Generation of Failure Models through Automata Learning

[Position Paper]

Sebastian Kunze, Wojciech Mostowski, Mohammad Reza Mousavi, Mahsa Varshosaz
Centre for Research on Embedded Systems, Halmstad University, Sweden
{sebastian.kunze,wojciech.mostowski,m.r.mousavi,mahsa.varshosaz}@hh.se

ABSTRACT

In the context of the AUTO-CAAS project that deals with model-based testing techniques applied in the automotive domain, we present the preliminary ideas and results of building generalised failure models for non-conformant software components. These models are a necessary building block for our upcoming efforts to detect and analyse failure causes in automotive software built with AUTOSAR components. Concretely, we discuss how to build these generalised failure models using automata learning techniques applied to a guided model-based testing procedure of a failing component. We illustrate our preliminary findings and experiments on a simple integer queue implemented in the C programming language.

Keywords

model-based testing; automatic test generation; automata learning; failure model; AUTOSAR standard

1. INTRODUCTION

Establishing the severity of a failed test case and predicting its possible consequences is a challenge in test processes. Very often failed test cases are dismissed on the basis that they concern only singular corner cases and will not lead to any further catastrophic failures. This problem is intensified in the context of model-based testing, where it is more difficult to relate the generated test-cases to the requirements. We have observed this problem in the context of our industrial collaboration in the AUTO-CAAS project [4], where our aim is to provide effective consequence analysis methods in the context of AUTOSAR software.

Our solution to the problem above is to come up with *generalised failure models*. Such models start from a single failing test case and produce a model which specifies under what other circumstances (e.g., other interaction scenarios with possibly different parameters), a similar failure can be observed. To this end, we build a wrapper around the model-based testing tool QuickCheck [3] that by using automata-learning algorithms turns failing test cases into generalised failure models. We currently have an prototype implementation of our approach to experiment with different methods to come up with succinct generalised failure models that are accessible for automotive software engineers.

The rest of this paper is organised as follows. Section 2 describes the main ideas behind model-based testing as implemented in QuickCheck, Sect. 3 elaborates on our method for failure model generation using automata learning, Sect. 4 discusses primary experiments with our prototype wrapper around QuickCheck, and finally Sect. 5 concludes the paper with a short discussion.

2. MODEL-BASED TESTING WITH QUICKCHECK

The scenario and particular set up that we focus in our work is testing of the AUTOSAR components [6] using the QuickCheck [3] model-based testing tool and AUTOSAR models developed by QuviQ.[1] For the purpose of this paper we abstract from AUTOSAR, as we have not yet performed any substantial experiments on the actual AUTOSAR components; this is part of our upcoming work. However, we do use the QuickCheck tool and its model-based testing methodology, adapting it to our needs.

QuickCheck models are *symbolic* state-full specifications of a behaviour of the underlying implementation expressed in the functional language Erlang [7]. Typically, this specification declares the model state of the underlying implementation and symbolic inputs, and describes the valid protocol of calling operations (functions, procedures, or methods, depending on the nomenclature of the implementation language) using preconditions for every declared operation. A precondition establishes if a given operation with its parameters is permitted under the current model state of the system, i.e., preconditions place operations in the call protocol sequence. Additionally, each operation can be annotated with a postcondition, i.e., a property that should be satisfied after the corresponding operation is completed. Hence the resulting testing method is called *property based testing*.

The QuickCheck testing method works by walking the model and randomly generating (a) sequences of operations to be called on the implementation under test and selecting those that satisfy the requirements of the specified call protocol, i.e., preconditions, and (b) concrete input parameters to these operations following the data generators specified in the model. The generated test sequence is run against the

WASA '16 , April 4, 2016, Venice, Italy

© 2016 ACM. ISBN xxx-xxxx-xx-xxx/xx/xx. . . $15.00

DOI: xx.xxx/xxx_x

[1] http://www.quviq.com.

implementation under test, and after each completed operation the postcondition specified in the model is checked. When it fails, a counter example comprising of the sequence of operations and their parameters executed so far and the failing property is reported to the user. Then QuickCheck applies its flagship procedure of *shrinking* the counter example to get a minimal failing test case. This works by recursive application of the shrinking procedure starting from the sequence of executed operations down to single data values for each operation involved. Each intermediate step applies the shrunk candidate test case to the implementation to establish the persistence of the fault under shrinking. The procedure stops when the discovered fault stops manifesting itself under subsequent shrinking steps. The suitable shrinking strategy is an inherent feature of each data generator used in the model, and each such strategy and can be modified to further guide the shrinking mechanism.

By providing suitable bridging mechanisms, QuickCheck can relate to implementations in several programming languages, in particular system and library implementations in the C programming language.

