Automated Testing of Refactoring Engines

Brett Daniel Danny Dig Kely Garcia Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{bdaniel3, dig, kgarcia2, marinov}@cs.uiuc.edu

ABSTRACT

Refactorings are behavior-preserving program transformations that improve the design of a program. Refactoring engines are tools that automate the application of refactorings: first the user chooses a refactoring to apply, then the engine checks if the transformation is safe, and if so, transforms the program. Refactoring engines are a key component of modern IDEs, and programmers rely on them to perform refactorings. A bug in the refactoring engine can have severe consequences as it can erroneously change large bodies of source code.

We present a technique for automated testing of refactoring engines. Test inputs for refactoring engines are programs. The core of our technique is a framework for iterative generation of structurally complex test inputs. We instantiate the framework to generate abstract syntax trees that represent Java programs. We also create several kinds of oracles to automatically check that the refactoring engine transformed the generated program correctly. We have applied our technique to testing Eclipse and NetBeans, two popular open-source IDEs for Java, and we have exposed 21 new bugs in Eclipse and 24 new bugs in NetBeans.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*

General Terms: Verification

Keywords: Automated testing, bounded-exhaustive testing, imperative generators, test data generation, refactoring engines

1. INTRODUCTION

Refactoring [9] is a disciplined technique of applying behavior-preserving transformations to a program with the intent of improving its design. Examples of refactorings include renaming a program element to better convey its meaning, replacing field references with calls to accessor methods, splitting large classes, moving methods to different classes, or extracting duplicated code into a new method. Each refactoring has a name, a set of preconditions, and a set of specific transformations to perform [19].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

Refactoring engines are tools that automate the application of refactorings. The programmer need only select which refactoring to apply, and the engine will automatically check the preconditions and apply the transformations across the entire program if the preconditions are satisfied. Refactoring is gaining popularity, as evidenced by the inclusion of refactoring engines in modern IDEs such as Eclipse (`http://www.eclipse.org`) or NetBeans (`http://www.netbeans.org`) for Java. Refactoring is also a key practice of agile software development methodologies, such as eXtreme Programming [4], whose success prompts even more developers to use refactoring engines on a regular basis. Indeed, the common wisdom views the use of refactoring engines as one of the safest ways of transforming a program, since manual refactoring is error-prone.

It is important that refactoring engines be reliable—a bug in a refactoring engine can silently introduce bugs in the refactored program and lead to difficult debugging sessions. If the original program compiles but the refactored program does not, the refactoring is obviously incorrect and can be easily undone. However, if the refactoring engine erroneously produces a refactored program that compiles but does not preserve the semantics of the original program, this can have severe consequences.

Since refactoring engines are very complex and must be reliable, developers of refactoring engines have invested heavily in testing. For example, Eclipse version 3.2 has over 2,600 unit tests for refactorings (publicly available from the Eclipse CVS repository). Conventionally, testing a refactoring engine involves creating input programs by hand along with their expected outputs, each of which is either a refactored program or an expected precondition failure. The developers then execute these tests automatically with a tool such as JUnit [10]. Writing such tests manually is tedious and results in incomplete test suites, potentially leaving many hidden bugs in refactoring engines.

We present a technique that automates testing of refactoring engines, both generation of test inputs and checking of test outputs. The core of our technique is a general framework for iterative generation of structurally complex test inputs. We instantiate the general framework in a library called *ASTGen*. ASTGen allows developers to write *imperative generators* whose executions produce input programs for refactoring engines. More precisely, ASTGen offers a library of generic, reusable, and composable generators that produce abstract syntax trees (ASTs). Using ASTGen, a developer can focus on the creative aspects of testing rather than the mechanical production of test inputs. Instead of manually writing input programs, a developer writes a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. For example, to test the RenameField refactoring, the input program should have a

class that declares fields and variables with potential name clashes. Our generators systematically produce a large number of programs that satisfy such constraints.

ASTGen follows the *bounded-exhaustive* approach [5, 11, 12, 16, 22] for exhaustively testing all inputs within the given bound. This approach covers all "corner cases" within the given bound. In contrast, manual testing requires identifying each corner case and covering it with a manually written test. Bounded-exhaustive testing has not been used before for test inputs as complex as Java programs, and the approach of imperative generators introduced in this paper differs from the previous techniques using declarative generators [5, 11, 12, 16, 22]. Section 7 further discusses related work.

An important problem in automated generation of test inputs is automated checking of outputs, also known as the *oracle problem*. Our technique uses a variety of oracles. The simplest oracles check that the refactoring engine does not crash (i.e., does not throw an uncaught exception) and that the refactored program compiles. More advanced oracles take into account the semantics of the refactoring and check specific properties such as invertibility, e.g., renaming an entity from A to B and then back from B to A produces the same starting input program. The oracles also check structural properties, e.g., moving an entity should indeed create the entity in the new location. Finally, we use *differential testing* [18] in which one implementation serves as the oracle for another implementation. Specifically, we run the same input programs on Eclipse and NetBeans and compare their refactored outputs or precondition violations. Section 5.1 presents our oracles in detail.

This paper makes three main contributions.

Framework for imperative generators: We present a novel framework for generation of structurally complex test inputs. Our framework uses imperative generators that specify *how* the inputs should be generated. Previous work [5, 11, 12, 16, 22] has mostly used declarative generators that describe *what* the inputs look like and thus require potentially expensive search to generate the actual inputs.

Instantiation for generating ASTs: We instantiate the general framework in ASTGen, a library that can generate abstract syntax trees (ASTs) representing Java programs. Our instantiation provides basic generators that follow the structure of simple ASTs and more complex generators that generate entire programs used as test inputs for refactoring engines.

Case study: We have used ASTGen to test several refactorings in Eclipse and NetBeans, two popular open-source IDEs for Java. We have implemented automatic execution of refactoring engines on the input programs that our generators produce. We have also implemented several oracles to verify that refactorings complete as expected. So far, our experiments have discovered 21 new bugs in Eclipse and 24 new bugs in NetBeans. We have reported these bugs in the bug-tracking systems of both IDEs, and the NetBeans developers have already fixed 2 of these bugs, declared 2 reports as duplicates, and confirmed 19 (all but 1) other reports as real bugs. Eclipse developers confirmed 20 (all but 1) reports as real bugs.

