

# Modelling of Autosar Libraries for Large Scale Testing\*

Wojciech Mostowski

Centre for Research on Embedded Systems  
Halmstad University, Sweden  
wojciech.mostowski@hh.se

Thomas Arts

Quviq AB, Sweden  
thomas.arts@quviq.com

John Hughes

Department of Computer Science and Engineering  
Chalmers University of Technology, Sweden  
Quviq AB, Sweden  
rjmh@chalmers.se, john.hughes@quviq.com

We demonstrate a specific method and technology for model-based testing of large software projects with the QuickCheck tool using property-based specifications. Our specifications are very precise, state-full models of the software under test (SUT). In our approach we define (a) formal descriptions of valid function call sequences (public API), (b) postconditions that check the validity of each call, and (c) *call-out* specifications that define and validate external system interactions (SUT calling external API). The QuickCheck tool automatically generates and executes tests from these specifications. Commercially, this method and tool have been used to test large parts of the industrially developed automotive libraries based on the Autosar standard. In this paper, we exemplify our approach with a circular buffer specified by Autosar, to demonstrate the capabilities of the model-based testing method of QuickCheck. Our example is small compared to the commercial QuickCheck models, but faithfully addresses many of the same challenges.

## 1 Introduction

The multi-vendor and multi-platform character of automotive system development is a major challenge in testing the underlying software due to the multiplicity of possible software and hardware interactions and differences between various versions of the referenced libraries. Testing such software off-line brings the additional challenge of substituting low-level hardware interfaces and APIs with software mock-ups.

The Autosar standard [5] is an automotive industry effort to support multi-vendor software development. In particular, the standard specifies the basic software library interfaces and behaviour that Autosar compliant implementations should adhere to. Ideally, compliant Autosar components from different vendors should be interchangeable with little additional effort necessary to validate a complete system based on Autosar. However, in practice small differences in implementation choices due to performance considerations, particular platform limitations, or interpretation freedom for (purposely or inadvertently) underspecified behaviour may lead to compliance issues.

To manage the complexity of testing and validation in this context, we employ Model-Based Testing (MBT) techniques [11] rather than the classical unit testing approach. Because of the number of test cases necessary to cover the interactions and behavioural differences, unit testing suffers from scalability issues. In comparison, the base for MBT testing are models of the system and its components, based on which tests in an arbitrary quantity are generated on the fly. A carefully chosen test data generation method ensures a sufficient coverage (quality) of the resulting tests. New configurations and library

---

\*This work is supported by the Swedish Knowledge Foundation grant for the AUTO-CAAS project.

versions of the system under test are simply covered by adapting the models and regenerating the tests for various configurations. Another important feature of MBT, as opposed to e.g. model checking, is that an actual running library or complete product compiled from a low-level language can be tested and validated, rather than its model (although in our particular case the model itself is also analysable, see Sect. 5). The infinite test space of this validation scenario obviously cannot be completely covered by MBT, however, the random test data generation ensures arguably good coverage of the code under test. The specification and the testing can be done with or without the source code availability for inspection, the latter being often the situation in our industrial projects.

Our MBT method of choice is property based testing using the QuickCheck tool developed by Quviq [2, 8]. In QuickCheck, a model is an Erlang [7] functional program that follows a predefined structure and API to form a behavioural description of a given software component. The properties that can be checked with generated tests are postconditions (test oracles) of single operations and *call-out* specifications. Call-out specifications are given in a process algebra-like language [10] and describe the calls to other components that the operation is allowed or required to make. The rest of the model defines the *symbolic execution* state of the component under test and operation preconditions, both of which define the valid execution traces for the tests to be generated. For the software under test written in C, which is the case for the Autosar components, QuickCheck provides a flexible interface that allows to abstract the C function calls in the model and run the underlying C implementation during the actual testing. The general philosophy of QuickCheck is to run several short tests quickly to detect problems early, rather than creating large and complex testing scenarios for which failures are extremely difficult to analyse. Should a larger test case need to be generated to exhibit a bug, QuickCheck offers a mechanism to *shrink* the test data to a smaller, possibly minimal, set that leads to the same failure. Consequently, the effort of debugging is also reduced.

QuickCheck has been used in large scale to specify and test Autosar basic software implementations [3]. In this paper we exemplify the QuickCheck method by presenting a small model for one of the Autosar components, a circular buffer for queuing arbitrary data. The client component is a message box system, that uses the circular buffer to specifically store message pointers (Sect. 2). Using this simple example we show how the multi-level component specifications are used to test Autosar implementations, and we show how C function mocking is employed in QuickCheck to test the call-out specifications (Sect. 3). Furthermore, we show how *clustering* specifications can be used to validate interactions between components within one functional set, i.e., that components respect each other APIs within a cluster (Sect. 3.4). The concrete implementation that we test comes from the open-source Autosar implementation by ArcCore<sup>1</sup>, the third partner in the AUTO-CAAS project.

The mocking mechanism of QuickCheck for testing the call-out specifications is also used for checking the failure consequences in the complete system. This is done by mocking parts of the system with a behaviour specified by a model, into which we can incorporate known non-compliant behaviours or simply inject faults. This is described in Sect. 4. Then, Sect. 5 concludes the paper. The complete QuickCheck model for this case study can be found in App. A and on-line at [http://ceres.hh.se/mediawiki/CirqBuff\\_and\\_MBox\\_QuickCheck\\_Models](http://ceres.hh.se/mediawiki/CirqBuff_and_MBox_QuickCheck_Models).

