Random Testing for Higher-Order, Stateful Programs

Casey Klein
Northwestern University
clklein@eecs.northwestern.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Abstract

Testing is among the most effective tools available for finding bugs. Still, we know of no automatic technique for generating test cases that expose bugs involving a combination of mutable state and callbacks, even though objects and method overriding set up exactly that combination. For such cases, a test generator must create callbacks or subclasses that aggressively exercise side-effecting operations using combinations of generated objects.

This paper presents a new algorithm for randomly testing programs that use state and callbacks. Our algorithm exploits a combination of contracts and environment bindings to guide the test-case generator toward interesting inputs. Our prototype implementation for Racket (formerly PLT Scheme)—which has a Java-like class system, but with first-class classes as well as gbeta-like augmentable methods—uncovered dozens of bugs in a well-tested and widely used text-editor library.

We describe our approach in a precise, formal notation, borrowing the techniques used to describe operational semantics and type systems. The formalism enables us to provide a compact and self-contained explanation of the core of our technique without the ambiguity usually present in pseudo-code descriptions.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing tools; D.2.4 [*Software/Program Verification*]: Assertion checkers

General Terms Verification, Reliability

Keywords Software Testing, Random Testing, Automated Test Generation, Racket

1. Introduction

Extensibility is the hallmark of object-oriented programming. The ability to pass objects—of unknown, and possibly not yet written classes—as arguments to methods is a powerful form of openness that enables programmers to more easily abstract and write reusable code. The book *Design Patterns* (Gamma et al. 1994) is full of examples for exploiting this power. IBM's San Francisco project (Rubin et al. 1998), which claims to be the largest Java development effort in the world, is predicated on such extensibility.

The flexibility of an open system incurs a commensurate cost: effective testing is difficult, because the space of behaviors for the systems is large and complex. Automated testing should be able to help; indeed, there hundreds of papers containing techniques for automatically testing object-oriented programs. Few of those techniques, however, create objects with new behavior to supply to the program under test. The vast majority of systems assume instead that the program is closed (i.e., that the tester is supplied with all possible classes), or they are even more limited, supporting only methods that operate on first-order data.

We have developed a new algorithm for automated testing that addresses the problem of generating objects with new behaviors. Our approach follows precedents in using programmer-supplied contracts (Findler and Felleisen 2002) to guide test generation and in building up pools of objects to use as test inputs. By synthesizing and generalizing those precedents, we arrived at an algorithm whose core can be described by just a few simple test-generation rules.

We validated our algorithm by applying it to the implementation of DrRacket (Findler et al. 2002), the programming environment for Racket (Flatt and PLT June 7, 2010). DrRacket's text editor is implemented by a class that supports an extensible set of editable items, as well customization through method overriding. Editing, drawing, measuring, copying, and configuring the text editor each involve a myriad of methods over multiple objects. Many methods adjust the internal state of the editor, and many state transformations require calling multiple methods, any of which can be overridden. Designing classes and methods that enable extensibility while maintaining the integrity of the editor's state is a formidable task. Our random-testing algorithm uncovered upwards of 58 previously unknown bugs, more than half of which could only be caught by synthesizing new subclasses that exercise the editor API in particular ways.

OOPSLA/SPLASH '10 October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10. . . $10.00

```
interface BoundedStackI {
  void push(Object item);
    //@ requires this.capacity() >= 1;
    //@ ensures !this.empty();

  Object pop();
    //@ requires !this.empty()

  void push_n(int n, Object item);
    //@ requires this.capacity() >= n;

  void add_obs(ObserverI o);
  int capacity();
  boolean empty();
}

interface ObserverI {
  void after_push(BoundedStackI s,
                  Object item);
    //@ requires !s.empty();
    //@ ensures !s.empty();
}
```

Figure 1: Bounded Stack Contract

2. Contracts as oracles and generators

A *contract* is a complement to a type system and a generalization of pre-/post-condition checking. A contract on a function or method refines the type of each argument by placing additional run-time checks on it. Simple checks can be performed immediately, while other checks must be delayed until the argument is used. For example, if a function receives as an argument another function or an object, then constraints on the behavior of the function or object apply only when the function is called or the object's methods are invoked. A key property of these checks in a contract system is the proper tracking of *blame*: when a contract check fails, the responsible party can be correctly identified.

Contracts are a natural stepping stone for random test generation. In particular, blame assignment for a contract violation determines whether the corresponding contract can play the role of a test generator or an oracle. To see how this works, consider figure 1. It contains a contract specification for a stack module, written using JML notation. The `BoundedStackI` interface supports six methods: `push`, `pop`, `push_n`, `add_obs`, `capacity`, and `empty`. The `push` and `pop` methods perform the usual addition and removal of objects in the stack. The `push` method has a pre-condition that states that there must be at least one more place in the stack and has a post-condition stating that the stack is not empty. The `pop` method has a pre-condition that the stack not be empty and no post-condition. The `push_n` method provides a shorthand for filling the stack with multiple copies of `item`.

The `add_obs` method registers an observer for the stack. The final two methods, `capacity` and `empty`, query the internal state of the queue, accepting no arguments, and returning the number of available slots in the stack and whether the stack is empty, respectively.

Finally, the `ObserverI` interface has a single method, `after_push`, that is called each time the `push` method is invoked. The `after_push` method is supplied the instance of the `BoundedStackI` interface whose `push` method was invoked and the value that was pushed. Since it is called after the `push` was invoked, its pre-condition indicates that it will receive a non-empty stack. To guarantee `push`'s post-condition, however, `after_push` has is own post-condition that guarantees that the result is also non-empty.

The contracts tell us a lot about the behavior of any implementation of the `BoundedStackI` interface. The contracts also indicate who is at fault when a contract violation occurs. On one hand, if the post-condition of the `push` method fails, then the `BoundedStackI` implementation is to blame. On the other hand, if the pre-condition of `pop` fails, then the client of the `BoundedStackI` method is to blame. Thus, if we wanted to test a `BoundedStackI` implementation, the pre-condition contract limits the space of the interesting inputs, implicitly giving us a test case generator, while the post-condition contract gives us a test oracle.