Our ASTGen code, all experimental results, and the reported bugs are publicly available online from the ASTGen web page at `http://mir.cs.uiuc.edu/astgen`.

2. EXAMPLE

We use the EncapsulateField refactoring as an illustrative example. This refactoring replaces all references to a field with accesses through setter and getter methods. The EncapsulateField refactoring takes as input the name of the field to encapsulate and the names of the new getter and setter methods. It performs the following transformations:

```
// before refactoring
class A {
  public int f;
  void m(int i) {
    f = i * f;
  }
}
```

```
// after refactoring
class A {
  private int f;
  void m(int i) {
    setF(i * getF());
  }

  public int getF() {
    return this.f;
  }
  public void setF(int f) {
    this.f = f;
  }
}
```

Figure 1: Example EncapsulateField refactoring

- creates a public getter method that returns the field's value
- creates a public setter method that updates the field's value to a given parameter's value
- replaces all field reads with calls to the getter method
- replaces all field writes with calls to the setter method
- changes the field's access modifier to private.

The EncapsulateField refactoring checks several preconditions, including that the code does not already contain accessor methods and that these methods are applicable to the expressions in which the field appears. Figure 1 shows a sample program before and after encapsulating the field `f` into the `getF` and `setF` methods.

We next discuss three of the generators that we wrote to test EncapsulateField. These generators illustrate how to generate simple ASTs, how to generate ASTs that satisfy involved semantic constraints, and how to combine simpler generators into more complex generators. We also present two bugs that these generators reveal, one each in Eclipse and NetBeans.

ASTGen allows the tester to write generators that can exhaustively generate programs containing field references. The tester has an intuition for which programs to generate, but it is quite tedious to manually write a large number of input programs that cover all kinds of field references. Using ASTGen, the tester can write a generator that produces many such programs, each of which is a class that contains a field and a method that references the field in all kinds of relevant expressions. Section 4.2.1 describes in detail how to write this generator, effectively codifying the tester's intuition into automatic generation. This generator is not only useful for testing EncapsulateField but can also be reused to test other refactorings that operate on fields such as RenameField and PushDownField.

Our generator of programs with field references produces, among others, the program in Figure 2 that reveals a bug in NetBeans. In this case, the parentheses around the field reference cause the refactoring engine to leave the field reference unencapsulated. This omitted encapsulation could cause problems if the developer wishes to add logic to the accessor methods, since the unencapsulated field reference would not trigger this additional logic. Our Differential Oracle (Section 5.1) caught this bug since NetBeans and Eclipse produced different refactored programs. Note that the "refactored" program from NetBeans compiles, so a simple oracle that only checks compilation does not catch the bug.

Another generator produces two classes, say `A` and `B`, that exhibit all possible relationships involving (1) class inheritance, (2) containment (i.e., inner or local class), or (3) class name reference. This generator produces many pairs of classes, including the pair shown in Figure 3, which illustrates all three relationship types:

```
// before refactoring
class A {
  int f;
  void m() {
    (new A().f) = 0;
  }
}
```

```
// after refactoring
class A {
  private int f;
  void m() {
    (new A().f) = 0;
  }

  ... getF ...
  ... setF ...
}
```

Figure 2: EncapsulateField bug in NetBeans: a field access remains unencapsulated

```
class A {
  class B extends A {
    void m(A a) {}
  }
}
```

Figure 3: Two classes illustrating all three relationship types

`B` inherits from `A`, `B` is an inner class of `A`, and `B` references `A` via method parameter. One can reuse this generator to test several types of refactorings that depend on class name or location, including RenameClass, MemberToTop, and PushDownField.

The power of ASTGen appears when building more complex generators from simpler ones. The developer can compose the previous two generators into a third generator that generates even more expressive programs in which one class declares a field and the other class references it in various ways. This generator outputs the program in Figure 4 that reveals a bug in Eclipse. In this case, the refactoring engine mistakenly identifies the `super.f` expression as a field read. The DoesNotCompile Oracle quickly determines this is a bug.

3. FRAMEWORK

This section describes our imperative approach to test data generation and presents some important design decisions of the general framework that we built. We first provide motivation for imperative generation, then describe basic generators, and finally show how to compose them. We present only the parts of the framework necessary to discuss ASTGen. Section 4 presents how we instantiate the framework into ASTGen for generation of Java programs.

3.1 Why Imperative Generation?

Our framework for test data generation is imperative, iterative, and bounded-exhaustive. It is *imperative* in that the tester defines how to build input data; *iterative* in that it generates inputs lazily, one at a time; and *bounded-exhaustive* in that it systematically explores the entire combinatorial space of a given set of generators. We believe that this approach offers several benefits:

- **Easy to understand:** Testers intuitively grasp the idea of looping over a set of generated inputs. It is a natural extension to the hand-written tests that testers usually write.
- **Easy to compose:** Testers can combine generators to create complex data or to tailor data generation to a particular testing domain. Our framework is abstract and generic, allowing testers to combine generators in an arbitrary fashion.
- **Scales well with data size:** Testers can build very large and complex data structures with a small number of generators.
- **Scales well with amount of data:** Generators produce inputs lazily, one at a time, so for generation of a large number of inputs, there is no time overhead for "pre-generation" of all inputs or space overhead for storing them all.
- **Catches corner cases:** Bounded-exhaustive testing covers all inputs within a given bound [5, 12, 22], including those that random testing [7, 15] may miss or that testers are unaware of.

```
// before refactoring
class A {
  int f;
}


class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
// after refactoring
class A {
  private int f;
  ... setF ...
  ... getF ...
}

class B extends A {
  void m() {
    getF() = 0;
  }
}
```

Figure 4: EncapsulateField bug in Eclipse: a write access encapsulated into a getter method

Additionally, we developed our framework in Java, so testers do not have to learn any new languages or syntax, and they can exploit the full power of a general-purpose language. The main disadvantage of our approach is that testers have to write *how* to generate test inputs instead of writing *what* the test inputs should look like. We discuss more in Section 7.

3.2 Basic Generators

We define an iterative generator as an iterator that produces a value of some type `T` every time a method `next` is called. The simplified interface for a generator is the following:

```
interface IGenerator<T> {
  T next();
  T current();
  boolean hasNext();
  void reset();
  boolean isReset();
}
```

The `current` method returns the previously-generated value without advancing the generator. The `hasNext` method checks whether the generator can output any more values; if it returns `false`, future calls to `next` are undefined. The `reset` and `isReset` methods allow repeating the sequence of values that the generator produces. Our generators support Java's `Iterable` interface, making it easy to loop over all generated values using the following pattern:

```
IGenerator<T> valueGen = ...;
for (T value : valueGen) { doTest(value); }
```

We built several simple generators that implement the `IGenerator` interface. The `Literal` class is the simplest example that produces a single value.

