List⟨List⟨Sym⟨T⟩⟩⟩();
    maxSize = maximumSize; lastLevel = −1; minModelSize = 0; }
  Sym⟨T⟩ getAny() {
    if (lastLevel < 0) return getNew();
    sym = new Sym⟨T⟩(lastLevel, false);
    levels.get(lastLevel).add(sym); }
  Sym⟨T⟩ getNew() {
    lastLevel++;
    newLevel = new List⟨Sym⟨T⟩⟩();
    levels.add(newLevel);
    sym = new Sym⟨T⟩(lastLevel, true);
    newLevel.add(sym);
    minModelSize++;
    assume(minModelSize <= maxSize); } }
```

Fig. 11: Delayed choice execution for object pools: data structures, getAny, getNew

of our path conditions, consider the example of an object pool of size 3:

```
p = new ObjectPool<Node>(3); n1 = p.getNew();
  a1 = p.getAny(); a2 = p.getAny(); a3 = p.getAny();
n2 = p.getNew(); n3 = p.getNew();
use(a1); use(a2); use(a3);
```

The delayed execution will pick the concrete values of `a1`, `a2`, `a3` only at their use points. When it picks the values, it must have enough information to deduce that all values `a1`, `a2`, `a3` must be equal; otherwise, it will be impossible, in the pool of size 3, to assign values `n2`, `n3` such that `n2` $\notin$ {`n1`, `a1`, `a2`, `a3`} and `n3` $\notin$ {`n1`, `a1`, `a2`, `a3`, `n2`}.

Figures 11 and 12 show the pseudo-code of the desired delayed execution algorithm for object pools, implemented in UDITA. Type `List`$\langle C\rangle$ denotes an indexable linked list (such as Java `ArrayList`) storing objects of type C. Type `Sym`$\langle T\rangle$ denotes a symbolic variable, whose `chosen` field denotes concrete field (and is null if the concrete object is not chosen yet). The methods `getAny` and `getNew` from Figure 11 introduce a new symbolic variable and store it into the appropriate position in the two-dimensional `levels` data structure; `getAny` stores the symbolic variable at the current level, whereas `getNew` starts a new level. This structure encodes, for `j < i` and for all applicable `k`, that

```
levels.get(i).get(0).chosen != levels.get(j).get(k).chosen
```

The `force` method from Figure 12 picks a concrete value for a given symbolic variable by respecting the recorded constraints. After selecting in the `candidate` variable the set of objects to which the symbolic variable could be made equal to, it either 1) selects one of these objects or 2) introduces a new concrete object. Finally, it recomputes the minimal size of the model given by the current constraints, ensuring that the current choice of variables is satisfiable in the pool of the given size. Note that, although the problem has the flavor of the NP-complete graph coloring problem, the structure of our constraints (building levels in layers) allowed us to design the efficient test in the `findMinModelSize` method from Figure 12.

```
void force(Sym⟨T⟩ x) {
  if (x.chosen == null) {
    List⟨T⟩ candidates;
    if (x.isGetNew) {
      candidates = new List⟨T⟩();
      for (int i = x.level; i ≤ lastLevel; i++) {
        List⟨T⟩ currentLevel = levels.get(i);
        for (int j = 1; j < currentLevel.size(); j++)
          Sym⟨T⟩ s = currentLevel.get(j);
          if (s.chosen != null &&
              !candidates.contains(s.chosen))
            candidates.add(s.chosen); }
    } else { // x created by getAny
      candidates = new List⟨T⟩(allChosen);
      for (int i = x.level+1; i ≤ lastLevel; i++) {
        Sym⟨T⟩ s = levels.get(i).get(0); // getNew
        if (s.chosen != null) candidates.remove(s.chosen); }
    }
    int choice = getInt(0, candidates.size());
    if (choice < candidates.size())
      x.chosen = candidates.get(choice);
    else {
      x.chosen = new T();
      allChosen.add(x.chosen); }
    findMinModelSize();
    assume(minModelSize <= maxSize);
  } }
void findMinModelSize() {
  List⟨T⟩ chi = new List⟨T⟩();
  minModelSize = lastLevel;
  for (int i = 0; i ≤ lastLevel; i++) {
    foreach (Sym⟨T⟩ s in levels.get(i))
      if (s.chosen != null && !chi.contains(s.chosen))
        chi.add(s.chosen);
    int levelModelSize = chi.size() + lastLevel − i;
    minModelSize = max(minModelSize, levelModelSize); } }
```

Fig. 12: Picking a concrete object for symbolic variable of object pool in delayed execution

4.3 Correctness of Delayed Object Pools

The correctness of our algorithm can be shown by viewing it as an efficient implementation of a symbolic execution with disequality constraints. The only subtle part is showing that the `findMinModelSize` method from Figure 12 correctly computes the size of the *smallest* model of the equality and disequality constraints imposed by current symbolic variables and any concrete values assigned to them. The correctness can be shown by considering the iteration I in which `minModelSize` reaches its maximum. The concrete nodes at levels up to I together with any `getNew` nodes at higher levels must all be distinct, so each model is at least of size `minModelSize`. Conversely, by a greedy assignment that favors previously chosen concrete objects, we can construct a model of size `minModelSize`.