In general, the blame assignment from a contract violation directly corresponds to the contract's role of as a test case generator or as a test oracle. If the program being tested would be blamed for a contract violation, the contact is a test oracle. If the test harness we generate is blamed for a contract violation, the failure provides information about how to generate test inputs. As another example, if we test clients of a `BoundedStackI` implementation, then the roles of the oracle and generator are reversed, again matching blame assignment.

The correspondence between blame, test-case generators, and oracles generalizes to higher-order interactions between components. For example, imagine that we are interested in testing how `BoundedStackI` objects interact with their observers. Specifically, consider the pre- and post-condition contracts on the `after_push` method. Since the test case generator constructs new observer objects, it is not responsible for their pre-conditions, but instead is responsible for their post-conditions. Following the correspondence with blame, the pre-condition is now playing the role of a test oracle, not a test-case generator.

In general, pre-conditions and post-conditions are not a priori tied to either oracles or test-case generators. The conditions swap roles in correspondence with the number of times the object has appeared as an input. In our stack example, the stack's pre-conditions and post-conditions are matched up with generators and oracles. In the case of the observer, because the observer is an input to the stack, the roles reverse, making pre-conditions oracles and the post-conditions generators. Of course, observers also accept stacks as inputs, and thus, for those stack objects, the roles reverse one more time, making pre-conditions generators and post-conditions oracles again.

In a manner similar to function-argument contravariance, contracts that are in negative positions (i.e., appearing an odd number of times to the left of a function arrow) play the role of generators, and contracts that are in positive positions (i.e., appearing an even number of times to the left of a function arrow) play the role of test oracles.

3. Test-generation strategy

At a high level, our strategy for automatically testing code can be represented as a function that accepts an environment (mapping identifiers to types) and a type. It randomly generates a program that uses the given environment and has the given type.

Our strategy uses two basic techniques for generating programs. The first technique creates a value directly, ignoring the environment. For base types like `int` or `boolean`, this means picking a random value of the given type. For function types, this means adding the parameter of the function to the environment and then using the larger environment to generate a function body with the type of the codomain. For object types, we first derive a new class, which amounts to overriding a subset of the methods in the object's class, where creating each overriding method is analogous to creating a new function; once we have the new class, we instantiate it to create an object.

The second technique involves the environment. Our strategy might use a variable directly from the environment, if one with the right type is available. Alternatively, it might pick an arbitrary a function or a method in the environment, recursively build something of the argument's type, invoke the function or method, bind the result to a new variable, and recur with the extended environment (still aiming to generate a program of the desired type). Note that the latter option is independent of the type being generated, and we are effectively hoping that the result of the function or method call is useful or that the function or method call changes the state of some object in an interesting way.

To see our strategy in action, consider a buggy revision of the `ObserverI` interface and `BoundedStack`, which is a buggy implementation of the `BoundedStackI` interface, shown in figure 2. Imagine that the goal is to build an `int`, and that the environment initially contains only the `BoundedStack` class.

Initially, the test generator's only options are to produce an `int` directly or to create an object `BoundedStack` and put it into the environment. Once the test generator builds an object, however, several other options become available. Imagine that the generator decides to invoke the `push_n` method. To do that, it must build an `int` for the `n` argument,

```
class BoundedStack
  implements BoundedStackI {
  int bound=10000;
  Object[] buffer = new Object[bound];
  int tos = -1;
  Set<ObserverI> observers =
     new HashSet<ObserverI>();

  public Object pop() {
    tos--;
    return buffer[tos+1];
  }
  public void push(Object item) {
    tos++;
    buffer[tos]=item;
    for (ObserverI o : observers)
      o.after_push(this,item);
  }
  public void push_n(int n,
                     Object item) {
    for (int i=0;i<n;i++)
      push(item);
  }
  public void add_obs(ObserverI o) {
    observers.add(o);
  }
  public int capacity() {
    return bound-tos;
  }
  public boolean empty() {
    return tos==-1;
  }
}

interface ObserverI {
  void after_push(BoundedStack s,
                  Object item);
}
```

Figure 2: Buggy bounded stack and observer

as well as an `Object` for the `item` argument. For the `item` argument, the generator could use the environment and pick the stack itself. For the `n` argument, the generator may decide to call the `capacity` method.

Assuming that the generator makes at least these decisions, it produces a program like the following, which triggers the first bug, an array bounds error.

```
BoundedStackI s = new BoundedStack();
s.push_n(s.capacity(),s);
```

```
e ::= λx. e                         τ ::= x:τ,e ⟶ x:τ,e
    | e(e)                                | Int
    | x                                   | Bool
    | let x = e in e
    | die(msg)
    | if (e) { e } else { e }
    | true
    | false
    | integer
```

Figure 3: Grammar

To fix this bug, the `capacity` method should return `bound-tos-1`. This bug is relatively easy to find. Our algorithm finds it 1 in 12 tries, or about once every three quarters of a second on a 3 GHz iMac.

To discover the bug in the revision of the `ObserverI` interface, the generator must decide to call the `add_obs` method, which would trigger the decision to create an implementation of the `ObserverI` interface. In the body of the generated `ObserverI`'s `after_push` method, the generator must invoke the stack's `pop` method. Finally, the generator would have to decide to push something onto the observed stack. Putting all of that together, the generator would produce code something like the following:

```
class Observer
 implements ObserverI {
 public void after_push(BoundedStack s,
                        Object item) {
     s.pop();
 }
}

BoundedStackI s = new BoundedStack();
s.add_obs(new Observer());
s.push("item");
```

This code signals an error blaming the `BoundedStack`'s `push` method for failing to establish its post-condition. The fix is to restore the pre- and post-condition contracts in the `ObserverI` interface, properly shifting the blame to the observer.

While this second bug requires the generator to make several fortuitous choices, our experiments suggest that it does so about 1 in 250 tries, i.e., about once every one and a half minutes on the same machine. To find this bug, the generator must construct and evaluate larger programs than were required to find the first bug; consequently, it performs fewer tests per second.

