Model-Based Fault Injection for Testing Gray-Box Systems

Wojciech Mostowski

Computing and Electronics for Real-time and Embedded Systems (CERES)
Halmstad University, Sweden

Abstract

Motivated by applications in the automotive domain, particularly the Autosar basic software standard, we present a technique to improve model-based testing by allowing model-level fault injections. These models are plugged into a larger system as executable components to test it for general tolerance to slightly varying, possibly faulty components or library implementations. Such model execution is possible through applying an automated mocking mechanism and model cross-referencing. Systematic modelling and testing is possible by having comprehensive fault models which both simulate faults and guide the model-based testing procedure towards quicker discovery of these faults. We show the principles of our method on an illustrative example and discuss how it is implemented in a commercial model-based testing tool QuickCheck and applied to a more realistic case study. More generally, this work explores multi-purpose (or meta) modelling – an approach where one parametric model is used for different test targets, like functional testing or safety testing.

Keywords: Model-Based Testing, fault injection, fault models, multi-purpose models, gray-box testing, functional testing, robustness testing

1. Introduction

In an earlier paper [1], using a small but complete case study from an automotive domain, we have described the method and challenges in model-

Email address: `wojciech.mostowski@hh.se` (Wojciech Mostowski)

based testing [2] of low-level C code designed to run in a multi-vendor library scenario [3]. In this paper we elaborate on the specific idea of model fault injection proposed in [1]. This allows us to test software for possible top-level defects stemming from the differences in particular vendor library implementations and inevitable specification drifts [4].

The principle idea and approach is to inject faults into *executable* models, rather than the actual (typically complicated and optimised) code. We plug such fault-injected models into the software to be tested using a *mocking mechanism* [5] – a model-based *stub* of a software component. A model with a fault is much easier to construct than injecting faults in the actual implementation, which we consider to be a gray-box. That is, the source code may or may not be available for inspection or fault injecting changes. Moreover, having full control over the model and its execution-time behaviour, it is much easier to lead the faulty component such that a failure in the complete system is witnessed. In other words, apart from the fault itself, the model also contains information, encoded in operation transitions, on how to reach the fault. A complete system with components substituted by executable models with (potential) faults then undergoes standard model-based testing [2]. The end result can be a nevertheless (despite injected faults) successful test indicating that the overall system design is resistant to possible library *glitches* – faults classified as "features", or benign specification divergence. A failing test, on the other hand, provides a witness to a failure that should be addressed in the system design to account for a particular glitch.

This further scales to the idea of multi-purpose modelling, where one, specification-parametric model is utilised for several testing purposes. In this work we concentrate on functional, component compatibility, and robustness (fault resilience) testing. In the automotive context one would also consider safety testing, which typically targets a different set of properties, but ones that still relate to the functional properties. Hence we consider the use of a common model for both justified.

Fault models can be constructed by hand to introduce particular kinds of faults or larger, configurable family of faults. Alternatively, in some simple cases, fault models can be generated with automata learning techniques [6] applied on known faulty or specification non-compliant implementations.

The particular model-based testing tool that we use in our work is QuickCheck [7] from Quviq. The capabilities and features of the tool that we leverage in our work are property-based models, component/model *cluster-*

ing, the mocking mechanism [5], controllable data generators, and, for our project case studies in the automotive domain, the C programming language integration. For the purpose of the presentation, we use a simple coffee machine example. However, we also validated our approach by experimenting with the case study presented in [1]. More generally, this work is part of the AUTO-CAAS project [8], where techniques to aid and automate model-based testing of automotive software adhering to the Autosar [3] industry standard are investigated.

The rest of this paper is organised as follows. In Sect. 2 we describe model-based testing and its particular implementation in QuickCheck, while in Sect. 3, first briefly introducing Erlang, we show how to specify our running example in QuickCheck. Section 4 begins to introduce our main idea by describing how models can be plugged into a system using the mocking mechanism, and how models can be cross-referenced with the model operator to modularise this process. In Sect. 5 we dive into the methodology of modelling faults and guiding test generation with these models. Sect. 6 briefly discusses the application of our technique to the case study from [1]. The paper concludes with Sect. 7 on related work and Sect. 8 summarising the paper and discussing future work.

2. Model-Based Testing with QuickCheck

In classical testing, a set of predefined unit tests – collections of input data with the associated expected outputs – is executed on a System Under Test (SUT) to establish the correctness of the implementation. Model-Based Testing (MBT) [2] is a technique where a behavioural model is first constructed which encapsulates the SUT behaviour, possibly with abstractions necessary to simplify the model w.r.t. the actual implementation. This model is then traversed to collect (a selection of) valid execution traces. The traces are executed on the SUT and the outputs are compared against the model using a suitable conformance criterion, e.g., IOCO, see [9]. Typically, the models are Labelled Transition Systems (LTS) [10], where the states are the abstractions of the implementation states, and labels are operations (functions, procedures, or methods depending on the given programming language nomenclature) from the Application Programming Interface (API) along with the allowed inputs and valid outputs. Since exhaustive traversal is not feasible for realistically sized systems, a limited set of execution traces is selected using either randomness or constraint solving [11] to generate inputs representing the model artefacts (data or operations). In either case, model traversal methods need to guarantee sufficient coverage [12], at least statistically, of the model and the corresponding coverage of the SUT.

Model-Based Testing is specifically useful for substantially sized and configurable systems or libraries. Configurable systems require separate sets of unit tests for each configuration which in practice can comprise of thousands of single test cases. On top of that, the collection of these unit tests needs to be managed and maintained. A concise model, on the other hand, represents an arbitrarily large set of tests which are generated on the fly for a given system configuration. 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 or abstraction. MBT is typically applied to black-box systems provided that the external API is given, although white-box systems are equally well testable using MBT. In this case the process can be further supported by structural analysis of the code.

2.1. Property-Based Testing

In our work we use a particular instance of MBT, a so-called *property-based testing*. Properties are boolean expressions attached to model transitions that define the correctness of the output produced by the corresponding operation in SUT. The property is defined not only in terms of the actual SUT inputs and outputs, but also in terms of the current state of the model that is traced along the SUT during test execution. During testing, the properties effectively become *test oracles* as they are typically called in unit testing. This is an extension of the original idea of property-based testing [13], where properties were defined over single operations, typically expressed as quantifiers over input data. Here, the quantification ranges also over possible traces of operations [14].

2.2. Example

As an example, consider a model of a coffee machine shown in Fig. 1. The coffee machine is capable of serving arbitrary coffee amounts (e.g., to fill a large jar) up to some M number of units fixed during initialisation. The states of the model represent the abstracted number of deposited coins. The state marked with 0 represents the situation when there are no coins, while state N is one where there is a at least one coin deposited. Differentiating precisely how many coins are deposited is done by manipulating the model

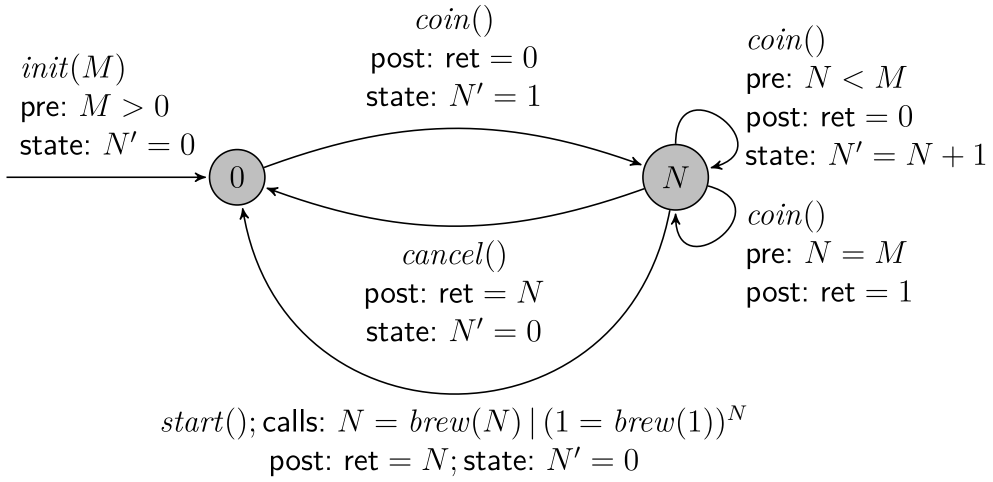

Figure 1: A state-based model of a coffee machine.

state variable N. The machine has two outputs; On operation *start* a corresponding number of units of coffee is served depending on the number N of deposited coins. Furthermore, the machine returns coins in two situations. On operation *cancel* it returns all N coins, and on operation *coin* when the coin deposit is full (coin overflow) it returns 1 coin. In the latter case the previously deposited coins are retained and the machine can still serve coffee. Finally, the *init* operation, parametrised with capacity M, represents the coffee machine initialisation or reset.

