Test Generation through Programming in UDITA

Milos Gligoric
University of Illinois
Urbana, IL 61801, USA
gliga@illinois.edu

Tihomir Gvero
Ecole Polytechnique Fédérale
Lausanne, Switzerland
tihomir.gvero@epfl.ch

Vilas Jagannath
University of Illinois
Urbana IL, 61801, USA
vbangal2@illinois.edu

Sarfraz Khurshid
University of Texas
Austin, TX 78712, USA
khurshid@ece.utexas.edu

Viktor Kuncak
Ecole Polytechnique Fédérale
Lausanne, Switzerland
viktor.kuncak@epfl.ch

Darko Marinov
University of Illinois
Urbana IL, 61801, USA
marinov@illinois.edu

ABSTRACT

We present an approach for describing tests using non-deterministic *test generation programs*. To write such 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 bugs in Eclipse, NetBeans, Sun javac, and JPF.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Verification

Keywords

Automated testing, test generation, test filtering, test predicates, test programs, UDITA, Java PathFinder, Pex

1. INTRODUCTION

Testing is the most widely used method for detecting software bugs in industry, and the importance of testing is growing as the consequences of software bugs 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*. 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).

Recent techniques aim to reduce the burden of manual testing using systematic test generation based on specifications [5, 24] or on symbolic execution [8, 26] and its hybrids with concrete executions [6, 9, 13, 20, 25, 34, 35, 38]. Modern (hybrid) symbolic execution techniques can handle advanced constructs of object-oriented programs, but practical application of these techniques were largely limited to testing units of code much smaller than hundred thousand lines, or generating input 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. Automatically handling programs of the *complexity of a compiler* remains challenging for current systematic approaches. Our approach is to allow testers to utilize their domain knowledge to scale these systematic approaches.

We propose a new technique 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* [29, 36] that validates the code under test for *all test inputs* within a given bound (e.g., all trees with up to N nodes). UDITA thus enables testers to avoid manual generation of individual tests. However, our approach does not attempt to fully automatically identify tests [6, 20], 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 several 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)

ICSE '10, May 2-8 2010, Cape Town, South Africa

restricts these choices. These constructs are familiar to users of model checkers such as Java PathFinder (JPF) [40]. 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 two styles—*filtering* (also previously called declarative) [5, 14, 18, 24, 28, 29] and *generating* (also previously called imperative) [11,23]—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 [29, 36]. The efficiency of our techniques is based on a general principle of *delayed choice* [31], i.e., lazy non-deterministic evaluation [15]. 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* abstraction. We postpone the choice of object identity until object's first non-copy use, reducing the amount of search. Furthermore, we avoid isomorphic structures [22, 28] 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 10 and 11), which is in contrast to NP-hard constraints in traditional symbolic execution [8, 26].

3) Implementation: We describe an implementation of UDITA and our optimizations on top of JPF [40], a popular model checker for Java, which makes it easy to provide UDITA as a library. Our code is publicly available [1].

4) Evaluation: We have performed several sets of experiments to evaluate UDITA, mostly for black-box testing. The *first set* of experiments, on six data structures, shows that **our optimizations improve the time** to generate test inputs up to a given bound.

The *second 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 [32]. 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 filtering/generating style shows that, compared to our prior approach, ASTGen [11,23], UDITA is **more expressive**, resulting in **shorter** (and easier to write)

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

Figure 1: A representation of inheritance graphs

test generation programs, with **sometimes faster generation** (even on a slower JPF virtual machine). Through these experiments, we revealed four **bugs in Eclipse and NetBeans** (all four have been confirmed by developers and assigned to be fixed), and even two bugs in the Sun Java compiler.

The *third 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.

The *fourth set* of experiments, for white-box testing, compared UDITA with Pex [38], 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**.

Our experimental results are publicly available [1]. Additional details are provided in a technical report [17].

2. EXAMPLE

To illustrate UDITA, we consider generation of inheritance graphs for Java programs. Such generation helps in testing real-world applications including compilers, interpreters, model checkers, and refactoring engines (Section 5). The example illustrates how UDITA can 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 classes or interfaces. (We do not explicitly model the `java.lang.Object` class.)

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 *filtering* and *generating* style. In a purely filtering style, embodied in techniques such as TestEra [24] and Korat [5,14,18,28,29], the tester writes the *predicates*—*what* the test inputs should satisfy; then the *tool searches* for valid tests. In contrast, in a purely generating style, embodied in techniques such as ASTGen [11,23], 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.