Correctness proof. The `levels` data structure encodes a path condition of the form $\wedge_i (x_i \neq x'_i)$. The non-null `chosen` fields encode the condition $x_i = o_k$ for concrete objects o_k. Each object pool also has an implicit condition $|\{x_1, \ldots, x_n\}| \leq$ `maxSize` where $x_1, \ldots, x_n$ are all symbolic variables.

To show the correctness of this method, note that `levelModelSize` in iteration `i` computes the size of the model consisting of 1) the `chi.size()` already allocated objects in levels up to `i`, and 2) the `lastLevel-i` objects that need to be chosen as values of **getNew** variables at levels strictly above `i`. All of these objects must be distinct, so the smallest model must have at least the sum of `minModelSize` elements, for each value of variable `i`.

Having shown that the computed value is the lower bound on the minimal model size, we next show the converse, by describing a construction of a model of size `minModelSize`. It suffices to specify the choice of concrete objects for all **getNew** variables (stored in `levels.get(i).get(0)`) that are not yet chosen: the remaining **getAny** variables can always be chosen equal to the **getNew** variables at the same level. Let us remove from the constraints all concrete objects already chosen by a **getNew** variable, and remove all **getAny** variables to which they are assigned. We use a greedy algorithm to choose the remaining **getNew** variables from level 0 to `lastLevel`. We choose either 1) a concrete **getAny** object at a higher or equal level, or, if there are not sufficiently many of those, 2) an additional fresh object. Let I be the largest level at which `levelModelSize` reaches maximum greater than `lastLevel`, and let C be the value of `chi.size()` at this step. Then for all levels below I the assignment process had sufficiently many objects already assigned to **getAny** variables and did not need to use any fresh objects. The number of concrete objects assigned to variables up to level I is therefore C. When the assignment process continues at levels above I, then all **getAny** objects are used up (if there were some left, the value I would not be the point of maximal `levelModelSize`). Consequently, the number of additional distinct objects at levels above I is exactly `lastLevel` − `I`. The total number of objects is therefore $C +$ `lastLevel` − `I`, which is exactly the value of `levelModelSize` when it reaches maximum. This shows that the constructed model has the size `minModelSize` computed by `findMinModelSize`.

4.4 Benefits of Object Pools

Specification advantage. Previous work on symbolic execution (e.g., CUTE [49]) uses equality and disequality constraints on *individual* object references (`==` and `!=`). Our work introduces the new object pool abstraction, which allows testers to conveniently express "freshness" disequality constraints of one reference against *all* references from a given user-defined set.

Algorithmic advantage. Note that an attempt to encode object pool constraints using equalities and disequalities over individual symbolic variables typically results in constraints whose satisfiability is *NP-hard*. In particular, consider a straightforward encoding of the constraint $|\{x_1, \ldots, x_n\}| \leq$ `maxSize` on the maximal size of object pool. The encoding would introduce `maxSize` fresh constants $a_1, \ldots, a_{\texttt{maxSize}}$ denoting distinct references and require

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{\texttt{maxSize}} x_i = a_j$$

Such encoding thus introduces non-trivial disjunctions into the problem. In contrast, we developed a *polynomial-time* algorithm to test the satisfiability of object pool constraints.

5. EVALUATION

We implemented UDITA by modifying Java PathFinder (JPF) version 4. The key changes were our delayed choice algorithms and object pools. We implemented them using JPF's attribute mechanism [47] to store non-deterministic values that have not been read yet. We correspondingly modified the implementation of `getInt` to generate such delayed values. We also implemented object pools as described

program	N	structs	JPF Baseline time [s]	JPF Baseline explored	Delayed Choice time [s]	Delayed Choice expl.
DAG	3	34	11.14	4802	2.19	321
	4	2352	o.o.m	-	12.42	21196
	5	769894	o.o.m	-	1673.91	4997210
HeapArray	6	13139	29.00	160132	12.50	27664
	7	117562	407.45	2739136	49.20	227494
	8	1005075	7892.88	54481005	417.70	2325069
NQueens	6	4	13.81	46656	1.82	746
	7	40	170.82	823543	3.60	3073
	8	92	3416.38	16777216	6.50	13756
RBTree	6	20	5.91	8448	5.79	3588
	7	35	21.76	54912	8.20	16983
	8	64	107.49	366080	22.27	80470
SearchTree	4	490	5.00	3584	2.26	1484
	5	5292	27.49	131250	8.29	21210
	6	60984	1810.93	6158592	60.67	305052
SortedList	6	924	11.70	55987	5.10	3967
	7	3432	126.14	960800	6.92	18026
	8	12870	2495.49	19173961	17.87	80089

Fig. 13: Enumeration of structures satisfying their invariants ("o.o.m." means "out of memory")

program	N	JPF Baseline time [s]	JPF Baseline explored	Delayed Choice time [s]	Delayed Choice expl.
