View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Explore Bristol Research

Gerdes, A., Hughes, J., Smallbone, N., Hanenberg, S., Ivarsson, S., & Wang, M. (2018). Understanding Formal Specifications through Good Examples. In *17th ACM Erlang Workshop 2018: Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang* (pp. 13-24). Association for Computing Machinery (ACM). https://doi.org/10.1145/3239332.3242763

Peer reviewed version

License (if available):
Other

Link to published version (if available):
10.1145/3239332.3242763

Link to publication record in Explore Bristol Research
PDF-document

This is the accepted author manuscript (AAM). The final published version (version of record) is available online via ACM at https://doi.org/10.1145/3239332.3242763 . Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/pure/about/ebr-terms

Understanding Formal Specifications through Good Examples

Alex Gerdes
alex.gerdes@cse.gu.se
University of Gothenburg
Gothenburg, Sweden

John Hughes
Chalmers University of Technology,
Quviq AB
Gothenburg, Sweden

Nicholas Smallbone
Chalmers University of Technology
Gothenburg, Sweden

Stefan Hanenberg
University of Duisburg-Essen
Essen, Germany

Sebastian Ivarsson
Chalmers University of Technology
Gothenburg, Sweden

Meng Wang
University of Kent
Canterbury, UK

Abstract

Formal specifications of software applications are hard to understand, even for domain experts. Because a formal specification is abstract, reading it does not immediately convey the expected behaviour of the software. Carefully chosen examples of the software's behaviour, on the other hand, are concrete and easy to understand—but poorly-chosen examples are more confusing than helpful. In order to understand formal specifications, software developers need *good examples.*

We have created a method that automatically derives a suite of good examples from a formal specification. Each example is judged by our method to illustrate one feature of the specification. The generated examples give users a good understanding of the behaviour of the software. We evaluated our method by measuring how well students understood an API when given different sets of examples; the students given our examples showed significantly better understanding.

CCS Concepts • **Software and its engineering** → *Software verification and validation*; *Software defect analysis*; Programming by example;

Keywords Formal specification, examples, QuickCheck, property-based testing

ACM Reference Format:
Alex Gerdes, John Hughes, Nicholas Smallbone, Stefan Hanenberg, Sebastian Ivarsson, and Meng Wang. 2018. Understanding Formal Specifications through Good Examples. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang (Erlang '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3239332.3242763

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Erlang '18, September 29, 2018, St. Louis, MO, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5824-8/18/09…$15.00
https://doi.org/10.1145/3239332.3242763

1 A Running Example

Let us begin by introducing our running example—the Erlang process registry. Erlang is a concurrent programming language in which systems are made up of very many lightweight processes, each with its own process identifier, or '*pid*'. For illustration, we shall create (dummy) processes using a function `spawn()`, which returns the created process' pid as its result. To enable processes to find each other, they may be *registered* under a name, names being Erlang atoms, which are similar to strings in many other languages. For example, we might create and register a process like this:

```
V = spawn()    -> '<0.27813.1>',
register(a, V) -> true,
whereis(a)     -> '<0.27813.1>'
```

Here we show the arguments and results of a sequence of calls in a sample run; `'<0.27813.1>'` is an example of a dynamically created pid, and the atom `a` is the name assigned to the pid in this case. We indicate reuse of a result by binding it to a variable: the result of `spawn` is bound to the variable `V`, then passed as an argument to `register`. Finally, `whereis` looks up a name in the registry and returns the associated pid.

Processes may only be registered once, as the following example shows:

```
V = spawn()    -> '<0.27164.1>',
register(b, V) -> true,
register(a, V) -> {'EXIT', badarg}
```

Here the second call fails with an exception, indicated by a result of `{'EXIT',badarg}`. Neither can the same *name* be registered twice:

```
V1 = spawn()    -> '<0.28614.1>',
V2 = spawn()    -> '<0.28615.1>',
register(c, V1) -> true,
register(c, V2) -> {'EXIT', badarg}
```

Processes can be removed from the registry using `unregister`:

```
V = spawn()    -> '<0.27285.1>',
register(c, V) -> true,
unregister(c)  -> true
```

after which they are no longer returned by `whereis`:

```
V = spawn()    -> '<0.28240.1>',
register(b, V) -> true,
unregister(b)  -> true,
whereis(b)     -> undefined
```

They cannot be removed a second time:

```
V = spawn()    -> '<0.28054.1>',
register(d, V) -> true,
unregister(d)  -> true,
unregister(d)  -> {'EXIT', badarg}
```

Once removed, a process can be re-registered, perhaps with a different name:

```
V = spawn() -> '<0.27520.1>',
register(c, V) -> true,
unregister(c)  -> true,
register(a, V) -> true
```

We hope that, thanks to these examples, the reader now has a good understanding of the behaviour of this simple API. The point, though, is that while the words above were written by the authors, *the examples were generated and chosen automatically* from a formal specification of the API, using the methods that are the subject of this paper.

2 The Question

The question we address in this paper is this:

> *How can we generate examples of API usage, that enable a person to understand its behaviour?*

Previous approaches have extracted examples from a corpus of uses of the API [4, 19]. Such examples are clearly helpful in understanding *how an API is used*, and also in writing new code to use the API in a similar way. But some useful examples may be missed, if the corpus does not contain instances of them. In particular, *negative examples* (such as those that raise exceptions above) are less likely to be found in real code, but do convey useful intuition about what *not* to do. Also, examples extracted from real code must usually be simplified before they are presented to the user, because real code tends to contain irrelevant operations as well as those to be exemplified. Static analysis and program slicing techniques have been used for this.

