2016 IEEE International Conference on Software Testing, Verification and Validation

How well are your requirements tested?

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

John Hughes
Chalmers University and Quviq AB
Göteborg, Sweden
Email: rjmh@chalmers.se

***Abstract*—We address the question: to what extent does covering requirements ensure that a test suite is effective at revealing faults? To answer it, we generate *minimal* test suites that cover all requirements, and assess the tests they contain. They turn out to be very poor—ultimately because the notion of *covering a requirement* is more subtle than it appears to be at first. We propose several improvements to requirements tracking during testing, which enable us to generate minimal test suites close to what a human developer would write. However, there remains a class of plausible bugs which such suites are very poor at finding, but which random testing finds rather easily.**

I. Introduction

It is common practice to base test cases on requirements. For example, DO-178C [1] (the primary document on which certification of commercial avionics software is based) insists that all test cases be traceable to requirements, and that test cases exist for each requirement. Now, DO-178C does insist on other forms of coverage also—structural code coverage and coverage of control and data coupling. Yet the emphasis on requirements coverage raises the question: *to what extent does covering requirements ensure that test suites will reveal faults?*

In this paper, we study this question, first in the context of a small but illustrative example, and then in the (much larger) context of an AUTOSAR Basic Software component (part of the standardised software that runs in cars). To eliminate human bias, we *generate* requirements-based test suites, from models annotated to track the requirements that each test case covers. Our test suites are *minimal*—they do no more than necessary to cover each requirement. We assess the generated test suites by inspection, comparing each test to a test case that a human developer might write, and also by comparison with random testing.

The results turn out to depend crucially on what it *means* to cover a requirement. We start from informal requirements in natural language, formalize them in our model, and then formalize the notion of *covering* a requirement in our requirements-tracking code. We will see that this latter notion is much subtler than it first appears to be, especially in the context of black-box testing where the internal state of the SUT is not observable. This leads us to refine our requirements tracking, resulting in much better generated minimal test suites. Even so, we identify a class of plausible bugs, appearing both in the illustrative example and in the AUTOSAR component, which our generated minimal test suites cannot find. We argue that our final generated suites are close to what a diligent developer would write—and thus our conclusions may apply not only to generated test suites, but also to those written by hand with requirements coverage as the primary goal.

In section II we introduce our motivating example, and explain how we model it and how we track its requirements. We generate a minimal test suite that covers all the requirements, which is surprisingly poor, and go on to introduce a number of refinements to requirements tracking that improve the generated minimal suite greatly. The final result—we claim—is a generated test suite that closely resembles what a diligent human developer would write. Nevertheless, we suggest a class of plausible bugs which this kind of test suite cannot find, but random testing finds easily. In section III we discuss a much larger example, the AUTOSAR COM stack, for which we had a model available, already annotated to track requirements covered in each test. We generated minimal test suites with full requirements coverage, and found that the lessons from the smaller example apply here too. However, we also identified several other improvements to requirements tracking, which are necessary to generate a good test suite for this more complex example. We also found that the same category of plausible bug is *not found* by a requirements-based test suite in this setting either—but is easily found by random testing.

II. A Simple Example: the Process Registry

The *process registry* is a heavily used part of the Erlang virtual machine, which maintains a mapping between names and Erlang process identifiers ("pids") so that Erlang programs can find named services on their local node. The API is simple, and we consider just the three most important functions:

- `register(Name,Pid)`, which adds the given process identifier ("pid") to the registry with the given name,
- `unregister(Name)`, which removes the given name and its associated registered process from the registry,
- `whereis(Name)`, which returns the pid currently associated with the given name in the registry.

The registry maintains the following invariants:

- Each name may appear at most once in the registry
- Each pid may appear at most once in the registry
- Every process in the registry is alive

To preserve the invariants, calls of `register` that would lead to an invalid state raise an exception and do nothing instead, and processes that die while in the registry are removed from it automatically. More precisely, the API should fulfill the requirements in Figure 1. Each of these requirements specifies

978-1-5090-1827-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICST.2016.23

REG001	When register is called with a name and pid, neither already in the registry, then it should add the pair to the registry and return true.
REG002	When register is called with a name that is already in the registry, then it should raise an exception and leave the registry unchanged.
REG003	When register is called with a pid that is already in the registry, then it should raise an exception and leave the registry unchanged.
REG004	When register is called with a pid that is dead, then it should raise an exception and leave the registry unchanged.
UNR001	When unregister is called with a name that is in the registry, then it should remove the name and its associated pid from the registry, and return true.
UNR002	When unregister is called with a name that is not in the registry, then it should raise an exception and leave the registry unchanged.
WHE001	When whereis is called with a name that is in the registry, then it should return the associated pid, and leave the registry unchanged.
WHE002	When whereis is called with a name that is not in the registry, then it should return the atom 'undefined', and leave the registry unchanged.
DIE001	When a process dies while its pid is in the registry, then the pid should be removed from the registry.

Fig. 1. Requirements on the process registry.

the behaviour of an API call in one case; since there are nine requirements, we might expect to cover them with nine test cases.

A. Testing the registry with QuickCheck

The testing tool we use in this paper is Quviq QuickCheck [2], a commercial tool based on Haskell QuickCheck [3] which uses a specification to generate, execute, and adjudge random tests. Quviq QuickCheck supports *state machine models*, used to generate test cases consisting of sequences of calls to the SUT [4], [5]. These models define a state transition function for each call, used to track the model state as a test is run, and pre- and postconditions used to determine whether or not a test has failed. Failing tests are *shrunk* to minimal counterexamples and reported to the user[1].

We generate tests for the registry consisting of calls to the three functions under test, together with `spawn` (which creates a new pid, for use as test data), and `kill` (which kills a pid). The model maintains a state containing

- the pids which have been spawned in the current test (and so are available for registering or killing),
- the name–pid pairs which should currently be in the registry, and
- the pids which have been killed.