For each of the coffee machine operations, with pre: we specify the precondition which has to be satisfied for the transition to be enabled. With post: we specify the postcondition that should hold when transition completes, effectively this is our test. The state: tag defines the model state update, here expressed simply as an assignment to the new value of the state N' in terms of the old one N. The special value ret represents the operation return value from the SUT. Within postconditions this actual return value can be compared to the expected one, and in state update expressions it can be used to update the model state accordingly. The models have the following defaults. Absent transitions denote that they are disabled altogether, e.g., *start* is disabled in the 0 state. Unspecified pre- and postconditions default to true. Finally, a missing state update expression defaults to $N' = N$ indicating no change in the model state.

Note that this model is still very simplified, in particular it does not account for accepting bank notes and giving change, or the possibility of the unit being empty, or the coin deposit box being full, or any other exceptional condition. However, the calls: tag does specify a requirement on an internal call. A particular operation *brew* of the brewing unit should be called when the *start* operation is serviced, we elaborate on this shortly. In other words, internal event occurrences can be specified.

To generate test cases for a coffee machine implementation, this model is traversed to produce valid call traces and expected results, e.g., the following two, for a model instance with $M = 2$:

$$\begin{aligned}
T_1 &: \mathit{init}, \mathsf{ret}_0 = \mathit{coin}, \mathsf{assert}(\mathsf{ret}_0 = 0), \mathsf{ret}_1 = \mathit{cancel}, \mathsf{assert}(\mathsf{ret}_1 = 1)\\
T_2 &: \mathit{init}, \mathsf{ret}_0 = \mathit{coin}, \mathsf{assert}(\mathsf{ret}_0 = 0), \mathsf{ret}_1 = \mathit{coin}, \mathsf{assert}(\mathsf{ret}_1 = 0),\\
&\qquad \mathsf{ret}_2 = \mathit{coin}, \mathsf{assert}(\mathsf{ret}_2 = 1), \mathsf{ret}_3 = \mathit{start}, \mathsf{assert}(\mathsf{ret}_3 = 2)
\end{aligned}$$

The subsequent *actual* results of the operations executed by the SUT are referred to with ret_X and the expressions within the assert statements are the concrete postconditions instantiated from the current model state. A failed assertion indicates the failure of the corresponding test. Traces that the model does not define as valid will not be generated, but rejected and not included in the test suite, for example:

$$\begin{aligned}
R_1 &: \mathit{init}, \mathsf{ret}_0 = \mathit{cancel}, \mathsf{assert}(\mathit{true})\\
R_2 &: \mathit{init}, \mathsf{ret}_0 = \mathit{start}, \mathsf{assert}(\mathit{true})
\end{aligned}$$

The models can be defined formally using functions and relations over the model states and operations (transition labels) along with their inputs and outputs, using a suitable formalism for Labelled Transition System, see, e.g., [10]. Intuitively, pre: and post: tags are predicates over the state of the model, transition label, operation inputs and outputs (output referred to with the ret place-holder only for post: predicates). The state: tag is a relation between two states. The calls: part of the model cannot be expressed so trivially, a simple call specification is formally expressed with a tuple (o, a, r) where o is the operation, a is the argument (or a tuple of arguments), and r is the expected return value. Hence, the calls: part of the specification $N = \mathit{brew}(N)$ is formally a tuple (brew, N, N) which states that the return value of *brew* should be the same as the input argument. The singleton call tuples can be then combined to express a richer call language using suitable constructs (alternative $|$, repetition $\cdot^X$, optional calls ?, etc.), see e.g. [5].

Considering it rather straightforward, we assume such formal definition for models later on in the paper when we explain model-based fault injection.

2.3. QuickCheck

Our tool of choice is QuickCheck developed by Quviq [7, 15] due to its applicability in real application scenarios that involve complex code written in the C programming language. QuickCheck has been used in large scale software testing [7], in particular to specify and test Autosar basic software implementations [4] and in [1] we presented a specially saturated model and implementation representative of the effort to comprehensively test the Autosar libraries.

QuickCheck models are written as functional programs in the Erlang programming language [16]. These models follow a predefined *callback* structure and API to form a behavioural description of a given component to be tested. A model defines the *symbolic execution* state of the component under test and operation specifications – preconditions and postconditions. The role of the precondition is to specify the execution eligibility of a given operation under the current model state and operation parameters. The postcondition defines the expected result or oracle, i.e., the property to be tested.

The properties that can be specified and checked during test execution are more elaborate than the property-based model example given above.[1] Namely, apart from simple checking of operation results for particular values also *call-out* specifications can be given. These specifications are given in a process algebra-like language [5] and describe the calls to other parts/operations of the SUT that the operation is *allowed* or *required* to make. The tracing of such calls during testing to check against call-out specifications is done by mocking the calls in question following the gray-box testing philosophy, see Sect. 2.5. In our example in Fig. 1 we have specified that the *brew* operation should be called either N times with 1 unit of coffee each, or one time with N units. Such a specification provides flexibility much needed in component based implementations – should the brewing unit of the coffee machine be replaced with one that does not allow requesting arbitrary number of units our specification should allow this letting the implementation

[1]In fact, QuickCheck offers much more than what we state here, temporal properties and concurrent testing are the further obvious supported features. They are, however, not crucial for the main contribution of the presented work.

make this choice. In fact, one could strive to make the call-out specification in our simple model of the coffee machine even more flexible:

$$\mathsf{calls}{:}(R_U = \mathit{brew}(U))^K; \mathsf{post}{:}\ \mathsf{ret} = K \cdot R_U \wedge U \cdot K \geq N$$

to state the brewing unit should produce at least N units of coffee using some single brewing unit U K times.

2.4. Components and Clustering in QuickCheck

The simplest approach to MBT assumes that the SUT is a complete black-box with a public API through which the SUT interacts with the model-based testing procedures. Nevertheless, in component-based systems with multi-layered libraries (e.g., communication protocol stacks [17]) an approach where system parts or components could be tested in isolation might be preferable for reasons of scalability, but also to more easily identify the source of the problem when tests fail.

To this end, QuickCheck supports model and test *modularisation* that enables specifying and testing single software components separately before the complete system is tested for the correct top-level behaviour. On top of that, it is also possible to specify and test intra-component interactions through the already mentioned call-out specifications. This enables, among other things, to check that components behave correctly in terms of the API use, e.g., that no private API parts are ever called by another component. The particular mechanism to achieve this in QuickCheck is called *clustering*, within a cluster a set of components is defined, and within this set API interactions are checked. This includes checking if particular operations within the cluster are invoked as required or are *not* invoked as prohibited by the specification, or that operations are always invoked with their preconditions satisfied. The existing infrastructure for component clustering allows us to employ our model-based injection technique without additional tool implementation cost, we explain this in more detail towards the end of Sect. 4 and in Sect. 5.