We instead generate examples from an implementation of the API, together with a *testable formal specification* built using QuickCheck [5], or rather its commercial version Quviq QuickCheck [1]. Such specifications are developed in order to test other code, and have been used to test a wide variety of software [11], the largest project to date being testing of AutoSAR Basic Software on behalf of Volvo Cars [2]. Given such a specification, QuickCheck generates test cases (sequences of API calls) at random, and checks that the implementation's actual behaviour conforms to the specification. When a test fails, QuickCheck searches for a similar-but-simpler test case which also fails, terminating with a (locally) minimal failing test, which is then presented to the user for diagnosis. This '*shrinking*' process can also be used to find minimal tests with other properties, such as tests covering a particular line of code, simply by temporarily redefining 'test failure' to mean 'covers this line'.

QuickCheck specifications can themselves become relatively large and complex (for example, the AutoSAR Basic Software specification is 20KLOC of Erlang [2]). As such, they can be impenetrable for stakeholders such as Volvo Cars. It can even be difficult to tell whether or not a particular test case can be generated (because API operations can have preconditions that might not be satisfiable) [9]. How can we know, then, whether the tests generated from a formal specification are testing the right thing? In this context *examples* of generated tests can be invaluable, not only to show to stakeholders for validation, but to help the specification developer understand what the specification actually says. Random examples are useless for both purposes, though, because they 'make no sense'—they combine unrelated operations in meaningless ways (and this is why they are effective at revealing unexpected interactions!). We need instead a *small* set of *salient examples*, that together convey, to a person, both what is being tested, and the intended behaviour of the API under test.

In the forthcoming sections we present heuristics for identifying such salient examples, and we use QuickCheck's generation and shrinking to produce a suite of minimal examples, each illustrating a different point. Because QuickCheck can generate arbitrary sequences of API calls, then we can construct positive and negative examples that may not exist in any corpus of code. Because QuickCheck always shrinks test cases to minimal ones with a given property, then we do not need any other technique for simplifying the examples that are found. We claim that the resulting suites of examples are good at enabling a person to understand the behaviour of the specified API.

Finally, we have evaluated this claim in human experiments. But how can we determine whether a person 'understands' an API? What do we mean by 'understands'? We have chosen to interpret 'understanding' as the ability to predict behaviour: we measure how well subjects can predict the return values in sequences of API calls, given examples generated by our heuristics. Our experiments show that subjects can predict behaviour significantly better given our examples, than given a very plausible alternative—examples generated to achieve 100% coverage of both specification and implementation.

3 State machine models

The heuristics we have developed work with QuickCheck state machine models [11], which are heavily used for specifying stateful APIs. Here we explain briefly what these models consist of.

The essential idea is to define a *model state* that reflects the state of the system under test at each point in a test case. In the case of our running example, the process registry, the model state consists of:

- a set of pids that have been spawned, and
- a set of name/pid pairs that should currently be associated in the registry.

As a test case is run, QuickCheck computes the model state at each step by calling *state transition functions* defined in the model for each operation. For example, the state transition function for `spawn()` adds the new pid to the model state:

```
spawn_next(Model,Result,[]) ->
  Model#model{pids = Model#model.pids ++ [Result]}.
```

while the state transition function for `unregister(Name)` removes any pair `{Name,Pid}` from the model state.

```
unregister_next(Model,_,[Name]) ->
  Model#model{
    regs = lists:keydelete(Name,1,Model#model.regs)}.
```

These are Erlang functions mapping the `Model` state, an immutable Erlang record (of type '`model`', with fields `pids` and `regs`), to a new record which is a copy of the original, with some different field values.

QuickCheck models specify how to generate calls to each function under test, given the current model state, so calls to `register`, for example, are generated by choosing a pid from the model state. Specifications can also define *preconditions* for each operation—for example, the precondition for `register` says there must be at least one candidate pid to choose from:

```
  register_pre(Model) ->
    Model#model.pids /= [].
```

This precondition depends only on the model state, but preconditions can of course also depend on the arguments of the call. QuickCheck only generates test cases in which all preconditions are satisfied.

Finally, models can specify a *postcondition* for each operation, that checks the actual result returned by the implementation against the model state before the call. For example, the postcondition for `whereis(Name)` checks that the correct pid was returned (or `undefined` if `Name` was not in the registry).

```
whereis_post(Model,[Name],Result) ->
 Result == proplists:get_value(Name,Model#model.regs).
```

Given such a state machine model, QuickCheck generates and runs test cases made up of random sequences of API calls, in which every precondition is satisfied, and considers a test to pass if all postconditions are true, and there are no uncaught exceptions. A failing test is reduced to a minimal example by shrinking, and then presented to the user for diagnosis.

The reader may be wondering whether the model just described is not essentially the same as the implementation? This is not the case. The model is just a few lines of Erlang code, while the implementation is one or two *thousand* lines of C in the guts of the Erlang virtual machine—because it uses far more efficient data structures, manages memory explicitly, and needs to be thread-safe on multicore processors. So the model is indeed much simpler than the implementation whose behaviour it specifies.

4 The idea: finding interactions

What makes an example 'interesting'? Our intuition springs from the *uninteresting* examples that random generation largely produces: sequences of calls that do not interact, so that one wonders why they appear together in the same example. Indeed, *all the calls in an interesting example should be essential to the example*—in other words, they should all interact with each other in some way. We focus, therefore, on finding examples that illustrate *an interaction between two calls in the sequence.*

For example, consider the very first example we presented in this paper:

```
V = spawn()    -> '<0.27813.1>',
register(a, V) -> true,
whereis(a)     -> '<0.27813.1>'
```

We consider this example to be interesting because it illustrates an interaction between `register` and `whereis`: the call of `register` affects the return value of the later call to `whereis`.

How can we verify that `register` affects `whereis`? We do so by considering a *negative example* obtained by deleting the call of `register`:

```
V = spawn(),
whereis(a)
```

Clearly, if we were to run this test, then `whereis(a)` would return a different result—`undefined`—because the name `a` has not been registered. But in general, we cannot identify a dependence by checking whether a return value changes when a test is re-run. The reason is that a return value may change for other reasons. For example, rerunning the *original* test case may result in

```
V = spawn()    -> '<0.62.0>',
register(a, V) -> true,
whereis(a)     -> '<0.62.0>'
```

in which `whereis` returns a different result just because pids are allocated dynamically, and so can vary between runs.