The `Chain` class takes a number of values or other generators and produces all values in order, for example:

```
Literal<String> literal = new Literal<String>("a");

Chain<String> chain1 = new Chain<String>("b", "c", "d");

Chain<String> chain2 = new Chain<String>();
chain2.add(literal);
chain2.add(chain1);
chain2.add("e");

for (String s : chain2) { System.out.print(s + " "); }
// Outputs: a b c d e
```

3.3 Generator Composition

`Literal`, `Chain`, and other simple generators are useful, but our framework's true power appears when linking generators together. When we build a more complex iterative generator from two or more child generators, we need to address two concerns: *generator iteration*, which determines how to iterate children when iterating the main generator, and *data composition*, which determines how to build the value for the main generator from the values of children. We first present a simple example that combines two children using the `reset`, `isReset`, and `current` methods. We then describe how our actual implementation decouples the two concerns and revisit the example.

Consider a composite generator that exhaustively produces pairs of values given two children that produce left and right components of the pairs. Let the left and right children be `lg` and `rg`, respectively, and the composite generator `cg`. If `lg` produces two values $[m, n]$ and `rg` produces two values $[x, y]$, then we want `cg` to iteratively produce their cross product: $[(m, x), (n, x), (m, y), (n, y)]$. Each time `next` is called on `cg`, `cg` iterates `lg`. When `lg` reaches the "end" of its values, `cg` resets `lg` to the first value and iterates `rg`:

```
class NaivePairGenerator
    implements IGenerator<Pair<L, R>> {
  IGenerator<L> lg;
  IGenerator<R> rg;
  ... constructors and accessors ...

  Pair<L, R> next() {
    if (!lg.hasNext()) {
      lg.reset();
      if (!rg.hasNext()) {
        error();
      }
      rg.next();
    }
    lg.next();
    if (rg.isReset()) {
      rg.next();
    }

    return new Pair<L, R>(lg.current(), rg.current());
  }
  ... other methods from IGenerator ...
}
```

This method `next` performs both generator iteration (on `lg` and `rg`) and data composition (by creating a new `Pair`). Our implementation actually separates these two concerns. The abstract class `CompositeGenerator` implements the generator iteration, basically generalizing the method shown above to an arbitrary number of children. This class delegates the data composition to its subclasses:

```
abstract class CompositeGenerator<T>
    implements IGenerator<T> {
  // return all children for generator iteration
  abstract List<IGenerator> getChildren();
  // perform the data composition
  abstract T composeData();
  ... implemented IGenerator methods ...
}
```

This decoupling makes it easier to develop composite generators since they need not reimplement iteration. The `CompositeGenerator` iterates the children returned from the `getChildren` method, and the `composeData` method implements data composition. We can now implement a generator for pairs simply by implementing the two methods:

```
class PairGenerator<L, R>
    extends CompositeGenerator<Pair<L, R>> {
  IGenerator<L> lg;
  IGenerator<R> rg;
  ... constructors and accessors ...

  List<IGenerator> getChildren() {
    return Arrays.asList(lg, rg);
  }

  Pair<L, R> composeData() {
    return new Pair<L, R>(lg.current(), rg.current());
  }
}
```

`PairGenerator` also illustrates the abstract and generic aspects of our framework. It is common to declare child generators as `IGenerator` objects, allowing the caller to reuse a particular generator in many contexts by simply supplying child generators that implement the interface. Unlike grammars, which remain static once written, this abstractness allows generators to be specialized for many different contexts.

Furthermore, the `CompositeGenerator` class is parametrized by the type of values that the subclass generates. In the example, it produces `Pair` objects that are further parametrized by the type of the left and right values. These generic signatures, which we use throughout the framework, allow one to compose generators more easily and verify that the compositions are type-correct.

Putting it all together, one can instantiate and use a `PairGenerator` in the following manner:

```
// child generators
Chain<String> leftGen = new Chain<String>("m", "n");
Chain<String> rightGen = new Chain<String>("x", "y");

// parent generator
PairGenerator<String, String> pairGen =
  new PairGenerator<String, String>(leftGen, rightGen);

// generate values
for (Pair<String, String> pair : pairGen) {
  System.out.print(pair);
}
// Outputs: (m,x) (n,x) (m,y) (n,y)
```

3.4 Dependent Composition

In the previous `PairGenerator` example, the values of child generators are independent of each other. We next describe how our framework can handle dependent values.

As an illustrative example, suppose that we have two generators X and Y such that X produces integers $[1, 2]$, and Y adds or subtracts one from the current value of X. The cross product of these two generators is $[(1, 2), (2, 3), (1, 0), (2, 1)]$. To implement an imperative generator for such pairs, we can consider three options. First, we can "pass" the current value of X to Y, but this would require iterating X and retrieving its value before iterating Y. As discussed in the previous section, `CompositeGenerator` decouples generator iteration and data composition, so a subclass has no control over which generator is iterated first. Second, Y can iterate X directly, but this introduces unwanted coupling if we hope to reuse X and Y elsewhere. Also, it might lead to iteration problems if multiple generators iterate X. Third, we can have Y produce objects that represent functions. We follow this approach.

Conceptually, Y should produce the values $[\lambda x.x+1, \lambda x.x-1]$. This approach allows the framework to iterate X and Y independently and in any order. The data composition step passes X's current value to the function produced by Y and returns the resulting pair of values. Since functions are not first-class entities in Java, we use *method objects* [3] to implement the Y generator:

```
interface YFunc { int execute(int xVal); }

IGenerator<YFunc> yGen = new Chain<YFunc>(
  new YFunc() { int execute(int xVal) { return xVal + 1; } },
  new YFunc() { int execute(int xVal) { return xVal - 1; } });
```

The dependent pair generator then uses this Y generator and defines the appropriate data composition:

```
class DependentPairGenerator
    extends CompositeGenerator<Pair> {
  IGenerator<Integer> x;
  IGenerator<YFunc> y;
  ... constructors and accessors ...

  Pair composeData() {
    // get child values
    int xVal = x.current();
    YFunc yFunc = y.current();

    // compute dependent values
    int yVal = yFunc.execute(xVal);

    return new Pair(xVal, yVal);
  }
}
```

4. INSTANTIATION FOR ASTS

ASTGen in an instance of our general framework used to produce abstract syntax trees (ASTs) for testing refactorings. We first show how to implement a generator for a simple syntax element. We then show how to implement more complex AST generators such as those described in Section 2.

