Find More Bugs with QuickCheck!

John Hughes
Chalmers University, Göteborg, Sweden
rjmh@chalmers.se

Ulf Norell
Quviq AB, Göteborg, Sweden
ulfn@chalmers.se

Nicholas Smallbone
Chalmers University, Göteborg, Sweden
nicsma@chalmers.se

Thomas Arts
Quviq AB, Göteborg, Sweden
thomas.arts@quviq.com

ABSTRACT

Random testing is increasingly popular and successful, but tends to spend most time rediscovering the "most probable bugs" again and again, reducing the value of long test runs on buggy software. We present a new automated method to adapt random test case generation so that already-discovered bugs are avoided, and further test effort can be devoted to searching for *new* bugs instead. We evaluate our method primarily against RANDOOP-style testing, in three different settings—our method avoids rediscovering bugs more successfully than RANDOOP and in some cases finds bugs that RANDOOP did not find at all.

Keywords

Random testing, avoiding bugs, bug slippage

1. INTRODUCTION

In recent years random testing has become increasingly popular and successful. For example, RANDOOP has been used very successfully to test object-oriented software, finding hundreds of errors in Java and .NET libraries in the very first experiment [9]. QuickCheck [7] has become the dominant testing tool in the Haskell community; the commercial version has been used to test implementations of the basic software used in vehicles against the AUTOSAR standard, finding hundreds of bugs and problems, many of them in the standard itself [2]. CSmith has been used to find hundreds of bugs in production quality C compilers, including `gcc` [11]. But random testing suffers an awkward problem: the same bugs tend to be found again and again and again.

The fundamental problem is that different bugs usually have widely different probabilities of appearing in generated test cases; running enough tests to find rarely occurring bugs will therefore find commonly occuring ones many, many times. The problem is exacerbated by test-case reduction (*e.g.* by delta-debugging [12]), which is often combined with random testing to produce small, understandable failing tests—reducing a test case that provokes a less likely bug may often result in a test case that provokes a more likely one instead. Chen *et al.* term this effect *bug slippage* [5].

AST'16, May 14-15 2016, Austin, TX, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4151-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896921.2896928

There are many approaches to mitigating this problem. RANDOOP generates sequences of method calls by *combining and extending* already-discovered sequences that do not fail, thus avoiding hitting exactly the same failing test case again and again, and then partitions failing tests into equivalence classes based on the last method call, reporting one error from each class. QuickCheck users develop *models* of the software-under-test, manually adding '*bug preconditions*' to avoid provoking already-discovered bugs, or even modelling buggy behaviour in a so-called 'variant' in order to test code that can only be reached after a buggy operation; this manual adaption of the model was a substantial part of the work involved in the AUTOSAR testing project referred to above. Chen *et al.* developed a ranking method for test cases reported by CSmith and other compiler fuzzers, based on metric spaces, with the goal of reporting diverse tests that reveal different bugs early in the list [5].

All these approaches suffer disadvantages, however. RANDOOP's feedback mechanism avoids executing exactly the same failed test case repeatedly, but will still find many variations on it. So, much testing time may still be wasted provoking the same bug repeatedly. Chen *et al.*'s ranking is a postprocessing step, and so does not avoid wasted time spent rediscovering already known bugs, or prevent test case minimisation from causing bug slippage. QuickCheck's approach *does* avoid spending time on already known bugs, and also avoids bug slippage, because QuickCheck's 'shrinking' (test case minimization) respects preconditions while reducing test cases—so will not reduce a test case to one already excluded by a bug precondition. On the other hand, it requires manual effort to diagnose bugs and formulate suitable bug preconditions or bug models. Our goal in this paper is to automate this process, so that, as bugs are discovered, we can automatically focus test effort on areas not yet known to be buggy, resulting in a set of minimized test cases that should (ideally) all represent different bugs. We have created an extension to QuickCheck that does just this.

Our approach is to generalize each bug as it is found, to a 'bug pattern', and then adapt test case generation so that test cases matching an existing bug pattern are never generated again, neither randomly nor during shrinking. We explain our technique (which we call 'MoreBugs') in section 2, with reference to a simple example taken from the Erlang run-time environment. Our evaluation (in section 3) shows that MoreBugs is able to find bugs that QuickCheck and RANDOOP cannot find. In section 4 we discuss limitations of our approach; sections 5 and 6 discuss other related work and conclude.

