From Test Cases to FSMs: Augmented Test-driven Development and Property Inference

Thomas Arts
Chalmers / Quviq AB, Gothenburg, Sweden
thomas.arts@ituniv.se

Simon Thompson
University of Kent, Canterbury, UK
S.J.Thompson@kent.ac.uk

Abstract

This paper uses the inference of finite state machines from EUnit test suites for Erlang programs to make two contributions. First, we show that the inferred FSMs provide feedback on the adequacy of the test suite that is developed incrementally during the test-driven development of a system. This is novel because the feedback we give is *independent* of the implementation of the system.

Secondly, we use FSM inference to develop QuickCheck properties for testing state-based systems. This has the effect of transforming a fixed set of tests into a property which can be tested using randomly generated data, substantially widening the coverage and scope of the tests.

Categories and Subject Descriptors D. Software [*D.2 SOFTWARE ENGINEERING*]: D.2.5 Testing and Debugging: Testing tools

General Terms Verification

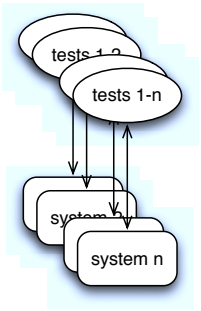
Keywords TDD, test-driven development, Erlang, EUnit, unit test, QuickCheck, property, inference, finite-state machine

1. Introduction

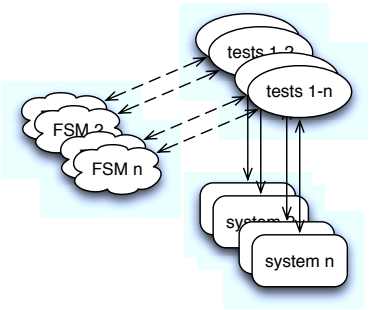
In this paper we show how finite state machines can be automatically extracted from sets of unit tests – here Eunit [6] tests for Erlang programs. We use these FSMs in two ways. First, they can in themselves provide feedback on the adequacy of a set of tests, independently of any implementation. Secondly, they can be transformed and used within Quviq QuickCheck [1, 14] to guide the random generation of test sequences for state-based systems. We discuss these contributions in turn now.

Test-driven Development

Test-driven development [3, 4] (TDD) advocates that tests should precede implementations. Systems should be developed incrementally, with each increment delivering enough functionality to pass another test, as illustrated here.



Under this approach, how can we validate the system? It will surely meet the tests, because it has been developed precisely to pass them. The question becomes one of *validating the tests themselves*. In this paper we propose that during TDD of state-based systems we can validate the tests by extracting the finite state machine (FSM) implicit in the current test set.



The FSM is extracted by means of grammar inference [23] over sets of positive and negative traces. This FSM provides feedback on the tests *independently* of any implementation, and thus 'triangulates' the process.

We would argue that that makes the process of test-driven development more robust. In particular, it allows us to give an answer to the question *"When have I written enough tests?"* on the basis of the tests alone, rather than by examining an implementation. We return to this question in Section 5.

We illustrate our approach to TDD by working through a case study in Erlang, developing unit tests in EUnit, and using StateChum [19] to extract a series of FSMs from the test suite as it evolves.

Testing state-based systems

This work was developed within the European Framework 7 ProTest project [18] to develop property-based testing for Erlang.

Erlang'10, September 30, 2010, Baltimore, Maryland, USA.

In particular we seek to develop QuickCheck[1] properties from sets of unit tests, thus providing a migration path from traditional unit testing to property-based testing. To test state-based systems in QuickCheck it is usual to develop a state machine model (using `eqc_fsm` or `eqc_statem`) which encapsulates the permissible sequences of API calls.

We show how the FSM extracted from a set of unit tests can be transformed into a QuickCheck FSM, and thus how a set of unit tests can be combined into a property. This has the benefit of allowing the system to be tested on many more inputs, namely all those permissible in the FSM, and a selection of these can be generated randomly using QuickCheck generators.

Abstraction

In modelling a system using a finite state machine we need to perform abstraction over the state data. In the case study the data consist of a finite collection of resources, and this is modelled by sets with small cardinalities before we seek to identify a general case. In the case of test development, this allows us to identify complete sets of tests for 'small' models before moving to the general case. In QuickCheck this process identifies candidate state data values.

Roadmap

We begin by discussing the background to this work in Section 2. We first introduce test-driven development, and then discuss EUnit and QuickCheck for testing Erlang systems. We also look at grammar inference as a mechanism for inferring finite-state machines from sets of words in the language and its complement. We use StateChum to do FSM inference in our case study.

Section 3 discusses a systematic approach to developing and assessing tests during test-driven development through the case study of a 'frequency server'. We use Eunit to express the tests and StateChum to infer finite state machines from test sets in a fully automated way. While doing this we discuss the question of how to abstract away from particular aspects of the system in forming a model of the system under test.