4.1 Simple AST Generators

We first discuss AST generators that simply mirror the structure of their corresponding AST nodes. As an illustrative example, we use an AST node that represent a (much-simplified) field declaration in Java:

```
class FieldDeclaration {
   Modifier modifier;
   Type type;
   Identifier identifier;
   ... constructors and accessors ...
}
```

This node has three other AST nodes as children: `modifier`, `type`, and `identifier`. Figure 5 shows `FieldDeclarationGenerator` that contains a child generator for each of the three AST node children. Like `PairGenerator` discussed in Section 3.3, `FieldDeclarationGenerator` extends `CompositeGenerator`.

To use the generator, the tester can initialize it by setting each child generator to an `IGenerator` of the correct type as illustrated in Figure 6. For simplicity, we show the Java syntax elements (e.g., `public`, `int`) as if they were defined as variables, rather than showing the code used to build the AST nodes for these elements. The tester can use the initialized `FieldDeclarationGenerator` to produce test data or to construct larger AST generators that require `FieldDeclaration` objects.

The specific instantiation of `FieldDeclarationGenerator` shown in Figure 6 effectively corresponds to the following grammar for declarations:

```
<FieldDeclaration> ::= <Modifier> <Type> <Identifier> ";"
        <Modifier> ::= "public" | "private"
            <Type> ::= "int" | "boolean"
      <Identifier> ::= "someField" | "anotherField"
```

For simple generators, it is more succinct to write a grammar than the corresponding Java code for generators. However, when generated AST nodes should satisfy more complex constraints (the simplest being, say, that an `int` field should always be `private`), it is necessary to express these constraints in a language outside of the grammar. ASTGen uses the same language, Java, to express both the constraints and the generators.

```
class FieldDeclarationGenerator
    extends CompositeGenerator<FieldDeclaration> {
  IGenerator<Modifier> modifierGen;
  IGenerator<Type> typeGen;
  IGenerator<Identifier> idGen;
  ... constructors and accessors ...

  List<IGenerator> getChildren() {
    return Arrays.asList(modifierGen, typeGen, idGen);
  }

  FieldDeclaration composeData() {
    FieldDeclaration generated = new FieldDeclaration();
    generated.setModifier(modifierGen.current());
    generated.setType(typeGen.current());
    generated.setIdentifier(idGen.current());
    return generated;
  }
}
```

Figure 5: Field declaration generator

```
IGenerator<Modifier> modifierGen =
  new Chain<Modifier>(public, private);
IGenerator<Type> typeGen =
  new Chain<Type>(int, boolean);
IGenerator<Identifier> idGen =
  new Chain<Identifier>(someField, anotherField);
FieldDeclarationGenerator fieldDeclGen =
  new FieldDeclarationGenerator(modifierGen, typeGen, idGen);
```

Figure 6: Example initialization of FieldDeclarationGenerator

We have implemented basic generators for 29 common Java syntax elements. These generators encapsulate AST generation and thus increase reusability in that one does not need to define hard-coded values for a `Chain` generator. For example, in Figure 6 we could have used `ModifierGenerator` rather than a `Chain` containing values `public` and `private`.

4.2 Complex AST Generators

We next show how our iterative approach can generate more complex AST nodes. We use as examples the three generators discussed in Section 2.

4.2.1 Field Reference Generator

The `FieldReferenceGenerator` generates classes, each of which contains a field and a method that references the field in some way. It can produce several thousand classes, one of which is shown in Figure 2. It has the following five child generators:

- An `IGenerator<FieldDeclaration>` (such as a `FieldDeclarationGenerator` from Figure 5), provided by the caller. This generator produced the `int f;` declaration in Figure 2.
- A `FieldReferenceExpressionGenerator` uses the field name from the field declaration to build a simple expression that references the field. If the field is `f` in class `A`, this expression can be `f`, `this.f`, `A.this.f`, or, as in Figure 2, `new A().f`.
- A `ParenthesizingExpressionGenerator` optionally parenthesizes an expression. This generator parenthesized the referencing expression yielding `(new A().f)` in Figure 2.
- A `NestedExpressionGenerator` nests the optionally parenthesized expression in one of the many possible expressions applicable to the field's type. In Figure 2, the generated expression is the assignment expression: `(new A().f) = 0`. Since the field has type `int`, other applicable expressions include the binary arithmetic operators, unary operators, and many others.

```
// Get method objects
FieldDeclaration fieldDecl = fieldDeclGen.current();
FieldReferenceExpressionMethObj fieldRefExprMO =
  fieldRefExprGen.current();
ParenthesizingExpressionMethObj parenExprMO =
  parenExprGen.current();
NestedExpressionMethObj nestedExprMO =
  nestedExprGen.current();
ExpressionInStatementMethObj exprInStmtMO =
  exprInStmtGen.current();

// Call method objects
Expression fieldRefExpr = fieldRefExprMO.fill(fieldDecl);
Expression parenExpr = parenExprMO.fill(fieldRefExpr);
Expression nestedExpr = nestedExprMO.fill(parenExpr);
Statement exprInStmt = exprInStmtMO.fill(nestedExpr);

// Build AST to return
MethodDeclaration methodDecl = makeMethod("m");
methodDecl.addStatement(exprInStmt);

TypeDeclaration typeDecl = makeClass("A");
typeDecl.addField(fieldDecl);
typeDecl.addMethod(methodDecl);
```

Figure 7: FieldReferenceGenerator generation

- An `ExpressionInStatementGenerator` nests the full expression in one of many types of statements. In Figure 2, the statement simply contains the expression itself, but it can also generate branching, looping, or other statements.

It is interesting to note that the `NestedExpressionGenerator` can accept another `NestedExpressionGenerator`, allowing one to create expressions nested within each other to an arbitrary depth. Indeed, our initial instantiation of the `FieldReferenceGenerator` included a `NestedExpressionGenerator` in place of the `ParenthesizingExpressionGenerator`, but we found that the resulting combinatorial explosion increased testing time substantially without yielding any new bugs. This illustrates how the tester can (and also that the tester should) tailor generators (by using different child generators) to produce data applicable to a particular test target.