Our main contributions are:

- A fully automatic method to avoid provoking already-known bugs *at test case generation time*, and to avoid bug slippage when failed tests are minimized.
- Experimental results showing that this method can, in some cases, reduce the number of tests needed to find a set of bugs quite dramatically, and even find new bugs in well-studied software.

2. THE "MORE BUGS" METHOD

We take as our motivating example the Erlang *process registry*, essentially a local name server. Processes in Erlang have dynamically allocated identifiers ("pids"), which can be stored in the registry along with a readable name; from then on, other processes can refer to them by name instead of by pid. The process registry provides the following API:

```
register(Name, Pid) → ok
unregister(Name) → ok
whereis(Name) → Pid | undefined
```

The operations are as follows: `register` associates a name with a process ID, `unregister` forgets that name, and `whereis` looks up a name to get a process ID.

We can create a simple QuickCheck model of the process registry, which just tests random sequences of calls to `register`[1], `unregister`, `whereis`, and `spawn` (which creates a new process). With no stated postconditions, the model just tests the property that no exceptions are raised. Generating tests from the model quickly finds a counterexample:

```
unregister(a)
```

In fact, `unregister` raises an exception if the name it is passed is not already registered—our model is wrong. Now, before we do anything else, we might like to know if there are any *other* problems with the model, but QuickCheck reports the *same* counterexample 19 times out of 20. One time out of twenty we find a different counterexample:

```
Pid = spawn()
register(a, Pid)
register(a, Pid)
```

That is, registering a name that is already registered also raises an exception. This bug is very simple, yet is reported very rarely.

Why is this? It is a combination of two factors:

1. Any test case that contains an `unregister` of a not-registered name will fail. As we generate longer and longer test cases, the probability that our test case contains such a call approaches 1.
2. QuickCheck's *shrinking* minimizes a failing test case by (among other things) removing as many function calls as possible. If the counterexample *contains* a call to `unregister`, it is very likely that QuickCheck will shrink it to *just* a call to `unregister`.

[1]We choose names randomly from a small set, so that collisions occur frequently.

How can we avoid finding the same bug over and over again? Our idea is to take a failing test case and automatically *generalize* it to a whole class of suspicious test cases, which we call a *bug*. We continue to test the system, but ignoring any test cases matching that bug.

More precisely, our algorithm maintains a set of bugs, which is initially empty. We test the system, ignoring any test cases matching any of the bugs.[2] If we find a failing test case, we generalize it, add the resulting bug to the bug set, and repeat. Eventually, the bugs will cover all possible failing test cases, and we will not be able to provoke a failure.

Note that we *overgeneralize* failing test cases. This is because our goal is not necessarily to find *all* bugs in the software under test—we want to find *more* bugs than using random testing, but not so many that the user is overwhelmed.

The remaining problem is: *how do we generalize failing test cases into bugs*? In the remainder of this section we will describe our approach. It is fully automatic, though the user can tune it for better results. It is syntactic, so that we can avoid existing bugs *during test case generation*, which is vital if we want to make effective use of the time available for testing. We have implemented it in QuickCheck, although it should apply equally well to RANDOOP or similar tools.

2.1 Take one: subsequence checking

A very simple approach is as follows: take the failing test case, look at the *names* of the functions it calls and ignore their arguments. This gives us a sequence of function names S, which we take as the bug.

Given a test case, we can compute the sequence T of its function names too; the bug then matches the test case if S is a subsequence of T.

Testing the process registry, we generalize the `unregister` bug to any test case that *contains* a call to `unregister`. Thus, when we re-run QuickCheck, it does not generate any calls to `unregister`, and we immediately find the second counterexample, which was so hard to find earlier:

```
Pid = spawn()
register(a, Pid)
register(a, Pid)
```

We generalize this counterexample to the bug `spawn`, `register`, `register`, and thereafter do not generate any test case that contains a call to `spawn`, followed by two calls to `register`. Since we have also ruled out calling `unregister`, the only test cases left are ones where we `spawn` some processes, and then call `register`—once—followed by `spawn`ing some more processes. This is not likely to provoke any new failures, and indeed it does not. This method finds two bugs; we will now improve it so that we can find more.

2.2 Take two: subsequence matching

Notice that the counterexample in the previous section does not apply to just any old sequence `spawn`, `register`, `register`: we must spawn a process and then register *that process* (`a`) twice. We would like to capture in our bugs the requirement that certain function arguments be identical.