RBTree bug1	$\Sigma(1\text{–}4)^*$	>1000.00	-	3.45	604
	$\Sigma(1\text{–}4)$	2.86	168	2.88	132
	6	4.83	3478	3.82	1514
	7	11.30	21706	6.90	7169
	8	48.98	157834	14.75	38457
RBTree bug2	$\Sigma(1\text{–}6)^*$	>1000.00	-	10.87	11881
	$\Sigma(1\text{–}6)$	8.83	7166	8.39	3136
	6	8.16	5548	7.81	2196
	7	15.06	31787	9.60	9454
	8	63.95	188384	22.11	42997
RBTree bug3	$\Sigma(1\text{–}4)^*$	>1000.00	-	3.86	604
	$\Sigma(1\text{–}4)$	2.66	168	2.89	132
	6	4.73	3478	5.02	1514
	7	10.54	21555	5.72	6960
	8	40.18	138853	11.61	32667
RBTree bug4	$\Sigma(1\text{–}3)^*$	>1000.00	-	1.51	98
	$\Sigma(1\text{–}3)$	1.46	30	1.41	27
	6	4.11	3451	4.55	1452
	7	10.23	21437	5.71	6807
	8	42.08	138853	13.31	32667

Fig. 14: Time taken and structures explored to find first counterexample. N = Σ(x–y) denotes the generation of all trees of sizes among $\{x, x+1, \ldots, y-1, y\}$, with y the smallest size that reveals the bug; * is declarative mode

in Section 4.2. Our code is publicly available, in the extension called `delayed`, from the JPF repository.

We performed several experiments to evaluate UDITA. The first set of experiments, on a number of data structures, compares delayed choice with base JPF for bounded-exhaustive test generation and for finding bugs. The second set of experiments compares UDITA with symbolic execution in Pex [52]. The third set of experiments, on testing refactoring engines, compares UDITA with ASTGen [19]. The fourth set of experiments uses UDITA to test parts of the UDITA implementation itself. We ran the experiments on an Intel Pentium 4 (with hyper-threading) 3.4GHz desktop running Sun Java Virtual Machine (JVM) 1.6.0_10, Eclipse 3.3.2, and NetBeans 6.5.

5.1 Testing Data Structures

We present an evaluation of delayed choice using a variety of data structure implementations: `DAG` represents directed acyclic graphs related to the example introduced in Section 2; `HeapArray` is an array-based heap data structure; `RBTree` is red-black tree; `SearchTree` is binary search tree; and `SortedList` is a doubly-linked list containing sorted elements. Additionally, `NQueens` is the traditional problem from constraint solving [4]. For each structure, we wrote its representation invariant using our combined declarative/imperative style, and for finding bugs in `RBTree` we also consider a purely declarative style. Our experimental setup compares base JPF against JPF extended with our delayed choice execution, using the same test generation program. We turn off JPF state hashing in our experiments, because duplicate states rarely arise in executions of our examples [29]. We perform two kinds of experiments: (1) enumerating all structures of a given size, and (2) finding bugs in code.

Enumerating structures. Figure 13 shows the efficiency of our approach for structure enumeration. For each program and several bounds N, we tabulate the total number of successful paths in the exploration tree (i.e., the number of structures generated), the exploration time, and the total number of explored paths. JPF generates the same number of structures with and without delayed choice, but delayed choice explores fewer paths than the base JPF, providing significant speed-ups, from 2X up to 500X as size increases.

Finding bugs using white-box testing. UDITA is primarily designed for "black-box testing" [55]: UDITA executes test generation programs to create test inputs, and then those inputs are run on the code under test without UDITA. However, UDITA can be also applied for "white-box testing" [55] by executing the code under test itself on UDITA. Consider, for instance, using the following driver code to test the `remove` method from a red-black tree (presented in Appendix A.1):

```
static void main(int N) {
  RBTree t = new RBTree(); t.initialize(N);// Pick a graph
  assume(t.isRBT());                        // satisfying invariant ,
  int v = getInt(0, N);                     // and pick a value.
  t. remove(v);                             // Run code under test ,
  assert(t.isRBT()); }                      // and check invariant .
```

Generating *any* tree that fails the assertion reveals a bug. Figure 14 shows the effectiveness of our approach for revealing bugs. Four bugs of omission were manually inserted into the implementation of `RBTree` by a student not familiar with our work. The "explored" column shows the number of candidate structures explored until the bug is hit. Note that the first row shows the results for purely declarative style (similar to figures 2 and 3, with `initialize` using `getInt/getAny/getNew`), in which base JPF is extremely slow. For combined declarative/imperative style, delayed choice once again outperforms base JPF execution, by 3X for the largest size.