Instead we consider a negative example in which each call returns the *same* result as in the original run, but the call to `register` is simply omitted:

```
V = spawn()     -> '<0.27813.1>',
whereis(a)      -> '<0.27813.1>'
```

The key observation is that *this behaviour can never happen*, because this sequence of calls and results would *violate the postcondition* of `whereis`. Thus we can determine that the call to `register` affects the result of `whereis` without running another test! All we need is the model, to verify that the actual result returned by `whereis` would have been rejected by the specification if the call to `register` had not occurred, thus implying that `whereis` would have returned a different value had the call to `register` not been made.

We say that this test case exhibits the

'`register/whereis postcondition`'

interaction, illustrating a call of `register` that affects the postcondition of a subsequent call of `whereis`. In general, any test case containing a call to `register`, whose removal would cause the postcondition of a subsequent call to `whereis` to fail, is said to exhibit the same interaction. Now we can build up a suite of examples illustrating different interactions by running random tests, and using the model to collect all the interactions that each test exhibits. If we observe an interaction in a random test for which there is not yet an example in our suite, then we use QuickCheck's shrinking to obtain a minimal test case that exhibits that interaction, and add it to the suite of examples. When random testing fails to find any new interactions after a sufficiently long period, the suite of examples is complete.

For example, suppose we add a `stop(Pid)` operation to our registry tests, which stops (kills) a running process. Suppose we extend the specification to cover the behaviour of `stop`, which requires adding a set of 'dead pids' to the model state, and specifying different behaviours for live and dead processes. Then our example generator finds three new examples (which we have labelled with the interaction that they exhibit):

```
%% stop/whereis postcondition
V = spawn()     -> '<0.27922.1>',
register(a, V) -> true,
stop(V)         -> ok,
whereis(a)      -> undefined
```

which shows that `whereis` will not find dead pids in the registry,

```
%% stop/unregister postcondition
V = spawn()     -> '<0.27387.1>',
register(c, V) -> true,
stop(V)         -> ok,
unregister(c)   -> {'EXIT', badarg}
```

which shows that dead pids can no longer be removed from the registry, and

```
%% stop/register postcondition
V1 = spawn()    -> '<0.28706.1>',
register(b, V1) -> true,
V2 = spawn()    -> '<0.28707.1>',
stop(V1)        -> ok,
register(b, V2) -> true
```

which shows that the names of dead pids can be re-used in the registry. In fact, all three examples illustrate the same behaviour—that processes are removed from the registry when they die—and show the impact on each of the three registry operations. These examples are typical of those that our heuristic generates: they contain one call which affects the postcondition of the final call in the example, and a minimal set of other calls to set up a situation in which the first call can affect the second.

4.1 A refinement

The reader should now find it easy to return to the very first section of this paper, and identify the interaction that each generated example exhibits—except that there are *two* examples in that section that exhibit the '`register/register postcondition`' interaction. They are:

```
V = spawn()     -> '<0.27164.1>',
register(b, V) -> true,
register(a, V) -> {'EXIT', badarg}
```

and

```
V1 = spawn()    -> '<0.28614.1>',
V2 = spawn()    -> '<0.28615.1>',
register(c, V1) -> true,
register(c, V2) -> {'EXIT', badarg}
```

These two examples illustrate an important refinement to our method.

So far, the method we described abstracts away completely from the *arguments* to each call—if we find a test case in which a call to register affects the result of a later call, we just say it exhibits the '`register/register postcondition`' interaction, ignoring the arguments completely. This is quite deliberate, since we aim to find an example for each such interaction, and we would not wish to generate *separate* examples that just differ in, say, the choice of name for a process. Yet, just because we don't want an example generated for each possible argument value, doesn't mean that arguments are entirely irrelevant. We have observed that, while the choice of argument value is often not important, *repetition of values* often is. Thus, instead of discarding argument values completely when identifying interactions, we discard them but record the repetition of values. We do so by including wild card arguments for those that don't matter, and named metavariables for those that are repeated. Thus we say that the first example above exhibits the

'`register(_,?A)/register(_,?A) postcondition`'

interaction, where `?A` is a metavariable that must take the same value in both calls for the interaction to be exhibited—in other words, we must `register` the same pid twice. The second example exhibits the

'`register(?A,_)/register(?A,_) postcondition`'

interaction instead, registering the same *name* twice, and so we are able to distinguish these two interactions and generate a separate example for each one.

5 Precondition interactions

As we explained in Section 3, QuickCheck models can specify a precondition for each operation under test. This introduces another kind of potential interaction: an earlier call in a test case might affect the *precondition* of a later call, rather than its result. These interactions are also interesting—they show that the earlier call *enables* the later one—and so we add '`precondition`' interactions to the set of interactions that test cases can exhibit, and gather examples to illustrate these interactions too.

In our running example, we have hitherto used a specification without preconditions. Both `register` and `unregister` raise exceptions sometimes, but we have used a model that catches those exceptions, and postconditions that check that exceptions are thrown when they ought to be. Test cases generated from this model include *negative tests* that deliberately provoke exceptions, and indeed, three of the seven examples presented at the beginning of the paper are of this sort.