Tests are generated by repeatedly choosing an operation at random, choosing names from a small predefined set, and choosing pids from the set in the model state. The model defines preconditions to ensure that, for example, we do not try to call `register` if no pids have been spawned (since there is no parameter we could pass as the pid), and postconditions to check that the return value of each operation is correct according to the applicable requirements (or, if a call is expected to fail, that an exception was indeed raised).

From this model we can generate and run random tests such as the following:

```
spawn() -> <0.917.0>
unregister(a) -> {'EXIT',...}
kill(<0.917.0>) -> ok
spawn() -> <0.918.0>
```

This is the form in which QuickCheck reports failing tests, showing calls made and their results (where `<0.917.0>` is a pid, and `{'EXIT',...}` denotes an exception). This is also the form in which we will present test cases in this paper—as a sample run, with the actual values returned. Note that these are not *expected* values—each time we run this test we expect different pids to be spawned. The return values are checked by the postconditions in the model, but these are not shown when we present the test cases.

QuickCheck state machine models support the annotation of each call in a test with a set of "features", which can be any Erlang values—for example, atoms representing the names of the requirements tested by that call. We annotated our model to collect each of the requirements in Figure 1 when a call fulfilling the conditions stated in the requirement is made.

B. Generating a minimal test suite

Given a model annotated with requirements, QuickCheck can generate a minimal test suite that covers every reachable requirement as follows. We start with an empty test suite S, and an empty set of covered requirements R. Then we use QuickCheck to test the property that *no requirements other than those in R are reachable*. This property is false, and QuickCheck reports a minimal counterexample—a minimal test that covers at least one requirement not in R. We add the test case to S, and the newly covered requirements to R, and repeat. Eventually QuickCheck will not be able to cover any new requirements in the time we allow for testing—usually a few minutes—and the process terminates. At this point we have a test suite which covers the same requirements as a very large number of random tests; moreover, we know *why* each test case was included—we know which requirements it tests that were not tested by any previous test.

Applying this method to the registry model we have described results in the test suite in Figure 2. (Note that requirements tested by earlier tests in the table are not mentioned again if they are also covered by a later test). Indeed, this test suite contains nice simple tests for each requirement—for example, test #1 tests that `whereis` returns `undefined` for a name which has not been registered, while test #8 tests that it

[1]QuickCheck never considers a test to be 'inconclusive'. Tests which might be considered inconclusive by other approaches are considered instead to pass—i.e. the test case should not be reported to the user.

	Reqs tested	Test case
1	WHE002	`whereis(a) -> undefined`
2	REG001	`spawn() -> <0.20612.0>` `register(a, <0.20612.0>) -> true`
3	UNR002	`unregister(a) -> {'EXIT',...}`

4	REG004	`spawn() -> <0.20635.0>`
		`kill(<0.20635.0>) -> ok`
		`register(a, <0.20635.0>) ->`
		`{'EXIT',...}`

5	UNR001	`spawn() -> <0.20650.0>`
		`register(a, <0.20650.0>) -> true`
		`unregister(a) -> true`

6	DIE001	`spawn() -> <0.20665.0>`
		`register(a, <0.20665.0>) -> true`
		`kill(<0.20665.0>) -> ok`

7	REG002, REG003	`spawn() -> <0.20680.0>`
		`register(a, <0.20680.0>) -> true`
		`register(a, <0.20680.0>) ->`
		`{'EXIT',...}`

8	WHE001	`spawn() -> <0.20695.0>`
		`register(a, <0.20695.0>) -> true`
		`whereis(a) -> <0.20695.0>`

Fig. 2. Requirements-based test suite for the registry.

returns the correct pid for a name which *has* been registered. (Recall that the correctness of the result is checked by the postcondition in the model, not shown here).

C. Requirements in combination

Most test cases in the table test one requirement, and simply set up the scenario that the requirement describes. However, one test case stands out—test #7 tests *two* requirements in one test! reg002 specifies that `register` should raise an exception if its name parameter is already registered, while `REG003` specifies that `register` should raise an exception if its pid parameter is already registered. Test case #7 cleverly tests both requirements at the same time, by passing *both* a name that is already registered, and a pid that is already registered!

But is this really a good test of *either* requirement? Both requirements demand the same behaviour—that `register` raise an exception—so, in fact, if the implementation of the registry fulfills either of these requirements, then the test case will exhibit this behavior. It follows that the other requirement might not be satisfied in general. Since we can't know, in this case, which of the two requirements is actually satisfied, then this test case actually tests neither requirement properly.

We can see from this example that, when requirements overlap, then it is not necessarily enough to test each requirement at least once. We might wish to test each requirement in isolation—or in different combinations. Test #7 *is*, after all, an interesting test case, because its behavior is constrained by two independent requirements, and the test does at least demonstrate that both can be fulfilled simultaneously.

Luckily, we can easily adapt our model to distinguish between different *combinations* of requirements. The adaptation consists of collecting *lists* of requirements as features of calls

	Reqs tested	Test case

9	REG003	`spawn() -> <0.23653.0>`
		`register(a, <0.23653.0>) -> true`
		`register(b, <0.23653.0>) ->`
		`{'EXIT',...}`

10	REG002	`spawn() -> <0.23668.0>`
		`spawn() -> <0.23669.0>`
		`register(c, <0.23669.0>) -> true`
		`register(c, <0.23668.0>) ->`
		`{'EXIT',...}`

11	REG002, REG004	`spawn() -> <0.23686.0>`
		`spawn() -> <0.23687.0>`
		`kill(<0.23686.0>) -> ok`
		`register(a, <0.23687.0>) -> true`
		`register(a, <0.23686.0>) ->`
		`{'EXIT',...}`

Fig. 3. Tests for combinations of requirements.

of `register`, rather than individual requirements. Instead of associating *both* features `REG002` and `REG003` with the second call of `register` in test case #7, we associate the *single* feature `[\req{REG002},\req{REG003}]`. When we do so, and regenerate our minimal test suite, we obtain the same eight tests appearing in Figure 2, along with three new tests, shown in Figure 3. As well as the expected tests for `REG002` and `REG003` separately, this new test suite also includes a test of `REG002` and `REG004` in combination (#11)—here the last call of `register` fails *both* because the name is already registered, and because the pid to be registered is dead. This test demonstrates the consistency of `REG002` and `REG004`, so it is interesting, but we suspect most developers would overlook this possibility!