The last four child generators are examples of *dependent generators* (see Section 3.4). The parent `FieldReferenceGenerator` produces an AST by first retrieving the method objects from each of its child generators. Then, it builds AST fragments by passing intermediate results down the sequence of method objects. Finally, it creates a top-level node (`TypeDeclaration`) that represents a class. Figure 7 lists the pseudocode for this procedure.

4.2.2 Class Relationship Generator

The `ClassRelationshipGenerator` produces combinations of inheritance, class name reference, or location-based (i.e., inner or local class) relationships between two generated classes. Given two literal class generators, this generator produces several hundred relationship pairs, one of which is shown in Figure 3. It has the following child generators:

- Two `IGenerator<TypeDeclaration>` generators, provided by the caller, that generate the classes to be related.
- An `InheritanceGenerator` determines whether one class inherits from the other.
- A `ClassNameReferenceGenerator` similar to the `FieldReferenceGenerator` from Section 4.2.1. It generates many expressions, statements, and declarations that can contain a reference to the name of a class. In Figure 3, this generator produced the method declaration with a parameter type `A`.
- A `LocationGenerator` determines where one class is located in relation to the other. In Figure 3, this generator specified that `B` is an inner class of `A`. Other possible locations are local (in which a class is declared inside a method) and separate (in which both classes are top-level elements).
- Three `DirectionGenerator` generators that determine the direction in which the three relationships "point". In Figure 3, all relationships are in the `B`-to-`A` direction: `B` inherits from `A`, `B` is an inner class of `A`, and `B` references `A` through a method parameter.

The three generators for relationships (`InheritanceGenerator`, `ClassNameReferenceGenerator`, and `LocationGenerator`) all generate method objects that consume two generated `TypeDeclaration` nodes and a generated `Direction`, and return the same two nodes related in a particular way. Note that these method objects need to modify the nodes generated by other child generators. After applying all three generated method objects, the pair of classes is returned to the caller.

Due to their exhaustive nature, generators often produce programs that do not compile. For example, depending on the combination of direction and relationship, certain generated class relationships may be invalid. The following code illustrates one such invalid relationship in which `B` is related to `A` by location, but `A` is related to `B` by inheritance:

```
class A extends B {
  class B {}
}
```

We can overcome this problem in three ways. First, we can filter invalid data by testing the current values of all child generators. In this example, we can "skip" the generated value if the `LocationGenerator`'s current value is "inner" in the `B`-to-`A` direction and the `InheritanceGenerator`'s current value is "extends" in the `A`-to-`B` direction. Second, the caller can limit the generation to only those programs applicable to a particular task. We shall see in the next section that the `DoubleClassFieldReferenceGenerator` generates classes in all location and inheritance relationships in all directions, but omits the class name reference relationship because it is irrelevant to testing field references. Finally, we can delegate to the compiler to filter out any generated programs that do not compile.

4.2.3 Double-Class Field Reference Generator

The `DoubleClassFieldReferenceGenerator` produces all possible class relationships in which one class declares a field and the other references it. When supplied the simplest possible class and field generators, this generator produces over 14,000 programs, one of which is shown in Figure 4.

This generator combines aspects of the other two complex generators that we have discussed. First, it uses the `ClassRelationshipGenerator` to generate the inheritance and location relationships between the supplied classes. Then, it uses the `SingleClassFieldReference` to generate references to the field. Construction of the AST proceeds similarly to the previous two cases. Note that for this generator, like for `ClassRelationshipGenerator`, some child generators need to use the nodes generated by other child generators.

5. TESTING REFACTORING ENGINES

We next present how we test the refactoring engines in Eclipse and NetBeans with the input programs that ASTGen generates. We describe the oracles that we use to verify whether a refactoring has completed correctly. We also discuss briefly how to automatically run Eclipse and NetBeans on the generated programs.

5.1 Oracles

An important problem in automated generation of test inputs is automated checking of outputs, also known as the *oracle problem*. A seemingly ideal oracle for a refactoring engine would tell whether an input program and its refactored version have the same semantics. However, checking that two programs have the same semantics is undecidable in general. Moreover, even if the two programs have the same semantics, the refactoring engine might not have performed the required changes on the program—a trivial "identity" engine that does not change any program always produces semantically equivalent refactored programs but does not implement refactoring correctly. (Recall the bug from Figure 2 where NetBeans left a field unencapsulated.) Fortunately, refactorings are program transformations that make well-defined structural changes, so we can still check several useful properties of a refactored program. We have implemented six oracles.

DoesCrash Oracle: Our simplest oracle checks that the refactoring engine does not throw an uncaught exception. Such an oracle is often used as a sanity check in "smoke testing".

DoesNotCompile Oracle: Our next oracle checks that the refactored program compiles. We filter all generated programs through the compiler and pass to the refactoring engine only those input programs that compile. A correct refactoring should thus always produce output programs that compile.

WarningStatus Oracle: Refactoring engines should warn the user when a refactoring might change the semantics of the program. This oracle determines if the refactoring engine produces a warning status message after checking the preconditions of a specific refactoring. This oracle is particularly useful with the generators that intentionally create programs that do not meet the preconditions. For example, for testing RenameField refactoring, our generators create programs such that the new field name would clash with names of other fields or variables. A refactoring engine should find that such programs do not meet the preconditions of the refactoring.

Inverse Oracle: Refactorings are invertible program transformations: given a transformation done by one refactoring, we can find another refactoring that "undoes" the transformation on the program. For example, renaming a program entity from A to B and then renaming again from B to A should produce the same original program. Due to the implementation of refactoring engines, these programs do not need to be exactly the same when viewed as sequence of characters. We can compare the original and twice-refactored programs by determining whether their ASTs are equivalent.

We implemented an *AST Comparator* that performs an approximate comparison: it first normalizes two ASTs by sorting the methods and fields by name and then compares the method bodies and field expressions of the appropriate pairs of methods/fields.

Custom Oracle: We implemented several refactoring-specific oracles. These oracles are aware of the structural changes that their corresponding refactorings should make and thus check that the refactored program exhibits the expected changes. For example, we can verify that RenameField leaves no occurrences of the old field name anywhere in the AST.