Instead of just extracting the list of function names from the test case, we *abstract* our test cases, replacing concrete

[2]To get a good distribution of test data, we in fact build up the test cases one command at a time, and never choose a command that would cause the test case we have built so far to match a bug.

values by free variables which can stand for anything. We give these abstracted variables names beginning with a question mark, to distinguish them from ordinary Erlang variables.

We allow the user to specify how to abstract test cases, but by default, we replace all function arguments by distinct variables—*except* that if the same value appears more than once in the test case, then we replace all occurrences with the same variable. For the example above, we get

```
?Pid = spawn()
register(?A, ?Pid)
register(?A, ?Pid)
```

(Note that `?A` is a variable, meaning that we do not care which name we choose for the process.)

Now we say that a bug matches a test case if there is a *ground instance* of that bug that is a subsequence of the test case. In this example, our bug matches any test case that spawns a process, and later registers that process twice, giving it the same name both times—perhaps with other function calls in between.

The strategy of the previous section amounts to an extremely weak form of abstraction where we replace all function arguments with a "don't-care" variable:

```
? = spawn()
register(?, ?)
register(?, ?)
```

By capturing equality constraints, our tool is able to more precisely characterize the `spawn, spawn, register` bug, and rules out fewer test cases having found it. Running the tool again, we find a third counterexample:

```
Pid1 = spawn()
Pid2 = spawn()
register(a, Pid1)
register(a, Pid2)
```

Unsurprisingly, we cannot give a process a name that is already taken. Unfortunately, our tool also finds two variations on this bug:

```
Pid1 = spawn()          Pid1 = spawn()
Pid2 = spawn()          register(a, Pid1)
register(a, Pid2)       Pid2 = spawn()
register(a, Pid1)       register(a, Pid2)
```

These three counterexamples differ only in the order of the calls—they lead to the same state, and provoke the same exception. This motivates finding *parallel bugs.*

2.3 Take three: parallel matching

The duplication we described in the last section is very common. For example, a system might have several initialisation functions, which the test case will have to execute before doing anything interesting, but it may not matter in which order they are called. We would like to capture in our generalisations the possibility that *order does not matter.*

We therefore augment our bugs with a *parallel composition* operator. A bug is a sequence, as before, but each element can be either a function call or a parallel composition `p|q`. Here, `p` must be a *single* function call, while `q` is a sequence, and may itself use parallel composition. There is no concurrency here: `p|q` just means that we run `p` at some point during `q`. A parallel bug matches a test case if some sequentialisation of the bug matches the test case.

We can express our latest family of counterexamples by the following parallel bug:

```
?Pid1 = spawn(),
register(?A, ?Pid1) |
  (?Pid2 = spawn(), register(?A, ?Pid2))
```

This bug expresses that we can execute `register(?A, ?Pid1)` at three possible points: before `?Pid2 = spawn()`, before `register(?A, ?Pid2)`, or after `register(?A, ?Pid2)`.

But how can generalize a *sequential* test case into a *parallel* bug? Our idea is to *detect parallelism by testing.* Suppose the failing test case consists of $\mathtt{n}$ statements $\mathtt{s}_1, \mathtt{s}_2, ..., \mathtt{s}_\mathtt{n}$. We take the first statement, $\mathtt{s}_1$, and try to move it later on in the test case. If the test case still fails, then evidently it did not matter exactly when we executed $\mathtt{s}_1$, and we introduce a parallel composition. More precisely, suppose the test case still fails if we move $\mathtt{s}_1$ after each $\mathtt{s}_\mathtt{i}$, i.e. for i from 2 to k $\mathtt{s}_2, ..., \mathtt{s}_\mathtt{i}, \mathtt{s}_1, \mathtt{s}_{\mathtt{i}+1}, ..., \mathtt{s}_\mathtt{n}$ still fails. Then we put $\mathtt{s}_1$ in parallel with statements $\mathtt{s}_2$ to $\mathtt{s}_\mathtt{k}$, giving us $\mathtt{s}_1 \mid (\mathtt{s}_2, ..., \mathtt{s}_\mathtt{k}), \mathtt{s}_{\mathtt{k}+1}, ..., \mathtt{s}_\mathtt{n}$. We then recursively perform this procedure on the two statement blocks that are left, $\mathtt{s}_2, ..., \mathtt{s}_\mathtt{k}$ and $\mathtt{s}_{\mathtt{k}+1}, ..., \mathtt{s}_\mathtt{n}$.