2.5. Gray-Box Testing

In a typical MBT scenario where a black-box system is tested a connecting interface has to be built to interact with the SUT from the MBT tool, where the two may not even reside on the same computing platform. In the worst case, for highly integrated embedded systems without direct software interface, one has to built a complete infrastructure to make the connection between the MBT tool and SUT, see e.g. [18]. Apart from providing the API access to the SUT, such infrastructure would also translate the high-level model data generated by the MBT tool to concrete platform-specific inputs for the SUT, and vice-versa, translate the concrete outputs (results) of the SUT back to the model domain to evaluate the test result correctness. The complete API access infrastructure and translation facility is typically called an adapter.

For SUTs written in C, QuickCheck provides an integrated flexible interface that allows to refer to the C function calls, including the call parameters, directly in the model and correspondingly execute the C implementation by creating a suitable adapter on the fly. This is possible for systems both with and without the source code available. In the former case, QuickCheck can be also instructed to compile the C code in a specific way before testing. This avoids the overheads associated with manual adapter construction, and provides the possibility for *gray-box* testing. That is, using the same facilities, single *internal* C components can be automatically wrapped or stubbed, the SUT together with the wrappers recompiled, and then component interactions and internal API behaviour can be tested by querying the wrapper.

2.6. Testing with QuickCheck

The general philosophy of QuickCheck is to run several short tests generated from models to detect problems early, rather than creating large and complex testing scenarios for which failures are difficult to analyse. Nevertheless, if a large test case is generated before a bug is exhibited for the first time, QuickCheck offers a mechanism to *shrink* the test case and repeat the test to come up with a smaller, possibly minimal, test case that leads to the same failure. The shrinking capability is built into data generators for the corresponding types, and applied recursively over the structure of the data. For example, shrinking a primitive data type can mean trying a boundary value (like 0 for unsigned integers) instead of the originally generated one. For complex types, a value that is structurally smaller can be tried, for example a prefix of the originally generated list. Consequently, the effort of debugging is also reduced, and as stated above, the process can be applied to a single component at a time further reducing the overall testing effort.

3. Example in QuickCheck

3.1. Erlang

As said, the models in QuickCheck are Erlang programs [16]. A brief overview of Erlang to facilitate understanding of its programs is the following. Erlang is an untyped (the necessary type compatibility is only checked at run-time) functional language with built-in concurrency based on message passing processes. Programs are divided into modules (name spaces) and consist of a collection of functions defined with (possibly higher-order) functional expressions by pattern matching on parameters. The expressions are built with numbers, atoms (which can be viewed as symbolic values or enumerations), lists denoted with `[]`, and tuples denoted with `{}`. The following simple example illustrates these concepts in a compact form, it filters a list of tuples to find ones with an `active` tag and places the corresponding data elements from such tuples (of unspecified type) to a new result list:

```
-module(filter). -compile(export_all).

active([], Result) -> Result;
active([{active, Data} | Tail], Result) ->
    active(Tail, Result ++ [Data]);
active([_ | Tail], Result) -> active(Tail, Result).
```

The ++ is a list concatenation operator, and the _ character is used to mark unused or irrelevant parameters in a given context. This is mostly conventional, a regular name can be used like `Head` in this case, but the compiler would give an ‘unused variable’ warning then. The `active` function is defined by three pattern matching cases separated with semicolons, the complete definition is terminated with a dot. With this, we can then call our fully qualified function from any other Erlang context to get a filtered list, e.g., `filter:active([{other, none}, {active, 2}], [])` produces `[2]`. Note that our input list is not consistently typed, this is not a problem in Erlang, expression types can be arbitrary until the point of run-time valuation of an expression in a given type context.

This example also shows the concept of atoms, in this case these are `other`, `none`, and `active`. Atoms are simply recorded in the atom dictionary on the first use and can be freely used from that point on. This is a sort of global, on demand enumeration type, but there are some more technical details to it of course [16]. In particular, the module name `filter` and function name `active` are also atoms (the `active` atom is thus used overloaded for both the function name and the tag in the list), as are **`true`** and **`false`** boolean constants. Record structures can be also used, however, records are nothing more than specifically tagged tuples with named indexing of tuple values and the `.` and `#` operators are helpers to select and update tuple items. Finally, macros can be defined and used, these start with `?` and have an all upper case name. Records and macros are used often in QuickCheck models, we explain the particular instances below. The Erlang processes are briefly explained along with the example in Fig. 3 to follow.

3.2. QuickCheck Model in Erlang

For our coffee machine example the QuickCheck model is given in Fig. 2. It follows the model given earlier in the state diagram in Fig. 1, where the different tags that we used before are now represented by specific callback functions. For example, `coin_next` (ls. 17–18) defines the next model state for the *coin* operation given earlier by the state: tag. It is expressed with an Erlang conditional construct **`if`** for which the evaluation result is the expression on the right hand side of the first true expression on left hand side in the `·->·` relation. If the `n` field of the model state `S#cm_state`, stored in local variable `N`, is smaller than the capacity `m` (also recorded in the state), then the new state is `S#cm_state{n=N+1}`. This expresses that the new state retains all its fields from `S`, apart from `n` which is updated to `N+1`. Otherwise, when the first **`if`** condition is not met, the state `S` remains unchanged.

For the time being, we skip the calls: tag. This is also the reason why the specifications for `cancel` (l. 22–) and `start` (l. 29–) are identical so far. For the actual operations the QuickCheck model specifies the calls to the implementation (the SUT), through the basic callback of a given operation. For example, the function `coin` (l. 14) specifies that the operation is implemented in an Erlang module `cm` (for Coffee Machine) with the same function name `coin` and no parameters. For operations that require parameters, a generator for that parameter has to be specified through the `_arg` callback. For the `init` operation `init_arg` (l. 8) specifies with `[nat()]` that a one element list (the operation has one parameter) with a natural number should be generated.

The `_return` callbacks specify the expected results of operations in terms of the current model state and the operation arguments. Essentially, these callbacks are the right hand sides of the post: tag relations in Fig. 1. Since all our postconditions in Fig. 1 have the same shape, i.e., they are equalities, we

```
-module(cm_model).
-record(cm_state, {n, m}).

initial_state() -> #cm_state{}.
postcondition_common(S, Call, Res) ->
    eq(Res, return_value(S, Call)).

init_args(_S) -> [nat()].
init(M) -> cm:init(M).
init_pre(S, [M]) -> S#cm_state.n == undefined andalso M > 0.
init_next(S, _R, [M]) -> S#cm_state{n=0, m=M}.
init_return(_S, [_M]) -> ok.

coin_args(_S) -> [].
coin() -> cm:coin().
coin_pre(S, _A) -> S#cm_state.n =/= undefined.
coin_next(S, _R, _A) -> N = S#cm_state.n,
  if N < S#cm_state.m -> S#cm_state{n=N+1}; true -> S end.
coin_return(S, _A) ->
  if S#cm_state.n < S#cm_state.m -> 0; true -> 1 end.

cancel_args(_S) -> [].
cancel() -> cm:cancel().
cancel_pre(S, _A) ->
    S#cm_state.n =/= undefined andalso S#cm_state.n > 0.
cancel_next(S, _R, _A) -> S#cm_state{n = 0}.
cancel_return(S, _A) -> S#cm_state.n.

start_args(_S) -> [].
start() -> cm:start().
start_pre(S, _A) ->
    S#cm_state.n =/= undefined andalso S#cm_state.n > 0.
start_next(S, _R, _A) -> S#cm_state{n = 0}.
start_return(S, _A) -> S#cm_state.n.
```

Figure 2: The QuickCheck specification for the coffee machine.

specify them all in QuickCheck with just one common postcondition (ls. 5–6). It states that for all operations (`Call`), the actual return value (`Res`) should be equal to the specified one (`return_value`).

