Linking Unit Tests and Properties

Alex Gerdes, John Hughes
QuviQ and Chalmers University of Technology, Gothenburg, Sweden
{alexg, rjhm}@chalmers.se

Nick Smallbone
Chalmers University of Technology, Gothenburg, Sweden
nicsma@chalmers.se

Meng Wang
University of Kent, Canterbury, UK
m.w.wang@kent.ac.uk

Abstract

QuickCheck allows us to verify software against particular properties. A property can be regarded as an abstraction over many unit tests. QuickCheck uses generated random input data to test such properties. If a counterexample is found, it becomes immediately clear what we have tested. This is not the case when all tests pass, since we do not (and shall not) see the actual generated test cases. How can we be sure about what is tested? QuickCheck has the ability to gather statistics about the test cases, which is insightful. But still it does not tell us whether the particular unit test scenarios we have in mind are included. For this reason, we have developed a tool that can answer this question. It checks if a given unit test can be generated by a property, making it easier to judge the property's quality. We have applied our tool to an industrial use case of testing the AUTOSAR basic software modules and shows that it can handle complex models and large unit tests.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords QuickCheck, Unit tests, Property-Based Testing, Erlang

1. Introduction

QuickCheck [6] is a tool that tests universally quantified *properties*, which are abstract specifications of the system under test. Using the Erlang QuickCheck library [1, 9], properties can be expressed via Erlang function definitions. For example, a property of the `delete` function from the Erlang `lists` module can be written as

```
prop_delete() ->
  ?FORALL({X, Xs}, {int(), list(int())},
          not lists:member(X, lists:delete(X, Xs))).
```

The property states that if we delete an integer from a list of integers it should no longer be present in the list returned by the `delete` function. QuickCheck verifies properties like this by generating a large number of random test cases, and reports test cases for which the property fails. When QuickCheck finds a failing test case, a so-called counterexample, it shrinks the failing test case automatically, by searching for similar, but smaller test cases that fail as well. The result of shrinking is a minimal failing test case simplifying the debugging process.

Properties are powerful: a good property gives a strong specification for a large set of test data. But the more powerful the property, the greater the risk that it becomes abstract and inscrutable. Unit tests are the opposite. A single unit test describes exactly one behaviour of the system. As a result, unit tests are concrete and easy to understand. It is not uncommon to have some unit tests in mind when defining a property. If we, for example, want to test the above mentioned `delete` function, we may want to have a test case that validates that every occurrence of an element is actually removed from the list. For example, if we delete `3` from `[1,2,3,1,3]` it should yield the following list: `[1,2,1]`. Another test case we can think of is that the list remains unchanged if we try to delete an element that is not present in the list.

Generalising these kind of unit tests leads to properties such as `prop_delete()`, which certainly captures the logic considerations, and the test data generator `{int(), list(int())}` produces pairs of an integer and a list of integers as test cases. Running `quickcheck` then performs (by default) 100 random tests of the property, and in this case, they all passed—each dot represents a successful test:

```
1> eqc:quickcheck(prop_delete()).
..................................................
..........................................
OK, passed 100 tests
```

We may think at this point, after a hundred successful tests, that we have tested the `delete` function well enough. However, if we apply the `delete` function on the unit test case we had in mind, we get an unexpected result:

```
2> lists:delete(3, [1,2,3,1,3]).
[1,2,1,3]
```

As it turns out the `delete` function only removes the first occurrence of an element in the list, and not all as we expected! So why did QuickCheck not find this error? In retrospect, it's clear why: `X` is chosen from `int()`, while `Xs` is chosen from `list(int())`, and for the property to fail then `X` must occur in `Xs` not just once, but *twice*. The probability of finding a random integer twice in a randomly generated list is not very high!

A consequence of using QuickCheck is that we no longer see the generated test data. In fact, we do not want to see it, because QuickCheck generates many test cases. As a result, we may be tricked into a false sense of security by a large number of passing tests, but fail to notice that the distribution is badly skewed. QuickCheck offers the possibility to *measure* the probabilities of different kinds of test data. We can do so by instrumenting a

QuickCheck property to collect statistics during testing. In the case of `prop_delete()` the generator is quite simple and was relatively straightforward to see what the source of the poor data distribution was. However, in more advanced test scenarios, such as QuickCheck state machine models, it not always that easy. Although gathering statistics about the test data is good practice and shall always be done, would it not be more convenient if we had a tool that can directly answer the question if a property can generate the unit test cases we have in mind?