4. Formal model

This section makes the intuitive description in section 3 precise via a formal model. Central to our model are judgments of the form

$$\Gamma \vdash \tau \Rightarrow e$$

This judgment is derivable when our technique could generate the expression *e* of type τ from an environment Γ. For a given Γ and τ, derivations for many different *e* are typically possible, corresponding to the many different expressions that a random generator might produce.

4.1 Language

Figure 3 shows a simple functional language that we use for our formal model. This language does not include an object system or state, which we omit to simplify the formal presentation. Section 5 discusses the generalization to objects and state. For now, it is enough that λ produces a particularly simple form of an object—one with a single method, where its fields are implicit in its environment.[1]

Beyond λ, our formal language contains application expressions, variables, **let** expression to bind temporary variables, **die** expressions that terminate the program with a fixed message, **if** expressions (analogous to ...?...:... expressions in Java and C), integers, and the booleans **true** and **false**.

We omit the operational semantics for this language, as it is standard (Felleisen et al. 2009, Ch. 4).

4.2 Types

The type system for the simple functional language layers contracts on top of a standard type system. Base types are either **Int** or **Bool**. Function types combine a standard arrow type with pre-condition and post-condition expressions of type **Bool**. The domain portion $x_1{:}\tau_1{,}e_1$ of an arrow type consists of the domain's type τ_1 as well as a boolean-valued pre-condition expression e_1. The variable x_1 is bound to the input to the function and is visible in e_1 but not in τ_1. The codomain portion $\mathrm{x}_2{:}\tau_2{,}e_2$ of the arrow type is similar, with τ_2 being the codomain type and e_2 the post-condition. Both variables x_1 and x_2 are bound in the post-condition, and the variable x_1 is bound in the type τ_2. For example, assuming a few more numeric operations, the type

$$\mathrm{d{:}\textbf{Int},d} \geq 0 \rightarrow \mathrm{r{:}\textbf{Int},abs(d*d\text{-}r)} \leq 0.01$$

describes the `sqrt` function.

To capture the various constraints about free variables, and to make sure that the expressions embedded in types are well-formed, we define the judgment

$$\Gamma \vdash \tau$$

[1] While this point of view may lead you astray for other aspects of relating functional and object-oriented languages, it works well for the purposes of our random test case generation algorithm.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma\text{+}\{x_1{:}\tau_1\} \vdash e : \tau_2 \qquad \Gamma \vdash x_1{:}\tau_1,e_1 \longrightarrow x_2{:}\tau_2,e_2}{\Gamma \vdash \lambda x_1.\ e : x_1{:}\tau_1,e_1 \longrightarrow x_2{:}\tau_2,e_2} \qquad \frac{\Gamma \vdash e : x_1{:}\tau_1,e_1 \longrightarrow x_2{:}\tau_2,e_2 \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash e(e') : \tau_2} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash (\lambda x.\ e_2)(e_1) : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash \text{integer} : \mathbf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathbf{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if}\ (e_1)\ \{\ e_2\ \}\ \mathbf{else}\ \{\ e_3\ \} : \tau} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathbf{die}(\mathtt{tag}) : \tau}$$

$$\boxed{\vdash \Gamma}$$

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma\text{+}\{x{:}\tau\}}$$

$$\boxed{\Gamma \vdash \tau}$$

$$\frac{}{\Gamma \vdash \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \mathbf{Int}} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma\text{+}\{x_1{:}\tau_1\} \vdash \tau_2 \qquad \Gamma\text{+}\{x_1{:}\tau_1\} \vdash e_1 : \mathbf{Bool} \qquad \Gamma\text{+}\{x_1{:}\tau_1,\ x_2{:}\tau_2\} \vdash e_2 : \mathbf{Bool}}{\Gamma \vdash x_1{:}\tau_1,e_1 \longrightarrow x_2{:}\tau_2,e_2}$$

Figure 4: Typing rules

$$\boxed{\Gamma \vdash \tau \Rightarrow e}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash \tau \Rightarrow x}\ [\text{Var}] \qquad \frac{\Gamma(f) = x_1{:}\tau_1,e_1 \longrightarrow x_2{:}\tau_2,e_2 \qquad \Gamma \vdash \tau_1 \Rightarrow e' \qquad \Gamma\text{+}\{x_2{:}\tau_2\} \vdash \tau_3 \Rightarrow e_3}{\begin{array}{l}\Gamma \vdash \tau_3 \Rightarrow \mathbf{let}\ x_1 = e' \\ \quad \mathbf{in\ let}\ x_2 = \mathbf{if}\ (e_1)\ \{\ f(x_1)\ \}\ \mathbf{else}\ \{\ \mathbf{die}(\mathbf{gen})\ \} \\ \quad \mathbf{in\ if}\ (e_2)\ \{\ e_3\ \}\ \mathbf{else}\ \{\ \mathbf{die}(\mathtt{prog})\ \}\end{array}}\ [\text{Call}]$$

$$\frac{}{\Gamma \vdash \mathbf{Bool} \Rightarrow \mathbf{true}}\ [\text{True}]$$

$$\frac{}{\Gamma \vdash \mathbf{Bool} \Rightarrow \mathbf{false}}\ [\text{False}] \qquad \frac{\Gamma\text{+}\{x_1{:}\tau_1\} \vdash \tau_2 \Rightarrow e'}{\begin{array}{l}\Gamma \vdash x_1{:}\tau_1,e_1 \longrightarrow x_2{:}\tau_2,e_2 \Rightarrow \lambda x_1.\ \mathbf{if}\ (e_1)\ \{\ \mathbf{let}\ x_2 = e' \\ \quad\quad \mathbf{in\ if}\ (e_2)\ \{\ x_2\ \}\ \mathbf{else}\ \{\ \mathbf{die}(\mathbf{gen})\ \}\ \} \\ \quad \mathbf{else}\ \{\ \mathbf{die}(\mathtt{prog})\ \}\end{array}}\ [\text{Fn}]$$

$$\frac{}{\Gamma \vdash \mathbf{Int} \Rightarrow \text{integer}}\ [\text{Number}]$$

Figure 5: Test case generation rules

as shown in figure 4. It determines when a type τ is well-formed, given an environment Γ that maps variables to their types, and it refers to the typing rules for expressions embedded in types. Similarly, the judgment

$$\vdash \Gamma$$

holds when Γ contains only well-formed types.