For this example we have chosen to show an Erlang implementation of the coffee machine designed as a concurrent Erlang process, see Fig. 3. However, it could equally well be a C implementation in which case we would simply forward the operation calls to the QuickCheck C interface instead. The implementation code in Fig. 3 is intentionally rather opaque and only briefly explained in the caption to illustrate the applicability of our technique to source code that is difficult to modify or absent all together.

Finally, to specify a test to be generated and run by QuickCheck one gives a property like the following:

```
property_cm() -> ?FORALL(Cmds, commands(cm_model),
    begin {_, _, Res} = run_commands(Cmds), Res == ok end).
```

The `?FORALL` macro is just a wrapper around Erlang's `foreach` call on lists to simplify the syntax. It binds the trace of operation calls generated from the model with `commands(cm_model)` to the variable `Cmds`. This single operation trace is then run on the SUT with `run_commands`. The collected results (some of which we ignore here with `_`) contain, among other things, the history of intermediate results in the trace or statistics and the distribution of the generated data. Here, we are only interested in the fact `Res == ok` indicating that all postconditions during trace execution evaluated to **`true`**. Based on this property, by default 100 random execution traces are generated and run on the `cm` implementation. For test failures (violations of postconditions), QuickCheck applies its shrinking procedure and reruns the test in the attempt to minimise the failing execution trace before reporting it.

4. Model-Based Fault Injection

The coffee machine example, although very simple, is representative of a typical problem in software engineering. Namely, one would like to devise a testing procedure that allows to state, with high confidence, that the coffee machine will behave correctly when working with different brewing units from different suppliers. This should range from very simple cases of poorly manufactured units that exhibit general failures, e.g., that in 10% of cases no coffee is actually produced, to more complex scenarios of brewing units that,

```
    -module(cm).
 2  -export([init/1, coin/0, cancel/0, start/0, loop/2]).

 4  init(M) when M > 0 -> case whereis(cm) of
      undefined -> register(cm, spawn(cm, loop, [0, M])), ok;
 6    P -> P ! {reset, M}, ok end.

 8  coin() -> whereis(cm) ! {coin, self()}, receive R -> R end.
    cancel() -> whereis(cm) ! {cancel, self()}, receive R -> R end.
10  start() -> whereis(cm) ! {start, self()}, receive R-> R end.

12  loop(C, M) -> receive
      {coin, P} ->
14      if C < M -> P ! 0, loop(C+1, M);
           true -> P ! 1, loop(C, M) end;
16    {cancel, P} when C > 0 -> P ! C, loop(0, M);
      {start, P} when C > 0 -> P ! brewer:brew(C), loop(0, M);
18    {reset, NM} -> loop(0, NM);
      {_, P} -> P ! undefined, loop(C, M) end.
```

Figure 3: Process based coffee machine implementation in Erlang. The header specifies module name and the top-level interface to expose. The main process control function `loop` is actually internal, but needs to be exported due to specifics of implicit referencing of functions in Erlang. The initialisation function (ls. 4–6) checks if a process named `cm` is already registered. If not, a new one is spawned and registered under the `cm` name. On spawning the main control function `loop` is specified along with its initial parameters – `0` coins and capacity `M`. If the process is already running, a reset message is sent (operator `!`) to its process identifier `P` (l. 6). Messages sent to the process are dispatched by the main control loop with the **`receive`** expression (l. 12) that pattern matches on the possible messages. The `reset` message (l. 18) simply restarts the loop with `0` coins and the newly specified capacity `NM`. The other messages are dispatched similarly and in essence update the current coin value in the new call to `loop` according to the required functionality, which should be now self-explanatory. Some messages also need to handle return values. Along with the message, the calling process identifier `P` is also received (ls. 13–17), to which the result is sent in return. For example, on `cancel` the amount of deposited coins `C` is sent to process `P` with `P ! C` (l. 16). The `start` message (l. 17) has an external interaction – function `brew` in the `brewer` unit is called to get the result. Finally, unrecognised messages (l. 19) send an **`undefined`** value to the caller and continue the `cm` process without a change. The definition for the top-level interface follows the same pattern for all operations (ls. 8–10). First, the cm process identifier is looked up (**`whereis`**), a corresponding message is then sent to that process along with process identifier of the caller (**`self`**), and finally the return value `R` is received which becomes the result of the top-level function.

due to some mechanical specifics, cannot always produce the exact requested amount of coffee, but nevertheless try to fulfil the request as close as possible.

To approach this problem we turn to the well-known technique of fault injection. In the testing vocabulary the term *mutant* [19] is often used with a seemingly similar meaning. However, mutants are used to establish that the set of test cases is sufficiently complete, and are, generally, temporary artefacts. In mutation testing a SUT is populated with random faults (e.g., swapped statements, flipped conditions, etc.), the said mutants. The SUT is then run through the set of test cases. The test case that fails *discovers* the injected fault establishing the test case set to be sufficient to discover this particular fault. The failing test case is said to *kill* the mutant. Mutants that do not trigger test failures (are not killed), prove the set of test cases to be insufficient for testing the particular SUT.

We strive to accomplish something different. First, we want to avoid injecting faults into the SUT's actual implementation code, mostly for the reasons of uncertain presence of the source code (or parts of it), or its high complexity. Instead, we replace complete components in the SUT with *executable* models and inject faults there. Second, we work on the initial assumption that by using MBT our test coverage is already sufficient and we test for the resistance of the top-level software against faults in low-level components. That is, a successful test indicates that the software is designed to withstand faults or inconsistencies of particular kind in components, while a failure indicates not necessarily a bug in the software, but rather an inability to deal with an underlying software component exhibiting a particular fault (or feature). In the latter case three possible actions can be taken: (a) components that are proved to exhibit such behaviour could be excluded from consideration when looking for a library supplier; (b) the library code and specification could be reviewed and in case of specification non-compliance the library should be fixed; (c) under the conclusion that the library is nevertheless implemented according to the specification (the faulty behaviour should be considered a *feature*) the top-level software could be modified to account for components of this kind. Apart from the first case of library exclusion, the testing of the library and the complete system needs to be repeated.

The concept of the model-based fault injection is shown in Fig. 4. The three example variants of one component are executable models with different injected faults or features. These models of the component replace the complete original component in the *System* implementation through the mocking

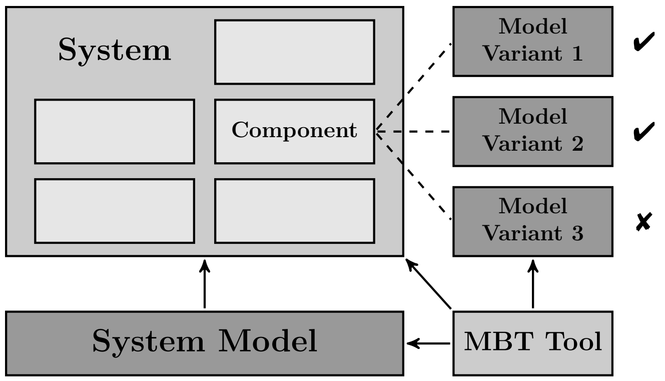

Figure 4: The general concept of model-based fault injection.

capability that the *MBT tool* provides. For each of the component variants, the *MBT tool* runs tests on the complete *System* based on its specification given in its high-level *System Model*. The system with the first two example variants is tested successfully (✔), while the third variant of the component exhibits a top-level system failure (✘) under the faulty component behaviour.

In a more general case, the model variants for a component do not have to represent low-level singular faults, but can be models for different testing contexts. For example, one variant can be a model for functional testing, while another variant can be a specifically constructed model for safety or robustness testing. In this scenario, the variants relate to multi-purpose models mentioned in the introduction.