Summary. The results show that delayed choice significantly improves both the time to enumerate test inputs up to a given bound, and the time to find the first bug in an input of a given (sufficiently large) size. All experiments that follow use delayed choice execution with UDITA.

5.2 Comparison with Symbolic Execution

Symbolic execution is a very active area of research, with a number of recent testing tools including Crest [8], CUTE [49], DART [31], DySy [17], EGT [10], EXE [11], KLEE [9], Pex [52], SAGE [32], Splat [57], JPF's Symbc [47]. Many of these tools build on the idea of combined concrete and symbolic execution. However, many of these tools are not publicly available and/or do not support symbolic references (either not at all or not with isomorphism avoidance). This left us with relatively few possibilities, but we were able to compare UDITA with Pex [52], a publicly available, state-of-the art tool from Microsoft Research. Pex aims at testing C#/.NET code. To solve path conditions, Pex uses Z3 [20], one of the very best constraint solvers (see `http://www.smtexec.org`). For the purpose of this comparison, we translated our inheritance graph and red-black tree code from Java into C#.

Enumerating structures. Pex, like other tools based on symbolic execution, aims at *exploring paths* of the code under test (with the goal of increasing coverage to find bugs), unlike UDITA that is designed for *generating all test inputs of a given size* (bounded-exhaustive testing). It turned out that the only way to compare Pex and UDITA for enumeration of structures was to implement object pools in Pex. We developed a simple, *eager* implementation of object pools using `ChooseValueFromRange`, which is the Pex's equivalent of the JPF's `getInt`. As expected, the results for enumeration showed that Pex can generate the same structures as UDITA, but Pex generates them much slower due to the simple implementation of object pools. We expect that a delayed/lazy implementation of object pools in Pex would similarly give substantial performance improvements, as we have found it to be the case for JPF.

Finding bugs. The results of comparing Pex and UDITA for finding bugs in data structures were more surprising. Symbolic execution can be generally effective at finding bugs by identifying the error path and solving path condition constraint for this path. We applied Pex to the the four buggy red-black tree versions discussed in Section 5.1. Pex was able to find three of these bugs (bug1, bug3, and bug4).

Surprisingly, Pex was not able to identify bug2 even when running for over an hour. However, when we ran Pex with our eager object pool extension, Pex was able to find all the bugs within seconds! We reported these findings to the Pex developers who are investigating bug2. Our view is that object pools are a powerful abstraction for guiding exploration, orthogonal to the path-bounding approaches used by tools such as Pex. We therefore expect tools like Pex to integrate object pools into their symbolic engines in the future, effectively implementing delayed choice for object pools.

Summary. The use of object pools from UDITA is helpful both for enumeration of test inputs and for finding bugs in symbolic execution tools such as Pex. In addition to the current JPF implementation, UDITA could be implemented on top of other platforms, with similar benefits.

5.3 Testing Refactoring Engines

We applied UDITA to generate Java input programs for testing refactoring engines as briefly described in Section 2, and as we previously did with ASTGen [19, Sec. 5]. Since the inputs are generated automatically, ASTGen validates outputs of refactoring engines using programmatic oracles such as checking for engine crashes, obtaining non-compilable output programs, or getting different outputs for Eclipse and NetBeans (the latter also being known as differential testing [43]). We perform two kinds of experiments: (1) rewriting some existing ASTGen generators in UDITA to compare the ease of writing generators and the efficiency of generation, and (2) writing new generators that were extremely difficult to express in ASTGen.

		ASTGen		UDITA	
generator	inputs	time [s]	LOC	time [s]	LOC
2ClsMethParent	2160	492.87	1316	117.92	835
3ClsMethChild	1152	265.19	1342	89.17	848
2ClsMethChild	576	135.34	1320	44.01	822
2Cls2FldChild	540	1.13	713	36.96	389
2Cls2FldRef	240	2.62	714	27.96	430

Fig. 15: Comparing ASTGen and UDITA

			Eclipse		NetBeans	
refactoring	time [s]	inputs	fail	bug	fail	bug
RenameMethod	105.15	207	0	0	75	1
IntroSuperType	85.80	402	59	1	7	1
InferGenericType	258.55	414	171	1	n/a	n/a

Fig. 16: Refactorings tested and faults found

Rewriting ASTGen generators. We rewrote five ASTGen generators in UDITA. Figure 15 shows the results. The generators in UDITA have fewer lines of code ("LOC", which includes the top-level generator and the library it uses) and are, in our experience, easier to write. UDITA generators are about as efficient as ASTGen generators—sometimes a bit faster, and sometimes a bit slower—which was quite surprising to us at first: ASTGen runs on top of a regular JVM, while UDITA runs on top of JPF, and JPF can be two orders of magnitude slower than JVM. We did expect UDITA generators to be easier to write but not to be faster, at least not without special optimizations [29]. However, our investigation shows that UDITA can be faster for three reasons: (1) it has a lighter framework for generators (one method in `IGenerator` for UDITA vs. five methods for ASTGen), (2) it has a faster backtracking due to JPF's storing and restoring of states rather than the re-execution of code in ASTGen, and (3) combined declarative/imperative style for iteration/generation allows more efficient positioning of backtracking points (UDITA need not build an entire input before realizing that backtracking is required).