Filtering approach. Figure 2 shows Java predicates that return `true` when the above inheritance graph prop-

```
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 (!isAcyclic(n, path, visited)) return false;
  return true; }
boolean isAcyclic(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 < supertypes.length; i++) {
    Node s = supertypes[i];
    // two supertypes cannot be the same
    for (int j = 0; j < i; j++)
      if (s == supertypes[j]) return false;
    // check property on every supertype of this node
    if (!isAcyclic(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; }
      } } }
```

Figure 2: Filtering 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 : 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; }

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

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

Figure 3: Examples of bounded-exhaustive generation

erties 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

```
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
}
```

Figure 4: Generating approach for inheritance graphs

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 `mainFilt` on JPF/UDITA generates all inheritance graphs of size N.

Generating 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 directed cycles (because `supertypes` of a node `i` can only be nodes `k` that were generated before `i`).

Writing generators instead of predicates can dramatically speed up generation. However, using generators alone is fairly involved. Although it is relatively easy to write a generator for all *arbitrary* DAGs, it is non-trivial to eliminate isomorphic graphs (Section 4.2) or to properly label nodes as classes and interfaces (`generateJavaInheritance`). Properties of other data structures can be even harder to express as generators. For example, an entire research paper was devoted to efficient generation of red-black trees [3]. In comparison, filtering is often easier, anecdotally confirmed by the fact that even undergraduate students are able to write appropriate checks [29]. 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 Figure 4 and 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**: one would write a new `main` that uses `generateDAGBackbone` and `assume(isJavaInheritance)`.

Test generation. After the tester writes some predicates and/or generators, it is necessary to execute them to generate the tests. JPF already provides an execution engine for

```
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(); }
```

Figure 5: 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`

```
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(); }
```

Figure 6: UseSupertypeWherePossible 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`

`getInt` and `getBoolean` non-deterministic choices. Naive implementations 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 naive implementations, which we call *eager* as they immediately return a value, result in a combinatorial explosion, e.g., `mainFilt` from Figure 3 for $N = 4$ does not terminate in 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, `mainFilt` from Figure 3 for $N = 4$ terminates in just 5.5 seconds with our delayed choice. Generating approach can be even faster than filtering search. Section 5.1.1 shows our experimental results for data structures. We evaluate mostly the combined filtering/generating style, since test programs are much easier to write than for purely generating style, and generation for purely filtering style is several orders of magnitude slower on basic JPF without delayed choice.

Section 5.1.2 shows our results for testing refactoring engines, where we built on the inheritance graph generator to produce Java programs as test inputs. Figures 5 and 6 show two example input programs, generated by UDITA, which found bugs in Eclipse, specifically in the InferGenericType and UseSupertypeWherePossible refactorings.

3. UDITA LANGUAGE

UDITA language makes it easy to develop generic, reusable, and composable generators. The key aspects of the UDITA are: (1) constructs for generating 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() { ... } }
```

Figure 7: Basic operations for object pools

```
interface IGenerator⟨T⟩ { T generate(); }
class IntGenerator implements IGenerator⟨int⟩ {
  int lo, hi;
  IntGenerator(int lo, int hi) { this.lo = lo; this.hi = hi; }
  int 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()); } }
```

Figure 8: UDITA interface for generators and some example generators

Basic Generators. The generators for UDITA borrow 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 a new notion, *object pools*, for non-deterministic choices of objects. Figure 7 shows the interface for object pools. The constructor can create finite (if `size > 0`) and 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 primitive values (ignoring any boxing of primitive values needed in Java) and an example `IGGenerator` that encapsulates filtering style predicates (`isDAG` and `isJavaInheritance`).

The design of UDITA generators is influenced by ASTGen [11] (which provides Java generators for abstract syntax trees for testing refactoring engines) and QuickCheck [7] (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* [11, Sec. 3.2]. The cause of that complexity is that ASTGen runs on a deterministic language; 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 supports non-determinism, with program execution enumerating all non-deterministic choices. Compared to QuickCheck [7], which supports only random generation, UDITA focuses on bounded-exhaustive generation, obtaining random generation for free as one of the possible explo-

ration strategies of non-deterministic choices (where additional strategies include depth-first and breadth-first).

Composing generators. An important feature of frameworks such as ASTGen, QuickCheck, or UDITA is to allow reuse and composition of basic generators into more complex generators [7, 11]. UDITA again offers a substantially simpler solution than ASTGen. Figure 8 shows an example generator that produces pairs of values based on generators for left and right pair elements. Note that the `generate` method of `PairGenerator` has only one line of code. In contrast, the corresponding ASTGen generator has *ten lines of code* [11, 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 [7] but is designed for the purely functional language Haskell and has no support for generating non-isomorphic graph structures. Neither ASTGen nor QuickCheck provide unified filtering/generating style like 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 that still preserves 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 [15, 31]. 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 [15]. The transformation changes the meaning of x=$\mathsf{getInt}(a, b)$ to be lazy, storing only a symbolic representation (a, b) of possible values. 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. Although in general both delayed and eager choice could explore exponentially many paths, in experiments we found exponential speedup when using delayed choice instead of eager choice (figures 12 and 16). Delayed choice provides speedup because it avoids exploring the values of variables not used in an execution that evaluates `assume(false)`.