Test coverage is not given *just* by applying MBT, it depends very much on the model traversal and data generation strategies.[2] Both might have to be adapted to improve coverage and confidence in our technique. The information about possible faults embedded in a model of a faulty component can support the process by guiding the test generation process to reach faults earlier. Namely, when a top-level operation is specified to invoke a given library operation, the model of the library is consulted (through an additional precondition) to decide if the library is likely to produce a fault (immediately, or potentially in the future). If it is not, the top-level operation is rejected from test generation and the next one is selected. To achieve this, the com-

[2]Although probably even the most naïve model traversal and dumb data generation (like fixed values instead of random ones) still give substantially better test coverage than a fixed set of manual test cases.

ponent model has to contain information about faulty (or at least undesired) behaviours, we call it a *fault model*. In other words, we make faults (or, more generally, behavioural features) to be a permanent (but optional/selectable) artefact in the model. This is in contrast to mutants mentioned earlier, that are introduced temporarily, and typically in the running code rather than the model (though not necessarily). We elaborate on this in Sect. 5.

4.1. Mocking

The prerequisite for achieving model-level fault injection is the ability to swap an actual component in the SUT with its model, provided it is specified as executable.[3] A general purpose technique for substituting code is called *mocking* or *stubbing* and it is typically used when a component needs to be emulated or replaced with a stand-in for certain purposes. In particular, testing during development of embedded systems would involve the use of mock-ups – other parts of the system, sensors, actuators etc. A mock-up is a software component with precisely the same interface as the original, but with simplified and artificial functionality. For our example, we would substitute the coffee brewing unit with a software component that brews virtual coffee and reports on the amount of brewed coffee.

An obvious basic approach to mocking is by manual construction of the corresponding software component. However, it scales poorly and lacks flexibility. For this reason, mocking frameworks have been developed where one specifies the components to be mocked using a specific language, and then the actual mock-ups are generated by the tool. One example of such a framework is Mockito from Google that allows to mock Java components [20].

The very basic role of the mocking language is to allow easy specification of which interface parts should be mocked, what should be the input-output behaviour of this interface, and the generation of the corresponding code. The specification can be a simple input-output value relation, but can also involve an elaborate behaviour description through a properly defined stand-in component, i.e., mock-up code with some level of emulation precision. Another possible functionality of a mocking framework is the ability to trace calls to the mocked interface.[4] For each mocked operation the runtime mock-

[3]If an interface or adapter between the model and implementation data is required, like described in Sect. 2.5, we assume its existence.

[4]In fact, tracing *any* calls within a SUT is very valuable for testing, however, doing it only for the mocked interface is technically much easier due to the readily available

ing infrastructure can verify that the specified calls have been made and that they have been made in a specific order. In our coffee machine state diagram in Fig. 1 we have specified that the brewing unit is called in one of two possible ways. The tracer can check that when the coffee machine implementation is under test. The correct execution order of the mocked calls becomes another test requirement – calls made not in sequence should fail the test, even if the overall system behaviour in terms of returned results is correct. This is a way to check that the APIs, even though internal, are used according to their specifications.

The mocking language of QuickCheck integrates both the behavioural specification of the mock-up (the stub's input-output behaviour) with the call sequencing specification. Effectively, this provides a process algebra-like language for mocking with well defined semantics, for details see [5]. These mocking specifications are given in the call-out parts of the model to indicate that underlying calls to the components should be checked. The particular API to be mocked is defined separately and tells the QuickCheck test executor which parts of the SUT should be stubbed for the test, and the call-out specifications give semantics to these stubs.

Let us model in QuickCheck the first option of the calls: specification from the state diagram given in Fig. 1, the $N = brew(N)$ call. First, we specify that the brewing unit should be mocked by listing the corresponding API of the `brewer` module:

```
api_spec() -> #api_spec{ language = erlang, modules = [
  #api_module{ name = brewer, functions = [
    #api_fun{ name = brew,  arity = 1 }]}]}.
```

We then add a call-out specification to the *start* operation to define the expected call to the brewing unit:

```
start_callouts(S, _A) ->
  ?CALLOUT(brewer, brew, [S#cm_state.n], S#cm_state.n).
```

The `?CALLOUT` macro is a syntax wrapper to create an involved Erlang tuple expression internal representation of call-outs, which would be hardly readable in its expanded form. This specification gives a rather straight-cut behaviour for the brewing unit. It never fails and always produces the required amount of coffee – the result is equal to the parameter, which is the

instrumentation code.

```
  -module(brewer_model).
2 initial_state() -> empty.

4 brew(C) -> brewer:brew(C).
  brew_args(_S) -> [nat()].
6 brew_pre(_S, [C]) -> C > 0.
  brew_return(_S, [C]) -> C.
```

Figure 5: A QuickCheck model for the brewing unit component. Note that this model does not (yet) have any meaningful state, but an initial state still has to be given (l. 2), we just state `empty`.

number of deposited coins. It is not surprising that the top-level system tests for the coffee machine generated by QuickCheck all pass. It shows that the coffee machine implementation, see Fig. 3 again, can at least cope with a *vanilla* brewing unit. In this example we have given a specification for a single call. The other possibilities in the mocking language are, e.g., exclusive alternatives, or sequences (also unordered) of calls.

4.2. Mating Mock-ups with Models

What we have presented so far is an existing method and technology for mocking parts of the system in the context of contemporary testing as well as MBT. Note the very direct character of providing the stub behaviour – it is given by relating the inputs and outputs of the operation explicitly.

In a more general setting of MBT where models are modular and given per each component, the mocked component would already have a specification for isolated testing with MBT. For our brewing unit, a QuickCheck model is given in Fig. 5 that specifies the vanilla version of the unit. A trivial implementation of the brewing unit – "`brew(C) -> C.`" – clearly satisfies the properties stated in the model and this can be easily tested.

The objective of our approach is to plug-in the model in the execution of the system. Essentially we want to be able to say: mock an implementation component and its function(s) in such a way that it behaves like the associated model. In the mock-up description we do not state any particular behaviour, but forward the request to the model, i.e., we *cross-reference* the model. For our state diagram model of the coffee machine from Fig. 1, we would like to turn the calls: specification $N = brew(N)$ into:

$$\mathsf{let}\ R = \mathsf{model}(brew, N)\ \mathsf{in}\ R = brew(N)$$

in which case the fact that $R = N$ should hold (i.e., the brewer always prepares the N requested units of coffee) is no longer present in the specification. It is the responsibility of the model of the brewing unit to state that with $\mathsf{ret} = N$ in its postcondition.

4.3. The model *Operator*

A bit more formally, the model operator is defined as follows. First, the state N of the top-level model (our coffee machine model) becomes a tuple (S_C, S_M) which holds the original state of the model for C and the state of the mocked component M called by C. Then, given the current extended state of the model (S_C, S_M), the model operator over S_M and a single call specification tuple (o, a, r) is defined as:

$$\mathsf{model}(S_M, o, a, r) = (S'_M, P, NC),$$

where (S'_M, S_M) satisfies the $o.\mathsf{state}$ relation in model M, P is the evaluated precondition $o.\mathsf{pre}$ of operation o with arguments a in model M, and NC is a new call specification $(o, a, o.\mathsf{ret})$ with $o.\mathsf{ret}$ being a return value of operation o in model M satisfying the postcondition $o.\mathsf{post}$ in M.

The new call specification NC replaces the old one, while the precondition P and the new state S'_M of model M are incorporated into model C in the following way. The precondition of the associated operation in model C is conjoined with P and the new tuple state in C becomes (S'_C, S'_M) where S'_C is the new state from the state: relation of the model C.

Intuitively, the model operator achieves the following. For the specified mock-up call o, the return value for that call is looked up in the model M for o, and the two model states for components C and M are updated simultaneously. The state of component C is used for generating the top-level tests, while the state of M is used to mock the operation o. Additionally, the precondition of the mocked operation o is checked in the model M to ensure that in C tests are generated such that the underlying operation o follows the API requirements according to the model M. Effectively, this excludes tests in C that would violate the API of M.