Writing new generators. We wrote three new generators in UDITA that would be extremely difficult to write in ASTGen. All these generators use inheritance graphs which, as discussed in Section 2, are much easier to express by combining declarative and imperative styles. UDITA is more expressive than ASTGen since UDITA allows natural mixing of these two styles. These generators allowed us to test some refactorings we did not test with ASTGen (IntroduceSuperType, which replaces one class/interface with its superclass/superinterface where possible, and InferGenericType, which finds the most appropriate generic type parameters for raw types [53]) and to more thoroughly test a refactoring we did test (Rename Method). Figure 16 shows the results. We revealed four new bugs in the Eclipse and NetBeans refactoring engines, two of which are described in figures 5 and 6. As can be seen from the table, the number of failing tests is much larger than the number of (unique) bugs; we used our recently proposed oracle-based test clustering technique [35] to inspect the failures.

Differential testing of compilers. While testing the refactoring engines, we effectively used the same input programs to also perform differential testing of the Sun javac (version 1.6.0_10) and Eclipse (version 3.3.2) Java compilers. This revealed two bugs in the Sun javac compiler, where it incorrectly rejects valid programs that are accepted by the Eclipse compiler. To confirm that these programs are indeed valid, we also compiled them with a third compiler, the GNU Compiler for Java (GCJ version 4.3), which ac-

```
interface C {
  public C m(); }
interface B {
  public B m(); }
interface A extends B, C {
  public A m(); }
```

```
public class A {
  private void m(){}
  class B extends A {
    void m(int i){m();}
  }
}
```

Fig. 17: These programs generated using UDITA are accepted by the Eclipse Java compiler but (incorrectly) rejected by Sun javac compiler

generator	time [s]	inputs	failures	bugs
AnnotatedMethod	31.28	1280	0	0 (2)
ReflectionGet	23.71	160	80	1
DeclaredMethods	7.91	64	0	0
DeclaredMethodReturn	41.07	288	32	1
ReflectionSet	26.97	160	32	1
NotDefaultAnnotatedField	48.53	1760	0	0
Enum	1.67	78	0	0
ConstructorClass	12.01	387	27	1 (4)
DeclaredFieldTest	14.38	180	12	1
ClassCastMethod	27.96	102	75	1

Fig. 18: Generators for testing JPF, Note: Faults in parentheses were found in the previous version of JPF (revision 954)

cepted both of them. Figure 17 shows these programs that were generated using UDITA.

Summary. The combined declarative/imperative style in UDITA is better than purely imperative style in ASTGen: UDITA is more expressive, results in shorter (and easier to write) test generation programs, and, in some cases, even provides faster generation (despite running on JPF, which is much slower than JVM). We found several new bugs with UDITA; details of all the bugs are online [1].

5.4 Testing JPF and UDITA

We also applied UDITA to generate Java input programs for testing parts of UDITA itself. Specifically, we used differential testing [43] to check (1) whether (base) JPF correctly implements a JVM, and (2) whether our delayed choice implementation behaves as non-delayed choice.

Testing JPF. JPF is implemented as a specialized JVM that provides support for state exploration of programs with non-deterministic choices [54]. For programs without non-deterministic choices, JPF should behave as a regular JVM. We knew from our experience with JPF that it does not always behave as JVM, especially for some standard libraries (e.g., related to reflection or native methods) or for latest Java language features (e.g., annotations or enums), even though the JPF implementation is developed to support them. We wrote generators to produce small Java programs that exercise these libraries/features. We also wrote a generic driver that would compile each generated program, run it on JPF and JVM, and compare the outputs from the two. Figure 18 shows the results. Through this process, we found eleven unique bugs in an older version of JPF (five of which were corrected in a more recent revision, 1829, from the JPF repository). Detailed results are online [1].

Testing delayed choice implementation. Although we proved that our delayed choice algorithm is correct, we still need to test its implementation, especially the challenging part of object pools (Section 4.2). We wrote a generator that produces Java programs with various sequences of `getAny` and `getNew` calls on an object pool (and then later reads the returned values in various orders). We also wrote a script to compile each program and run it on JPF both with and without delayed choice. This process found an old bug in our implementation (related to the computation of `levels` from Section 4.2) which would manifest only for some sequences that mix between `getNew` calls a number of `getAny` calls exactly equal to the pool size. We subsequently corrected the bug, and (after manually proving the algorithm in Section 4.2 correct), used the generator to increase our confidence in the corrected implementation.