## 2 The Case Study

The case study we focus on is a circular buffer implementation `CirqBuff` [5]. The buffer has three core API functions, defined by the following C prototypes:

---

<sup>1</sup><https://www.arccore.com>.

```

typedef struct {
    int maxCnt;           /* The max number of elements in the list */
    int currCnt;         /* The current number of elements */
    size_t dataSize;     /* Size of the elements in the list */
    void *head; void *tail; /* List head and tail */
    void *bufStart; void *bufEnd; /* Buffer start/stop */
} CirqBufferType;

```

Figure 1: The CirqBufferType C data type definition.

- `CirqBufferType *CirqBuffDynCreate(size_t size, size_t dataSize);` that creates a circular buffer of the given size which stores arbitrary data (byte sequences) of a given size;
- `int CirqBuffPush(CirqBufferType *cPtr, void *dataPtr);` that enqueues an element into the buffer, the element is passed through a pointer to the array containing the data to be enqueued. The result is 0 on success, otherwise 1; and
- `int CirqBuffPop(CirqBufferType *cPtr, void *dataPtr);` that dequeues an element from the buffer by placing it in an array, and indicating the operation status with the return value.

The `CirqBufferType` is a C structure that keeps track of the current queue contents through the current element count `currCnt` and head/tail pointers to the allocated queue storage buffer `bufStart...bufEnd`, as well as the static limits of the buffer – maximum size `maxCnt`, and the `dataSize` field, see Fig. 1 for the complete C structure definition. The data is copied to and from the buffer using one of the `memcpy` routines available on the target platform, this is one of the configuration options of this component that Autosar has multiplicity of, and that in general complicate the testing effort.

This circular buffer is used in a few places in the open source Autosar implementation, e.g., for communication transmission buffers. For presentation, we picked the simplest code available, a message box component that uses the circular buffer to store message pointers. The `Mbox` API is the following:

```

Arc_MBoxType* Arc_MBoxCreate(size_t size);
sint32 Arc_MBoxPost(const Arc_MBoxType *mPtr, void *msg);
sint32 Arc_MBoxFetch(const Arc_MBoxType *mPtr, void *msg);

```

The only field of the `Arc_MBoxType` structure is `cirqPtr` that keeps the circular buffer reference. The post and fetch functions are each a simple delegation of calls to `CirqBufferPush` and `CirqBufferPop`, respectively, with the same meaning of result values. `Arc_MBoxCreate` creates a circular buffer with the data size fixed to the pointer size `sizeof(void *)`. Thus, `MBox` specialises `CirqBuff` to store pointers.

### 3 Specifying and Testing the Implementations with QuickCheck

Tens of software components for vehicle software are specified in Autosar in a modular way. The task at hand is to modularly test both `CirqBuff` (Sect. 3.1) and `Mbox` (Sect. 3.3) implementations. However, we also want to test the software together, i.e., the component integration. We do so by automatically deriving its specification (Sect. 3.4).

Using MBT, one first builds a model, the specified behaviour of the software under test (SUT). Based on the model, an arbitrary number of test cases can be generated. Test cases are sequences of calls to the API functions with generated data parameters. In addition, the test contains information of what

```

-module(qc_cb).

-record(cb_state, {ptr, size, data_size, elements}).
initial_state() -> #cb_state{ elements=[] }.

invariant(S) -> length(S#cb_state.elements) =< S#cb_state.size.
postcondition_common(_S, {call, _, create, _, _ }, _Res) -> true;
postcondition_common(S, Call, Res) -> eq(Res, return_value(S, Call)).

create(Size, DSize) -> cb:'CirqBuffDynCreate'(Size, DSize).
create_args(_S) -> [nat(), nat()].
create_pre(S, [Size, _DSize]) -> S#cb_state.ptr == undefined andalso Size > 0.
create_next(S, R, [Size, DSize]) ->
    S#cb_state{ptr=R, size=Size, data_size=DSize }.

push(Ptr, Val) -> DataPtr = eqc_c:create_array(unsigned_char, Val),
    CallRes = cb:'CirqBuffPush'(Ptr, DataPtr), eqc_c:free(DataPtr), CallRes.
push_args(S) -> [S#cb_state.ptr, vector(S#cb_state.data_size, char())].
push_pre(S) -> S#cb_state.ptr /= undefined andalso
    length(S#cb_state.elements) < S#cb_state.size.
push_return(_S, _Args) -> 0.
push_next(S, _R, [_Ptr, Val]) ->
    S#cb_state{elements = S#cb_state.elements ++ [Val]}.

```

Figure 2: Excerpt of the CirqBuff model.

external API calls the SUT makes. These API calls are automatically mocked. The generated tests are then executed on the SUT and the tool validates postconditions and call-out behaviour for each call.

As MBT tool we use Quviq's QuickCheck with two distinguishing features: expressive, state-full, *executable* models, and failed test minimisation, called *shrinking*. Shrinking is a mechanism for reducing the size of the test case that failed, so that it still leads to the failure, but with a possibly (much) shorter execution trace. The controllable (see Sect. 4) *data generators* make this possible.

### 3.1 The CirqBuff Model

A snapshot of the QuickCheck model for CirqBuff for discussing is shown in Fig. 2, the complete model is quoted in App. A. The model is a collection of Erlang [7] functions describing the different aspects of the behaviour of the component under test. First, the declared record type `cb_state` holds the *symbolic execution* state of the model, i.e., a model view of the CirqBuff's current state. It contains the pointer `ptr` reference of the freshly created buffer by the `CirqBuffDynCreate`, the `size` and the `data_size` requested during the creation, and the model contents of the buffer `elements`, an Erlang list of arbitrary values. The state is initialised by `initial_state` which defines `elements` to be an empty list representing an empty buffer. The remaining fields of `cb_state` are set to **undefined**.