D. Observing Requirements on the State

In fact, we still have reason to be dissatisfied with the generated test suite. Consider test case #6,

```
spawn() -> <0.20665.0>
register(a, <0.20665.0>) -> true
kill(<0.20665.0>) -> ok
```

which is supposed to test `DIE001`: "When a process dies while its pid is in the registry, then the pid should be removed from the registry". The generated test case is clearly unsatisfactory: while it does, indeed, invoke kill on a pid that is in the registry, it does *not* check that the pid was removed as a result (because this is not—and cannot be—part of the postcondition of the call)! Thus we cannot really claim that the requirement has been properly tested.

In fact, the same problem appears in many of our generated tests. Consider test #5,

```
spawn() -> <0.20650.0>
register(a, <0.20650.0>) -> true
unregister(a) -> true
```

which is supposed to test `UNR001`: "When `unregister` is called with a name that is in the registry, then it should remove the name and its associated pid from the registry, and return `true`". When this test is run, then the postcondition for

`unregister` does check that the final call returns `true`, but it does *not* check that the name has been removed from the registry. In general, the simple test cases we have generated above only check the return-value part of the requirements, not that the state changes specified have occurred.

A human developer would probably address this by extending test case #6 to check that the name and pid were indeed removed, as in

```
spawn() -> <0.20665.0>
register(a, <0.20665.0>) -> true
kill(<0.20665.0>) -> ok
whereis(a) -> undefined
```

Now the postcondition for `whereis` holds only if the name `a` was removed from the registry by the call to `kill`. The call to `whereis` *observes* the required effect of the `kill`, and the requirement has not really been properly tested until this observation is made. Thus our original approach—to consider a requirement "tested" once a call matching its conditions has been made—was inadequate, and needs to be refined to take observations into account. This is strongly reminiscent of Whalen et als. extension of MCDC to take observation into account [6].

E. Deferred requirements

We extend our model by adding "deferred requirements" to the model state. A deferred requirement is a requirement which *will have* been tested, once a given observation is made. In our model, we simply keep a set of requirement/observation pairs. We defer requirements by adding them to this set, along with a name whose state in the registry needs to be observed to complete the test of the requirement. Note that the decision to defer a requirement is made by the human developer, not a tool, often in response to dissatisfaction with the generated minimal tests.

Specifically, we adapt the model so that

- `REG001` is deferred until the registered name is observed,
- `UNR001` is deferred until the unregistered name is observed,
- `DIE001` is deferred, if the process being killed is in the registry, until the corresponding name is observed.

The other requirements (corresponding to failing calls) have no effect, and so we do not defer them. Names are observed by calling `whereis`—so we define the "features" of a call of `whereis` to include all the requirements deferred until an observation of that name.

Regenerating a minimal test suite results in a suite of only *10* tests, made up of tests presented earlier and three new ones in Figure 4. The second and third tests are just tests #6 and #5 presented above, extended with a final call of `whereis` to verify that the registry was correctly updated. The first test was actually already present in our previous suite (#8), where its purpose was to test a successful call of `whereis`. But of course, it also tests a successful call of `register`, and observes its effect on the registry, so it is now labelled with requirement `REG001` as well.

Reqs tested	Test case

`WHE001`, `REG001`	`spawn() -> <0.29956.1>`
	`register(a, <0.29956.1>) -> true`
	`whereis(a) -> <0.29956.1>`

`DIE001`	`spawn() -> <0.29971.1>`
	`register(a, <0.29971.1>) -> true`
	`kill(<0.29971.1>) -> ok`
	`whereis(a) -> undefined`

`UNR001`	`spawn() -> <0.9814.2>`
	`register(a, <0.9814.2>) -> true`
	`unregister(a) -> true`
	`whereis(a) -> undefined`

Fig. 4. New tests to cover deferred requirements.

F. Generalizing observations

In the previous section we defined deferred requirements, which we considered tested when the change was observed using `whereis`—but there is more than one way to observe the registration state of a name. In fact, not only `whereis`, but also `register` and `unregister` behave differently depending on whether or not the name they are passed is already registered. So we can actually observe the state of a name with a call to one of these functions instead. We extended our model to make the 'observations' made by each call explicit; specifically

- `register(Name,Pid)` observes `Name`,
- `unregister(Name,Pid)` observes `Name`,
- `whereis(Name)` observes `Name`, and
- `kill` and `spawn` observe nothing.

Recall that each deferred requirement is paired in the model state with the name that should be observed to trigger the requirement; to generalise the notion of observation, we just added features to our model so that any call to a function `F`, which observes `Name`, also has the feature `R/F` for each deferred requirement `R` that is paired with the same `Name`. Since we tag each deferred requirement with the *name* of the function which observes it, then our generated test suites will contain tests for *each* way that a deferred requirement can be observed.

With this more refined notion of deferred features, we obtain a suite of 15 tests. The additional tests (in Figure 5) do serve a real purpose: imagine, for example, an implementation of the registry which did not remove dead pids as soon as the corresponding process dies, but only when a call of `whereis` would return a dead pid. This wrong implementation would pass all the tests in the minimal suites generated before this section, but would be caught by two of the 4 new tests added at this stage.