When performing the recursive parallelisation we must decide which sequentialisations to test for previously introduced parallel compositions; clearly testing all of them is not feasible. We settled for testing two sequentialisations: one where all operations on the left-hand side of a parallel composition are called before the right-hand side, and one where they are called after all of the operations in the right-hand side. This may lead us to overgeneralize some bugs, but the effect is dwarfed by the other generalizations we do.

2.4 Bug subsumption

In fact, our tool finds yet another counterexample:

```
Pid = spawn()
register(a, Pid)
register(b, Pid)
```

Is this not the same bug we discussed in section 2.1? No: the earlier bug registered a process twice with the same name, but here we register a process with *two different names*! This also raises an exception, hence the counterexample.

Our abstraction generalizes this counterexample into the following bug:

```
?Pid = spawn()
register(?X, Pid)
register(?Y, Pid)
```

Notice, though, that *this bug is a generalisation of the first* `spawn, register, register` *bug that we found in section 2.2!* Any instance of the first bug is an instance of this one. Hence we do not need to keep the first bug any more.

Therefore, whenever we find a new bug, we check whether it subsumes any existing bugs; if it does, we forget the old bug. How do we check if bug A subsumes bug B? If bug B is sequential, we simply check if A matches B, as if B were a normal test case. If B is parallel, we enumerate all of B's sequentialisations and check if A subsumes all of them; if B has more than 1000 sequentialisations, we give up and declare that A does not subsume B. 1000 is a bit arbitrary; we found it small enough that subsumption testing is fast, while large enough that it was almost never exceeded. We discuss briefly the consequences of this limitation in Section 4.

3. COMPARISON WITH RANDOOP

MoreBugs uses feedback from failed tests to guide the generation of future tests; it clearly employs a form of feedback-directed random testing. Feedback-directed random testing originated with RANDOOP [9], still the best known tool of this type, so we aimed to compare MoreBugs to it. However, RANDOOP tests Java classes fully automatically, using reflection to identify the API under test, while QuickCheck tests Erlang or C code, against a state machine model provided by the user that specifies both how tests should be generated, and the properties that ought to hold. A direct comparison is thus not straightforward.

We decided therefore to carry out *two* experiments. First, we extended QuickCheck to test Java classes without a specification, using reflection to identify the API as RANDOOP does—which allowed us to compare MoreBugs and RANDOOP on examples from the RANDOOP test suite. Second, we implemented a RANDOOP-style tool on top of QuickCheck, and compared it to MoreBugs on two examples: a version of the registry example, and some vehicle software in C previously tested with QuickCheck.

It may seem tempting to compare MoreBugs with Adaptive Random Testing [4] instead, but as we discuss in section 5, ART is not directly comparable because it requires a *distance metric* for test cases. While finding such a distance metric for QuickCheck-style test cases is a good research problem, it is beyond the scope of this evaluation section.

3.1 MoreBugs on the RANDOOP test suite

Our first experiment compares the *variety* of bugs found by MoreBugs and RANDOOP: in a codebase with both easy- and hard-to-find bugs, do the easy bugs mask the hard ones? We decided to test this using RANDOOP's test suite.

As RANDOOP is for Java, we built a QuickCheck model for testing Java code. Our model uses an Erlang-to-Java interface and reflection to generate random sequences of Java API calls. The model postcondition checks two properties: 1) there are no `NullPointerException`s, provided that the test case itself does not use `null`; and 2) `equals` is reflexive, symmetric, and compatible with `hashCode`. These are the same conditions that RANDOOP checks. Other exceptions are *not* considered to be bugs—they are more likely to indicate a random test case that misuses the API in some way.

We then ran QuickCheck with shrinking, both with and without MoreBugs, and RANDOOP on two examples from RANDOOP's test suite: the Apache Commons mathematics and collections libraries. We ran each tool until it had generated a large number of failing test cases, which we then classified. For the mathematics library the results were:

Kind of bug	QuickCheck	MoreBugs	RANDOOP[3]
Null pointer	98%	68%	98%
Equality	2%	32%	2%
Total failures	161	207	54

Many classes in the mathematics library can be created *without* being initialised, and initialised later—using the class between creation and initialisation provokes a `NullPointerException`. These failures are very easy to find, which explains the high number of `NullPointerExceptions` found by QuickCheck and RANDOOP.