The `invariant` function declares a condition to be checked throughout the execution of the *actual* tests. Here it is defined somewhat artificially, because its validity is guaranteed by the construction of the rest of the model (it only refers to the model state, not the implementation state), but in general it can

be used to check specific validity conditions of the implementation, e.g., the head pointer to be within the bounds of the `CirqBuff` storage buffer. Similarly, the common postcondition is checked after every `CirqBuff` operation call during the actual test. Using Erlang parameter matching, we state that no extra checks are necessary for the `create` call, the postcondition is `true`, while all the other calls should have their actual return values equal to the ones specified by the model. These model return values are specified separately for each operation, as we describe next.

For every API operation we declare a group of *callback* functions that define:

1. How to call the C function to execute the actual test (the base callback);
2. How to generate its arguments (the `_arg` callback);
3. What is the precondition, defined over the model state and operation arguments, for the operation to be eligible for execution (the `_pre` callback);
4. What is its expected return value (the `_return` callback), and finally;
5. How to update the model state after the operation is executed (the `_next` callback).

Each operation can have its own specific postcondition defined with a `_post` callback and a call-out callback to specify internal and external API interactions, we describe this in the next section. The obligatory callbacks are the base one and the arguments one, through which QuickCheck recognises that an operation should be part of test generation. The others have default values if unspecified.

The `create` arguments are two positive integers indicating the queue size and data size. The queue size is restricted to be strictly positive by the precondition of `create`. Thus, all data sets generated that have the size equal to 0 are rejected during test generation process. The precondition also requires that `create` is called only once per test when the queue is not yet initialised, i.e., when no queue pointer reference has been recorded in the model state. The new model state after `create` is one with the newly created pointer and the creation arguments recorded in the corresponding fields of the state record.

The first argument of the `push` operation is the pointer stored earlier in the model state. The second argument is an element to be enqueued, here a random `data_size` length sequence of characters. The precondition limits the `push` calls to initialised and not yet full queues. Due to this precondition, the expected return value is 0. In the next state the newly enqueued element is recorded in the `elements` list. The `pop` operation is a mirror of `push` with no new specification concepts, hence we skip it in Fig. 2.

The precondition, arguments, and the next-state callbacks are primarily used during test case generation, i.e., when the model is symbolically executed to generate valid execution traces with sample input data. The base and the return value callbacks, together with the postcondition and the invariant described above, are used during the test execution to (a) trigger the actual C implementation call, and (b) check the correctness of the result. The model state is updated both during the test generation, and test execution for bookkeeping. In the former case we refer to it as *symbolic state*, in the latter as *dynamic state*.

The execution of `create` involves a direct call to its C implementation `CirqBufDynCreate` with the generated arguments. This wrapped C call is automatically generated by the QuickCheck C interface generation library `eqc_c`. A particular instance is set up by calling:

```
eqc_c:start(cb, [ {cppflags, "-std=c99"}, {c_src, "cirq_buffer.c"} ]).
```

However, the `CirqBuffPush` C implementation expects the data to be passed through a pointer, and this pointer has to be created first from the pure data generated by QuickCheck. This is done with the QuickCheck library function `eqc_c:create_array` that delegates pointer creation to the C executable, which is then passed on to `CirqBuffPush`, after which it can be immediately freed, see again the specification of `push` in Fig. 2.

```

prop_cb() -> ?FORALL(Cmds, commands(qc_cb),
  begin {_, _, Res} = run_commands(Cmds), Res == ok end).

> eqc:quickcheck(qc_cb:prop_cb()).
...[Repeated 100 times]...
OK, passed 100 tests

```

Figure 3: Testing CirqBuff.

### 3.2 Testing CirqBuff with QuickCheck

The model-based test execution of a specified component in QuickCheck is done through defining a test property and passing it on to QuickCheck, see Fig. 3. A test sequence is generated with `commands` for the model specified in module `qc_cb`. The generated test sequence is executed with `run_commands(Cmds)`. The repetitive generation of test cases is controlled with the `?FORALL` construct, by default 100 random tests are generated. Additional parameters can be used in the property and the model, e.g., to induce the creation of more or longer test cases or to collect specific test statistic (e.g., requirement coverage [1]). In the simplest form we just state `Res == ok` to check for the result of running each test. Passing this property to QuickCheck results in passing all 100 tests.

In fact, we can run thousands of tests that exercise the creation of circular buffers with different sizes and data sizes. We push and pop from these buffers without finding any error in the implementation. Notably, with the specific model construction we do respect the interface and never try to overflow or underflow the buffer.

### 3.3 The MBox Model

By essentially copying and pasting the specification, we could test `MBox` in exactly the same way. However, being a client implementation that uses `CirqBuff` as a library we want to take a different approach that details this dependency. We test `MBox` in isolation and use *mocking* for the `CirqBuff` calls.

The general idea of mocking is to execute a test of an upper level operation (here from `Mbox`) in an environment, where the lower level operations (here from `CirqBuff`) are replaced by *emulated* (mocked) ones. This enables tracing of the particular low-level calls actually made (or omitted, in case something should be forbidden), and to check the call arguments. It also facilitates testing of software that uses hardware interfaces that cannot be run outside of the target platform (e.g., ECU ports or A/D converters).

Code mocking is generally a tedious task involving writing replacement code of the mocked functions. QuickCheck has an interface that makes the process considerably easier and provides a flexible language with clearly defined semantics to specify the interactions with the mocked component [10]. Mocking of a component is enabled by providing its API description in the model, such definition for `CirqBuffPush` is given in Fig. 4. With just the API, QuickCheck replaces the `CirqBuff` implementation with an emulated alternative. The `stored_type` and `buffer` clauses indicate that the `dataPtr` parameter should not be simplified from a pointer to just a value during mocking (which is the default behaviour), and because `void` has no values, the type is changed to `unsigned char` to enable dereferencing values from pointers.