The remaining rules in figure 4 give types to expressions, and are standard, except that they use $\Gamma \vdash \tau$ judgments to ensure that programs have only well-formed types.

4.3 Generating programs

The core system is given in figure 5. The rules are divided into two groups. The upper two rules use the environment to generate new expressions. The upper-left rule [Var] simply says that if the environment contains a variable bound to the type to be synthesized, then the generator's result can be a reference to that variable. The upper-right rule [Call] optimistically generates a call to some function in the environment to see if that will help with generation. Specifically, it says that if f is bound to a function of type $x_1{:}\tau_1{,}e_1 \rightarrow x_2{:}\tau_2{,}e_2$, and if generating a value of type τ_1 succeeds, then the generator can try to produce a τ_3 (the original goal) using an environment that contains a binding of x_2 to some value of type τ_2.

Since the [Call] rule generates a call to a function in the environment, it must verify that the pre-condition holds, so it produces an **if** expression that tests the pre-condition. When it fails, we know that the generator produced a bad test case, and thus the else branch of the **if** blames the generator. Dually, when the called function returns, we test the function's post-condition to see if the call reveals bug. If so, the test case aborts, blaming the function.

The remaining rules generate expressions directly, without using the environment. The [True], [False], and [Number] rules generate base values. The final rule [Fn] generates a function by adding the parameter to the function to the environment and generating a body expression of the range type. Of course, the function may not live up to its post-

$$
\dfrac{
\dfrac{\dfrac{}{\Gamma+\{d_1{:}\textbf{Int}\} \vdash \textbf{Int} \Rightarrow 2}\,[\text{Base}]}{\begin{array}{l}\Gamma \vdash \textbf{Increasing} \Rightarrow \\ \lambda d_1.\ \textbf{let}\ r_1 = 2 \\ \quad \textbf{in if}\ (d_1 \le r_1)\ \{\ r_1\ \} \\ \qquad \textbf{else}\ \{\ \textbf{die}(\textbf{gen})\ \}\end{array}}\,[\text{Fn}]
\qquad
\dfrac{\dfrac{}{\Gamma+\{g{:}\textbf{Doubling}\} \vdash \textbf{Int} \Rightarrow 2}\,[\text{Base}] \quad \dfrac{}{\Gamma+\{g{:}\textbf{Doubling},\ r_2{:}\textbf{Int}\} \vdash \textbf{Int} \Rightarrow r_2}\,[\text{Var}]}{\begin{array}{l}\Gamma+\{g{:}\textbf{Doubling}\} \vdash \textbf{Int} \Rightarrow \textbf{let}\ d_2 = 2 \\ \qquad \textbf{in let}\ r_2 = g(d_2) \\ \qquad\quad \textbf{in if}\ (2{*}d_2 \le r_2)\ \{\ r_2\ \} \\ \qquad\qquad \textbf{else}\ \{\ \textbf{die}(\texttt{prog})\ \}\end{array}}\,[\text{Call}]
}{
\begin{array}{l}\Gamma \vdash \textbf{Int} \Rightarrow \textbf{let}\ g = f(\lambda d_1.\ \textbf{let}\ r_1 = 2 \\ \qquad\qquad \textbf{in if}\ (d_1 \le r_1)\ \{\ r_1\ \} \\ \qquad\qquad\quad \textbf{else}\ \{\ \textbf{die}(\textbf{gen})\ \}) \\ \quad \textbf{in let}\ d_2 = 2 \\ \qquad \textbf{in let}\ r_2 = g(d_2) \\ \qquad\quad \textbf{in if}\ (2{*}d_2 \le r_2)\ \{\ r_2\ \} \\ \qquad\qquad \textbf{else}\ \{\ \textbf{die}(\texttt{prog})\ \}\end{array}
}\,[\text{Call}]
$$

Shorthands

$\textbf{Increasing} = d_1{:}\textbf{Int} \longrightarrow r_1{:}\textbf{Int}, d_1 \le r_1$

$\textbf{Doubling} = d_2{:}\textbf{Int} \longrightarrow r_2{:}\textbf{Int}, 2{*}d_2 \le r_2$

$\Gamma = \{f{:}\textbf{Increasing} \longrightarrow \textbf{Doubling}\}$

Figure 6: Example derivation

condition. In that case, the generated expression is bad, and the test case will abort while blaming the generator to indicate a test-generation failure. If the pre-condition fails, however, then the program being tested supplied a bad input to the function, and so the test case also aborts—but this time blaming the program to indicate a bug.

4.4 An example

To get a better feel for how the rules work, consider the example derivation in figure 6. The derivation uses a number of shorthands to avoid clutter. Specifically, if any of the pre- or post-conditions are just **true**, then they are omitted; similarly any **if** expressions that would have been generated to test such pre- or post-conditions are also omitted. In addition, the three shorthands in the box are used throughout.

The derivation starts by trying to generate an **Int** using an environment that contains a single function f. The function f accepts a monotonic function, specified via the type **Increasing**. It returns another function that promises to return something larger than *twice* its input, as specified via the type **Doubling**.

The first step of the derivation uses the [Call] rule, picking f from the environment. The left-hand derivation sub-tree generates the argument to pass to f, and the right-hand subtree uses the larger environment to generate the original goal, an expression of type **Int**.

First, consider the generation of f's argument. It must generate something of the type **Increasing**, since that is the type of f's argument. To do so, this derivation uses the [Fn] rule, which extends the environment with something of the input type and generates a function body. In this case, the derivation uses the [Base] rule to generate the result of the function, namely the number 2. Once the body of the function is generated, it can be packaged into the λ-expression by the [Fn] rule, along with a check of **Increasing**'s post-condition.

Having completed the derivation of the the argument to f, turn to the other half of the main derivation, which is the second occurrence of the [Call] rule. This time, the derivation uses the [Call] rule to generate a call to g, the variable it chose for the result of the call to f. Thus, g has the type **Doubling**. This time, the premises of the [Call] are fulfilled with a use of the rule [Base] and a use of the rule [Var]. The first builds the argument to g, and the second builds the overall result of the program, using the result of the call to g.