```
class ObjectPool⟨T⟩ {
  ArrayList⟨T⟩ allocated; int maxSize;
  ObjectPool⟨T⟩(int size) {
    allocated = new ArrayList⟨T⟩();
    maxSize = size; }
  T getAny() {
    int i = getInt(0, allocated.size());
    if (i < allocated.size()) return allocated.get(i);
    else return getNew(); }
  T getNew() {
    assume(allocated.size() < maxSize);
    T res = new T(); allocated.add(res);
    return res; } }
```

Figure 9: Eager implementation of object pools

4.2 Test Generation for Linked Structures

Eager implementation. Figure 9 presents a Java-like pseudo code for an eager 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 unbounded 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 [5, 22]. For instance, DAGs that have the same structure but differ in the identity of nodes are isomorphic. In a purely generating approach, the control of isomorphism is up to the tester and not UDITA. (Indeed, the code in Figure 4 generates isomorphic DAGs.) In a filtering approach that uses the `getAny` method from object pools, UDITA automatically avoids isomorphic structures, like Korat [5]. The implementation in Figure 9 avoids isomorphism by returning only the first fresh object (rather than several different fresh objects).

Delayed execution implementation. The eager implementation in Figure 9 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 command that changes the state (the `allocated` set). As a result, simply creating a suspension around the methods from Figure 9 would *not* preserve the semantics because the side effects on the `allocated` set would occur in a different order.

To preserve the set of reachable states of the eager 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

```
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 size) {
    allChosen = new List⟨T⟩(); levels = new List⟨List⟨Sym⟨T⟩⟩⟩();
    maxSize = size; 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); } }
```

Figure 10: Delayed execution for object pools: data structures, getAny, getNew

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 [26], UDITA maintains an efficient representation of the path condition, which expresses that certain symbolic objects are distinct, and ensures that the path condition is satisfiable. To see the non-triviality of our path conditions, consider this example with 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 10 and 11 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 10 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 11 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 under the current constraints, ensuring that the current choice of variables is satisfiable in the pool