The specification of `MBox` follows the same pattern as in `CirqBuff`, here we only underline the core differentiating elements. The message enqueueing operation `post` is specialised to store pointers, i.e., byte sequences of `?PTR_SIZE` length exactly, thus data size is not stored in the model state. The

```

-module(qc_mbox).

api_spec() -> #api_spec { language = c, mocking = eqc_mocking_c,
  modules = [ #api_module{ name = mbox, functions = [
    #api_fun_c{ name = 'CirqBuffPush', ret = int,
      args = [ #api_arg_c{type = 'CirqBufferType_', name = cPtr, dir = in},
        #api_arg_c{ type = 'void_', name = dataPtr, dir = in,
          stored_type = 'unsigned_char_',
          buffer = {true, "cPtr->dataSize"}} ] ... }].

post_args(S) -> [S#mbox_state.ptr, vector(?PTR_SIZE, char())].
post_callouts(_, [_ , Value]) ->
  ?CALLOUT(mbox, 'CirqBuffPush', [?WILDCARD, Value], 0).

```

Figure 4: Excerpt of the MBox model.

`_callouts` callback specifies the external calls of `post`. Here, a single call to `CirqBuffPush` is required with two arguments. The first is the pointer to the `CirqBuffType` structure that is created by `MBox` earlier with a call to `CirqBuff`'s `create` function. We choose to ignore it with `?WILDCARD`, meaning that we do not check if `MBox` uses this pointer consistently with all `CirqBuff` calls. The second argument is the value pushed onto the circular buffer and it is simply passed on from the call to `post`. The result of the call should always be 0 indicating a success.

Specified this way, `MBox` can be tested with the `CirqBuff` fully mocked and with all the calls to `CirqBuff` traced and checked. The `MBox` test property is identical to the one for `CirqBuff`, the difference in test generation and execution lies in the mocked API specified in the model. Should the calls to `CirqBuff` made by `MBox` be different from the specified ones, or missing, the test fails and the details of the mismatch between the expected and actual calls is reported.

### 3.4 Analysing Models and API Interactions through Clustering

We have specified only two components, one depending on the other, and we have shown how the interactions down this simple hierarchy can be modelled and tested through mocking and call-out specifications. This scenario, however, is very simple. Moreover, we only tested for the occurrence of the lower-level call in relation to the state of the higher-level model and the particular operation parameters. More generally, we would like to test multi-level dependencies of several components and check that they all fully respect each other APIs according to the corresponding model specifications. For example, so far we did not check that the `MBox` is not trying to overflow the circular buffer by placing too many elements in it.<sup>2</sup> A typical example of a scenario requiring such elaborate check is a protocol stack, in Autosar there are several such protocols specific to automotive applications, including CAN-Bus and Flex-Ray.

To enable such a check, the set of interacting components is placed in a so-called *cluster* model. The models of components placed in a cluster only need slight extensions to facilitate the process. First, the operations that can be called by other components have to name the possible callers by their module

<sup>2</sup>Actually, incidentally it was checked implicitly, but only because of the specific model construction and the fact that the sizes of the two components are always the same. It was not done explicitly by checking the preconditions of the circular buffer.

```

-module(qc_cluster).

components() -> [qc_cb, qc_mbox].

api_spec() -> eqc_cluster:api_spec(?MODULE).

property_cluster() -> ?FORALL(Cmds, commands(?MODULE),
    begin {_, _, Res} = run_commands(Cmds), Res == ok end).

```

Figure 5: The cluster model.

name. Second, the API specifications of the mocked calls to the lower-level routines need to have a direct indication specified as to which modelled component and operation in the cluster they refer to.

In our case study, each operation in the model of the `CirqBuff` has to name the `MBox` model as the caller of the operation. For the push operation this is done with `push_callers() -> [qc_mbox]`. Then, the mocked API for `CirqBuffPush` is extended with a `classify = {qc_cb, push}` clause to indicate the binding to the appropriate operation in the `CirqBuff` model. These two simple extensions repeated for every operation allow us to now give a short and straightforward cluster specification for the two components shown in Fig 5. It simply lists the components in the cluster and defines their underlying collective API by collecting the APIs of all included components. Testing the specified cluster property by QuickCheck establishes that `MBox` respects the API of `CirqBuff`. Namely, if any of the `MBox` calls is eligible to be included in the generated test case for `MBox`, then the corresponding call-out to `CirqBuff` is checked to be eligible in the `CirqBuff` model. This includes checking that the caller is an authorised one. This is done by pure *symbolic* execution of both models, no actual implementations are tested in this process, it is only the models and their interactions that are checked. The actual implementations are tested one by one using their corresponding models like described above.

## 4 Faults, Deviations, and Consequence Testing

The mocking mechanism discussed in Sect. 3.3 also allows us to test implementation against *mutations* of sub-components to test for fault tolerance. This is part of on-going research [4, 9], here we exemplify the idea and show the capabilities of QuickCheck to guide test generation to trigger a hidden fault.

Assume a scenario where the `CirqBuff` has either faults, or deliberate modifications dictated by, e.g., performance optimisations or target platform limitations. These changes may or may not be compliant to the specification given by the standard. When they are not we call them *deviations*, rather than just faults. Done in good faith, they do not necessarily have to lead to problems or faults in the top-level behaviour. We want to model and test for this, without going through the burden of modifying the implementation of the low-level component, or constructing several implementations for this purpose. Instead, we test using the mocked version of the possibly deviating component and introduce the deviation in the mocking.