Notice that there are two non-trivial contracts embedded in in the original type for f. One appears as the post-condition to f's input, and the other appears as the post-condition to f's result. In the final program generated by the derivation, there are two uses of **die**, one corresponding to each contract. The first use of **die** comes from the post-condition of f's input; since it appears an odd number of times to the left of the arrow, the blame lies with the test case, so we see `gen`. The second use of **die** comes from the post-condition of f's result; since it appears an even number of times to the left of the arrow (i.e., zero times), the blame lies with the function, so we see `prog`.

Finally, if f were the function λg.g(2)*10, then the test case would abort with the blame assigned to the generator. If, on the other hand, f were λg.λx.g(g(x)), then the test case would abort with the blame assigned to the program, detecting a bug.

4.5 A theorem

Using a formal system to define our generation strategy lets us establish some properties. For example, a natural question about this system is whether all generated terms indeed have the types they should. Formally, we can express the answer as a theorem.

Theorem. If $\vdash \Gamma$ and $\Gamma \vdash \tau \Rightarrow e$, then $\Gamma \vdash e : \tau$

The proof of this theorem is a straightforward induction on the structure of the derivation of $\Gamma \vdash \tau \Rightarrow e$. If the [Var] rule is the last rule in the derivation, then we can use the type checking rule for variables, since the premises match. If the [Call] rule is used, then, by induction, we have derivations of $\Gamma \vdash e' : \tau_1$ and $\Gamma + \{x_2{:}\tau_2\} \vdash e_3 : \tau_3$. Also, since $\vdash \Gamma$, we know that the expressions e_1 and e_2 both have type **Bool** . Given that, and the rules for typing expressions, we can construct a derivation that the right-hand side of the [Gen] has type τ_3. In the base type cases, we know that the generated term type checks, and the [Fn] rule follows a similar argument to the [Call] rule.

4.6 The value of a formal model

Most of the testing literature uses algorithmic pseudocode (or sometimes just prose) to describe its techniques. We instead follow the established conventions of the type systems and operational semantics literature and use a formal system to describe the tests our technique can create.

We believe this shift benefits both readers and authors. A formal model provides readers a more precise and, in our opinion, a clearer account of our technique than we could give with pseudocode, which easily hides subtle but important details. Precision and clarity benefit authors too. Constructing a formal model helped us better understand our own technique—even after implementing a working prototype. For example, consider the [Call] rule in Figure 5. The final premise generates e_3 using an environment that does not include x_1, though that result could fruitfully contribute to e_3. We devised the rule in the figure by examining the prototype's implementation, but only after seeing the binding structure in the rule's precise notation did it become obvious that the generated programs could make better use of intermediate results.

5. From λ to a real language

Our prototype tool converts the rules of section 4 into a test-generation algorithm. It also scales the test-generation rules to handle many features of Racket, including contracts (Findler and Felleisen 2002), classes (Flatt et al. 2006), richer function-calling conventions (Flatt and Barzilay 2009), and the Scheme numeric tower. Finally, it addresses the practical problems of avoiding bad test cases and dealing with non-termination.

5.1 A full-fledged programming language

Generalizing from the simple programming language in section 4 to Racket requires a number of extensions to the rules. The primary extension adds support for Racket's object system. Like Java, Racket has a class-based object system; unlike Java, classes in Racket are first-class values, and the object system supports a wide range of method combinations. Like Java, subclasses may override methods; unlike Java, subclasses may augment methods, a la gbeta (Ernst 1999; Flatt et al. 2006).

To support Racket's object system, we extend the type system to allow class types and object types. A new rule, similar to the [Fn] rule, allows the system to generate a subclass of an existing class. Another new rule, similar to the [Call] rule, generates a method invocation.

In addition, we generalize the [Call] rule to support Racket's more sophisticated function-call conventions: multi-arity functions that support optional arguments and keyword arguments, as well as a `case-lambda` form that allows dispatching on the number of arguments at a call site. Support for these function forms mostly requires beefing up the [Call] rule (and the rule for method invocation).

Finally, we add many new rules like the ones for base types, to handle generation of expressions satisfying the contracts that commonly appear in the Racket codebase.

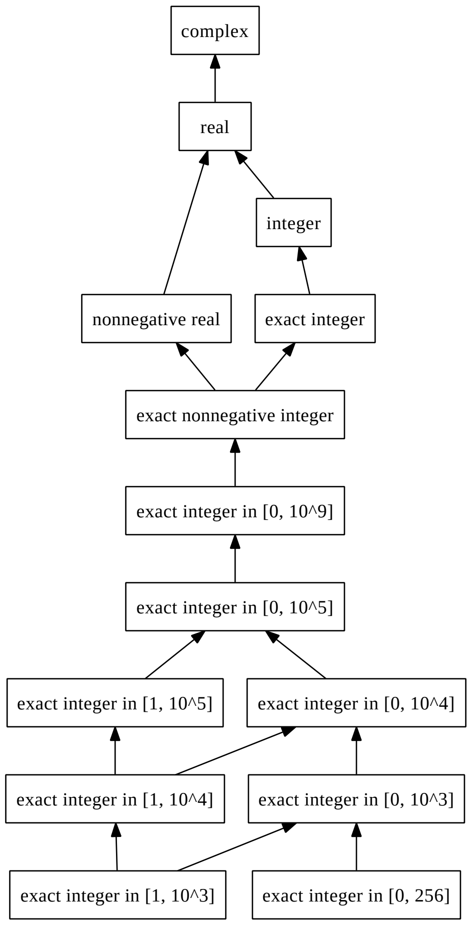
complex
real
integer
nonnegative real
exact integer
exact nonnegative integer
exact integer in [0, 10^9]
exact integer in [0, 10^5]
exact integer in [1, 10^5]
exact integer in [0, 10^4]
exact integer in [1, 10^4]
exact integer in [0, 10^3]
exact integer in [1, 10^3]
exact integer in [0, 256]

Figure 7: Subyping

5.2 Subtyping

Racket inherits Scheme's sophisticated number system, including complex numbers, exact rationals, and IEEE floating point numbers. The library that we tested required only some of the many different number types in Racket; figure 7 shows the precise ones that we used and their subtyping relationships. The library we tested also used many other more conventional subtyping relationships, such as the contravariant relationship for function types, the class inheritance hierarchy, and the normal relationships between pairs, lists, and other compound types.