Section 4 builds on this by developing a QuickCheck state machine for the example. This machine is based on the FSM inferred in the previous section, and we discuss the process of building the QuickCheck machine from this FSM with a view to automating the process as much as possible in the future.

Finally we discuss related work in Section 5 and draw some conclusions in Section 6.

2. Background

In this section we give a brief overview of the main topics which form the background to the work reported this paper, as well as providing references where more information can be found.

2.1 Test-driven development

A manifesto for test-driven development (TDD) is given in Beck's monograph [4]. This gives advice on adopting TDD in practice, as well as answering frequently-asked questions. The thesis of test-driven development is that it is the formulation of tests which should be used to drive the development process.

Specifically, the requirements for the system are given by a series of tests developed incrementally. At each stage the implementor will write enough to satisfy the existing tests and (in theory at least) nothing more. Hence the importance of the tests in specifying the system, and so the importance of finding mechanisms by which the tests can be validated in some independent way. In Section 5 we compare our approach to others in the TDD community.

[1] In this article, QuickCheck refers to Quviq QuickCheck version 1.18

2.2 EUnit

EUnit [5, 6] provides a framework for defining and executing unit tests, which test that a particular program unit – in Erlang, a function or collection of functions – behaves as expected. The framework gives a representation of tests of a variety of different kinds, and a set of macros which simplify the way EUnit tests can be written.

For a function without side-effects, a test will typically look at whether the input/output behaviour is as expected, and that exceptions are raised (only) when required.

Functions that have side-effects require more complex support. The infrastructure needed to test these programs (called a *fixture*) includes a facility to setup a particular program state prior to test, and then to cleanup after the test has been performed.

Examples of EUnit tests are given in the body of the paper, and are explained as they occur. The text [7] gives an introduction to EUnit testing; further details can be found in [5, 6] and the online documentation for the system.

2.3 Grammar and state machine inference

The StateChum tool extracts a finite state machine from sets of positive and negative instances [19]. That is, the user provides sets of words which are in (resp. out) of the language of the state machine, and grammar inference techniques are used to infer the minimal machine conforming to this requirement.

The algorithm uses a *state merging* technique: first the (finite) machine accepting exactly the positive cases is constructed, then states are merged in such a way that no positive and negative states are identified. The particular implementation assumes that the language accepted is prefix-closed, so that in terms of testing a single positive case can be seen as representing a number of positive unit tests. Further details of the algorithm are in [23, 24].

FSM and grammar inference is a well-established field: an early introduction can be found in [22].

2.4 QuickCheck

QuickCheck [1, 14] supports random testing of Erlang programs. Properties of the programs are stated in a subset of first-order logic, embedded in Erlang syntax. QuickCheck verifies these properties for collections of Erlang data values generated randomly, with user guidance in defining the generators where necessary.

When testing state-based systems it makes sense to build an abstract model of the system, and to use this model to drive testing of the real system. Devising this model is crucial to the effective testing, and the technique outlined in this paper facilitates model definition from existing test data rather than from an informal description of the system under test.

QuickCheck comes with a library (`eqc_fsm`) for specifying test cases as finite state machines. For each state in the FSM it is necessary to describe a number of things.

- The possible transitions to other states.
- A set of preconditions for each transition.
- How to actually perform the transition (that is, a function that performs whatever operations are necessary).
- Postconditions to make a check after the state transition.
- A description of the changes on the state as a result of the transition.

This information is supplied by defining a set of callback functions; we will see an example of this in practice in Section 4.

3. Test-driven development

In this section we introduce a procedure for systematically developing the unit tests that are used in the test-driven development process of systems. This is illustrated through the running example of a simple server.

3.1 Example: a frequency server

As a running example (taken from [7]) we write tests for a simple server that manages a number of resources – frequencies for example – which can each be allocated and deallocated. The server process is registered to simplify the interface functions, so that it is not necessary to use the process identifier of the server to communicate with it. The Erlang type specification for the interface functions is as follows:

```
-spec start([integer()]) -> pid().
-spec stop() -> ok.
-spec allocate() -> {ok,integer()} |
                    {error, no_frequency}.
-spec deallocate(integer()) -> ok.
```

The `start` function takes a list of frequencies as argument and spawns and registers a new server that manages those frequencies. The `stop` function communicates with the server to terminate it in a normal way.

The `allocate` function returns a frequency if one is available, or an error if all frequencies have already been allocated. The `deallocate` function takes a previously allocated frequency as argument and has the server release that frequency.

3.2 Testing start/stop behaviour

As straightforward as this server seems to be, it is still a good idea to define some tests before we write the code. We use EUnit [6] as a framework for writing our unit tests, but the principles in this paper apply however we write unit tests.