G. The Problem of Opaque State

We believe the suite of 15 test cases we have now generated to be very close to what a (rather diligent!) human programmer might write, if asked to test the nine requirements we stated—in fact, we doubt that most human programmers would go to the trouble to observe each state change using each of `whereis`, `register`, and `unregister`. But we are not yet satisfied! Compare the generated test for `WHE001`:

Reqs tested	Test case

DIE001/ register	`spawn() -> <0.18281.2>`
	`register(b, <0.18281.2>) -> true`
	`kill(<0.18281.2>) -> ok`
	`register(b, <0.18281.2>) ->`
	`{'EXIT',...}`

UNR001/ unregister	`spawn() -> <0.18337.2>`
	`register(b, <0.18337.2>) -> true`
	`unregister(b) -> true`
	`unregister(b) -> {'EXIT',...}`

UNR001/ register	`spawn() -> <0.18365.2>`
	`register(c, <0.18365.2>) -> true`
	`unregister(c) -> true`
	`register(c, <0.18365.2>) -> true`

DIE001/ unregister	`spawn() -> <0.18380.2>`
	`register(d, <0.18380.2>) -> true`
	`kill(<0.18380.2>) -> ok`
	`unregister(d) -> {'EXIT',...}`

Fig. 5. New tests with generalized observations.

```
spawn() -> <0.29956.1>
register(a, <0.29956.1>) -> true
whereis(a) -> <0.29956.1>
```

with the requirement it is intended to test: "When `whereis` is called with a name that is in the registry, then it should return the associated pid, *and leave the registry unchanged*".

It is clear that nothing in the generated test case checks the italicized part of the requirement! In fact, the same objection can be raised to all of the generated test cases: many requirements specify that the registry should be unchanged, but even when a change of state is required, there is an implicit requirement that the rest of the registry state remains *unchanged*. None of the generated test cases check this.

Unfortunately, it is far from clear how test cases *could* check this part of each requirement. The problem is that the state of the registry is opaque, and the API we have available does not allow us to observe it completely. There are infinitely many possible registered names, which implies that no finite test case can observe the entire registry state. Thus the requirements we started with are fundamentally untestable.

Thus we are forced to satisfy ourselves with testing these requirements partially. So far, we counted a deferred requirement as tested if the name whose state it changed was observed by a later operation in the test case. We could also insist that, say, at least one other name has its state observed. Or we could insert operations that call `whereis` on all the five names used in our tests, and check those results, thus observing the part of the registry state that we suspect to be relevant. Or we could require that calls to `register` that fail to modify a name nevertheless are followed by an observation of that name (to verify that no registration occurred), before we consider the associated deferred requirement to be tested. If we generate a test suite using this last idea, we obtain a suite of 26 tests, including for example the following:

```
spawn() -> <0.22.4>
spawn() -> <0.23.4>
register(a, <0.22.4>) -> true
register(a, <0.23.4>) -> {'EXIT',...}
whereis(a) -> <0.22.4>
```

This tests that registering the same name twice fails—but, compared to our previously generated test case (#10), performs a final call of `whereis` to check that the correct pid remains in the registry.

We could continue to enrich our model to capture more of these possible criteria, leading to larger and larger generated test suites to cover the requirements—but to do so begins to feel more and more ad hoc. It is far from clear that there is *any* very canonical way to test for *absence* of state changes. Let us instead consider a plausible bug, and see what it takes to find it.

H. A Plausible State Corruption Bug

The bug we consider is that `register`, in the successful case when it performs a registration, overwrites the contents of the registry, rather than adds to it. It is easy to imagine that an error in manipulating the registry data-structures could result in this behaviour. It is also easy to simulate this bug in our tests: in the case when `register` returns true, we just unregister all the other names in the registry, before continuing the test.

Simulating the bug in this way, we reran our generated test suites to see which test cases catch it. Shockingly, *all the test cases we generated to test requirements pass!* Not only do the 15 tests we think a human programmer might write pass, but so do all 26 tests in the most extensive test suite we generated above. It appears that the test cases we have generated to cover the requirements are not effective at detecting corruption of the state. Yet random testing with QuickCheck finds a counterexample within a fraction of a second:

```
spawn() -> <0.6743.4>
spawn() -> <0.6744.4>
register(b, <0.6744.4>) -> true
register(a, <0.6743.4>) -> true
whereis(b) -> undefined
```

The counterexamples can appear in several forms, but this one is typical. In this example, the fourth step (the second call of register) fails to fulfill requirement `REG001`—it not only adds `a` to the registry, but also removes `b`. This is observed by the `whereis` in step 5. Interestingly, notice that, without step 3, we would not be able to detect the bug even if we could observe the state of the registry perfectly—because the bug only appears when we set up the registry state so that the corruption is detectable, by registering `b` before registering `a`, so that there is data in the registry to be lost.

If we are to test robustly against state corruption, then we need test cases which set up the state in many different ways, perform the operation under test, and then observe the state in many different ways—and it is very unclear what definition of coverage might force us to generate this kind of test. Even given such a definition, it would require a very large number of test cases to achieve perfect coverage. For example, adding the counterexample above to the test suite would not help, if the bug were instead that names lexically earlier than the one being registered were lost. . . a very plausible bug if an ordered binary tree were used to represent the registry.

Random testing, on the other hand, does precisely what is needed to detect this kind of bug: it tests each operation

in many different random states, and makes many different random observations afterwards. This implies that if there is a state-corruption bug, then it will eventually be discovered.

We may think of this as taking a test case that should pass, and interspersing it with many other operations which—according to the model—should not interfere with the passing test. If the test fails, then one of those interspersed operations must have corrupted the state.

I. The Distribution of Tested Requirements

In practice, we find that to get good results from random testing, it is essential to collect *statistics* on our generated tests, and tune generation to obtain a satisfactory distribution. For example, it is invaluable to know *how often* each requirement is tested, not just whether or not it is covered. If the statistics we collect are misleading, then our tuning will be ineffective. Deferring requirements leads to more accurate statistics. To evaluate this effect, for each requirement in the set `REG001`–`REG004`, `UNR001`, and `DIE001`, we measured the effect of deferring it in 20 seconds of random tests. We found each one to be covered 1.5–2× less often when deferred. In other words, *not* deferring requirements leads us to *overestimate* how often they are tested by up to a factor of 2.

III. A Larger Example: AUTOSAR COM

The registry example suggests that, for models instrumented to track requirements tested:

- Minimal test suites generated to achieve requirements coverage may test those requirements poorly.
- Inspecting the generated minimal tests can suggest significant improvements to requirements tracking.
- Deferring counting a requirement as 'tested', until the required side-effects are observed, results in generated minimal test suites resembling those a human developer would write.
- Even so, these test suites are poor at detecting state corruption bugs.