Adding subtyping to the system amounts to changing the [Var] rule so that it allows picking a type out of the environment that is a subtype of the desired type.

5.3 An algorithmic version of the rules

The rules in section 4 admit a wide range of possible programs, and they allow multiple derivations for most combinations of a particular type and an environment. To build an algorithm for generating a program, we use three techniques: randomness to choose a rule when multiple rules (or multiple instances of a given rule) apply, a depth bound to limit the size of derivations, and continuation-passing style to support backtracking when some part of a generation fails.

Our top-level generator function has the signature

```
generate : type env depth
           (expression -> X)
           (-> X)
        -> X
```

The first two arguments match the rules. The third argument specifies a depth bound; when it reaches zero, the generator fails. The final two arguments are success and failure continuations. When the generator succeeds, it invokes the success continuation with the expression that it built. When the generator fails, it invokes the failure continuation. Using continuations in this manner make it easy to implement backtracking.[2] For example, the generator may attempt to use the [Call] rule and perhaps even succeed to build one of the arguments to the function or method, but then fail to build a later argument. In that case, the failure continuation will abort the entire attempt to use [Call] rule and try something else.

Our algorithm gives priority to the [Call] rule, specialized to imperative methods and functions. It will, with 80% chance, pick an imperative-looking method or function from the environment and use the [Call] rule to invoke that method or function. We use a simple test to determine if a method or function is imperative: its name ends with `!` or begins with `set-` (these being standard conventions in the Racket codebase) or it produces a result of type `void`. For the remaining 20% of the time, the generator chooses equally among the other three rules or a use of the [Call] rule that invokes a function or method that can produce a subtype of the goal type. (This second sort of use of the [Call] rule is what would have triggered the derivation shown in section 4.4.)

The depth bound is important for controlling the size of generated expressions. For example, suppose that the goal type is integer and that the environment contains an imperative function on integers. Choosing to call this function doubles the size of the generation problem—we now need to construct two integers: one as an argument to the imperative function, and one to satisfy the initial goal. Choosing to double the problem size 80% of the time can easily produce an expression whose representation consumes hundreds of megabytes of memory. To avoid such a distribution, we decrement the depth bound each time we apply a rule that can increase the number of goals; if the bound reaches zero, we avoid such rules.

[2] In our experience, backtracking does not substantially increase the generator's running time. For example, in generating a batch of programs with average length 120 LOC, our algorithm invokes 150 failure continuations per program on average.

5.4 Trying harder to avoid failure

The rules contain two expression templates that signal errors assigning blame to the generated program. In both cases, an **if** expression guards the failure. In our prototype, we use that conditional to make a few additional attempts to try to satisfy the pre-condition.

The failure that blames the generator in the [Fn] case matches the generator's failure when our prototype tries to construct the result of a method. Our generator often constructs methods to override other methods. In that case, the generated expression can avoid signaling blame directly, and instead try to make a call to a `super` method. Of course, the `super` call may also fail and blame the generator if the pre-condition of the method fails, but that kind of failure is much less likely.

A failure in the [Call] rule is also easily avoided in practice. Specifically, the generator creates a backup expression that it puts in place of the expression that blames the generated program. When the depth is too low to generate a backup expression, the generator aborts the entire call attempt.

5.5 Non-terminating programs

When overriding a methods that involves callbacks, the test generator can easily produce an infinite loop. For example, with the bounded stack example from section 3, overriding the `after_push` to invoke the stack's `push` method would produce a program that loops forever. To avoid infinite loops, the generator adds a counter to every generated method and function; each time the method or function is called, the counter decrements. If the counter reaches zero, the program aborts with a message indicating why, and we discard the test case.

6. Evaluation

To evaluate our random-testing technique, we applied it to the implementation of DrRacket's editor class. The editor implementation is more flexible and open than a typical text editor. As with a conventional editor, it supports linear sequences of text, but it abstracts over the objects that can appear in an editor, insisting only that they be derived from the `snip%` class. Beyond simple string snips, images and even nested editors can appear inside an editor. (DrRacket uses nested editors to allow, for example, embedded XML code inside a Racket program).

The editor classes are also extensible in the way that the snips are positioned. The `text%` class implements an editor that arranges the snips in lines and inserts newlines as needed to make the content fit. The `pasteboard%` class allows snips to float in a free-form manner (which DrRacket uses

Bug category	Requires overriding	Number of bugs
documentation bugs	no	16
first-order bugs	no	13
system dependence on default implementation	yes	1
missing state flag-based pre-condition	yes	lots (>20)
methods that should have been final	yes	7
state flags wrong during object initialization	yes	1
		Total: lots + 38

Figure 8: Bug summary

to display graphs and to implement a GUI-layout editor). All of this extensibility makes the editor classes a perfect opportunity to test our randomized-testing tool.

The results of our evaluation are summarized in figure 8. Each row corresponds to a certain category of bugs, showing a brief description of the bugs, whether the bugs require overriding, and the number of bugs found in that category. The categories are ordered by increasing complexity.

The first row contains bugs that we are embarrassed to report. Specifically, the Racket contract system does not yet support the language's higher-order class system. Consequently, the contracts on editor classes were written only in the documentation; separate, hand-rolled checks are in the implementation. To run our prototype, we wrote a parser that builds contracts from the documentation source. If the resulting contracts were added in a checkable way instead of just to documentation, we believe that all of the errors in these contracts would have been found by simply starting up DrRacket.[3]

The second row in the table represents bugs that could have been found by many existing random-testing tools. Here is an example of a (simplified) test case for one of those bugs:

```
(define sl (new style-list%))
(define bs (send sl basic-style))
(define js
  (send sl find-or-create-join-style
        bs bs))
(send sl new-named-style
      "The Name"
      js)
```

Finding this bug requires only that the testing tool be able to call methods and put their results together.