Alternatively, we could specify preconditions for `register` and `unregister` that exclude cases in which an exception would be raised. This alternative model would generate only *positive* tests of the registry. The suite of examples generated from this alternative model contains examples that illustrate precondition interactions, such as this one:

```
%% register(?A,_)/unregister(?A) precondition
V = spawn()     -> '<0.29350.1>',
register(b, V) -> true,
unregister(b)  -> true
```

Interestingly, essentially the same example appears in the very first section of this paper, but there it was illustrating the

'`register(?A,_)/unregister(?A) postcondition`'

interaction! The same example was generated from our original model, because the negative example

```
V = spawn()     -> '<0.29350.1>',
unregister(b)  -> true
```

violates the postcondition of `unregister` (because it must raise an exception in this case, rather than return `true`), while it is generated from our alternative model because the precondition of `unregister` is not satisfied in this negative example. The underlying reason why the example is interesting is the same, but our two models represent `unregister`'s failure behaviour differently, and so the same example is generated to illustrate different interactions in each case.

In fact, the set of examples generated for the registry is essentially the same whichever model we use, except that examples that illustrate how to provoke exceptions cannot be generated from the second model, because the preconditions in the model exclude those tests. As a result, we end up with six examples rather than ten, of which three illustrate precondition interactions, and three illustrate postcondition ones.

Finally, the reader may at this stage be wondering why preconditions are useful at all—wouldn't it always be better to check for failure in postconditions instead, and include negative tests in one's testing? In this simple case that is probably so, but this is only because `register` and `unregister` have well-defined behaviour even when their preconditions are *not* satisfied. In general, violating preconditions in tests may cause some operations to behave quite unpredictably, making it dangerous or pointless to continue a test afterwards. So the ability to restrict generated tests by specifying preconditions is essential—and thus it's important to take them into account in example generation too.

6 Extensions

Initial experiments with our generator showed promise, but also revealed weaknesses. This led us to make three further extensions to the generator, before conducting our main evaluation.

6.1 Argument interactions

Hitherto, we detect an interaction between operation A and operation B by asking whether *removing* operation A would change the result of operation B. That is, the only modification we consider to operation A is removing it completely. But smaller changes to operation A—such as changing some of its arguments—might also have an impact on the result of B. To find examples of this sort we implemented a heuristic that tries to construct a negative example by replacing the arguments of an operation in a test case with other randomly chosen arguments, and then asking whether the pre- or post-condition of a later operation now fails.

An example that can be found using this heuristic is the following, taken from the 'bank server' API used for our main evaluation:

```
open()              -> ok
create_user(u1,p1) -> {u1,p1}
create_user(u2,p1) -> {u2,p1}
```

Here `open()` starts the bank server, and `create_user` creates a user with a given username and password. The example shows that multiple users can be created, but it would not be generated by our original heuristics because removing the first call of `create_user` does not change the behaviour of the second. However, changing the *username* on the second

line to u2 causes the call on the third line to fail, because it attempts to create a user with an existing username.

Argument interactions can also affect the precondition of a later command. This example, using an API for a fixed capacity queue (represented in C by a pointer to a struct Queue), is generated to illustrate this kind of interaction:

```
V = q:new(2)  -> {ptr,Queue,876123480}
q:put(V,0)    -> ok
q:put(V,0)    -> ok
```

The example illustrates that we can put two items into a queue with capacity two, and it is generated by our new heuristic because changing the first call to q:new(1) (which creates a queue with a capacity of one item) causes the precondition of the second put operation to fail.

We did find that this heuristic can generate some of the *same* examples as our original heuristic, because it is often possible to choose arguments for the first operation that make it irrelevant, and this has much the same effect as removing the operation altogether. To avoid a proliferation of similar examples, we include examples generated by this heuristic if they do not also exhibit the same interaction according to our original heuristic.

6.2 'Undo' interactions

One of the APIs we used in our preliminary evaluation was the Erlang dets API[1], which stores a table consisting of a set of key-value pairs in a file. A simple example using the API is this:

```
%% insert/lookup postcondition
open_file(table,[{type,bag}]) -> table
insert(table,{2,0})           -> true
lookup(table,2)               -> [{2,0}]
```

which shows a pair ({2,0}) being added to the table, and then retrieved by looking up its key. To our surprise, we found that given only our generated examples, our experimental subjects were uncertain whether the contents of the file would persist when the file was closed and re-opened.

To resolve this uncertainty, we need an example such as this one:

```
open_file(table,[{type,bag}]) -> table
insert(table,{2,0})           -> true
close(table)                  -> ok
open_file(table,[{type,bag}]) -> table
lookup(table,2)               -> [{2,0}]
```

But this example cannot be generated using the heuristics presented so far! While it *does* exhibit the 'insert/lookup postcondition' interaction, it is not a *minimal* example of that interaction; QuickCheck will shrink it to the first example above. Therefore it will never be included in an example suite.

[1] http://erlang.org/doc/man/dets.html

Indeed, the interesting thing about this pair of examples is *not* that anything behaves *differently* in the two cases—it is that *everything behaves the same*, whether the close/open_file sequence is included, or not. Or, to put it another way, we can *undo* the effect of close(table) on the subsequent lookup (closing the table causes it to fail), by adding an open_file call after the close.

We call this kind of interaction an '*undo interaction*' between close and lookup, and generate minimal examples of all undo interactions we can find also. To qualify as an undo interaction, just removing the call to the first operation (close) must cause the second (lookup) to fail, but there must be a *sequence of calls* starting with close whose removal repairs the damage, allowing the second call to succeed again. Such a sequence first changes the state, and then restores it, at least well enough for the second call to behave as it did before. This is a fundamentally different kind of interaction from those we considered above, but no less interesting.

Notice that we classify this interaction as a 'close/lookup undo' interaction, not a 'close/open_file/lookup undo' interaction. The reason is that we aim to find a *minimal* example for each class of interaction—that is, a minimal way of 'undoing' the effect of a close before a lookup. If we included the operations that implement undo in the class of the interaction, then shrinking would not be able to eliminate irrelevant operations *without changing the class.* Since we include one example in our generated suite for each class of interaction that we observe, then that would have the effect of including an example for *every possible* sequence of operations that can undo a close, including sequences with many redundant operations. This would be quite undesirable.