There is one more interesting bug—the `Complex` class overloads `equals` but not `hashCode`, so that two equal complex numbers can have different `hashCode`s. QuickCheck struggles to find it, because it is much easier to get a `NullPointerException`, but MoreBugs hits it a third of the time.

[3]We ran RANDOOP until it stopped finding more failing test cases, and allowed it to shrink the resulting test cases.

On the collections library we get the following results:

Kind of bug	QuickCheck	MoreBugs	RANDOOP
Null pointer	99%	71%	92%
Equality	1%	29%	8%
Total failures	116	96	49

As before, QuickCheck only finds one test case involving `equals`. But in this case, there are actually *several* classes with a buggy implementation of `equals`—so QuickCheck missed some bugs! In fact, RANDOOP also mostly provokes `NullPointerException`s, and although it finds several bugs involving `equals`, it also misses some. Here is one MoreBugs finds, which is still present in the latest version of the library:

```
CompositeSet x = newCompositeSet();
Collection y = x.getCollections();
Collection z =
  TransformedCollection.decorate(y,
    NOPTransformer.getInstance());
```

The test case creates a collection `y`, and then transforms the collection by applying the identity function to each element to obtain `z`. After running this code, `z.equals(y)` is `true`, but `y.equals(z)` evaluates to `false`, a violation of symmetry. Neither QuickCheck nor RANDOOP find this bug.

Bug masking is a serious problem in these two examples, with many trivial null pointer bugs masking more interesting equality bugs. As we might expect QuickCheck does extremely poorly at uncovering more than one bug. RANDOOP does better but few of its counterexamples are equality bugs. MoreBugs finds the highest variety of bugs, and finds the bug above which neither of the other tools can find.

3.2 RANDOOP-style testing of QuickCheck properties

RANDOOP is designed to test object-oriented software, in which state is encapsulated in objects. It collects *non-failing sequences of method calls* which construct objects, then generates each new test by first choosing a method to test, then randomly choosing previously identified non-failing sequences that construct suitable arguments for the chosen method; the resulting test case is a concatenation of the sequences that construct the arguments, followed by a call of the method under test. Method arguments which are not of an object type are chosen randomly from a small set—for example, by default, integer arguments will be 1, 10, or 100. The test fails if there is a null-pointer exception or a contract violation in the last method call. Passing tests add to the non-failing sequences for generating future tests, if the resulting object has a different state from those objects that RANDOOP already knows how to construct. Object states are compared using the Java `equals` method, which by default compares representations, but can be overridden by the programmer to perform a more abstract comparison. RANDOOP may thus overlook errors which are provoked by an object O_2, if it first finds a way to construct O_1 such that O_1.`equals`(O_2).

QuickCheck state machines test systems whose state is usually opaque. For example, we cannot inspect the internal state of the process registry, although we *can* compute the *model state* at each point in a test case. The operations in a test case have preconditions that constrain the model state in which they can be invoked, which means that operation sequences cannot simply be concatenated. Operations have postconditions, checked against the model state; tests may fail either because of an exception, or because a postcondition fails to hold. Operation arguments are not usually objects, and generating random, but appropriate, arguments can be quite complex.

We decided to *reuse* existing QuickCheck state machines for RANDOOP-style testing (henceforth called RDS), inheriting the existing generators for operation arguments, and the existing pre- and post-conditions (corresponding to the contract checks made in the original RANDOOP experiments). After each successful test, we recorded the sequence of operations in the test case, and the model state reached at the end of the test—but we kept only *one* (the shortest) sequence reaching each model state. (We thus use model state comparison in the same way RANDOOP uses the `equals` method, as an abstract comparison of opaque representations). We generated test cases by choosing an operation to test, then choosing an already-reached model state satisfying the operation's precondition, and finally appending a newly generated call of the chosen operation to the recorded sequence that reached the chosen model state. We avoided generating the same test case twice, both for RDS and for QuickCheck generation. The RANDOOP paper [9] partitioned failed tests into equivalence classes using a tool called REDUCE (which considered test cases equivalent if the same method failed with the same exception), and reported one randomly chosen test from each equivalence class. We use the same approach, but report a *minimal length* test case from each class.