The tool we are going to present in this paper does exactly this. Giving a QuickCheck (finite) state machine model property and a unit test, it will tell you whether the unit test will be generated by the property. For the `delete` example, our tool will say no to the unit test `(3, [1,2,3,1,3])`, a direct warning to the tester that the property (more specifically the test data generator `{int(), list(int())}`) needs to be improved.

A First Example As a teaser, we demonstrate our tool at work with the `delete` example. Since the focus is the more general state machine properties, we transform the simple property `prop_delete()` by wrapping it up in a minimal state machine:

```
initial_state() ->
  not_used.

delete_args(_S) ->
  [int(), list(int())].

delete(X, Xs) ->
  lists:delete(X, Xs).

delete_post(_S, [X, _], Res) ->
  not(lists:member(X, Res)).

prop_delete() ->
  ?FORALL(Cmds, commands(?MODULE), prop_delete(Cmds)).
prop_delete(Cmds) ->
  {H,S,Res} = run_commands(?MODULE, Cmds),
  pretty_commands(?MODULE, Cmds, {H,S,Res}, Res == ok).
```

We will explain the details of state machines in Section 2. For now, it is sufficient to know that we model the same property as previously: the call to `delete` is given the same input data (defined in the `delete_args` function), and checks the same condition (defined in the `delete_post` function).

To run our tool we first create a unit test `U`, which calls the `delete` function with the chosen arguments. The annotations on the arguments, such as `{say, 3}`, state that the actual values are not important, as long as they are the same as the other occurrences. As a result, logically equivalent unit tests such as `(3, [1,2,3,1,3])` and `(30, [10,5,30,10,30])` are considered the same. The result of the `delete` call is bound to `{var,1}` and is checked against the expected outcome. Note that the actual number of the variable, `1` in this case, does not matter.

```
2> U=[{set,{var,1},
      {call,delete,delete,
        [{say,3},[{say,1},{say,2},3,1,3]]},
      {var,1} == [1,2,1]}].
3> possible:possible(delete, U).
Cannot generate
  V1 = lists:delete({say,3}, [{say,1},{say,2},3,1,3])
false
```

The function `possible:possible/2` applies to a state machine module (`delete` in this case) and a unit test. The execution result above correctly predicts that `U` or any of its equivalent variants cannot be generated by the property.

On the other hand, the other unit test we had in mind, testing the scenario that a list remains unchanged when the number for deletion is not present in the list, can be generated by the property as predicted by our tool:

```
4> U=[{set,{var,1},
      {call,delete,delete,
        [{say, 1}, [{say, 2}, {say, 3}, {say, 4}]]},
      {var,1} == [2, 3, 4]}].
5> possible:possible(delete, U).
.........................................
OK, passed 1000 tests
true
```

Note that the annotations to the numbers are essential; otherwise our tool will report false since the probability for generating exactly these numbers is very low.

In the sequel, we explain in detail the design of our tool (Section 3) and its application to large scale industrial AUTOSAR models (Section 4), before concluding in Section 5.

2. Background: State Machine Specification

The original Haskell QuickCheck [6] has inspired different versions for many programming languages, such as Java [10] and ML [11]. We use the QuviQ QuickCheck implementation for the Erlang programming language in this paper. Quviq QuickCheck [1, 9], further referred to as QuickCheck in this paper, is a commercial version of QuickCheck with a rich set of additional libraries. One of these libraries is the *eqc_statem* [1] library for testing software against a state machine model. This library generates and tests random sequences of API calls to the software under test that are well-formed according to the model. In addition to the `eqc_statem` QuviQ QuickCheck also offers a library for *finite* state machine, called `eqc_fsm`.

In this section we explain the QuickCheck finite state machine facility with an example of *bounded queue* implementation. The implementation offers four operations:

- `Q = new(N)`, creates and returns a new queue of capacity `N`,
- `put(Q, X)`, puts `X` into the queue,
- `get(Q)`, removes and returns an element from the queue,
- `size(Q)`, returns the number of elements stored in the queue.