We did find some of the examples generated by this heuristic to be a little uninteresting, such as this one:

```
open_file(table,[{type,bag}]) -> table
close(table)                  -> ok
open_file(table,[{type,bag}]) -> table
lookup(table,0)               -> []
```

In a sense this illustrates a 'close/lookup undo' interaction—because we need to reopen the table in order to call lookup. But this call to lookup is rather uninteresting, because it fails to find anything (since the key has not been inserted). To prevent this example from displacing the one above in our example suite, we allow the specification author to classify some calls as 'boring'—calls of lookup that find nothing, calls that raise an exception, and so on. Boring calls are not used as the second call in an undo-interaction, and so by classifying this call of lookup as boring, we can ensure that our example generator chooses the first rather than the second 'close/lookup undo' interaction example.

This heuristic contributed examples such as the following one to the suite of 'bank server' examples we used for our evaluation:

```
open()                  -> ok
create_user(u1, p1)     -> {u1, p1}
login(u1, p1)           -> ok
create_account(a2, u1) -> {a2, u1}
close()                 -> ok
open()                  -> ok
login(u1, p1)           -> ok
deposit(u1, p1, a2, 1) -> 1
```

This example illustrates a 'close/deposit undo' interaction. The first four steps open the bank, create and login a user, and create an account—at which point, it would be possible to deposit money in the account. Instead, we close the bank, and to recover to the point where we can make a deposit, we have to both reopen the bank, and log the user back in.

6.3 Negative examples

We initially presented examples to users along with an explanation of why the example was considered interesting. For example,

```
Good example:
1.  V = spawn()     -> '<0.29507.1>',
2.  register(d, V) -> true,
3.  whereis(d)      -> '<0.29507.1>'
Deleting command 2 changes the behaviour of command 3
1.  V = spawn()     -> '<0.29507.1>',
3.  whereis(d)     -> should not return '<0.29507.1>'
```

We soon found that these explanations were not particularly helpful to users, who, after all, should not need to understand our heuristics to find the examples useful. In particular, the concept of 'should not return' was hard to explain. We therefore simply added the negative examples to the suite, and reran them to show what those calls actually *do* return instead. In this case, we might add the example

```
V = spawn()     -> '<0.65.0>',
whereis(d)      -> undefined
```

to our suite, showing that if no process is registered, then `whereis` returns `undefined`. In most cases, we found the negative examples are themselves of some interest.

7 Evaluation

We conducted an experiment to support our claim that our method generates a suite of good, representative examples for understanding the behaviour of a specified API. We have chosen to interpret 'understanding' as the ability to predict behaviour: we measure how well subjects can predict the return values in sequences of API calls, given examples generated by our heuristics.

We compare the suite of good examples to a suite of reference examples. Instead of comparing to random examples, we use a suite of generated examples that achieves a 100% coverage of both the specification and implementation. We believe this is a fair comparison since code coverage is a widely used method to measure the quality of a test suite. We constructed the reference suite by accumulating generated random examples until we obtained full code coverage. This resulted in a suite of 22 reference examples. Note that the generated reference examples are *minimized* test cases (i.e., they are 'shrunk'). Even though we use randomness in its generation, this minimal reference test suite with 100% coverage is generated in essentially the same form (modulo renamings) in every run.

The set of good examples generated by our heuristics had a size of 55, and was obtained by combining the results of several iterations. Although we use randomness in the generation of our example suites, the final example suites generated differ only inconsequentially from generation run to generation run. Arbitrary names may be different, but there are few other differences. This is because we *minimize* each new test as it is found, to the simplest similar test that covers a new feature. Moreover, we run so many random tests that we are almost certain that any features that do not appear in the suite of examples are not reachable.

The experiment tests the following hypotheses:

$H0_1$: There is no difference in the number of correct answers for subjects given reference examples and good examples.

$H0_2$: The number of correct answers does not depend on the given tasks.

$H0_3$: There is no interaction between the given tasks and the given examples with respect to the number of correct answers.

To test the above hypotheses we have defined an experiment in which subjects perform a number of tasks. A task consists of a sequence of API calls, which is similar to a generated example, of which the subject needs to predict the evaluation. Each individual API call needs to be predicted. We have used the 'bank server' API in our experiment, which supports the following API calls:

- `open/0`
- `close/0`
- `create_user/2`
- `create_account/2`
- `logged_in/2`
- `login/2`
- `logout/1`
- `deposit/4`
- `withdraw/4`

In Section 6 we showed a number of generated example interactions using the above API. The model (state machine specification) for the 'bank server' API consists of 250 lines of Erlang code and is developed by the authors, the same goes for the other models in this paper. Note that these models are not primarily created for generating examples, but for testing the implementation of an API.

The subjects in our experiment received a sheet of paper with seven tasks and some guidelines. The subjects didn't have access to the model. The subjects were, for example, asked to mark the evaluation with a question mark in case the subject could not predict an evaluation, rather than taking a wild guess. Furthermore, the subjects were given a completed example task:

```
1.  create_user(u1, p1)         -> call not allowed
2.  open()                      -> ok
3.  create_user(u1, p1)         -> {u1, p1}
4.  cmd you are unsure about()  -> ?
5.  withdraw(u1, p1, a1, 42)    -> false
```

Selection of tasks The tasks were constructed such that they are non-trivial and require a good understanding of the 'banking server', but are still solvable in about 15 minutes. We ended up with seven tasks with a varying number of API calls ranging from 4 to 21. Each correct prediction of the evaluation of an API call is rewarded with one point. The maximum score for a task is equal to the number of API calls. The next sequence of API calls is an example of the tasks presented to the subjects:

```
1.  open()                   -> ____
2.  create_account(a1, u2)   -> ____
3.  create_user(u3, p2)      -> ____
4.  login(u3, p2)            -> ____
5.  create_account(a1, u3)   -> ____
6.  create_user(u2, p1)      -> ____
7.  deposit(u3, p2, a1, 4)   -> ____
8.  withdraw(u3, p2, a1, 3)  -> ____
9.  withdraw(u3, p2, a1, 2)  -> ____
```