Example 1. Figure 1 shows a fragment of a QuickCheck model of a FIFO queue of integers. The state of the model keeps the pointer ptr to the underlying queue C structure, the size of the queue, as well as the model contents of the queue – a list of elements currently stored in the queue. The put operation refers the model to the C implementation q:put of the queue passing the pointer Ptr and value Val parameters. The remaining put_* model functions define the intended behaviour and constraints of the put operation. The arguments function put_args defines the arguments to be used with put – the pointer currently stored in the model state, and a randomly generated integer. The latter is expressed with a data generator int(), which provides random generation and shrinking facilities for integer parameter types. The precondition put_pre specifies that put can only be called on already initialised queues (with a new operation not quoted here) that are also not full. The postcondition put_post checks that the return value of put is equal to its parameter, and finally, the put_next function defines how the state of the model changes after the operation. Other queue operations are declared in the similar fashion, after which a test property prop_q is defined that is responsible for generating model conforming test cases and reporting the (possibly failing) test results.

```
-record(state, {ptr, size, elements}).
initial_state() -> #state{ elements=[] }.
...
put(Ptr, Val) -> q:put(Ptr, Val).
put_args(S) -> [S#state.ptr, int()].
put_pre(S, [_P, _E]) ->
  S#state.ptr /= undefined andalso
  length(S#state.elements) < S#state.size.
put_post(_S, [_P, E], R) -> R == E.
put_next(S, _R, [_P, E]) ->
  S#state{ elements = S#state.elements ++ [E] }.
...
prop_q() -> ?FORALL(Cmds, commands(?MODULE),
  begin
    {H, S, Res} = run_commands(?MODULE, Cmds),
    collect(S, pretty_commands(?MODULE, Cmds,
      {H, S, Res}, Res == ok))
  end).
```

Figure 1: Specification of the queue of integers put operation in QuickCheck.

3. GUIDED TEST PROCEDURE FOR FAILURE MODEL CONSTRUCTION

In earlier work, automata learning has been investigated for the purpose of building *correct* QuickCheck models of complex implementations, see e.g. [5]. Our practical goal for this paper is to build a formal description of a *failure* detected by QuickCheck in the implementation under test in a form of a model, i.e., a *specification* rather than just one counter example. This model consists of an automaton formalising the execution paths leading to a failure, and the property that failed to verify after the last operation on the given execution path, essentially the failing postcondition of the last operation. Given an implementation that is known to have a failure from an earlier test run, the failure model is built automatically by bridging automata learning to a refined and guided test procedure applied by QuickCheck to this failing implementation. More precisely, the original test procedure of QuickCheck is adapted so that (a) the input alphabet of the operations under test is suitably abstracted to support the learning procedure, (b) the test cases used in the learning process are *similar* to each other to reduce learning *noise* and consequently keep the resulting model small, and (c) test results reported by QuickCheck are adapted so that the behaviour of one particular failure can be determined. Our learning procedure requires a user supplied configuration with implementation specific information to guide the process, hence certain a priori knowledge about the implementation under test is required. We elaborate on these points in the remainder of this section.

3.1 Automata Learning with LearnLib

The automata learning framework we have chosen for our work is the LearnLib platform[2] [8] developed at TU Dortmund. The main reason for choosing this platform is the fully functional and flexible implementation in Java, as well as earlier very successful results of applying LearnLib to non-trivial, realistic implementations [2, 1]. Additionally, an easy to use visualisation interface is available in LearnLib making it an ideal tool for experimentation.

LearnLib provides several different learning (inference) algorithms for several different kinds of automata. For our work we use Mealy machines, i.e., deterministic finite state machines with input and output labelled transitions. With its set of inputs, the generated machine characterises the operation sequences leading to one distinguished state representing the failure. The outputs of the machine are essentially the status of conformance to the model, with the failure status always leading to the failure state.

The basic learning setup consists of a *learner*, in our case the failure model learning module, and the *teacher*, in our case the adapted system under test. The inference proceeds by the learner invoking membership queries on the teacher and building a hypothesised state machine $\mathcal{H}$ that reflects the behaviour of the system under learning. Membership

[2] http://learnlib.de.

queries are simply operations and their parameters to be invoked on the system under learning, and responses to these queries are the output results of the operations. We do not report the actual operation results to the learner, but the state of the test after the given operation, i.e., whether the given operation is permitted according to the model, and if so whether it caused any failures in the implementation. The inference procedure is finished once the hypothetical Mealy machine $\mathcal{H}$ becomes stable[3] and $\mathcal{H}$ becomes the resulting model of the failure in the system under learning.