3.3 Evaluation: the Process Registry

We evaluated our test generation strategies on an extended version of the registry model discussed above. The extensions were: 1) a postcondition for `whereis(Name)` requiring it to return the correct registered pid, or `undefined` if `Name` is not registered; and, 2) a new operation `kill(Pid)` to kill a process, treated as a no-op in the model. In fact, dead pids cannot be registered, and killing a registered process removes it from the registry, so this extension introduces more inconsistencies between the model and the implementation (or 'bugs' for the purposes of this experiment).

Running MoreBugs on this example generates a list of five reduced bugs, shown in Figure 1. Bugs 1, 2 and 5 were discussed above; bugs 3 and 4 were introduced by killing processes. There is actually a *sixth* way of provoking an exception in this example:

```
V1 = registry_eqc:spawn(),
registry_eqc:register(d, V1)
registry_eqc:kill(V1),
registry_eqc:unregister(d)
```

Here the `unregister` fails, even though the name `d` was registered, because `V1` was removed from the registry by the `kill`. MoreBugs does not find this case because it is an instance of Bug 1 (it contains a call of `unregister`).

RDS testing reports three failed tests in this case, corresponding to bugs 1, 3 and 4 in Figure 1. Only three bugs are reported because REDUCE-style partitioning finds only three equivalence classes: there are only three functions that can fail, and with only one exception or contract failure in each case. Bugs 2 and 5 provoke the same exception in the

```
1. registry_eqc:unregister(a)

2. V1 = registry_eqc:spawn(),
   registry_eqc:register(a, V1)
   | registry_eqc:register(b, V1)

3. V1 = registry_eqc:spawn(),
   registry_eqc:kill(V1),
   registry_eqc:register(a, V1)

4. V1 = registry_eqc:spawn(),
   registry_eqc:register(d, V1)
   | registry_eqc:kill(V1),
   registry_eqc:whereis(d)

5. V1 = registry_eqc:spawn(),
   registry_eqc:register(a, V1)
   | V3 = registry_eqc:spawn(),
   registry_eqc:register(a, V3)
```

Figure 1: Registry bugs found by MoreBugs.

same function (`register`) as bug 3, so only one of these bugs can be reported. However, this is really just an artefact of the fact that `register` raises the *same* exception in the event of failure, no matter *why* the registration failed. We therefore replaced the REDUCE test case classification with a custom one which distinguished the three reasons why a call to `register` might fail. Using this refined classification, RDS testing reports the same bugs as MoreBugs.

The real question, though, is *how quickly* are the bugs found? We first measured *how often* each bug is provoked in (a) random test cases generated by QuickCheck, (b) *shrunk* test cases generated by QuickCheck, (c) RDS test cases. In each case, we ran long enough to generate 10,000 failing test cases. We report how often each bug was found *per thousand calls to the API under test* (since generated test cases vary in length between the three methods). The results were:

Bug	QC-S	QC+S	RDS
1	258.1	53.2	12.8
2	13.9	3.1	3.1
3	18.9	3.8	3.1
4	1.1	0.24	1.8
5	5.7	2.0	5.1
Total	297.7	62.7	25.9

In this table, QC-S denotes QuickCheck without shrinking, and QC+S denotes QuickCheck with shrinking. We see that bug 1 (`unregister`) is provoked *far* more often by QuickCheck than the other bugs, with shrinking (by a factor of $14 - 220\times$) or without ($13 - 480\times$). RDS finds them with a more even distribution (with the most common bug only found $2.5 - 7\times$ as frequently as the others). Shrinking is costly, accounting for 80% of the cost of QuickCheck testing when it is enabled—although this is probably because these bugs are so easy to provoke, so much more time is spent shrinking them to a minimal example than finding them in the first place. RDS performed many more calls per bug found, probably because RDS constructs longer and longer sequences that do *not* fail, to which a single possibly-failing call is appended—which means that as time goes by it is less and less likely that a call will provoke a failure. On the other hand, there is a bound on the length of test cases that QuickCheck will generate.

We know how often each single bug is provoked, but how quickly can we find *all 5* bugs? To measure this, we ran each tool until all five bugs had been found, and measured how many API calls were performed up to that point. Since this varies from run to run, we repeat each experiment as many times as possible in 30 minutes, and report both the number of repetitions and the mean number of API calls per experiment. The results were:

QC-S		QC+S		RDS		MoreBugs	
N	T	N	T	N	T	N	T
212	944	88	4893	573	781	604	713

N denotes the number of experiments run for each method, and T denotes the average number of API calls per experiment.