To see whether these conclusions hold more generally, we took as a second example a QuickCheck model of an AUTOSAR component, already annotated to track the requirements covered in random tests. This model was developed for Volvo Cars as part of a larger project to test implementations of the AUTOSAR Basic Software for conformance with the standard—during which more than 200 independent defects were discovered, in six implementations from different vendors, and in the AUTOSAR standard itself [7].

We chose the COM component for this study, which handles communication of signals and packets over different communication busses. The text based specification (AUTOSAR version 4.0 rev 3) consists of 179 pages and contains 56 requirements; the QuickCheck model consists of 1445 lines of Erlang code. The model and implementation are proprietary, but the requirements are standardized [8].

AUTOSAR software is highly configurable. A configuration file containing thousands of parameters is used to generate parts of the software; the configuration determines the software's behaviour. We test a software version generated from a configuration carefully constructed so that most requirements are testable—but some are out of scope because they do not apply to the configuration we use. Other requirements concern software code structure and configuration, rather than dynamic behaviour, and cannot be tested either. When untestable requirements, and those that do not apply to our configuration are discounted, 35 requirements remain, and these are the ones we discuss in the rest of this paper.

Our model is instrumented to record the requirements covered by each test. Whenever a random test is generated, the requirements considered to be covered by that test are saved, and at the end of testing then all the requirements covered are reported. In practice, test runs consist of several thousand generated random test cases, and the requirements coverage information is used in much the same way as line coverage, to determine whether any requirement is tested poorly or not at all (in which case the model, or the test case distribution, should be improved).

When testing the registry, we assumed that the actual names used to register processes were unimportant—we could just use names like `a` or `b`, and the actual choice of name did not affect requirements coverage. But in the case of the COM component, in which almost all functions take *signals* or *pdus* (packets of signals) as parameters, then it really matters *which* signal or PDU is used. The reason is that different signals and different PDUs are configured differently, and so different requirements may apply to them.

A. Improving poor test cases

Consider requirement `COM330`—a typical requirement from the standard—in which ComTransferProperty and ComTxModeMode are configurable values.

> [`COM330`] At any send request of a signal with ComTransferProperty TRIGGERED assigned to an I-PDU with ComTxModeMode DIRECT or MIXED, the AUTOSAR COM module shall immediately initiate ComTxModeNumberOfRepetitions transmissions of the assigned I-PDU.

For TRIGGERED signals in DIRECT or MIXED I-PDUs, a send request must initiate transmission. Our test configuration contains no MIXED I-PDUs, so we need consider only the DIRECT case; to test this requirement we need a test case in which a signal is configured TRIGGERED and its assigned I-PDU has mode DIRECT.

This requirement does *not* state that transmission is *not* initiated in other cases—although this is often implicitly assumed. This kind of ambiguity is problematic to model. We chose *not* to model initiating transmission in other cases, so QuickCheck reports a discrepancy if the software under test does so—although we have no really strong argument for rejecting such an implementation, since the requirements say nothing explicit about these cases.

Our COM model was annotated with requirements as described in section II-A, so we could use the test suite

generation of Sect. II-B to generate a minimal test suite. The test case covering COM330 was this one:

```
com_spec:init() -> ok.
com_spec:send('ComConf_ComSignal_S11',
              <<0, 0, 0, 0, 0, 0>>) ->
  com_service_not_available.
```

This test case initialises the COM component and then sends a binary array of six zeros to signal S11. This signal has ComTransferProperty configured TRIGGERED and is assigned to I-PDU P20, which has ComTxModeMode configured DIRECT, so COM330 applies. But, this is a very poor test case of the requirement! As we can see from the return value, the COM service is not available at all! This is quite correct—and the test passes—because of additional requirements: COM444 states that all I-PDUs are initially stopped, and requirement COM197 says that, for stopped I-PDUs, sending a signal shall indeed return com_service_not_available. COM330 ought really to have an additional precondition—that the I-PDU concerned is active—but we quoted the requirement in full above, and as the reader can easily verify, this precondition does not appear.

Nevertheless, we updated our requirement annotation with this extra precondition; we counted COM330 as covered only for an active I-PDU. When we regenerated a minimal test suite, COM330 was tested by this test case instead:

```
pdur_spec:init() -> ok.
com_spec:init() -> ok.
com_spec:ipdu_group_control(
    [{'MainTxPduGroup1', true},
     {'TxPduGroup1', true},
     {'TxPduGroup2', true},
     {'MainRxPduGroup1', false},
     {'RxPduGroup1', false},
     {'RxPduGroup2', false}],
    true) ->
  ok.
com_spec:send('ComConf_ComSignal_S11',
              <<1, 2, 3, 4, 5, 6>>) ->
  com_ok.
```

This test initialises both COM and PDUR (which is needed to actually transmit the I-PDU). Then the Tx (transmission) groups are started, making the I-PDUs for sending active. Now all the preconditions for covering the requirement hold. The test sends the data sequence 1 to 6 to signal S11—and, for the implementation we were testing, the test fails! QuickCheck reports the following error:

```
Post-condition failed:
expected:
  'CanIf_Transmit'(
         'CanIfConf_CanIfTxPduCfg_P21',
          <<1, 2, 3, 4, 5, 6, 72, 0>>)
```

This is a message from our mocking framework, indicating that a mocked call, expected by our model, was not actually made. The missing call is to the CAN bus interface (a bus commonly used in cars), and it would actually transmit the signal sent by the test. But the COM software under test did not actually transmit the signal—despite COM330—and so our improved test can detect a previously undetected fault.