Summary. The use of UDITA helped us identify a number of faults in parts of the UDITA implementation, including in JPF.

6. RELATED WORK

There is a large body of work on automated test generation. This paper focuses on test generation programs, combining declarative [7, 22, 29, 36, 41, 42] and imperative [19, 35] styles in a general-purpose programming languages. Related work on topics such as specification-, constraint-, and grammar-based testing [39] is reviewed in detail in a recent paper [19] and a PhD thesis [41]. The key technique that enables efficient test generation for UDITA is delayed execution, so we review here related work on that topic.

Noll and Schlich [44] proposed delayed non-deterministic execution for model checking assembly code. While their and our approaches share the name, the algorithms differ: UDITA precisely *shares* non-deterministic values when they are copied, using lazy evaluation, whereas [44] *copies* non-deterministic values, effectively using call-by-name semantics and over-approximating state space, possibly exploring executions that are infeasible in non-delayed execution. Further differences stem from different abstraction levels, with UDITA modeling each non-deterministic integer as one symbolic value as opposed to a set of bits, and UDITA handling graph isomorphism for allocated objects.

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 [37] uses "lazy initialization" to make non-deterministic field assignments on first-access. Deng et al.'s [21] "lazier initialization" builds on generalized symbolic execution and makes non-deterministic field assignments on first-use. Visser et al. [55] use preconditions written in Java for checking satisfiability but require the users to provide "conservative preconditions" which are hard to provide manually or generate automatically. A key difference between previous work and this paper is that we provide a generic framework that supports delayed choice execution for arbitrary Java code extended with non-deterministic choices for primitive values and objects. We also apply UDITA on testing much larger code bases, finding bugs in Eclipse, NetBeans, Sun javac, and JPF.

The ECLiPSe constraint solver [4] provides a constraint logic programming (CLP) interface for writing declarative constraints. ECLiPSe provides *suspensions* that delay testing of predicates until more information is available. Researchers have proposed translating imperative programs into constraint programming engines [24] but faced limitations of current CLP implementations. We believe that non-deterministic extensions of popular programming languages such as Java can lead both to advances of software model checking and to scalable implementations of constraint solvers.

Recent research on automated test generation includes approaches based on exploration of method sequences [18, 51, 56] for generation of object-oriented unit tests. Such exploration cannot be used to generate complex test inputs when there are no appropriate methods, e.g., for building inheritance graphs. UDITA can directly generate complex test inputs, and imperative style generators in UDITA can even use method sequences.

Unlike symbolic execution [2, 3, 13, 14, 15, 17, 26, 27, 38, 48], UDITA relies primarily on concrete execution to generate test inputs, and uses a polynomial-time algorithms (Section 4) to ensure the feasibility of the currently explored path. This is in contrast to traditional symbolic execution approaches whose path conditions belong to NP-hard logics (often containing propositional logic, uninterpreted functions, and bitvector arithmetic). Several recent approaches show promising results by combining symbolic with concrete execution [11, 17, 31, 47, 49, 52] or with grammar-based input generation [30]. In contrast to combination of concrete executions with abstraction [6, 46], UDITA focuses on test generation by efficiently covering a set of concrete executions, without approximation. We have no evidence that any of the existing approaches can find bugs such as those that UDITA found in Eclipse, NetBeans, Sun javac, or JPF. Our experience with Pex [52] suggests that other tools by themselves do not give the benefits of UDITA, but could be used, similarly to Java PathFinder, as a platform on which to build UDITA-like techniques.

7. CONCLUSIONS

We have found UDITA to be an expressive and convenient framework for specifying complex test inputs. Because it extends Java, it has the expressive power and the appeal of a full-fledged programming language. Because it contains non-deterministic constructs, it is appropriate for describing tests in a wide range of styles, from predicates that indicate properties, to generators that create only desired structures. To describe linked structures, we have found the new object pool abstraction to be particularly helpful. We have found UDITA easier to use than previous frameworks.

UDITA quickly revealed bugs in data structure implementations, and was effective in systematically generating structures up to a given size. The effectiveness of UDITA was in large part due to our lazy evaluation technique for non-deterministic choices, and the algorithms for delayed execution of object pool operations without solving NP-hard constraints. We have applied UDITA to real-world software and uncovered previously unknown bugs in Eclipse, NetBeans, Sun javac, and Java PathFinder.