Our experiment shows that MoreBugs was able to find all five bugs a little faster than RDS, on average, presumably because it is more successful at avoiding retesting already discovered bugs. But it is also important to recall that QC+S and MoreBugs are finding *minimized* test cases in this time; QC-S and RDS are finding unminimized random test cases, which are unlikely to be very useful for debugging until they *have* been minimized, either manually or by a separate tool. Minimizing test cases is quite costly in itself—for this example, QuickCheck's shrinking is several times more costly than finding the bugs in the first place. So, that MoreBugs can find *minimal* test cases for all five bugs, in less time than RDS can find *random* test cases, is an excellent result. If finding minimal failing tests is the goal, then MoreBugs' speedup over QC+S is particularly striking.

3.4 Evaluation: An AUTOSAR CAN stack

For a larger example we chose an implementation of a network stack for a CAN bus for which we happen to have a QuickCheck model [2] of the AUTOSAR standard [3]. The model consists of 5,000 lines of Erlang code modelling 50 API functions, and the implementation is around 20,000 lines of C code. The model is configurable to support a wide range of valid behaviours, so in order to emulate a faulty implementation with a suitable number of bugs we simply configured the model so that it did not match the behaviour of the actual implementation. For the experiment we chose to enable 9 different "bugs" of varying complexity. We then measured for each method the average number of calls required to find each bug.

Bug	QC+S	RDS	MoreBugs
1	5k	100k	5k
2	1,000k	-	1,000k
3	2,000k	-	-
4	500k	-	2,000k
5	1,000k	-	-
6	2,000k	-	500k
7	-	-	500k
8	-	-	1,000k
9	1,000k	-	1,000k

In this example there is no measurable difference in the bug finding capabilities of QC+S and MoreBugs. This is disappointing, but perhaps not surprising given that even the easiest to find bug only fails in less than 1% of test cases. This means that QC+S does not spend a significant time generating and shrinking previously found bugs, and so there is not much need for MoreBugs in this example.

RANDOOP-style testing, however, performs very poorly, only finding the simplest bug. The likely explanation for this is that getting the CAN stack into a state where it is operational and interesting things can happen is a rather delicate procedure, involving around ten API calls. The generators used by QC+S and MoreBugs are aware of this and are biased to frequently generate the proper setup sequence. It is not immediately obvious how to best adapt RDS to deal with this problem. One possibility would be to sometimes pick a sequence of operations—such as the CAN setup sequence—instead of a single operation when extending a previous test case.

It is also the case that RDS runs much shorter test cases. MoreBugs and QC+S use an average of 40 operations per test case with several hundred operations in the longest tests, while running RDS for an hour yields no test case longer than 40 operations with an average of 13. Adding sequences would improve this slightly, but most likely not remove the problem.

4. LIMITATIONS

MoreBugs performs badly when a test fails *because an operation that should be invoked was not.* For example, in the process registry example, the test case `unregister(a)` fails because `a` has not been registered, after which MoreBugs will never generate any calls to `unregister`—and so will not find any more complex bugs involving `unregister`. We have no fix for this problem at present—except to correct the bugs and run MoreBugs again, when a different set of bugs will be found.

MoreBugs aims to generate a single instance for each bug, but there are a few cases where this may fail. One case is when subsumption checking fails due to too many possible sequentialisations of a parallel test case. In this case MoreBugs can report a subsumed bug. This is unfortunate, but not a big problem in practice. Failure to discard a subsumed test case will not affect subsequent testing, except by a small performance penalty having to avoid more bugs. Note that discarding a bug that is not in fact subsumed could lead to non-termination, so we need to err on the side of not discarding bugs.

We generalize argument values all at once. It would be interesting to consider an incremental generalization of values, backed by tests to see whether each generalization is valid—just as we do when parallelizing bugs.

MoreBugs only finds parallel compositions where the left hand side is a *single* command. Sometimes the most general form of a bug is a parallel composition with *multiple* commands on both sides, and in this case MoreBugs will report several bugs instead of one. We have not found this to be a big problem in practice.

Finally, MoreBugs only generates test cases consisting of sequences of function calls, and our bug generalizations depend heavily on this. But the same problem (repeated provocation of the same bug) may arise when test cases take other forms, such as a collection of input values, and with other test case generation methods, such as concolic testing [10]. It would be interesting to see whether similar ideas can be developed in other contexts.