Actually, it is debatable whether the test *should* fail. COM330 only says that the COM module 'initiates transmission'. What does this mean? Some argue that it is sufficient to send the signal the next time COM's MainFunction is called (this is a function which is invoked regularly by the real-time scheduler). With this interpretation, this behaviour is not a bug. To accept it, we had to develop a variant of the model in which transmissions are delayed until the next call of MainFunction. If we regenerate a minimal test for COM330 from the variant model, *we get the same test case*. The test now passes, since the variant model no longer expects an immediate transmission when send is called—but it is now a very *poor* test case for COM330, since it does not include a call to MainFunction, and so cannot check that the signal is actually sent *at all*.

B. Deferring requirements

Evidently, we should not consider COM330 to be tested until we have *observed* the transmission of the corresponding I-PDU. Therefore, we defer the requirement, as in Sect. II-E, until the I-PDU is transmitted. We consider transmitting an I-PDU to observe both the I-PDU itself, and all of its signals, so that we can defer requirements until either an I-PDU or an individual signal is observed (cf. Sect. II-D). With this modification, the generated minimal test case for COM330 is as we would expect; it calls MainFunction after sending to observe the transmission:

```
                    ⋮

com_spec:send('ComConf_ComSignal_S11',
              <<1, 2, 3, 4, 5, 6>>) -> com_ok.
com_spec:tx_main() ->
  ok = 'CanIf_Transmit'(
         'CanIfConf_CanIfTxPduCfg_P01',
          <<72, ..., 170>>),
  ok = 'CanIf_Transmit'(
         'CanIfConf_CanIfTxPduCfg_P21',
          <<1, 2, 3, 4, 5, 6, 72, 0>>),
  ok = 'CanIf_Transmit'(
         'CanIfConf_CanIfTxPduCfg_P24',
          <<170, ..., 170>>),
  ok = 'LinIf_Transmit'(
         'LinIfConf_LinIfTxPdu_P67',
          <<0, ..., 170>>),
  ok = 'LinIf_Transmit'(
         'LinIfConf_LinIfTxPdu_P71',
          <<0, ..., 170>>),
  ok.
```

In this output the *nested* calls (ok = ...) denote *mocked calls* made by the SUT to transmit over CAN or LIN (another protocol used in cars). Our configuration contains 'periodic' I-PDUs, which are always transmitted when MainFunction is called, whether there is new data or not—this is why we see five calls rather than one. Signal S11 (which appears in the send) is a part of I-PDU P20, which is routed by the PDUR component to the CAN interface CanIf and mapped onto the PDU CanIfConf_CanIfTxPduCfg_P21—so the *second* mocked call above enables us to observe that signal S11 was, in fact, transmitted. Were we to write this test case by hand, then we would have to be aware of this mapping, but our model

of the COM internals derives this information automatically. So, not only is this test the kind of test developers would write, but it is also tricky to get right—and the benefits of automatic generation are clear.

In our requirements tracking, we associate `COM330` with the *signal* appearing in the test—so, for the example above, we record that `COM330` was tested for signal S11. As a result, the minimal test suite we generate contains one test case for *each* signal that has TRIGGERED configured, and is assigned to an I-PDU with mode DIRECT. For our test configuration, this results in 10 tests, each sending a different signal, but in some cases observing the same I-PDU. These tests are not redundant, because the signals may differ in other configuration parameters. This is not only a much better test suite than the single bad test we initially generated for `COM330`—we suspect it is also likely more complete than many human developers would write.

C. Observing omission of activity

In the registry example, we never needed to observe that something *cannot* happen in a certain state. But in the AUTOSAR case we find the following requirement:

> [`COM444`] By default, all I-PDU groups shall be in the state stopped and they shall not be started automatically by a call to Com_Init.

How can we test this requirement? We have to observe that all I-PDU groups are stopped—without inspecting the internal state of the SUT, to which we have no access. We initially marked this requirement as "covered" by a call to Com_Init, but a good test should really also send a message and observe that the actual transmission does not take place. We therefore added a deferred `COM444` requirement, tagged with each I-PDU initialised in Com_Init, to be triggered by an observation that the I-PDU is, indeed, inactive. We *make* such an observation when we send a signal and receive a `com_service_not_available` result (or a corresponding case for receiving a signal). This results in 18 new test cases in our minimal suite, of the form:

```
com_spec:init() -> ok.
com_spec:send('ComConf_ComSignal_S4',
              72) ->
  com_service_not_available.
```

Indeed, these are reasonable tests to observe that the groups are not active.

However, these reasonable-looking tests conceal a problem. When we track requirements covered during *random* testing, we will consider `COM444` to be tested whenever a call to `init` is followed by a call to `send` that returns `com_service_not_available`, *regardless of other calls appearing in between*. In particular, a test might initialise COM, then activate an I-PDU, inactivate it again, and then send a signal for which `com_service_not_available` is returned. This would be counted as a test case that tests requirement `COM444`, but it does not! It only tests that the *last inactivation* of the I-PDU worked. We do not see this kind of bad test among the 18 generated tests because of QuickCheck's shrinking—the test can always be simplified by removing the activation and inactivation, and so QuickCheck does so when generating the minimal test suite. But the problem *will* cause inaccurate measurement of requirements coverage during random testing.

To solve this problem, we need to *discard* deferred requirements which are waiting for observations which can no longer be made. In particular, when an I-PDU is activated, then the observation that it is *inactive* can no longer be made, and any deferred requirements which are waiting for that observation should be discarded. We therefore introduce another primitive operation on requirements, *cancelling deferred requirements*. Whenever we start an I-PDU group, we cancel any deferred requirements waiting to observe that it is inactive. This does not change the minimal test suite that we generate, but results in a more faithful reporting of which requirements have been tested by randomly generated test cases.

D. Preconditioned deferred requirements

Even with these changes, we still found apparently deficient tests in our minimal generated suite. For example, consider `COM050`:

> If Com_SendSignalGroup is called for the signal group, the AUTOSAR COM module shall copy the shadow buffer atomically to the I-PDU buffer.