Differential Oracle: The last oracle we implemented is used in differential testing [18]. This oracle takes an input program and a refactoring and feeds this pair to both Eclipse and NetBeans. It then takes the output programs returned by the two engines and checks whether their ASTs are equivalent using the AST Comparator described above. If the two ASTs differ, a human inspects the two output programs to check whether the difference is caused by a bug in one of the refactoring engines.

```
String fieldName = "f";
FieldDeclarationGenerator fieldDeclGen =
  new FieldDeclarationGenerator(fieldName);
IGenerator<Program> testGen = new ...(fieldDeclGen);
for (Program in : testGen) {
  if (!in.compiles) { continue; }

  Refactoring r = new EncapsulateFieldRefactoring();
  r.setTargetField(fieldName);
  Program out = r.performRefactoring(in);

  checkOracles(out);
}
```

Figure 8: Pseudocode for testing EncapsulateField

5.2 Running Refactorings

We next illustrate how to run the Eclipse and NetBeans refactoring engines on the input programs that ASTGen generates. We use the EncapsulateField example. Figure 8 shows the pseudocode for this process. It first creates a `FieldDeclarationGenerator` initialized with the name of the field we expect to encapsulate. It then passes this generator to a program generator like those described in Section 4.2. For each program that the generator produces, the code first tests if it compiles. If so, the code then instantiates a refactoring provided by the IDE, initializes it with the field name, and invokes the refactoring engine. The engine yields a refactored program that the code passes to each of the oracles.

We implemented this process in Eclipse as a custom plug-in that uses the platform's built-in test harness. In NetBeans, we extended the existing unit test suite.

6. CASE STUDY

We next present the results of using ASTGen to test refactoring engines. Specifically, our goal is to find and report bugs in Eclipse and NetBeans. We have tested several refactorings and found 21 bugs in Eclipse and 24 in NetBeans. We list the refactorings tested, present the generators used for those refactorings, discuss the generation results, comment on the effort required to write some generators, discuss how well various oracles performed, and summarize the reported bugs.

6.1 Refactorings Tested

We tested the following eight refactorings:

- **Rename:** Rename a class, method, or field and update all references to it.
- **EncapsulateField:** Replace every reference of a field with an accessor method.
- **PushDownField:** Move a field from a superclass to all subclasses.
- **PullUpField:** Move a field from a subclass to some superclass.
- **PushDownMethod:** Move a method from a superclass to all subclasses.
- **PullUpMethod:** Move a method from a subclass to some superclass.
- **ChangeSignature:** Change a method signature by changing its return type, adding parameters, or removing parameters.
- **MemberToTop:** Move an inner class out of its containing class and declare it in a top level class.

Refactoring	Generation				Oracles						**Bugs Reported**	
	Generator	TGI	Time	CI	WS		DNC		C/I	Diff.		
					Ecl	NB	Ecl	NB			Ecl	NB
Rename(Class)	ClassRelationships	108	1:02	88	0	0	0	0	0	0	0	0
Rename(Method)	MethodReference	9540	89:12	9540	0	0	0	0	0	0	0	0
Rename(Field)	FieldReference	3960	28:20	1512	0	0	0	304	0	40	0	1
Rename(Field)	DoubleClassFieldRef.	14850	76:55	3969	0	0	0	0	0	0	0	0
EncapsulateField	ClassArrayField	72	0:45	72	0	0	48	0	0	48	1	0
	FieldReference	3960	15:19	1512	0	0	320	432	14	121	4	3
	DoubleClassFieldRef.	14850	41:45	3969	0	0	187	256	100	511	1	2
	SingleClassTwoFields	60	1:16	48	0	0	0	0	48	15	1	0
	DoubleClassGetterSetter	576	8:45	417	216	0	162	162	18	216	3	3
PushDownField	DoubleClassFieldRef.	4635	10:56	1064	760	380	152	228	0	380	2	3
	DoubleClassParentField	360	6:50	270	246	168	18	90	0	78	1	1
PushDownMethod	DoubleClassParentMethod	960	17:11	820	784	300	16	428	0	484	2	3
PullUpField	DoubleClassChildField	60	1:14	44	0	18	10	6	0	44	1	1
	TripleClassChildField	144	3:06	108	0	42	36	20	0	42	2	2
PullUpMethod	DoubleClassChildMethod	576	14:38	448	0	176	0	48	0	224	0	1
	TripleClassChildMethod	1152	29:08	864	0	336	160	160	0	336	2	2
CS(ChangeReturnType)	MethodReference	3816	37:36	3816	1992	n/a	0	n/a	0	n/a	0	n/a
CS(RemoveParameter)	MethodReference	5724	54:29	5724	1908	0	0	0	0	0	0	0
CS(RemoveParameter)	MethodParamRef.	1680	7:11	772	772	772	0	0	0	0	0	0
MemberToTop	ClassRelationships	70	0:36	51	0	0	0	2	0	2	0	1
	DoubleClassFieldRef.	6600	29:04	2824	0	0	353	507	0	2824	1	1
										Total Bugs:	**21**	**24**

Figure 9: Refactorings tested and bugs reported; CS = ChangeSignature, Ecl = Eclipse, NB = NetBeans; Generation: TGI = Total Generated Inputs, Time is in min:sec, CI = Compilable Inputs; Oracles: WS = WarningStatus, DNC = DoesNotCompile, C/I = Custom/Inverse, Diff. = Differential

We chose these refactorings because they demonstrate a variety of refactoring targets, e.g., EncapsulateField and PushDownField target field declarations, ChangeSignature targets method declarations, and MemberToTop targets inner classes. Eclipse and NetBeans have many more refactorings, and we leave it as the future work to test more of them. We expect our approach to be general enough to test these other refactorings.

Figure 9 shows the results of our experiments. The first column lists the specific refactorings performed. The last two columns list the number of bugs reported for each. Note that even "trivial" refactorings such as Rename are susceptible to bugs. The ChangeReturnType refactoring is not available in NetBeans, so we put "n/a" in the corresponding cells.