3.2 Input Data Abstraction

The automata inference algorithms work efficiently only when the language alphabet is relatively small. This is clearly not the case for automotive software systems with several operations that can take parameters from a large domain. Hence, abstraction of the concrete input and output data is required to keep the automata language small and the learning process feasible. This abstraction is done by mediating between the learner and the system under learning, i.e., the teacher. The learner uses a possibly small set of abstract parameters that are made concrete by the mediator for each abstract set. The mediator in our case is the adapted QuickCheck testing procedure, which chooses suitable representatives for the abstract domains of operation parameters.

An additional aspect that we need to take into account during input parameter mediation is to make the subsequent test cases *similar* to each other, so that our hypothetical model $\mathcal{H}$ does not diverge and stabilises quickly. This is best explained with the following example.

Example 2. Consider the new operation of the queue that takes an integer parameter size and initialises a new queue accordingly. Two queues of different sizes will give rise to two different Mealy machines, each one having essentially the number of states equal to the size of the queue. If we allow the queue to be initialised with random sizes in each new operation trace during the inference procedure, the process will either not terminate within reasonable time, or it will produce a very complicated model that attempts to formalise queues of different sizes in one Mealy machine. Thus, it is better to use a fixed parameter for all new operations in one learning process, either by fixing the parameter value a priori, or by consistently reusing one that has been chosen by the test generator for the first use of new.

This gives rise to the following two methods of parameter mediation between the learner and system under test. The first method is to simply allow the user to fix the parameters of selected operations to constant values and bypass the data generation mechanism of QuickCheck that would be used otherwise. For our queue example, we would fix the parameter to the new operation, e.g., to size 3, and allow the test procedure to use randomly generated parameters for the put operation. These would not influence the shape of the resulting model which only reflects the number of elements stored and not the value of elements in its state space.

The second method generalises the first one. Instead of fixing the parameters we allow the test procedure to decide which parameters should be fixed after the first use. This is done by defining an oracle that traces the changes in the model state of the implementation under test and decides whether the state change incurred by the given operation requires fixing the operation parameter for subsequent calls to the same operation. In particular, on its first occurrence the new operation would update the size field of the model state of the queue. The oracle would detect this change in the model state, by seeing that the size field is changed from one value to another, and save the parameter used with the operation that triggered this change. In other words, our oracle divides operations and their parameters to ones that cause *meaningful* and *meaningless* changes of the model state for the purpose of learning a concise failure model.

Both methods of parameter mediation require a priori knowledge about the system under test. This is inevitable to facilitate efficient model inference and also the readability of the result. On the practical side, the user has to prepare a suitable configuration for the adapted QuickCheck test generation procedure to guide the learning process. This test configuration can include an association of fixed parameters to particular operations, or an oracle function described above to compare subsequent model states, or a mixture of both. We show a brief example of this in Sect. 4.

3.3 Adapting Test Results for Learning

Similarly to the input parameters, the outputs of the operations invoked on the concrete implementation during inference have to be abstracted too. The first obvious reason is to narrow down the language alphabet of the model under construction. More importantly, however, the outputs have to be adapted so that the constructed model represents the fault in the system that we try to formalise. To this end, we ignore the actual operation results (e.g., the concrete value that a get queue operation returns), and concentrate on the current state of the model and the current status of the test suite executed on the implementation.

More precisely, we report the following outputs to the learner. If an operation is not permitted by the model according to a precondition, we report notrace result to inform the learner that the given operation is not in the scope of the model. Otherwise, operations permitted by the model can result in two values. We report the value ok if the operation does not cause any test failure, or we report fail otherwise. Since we are interested in scenarios leading to a failure, and not ones after the failure has occurred, all subsequent operations after a failure return notrace to signal no further need to probe the system. The resulting Mealy machine has several states representing the behaviour of the implementation up to the failure, and then one special failure state F to which the failing operation from selected normal state leads. To improve readability we project out all the notrace transitions from the resulting failure model.

4. EXPERIMENTS

We have implemented a prototype of our failure model generator following what we presented in Sect. 3. Our tool is a wrapper around QuickCheck that adds facilities to define test configurations (operation parameter fixing and state change tracing oracles, see Sect. 3.2), mediate operation inputs and outputs, and communicate with LearnLib through

[3] In an ideal situation the teacher knows the actual state machine $\mathcal{M}$ that is being inferred by the learner, in which case part of the process is to probe the teacher with an equivalence query $\mathcal{M} = \mathcal{H}$. Our teacher, however, is a black-box system, and this equivalence can be only established based on statistical stabilisation of $\mathcal{H}$ under subsequent queries.

```
tconf1() -> eqc_learn_fault:init_learn(q_eqc, [{new, [3]}]).

cmp_state(OS, NS) -> OS#state.size /= NS#state.size.
tconf2() -> eqc_learn_fault:init_learn(q_eqc, [],
    fun(OS, NS) -> cmp_state(OS, NS) end).
```

Figure 2: Two kinds of configurations for learning the fault in the queue implementation.