To observe the copying, we deferred this requirement until the target I-PDU is observed. The resulting minimal test suite contained one test of this requirement for each I-PDU, calling 'Com_SendSignalGroup', activating the I-PDU, and then calling MainFunction to observe the transmission. However, we noticed that these tests appeared in *two different variations*: one in which the I-PDU is activated *before* calling 'Com_SendSignalGroup', and one in which it is called *after*. Both tests are valid, because `COM050` does not require the I-PDU to be active at the time Com_SendSignalGroup is called, but arguably *both* tests are useful, and both should be included in a test suite—but our generated suites contained only *one* of these tests for each I-PDU. This suggests that we ought to split `COM050` into two requirements, one for active I-PDUs, and the other for inactive ones, so that each generated test suite contains a test for both cases—but the text of the requirement above gives no clue that we need to do this.

However, both variations are *very poor* tests for observing the actual copying—because there is nothing interesting to copy in the initial state! When we initialise COM, standard initial values are already copied to the I-PDU, so calling 'Com_SendSignalGroup' immediately afterwards just copies the *same* values again! This is not observable. To test `COM050` properly, we need to call 'Com_UpdateShadowSignal' with a value *different* from the initial value, before we call 'Com_SendSignalGroup'. This is an additional precondition that must be met, before we can say that `COM050` has been tested. We modified our requirements tracking to record `COM050` only when this precondition is met—but note that this is *not* a precondition for the requirement itself, and once again, there is no indication in the text of the requirement that we need to do this.

E. Quality of the test suite

Before we began to defer requirements, we generated a minimal test suite of 17 tests that purported to cover all 35 requirements—but the tests were very poor. We inspected the tests manually, and improved our requirements tracking as explained above. This involved modifying 67 out of 1445 lines in the model, or 4.6% of the code. As a result, the minimal suite generated to test the 35 requirements now consists of 74 tests—often with several tests per requirement, but also almost always with several requirements per test. Requirements coverage was of course 100%, both before and after this process—all requirements are exercised in each case, but the question is *how well*?

We inspected the resulting test cases manually, to assess how closely they resemble tests that a human developer would write for the given requirements. This is rather subjective—for example, the standard defines requirement `COM619`:

> [`COM619`] Configuration of Com_GetConfiguration-Id: The provided Identification shall be set during configuration process and cannot be changed by the AUTOSAR COM module.

But this requirement is really untestable. We generated the test case:

```
com_spec:init() -> ok.
com_spec:get_config_id() -> 0.
```

where 0 is the configured value. This seems reasonable, and it is hard to see that a human could do better.

The test case presented in Sect. III-C to verify requirement `COM444` is repeated once for each signal. A human tester might write only a few of these, or—better—write *one* test that initializes COM and then tests *all* signals.

We concluded that 28 requirements are well tested by our test suite, with similar tests to those a human would write, while 6 requirements were tested by test cases that a human would have written differently, perhaps better. We consider that 7 requirements were inadequately tested by the generated test suite. In some cases, adding more preconditioned requirements or preconditions on specific events could have improved this.

F. State corruption errors

We now have a suite of 74 tests, constructed to cover all 35 requirements. How good is it at detecting realistic state corruption errors? To find out, we modified the send command to write the signal into the assigned I-PDU, and at the same time overwrite the first byte of the next I-PDU. These I-PDUs are logically unrelated, but happen to be next to each other in memory. This is a realistic software defect, and can cause major damage if the signal in the overwritten I-PDU is safety critical.

When testing this modification with randomly generated test cases, we detected the failure after generating and executing 56 random test cases (and within a minute, including shrinking to a minimal failing test).

```
com_spec:init() -> ok.
pdur_spec:init() -> ok.
com_spec:send('ComConf_ComSignal_S1', 14) ->
  com_service_not_available.
com_spec:ipdu_group_control([
    {'MainTxPduGroup1', true},
    {'TxPduGroup1', true},
    {'TxPduGroup2', true},
    {'MainRxPduGroup1', false},
    {'RxPduGroup1', false},
    {'RxPduGroup2', false}],
    false) ->
  ok.
pdur_spec:frif_trigger_transmit(
          'PduRConf_PduRDestPdu_P06') ->
  {<<11, 0, ..., 8, 0>>, ok}.

Reason:
  Failed postcondition:
      {<<11, 0, ..., 8, 0>>, ok} /=
        {<<72, 0, ..., 8, 0>>, ok}
```

Here 14 is sent on signal S1 before any I-PDU is activated. Then we activate all transmission (Tx) I-PDUs, and trigger transmission of the I-PDU containing signal S4. But S4 has value 11 instead of the expected 72 (the first byte in the binary value in the postcondition). Sending a value on the unrelated signal S1 has overwritten signal S4 in a different I-PDU!

None of the 74 tests in our suite is able to detect this fault, indicating that state corruption errors in which two features interact are hard to detect using test cases designed to validate specific requirements.

G. A Further Improvement

Some requirements are very hard to cover with random tests, a problem that is exacerbated when the requirement is deferred and a specific observation is needed. For example, requirement `COM734` took about an hour to generate a test case for.

> [`COM734`] At a send request of a signal with ComTransferProperty TRIGGERED_ON_CHANGE assigned to an I-PDU with ComTxModeMode DIRECT or MIXED, the AUTOSAR COM module shall immediately initiate ComTxModeNumberOfRepetitions transmissions of the assigned I-PDU, if the new value of the sent signal differs to the locally stored (last sent or init) value.

It is quite hard even to satisfy the conditions under which we can defer the requirement, let alone observe it.

We plan to extend our search strategy to remember sequences that defer requirements, but never observe them, and then retry by continuing from that point to generate commands that make the observation. This should increase the probability of finding a test for the requirement.

IV. Discussion and Related Work

In a recent taxonomy of model-based testing tools, Utting *et al* identify six *test selection criteria*, of which requirements coverage is one [9]. However, they do not discuss what it means to cover a requirement, other than to suggest that requirements may be attached to state transitions or formulæ in a postcondition. Shafique and Labiche's systematic review of state based MBT tools mentions that requirements may also

be attached to *paths* in the state space, but gives no further details [10].