Acknowledgments

We would like to thank Nikolai Tillmann for discussions about Pex, Igor Andjelkovic for creating faulty versions of the Red Black Tree example, Yun Young Lee for help with NetBeans, and Brett Daniel for help with Eclipse. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-0845628, CCF-0746856, CCF-0702680, CNS-0615372, and IIS-0438967; US Air Force Office of Scientific Research Grant No. FA9550-09-1-0351; and Swiss Science Foundation Grant "Precise and Scalable Analyses for Reliable Software". Milos Gligoric was supported by Saburo Muroga fellowship while working on a part of this project.

References

[1] UDITA website. http://mir.cs.illinois.edu/udita.

[2] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design*, 18(2):97–116, 2001.

[3] S. Anand, A. Orso, and M. J. Harrold. Type-dependence analysis and program transformation for symbolic execution. In *TACAS*, pages 117–133, 2007.

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

[5] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. J. White. State generation and automated class testing. *STVR*, 2000.

[6] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, 2008.

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

[8] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, 2008.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[10] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.

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

[12] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.

[13] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. Prentice-Hall, Inc., 1981.

[14] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using symbolic execution for verifying safety-critical systems. In *ESEC / SIGSOFT FSE*, pages 142–151, 2001.

[15] A. Coen-Porisini, F. D. Paoli, C. Ghezzi, and D. Mandrioli. Software specialization via symbolic execution. *IEEE Trans. Software Eng.*, 17(9):884–899, 1991.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. 2001.

[17] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

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

[19] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC/FSE*, 2007.

[20] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[21] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE*, 2006.

[22] B. Elkarablieh, D. Marinov, and S. Khurshid. Efficient solving of structural constraints. In *ISSTA*, 2008.

[23] S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy non-deterministic programming. In *ICFP*, 2009.

[24] C. Flanagan. Automatic software model checking via constraint logic. *J-SCP*, 50(1-3), 2004.

[25] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. 1999.

[26] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. Symbolic execution of concurrent systems using petri nets. *Comput. Lang.*, 14(4):263–281, 1989.

[27] C. Ghezzi, D. Mandrioli, and A. Tecchio. Program simplification via symbolic interpretation. In *FSTTCS*, pages 116–128, 1985.

[28] M. Gligoric, T. Gvero, S. Khurshid, V. Kuncak, and D. Marinov. On delayed choice execution for falsification. Technical report, EPFL-IC-LARA, 2008.

[29] M. Gligoric, T. Gvero, S. Lauterburg, D. Marinov, and S. Khurshid. Optimizing generation of object graphs in Java PathFinder. In *ICST*, 2009.

[30] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.

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

[32] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[33] E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *CAV*, 2000.

[34] R. Iosif. Symmetry reduction criteria for software model checking. In *SPIN*, 2002.

[35] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov. Reducing the costs of bounded-exhaustive testing. In *FASE*, 2009.

[36] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *J-ASE*, 11(4), 2004.

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

[38] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[39] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, 2006.

[40] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[41] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005.

[42] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, 2003.

[43] W. M. McKeeman. Differential testing for software. *J-DTJ*, 10(1), 1998.

[44] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *HVC*, 2007.

[45] W. F. Opdyke and R. E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA*, 1990.

[46] C. Pasareanu, R. Pelánek, and W. Visser. Predicate abstraction with under-approximation refinement. *J-LMCS*, 3(1), 2007.

[47] 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.

[48] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.

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

[50] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA*, 2004.

[51] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *ESEC/FSE*, 2009.

[52] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, 2008.

[53] F. Tip. Refactoring using type constraints. In *SAS*, 2007.

[54] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *J-ASE*, 10(2), 2003.

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

[56] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, 2006.

[57] R. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA*, 2008.

APPENDIX

A. FURTHER EXAMPLES

A.1 Red-Black Trees

As another example to illustrate UDITA, we use red-black trees [16], a widely used data structure underlying, for example, the `TreeMap` class implementation of the Java Standard library. Due to non-trivial data structure invariants, red-black trees often appear as a benchmark for techniques that check expressive properties [5, 36, 37, 40]. Figure 19 shows the skeleton of a red-black tree implementation in Java.

Specification of red-black trees. The class invariant for red-black trees consists of the following properties [16]:

1. **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.

2. **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.

3. **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`.

Suppose that we want to test the method `remove` which deletes an element from a red-black tree. We need to generate inputs that satisfy the class invariant, run them on `remove`, and check that the modified state still satisfies the invariant (and potentially other properties).

Declarative test abstractions. The UDITA approach follows Korat [7] in that the developers can use arbitrary (Java) code to write *predicates* that encode properties of test inputs. Figure 20 shows the `isRBT` Java predicate that returns `true` exactly when the class invariant holds in the current state (*declarative* style of specification).