Suppose the `CirqBuff` silently accepts all buffer sizes when creation is requested, however, due to platform limitations only buffers of size 128 are created, under the assumption that the client code can check the available space each time a new elements is enqueued (which in fact our particular client `MBox` *does not do*). To include this behaviour in our model it is sufficient to make one line change in the call-out specification of `CirqBuffPush` to return 1 on reaching the “silent” limit of 128 elements:

```

weight(#mbox_state{ptr=undefined}, Op) ->
  case Op of create -> 1; _ -> 0 end;
weight(_S, create) -> 0;
weight(_S, post) -> 5;
weight(_S, fetch) -> 1.

prop_mbox() -> ?FORALL(Cmds, more_commands(50, commands(?MODULE)), ...

> eqc:quickcheck(qc_mbox:prop_mbox()).
...[snip].Failed! After 23 tests.
[Long trace of 218 commands]
Shrinking xx[snip].x(1056 times)
[Shrunk failure trace of 130 commands]
Reason:
  Post-condition failed:
  Failed postcondition: common: 1 /= 0
false

```

Figure 6: Extending the test scenario.

```

?CALLOUT(mbox, 'CirqBuffPush', [?WILDCARD, Value],
  if length(S#mbox_state.elements) < 128 -> 0; true -> 1 end)

```

Regenerating the tests with this updated model *does not* reveal any problems in the implementation of MBox, while it *should*. The reason is the `nat()` size generator that does not attempt to create MBox-es of sizes large enough, and even if it would, the generated traces do not contain enough post operations to fill up the buffer and trigger the fault. To stimulate QuickCheck towards reaching the faulty state we first change the size data generator from `nat()` to `choose(1, 256)` that uniformly generates integers in the prescribed range rather than conservatively small ones. Then, we alter the test property with `more_commands` to increase the expected length of generated test traces by the factor of 50, and we provide relative *weights* to ensure that the post operation dominates in the generated tests to quickly fill up the buffer, see Fig. 6. This is sufficient to discover the fault after a handful of test runs by QuickCheck. Expectedly, the shrinking algorithm is not able to generate test cases shorter than 130 commands to trigger this fault, however, 130 *is* the minimal fault triggering trace – one create, 128 post-s to fill up the buffer, and one last to overflow it.

Similarly to `more_commands`, with the `more_bugs` directive QuickCheck attempts to discover more than one fault in one testing sessions, instead of halting after the first test failure. This way, different kinds of faults (different postconditions failing) can be identified sooner.

An often quoted criticism towards MBT is the high specification to implementation ratio, here roughly 1:1 which is equivalent to a reimplementing effort. However, this is *all* the effort required – with one or two line changes to the model we have just shown how to test the implementation against a different sub-component and how to substantially change the shape of the generated test cases to trigger a hidden fault. In the classical testing approach, a large number of new test cases would have to be developed as well as complete component implementations to achieve the same effect.

Finally, some remarks about the test execution and bug finding time performance should be made. The complete process of generating the basic 100 test cases for the MBox implementation in its initial

configuration from Sect. 3.3 (i.e., with `CirqBuff` mocking, but without custom operation weights and without `more_commands`) and executing them takes a mere 3 seconds on a standard laptop. Expanding the test scenario to hit the bug increases this time to 36 seconds, which is also acceptable. Moreover, this is the time for generating and executing all 100 tests, but the bug is found after just  $\sim 20$  tests in 7 seconds. Specific test scenarios that attempt to find an infrequently occurring bug may require increasing the standard 100 random tests towards a much larger number (tens or hundreds of thousands). However, the running time will grow only linearly to the number of tests, which is perfectly acceptable considering the goal of finding the bug and the full process automation.

What does take considerable time is the shrinking process for large counterexamples. In the example we have shown in Fig. 6 it already took just under 6 minutes for the counterexample trace of 218 operations. This is because each single shrinking attempt requires at least one complete rerun of one test case to check if the bug still occurs with the smaller trace. This procedure, however, is again fully automatic and replaces laborious, time consuming, manual process of analysing and rearranging test cases by the test engineer to find a shorter counterexample.

## 5 Conclusions

Using a small example from the automotive Autosar standard we have presented the QuickCheck methodology for model-based testing. By generating tests from models, instead of manually writing them, we can cover an arbitrary set of test scenarios. By controlling test data generators, we can drive the process towards a specific testing goal. The randomness of test generation brings the additional advantage of creating atypical test sequences that would not be considered by a testing engineer.

The added value of model-based testing is that we can arbitrarily raise the number of tests to cover new behaviours of the system under test with very little additional effort. In our small case study, changing a couple of lines in the model exhibited a hidden bug, while the shrinking mechanism of QuickCheck provided a minimal counter example for the bug without extra development cost, in favour of additional running time to crunch the test data.

An indispensable feature of QuickCheck is the flexible API mocking mechanism. Apart from enabling software abstractions of hardware interfaces, which is crucial for off-line testing of embedded software, we have demonstrated how it can be used to test for fault tolerance of the complete system. We achieved this by injecting faults in the component mock-ups, against which the system is tested. This allows to test against many different (faulty or correct) component implementations, that in fact do not have to and may not even exist, provided a configurable model of the component is available.

Similar to writing test cases, building the models requires domain knowledge and the ability to interpret the specifications written in English. But by domain experts, it is conceived harder to write a model than to write a single test case. This is not QuickCheck specific, but a general observation that reasoning about all cases at once is harder than reasoning about one specific case. When the hurdle of writing models is overcome, a second challenge, a technical one, is to learn the modelling language. It takes a course and some weeks of experience before engineers feel comfortable with modelling C code in Erlang.