6.2 Generators Used

The second column of Figure 9 lists the generators used to test each refactoring. Full descriptions of all generators can be found on the ASTGen website. We give a short description of five generators that we discuss in the rest of the paper:

- **FieldReference:** Generates many classes. Each class contains a field and a method that references the field in many ways. See Section 4.2.1.
- **ClassRelationship:** Generates pairs of classes that are related in many ways. See Section 4.2.2.
- **DoubleClassFieldReference:** Generates pairs of classes related in many ways. One class declares a field and the other references it in many ways. See Section 4.2.3.
- **MethodReference:** Generates many classes with two methods. One method calls the other and may overload it.
- **MethodParamReference:** Generates method declarations, each of which has a parameter referenced.

6.3 Experimental Results

The third column lists the total number of programs generated. This number is very sensitive to the way in which the tester initializes the generator. For example, a fully-exhaustive DoubleClassFieldReferenceGenerator used for EncapsulateField produces 14,850 programs, whereas a version limited to producing inheritance relationships for PushDownField produces just 4,635 programs.

The fourth column shows execution time needed both to generate all input programs *and* to perform the refactoring on these programs in Eclipse. We ran our tests on a dual-processor 1.8 GHz Dell D820 laptop with 1 GB of RAM. Performing the refactoring takes up the vast majority of the execution time. To illustrate, it takes just 13 seconds for the DoubleClassFieldReference generator to produce 14,850 programs, but executing the EncapsulateField refactoring on 3,969 compilable inputs takes about 41 minutes. In general, Eclipse executes refactorings on compilable inputs at a rate of about 100 per minute, while ASTGen generates inputs at a rate of about 1,000 per second.

The fifth column shows the number of compilable inputs. The generators that we wrote do not always produce input programs that compile, so we used filtering as described in Section 4.2.2. It is possible to write generators that produce only compilable inputs—indeed, we did so with the MethodReference generator—but it does require additional effort. For our specific application of generators to test refactoring engines, this effort was rarely justified: checking whether a program compiles is faster than performing a refactoring on it. However, this approach assumes that the compiler is correct and would not be directly applicable for testing the compiler itself. In the future, we plan to investigate improved approaches that produce only compilable inputs.

6.4 Effort to Write Generators

We next discuss our anecdotal experience with the effort required to develop generators. Section 4.1 describes how one can build a

simple generator that produces AST nodes. Building such generators is fairly straightforward. We built a large library of such simple generators, but we did not closely track our effort since we were still experimenting with the design of our framework. Therefore, we asked two colleagues to write a simple AST generator, similar to FieldDeclarationGenerator, after the design of the framework had solidified. They had no experience with ASTGen but had limited experience with the AST data structures. It took them each about an hour, including the time needed for us to briefly describe the important classes in ASTGen.

Section 4.2 describes how one can build complex generators useful for testing refactorings. We tracked the effort required to write two complex generators: MethodReferenceGenerator and MethodParamReferenceGenerator. These two generators are fairly representative because they were built on the existing generators after we already had had some experience with ASTGen.

It took the first author about two workdays to write the two generators *and* the infrastructure needed to run the four refactorings that they test. While we cannot precisely divide the time for developing generators and the infrastructure, we found the two to be roughly equal across most complex generators. Together, these two generators produce 20,760 input programs (of which 19,852 compile). We believe that the number of inputs that the generators produce is larger than even the most talented tester could produce by hand in the same amount of time. Note, however, that this does not imply that ASTGen is better than manual testing. In the future, we plan to conduct a larger empirical study to compare our approach for imperative generation with other automated testing approaches and manual testing.

6.5 Oracle Evaluation

Columns six through eleven of Figure 9 show the number of programs that each oracle flagged as potentially incorrect refactored programs. The DoesNotCompile Oracle revealed the most bugs in Eclipse, while the Differential Oracle revealed the most bugs in NetBeans. The Custom Oracle revealed one bug for EncapsulateField. We do not show a column for the DoesCrash Oracle since neither refactoring engine crashed.

We use Custom Oracle, Inverse Oracle, and Differential Oracle in sequence. We apply the first two oracles directly on the refactored program in Eclipse since these oracles operate on the program's AST. (We do not apply these oracles on the output from NetBeans.) If the two oracles flag the program, we definitely need to inspect it. We then use the Differential Oracle to compare the outputs from Eclipse and NetBeans. Whenever there is a difference, we need to manually inspect both outputs; based on whether Custom/Inverse did or did not flag the Eclipse output, we expect the bug to be in Eclipse or NetBeans, respectively. Finally, even if there is no difference but Custom/Inverse flagged the Eclipse output, we also need to inspect the NetBeans output.

The WarningStatus Oracle and Differential Oracle produce some false positives. For WarningStatus, the reason is that some generators produce programs (1) for which the refactoring is expected to find preconditions violations and (2) for which the refactoring is expected to proceed and refactor the input program. One could reduce this problem by writing generators that produce only one or the other kind of program. For Differential, the reason is that our AST Comparator (described in Section 5.1) sometimes finds two programs different even though they are semantically equivalent at the sub-method level. While the AST Comparator does not compare the programs purely syntactically, it does not (and cannot) compare their full semantics. Additionally, Differential Oracle can be triggered by the different ways in which the two engines perform refactorings. For example, when renaming a method `m`, one engine may also rename all overloaded methods, while the other may not. Neither approach is semantically incorrect, so there is no bug. In the future, we plan to better account for semantics and engine differences to reduce or eliminate the number of false positives.

6.6 Bugs Reported

The last two columns of Figure 9 show the (likely) *unique* bugs that we found with ASTGen and reported: 21 bugs in Eclipse and 24 bugs in NetBeans. ASTGen found even more bugs, but these were either already reported or fixed in the latest versions of the IDEs (Eclipse version 3.3 and NetBeans version 6.0; our testing infrastructure ran on slightly older versions).

Since ASTGen generates a large number of input programs, in a bounded-exhaustive fashion, the oracles can report many failures for each unique bug. For example, for RenameField, we reported only one unique bug in NetBeans for the 40 variations caught by the Differential Oracle.

7. RELATED WORK

There is a large body of work in the area of test-input generation. The most closely related to ASTGen are grammar-based and bounded-exhaustive testing approaches.

Grammar-based testing [13, 14, 17, 20, 21] requires the user to describe test inputs with a grammar, and the tools then generate a set of strings that belong to the grammar (or sometimes a set of strings that intentionally do not belong to the grammar). In 1972, Purdom [20] pioneered the algorithms for selecting a minimal set strings that achieve certain coverage criteria for grammars, e.g., strings that cover all terminals, all non-terminals, or all productions. More recently, Maurer [17], Sirer and Bershad [21], and Malloy and Power [14] developed tools for grammar-based generation that were used to find bugs in several applications. We can view grammar-based approaches as effectively using first-order functional programs to specify the generation. The tools interpret these programs typically to generate *random* strings that belong to the grammar.