4.4. The model *Operator in QuickCheck*

The model operator to connect a mock-up of an operation with its model is actually readily available in QuickCheck, although it has a slightly different form. Its initial intention was to allow for intra component API conformance checking for clustered components by checking associated preconditions between linked models. However, the underlying mechanism is flexible enough to allow us to accomplish full and seamless model execution for the mocked operations. Similarly as before, this is achieved through the call-out specification. For our `start` operation of the coffee machine we can give the following call-out specification for the brewing unit:

```
start_callouts(S, _A) ->
  ?MATCH(Res, ?APPLY(brewer_model, brew, [S#cm_state.n])),
  ?CALLOUT(brewer, brew, [S#cm_state.n], Res).
```

The `?APPLY` construct makes the model transition `brew` for the brewer unit specified in `brewer_model` and records the return result `Res` of that transition through `?MATCH`. Effectively, this is our model operator. The value `Res` is then associated with the specified mock-up call to `brew` in the coffee machine. As before, the `?APPLY` and `?MATCH` Erlang macros are just syntactic sugar for complex tuple expressions to build the internal specification structures for QuickCheck.

The remaining part is to keep track of two model states simultaneously and to check corresponding preconditions across models. This is done simply by putting the two models into one cluster:

```
components() -> [ cm_model, brewer_model ].
```

which directs QuickCheck to make the mentioned model transitions and precondition checking for the brewing model behind the scenes when tests are generated and executed for the coffee machine model.

5. Modelling Faults

The presented technique for mocking components in MBT using their models (effectively making these models executable and pluggable into the SUT) allows us to fully detach the mock-up interface definition from its run-time behaviour. The behaviour is now fully and solely delegated to the model of the mocked component and this is where we intend to inject faults to perform robustness testing.

Suppose we want to test our coffee machine when the brewing unit fails to deliver the requested amount of coffee every 5 uses when at least 2 units were requested. It then delivers one unit of coffee less than requested, but always at least one unit. Our so far state-less model for the brewer now needs to keep track of the number of uses and depending on this and the requested units of coffee we model the result of operation `brew` differently. The state of the model becomes a number and the `brew` operation is now modelled in the following way that reflects the stated fault characteristics:

```
initial_state() -> 0.

brew_return(S, [C]) ->
    if S==5 andalso C>1 -> C-1; true -> C end.
brew_next(S, _R, [C]) ->
    if C>1 -> if S==5 -> 0; true -> S+1 end; true -> S end.
```

Running the generated tests for the coffee machine quickly shows that the coffee machine is not resistant to such faulty brewing units. Assuming that the manufacturer specification of the brewing unit allows for such imperfection, we should change the implementation of the coffee machine to account for this behaviour. The call to `brewer.brew(C)` in Fig. 3 should become an iteration that keeps invoking the brewer until the total requested amount of coffee is produced:

```
brewmore(0, R) -> R;
brewmore(C, R) -> BC = brewer:brew(C), brewmore(C-BC, R+BC).
```

With this modification[5] our coffee machine implementation can be tested to never fail with this particular kind of a faulty brewer.

The ability to inject faults on the model level has at least two advantages. First of all, the actual implementation of the component in which we inject faults does not have to be touched, in fact, it does not even have to be available. For ready compiled libraries the task of introducing a specific fault might simply be infeasible. Furthermore, since we have the whole modelling language at our disposal, we can model several faults in parallel and provide a switching mechanism for these faults, e.g., to include all of them in the test, or only a selection. This switching can be done trivially by passing a parameter to the model during initialisation. In reality, such model variations are very common for actual software components. When testing the Autosar automotive library components with QuickCheck such models had to be created to account for different interpretations and versions or variants

[5]After also allowing more than one call-out to the `brew` operation in the specification for `start`.

of the Autosar standard [3]. In fact, such variations are one of the motivations for this work, they may or may not cause problems in software built with the corresponding libraries, in our project we look for effective ways to test for that using MBT.

5.1. Guiding Tests Towards Faults

The last issue we want to address is that realistic faults in software are *infrequent*. Actually, in the general case we specifically look for rare corner cases in components to see if they cause problems for the complete SUT, frequent or obvious faults manifest themselves rather quickly even in regular unit testing.

Already in our toy example the fault occurs every 5 uses of the brewing unit, and to detect this fault a trace with the following characteristics has to be generated. The `start` operation is invoked at least 5 times in one test case for the coffee machine, and each one call to `start` is preceded by at least two `coin` calls (recall that the fault only occurs for two requested units or more). Thus, it is not at all certain that the generated tests for the coffee machine will guide the system into a state where the brewing component gets to exhibit its fault. In reality, we had to configure the test property for the coffee machine in QuickCheck to include much longer traces than the default ones to be able to hit the brewer fault after approximately 70 tests out of the default 100. Without enlarging the traces, QuickCheck was able to hit our fault only once per couple of complete test suite executions. Once the fault has been identified with a failed test case, QuickCheck was able to construct a minimal test case showing the same fault, but finding the fault in the first place is the prerequisite for our method to be useful.

The semantics that we have defined for our model operator and the corresponding QuickCheck implementation can support this process. Apart from the simultaneous change to the model of the mocked component and retrieving the mocked operation result, our operator also evaluates the precondition of the mocked operation w.r.t. its model state and parameters. A false precondition indicates that the top-level operation in the parent component should be itself rejected from the test. We can use this to facilitate quicker fault discovering by redefining the preconditions of the mocked operations to state false if a corresponding call is not likely to cause a fault. Consequently, complete call traces not likely to cause a fault in the mocked component are rejected up-front during test generation and other candidates are explored instead.

To illustrate this, our `brew` model could have the following precondition:

```
brew_pre(S, [C]) -> C > 1.
```

This excludes any call traces in the generated cases where an attempt to call `brew` right after just one `coin` call is made. For QuickCheck just this change to guide the testing process towards fault hitting reduces the required number of test attempts from ∼70 to ∼15 to find the first occurrence of the fault.

This technique of guiding test generation by altering the preconditions can help to filter out the unwanted test cases, however, not in producing the long enough ones. For our example we need a trace with at least 16 operations – an `init` operation followed by 5 repetitions of two `coin` operations with one `start` operation. It is impossible to encode this requirement in the model of the brewing unit alone. In particular, the model cannot reject `brew` operations before 5 are made, because none at all would be actually generated. More generally, the test generation procedure does not consult models about the desired length of operation traces. This is defined in the testing property for a given model, and it should be maintained separately from our guiding preconditions.

5.2. More General Fault Models

What we proposed so far is in essence a slight "violation" of the model (a) to contain fault injections for the purpose of testing top-level software through mocking, (b) by altering preconditions, to contain information on how to guide the test generation process to reach these faults quicker. This is a violation in the sense that we change the behaviour given in the model. For faults, we have already stated that these can be parametrised and switched easily according to the needs. This allows us to maintain one model for two purposes – regular model-based testing of the associated component and model-based fault injection for testing the surrounding software.

For the precondition guided test generation we propose a similar solution. A model with hard-coded precondition changes for fault hitting becomes unusable for any other purpose. In particular, the precondition `C > 1` for the `brew` operation disqualifies the model from the use for regular model-based testing of other brewing units for which this precondition seriously limits the allowed test space. Furthermore, we manually modified this *one* precondition based on *one* particular fact about the fault that we modelled, and this lacks generality.

As with injecting faults, we want to delegate the task of adapting the preconditions for fault searching guidance into a separate specification module, and allow for easy deactivation of the feature to use the model for regular testing of components. This specification module can allow for much more flexible descriptions (than just a simple predicate over one operation) using several modelling possibilities of what, in terms of a complete trace and its parameters, may or may not lead to a fault. We call the complete model a *fault model* – one that contains the modelling of both correct and incorrect component behaviours, with the latter ones being specifically marked to enable model-based fault-injection that we have been presenting so far.