5. RELATED WORK

Adaptive Random Testing (ART) [4] is a way of increasing diversity in random tests. Rather than running every random test generated, ART repeatedly generates a set of candidate tests, but only *runs* the one 'furthest' from any previous run test. This leads to a greater variety of tests, and has been shown to reduce the mean number of test to find the first bug by up to 50%. However, it requires a *distance metric* to be defined on test cases, which is not easy for non-numerical software, and the overheads of the method can make it slower than random testing in practice [1].

Ciupa *et al.* adapt ART to objected oriented software, by defining a distance metric on *object states*, and selecting objects with the *greatest average distance* to previously used objects for use in test cases [6]. Evaluation on the EiffelBase library showed a reduction in the number of tests needed to find the first fault by a factor of five, on average, compared to random testing. But this method requires access to object states; MoreBugs is a black-box testing method based entirely on recognising patterns in test cases.

Directed random [8] or *concolic* testing [10] combines random testing with constraint solving to force control down new *paths* in the software under test; the branch conditions along the path taken by a random test are collected, then one of them is negated to produce a constraint on the test inputs that will force one of the branches to made differently. These approaches have been very successful at improving code coverage and finding bugs, but they are white box methods that require instrumenting the code under test.

6. CONCLUSION

We have presented MoreBugs, an extension to QuickCheck that, once a bug is discovered, avoids generating more test cases that are likely to provoke the same bug. The goals are to reduce debugging time, by presenting each bug only once, and to improve testing effectiveness by concentrating test effort on areas not yet known to be buggy.

We compared MoreBugs against RANDOOP, which proved harder than expected, because QuickCheck and RANDOOP are really designed for different purposes, and test different kinds of software in different ways. We had to adapt both tools to make them applicable in the other's context, reimplementing the core RANDOOP algorithms to work with QuickCheck examples, and building a trivial QuickCheck model of a Java class library to apply QuickCheck and MoreBugs to RANDOOP examples. Nevertheless, despite these difficulties we were able to make comparisons on several large examples.

First, in a small experiment with several easily-provoked bugs, MoreBugs was able to find all the bugs, and report *minimized* test cases, more quickly than QuickCheck or "RANDOOP-style" testing could find them at all. In a much larger experiment using automotive software, with much harder-to-find bugs, then MoreBugs was much quicker to find some bugs, but not others. In this experiment, "RANDOOP-style" testing performed very poorly, perhaps because this is not its intended domain. In a final experiment using Java libraries from the RANDOOP test suite, and the real RANDOOP tool, MoreBugs was able to provoke subtler bugs much more often than either QuickCheck or RANDOOP, finding some bugs which RANDOOP missed altogether—and which are still present in those libraries today.

MoreBugs does have limitations, because of its syntactic generalization of counterexamples, but our experiments show that it can increase the effectiveness of random tests and does find more bugs at once than random testing.

Acknowledgment

We would like to thank Hans Svensson for his help in getting the AUTOSAR tests set up and running. This work was partly funded by EU FP7 project PROWESS, and by the Swedish Foundation for Strategic Research project RAWFP.

7. REFERENCES

[1] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proc. 2011 Int. Symp. on Softw. Testing and Analysis*, ISSTA '11, pages 265–275. ACM, 2011.

[2] T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *Softw. Testing, Verification and Validation Workshops (ICSTW), 8th Int. Conf.*, pages 1–4. IEEE, 2015.

[3] AUTOSAR consortium. AUTomotive Open System ARchitecture specifications. http://www.autosar.org.

[4] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: the ART of test case diversity. *J. Syst. Softw.*, 83(1):60–66, Jan. 2010.

[5] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13, pages 197–208. ACM, 2013.

[6] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proc. 30th Int. Conf. on Softw. Eng.*, ICSE '08, pages 71–80. ACM, 2008.

[7] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. 29th Int. Conf. on Softw. Eng.*, ICSE '07, pages 75–84. IEEE Computer Society, 2007.

[10] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. 10th European Softw. Eng. Conf. held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Softw. Eng.*, ESEC/FSE-13, pages 263–272. ACM, 2005.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. 32Nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '11, pages 283–294. ACM, 2011.

[12] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Softw. Eng., IEEE Transactions on*, 28(2):183–200, Feb 2002.