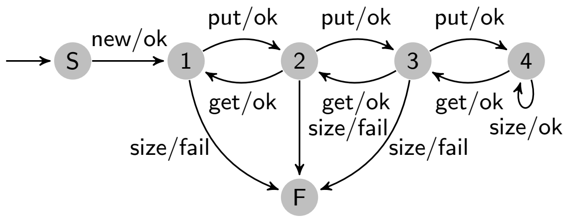

Figure 3: Failure model of the queue of size 3.

a socket. For experimentation we used the FIFO queue of integers that is used as a standard QuickCheck tutoring example. The queue implementation contains two subtle bugs that cause the space function reporting available space in the queue to return incorrect results.

Example 3. Figure 2 shows a snapshot of two configuration definitions for learning the faulty queue behaviour, the first one fixes the size parameter of new to 3, the second one uses an oracle function detecting a change of size in the queue model to fix the parameter to be used with new. Fig. 3 shows the result of learning the fault in the queue with the first configuration, while Fig. 4 is the result of using the second configuration where the first instance of calling new has been randomly chosen to use 2 for the queue size, which was subsequently reused during the inference process.

In both cases it is clear that the space operation is not behaving according to the specification. The failure model in Fig. 3 shows that space only gives a correct result on a full queue. In Fig. 4, however, despite a smaller queue size chosen by the generator, the space operation shows seemingly more unpredictable behaviour. Its correct behaviour depends on the interleaving of earlier put and get operations. The underlying problem in the implementation is actually due to the input and output pointers to a circular array and size working incorrectly when one pointer "overtakes" the other. From both figures we can also conclude that the parity of the queue size plays a role in the detected failure.

5. DISCUSSION AND CONCLUSIONS

We have presented preliminary ideas for using automata learning to generate models of failing or non-conformant behaviour of software components. In the AUTO-CAAS project, we will use these models to match and find a component contributing to a failure in a larger system. So far we have concentrated only on mediating the inputs and outputs between the learner and the system under test to enable creation of concise and readable models. At this stage our failure models are still quite limited. First, they only show the failure for one particular configuration of the underlying implementation (in our examples, a queue of a fixed size). As our experiments show, different configurations can show varying failure characteristics of the same implementation defect. Second, our generated model only shows which operation exhibits the failure, but not exactly how. For our example, we know that the size operation returns the wrong value, but not how this actual value generally relates to the expected one.

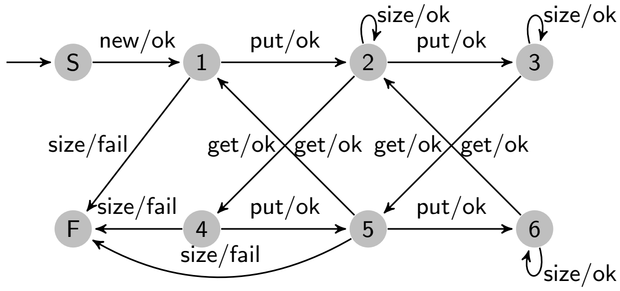

Figure 4: Failure model of the queue of size 2.

More generally, our method does not yet indicate if the generated models show the failure caused by one or several defects in the implementation. If it is several defects, then the question is how can we separate them in our method. For now, our first approximation is that we associate a defect with the failing operation, in our example the size function.

In the course of our upcoming work we will be addressing all these issues, and also we will apply our methods on more realistic examples in the automotive domain using the AUTOSAR software components.

Acknowledgements

Our work is supported by the Swedish Knowledge Foundation grant for the AUTO-CAAS project and by the Swedish Research Council grant for the project on Effective Model-Based Testing of Concurrent Systems (EFFEMBAC).

6. REFERENCES

[1] F. Aarts, J. D. Ruiter, and E. Poll. Formal models of bank cards for free. In *Software Testing Verification and Validation Workshop*. IEEE Computer Society, 2013.

[2] F. Aarts, J. Schmaltz, and F. Vaandrager. Inference and abstraction of the biometric passport. In *Proceedings of ISoLA 2010*. Springer, 2010.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with QuviQ QuickCheck. In *Proceedings of ERLANG'06*. ACM, 2006.

[4] T. Arts and M. Mousavi. Automatic consequence analysis of automotive standards (AUTO-CAAS). In *First International Workshop on Automotive Software Architectures (WASA 2015)*. ACM Press, 2015.

[5] T. Arts, P. Seijas, and S. Thompson. Extracting QuickCheck specifications from EUnit test cases. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*. ACM, 2011.

[6] *AUTOSAR BSW and RE Conformance Test Specification, Release 4.0, Revision 2*, 2011.

[7] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly, 2009.

[8] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *Proceedings of TACAS 2011*. Springer, 2011.