In the presented case study that mimics the real testing effort of the Autosar software, the specification to implementation ratio is 1:1. However, as the code grows towards the complete Autosar library, the size of the model does not grow as fast. For the complete set of Autosar libraries that has been tested commercially [3] the models are only about 20-25% the size of the feature rich C source code under test. Obviously, specifying real Autosar libraries on a larger scale brings additional challenges and benefits compared to what we have described in the paper.

On the benefits side, we can create test suites for really complex library configurations using the same models. This is not so obvious in our example, but an Autosar configuration file consists of thousands of parameters. Inspecting and updating manually written test cases for such a parameter space is really hard. When using models, we can do this automatically.

Another benefit that is hard to see in the paper, but showed valuable in production is the clustering. A typical test case for the CAN bus would cover five different specifications, from driver up to CAN State Manager (CanSM) and CAN Interface (CanIf) [6]. Each such specification is a separate document. There is no detailed specification on how the complete stack should behave. Therefore, engineers manually constructing those tests must have all possible interactions in their head. And indeed, this is error prone. We encoded the six test cases provided in [6] in our format, and executed them against the model. We found that three of the six tests agreed with our model (i.e., these tests would possibly be generated by our model); the other three were not accepted by it. So we could never have generated these tests – but this turned out to be a *good* thing: the three standardised test cases failed to consider a specific transmission recovery scenario. This has been acknowledged as an error, and is filed in Autosar’s Bugzilla for correction in the next release. In other words, being able to compute the complex interactions instead of trying to imagine them, is an advantage.

On the challenges side when dealing with larger models is to get those models generate valid test cases. There we used implementations from different vendors to validate against and to discuss cases in which our models and C implementation did not agree. In this process, ambiguities or underspecifications were found in the standard that lead to differently behaving stacks, whereas they were all arguably correct. This is a strength and weakness of low-level modelling; on the one hand one wants to make all differences on the API level visible, on the other hand, some design decisions that make an implementation faster or less memory intensive, may be tolerated. The QuickCheck methodology had problems in tolerating “that is also fine” approach.

Another effort not visible in small, simple examples is finding good values for test distribution. In the larger models we needed to add certain distributions to get into some interesting states more often. For example, if a large message is sent in chunks over the CAN bus, one wants at least one test that finishes the complete transfer. If the API calls are chosen *just* at random, this is unlikely to happen. Therefore, such behaviour needs to be guided by test distribution values.

## References

- [1] T. Arts & J. Hughes (2016): *How Well are Your Requirements Tested?* In: *2016 IEEE International Conference on Software Testing, Verification and Validation*, pp. 244–254, doi:10.1109/ICST.2016.23.
- [2] T. Arts, J. Hughes, J. Johansson & U. Wiger (2006): *Testing telecoms software with QuviQ QuickCheck*. In: *Proceedings of ERLANG’06*, ACM, pp. 2–10, doi:10.1145/1159789.1159792.
- [3] T. Arts, J. Hughes, U. Norell & H. Svensson (2015): *Testing AUTOSAR software with QuickCheck*. In: *Eighth IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 1–4, doi:10.1109/ICSTW.2015.7107466.
- [4] T. Arts & M.R. Mousavi (2015): *Automatic Consequence Analysis of Automotive Standards (AUTO-CAAS)*. In: *First International Workshop on Automotive Software Architectures (WASA 2015)*, ACM Press, pp. 35–38, doi:10.1145/2752489.2752495.
- [5] AUTOSAR Consortium (2013): *AUTomotive Open System ARchitecture, standard documents*. <https://autosar.org/>.
- [6] AUTOSAR Consortium (2014): *Acceptance Test Specification of Communication on CAN bus – Release 1.0.0*.

- [7] F. Cesarini & S. Thompson (2009): *Erlang Programming*. O'Reilly.
- [8] J. Hughes (2007): *QuickCheck testing for fun and profit*. In: *Proceedings of PADL'07*, Springer, pp. 1–32, doi:10.1007/978-3-540-69611-7\_1.
- [9] S. Kunze, W. Mostowski, M.R. Mousavi & M. Varshosaz (2016): *Generation of Failure Models through Automata Learning*. In: *Second International Workshop on Automotive Software Architectures (WASA 2016)*, IEEE Society, pp. 22–25, doi:10.1109/WASA.2016.7.
- [10] J. Svenningsson, H. Svensson, N. Smallbone, T. Arts, U. Norell & J. Hughes (2014): *An Expressive Semantics of Mocking*. In: *Fundamental Approaches to Software Engineering, LNCS 8411*, Springer, pp. 385–399, doi:10.1007/978-3-642-54804-8\_27.
- [11] J. Tretmans (2011): *Model-Based Testing and Some Steps towards Test-Based Modelling*. In: *Formal Methods for Eternal Networked Software Systems, LNCS 6659*, Springer, pp. 297–326, doi:10.1007/978-3-642-21455-4\_9.

## A The Complete CirqBuff and MBox QuickCheck Model in Erlang

Below we present the complete model discussed in this paper. The snippets used for presentation are slightly abbreviated and simplified for clarity, here the specifications are given in their full form, also to be found at [http://ceres.hh.se/mediawiki/CirqBuff\\_and\\_MBox\\_QuickCheck\\_Models](http://ceres.hh.se/mediawiki/CirqBuff_and_MBox_QuickCheck_Models).