```
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); } }
```

Figure 11: Picking a concrete object for symbolic variable of object pool in delayed execution

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.

Correctness proof. 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 11 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`. The accompanying technical report [17] has more proof details.

Novelty of object pools. Previous work on symbolic execution also uses equality constraints on individual object references (e.g., CUTE [35]), but can only encode them into constraints whose satisfiability is *NP-hard* (in particular, the constraint on the maximal number of distinct references typically introduce disjunctions). 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. Moreover, we developed a *polynomial-time* algorithm to test the satisfiability of these constraints.

			JPF Baseline		Delayed Choice	
program	N	structs	time [s]	explored	time [s]	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

Figure 12: Enumeration of structures satisfying their invariants ("o.o.m." means "out of memory")

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 [34] 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 in Section 4.2. Our code is publicly available [1].

We performed several experiments to evaluate UDITA. UDITA is most applicable for black-box testing. The first set of experiments, on six data structures, compares delayed choice with base JPF for bounded-exhaustive test generation. The second set of experiments, on testing refactoring engines, compares UDITA with ASTGen [11]. The third set of experiments uses UDITA to test parts of the UDITA implementation itself. UDITA can be also used for white-box testing. The fourth set of experiments compares UDITA with symbolic execution in Pex [38]. We ran the experiments on an AMD Turion 1.80GHz laptop with Sun JVM 1.6.0_12, Eclipse 3.3.2, NetBeans 6.5, and Pex 0.19.41110.1.

5.1 Black-Box Testing

5.1.1 Generating 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 [2]. For each structure, we wrote its representation invariant using our combined filtering/generating 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 [18].

Enumerating structures. Figure 12 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.

Summary. The results show that delayed choice significantly improves the time to enumerate test inputs up to a given bound.

		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

Figure 13: Comparing ASTGen and UDITA

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

Figure 14: Refactorings tested and bugs found

5.1.2 Testing Refactoring Engines

We applied UDITA to generate Java input programs for testing refactoring engines as briefly described in Section 2 and as previously done with ASTGen [11, Sec. 5]. Since the inputs are generated automatically, the outputs are validated using programmatic oracles such as checking for refactoring engine crashes, obtaining non-compilable output programs, or getting different outputs for Eclipse and NetBeans (known as differential testing [30]). 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 would be very difficult to express in ASTGen.

Rewriting ASTGen generators. We rewrote five (randomly chosen, advanced) ASTGen generators in UDITA. Figure 13 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, often easier to write. UDITA conceptually subsumes ASTGen, so we could not find a case where UDITA code would be more complex than ASTGen code. 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 [18]. Our investigation shows that UDITA can be faster for two reasons: (1) it has a faster backtracking due to JPF's storing and restoring of states rather than the re-execution of code in ASTGen, and (2) combined filtering/generating 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 filtering and generating 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 (UseSupertypeWherePossible, which replaces one class/interface with its superclass/superinterface where possible, and InferGenericType, which finds the most appropriate generic type parameters for raw types [39]) and to more thoroughly test a refactoring we did test (RenameMethod). Figure 14 shows the results. We revealed four bugs in Eclipse and NetBeans, two of which are shown 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 oracle-based test clustering technique [23] 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 differences, which are likely bugs in the Sun javac compiler as it incorrectly rejects valid programs accepted by the Eclipse compiler. We had reported these bugs to Sun, but they were not confirmed as of this writing.

Summary. The combined filtering/generating style in UDITA is better than purely generating 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].

5.1.3 Testing JPF and UDITA

We also applied UDITA to generate Java input programs for testing parts of UDITA itself. Specifically, we used differential testing [30] 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 [40]. 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 the latest Java language features (e.g., annotations or enums). 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 15 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 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 a bug in our implementation (related to the computation of `levels` from Section 4.2) which occurred 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 used the generator to increase our confidence in the corrected implementation.

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

Figure 15: Generators for testing JPF; bugs in parentheses were found in an older JPF version (revision 954)

Summary. The use of UDITA helped us identify a number of bugs in parts of the UDITA implementation.

5.2 White-Box Testing