We started out to generate the tasks randomly to avoid any human bias, but discovered that the generators needed much tuning to generate non-trivial and proper tasks. This tuning is not straightforward and would introduce a human bias again. Instead, we created the tasks by hand using randomly generated examples as a starting point. We made sure that the following interesting scenarios were included:

- Failing preconditions
- Persisting state after close-open
- Users being logged out after closing the bank
- Several successful deposit and withdraw operations with a changing balance
- Using several users and accounts in the same task

Experiment execution The subjects for the experiment were students enrolled in a course on parallel functional programming at Chalmers University. We chose this particular course because it makes extensive use of the programming language Erlang. In total there were 22 subjects taking part in the experiment. The subjects were randomly divided into two groups: a subject group and reference group. The subject group received a sheet of paper with the generated suite of good examples based on our heuristics, whereas the reference group were handed a sheet of paper with the suite of examples with 100% code coverage. Both groups were given the same tasks. The experiment was carried out anonymously and no personal information about the subjects was recorded. The subjects were given 15 minutes to perform the given tasks, which was enough for nearly all subjects to complete all tasks.

Table 1. Raw measurements and descriptive statistics

		Tasks						
Subject	**Group**	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Task 6**	**Task 7**
1	1	100.00	50.00	100.00	100.00	90.00	64.29	61.90
2	1	100.00	75.00	66.67	88.89	70.00	64.29	80.95
3	1	80.00	25.00	66.67	66.67	80.00	42.86	4.76
4	1	80.00	50.00	100.00	44.44	80.00	57.14	0.00
5	1	100.00	75.00	66.67	100.00	10.00	0.00	0.00
6	1	80.00	50.00	66.67	77.78	80.00	57.14	28.57
7	1	100.00	50.00	100.00	77.78	100.00	14.29	0.00
8	1	60.00	75.00	100.00	44.44	0.00	0.00	0.00
9	1	100.00	50.00	83.33	100.00	80.00	92.86	76.19
10	1	60.00	50.00	83.33	100.00	60.00	28.57	23.81
11	1	100.00	100.00	100.00	100.00	80.00	71.43	80.95
12	2	80.00	100.00	100.00	88.89	100.00	57.14	85.71
13	2	80.00	100.00	100.00	88.89	100.00	57.14	85.71
14	2	80.00	100.00	100.00	100.00	100.00	100.00	95.24
15	2	100.00	100.00	83.33	77.78	70.00	71.43	85.71
16	2	100.00	50.00	100.00	100.00	80.00	78.57	33.33
17	2	80.00	100.00	100.00	100.00	100.00	85.71	33.33
18	2	100.00	100.00	100.00	100.00	80.00	100.00	0.00
19	2	80.00	100.00	100.00	100.00	100.00	85.71	0.00
20	2	80.00	100.00	66.67	77.78	80.00	64.29	0.00
21	2	100.00	75.00	100.00	77.78	80.00	28.57	28.57
22	2	100.00	75.00	100.00	100.00	90.00	85.71	42.86

		Tasks						
metric	**Group**	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Task 6**	**Task 7**
minimum	1	60.00	25.00	66.67	44.44	0.00	0.00	0.00
	2	80.00	50.00	66.67	77.78	70.00	28.57	0.00
maximum	1	100.00	100.00	100.00	100.00	100.00	92.86	80.95
	1	100.00	100.00	100.00	100.00	100.00	100.00	95.24
arithmetic mean	1	87.27	59.09	84.85	81.82	66.36	44.81	32.47
	2	89.09	90.91	95.45	91.92	89.09	74.02	44.59
median	1	100.00	50.00	83.33	88.89	80.00	57.14	23.81
	2	80.00	100.00	100.00	100.00	90.00	78.57	33.33
standard-deviation	1	15.43	19.28	15.00	20.80	30.53	29.07	33.78
	2	9.96	16.07	10.28	9.58	10.83	20.27	35.82

7.1 Raw Measurements and Descriptive Statistics

Table 1 contains the raw measurements for the experiment and the descriptive statistics in terms of percentages of corrects answers given by subjects for the corresponding task. Percentages are used in order to increase the comparability between the different tasks, because the number of possible faults differ between different tasks (because the tasks differ in the number of API calls). The data indicates that the group with the good examples performs in general better than the

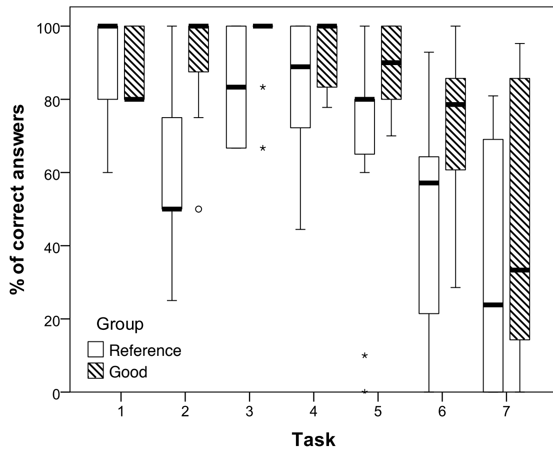

Figure 1. Boxplot for measurements

reference group: for all tasks (except task 1) arithmetic mean and median is larger for group 2 (the good examples group) compared to group 1 (reference examples). For task 1, the median for group 2 is smaller than for group 1, but the arithmetic mean is still larger for group 2. If we focus on minimum and maximum, we get the same expression: for group 2 the minimum is at least as large as the minimum for group 1 (for 5 tasks it is larger) and for group 2 the maximum is at least as large as the maximum for group 1 (for five tasks, the maximum for both groups is 100%, in the other two cases group 2 has a larger maximum). Additionally, the standard deviations for group 2 are in all cases except task 7 smaller than for group 1.