### A.1 File `source_path.hrl`

```
-define(AUTOSAR_SRC, ".").
```

### A.2 File `qc_cb.hrl`

```
-record(cb_state, {ptr, size, elements, data_size}).
```

### A.3 File `qc_cb_setup.erl`

```
-module(qc_cb_setup).
-export([ setup/0 ]).

-include_lib("eqc/include/eqc_c.hrl").
-include_lib("source_path.hrl").

setup() -> eqc_c:start(cb,
  [ {cppflags, "-std=c99-I" ++ ?AUTOSAR_SRC ++ "/include"},
    {c_src, ?AUTOSAR_SRC ++ "/src/cirq_buffer.c"} ]).
```

### A.4 File `qc_cb.erl`

```
-module(qc_cb).

-compile(export_all).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_component.hrl").
-include_lib("cb.hrl"). % Auto-generated
-include_lib("qc_cb.hrl").
```

```

initial_state() -> #cb_state{ elements=[] }.

invariant(S) -> length(S#cb_state.elements) =< S#cb_state.size.

postcondition_common(_S, {call, _, create, _, _ }, _Res) -> true;
postcondition_common(S, Call, Res) -> eq(Res, return_value(S, Call)).

create(Size, DataSize) -> cb:'CirqBuffDynCreate'(Size, DataSize).
create_args(_S) -> [nat(), nat()].
create_pre(S, [Size, _DataSize]) -> S#cb_state.ptr == undefined andalso Size > 0.
create_next(S, R, [Size, DataSize]) ->
  S#cb_state{ptr=R, size=Size, data_size = DataSize }.
create_callers() -> [qc_mbox].
create_return(_S, [Size, DataSize]) ->
  #'CirqBufferType' { maxCnt = Size, currCnt = 0, dataSize = DataSize,
    head = 'NULL', tail = 'NULL', bufStart = 'NULL', bufEnd = 'NULL' }.

push(Ptr, Val) -> DataPtr = eqc_c:create_array(unsigned_char, Val),
  CallRes = cb:'CirqBuffPush'(Ptr, DataPtr), eqc_c:free(DataPtr), CallRes.
push_args(S) -> [S#cb_state.ptr, vector(S#cb_state.data_size, char())].
push_pre(S) ->
  S#cb_state.ptr /= undefined andalso length(S#cb_state.elements) < S#cb_state.size.
push_next(S, _R, [_Ptr, Value]) ->
  S#cb_state{elements = S#cb_state.elements ++ [Value]}.
push_return(S, _Args) -> % 0.
  if length(S#cb_state.elements) < 128 -> 0; true -> 1 end.
push_callers() -> [qc_mbox].

pop(Ptr, DataSize) ->
  DataPtr = eqc_c:create_array(unsigned_char, [0 || _E <- lists:seq(1, DataSize)]),
  CallRes = cb:'CirqBuffPop'(Ptr, DataPtr),
  DataRes = eqc_c:read_array(DataPtr, DataSize),
  eqc_c:free(DataPtr), {DataRes, CallRes}.
pop_args(S) -> [S#cb_state.ptr, S#cb_state.data_size].
pop_pre(S) ->
  S#cb_state.ptr /= undefined andalso length(S#cb_state.elements) > 0.
pop_next(S, _R, _Args) -> S#cb_state{elements = tl(S#cb_state.elements)}.
pop_return(S, _Args) -> {hd(S#cb_state.elements), 0}.
pop_callers() -> [qc_mbox].

prop_cb_length() -> ?SETUP(fun() -> qc_cb_setup:setup(), fun() -> ok end end,
  ?FORALL(Cmds, commands(?MODULE),
  begin {H, S, Res} = run_commands(Cmds),
    aggregate(command_names(Cmds),
      measure(num_commands,
        commands_length(Cmds),
        pretty_commands(?MODULE, Cmds, {H, S, Res}, Res == ok))) end)).

prop_cb() -> ?SETUP(fun() -> qc_cb_setup:setup(), fun() -> ok end end,
  ?FORALL(Cmds, commands(?MODULE),
  begin {_, _, Res} = run_commands(Cmds), Res == ok end)).

```

**A.5 File qc\_mbox.hrl**

```
-record(mbox_state, {ptr, size, elements, cb_state}).
```

**A.6 File qc\_mbox\_capi.erl**

```
-include_lib("source_path.hrl").
```

```
api_spec() -> #api_spec {
  language = c, mocking = eqc_mocking_c,
  modules = [ #api_module{ name = mbox, functions = [
    #api_fun_c{ name = 'CirqBuffDynCreate', ret = 'CirqBufferType_',
      args = [ #api_arg_c{type = size_t, name = size, dir = in},
        #api_arg_c{type = size_t, name = dataSize, dir = in} ],
      classify = {qc_cb, create} },
    #api_fun_c{ name = 'CirqBuffPush', ret = int,
      args = [ #api_arg_c{type = 'CirqBufferType_', name = cPtr, dir = in},
        #api_arg_c{ type = 'void_', name = dataPtr, dir = in,
          stored_type = 'unsigned_char_',
          buffer = {true, "cPtr->dataSize"}} ],
      classify = {qc_cb, push} },
    #api_fun_c{ name = 'CirqBuffPop', ret = int,
      args = [ #api_arg_c{type = 'CirqBufferType_', name = cPtr, dir = in},
        #api_arg_c{ type = 'void_', name = dataPtr, dir = out,
          stored_type = 'unsigned_char_',
          buffer = {true, "cPtr->dataSize"}} ],
      classify = {qc_cb, pop} }]]].
```

```
start_mocking() -> eqc_mocking_c:start_mocking(mbox, ?MODULE:api_spec(),
  [ "Std_Types.h", "cirq_buffer.h", "mbox.h" ],
  [ ?AUTOSAR_SRC ++ "/src/mbox.c" ],
  [ {cppflags, "-std=c99-I_-I_-I_" ++ ?AUTOSAR_SRC ++ "/include"} ]).
```