We achieve this by defining a special global precondition for our fault-injected model. In our general view of models that we have given in Sect. 2.1, preconditions are already encoded as a global predicate over the model state and the associated operation with its parameters. Instantiating this predicate with a particular operation label defines the operation precondition that is specified in the pre: tags. We simply need another such global predicate and another model state (or an extension of the existing state) to encode the information about possible behaviours leading to faults. This additional precondition is then conjoined with the first one, but only for the scenario where fault search guidance is needed, for regular testing we only use the base precondition.

Apart from the ability to be deactivated depending on the model purpose, there is nothing special about the additional precondition, it is the responsibility of the modeller to encode the faulty behaviour so that fault searching is supported. A first possible approach is to check the currently constructed execution trace against a collection of known failure traces (these can be obtained from, e.g., earlier testing of the component) by comparing with their prefixes. A current trace that cannot be matched to any known failure trace by its prefix is rejected – the precondition is false, while the current trace that can be still matched to one of the failure trace prefixes evaluates as true.

Another possibility, although much more involved, is to evaluate the current trace against a general description of known failures given as, e.g., regular expressions or similar formalism. In our earlier work we have shown how such general failure descriptions can be automatically built from the set of failed tests using automata learning [6]. Although this technique provides generality, it does not scale well for systems beyond simple. Furthermore, in [21] a technique for deriving QuickCheck models from Erlang unit tests is described, it could be adapted to also extract the information about faults.

5.3. Fault Models in QuickCheck

Let us come back to our coffee machine example specified for QuickCheck. QuickCheck does not really offer a specific separate precondition for the model (not without extending the tool's implementation). However, the existing precondition mechanism is flexible enough to allow us to encode our fault searching mechanism in the existing infrastructure in an equally modular way. Operation specific preconditions are declared along the side of the associated operation (with the specific `_pre` function), but it is also possible to declare a global precondition through `common_precondition` which has access to the state of the model and the currently considered operation with its parameters. In this global precondition we first check a global guard `f` for fault searching (we extract this from the model state) and then match on the concrete operation and parameters to enforce specific calls to `brew`:

```
precondition_common(#brewer_state{f=false}, _Call) -> true;
precondition_common(_S, {call, brewer_model, brew, [C], _})
    -> C > 1.
```

To generalise our specification completely, we also parametrise the functional behaviour of `brew` with `f` for the injected faults. Our complete model for the brewer is given in Fig. 6, which contains both the regular and the fault behaviour variants, i.e., it is a two-purpose model. Most importantly, the coffee machine model, since including the call-out specification for brewing unit and cross referencing it to the brewer model, is left totally unchanged. Any alteration of the SUT to test for faulty brewing units is done through the model of the brewing unit.

6. Autosar Case Study

In this work, we have used a simplified example of a coffee machine implemented in Erlang. The main motivation for our work comes from the automotive domain and real software components implemented in C according to the Autosar standard [3, 4]. In an earlier work [1] we presented a realistic and complete case study from the Autosar standard, there we gave the complete model and described the typical difficulties in model-based testing of such software. We also already mentioned the possibility of injecting faults on the model level, but did not apply the ideas to that particular case study. Instead, to guide fault finding, we simply brute forced QuickCheck into finding faults by enforcing very long traces and providing explicit weight

```
-module(brewer_model).
-record(brewer_state, {n, f}).

initial_state() -> #brewer_state{n=0, f=true}.
postcondition_common(S, Call, Res) ->
    eq(Res, return_value(S, Call)).

% Normal testing:
precondition_common(#brewer_state{f = false}, _Call) -> true;
% Fault guiding:
precondition_common(_S, {call, brewer_model, brew, [C], _}) ->
    C > 1;
% Otherwise:
precondition_common(_S, _Call) -> true.

brew(C) -> brewer:brew(C).
brew_args(_S) -> [nat()].
brew_pre(_S, [C]) -> C > 0.

% The fault:
brew_return(#brewer_state{n=5, f=true}, [C]) when C>1 -> C-1;
% Normal behaviour:
brew_return(_S, [C]) -> C.

% State transition under normal behaviour:
brew_next(S, _R, _A) when not S#brewer_state.f -> S;
% State transition under the fault:
brew_next(S, _R, _A) when S#brewer_state.n == 5 ->
    S#brewer_state{n=0};
brew_next(S, _R, _A) -> S#brewer_state{n=S#brewer_state.n+1}.

brew_callers() -> [cm_model].
```

Figure 6: The complete QuickCheck model for the brewing unit. Field `f` of the `brewer_state` record defines the *fault mode* (the purpose) of the model.

specifications for the operations. In other words, we manually guided the top-level test, instead of guiding it on the-level of the library model through operation specifications and proper fault models.

For this work, we have modified our case study from [1] to apply the ideas we have just presented. The core issue in the case study that we modelled was an internal, undocumented and unannounced (the API does not report it in any way) size limitation of 128 elements of a low-level FIFO queue implementation `cirq_buffer`. To guide the top-level message posting system `mbox` based on this queue towards hitting the internal fault we had to provide fault specific data generators in QuickCheck with specific weights to slant the random data generation towards the fault. This was necessary, the minimal required execution trace to hit the fault has to contain 130 specific operations, without guidance the discovery of the fault through default random trace generation was not possible.

Using the methodology described in this paper, we have modelled the fault of the `cirq_buffer` implementation in its model and stated two guiding properties. First, the fault occurs only for FIFO queues initialised with the size of at least 128 elements. Second, dequeuing operations specifically drive the queue *away* from hitting the 128 boundary fault. Injecting the fault into `cirq_buffer` and modelling the two mentioned facts about the fault allowed QuickCheck to find the system failure in `mbox` after 15 test cases on average. One particular gain compared to the previous case is that now the initially found counterexample is closer to the minimal one, by fault model construction it does not contain the irrelevant dequeuing operations. However, we still had to ask for long enough traces to be produced. For realistic systems with concealed corner cases this is not unusual and a well designed test suite should certainly require running very long operation traces to reach high confidence levels. Thus, we do not consider this a shortcoming, but rather a reminder of good test engineering. With the fault deactivated in the model, we were also able to thoroughly test a healthy `cirq_buffer` implementation in isolation, effectively exercising two purposes – functional and robustness testing – of the model on a realistic example.

7. Related Work

Fault-injection is an often used technique in software validation, and in testing in has been used in at least two contexts.

In mutation testing [19] that we have already mentioned, the implementation code is injected with a fault in an attempt to witness a test failure in some set of test cases. This is used to evaluate the adequacy of the set of test cases, i.e., a set of test cases that do not discover the injected fault is defined to be insufficient. This technique is applicable generally in testing, thus also in particular in MBT [22]. Alternatively, in the MBT context faults can be injected into models to discover new correct or incorrect system behaviours [22]. The term *fault model* in mutation testing means the type or class of an injected fault (like flipped condition or statement swap) rather than a complete modelling of a faulty behaviour that we refer to in our work. The primary purpose of mutation testing and similar techniques is the elimination of bugs by ensuring the existence of suitable test cases. Moreover, faults in mutation testing are generally temporary artefacts, while we add faults to our models permanently so that models can interchangeably serve two purposes in MBT.

The second testing technique involving fault injection is *fuzzing* or *fuzz testing* [23]. In this context, it is the input data to the system that is injected with faults – unexpected values, format, etc. (in [23] also derived from symbolic analysis of the source code). The system is then tested for resilience to these kinds of faults. This is applied primarily in the context of security [24, 22], where insufficient input validation often leads to reliability and security issues. Our approach is similar in this respect, we also want to test for fault resilience or robustness, but we try to invalidate complete system components, rather than just input data, and we do it on the model level. We work on the assumption that faults and defects in the software are inevitable (due to complexity, version variation, and insufficient specifications). Thus, similar to fuzz testing we look for *fault tolerance*, rather than fault absence like in mutation testing.