The boxplot in Figure 1 seems to suggest the same: the boxplots for group 2 are at least as high as the boxplots for group 1 and for task 2, 3, and 5 the differences seem to be quite large. On the other hand, there does not seem to be a clear rule that could explain the differences. For example, for task 7, which has the lowest median for group 1, the difference to group 2 does not seem to be that strong as for example the difference in task 5 (where the median for group 1 is much larger than the median for task 7).

7.2 Analysis

We analyse the data using a repeated measures ANOVA with the within-subject variable task (with seven treatments) and the between-subject variable group. The analysis was performed with the statistics package SPSS v24.

The repeated measures ANOVA (where based on Mauchly's test the assumption of sphericity was not violated, because $\tilde{\chi}^2(20)=26.51$, p=.16) reveals that the within-subject factor task had a significant influence on the response variable correctness (F(6,120)=18.67, p<.001). The effect size of the factor task is $\eta_p^2=.48$, which is according to Cohen [6] a large effect. The between-subject factor group was significant as well (F(1,20)=9.89, p=.005) and the effect was large as well $\eta_p^2=.33$. Finally, there was no significant interaction between both factors task*group (F(6,120)=2.01, p=.141, $\eta_p^2=.08$).

The difference in means between group 1 (M=65.24) and group 2 (M=82.15) over all tasks was -16.92±11.22 (95% CI), i.e. group 2 had on average a 16% higher score in the answers.

However, the measured effect of the factor task says that tasks themselves differ (independent of the group). But since the main goal of the experiment was not to reveal differences in tasks, but differences in groups (for different tasks), we just run a separate t-test for each task. This test does not reveal any differences for the tasks 1, 3, 4, and 7 (p_{task1}=.76, p_{task3}=.08, p_{task4}=.185, p_{task7}=.44).

For task 2 group 2 was on average 31%±16.55 better (p=.001), for task 5 group 2 performed on average 22.72%±21.37 better, and for task 6 group 2 was on average 29.22%±23.38 better – it seems as if the variable group had the strongest effect on task 2 while the influence for task 5 and 6 was comparable. Note, that there was no difference between both groups for task 1 although the boxplot seemed to suggest a rather negative impact of group 2 on the results.

7.3 Threats to Validity

Following the reporting guidelines by Wohlin et al. [22], experiments should report potential threats to validity. However, while Wohlin et al. propose a classification for these threats, we mention here only those ones that we consider very specific to the experiment (and ignore others that depend for example on the sample size and resulting problems such as the statistical power of the chosen experimental design).

Chosen environment: Although we did not measure a difference in the tasks, we still believe that the impact of the good examples depends on the given API, the given programming language, etc. For example, it is known that just the choice of identifiers influence the understandability of code (see for example [10, 15]). Furthermore, it is known that language constructs such as type systems influence the understandability of code as well (see for example [7, 8] among many others). Although we believe these factors influence the usage of examples as well, we have no evidence for it yet.

Complexity of API code: We think the complexity of API code has an influence on its usability as well. Because of that, we have chosen tasks with different complexities (especially task 1, 2, and 3 are from our point of view different in their complexities in comparison to tasks 6 or 7). However, while the experiment revealed differences between the tasks (factor task was significant) which says that the differences of the tasks had an influence on the response variable, the experiment did not reveal any interaction effect. This means, the experiment was not able to show whether good examples have different effects on code with different complexities.

Measurements (I): We decided to use correctness of all tasks as the dependent variable while we kept the time required by the subjects constant. As a consequence, subjects were permitted to decide on their own, how much time they invest for each task. We cannot exclude that the rather positive results for the first few tasks (except task 2 where the reference group's were quite incorrect) and the rather negative results for the latter tasks were just the results of the given time constraints in combination with the ability to decide on their own where the time should be invested. It is possible that the rather negative results of the latter tasks were just caused by time pressure.

Measurements (II): Another alternative could have been to measure time until correct answers were given, an approach that was practiced a lot for different language constructs (see for example [7, 8, 13, 21] among many others). We think that such a different kind of measurement could be possible for the present approach as well, but probably requires a totally different kind of experimental setup. The mentioned works that use time measurements use within the experiment programming tasks with feedback where feedback "the proposed solution is wrong" does not directly give advice how to correct the proposed solution. This probably means that much more complex APIs would be required (with more possible responses). So far, we do not know whether a different kind of measurement would lead to different results.

7.4 Interpretation and Discussion

We found a significant and large effect of the factor group: having good examples increases on average the number of correct answers by -16.92±11.22. The effect size for this factor was large(η_p^2 =.33), but the effect of the factor tasks which was significant as well, was even larger (η_p^2 =.48). I.e. hypotheses $H0_1$ and $H0_2$ can be rejected which supports our intuition that good examples do help using APIs.

For the whole experiment, we did not find an interaction effect between the group and the tasks, i.e., we accept hypothesis $H0_3$. However, making an individual analysis of each single task showed that four of the seven tasks did not reveal a measurable impact of the factor group while for the other three tasks, this effect existed. Furthermore, it looks like the effect for task 2 was slightly stronger than for task 5 and 6 (measured in terms of difference in means). However, again, there was no measurable interaction effect between both variables and we should not interpret too much into it. Actually, we still do believe that good examples will have a different impact on easy or hard to use APIs (in comparison to reference examples), but so far, we are not able to show it.

Although the positive and large effect of good examples was shown, we think that future studies should concentrate on possible time benefits caused by good examples. That is, although we have now evidence that good examples increase the correctness of given answers, and although we believe that this should in principle mean that good examples give a time benefit for the usability of APIs, we have so far no evidence for that.