UDITA is primarily designed for *black-box testing* [41]: UDITA executes test generation programs to create test inputs, and then those *inputs are run on the code under test as usual, without UDITA*. However, UDITA can be also applied for a limited form of *white-box testing* [41] by *executing the code under test itself on UDITA*. Note that UDITA does not use the information about the code under test, e.g., to increase syntactic coverage. Instead, UDITA executes the code just to speed up the full coverage of the specified, bounded region of the input space. Consider, for instance, using the following code to test the `remove` method from a red-black tree [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 16 shows the effectiveness of our approach for revealing bugs. Eight bugs of omission were manually inserted into an implementation of `RBTree` by students not familiar with our work. For each bug, the first row is for Pex (Section 5.2.1). The second row is for purely filtering style (as in figures 2 and 3, with `initialize` using `getInt/getAny/getNew`), in which base JPF is extremely slow. The third row is for combined filtering/generating style, and delayed choice again outperforms base JPF for larger sizes.

5.2.1 Comparison with Symbolic Execution

Symbolic execution is a very active area of research, with a number of recent testing tools including Crest, CUTE, DART, DySy, EGT, EXE, KLEE, Pex, SAGE, Splat, JPF's Symbc. (Our technical report [17] has a detailed list of references.) 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). Pex [38] is a publicly available, state-of-the art tool from Microsoft Research that supports symbolic references and avoids isomorphism.

		UDITA Eager		UDITA Delayed		Pex
bug#	style N	time [s]	expl.	time	expl.	time
1	filter 1-*					22.05
	filter 1–4	timeout	-	1.89	799	14.49
	f/g 1–4	1.61	168	1.59	132	8.13
2	filter 1-*					timeout
	filter 1–6	timeout	-	12.37	16620	137.66
	f/g 1–6	10.26	7166	8.20	3163	89.34
3	filter 1-*					21.53
	filter 1–2	9.20	10710	0.70	27	9.83
	f/g 1–2	0.61	9	0.62	9	5.14
4	filter 1-*					8.93
	filter 1–3	timeout	-	0.84	136	7.10
	f/g 1–3	0.75	30	0.80	27	5.03
5	filter 1-*					24.65
	filter 1–3	timeout	-	1.12	151	12.45
	f/g 1–3	1.08	31	1.09	28	5.59
6	filter 1-*					4.55
	filter 1–1	0.50	1	0.47	1	4.69
	f/g 1–1	0.36	1	0.39	1	4.19
7	filter 1-*					2.72
	filter 1–1	0.53	1	0.53	1	4.99
	f/g 1–1	0.47	1	0.49	1	4.27
8	filter 1-*					12.50
	filter 1–4	timeout	-	1.58	676	22.95
	f/g 1–4	1.22	145	1.36	120	7.87

Figure 16: Time taken and structures explored to find the first bug in remove/put methods of red-black tree. "timeout" denotes time over 1 hour. "filter" denotes using purely filtering; "f/g" denotes combined filtering/-generating style. UDITA requires bounds; "1–s" for N denotes the generation of all trees of sizes from 1 to s, where s is the smallest size that reveals the bug. Pex can also work without bounds (denoted "filter 1-*").

Pex is used for testing C#/.NET code. To solve path conditions, Pex uses Z3 [12], one of the very best constraint solvers (see http://www.smtexec.org).

We compared UDITA with Pex. To enable this comparison, we translated buggy red-black tree code from Java into C#. We also translated the (filtering) predicates and (generating) generators for red-black tree to C#. We used Pex, as UDITA, to test one method in isolation, `remove` or `put`. (An alternative is to test several methods at once through method sequences [10, 37, 42]). The predicate is required to specify pre- and post-condition for the method under test, in both Pex and UDITA. Note that Pex, unlike UDITA, does *not* require specifying bounds on the input size, but we wrote a simple, *eager* implementation of object pools in Pex/C# to be able to limit the search space for Pex.

Figure 16 shows the results. Pex times are averaged over five runs, because the results can differ as objects get allocated at different locations in different runs. Pex is able to quickly find all the bugs except that none of the five runs found bug2 in filtering mode with no object pools. 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). We hypothesized that Pex misses bug2 because it requires a path with several repeated branches (resulting from loop unrolling) for a tree of size 6, while Pex aims at increasing branch coverage, thus giving less priority to repeated branches. Pex developers investigated bug2 and found that it is indeed missed because Pex's default exploration strategy does not give priority to paths that could find this bug.

However, when we ran Pex with object pools (even a simple, *eager* implementation), Pex was able to find bug2 in about 137 seconds. Despite these results, we do not expect UDITA by itself (concrete execution with object pools) to be better than Pex for white-box testing. 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. In addition to the current JPF implementation, UDITA can be implemented on top of other platforms, with similar costs and benefits: if the tester spends more time guiding the exploration, the tool may find some bugs faster.

6. RELATED WORK

There is a large body of work on automated test generation. This paper focuses on test generation programs, combining filtering [5, 14, 18, 24, 28, 29] and generating [11, 23] styles in a general-purpose programming language. Related work on topics such as specification-, constraint-, and grammar-based testing [27] is reviewed in more detail in a previous paper [11] and a PhD thesis [28]. 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 [31] 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 that are copied, using lazy evaluation, whereas their approach [31] *copies* non-deterministic values, effectively using call-by-name semantics and over-approximating state space, possibly exploring executions that are infeasible in regular 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—for constraints written in both imperative and declarative languages. For example, Korat [5] implicitly uses delayed choice by monitoring field accesses and using them in field initializations for the new candidates it explores. Generalized symbolic execution [25] uses "lazy initialization" to make non-deterministic field assignments on first access. Deng et al.'s [13] "lazier initialization" builds on generalized symbolic execution and makes non-deterministic field assignments on first use. Visser et al. [41] 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 [2] 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 CLP engines [16] but faced limitations of current CLP implementations. We believe that non-deterministic exten-