[3] Happily, we expect this implementation flaw to be remedied soon (Strickland and Felleisen 2010).

The remaining four lines correspond to bugs that, to the best of our knowledge, can be found automatically only by our tool.

The first of these lines represents a bug where a derived editor class, `scheme:text%`, overrides the `get-keymap` method from the `text%` class in such a way that it never returns `#f` (even though its contract allows that). Other parts of the same subclass assume that `get-keymap` never returns `#f`, but the subclass does not prohibit future extensions to the `get-keymap` method. To expose the bug, the test generator created an example like the following, making `get-keymap` return `#f`; simply creating an instance of the derived class triggers the error.

```
(define t%
  (class scheme:text%
    (define/override (get-keymap)
      #f)
    (super-new)))

(new t%)
```

The final three lines in figure 8 concern bugs that incorrectly cope with the locking mechanism in the `text%` class. A text editor uses four different levels of locking: *unlocked*, where any method can be called; *flow locked*, where methods that can change the locations of soft line breaks are forbidden; *write locked*, where methods that change the content of the editor are forbidden; and *read locked*, where nearly all of the methods are forbidden. Each of these modes has a matching boolean-valued method (that can always be called) that exposes whether the editor is in the corresponding mode.

The most common bug in these categories concern inaccurate method pre-conditions. For example, this program triggers a contract violation of the `delete` method's pre-condition:

```
(define t%
  (class text%
    (define/augment (on-insert x y)
      (send this clear))
    (super-new)))

(send (new t%) insert "y")
```

The `delete`'s method can only be called in the unlocked state, but during the dynamic extent of `clear`, the editor is write locked. The bug is that the `clear` method's pre-condition should say that it can only be called in the unlocked state.

We found 7 bugs where methods that were public should have been final. For example, the `in-edit-sequence?` method queries some internal state of an object. If it is overridden to return a bogus version of the state, then this program will fail.

```
(define t%
  (class text%
    (define/override (in-edit-sequence?)
      #f)
    (super-new)))

(send (new t%) undo)
```
This program fails because the contract for the `undo` method calls `end-edit-sequence`, whose contract is phrased in terms of the overridden method, `in-edit-sequence?`. Since that method is now returning bogus values, the contract fails when it should not.

Five of the other bugs in this category are similar to this one, where the overridden method fails to accurately report some internal state of the object. The remaining bug, however, required the `paragraph-start-position` method to be overridden. That method's job is to return the position in the `text%` object where a paragraph begins, i.e., a position just after a newline in the editor. When it fails to do so, it causes the line-breaking algorithm to fail internally with an array-indexing error.

Finally, the remaining bug was that the `text%` state variables were not initialized properly during the dynamic extent of the call to the constructor. The object should have been completely locked during this time and, once the constructor finished, it should have been unlocked. Discovering this bug requires the test case generator to override a method that is called during the dynamic extent of the constructor, like the `default-style-name` method is in this case:
```
(define t%
  (class text%
    (define/override (default-style-name)
      (send this line-paragraph 1)
      "Standard")
    (super-new)))

(new t%)
```
The other editor class, `pasteboard%`, does not suffer from this bug, because it does not invoke any overridable methods during its initialization.

7. Related Work

The test generation literature is vast. To narrow the discussion, we restrict our attention to techniques that test imperative object-oriented programs using a sequence of method and constructor calls. This distinction excludes tools such as DART (Godefroid et al. 2005), CUTE (Sen and Agha 2006), and EXE (Cadar et al. 2006), which test a program by generating its top-level input, since our focus is unit-level tests of stateful, reusable components.

Even within this restricted class, testing techniques vary widely. Rostra (Xie et al. 2004) and JPF (Visser et al. 2006) perform bounded exhaustive testing, using state matching to avoid generating redundant sequences. Symstra (Xie et al. 2004), and PEX (Tillmann and Halleux 2008) choose sequences based on symbolic execution and constraint solving. A recent refinement of Pex (Barnett et al. 2009) uses first-order contracts to guide symbolic execution and detect bugs. Crash 'n' Check (Csallner and Smaragdakis 2005) and its successor DSD-Crasher (Csallner and Smaragdakis 2006) use a constraint solver to extract concrete tests from warnings produced by ESC/Java (Flanagan et al. 2002). eToc (Tonella 2004) evolves test sequences using a genetic algorithm based on coverage. JCrasher (Csallner and Smaragdakis 2004) and Randoop (Pacheco et al. 2007) construct sequences at random. Others, such as Evacon (Inkumsah and Xie 2009) and MSeqGen (Thummalapenta et al. 2009), combine several techniques.

Our technique builds on the generation strategies employed in JCrasher and Randoop. The generation rules in figure 5, which allow tests to construct arguments by combining values in the environment, can be seen as an extension of the parameter-graph JCrasher uses to select method arguments. Following Randoop, our generation algorithm offers two improvements over JCrasher. First, our rules give names to (some) intermediate results, allowing generation of sequences like the following:
```
BoundedStack s = new BoundedStack();
s.push_n(s.capacity());
```
As Pacheco et al. note, JCrasher's algorithm cannot produce a `BoundedStack` in this configuration because the objects it constructs as method arguments flow only to the corresponding call sites, without contributing to any other expression. Second, JCrasher's algorithm never chooses to call a method returning `void`, in order to reduce the size of its search space; our algorithm, on the other hand, deliberately generates calls to methods with side-effects (identified heuristically according to their name and result type). This is crucial for generating the methods of a class whose objects are used as callbacks.

Unlike our algorithm, which promises to generate an expression of a requested type, Randoop generates sequences that result in a value of a random type. Its algorithm generates a sequence by iteratively appending a call to a method chosen at random, without regard for its result type. For many classes, this strategy works well, due to Randoop's feedback-directed heuristics, but because it does not seek a result of any particular type, its sequences cannot be used as method bodies, as those require a result of a specified type.