The state of the system consists of the elements in the queue and the maximum size of its buffer. We use an Erlang record with three fields to represent this state:

```
-record(state, {queue, size, contents = []}).
```

The `queue` field is a pointer to the actual queue. The elements that are stored in the queue are modelled as a list in the `contents` field. When defining a finite state machine model it is mandatory to specify the *initial state*, the initial state data, and a state transition function for each state. We define these as follows:

```
initial_state() ->
  init.

initial_state_data() ->
  #state{}.

init() ->
  [{created, new}].

created() ->
```

```
[{created, put},
 {created, get},
 {created, size}].
```

We start in the `init` state with an empty state record. The only way to go the `created` state is by calling the `new` operation, and in this state we can call the other operations. Thus we ensure that we perform `new` before the other operations.

For each operation we specify how to generate random *arguments* for it and we define a *precondition*, specifying when the operation can be part of the test case. Given the present state and the result of the operation, we specify the *next state* and the *postcondition* that should hold after the operation has been completed. These are specified as *callback* functions and share the same name as the operation but have a suffix indicating their purpose. All callbacks, with the exception of `_args`, have a default implementation. By omitting a callback we use this default implementation. The default for a pre- or post-condition is `true`, and for the next state is the current state.

For the `new` operation we give the following specification (we use a grouped style specification, where each API function is specified separately as opposed to defining single functions for the preconditions, next state, etc.):

```
new_args(_, _, _S) ->
  [choose(1, 10)].

new(Size) ->
  q:new(Size).

new_next(_, _, S, Q, [Size]) ->
  S#state{queue = Q, size = Size, contents = []}.
```

The `new` function is called in the module `q` (the software under test) with a size between one and ten. The size is generated by the argument generator `new_args`. Since `new` has arity 1, `new_args` generates a list with one argument. The state is updated by the `new_next` function and stores the pointer to the queue and the generated size in the state record. The `new` call should always return a valid pointer to the queue but we do not check this in a postcondition. Note that the first two arguments of the callback functions are the 'from' and 'to' state, which we do not use in this model.

We continue with the `put` operation, which is specified as follows:

```
put_args(_, _, S) ->
  [S#state.queue, int()].

put_pre(_, _, S, _) ->
  S#state.size > length(S#state.contents).

put(Q, X) ->
  q:put(Q, X).

put_next(_, _, S, _, [_, X]) ->
  S#state{contents = S#state.contents ++ [X]}.
```

The `put_args` function generates two arguments: the pointer to the queue and a random integer to put into the queue. The precondition for putting an element in the queue is that there is at least one more free space left. Note that a precondition is checked during *test case generation*. A precondition on the generation of test cases depends only on the state of the model and not on the actual execution of the software under test. The state is updated by adding the generated element to the contents of the queue.

Specifying the `get` operation is straightforward:

```
get_args(_, _, S) ->
  [S#state.queue].

get_pre(_, _, S, _) ->
  S#state.contents /= [].

get() ->
  q:get().

get_next(_, _, S, _, _) ->
  S#state{contents = tl(S#state.contents)}.

get_post(_, _, S, _, Res) ->
  eq(Res, hd(S#state.contents)).
```

The precondition ensures that the queue is not empty. We can therefore safely use the partial functions `hd` and `tl`, since we know the contents list is not empty. We specify in the postcondition that the result of `get()` should match the first element of the queue, and remove the element from the contents in the next state function. The model state `S` used in the postcondition is the state *before* the action has taken place.

We define the final operation `size` as follows:

```
size_args(_, _, S) ->
  [S#state.queue].

size() ->
  q:size().

size_post(_, _, S, _, Res) ->
  eq(Res, length(S#state.contents)).
```

We have now completed the specification of all operations in the state machine model. To validate the software under test using this model we define the following QuickCheck property:

```
prop_queue() ->
  ?FORALL(Cmds, commands(?MODULE),
  begin
    {H, S, Res} = run_commands(?MODULE, Cmds),
    pretty_commands(?MODULE, Cmds, {H, S, Res},
                    Res == ok)
  end).
```

The above property is nearly identical to the example property in Section 1 and states that any sequence of operations that satisfies the preconditions, will satisfy the postconditions. For each test of this property a random buffer size between 1 and 10 is created, and a random sequence of put, get, and size calls on a queue of that size is generated. Then a fresh buffer is created (taking care to shut down any previously running buffer) and the operation sequence is executed (by `run_commands`). This produces a result and a trace of the execution. If the result is `ok` the property passes, otherwise the property fails and the buffer size and execution trace is printed.