To generate test inputs from predicates, the developer needs to specify possible values for input elements, which in our example are tree nodes, their colors, and values. For this purpose, UDITA uses non-deterministic assignments `k=getInt(1, N)`. Figure 21 shows the non-deterministic initialization of a red-black tree data structure. It proceeds in several steps: 1) pick the tree size (the number of nodes), 2) create a pool of objects of this size, and 3) iterate over all objects in the pool and non-deterministically initialize their fields to point to other objects in the pool. The `getAny` method picks an arbitrary object from the pool and can be, in the simplest form, implemented using `getInt`.

Generating inputs and testing code. Figure 22 shows how we can put together generation of trees and values with testing of the `remove` method, including checking the invariant with the usual Java `assert` statement that signals an error if the condition is not satisfied. The `assume` method silently ignores the current execution and moves to other executions if the condition is not satisfied.

```
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. 19: A red-black tree implementation in Java

```
boolean isRBT() {
  return treeness() && coloring() && ordering();
}
boolean treeness() {
  if (root == null) return size == 0;
  Set<Node> visited = new HashSet<Node>();
  visited.add(t.root);
  List<Node> workList = new 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 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. 20: Red-black tree class invariant as an executable predicate

```
void initialize(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. 21: Method performing non-deterministic initialization of a red-black tree

```
static void main(int N) {
  RBTree t = new RBTree(); t.initialize(N);// Pick a graph
  assume(t.isRBT());                      // satisfying   invariant ,
  int v = getInt(0, N);                   // and pick a value.
  t.remove(v);                            // Run code under test ,
  assert(t.isRBT()); }                    // and check invariant .
```

Fig. 22: Testing the `remove` method

```
Tree generateRBT(int N) {
  Tree t = new Tree();
  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. 23: Code that directly generates trees

Imperative test abstractions. Instead of generating possible graphs in Figure 21 and then filtering (Figure 20) those that are not trees using the `treeness` method, Figure 23 shows a simpler and faster alternative that directly generates trees of size N with the `generateTreeBackbone` method. While the `treeness` method from Figure 20 presents a *predicate* characterizing trees, Figure 23 presents a *generator* for trees. As mentioned, we call the former style *declarative* as it only specifies *what* the trees look like, and we call the latter style *imperative* as it specifies *how* to generate trees [19].

Writing (imperative) generators instead of (declarative) predicates can dramatically speed up the generation. 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 [5]. In comparison, declarative generation is often easier, anecdotally confirmed by the fact that undergraduate students are often able to write appropriate checks [42]. This trade-off justifies delayed choice execution as an optimization for predicate-based execution exploration, but also asks for an approach to combine predicates and generators.

Unifying predicates and generators. Our UDITA approach makes combination of predicates and generators possible because they are both expressed in a unified framework: systematic execution of non-deterministic assignments. Consider the properties in our running example. For the **treeness** property, comparing the imperative generation (Figure 23) and declarative generation (figures 20 and 21), one could argue that it is easier to write a generator than a predicate. However, for the **coloring** property, it is much easier to write a predicate than a generator. UDITA allows the tester to combine, for example, a generator for **treeness** with a predicate for **coloring**. One would generate trees with the `generateTreeBackbone` method of Figure 23 and find

```
void sort(int[] keys, int[] vals) {
  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 = vals[i]; vals[i] = vals[j]; vals[j] = tmp; }
}
static void main() {
  int len = getInt(0, N);
  int[] keys = new int[len]; int[] values = new int[len];
  for (int i = 0; i < len; i++) {
    keys[i] = getInt(0, N); values[i] = getInt(0, V−1); }
  sort(keys, values);
  assert (sorted(keys));
}
```

Fig. 24: Checking code that sorts data stored in two arrays

appropriate node colors using the `coloring` predicate from Figure 20.

In Section 5.1 we evaluate only the combined declarative/imperative style, since test abstractions are much easier to write than for purely imperative style, and generation for purely declarative style is several orders of magnitude slower on basic JPF without delayed choice than with delayed choice (compare over an hour to 1.2 seconds just for size 3). For red-black tree, delayed choice speeds up basic JPF both for test generation and for finding bugs; the speedups for various sizes (from 6 to 9) range from 3X to 7X.

A.2 Copy Propagation Example

We illustrate the copy propagation feature of UDITA, 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 nodes are stored according to `key` values, no `value` field in a tree will be read by the `remove` method. Therefore, even if the `value` fields are initialized with values belonging to a large set, the search for code from Figure 22 will terminate equally fast, proving that the initial `value` fields do not affect the execution of `remove`.

Consider, however, code shown in Figure 24 for 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 $\mathtt{V}^N$ 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 for primitives or field dereference for pointers. In this example, such non-copy operations do not arise for the elements of the `values` array. For $N = 5$, copy propagation makes the exhaustive exploration finish in 4.8 seconds, regardless of the `V` bound. In contrast, the exhaustive exploration without copy propagation does not complete within an hour even for `V`=5.

***

EPFL Technical Report LARA-REPORT-005, September 2009