8 Comparison to related work

There is a large body of work on picking out good examples for an API from an *existing* corpus of code which uses the API. The most popular approach [4, 12, 16, 18, 19] is to pick one API call from a program in the corpus, then choose a fragment of the program which contains that API call and enough context to be a useful example. The difficulty is expanding the single API call into a program fragment which is neither too small (no context) nor too large (irrelevant information); most approaches use program slicing [12, 18, 19] or static analysis [4], but one approach [16] simply takes the whole method containing the API call as the example. Another problem is removing similar or equivalent examples; this is often solved by computing clusters [4, 12] or detecting clones [19]. The generated examples are high quality, provided that a sufficiently large and varied code corpus can be found.

The difference between this work and ours is: we work from a specification, not an existing code corpus, and we use black-box testing to compute examples, not static analysis or program slicing. By avoiding program analysis, our approach is simple and widely usable—all we need is to be able to run the API functions and the state machine specification. Because we work from a specification, we can generate a use of a method without seeing it in a code corpus—this means we can easily generate unusual examples and negative examples, which are unlikely to be found in a corpus but give useful intuition about the API.

Mittal and Paris [17] generate examples of valid Lisp syntax given a grammar for Lisp. Like us, they generate both positive and negative examples; the authors stress the importance of negative examples in understanding what expressions are valid. The tool takes great care to generate good examples: it simplifies the examples, mutates the examples to see which parts are essential and which are optional, and even presents the examples in a careful order to build a narrative. The idea of simplifying examples to minimal interesting ones and analysing them by mutating them is similar to our approach.

Finally, there have been attempts to translate specifications and models to natural language. Lavoie et al. [14] translate data models into prose describing the constraints on the data, together with example data fulfilling the constraints. The generated prose and examples are excellent, and resemble what a human would write. Swartout [20] and Burke and Johannisson [3] translate function pre- and postconditions into natural language in a readable way; Swartout's work also produces a description of the data model.

These tools generally try to describe the model, rather than provide concrete examples. This seems to work well for describing structural information such as class hierarchies, data models and so on. However, it is less convincing when applied to specifications: formal specifications translated to readable English still hide many subtleties. Examples are needed to explain those subtleties.

9 Conclusion

Formal specifications are, by their very nature, terse and technical; examples may aid understanding, but only when they are *good*. In this paper, we have presented a system that refines random test cases into good examples. The system relies purely on black box testing, and does not need to be supplied with a code corpus.

In our approach, the *interaction* between API calls in a test case guides the choice of relevant statements and their relative ordering in examples. As a result, the example suites we generate are representative and minimal but at the same time comprehensive. We have implemented the system using Quviq QuickCheck in Erlang. The approach, however, is not specific to Erlang; it only assumes that sequences of API calls can be run and their results checked. Our current implementation happens to be built using Erlang QuickCheck (but the system under test can be written in any language which has an Erlang bridge, e.g., C, Java).

Our experimental evaluation tests students' understanding of an API when given differing sets of examples. The results show that our examples significantly help with program understanding.

Acknowledgments

We would like to thank the students who were kind enough to take part in the evaluation.

We would also like to thank the anonymous referees for their valuable comments and helpful suggestions. The work is partially supported by GRACeFUL (Horizon 2020 Framework project, grant No.: 640954), Prowess (Framework 7 project, grant No.: 31782), RAWFP (Swedish Foundation for Strategic Research), and SyTeC (Vetenskapsrådet).

References

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, 2–10.

[2] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 1–4.

[3] David A Burke and Kristofer Johannisson. 2005. Translating formal software specifications to natural language. In *International Conference on Logical Aspects of Computational Linguistics*. Springer, 51–66.

[4] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 782–792. http://dl.acm.org/citation.cfm?id=2337223.2337316

[5] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.

[6] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences*. Routledge.

[7] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How Do API Documentation and Static Typing Affect API Usability?. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 632–642. https://doi.org/10.1145/2568225.2568299

[8] Lars Fischer and Stefan Hanenberg. 2015. An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability Using TypeScript and JavaScript in MS Visual Studio. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 154–167. https://doi.org/10.1145/2816707.2816720

[9] Alex Gerdes, John Hughes, Nick Smallbone, and Meng Wang. 2015. Linking Unit Tests and Properties. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang (Erlang 2015)*. ACM, New York, NY, USA, 19–26. https://doi.org/10.1145/2804295.2804298

[10] J. Hofmeister, J. Siegmund, and D. V. Holt. 2017. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 217–227. https://doi.org/10.1109/SANER.2017.7884623

[11] John Hughes. 2016. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*. Springer, 169–186.

[12] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2009. Adding examples into Java documents. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 540–544.

[13] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. Do static type systems improve the maintainability of software systems? An empirical study. In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*. 153–162. https://doi.org/10.1109/ICPC.2012.6240483

[14] Benoit Lavoie, Owen Rambow, and Ehud Reiter. 1996. The ModelExplainer. In *Proceedings of the 8th international workshop on natural language generation*. 9–12.

[15] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*. IEEE Computer Society, Washington, DC, USA, 3–12. https://doi.org/10.1109/ICPC.2006.51

[16] Lee Wei Mar, Ye-Chi Wu, and Hewijin Christine Jiau. 2011. Recommending proper API code examples for documentation purpose. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*. IEEE, 331–338.

[17] Vibhu O Mittal and Cécile Paris. 1994. Generating examples for use in tutorial explanations: using a subsumption based classifier. In *Proceedings of the 11th European Conference on Artificial Intelligence*. John Wiley & Sons, Inc., 530–534.

[18] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. 2013. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 401–408.

[19] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 880–890. http://dl.acm.org/citation.cfm?id=2818754.2818860

[20] William R Swartout. 1982. GIST English Generator.. In *AAAI*. 404–409.

[21] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An Empirical Study on the Impact of

C++ Lambdas and Programmer Experience. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 760–771. https://doi.org/10.1145/2884781.2884849

[22] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2