3. Have I tested this?

If we already have an idea what a property should test, we would like to check that idea by writing some test cases and seeing if the property captures them. Or perhaps we already have a test suite, and we would like to see which of our test cases the property captures. This is hard, error-prone work by hand. We have developed a technique that can say with high probability whether a property captures a given test case. The tool is applicable to QuickCheck

(finite) state machine model properties. We check both that the property can generate the sequence of commands in a given unit test case and that the property checks the same assertions as the test case. We start by describing how to check if a property subsumes a given test case.

Our technique deals exclusively with (finite) state machine specifications. A state machine specification can be considered to consist of two parts: a test case *generator* and an *oracle*. There are three steps in running a property:

1. Generate a test case, i.e. a random sequence of commands. For our queue example a simple test case might be:

```
[{set,{var,1},{call,q,new,[3]}},
 {set,{var,2},{call,q,put,[{var,1},2]}},
 {set,{var,4},{call,q,size,[{var,1}]}},
 {set,{var,5},{call,q,get,[{var,1}]}}]
```

   In the remainder of this section we will use a short hand notation, which is a bit more easy to read:

```
Q = q:new(3)
q:put(Q, 2)
q:size(Q)
q:get(Q)
```

2. Execute the test case, and record the result of each function call. In the example above, the result of `new` is a queue of some kind, the result of `put` is simply `ok`, `size` returns `1`, and the result of `get` is 2.
3. Check the result of the test case. The oracle is given the test case and its result and returns either "OK" or "not OK"; in the example above, it would check that `size` returned `1` and that the return value of `get` was 2.

Following this pattern, we split our problem into two parts. First, given a property and a test case, can the property's generator produce the sequence of calls in the test case? Secondly, does the property's oracle check at least as much as the assertions in the test case? We will describe both parts in turn.

3.1 Testing with State Machines

We first need to describe state machines in more detail than the generator/oracle view. A (finite) state machine specification models the system under test as manipulating some sort of *abstract state*; executing a command conceptually modifies that state. The set of allowed commands, and the expected behaviour of those commands, can vary based on the current state. The specification consists of five parts:

- An *initial state* that the system starts in.
- A *command generator* that randomly picks the next command to execute. It is given access to the current state.
- A *precondition* function that takes the current state and a command and checks whether we are allowed to execute the command.
- A *postcondition* function that takes the current state, a command and its result and checks whether the result was correct.
- A *state transition* function that takes the current state, a command and its result and computes the new state.

For example, if we are specifying the queue example, and supposing for simplicity that we remove `new` and only model one unbounded queue, the abstract state will be the list of elements contained in the queue. The rest of the specification is as follows:

- The initial state is the empty list.
- The command generator randomly picks one of `put` (with a random argument) and `get`.
- The precondition for `get` checks that the abstract state is not the empty list; for `put` the precondition always returns true.
- The postcondition for `get` checks that the returned value was the first element in the abstract state; for `put` it checks that the returned value is the atom `ok`.
- The state transition function for `get` removes the first element from the abstract state; for `put` it appends the new element to the end of the abstract state.

It is quite easy to see how to generate and execute a test case in this setting: start in the initial state, apply the command generator to get a command, check if the precondition holds, if not, try the command generator again. Once we have a command that satisfies the precondition, run it, record its result, check that the postcondition holds. Finally, use the state transition function to compute a new state and repeat.

One complication when generating test cases is that executing a command can bind a variable, which we refer to later. Look at `Q` in our example test case:

```
Q = q:new(3)
q:put(Q, 1)
q:get(Q)
```

In order to produce test cases with variables in, we cannot use the simple generation and execution method described above. Rather, QuickCheck separates generation from execution. During test case generation, QuickCheck behaves differently from our description in two ways:

1. It does not check postconditions.
2. When calling the state transition function, instead of passing it the actual result of the command (which has not been run yet), it generates a fresh *symbolic variable* and passes that instead. These symbolic variables may make their way into the state, from where they might be selected to appear in subsequent commands.

The final stage is to execute the symbolic test case. This works exactly as our naive algorithm above, except that it is given a particular test case to execute instead generating commands at random. During execution, the state will not contain symbolic variables but only concrete values.