Bouquet *et al* presents a method to trace requirements in generated tests [12], based on structural coverage of model fragments tagged with requirement names. Their test generator can generate additional API calls to observe state changes, which may be analogous to our deferred requirements. They do not discuss testing for state corruption bugs.

Jensen *et al* consider test generation from business rules, by using a constraint solver to find operations that establish the precondition of the rule under test [13]. However, they assume that a rule is covered as soon as the operation it applies to is executed, so their generated tests do not contain operations to *observe* the state changes that business rules specify. Perhaps their approach could be extended to use the constraint solver to generate a "postscript" following the rule under test, that observes the effect of the postcondition—in a similar way to our deferred requirements—thus improving the fault finding effectiveness of the generated tests.

A popular approach to generate test suites fulfilling coverage goals is to use a model checker—for a recent survey, see Fraser *et al.* [14]. In this approach, coverage goals are expressed in temporal logic, so if requirements are expressed in temporal logic then we could generate minimal test suites for them in this way. For example, if $f(a_1 \ldots a_n) \mapsto x$ is true when f is called with arguments $a_1 \ldots a_n$ and returns x, then a requirement on the registry might be

$$\begin{array}{l}\Box\ \forall n, p.\ \mathit{register}(n, p) \mapsto \mathsf{true} \implies \\ \quad (\forall p'.\ \mathit{whereis}(n) \mapsto p' \implies p = p')\ \mathcal{U} \\ \quad\quad (\mathit{unregister}(n) \mapsto \mathsf{true} \vee \mathit{kill}(p) \mapsto \mathsf{ok})\end{array}$$

But this is not one of the informal requirements we stated—those requirements were expressed in terms of state, not temporal logic. Formulating them in terms of temporal logic requires a non-trivial translation step, and it is not clear that covering the reformulated requirements would be equivalent to covering the original ones. While some of the AUTOSAR requirements might naturally be formalised in temporal terms (*e.g.* a message should be sent), many others refer to changes in an opaque state. However, our deferment of requirements that cannot be tested in one step (because of the need to observe state changes later) might be naturally represented in temporal logic.

We studied requirements coverage here: a related question is to what extent complete *code* coverage ensures that test suites will reveal faults? A recent study by Gay et al. [15] generated minimal test suites to achieve maximal structural coverage for a number of real applications, and measured their ability to find faults. The results were quite alarming—for several systems, randomly generated suites of the same size were equally or more effective at revealing faults! They concluded "We believe that structural coverage criteria are, for the domain explored, potentially unreliable, and thus, unsuitable, as a target for determining the adequacy of automated test suite generation." A part of the problem was that the real faults studied were often faults of *omission*—the developers forgot to implement a feature, for example—and code coverage cannot be expected to account for code that does not exist. One would expect test suites generated to achieve full *requirements coverage* to do better, because missing features ought always to be reflected by unsatisfied requirements, but as we have seen, state corruption bugs still pose a problem.

DO-178C [1] does mandate coverage of data coupling also, which may be intended to catch state corruption bugs. However, it is not entirely clear how to interpret this mandate, and is in any case inapplicable to black box testing. It is interesting that random testing appears able to find these bugs quite easily.

We have claimed several times that the minimal test cases we generate resemble those a human developer would write for the same requirements, based on our observations of real test suites including AUTOSAR test cases [16]. It would be interesting to study actual hand-written test cases in more detail, to evaluate this subjective claim.

V. Conclusion

Through a small motivating example, and a much larger example in the automotive domain, we have shown that the notion of *testing a requirement* is more subtle than it might seem. Minimal test suites generated to maximise requirements coverage can contain surprisingly poor tests. On the other hand, inspecting such test suites can suggest major improvements to requirements tracking, and we showed how deferred requirements and preconditioned requirements can help. Better requirements tracking not only leads to better minimal generated test suites—close to those a human developer would write—but also provides invaluable information for tuning the distribution of *random* tests, so as to cover hard-to-reach requirements more often. There is a class of plausible bugs—state corruption bugs—which are hard to find using requirements-based tests, and this conclusion may well apply to hand-written tests also. Fortunately, random tests find these bugs quite easily.

Acknowledgment

This work was funded by EU FP7 project PROWESS, and by Swedish Strategic Research Foundation project RAWFP.

References

[1] SC-205, "Do-178c, software considerations in airborne systems and equipment certification," RTCA, Tech. Rep., 2011.

[2] J. Hughes, "Quickcheck testing for fun and profit," in *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, ser. PADL'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 1–32.

[3] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279.

[4] U. Norell, H. Svensson, and T. Arts, "Testing blocking operations with quickcheck's component library," in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13. New York, NY, USA: ACM, 2013, pp. 87–92. [Online]. Available: http://doi.acm.org/10.1145/2505305.2505310

[5] J. Hughes, "Software testing with quickcheck," in *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, 2009, pp. 183–223.

[6] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 102–111. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486803

[7] 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*, April 2015, pp. 1–4.

[8] AUTOSAR Consortium, "Automotive open system architecture, standard documents," https://autosar.org/, 2013.

[9] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[10] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 1, pp. 59–76, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10009-013-0291-0

[11] J. Botella, F. Bouquet, J.-F. Capuron, F. Lebeau, B. Legeard, and F. Schadle, "Model-based testing of cryptographic components – lessons learned from experience," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ser. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 192–201. [Online]. Available: http://dx.doi.org/10.1109/ICST.2013.42

[12] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting, "Requirements traceability in automated test generation: Application to smart card software validation," in *Proceedings of the 1st International Workshop on Advances in Model-based Testing*, ser. A-MOST '05. New York, NY, USA: ACM, 2005, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083282

[13] S. Jensen, S. Thummalapenta, S. Sinha, and S. Chandra, "Test generation from business rules," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, April 2015, pp. 1–10.

[14] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009. [Online]. Available: http://www.nada.kth.se/ karlm/sofrel/SNA-TR-2007-P2-04.pdf

[15] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "The risks of coverage-directed test case generation," *Software Engineering, IEEE Transactions on*, vol. 41, no. 8, pp. 803–819, Aug 2015.

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