sions of popular programming languages such as Java can lead both to advances of software model checking and to scalable implementations of constraint solvers.

Approaches to automated test generation includes those based on exploration of method sequences for generation of object-oriented unit tests [10, 37, 42]. 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 generators in UDITA can even use method sequences.

Unlike symbolic execution [8, 26], UDITA relies mostly on concrete execution to generate test inputs, and uses a polynomial-time algorithm (Section 4.2) to ensure the feasibility of the currently explored path. This is in contrast to traditional symbolic execution where 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 [6, 9, 20, 21, 34, 35, 38] or with grammar-based input generation [19]. In contrast to combination of concrete executions with abstraction [4, 33], UDITA focuses on test generation by efficiently covering a set of concrete executions, without approximation. UDITA is most applicable for black-box testing as shown by finding bugs in Eclipse, NetBeans, Sun javac, and JPF. However, UDITA requires/allows the tester to manually guide the exploration. Tools based on symbolic execution are more automated and better than UDITA for white-box testing. Our experience with Pex suggests that other tools can benefit by incorporating object pools from UDITA.

Acknowledgments. We thank our shepherd and the anonymous reviewers for help in improving this paper; Jonathan de Halleux, Suresh Thummalapenta, Nikolai Tillmann, Xusheng Xiao, and Tao Xie for help with Pex; Yun Young Lee for help with NetBeans; Brett Daniel for help with Eclipse; Rohan Sharma for preparing red-black tree code; and Igor Andjelkovic, Nima Honarmand, Dusan Matic, and Milos Siroka for creating faulty versions of the red-black tree example. This material is based upon work partially supported by the US NSF under Grant Nos. CCF-0845628, CCF-0746856, and IIS-0438967; US AFOSR Grant No. FA9550-09-1-0351; the Microsoft Innovation Cluster for Embedded Software; and the Swiss State Secretariat for Education and Research. Milos Gligoric was supported in part by the Saburo Muroga fellowship.

7. REFERENCES

[1] UDITA website. http://mir.cs.illinois.edu/udita.
[2] K. Apt and M. G. Wallace. *Constraint Logic Programming using Eclipse*. CUP, 2006.
[3] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. J. White. State generation and automated class testing. *STVR*, 2000.
[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, 2008.
[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.
[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, 2006.
[7] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.
[8] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. Prentice-Hall, Inc., 1981.
[9] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, 2008.
[10] 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.
[11] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC/FSE*, 2007.
[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
[13] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE*, 2006.
[14] B. Elkarablieh, D. Marinov, and S. Khurshid. Efficient solving of structural constraints. In *ISSTA*, 2008.
[15] S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy non-deterministic programming. In *ICFP*, 2009.
[16] C. Flanagan. Automatic software model checking via constraint logic. *J-SCP*, 50(1-3), 2004.
[17] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. On test generation through programming in UDITA. Tech. report LARA-REPORT-2009-005, EPFL, 2009.
[18] M. Gligoric, T. Gvero, S. Lauterburg, D. Marinov, and S. Khurshid. Optimizing generation of object graphs in Java PathFinder. In *ICST*, 2009.
[19] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.
[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.
[21] W. Grieskamp, X. Qu, X. Wei, N. Kicillof, and M. B. Cohen. Interaction coverage meets path coverage by SMT constraint solving. In *TestCom/FATES*, 2009.
[22] R. Iosif. Symmetry reduction criteria for software model checking. In *SPIN*, 2002.
[23] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov. Reducing the costs of bounded-exhaustive testing. In *FASE*, 2009.
[24] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *J-ASE*, 11(4), 2004.
[25] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.
[26] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.
[27] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, 2006.
[28] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005.
[29] 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.
[30] W. M. McKeeman. Differential testing for software. *J-DTJ*, 10(1), 1998.
[31] T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *HVC*, 2007.
[32] W. F. Opdyke and R. E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA*, 1990.
[33] C. Pasareanu, R. Pelánek, and W. Visser. Predicate abstraction with under-approximation refinement. *J-LMCS*, 3(1), 2007.
[34] 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.
[35] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.
[36] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA*, 2004.
[37] 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.
[38] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, 2008.
[39] F. Tip. Refactoring using type constraints. In *SAS*, 2007.
[40] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *J-ASE*, 10(2), 2003.
[41] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, 2004.
[42] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, 2006.