3.2 Checking the Sequence of Calls

We are now in a position to describe how to check if a state machine property can generate a particular test case. Our idea is simple. Instead of generating a random command each time, we have a target command that must appear next, and we *check* if the command generator can generate the target command in the current state. Other than that, we run the test case generation algorithm as in normal QuickCheck. Once we have established that we can generate the target command in the current state, we can use the state transition function as normal to compute the next state, so this technique scales linearly with the number of commands in the test case. If we can't generate the target command, then we stop and complain that we couldn't generate the test case as a whole.

But how to check if we can generate the target command? It seems that we have replaced one problem—"can the test case generator generate this test case?"—with almost the same one: "can the command generator generate this command?" To solve this

new problem, we use brute force: generate ten thousand random commands, and see if any of them is the target command! If so, the generator could generate the command; if not, it couldn't.

The reader may be horrified by this! If we fail to generate the target command in 10000 attempts, we assert that we can't generate it at all. How can this blind brute force be reasonable or principled?

We claim that it is. Here is why. If we fail to generate the target command in 10000 attempts, the probability of generating it at random must be low, and so during normal testing, we would be unlikely to come up with this command too—let alone the rest of the commands in the test case. We are therefore justified in saying that it is *very unlikely* that the property generates the test case—and if we are very unlikely to test it, we perhaps shouldn't claim that we can test it.

Just how unlikely is it that we can generate a particular command, if we failed to find it in 10000 attempts? We can compute this. Letting p be the probability that the command generator can generate our command, and supposing we want to conclude something about p with 99% certainty, we reason that

$$\begin{aligned} & (1-p)^{10000} < 0.01 \\ \Rightarrow \quad & 1-p < 0.01^{1/10000} \\ \Rightarrow \quad & p > 1 - 0.01^{1/10000} \end{aligned}$$

which comes out to $p > 0.9995$, i.e., we can say with 99% certainty that our command generator picks this command less than $1/2000$ of the time, starting from the current state—let alone generating the rest of the test case. Thus any test case we reject is not going to be tested by our property in practice.

Why is it OK to use brute force for testing single commands, but not whole test cases? In other words, why couldn't we just generate 10000 random test cases and see if any of them were the one we are looking for? Clearly, this wouldn't work: the problem is that the number of possible test cases grows exponentially with the length of the test case. Our technique rests on the assumption, which seems to be true in most QuickCheck models, that the number of possible individual commands is normally quite low.

3.3 Small Variations

Sticking with our queue example, a common technique to obtain good random tests is to generate data values from a small domain. For example, we might decide to only `put` values in the range 0 to 10. We can express this when writing our generator for values:

```
value() ->
  choose(0, 10).
```

Now, imagine that our test case reads:

```
Q = q:new(10)
q:put(Q, 100)
q:put(Q, 200)
q:size(Q) -> 2
q:get(Q)  -> 100
q:get(Q)  -> 200
```

Our algorithm will claim that our property can *not* generate this test case. This is true, strictly speaking! But we can generate a very similar test case:

```
Q = q:new(10)
q:put(Q, 1)
q:put(Q, 2)
q:size(Q) -> 2
q:get(Q)  -> 1
q:get(Q)  -> 2
```

We might like to consider these two test cases to be equivalent, even though they are syntactically different. It's not safe to do this automatically, because it requires domain knowledge, so instead the user must help us by *annotating* the unit test. We introduce the function `say` to mark irrelevant values

```
Q = q:new(10)
q:put(Q, say(100))
q:put(Q, say(200))
q:size(Q) -> 2
q:get(Q)  -> 100
q:get(Q)  -> 200
```

By writing `say`, the user specifies that the particular *values* we put into the queue are not important, only the relationship between the various values (which ones are equal and which are not equal). We will then accept the generated `1/2` test case by comparing it with the annotated `100/200` test case, because 1 "might as well" be 100 while 2 "might as well" be 200. Note that we should not annotate the `2` returned by `size`, because this really needs to be exactly `2` since we have put two elements in the queue.

In more detail, we look for any annotated values in a command from the unit test case. We treat these annotated values as symbolic variables and first-order match the generated command against the generated command from the command generator. If this succeeds, it yields a substitution. We then make sure, if we see the same value in a later command, to apply the substitution to it. In our example, after we have matched the `q:put(Q, say(100))` with `q:put(Q, 1)`, whenever we later on match a command with the value `100` we will replace it with `1`.

We only allow to annotate a particular value once. If we happen to have a unit test with two equal values, which do not necessarily need to be the same, we have to make them distinct. This is unfortunate and we may improve this in the future. We could, for example, extend the value annotation with a variable name, which can be used in the rest of the unit test.