To the best of our knowledge, none of the aforementioned tools generate tests for open classes such as `BoundedStack`, where a method takes an instance of an abstract class of which implementations do not exist or represent only a small portion of possible use cases. JSConTest (Heidegger and Thiemann 2010), a contract-driven random test generator for JavaScript, offers some support for testing open classes in the form of higher-order functions, but the functions it constructs as test inputs do not have side-effects and do not make use of their arguments (except to check their contracts). Kiasan/KUnit (Deng et al. 2007), another contract-driven tool supporting open classes, constructs more sophisticated inputs using symbolic execution and constraint solving, but the mock objects it constructs as inputs have no side-effects except the ones prescribed by their contracts. This restriction hides the stack observer bug, where a loose contract is precisely the problem.

QuickCheck (Claessen and Hughes 2000), a random testing library for Haskell, provides flexible support for testing higher-order functions. Given user-defined specifications of a program's types, QuickCheck automatically derives generators for functions that consume and produce those types. Although targeted at pure functional programs, QuickCheck works with imperative programs too (Claessen and Hughes 2002). The approach Claessen and Hughes describe is likely to reveal the stack observer bug, but finding the off-by-one error is more difficult. QuickCheck is less automated than the above tools, and its effectiveness depends substantially on user configuration. For example, QuickCheck is unlikely to find the off-by-one error unless the user has the insight that the `capacity` method produces interesting arguments to the stack's other operations.

8. Conclusion

Testing open object-oriented systems is difficult because unknown classes may drive them in surprising ways. We describe a new algorithm for testing such systems which works by randomly constructing new subclasses based on the contracts specified for their interface. Our experience shows that our technique works well, finding dozens of bugs in DrRacket, and is easy to implement.

Still, our experience suggests some improvements. First, the test cases produced by our prototype are often unnecessarily large. The algorithm described in section 5.3 takes as input a bound on the depth of program derivations to consider, and the test programs it produces tend to grow exponentially with this bound. Often, a derivation of depth 5 is necessary to uncover a bug, but typically only one particular branch of the derivation actually requires this depth. We speculate that using a different strategy for limiting the size of the generated terms (e.g., randomly divving up a bound between subderivations to limit the tree's size instead of its depth) would enable us to find the same bugs, but without generating such large programs. Second, as discussed in section 4.6, our prototype would likely be more effective if the programs it produced reused more intermediate results.

Bibliography

Mike Barnett, Manuel Fahndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the Synergy between Automated-Test-Generation and Programming-by-Contract. In *Proc. Intl. Conf. Soft. Eng.: Companion Volume*, pp. 401–402, 2009.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Gill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. ACM Conf. Computer and Communications Security*, pp. 322–335, 2006.

Koen Claessen and John Hughes. Testing Monadic Code with QuickCheck. In *Proc. ACM SIGPLAN Haskell Wksp.*, pp. 47–59, 2002.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 268–279, 2000.

Cristoph Csallner and Yannis Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Software – Practice & Experience* 34(11), pp. 1025–1050, 2004.

Cristoph Csallner and Yannis Smaragdakis. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *Proc. Intl. Symp. Soft. Testing and Analysis*, pp. 245–254, 2006.

Cristoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In *Proc. Intl. Conf. Soft. Eng.* , pp. 422–431, 2005.

Xianghua Deng, Robby, and John Hatcliff. Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-Oriented Systems. In *Proc. Testing: Academia & Industry Conference – Practice & Research Techniques*, pp. 3–12, 2007.

Erik Ernst. gbeta — a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD dissertation, Department of Computer Science, University of Aarhus, Denmark, 1999.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *J. Functional Programming* 2(12), pp. 159–182, 2002.

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 48–59, 2002.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 234–245, 2002.

Matthew Flatt and Eli Barzilay. Keyword and Optional Arguments in PLT Scheme. In *Proc. Scheme and Functional Programming*, pp. 94–102, 2009.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with Classes, Mixins, and Traits (invited tutorial). In *Proc. Asian Symp. Programming Languages and Systems*, pp. 270–289, 2006.

Matthew Flatt and PLT. Reference: Racket. June 7, 2010. `http://www.racket-lang.org/tr1/`

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley Publishing Company, 1994.

Patrice Godefroid, Nils Karlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 213–223, 2005.

Phillip Heidegger and Peter Thiemann. JSConTest - Contract-Driven Testing of JavaScript Code. In *Proc. Intl. Conf. Objects, Models, Components, Patterns*, pp. 154–172, 2010.

Kobi Inkumsah and Tao Xie. Improving Structural Testing of Object-Oriented Programs via Integrated Evolutionary Testing and Symbolic Execution. In *Proc. IEEE/ACM Intl. Conf. Automated Soft. Eng.* , pp. 297–306, 2009.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proc. Intl. Conf. Soft. Eng.* , pp. 75–84, 2007.

B.S. Rubin, A. R. Christ, and K. A. Bohrer. Java and the IBM San Franscisco project. *IBM Systems Journal* 37(3), pp. 365–371, 1998.

Koushik Sen and Gul Agha. CUTE and jCUTE: Unit Testing and Explicit Path Model-Checking Tools. In *Proc. Intl. Conf. Computer Aided Verification*, pp. 419–423, 2006.

T. Stephen Strickland and Matthias Felleisen. Contracts for First-Class Classes. In *Proc. Dynamic Languages Symposium*, pp. 97–112, 2010.

Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In *Proc. Joint Euro. Soft. Eng. Conf. and ACM Symp. Foundations of Soft. Eng.* , pp. 193–202, 2009.

Nikolai Tillmann and Jonathan de Halleux. Pex – White Box Test Generation for .NET. In *Proc. Intl. Conf. Tests and Proofs*, pp. 134–153, 2008.

Paolo Tonella. Evolutionary Testing of Classes. In *Proc. Intl. Symp. Soft. Testing and Analysis*, pp. 119–128, 2004.

Willem Visser, Corina S. Pasareanu, and Radek Pelanek. Test Input Generation for Java Containers using State Matching. In *Proc. Intl. Symp. Soft. Testing and Analysis*, pp. 37–48, 2006.

Tao Xie, Darko Marinov, and David Notkin. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *Proc. IEEE Intl. Conf. Automated Soft. Eng.* , pp. 196–205, 2004.

Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In *Proc. Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, pp. 365–381, 2004.