All related techniques, including ours, can be of course combined together or at least applied on the same system for different purposes. Our idea of multi-purpose models should facilitate model reusability in this context.

8. Conclusion

Motivated by an industrial application domain and case studies from an automotive area, we have shown a modelling technique and model-based testing methodology to (a) inject faults into models, (b) execute these models during tests in place of actual components, (c) guide test generation towards fault finding through additional modelling. Overall, the method uses the concept of fault models, which encode both the correct behaviour for regular model-based functional testing, and incorrect behaviour for robustness testing. Using a simplified notion of labelled transition systems we have semi-formally defined the `model` operator to cross-reference models and explained how this is readily implemented in the existing MBT tool QuickCheck. We used a toy example of a coffee machine with a brewing unit to explain our ideas and related to a larger case study of realistic C code from the Autosar standard. We presented what we believe is a general methodology of multi-purpose modelling applicable in different MBT settings. However, a specific tool support is required to make it practical and the method application still requires specific domain knowledge and modelling approach to use MBT in this context (as does any other software validation technique applicable on a larger scale).

8.1. Future Work

We have so far experimented with two case studies to evaluate our work, the next step is to apply it on something substantially larger and more complex. The major challenge that we foresee there is to tackle the level of detail of models such that it would still allow us to effectively plug them in as runnable code into a larger system. Furthermore, our fault guiding information is based on explicit boolean assertions – the model transition does or does not lead the system towards a possible fault. More general models and methodology could use less discrete notion of this likelihood, i.e., weights or probabilities. QuickCheck already uses weights in execution trace generation, however, this mechanism would need to be tailored to the method that we presented here. Down the road, this work and the idea of multi-purpose models set grounds for model-based testing of (semi-)autonomous systems where different aspects, ranging from basic functional testing to Artificial Intelligence on-line testing (monitoring), have to be incorporated into one software validation process. Finally, we would like to pursue the idea of automata learning for automatic building of fault models from collections of failed tests cases and corresponding execution traces.

Acknowledgements

This work was supported by the Swedish Knowledge Foundation grant for the AUTO-CAAS project. We would also like to thank the anonymous reviewers for very detailed remarks about this work.

References

[1] W. Mostowski, T. Arts, J. Hughes, Modelling of Autosar libraries for large scale testing, in: H. Hermanns, P. Höfner (Eds.), 2nd Workshop on Models for Formal Analysis of Real Systems (MARS 2017), Vol. 244 of EPTCS, Open Publishing Association, 2017, pp. 184–199. doi:10.4204/EPTCS.244.7.

[2] J. Tretmans, Model-based testing and some steps towards test-based modelling, in: Formal Methods for Eternal Networked Software Systems, Vol. 6659 of LNCS, Springer, 2011, pp. 297–326. doi:10.1007/978-3-642-21455-4_9.

[3] AUTOSAR BSW and RE Conformance Test Specification, Release 4.0, Revision 2 (2011).

[4] T. Arts, J. Hughes, U. Norell, H. Svensson, Testing AUTOSAR software with QuickCheck, in: Eighth IEEE International Conference on Software Testing, Verification and Validation Workshops, 2015, pp. 1–4. doi:10.1109/ICSTW.2015.7107466.

[5] J. Svenningsson, H. Svensson, N. Smallbone, T. Arts, U. Norell, J. Hughes, An expressive semantics of mocking, in: Fundamental Approaches to Software Engineering, Vol. 8411 of LNCS, Springer, 2014, pp. 385–399. doi:10.1007/978-3-642-54804-8_27.

[6] S. Kunze, W. Mostowski, M. Mousavi, M. Varshosaz, Generation of failure models through automata learning, in: Second International Workshop on Automotive Software Architectures (WASA 2016), IEEE Society, 2016, pp. 22–25. doi:10.1109/WASA.2016.7.

[7] T. Arts, J. Hughes, J. Johansson, U. Wiger, Testing telecoms software with QuviQ QuickCheck, in: Proceedings of ERLANG'06, ACM, 2006, pp. 2–10. doi:10.1145/1159789.1159792.

[8] T. Arts, M. Mousavi, Automatic consequence analysis of automotive standards (AUTO-CAAS), in: First International Workshop on Automotive Software Architectures (WASA 2015), ACM Press, 2015, pp. 35–38. doi:10.1145/2752489.2752495.

[9] M. Weiglhofer, B. K. Aichernig, Unifying input output conformance, in: A. Butterfield (Ed.), Unifying Theories of Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 181–201. doi:10.1007/978-3-642-14521-6_11.

[10] J. Tretmans, Model Based Testing with Labelled Transition Systems, in: R. M. Hierons, J. P. Bowen, M. Harman (Eds.), Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 1–38. doi:10.1007/978-3-540-78917-8_1.

[11] H. Felbinger, C. Schwarzl, Suitability analysis of CSP- and SMT-solvers for test case generation, in: Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, ACM, New York, NY, USA, 2014, pp. 40–49. doi:10.1145/2593735.2593741.

[12] T. Arts, J. Hughes, How well are your requirements tested?, in: 2016 IEEE International Conference on Software Testing, Verification and Validation, 2016, pp. 244–254. doi:10.1109/ICST.2016.23.

[13] G. Fink, M. Bishop, Property-based testing: A new approach to testing for assurance, SIGSOFT Softw. Eng. Notes 22 (4) (1997) 74–80. doi:10.1145/263244.263267.

[14] T. Arts, S. Thompson, From test cases to FSMs: augmented test-driven development and property inference, in: Proceedings of the 9th ACM SIGPLAN workshop on Erlang, ACM, 2010, pp. 1–12. doi:10.1145/1863509.1863511.

[15] J. Hughes, QuickCheck testing for fun and profit, in: Proceedings of PADL'07, Springer, 2007, pp. 1–32. doi:10.1007/978-3-540-69611-7_1.

[16] F. Cesarini, S. Thompson, Erlang Programming, O'Reilly, 2009.

[17] AUTOSAR Consortium, Acceptance Test Specification of Communication on CAN bus – Release 1.0.0 (July 2014).

[18] W. Mostowski, E. Poll, J. Schmaltz, J. Tretmans, R. W. Schreur, Model-based testing of electronic passports, in: M. Alpuente, B. Cook,

C. Joubert (Eds.), Formal Methods for Industrial Critical Systems 2009, Proceedings, Vol. 5825 of LNCS, Springer, 2009, pp. 207–209. doi:10.1007/978-3-642-04570-7_19.

[19] L. J. Morell, A theory of fault-based testing, IEEE Transactions on Software Engineering 16 (8) (1990) 844–857. doi:10.1109/32.57623.

[20] M. Grzejszczak, Instant Mockito, Packt Publishing, 2013.
URL https://books.google.se/books?id=oIGJnQAACAAJ

[21] T. Arts, P. Seijas, S. Thompson, Extracting QuickCheck specifications from EUnit test cases, in: Proceedings of the 10th ACM SIGPLAN workshop on Erlang, ACM, 2011, pp. 62–71. doi:10.1145/2034654.2034666.

[22] F. Bouquet, F. Peureux, F. Ambert, Model-based testing for functional and security test generation, in: A. Aldini, J. Lopez, F. Martinelli (Eds.), Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures, Springer International Publishing, Cham, 2014, pp. 1–33. doi:10.1007/978-3-319-10082-1_1.

[23] P. Godefroid, M. Y. Levin, D. Molnar, Automated whitebox fuzz testing, in: Proceedings of Network and Distributed Systems Security (2008), 2008, pp. 151–166. doi:10.1.1.129.5914.

[24] P. Godefroid, M. Y. Levin, D. Molnar, Sage: Whitebox fuzzing for security testing, Communications of ACM 55 (3) (2012) 40–44. doi:10.1145/2093548.2093564.