A final note. In our example we only generate values in the range $[0 \ldots 10]$. This means that if the test case considers, say, 20 distinct values, we will say that the property cannot test it. This seems like a limitation—but is in fact honest, since our property can't generate tests with 20 distinct values! Our implementation will report an error as soon as the 12th value appears; the user can take this as a hint to alter the property.

3.4 Checking Assertions

A test case is not just a sequence of commands: it can also contain *assertions*. We separate the assertions from the commands: the commands themselves do not check anything (except implicitly that there are no runtime exceptions), but we can attach an assertion that checks the result of the command. For our example one test case would be

```
Q = q:new(10)
q:put(Q, say(100))
q:put(Q, say(200))
X = q:size(Q), assert(X == 2)
Y = q:get(Q),  assert(Y == 100)
Z = q:get(Q),  assert(Z == 200)
```

We would like to check that, when our property generates this test case, it also checks the assertions. Formally, the postcondition of

each command should imply the assertion for that command, or: *when the assertion fails the postcondition is false*.

We check this property by simply using QuickCheck. After we have generated a particular prefix of the test case, we will have a particular symbolic state—a state containing symbolic variables: our property is that, whichever concrete values we choose for the variables in the state and test case, if the assertion fails for those variables then so does the postcondition.

Currently, the user must define a generator that gives reasonable values that each command might conceivably return—reasonable in the sense of being well-typed, not necessarily correct. When generating random values for the variables we pick from this set.

3.5 An Example

We will briefly describe the behaviour of our tool on four variants of an example: one that is accepted by the property, one where the assertion is not checked, one where a precondition fails, and one which is not generated at all.

```
Q = q:new(10)
q:put(Q, say(15))
q:put(Q, say(23))
V = q:size(Q), assert(V == 2)
q:put(Q, say(8))
W = q:get(Q),  assert(W == 15)
q:put(Q, say(11))
X = q:get(Q),  assert(X == 23)
Y = q:get(Q),  assert(Y == 8)
Z = q:get(Q),  assert(Z == 11)
```

This example should be generated by our property, and indeed, our tool confirms this and discharges the assertions instantly. Suppose now that we remove the call `q:put(Q, say(11))`. The final `get` should no longer be allowed, because the queue is empty. Our tool complains that it can't generate the final `get`—the postcondition is false. Now suppose instead that we change one of the assertions—perhaps we assert that `Y` should be 11. The tool then complains that the postcondition might succeed when the assertion does not, and gives a counterexample—what if `Y` is 8?

All three examples are checked instantly. As a final example, suppose that we insert a call to a completely different function, such as `erlang:spawn`, in the the test case. Our tool reports that it cannot generate the call to `spawn`. In this case, it takes a second or so before it gives up.

Even though our technique relies on brute force to some extend, it scales up to arbitrarily long test cases. We continue with the evaluation of it in a more demanding setting: Quviq's models for testing AUTOSAR basic software components.

4. Case study: AUTOSAR Unit Tests

AUTOSAR is a software standard developed by the automotive industry, to lower costs by enabling manufacturers to buy off-the-shelf components [3]. The AUTOSAR basic software, which is part of the operating system, contains a number of protocol stacks, such as CAN (Controller Area Network) and Ethernet, a communication and routing module, network management, and diagnostics. Each module is specified by a document, typically a few hundred pages, describing a C-API and hundreds of requirements. In this section we focus on the CAN stack.

QuviQ developed QuickCheck state machine models [13] for each module, which follow the specification as close as possible. These models can be regarded as executable formal specifications. These models are used for acceptance tests for vendor implementations of AUTOSAR modules. Each model can be used on in isolation to generate random test cases for the module it specifies. An AUTOSAR module often depends on and makes calls to other AUTOSAR modules. These external modules are *mocked* [14]; the models include a specification of which (mocked) calls to other modules are to be expected, and what results they should return. In addition to testing a model on its own, it is also possible to combine models together in a cluster. The matching mocked calls from one module to API calls of another are constructed automatically. The CAN stack is specified as a cluster of modules, including for example CanIf and CanSm.