In contrast to random generation, the approach of Lämmel and Schulte [13] and our approach *systematically* generate input data, which can often catch "corner cases" that random testing misses. Lämmel and Schulte base their approach on grammars and provide several parameters with which testers can control the "exhaustiveness" of generation of strings from the grammar. Our approach allows testers to use the full expressive power of a familiar programming language such as Java to write *imperative generators* that produce test inputs.

With our approach, testers can freely compose more basic generators into more advanced generators. Achieving such reusability with grammars is fairly hard. For example, it is not obvious how, in a grammar-based approach, one could combine the first two generators from Section 2 to obtain the third. In addition, dependent composition, discussed in Section 3.4, becomes difficult or impossible with grammars. It has been long realized that even the simplest cases require extensions to the grammar, e.g., the use of attributed grammars [8], and to generate valid programs as inputs (e.g., to test compilers) requires even further extensions to context-free grammars [2, 6].

Our general framework for imperative generation was inspired by QuickCheck [7], a Haskell library for random generation of test data. QuickCheck provides a set of basic generators (each of which is a Haskell monad) and combinators for building complex generators from simpler ones. Our framework uses Java classes instead of Haskell monads and provides bounded-exhaustive generation.

In our framework, both generators and their composition are expressed in Java, but we plan to consider other approaches for composition such as GenVoca [1].

Bounded-exhaustive testing [5, 11, 12, 16, 22] is an approach for testing code exhaustively on all inputs within the given bound. We previously developed two approaches, TestEra [12] and Korat [5], that can in principle be used for bounded-exhaustive generation of complex test inputs such as Java programs. These two approaches are *declarative*: they require the user to specify the constraints that describe *what* the test inputs look like (as well as the bound on the size of test inputs), and the tools then automatically search the (bounded) input space to generate all inputs that satisfy the constraints. TestEra requires the user to specify the constraints in a declarative language, while Korat requires the users to specify the constraints in an imperative language. In both previous approaches, the user just specifies the constraints.

The approach presented in this paper differs in that it is *imperative*: the programmer specifies *how* the test generation should proceed. The imperative approach makes the generation faster since no search is necessary. Also, the imperative approach gives the programmer more control over the generation, for example over the order of generation. Finally, the two previous declarative approaches have not been applied to generate inputs as complex as Java programs, whereas we have applied our new imperative approach to generate Java programs to test refactoring engines in Eclipse and NetBeans.

Nevertheless, the imperative approach has some disadvantages compared to the declarative approach. The declarative approach may be more appealing when the testers are not willing to write generators or the constraints are fairly simple. Also, the declarative approach always generates valid inputs, whereas most of our imperative generators rely on the compiler to filter valid inputs, which can reduce performance of generation for larger ASTs. We plan to investigate whether the two approaches can be combined such that testers can use, for various parts of the same generation, a declarative or imperative approach, depending on which one appears more appropriate for which part.

8. CONCLUSIONS

Refactoring engines have become popular because they allow programmers to quickly and (for the most part) safely change large programs. These tools also influence the culture of software development: programmers who use refactoring engines are more inclined to change large programs. Despite the high quality and widespread use of existing refactoring engines, they still contain bugs. Our goal is to help the developers of refactoring engines to find bugs and reduce their number.

We have presented a practical approach that automates testing of refactoring engines. The key part of our approach is ASTGen, a library for generating abstract syntax trees (ASTs) of Java programs. With ASTGen, testers can quickly write generators that mirror the AST nodes of Java programs. The generators can be also be composed and reused to generate complex programs. Our approach found 45 previously unreported bugs in Eclipse and NetBeans, two of the most popular refactoring engines for Java.

In the future, we plan to test more refactorings. We also plan to apply ASTGen in new domains; we believe that ASTGen can help in generating test inputs for a variety of applications that operate on ASTs, including compilers, code editors and IDEs, and program analyzers. Although we have presented generation of Java programs, the ideas behind ASTGen directly translate to languages other than Java. Finally, we plan to investigate further the approaches for generation of structurally complex inputs.

Acknowledgments: We thank Jesus DeLaTorre for his help in the initial stages of this research; Marcelo d'Amorim and Steven Lauterburg for developing some example generators with ASTGen; Ralph Johnson and the SAG group at UIUC, Don Batory, and the anonymous reviewers for their insightful comments on a previous version of this paper; Jan Becicka (NetBeans), Martin Aeschlimann and Markus Keller (Eclipse) for reviewing our bug reports. Darko also thanks Wolfram Schulte and the FSE group at Microsoft Research for their help in developing an earlier framework for imperative generation. This material is based upon work partially supported by the NSF under Grant Nos. CNS-0613665 and CNS-0615372. We also acknowledge support from Microsoft Research.

9. REFERENCES

[1] D. S. Batory and S. W. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

[2] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Trans. Software Eng.*, 8(4):343–353, July 1982.

[3] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[4] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[6] A. Celentano, S. C. Reghizzi, P. D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software - Practice and Experience*, 10(11):897–918, 1980.

[7] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

[8] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proc. of the 5th International Conference on Software Engineering (ICSE)*, pages 170–178, 1981.

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

[10] E. Gamma and K. Beck. JUnit, 1997. `http://www.junit.org`.

[11] S. Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2003.

[12] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 2004.

[13] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, pages 19–38, 2006.

[14] B. A. Malloy and J. F. Power. An interpretation of Purdom's algorithm for automatic generation of test cases. *1st Annual International Conf. on Computer and Information Science*, 2001.

[15] J. J. Marciniak. *Encyclopedia of Software Engineering*, chapter Random Testing, pages 1095–1104. Wiley-Interscience, 2001.

[16] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2004.

[17] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4), July 1990.

[18] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[19] W. F. Opdyke and R. E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *Proc. Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, Sept 1990.

[20] P. Purdom. A sentence generator for testing parsers. *Behavior and Information Technology*, 12(3):366–375, 1972.

[21] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proc. 2nd conference on Domain-specific languages*, 1999.

[22] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proc. International Symposium on Software Testing and Analysis*, 2004.