**A.7 File qc\_mbox\_setup.erl**

```
-module(qc_mbox_setup).
```

```
-compile(export_all).
```

```
-include_lib("eqc/include/eqc_c.hrl").
```

```
-include_lib("eqc/include/eqc_mocking.hrl").
```

```
-include("qc_mbox_capi.erl").
```

**A.8 File qc\_mbox.erl**

```
-module(qc_mbox).
```

```
-compile(export_all).
```

```
-include_lib("eqc/include/eqc.hrl").
```

```
-include_lib("eqc/include/eqc_component.hrl").
```

```

-include_lib("qc_mbox.hrl").
-include_lib("qc_cb.hrl").

-include("mbox.hrl"). % Auto-generated
-include("qc_mbox_capi.erl").

-define(PTR_SIZE, 8). % Change resp. for test running platform (here 64 bit arch.)

initial_state() -> #mbox_state{ elements=[] }.

invariant(S) -> length(S#mbox_state.elements) =< S#mbox_state.size.

postcondition_common(_S, {call, _, create, _, _ }, _R) -> true;
postcondition_common(S, Call, Res) -> eq(Res, return_value(S, Call)).

create(Size) -> mbox:'Arc_MBoxCreate'(Size).
create_args(_S) -> [choose(1, 256)].
create_pre(S, [Size]) -> S#mbox_state.ptr == undefined andalso Size > 0.
create_next(S, R, [Size]) -> S#mbox_state{ptr=R, size=Size }.
create_callouts(_S, [Size]) ->
    ?CALLOUT(mbox, 'CirqBuffDynCreate', [Size, ?PTR_SIZE],
    #'CirqBufferType' { maxCnt = Size, currCnt = 0,
        dataSize = 8, head = 'NULL', tail = 'NULL',
        bufStart = 'NULL', bufEnd = 'NULL'}).

post(Ptr, Val) ->
    DataPtr = eqc_c:create_array(unsigned_char, Val),
    CallRes = mbox:'Arc_MBoxPost'(Ptr, DataPtr),
    eqc_c:free(DataPtr), CallRes.
post_args(S) -> [S#mbox_state.ptr, vector(?PTR_SIZE, char())].
post_pre(S) -> S#mbox_state.ptr /= undefined
    andalso length(S#mbox_state.elements) < S#mbox_state.size.
post_next(S, _R, [_Ptr, Value]) ->
    S#mbox_state{elements = S#mbox_state.elements ++ [Value]}.
post_return(_S, _Args) -> 0.
post_callouts(S, [_Ptr, Value]) ->
    ?CALLOUT(mbox, 'CirqBuffPush', [?WILDCARD, Value],
    if length(S#mbox_state.elements) < 128 -> 0; true -> 1 end).

fetch(Ptr) ->
    DataPtr = eqc_c:create_array(unsigned_char, [0 || _E <- lists:seq(1, ?PTR_SIZE)]),
    CallRes = mbox:'Arc_MBoxFetch'(Ptr, DataPtr),
    DataRes = eqc_c:read_array(DataPtr, ?PTR_SIZE),
    eqc_c:free(DataPtr), {DataRes, CallRes}.
fetch_args(S) -> [S#mbox_state.ptr].
fetch_pre(S) ->
    S#mbox_state.ptr /= undefined andalso length(S#mbox_state.elements) > 0.
fetch_next(S, _R, [_Ptr]) -> S#mbox_state{elements = tl(S#mbox_state.elements)}.
fetch_return(S, [_Ptr]) -> {hd(S#mbox_state.elements), 0}.
fetch_callouts(S, [_Ptr]) ->
    ?CALLOUT(mbox, 'CirqBuffPop', [?WILDCARD], { hd(S#mbox_state.elements), 0}).

```

```
weight(#mbox_state{ptr = undefined}, Op) -> case Op of create -> 1; _ -> 0 end;
weight(_S, create) -> 0; weight(_S, post) -> 5; weight(_S, fetch) -> 1.
```

```
prop_mbox() ->
  ?SETUP(fun() -> qc_mbox:start_mocking(), fun() -> ok end end,
  ?FORALL(Cmds, more_commands(50, commands(?MODULE)),
  begin {H, S, Res} = run_commands(Cmds),
    aggregate(command_names(Cmds),
      pretty_commands(?MODULE, Cmds, {H, S, Res}, Res == ok)) end)).
```

## A.9 File qc\_cluster.erl

```
-module(qc_cluster).

-compile(export_all).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_cluster.hrl").
-include_lib("source_path.hrl").

components() -> [qc_cb, qc_mbox].

api_spec() -> eqc_cluster:api_spec(?MODULE).

setup_mocking() ->
  eqc_mocking_c:start_mocking(mbox, api_spec()),
  [ "cirq_buffer.h", "mbox.h" ],
  [ ?AUTOSAR_SRC ++ "/src/mbox.c", ?AUTOSAR_SRC ++ "/src/cirq_buffer.c" ],
  [ {cppflags, "-std=c99-I_" ++ ?AUTOSAR_SRC ++ "/include"} ],
  components()).

property_mbox_cluster() ->
  ?SETUP(fun() -> setup_mocking(), fun() -> ok end end,
  ?FORALL(Cmds, commands(?MODULE),
  begin {H, S, Res} = run_commands(Cmds),
    aggregate(command_names(Cmds),
      pretty_commands(?MODULE, Cmds, {H, S, Res}, Res == ok)) end)).
```