The AUTOSAR consortium developed acceptance tests as well, including six tests for the CAN stack [4]. We encoded these test cases in our format, and before the current tool is developed, we had already tried manually to validate them against the model. This validation is hard-coded and was specifically implemented for these six tests. Our effort showed that three of the six tests agreed with the model, and the other 3 were not accepted and turned out to be incorrect [2]. We have since corrected the mistakes. Of course, we would like to validate whether our tool is able to save us the effort and perform this validation automatically.

We then extended our tool such that it can handle QuickCheck component models and clusters, which are somewhat different from the normal QuickCheck state machine models. Before running our tool on the CAN cluster and the six acceptance tests, we first needed to annotate the unit tests. For example, the unit tests contain commands that transmit random payloads of binary data. To give an impression of the size of the AUTOSAR unit tests we show one of the six tests:

```
[{call,canif_spec,init,[],[]},
 {call,cansm_403_spec,init,[],[]},
 {call,cansm_403_spec,main,[],[]},
 {call,canif_spec,controller_mode_indication,
       ['CanIfConf_CanIfCtrlCfg_ECU2COMCan12',
        'CANIF_CS_STOPPED'],
       []},
 {call,cansm_403_spec,main,[],[]},
 {call,canif_spec,controller_mode_indication,
       ['CanIfConf_CanIfCtrlCfg_ECU2COMCan12',
        'CANIF_CS_SLEEP'],
       []},
 {call,cansm_403_spec,main,[],[]},
 {call,canif_spec,trcv_mode_indication,
       ['CanIfConf_CanIfTrcvCfg_CanIfTrcvCfg_0',
        'CANTRCV_TRCVMODE_NORMAL'],
       []},
 {call,cansm_403_spec,main,[],[]},
 {call,canif_spec,trcv_mode_indication,
       ['CanIfConf_CanIfTrcvCfg_CanIfTrcvCfg_0',
        'CANTRCV_TRCVMODE_STANDBY'],
       []},
 {call,cansm_403_spec,main,[],[]},
 {call,cansm_403_spec,request_com_mode,
       ['CanSMConf_CanSMManagerNetwork_CanNetwork_0',
        'COMM_FULL_COMMUNICATION'],
       []},
 {call,cansm_403_spec,main,[],[]},
 {call,canif_spec,trcv_mode_indication,
       ['CanIfConf_CanIfTrcvCfg_CanIfTrcvCfg_0',
        'CANTRCV_TRCVMODE_NORMAL'],
       []},
 {call,cansm_403_spec,main,[],[]},
 {call,canif_spec,controller_mode_indication,
       ['CanIfConf_CanIfCtrlCfg_ECU2COMCan12',
        'CANIF_CS_STOPPED'],
       []},
 {call,cansm_403_spec,main,[],[]},
 {call,canif_spec,controller_mode_indication,
       ['CanIfConf_CanIfCtrlCfg_ECU2COMCan12',
        'CANIF_CS_STARTED'],
```

```
      []},
{call,cansm_403_spec,main,[],[]},
{call,canif_spec,transmit,
      ['CanIfConf_CanIfTxPduCfg_P02',
       <<70,151,71,255,9,83>>],
      []},
{call,canif_spec,tx_confirmation,
      ['CanIfConf_CanIfTxPduCfg_P02'],
      []},
{call,cansm_403_spec,many_main,"",[]},
{call,canif_spec,controller_busoff,
      ['CanIfConf_CanIfCtrlCfg_ECU2COMCan12'],
      []},
{call,cansm_403_spec,main,[],[]},
{call,canif_spec,controller_mode_indication,
      ['CanIfConf_CanIfCtrlCfg_ECU2COMCan12',
       'CANIF_CS_STARTED'],
      []},
{call,cansm_403_spec,many_main,"d",[]},
{call,canif_spec,transmit,
      ['CanIfConf_CanIfTxPduCfg_P02',
       <<70,151,71,255,9,83>>],
      []},
{call,canif_spec,tx_confirmation,
      ['CanIfConf_CanIfTxPduCfg_P02'],
      []},
{call,cansm_403_spec,get_current_com_mode,
      ['CanSMConf_CanSMManagerNetwork_CanNetwork_0'],
      []},
{call,cansm_403_spec,many_main,"",[]},
{call,cansm_403_spec,get_current_com_mode,
      ['CanSMConf_CanSMManagerNetwork_CanNetwork_0'],
      []},
{call,canif_spec,transmit,
      ['CanIfConf_CanIfTxPduCfg_P02',
       <<70,151,71,255,9,83>>],
      []},
{call,canif_spec,tx_confirmation,
      ['CanIfConf_CanIfTxPduCfg_P02'],
      []}]
```

Using our tool to determine if the six tests can be generated by the CAN cluster showed a surprising result. Only five out of the six "correct" tests could be generated! After careful inspection with a CAN cluster expert it turned out that a small error slipped into that particular unit test. This is exactly what our tool aims to do: give insight into the model/property and increase our confidence that we are testing the right thing.

5. Related work and Conclusions

Most of the research that concerns the relationship between properties and unit tests is in the area of automated test generation. QuickCheck properties typically serve the dual role of templates for generated test inputs and oracles for verifying correctness. This can be seen as a specific instance of the more general concept of model-based testing [12], where test inputs may be derived from an abstract model of the system under test, and if being executable the models may double as the oracles.

For more complex test inputs, especially custom data structures with invariants, specialised generators are still needed to avoid inefficiency (i.e., non-invariant compliant) and redundancy (i.e., poor distribution). QuickCheck provides a domain-specific language for defining specialised generators, and techniques in constraint satisfaction domain may be used [5, 7]. For deriving uniformly distributed test inputs, there exist automatic generation methods based on clever enumeration for arbitrary datatypes [8]. However, such enumeration-based techniques are stand-alone and not linked to properties.

None of the techniques mentioned above is able to connect individual tests to the properties that supposedly produce them. To answer the question "Have I tested this?", one has no choice but to manually study the very often complicated properties and generation methods. As far as we are aware, our work is the first to establish this missing link through an automated system. Giving a property and a unit test, our tool is able to tell whether the particular unit test or an equivalent variant will be generated by the property with a reasonable probability. The scalability of our system is demonstrated with a case study of testing the AUTOSAR basic software modules.

In the future, we would like to look into more precise control over acceptable deviations from the test case. At the moment, we require that annotated values are distinct, but for some applications we might need more than that: for example, in testing a binary search tree, we may want the annotated value `1` matches to be less than the annotated value `2` matches, since `1 < 2`. Moreover, our assertion-checking mechanism is quite simplistic at the moment, though it has worked just fine so far. We can imagine checking the assertions more smartly than pure random testing. One possibility is to use *symbolic reasoning* on the postcondition and the assertion: this at least should give us a hint about what sort of random data to test with.

Acknowledgements We acknowledge the European Commission for its support of the Prowess project, Framework 7 project 317820. The authors would like to thank Hans Svensson, Thomas Arts, and Ulf Norell for their comments and help.

References

[1] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with Quviq QuickCheck. In *Erlang Workshop*, pages 2–10, 2006.

[2] T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–4, April 2015.

[3] AUTOSAR Consortium. AUTomotive Open System ARchitecture, standard documents. `https://autosar.org/`, 2013.

[4] AUTOSAR Consortium. Acceptance Test Specification of Communication on CAN bus - Release 1.0.0, July 2014.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[6] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[7] K. Claessen, J. Duregård, and M. Palka. Generating Constrained RandomData withUniformDistribution. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 18–34. Springer International Publishing, 2014.

[8] J. Duregård, P. Jansson, and M. Wang. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 61–72, New York, NY, USA, 2012. ACM.

[9] J. Hughes. QuickCheck Testing for Fun and Profit. In M. Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, 2007.

[10] T. Jung. Java implementation of QuickCheck. `http://quickcheck.dev.java.net/`, 2010.

[11] C. League. QCheck/SML. `http://contrapunctus.net/league/haques/qcheck/`, 2010.

[12] J. Rushby. Verified Software: Theories, Tools, Experiments. chapter Automated Test Generation and Verified Software, pages 161–172. Springer-Verlag, Berlin, Heidelberg, 2008.

[13] Svenningsson, Johansson, Arts, Norell, Svenningsson, and Svensson. Testing AUTOSAR software components with QuickCheck. In *Proceedings of IXe Conf. on AMCTM*. SP, Sweden, 2011.

[14] J. Svenningsson, H. Svensson, N. Smallbone, T. Arts, U. Norell, and J. Hughes. An Expressive Semantics of Mocking. In S. Gnesi and A. Rensink, editors, *Fundamental Approaches to Software Engineering*, volume 8411 of *LNCS*, pages 385–399. Springer Berlin Heidelberg, 2014.