We start by defining tests for starting and stopping the server, not worrying about allocation and deallocation. Of course we want a test in which we start and stop the server, but we also want to test that we can start it again after stopping. Since the second test subsumes the first, we only define the second.

```
startstop_test() ->
   ?assertMatch(Pid1 when is_pid(Pid1),start([])),
   ?assertMatch(ok,stop()),
   ?assertMatch(Pid2 when is_pid(Pid2),start([1])),
   ?assertMatch(ok,stop()).
```

We start the server twice, each with a different list of resources, more or less an arbitrary choice. The second call to `stop` is performed to clean up and return to the state in which no server is registered. Note that we match the returned values of the calls, viz. `ok` for stop and a pid for `start`, precisely as required by the specification.

Note that although we have defined this as a single EUnit test, it can also be seen as representing four separate tests, one performed by each `?assertMatch` expression. The four tests check that that the system can be started, that it can be started and then stopped, and so forth: one test case for each prefix of the sequence of `?assertMatch` statements.

Now we would be able to write our first prototype, but it is obvious that if we write `start` and `stop` to just return the correct return types, then the test would pass. This indicates that we have too few tests for a proper test-driven development of a non-trivial server. How do we find out which additional tests to add?

One answer is to appeal to our programmers' intuition, but a more satisfactory – and principled – approach is to look at the set of tests and see what state space is implicit in these. More specifically, we can extract the minimal finite state machine (FSM) from the traces, and then judge the adequacy of the resulting FSM in modelling the proposed system, thereby assessing the tests themselves.

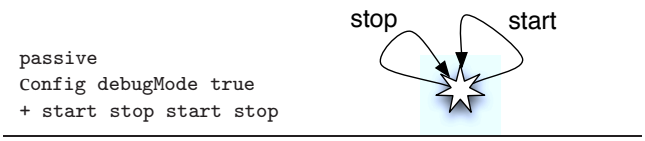


Figure 1. Start/stop behaviour: first model

3.3 Visualizing the state machine

In this section we demonstrate how we can use the StateChum library [19], to improve our set of unit tests by generating a Finite State Machine which represents the minimal FSM implicit in the tests. Inspecting that FSM allows us to decide which tests should be added (or indeed removed) in order to make the state space correspond to the intended model, and thus to establish the completeness of the test data set.

Translating EUnit tests to sequences

In order to use StateChum on a set of given EUnit tests, we need an algorithm to translate EUnit tests to sequences that are given as input to StateChum. The translation we start of with is to replace each `?assertMatch(`$Result$`,`$Fun(A_1, ..., A_n)$`)` in a test by the function name Fun to obtain a sequence of function calls. In particular, the `startstop_test()` above is translated into the sequence:

```
+ start stop start stop
```

where the leading '`+`' indicates that this is a *positive trace*, that is a trace that is to be accepted by the inferred FSM.

Note also that the algorithm used by StateChum assumes that the positive traces are closed under initial segments, so that the single trace is in fact equivalent to

```
+ start
+ start stop
+ start stop start
+ start stop start stop
```

Finally it should be noted that the transformation from the EUnit tests to the StateChum input can be fully automated.

3.4 Using the derived FSM to assess tests

In order to use StateChum on our example, we need to abstract from the data part in our test case and concentrate on the sequence of function calls performed. This sequence is input to StateChum and this input together with the derived FSM is shown in Fig. 1.

This figure indicates that there is a single state in which it is possible both to start and to stop the server. Starting and stopping the server don't result in a state change; at least, not on the basis of this single test case. In particular, the picture suggests that one can successfully perform a stop in the initial state, and also start the system twice.

In order to make two distinguishable states we need to supply StateChum with two *negative sequences*, which correspond to two negative test cases, that is, test cases that result in erroneous behaviour of some kind. The first test case verifies that one cannot stop in the initial state. This is added to the input for StateChum by adjoining the line `- stop`. After doing so, we observe an FSM with three states, depicted in Fig. 2.

In the initial state – indicated by a *starred* icon – a `stop` leads to the error state and a `start` leads to a second state. From that

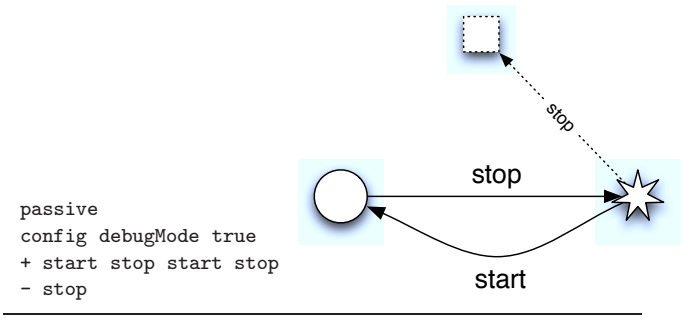


Figure 2. Start/stop behaviour: second model

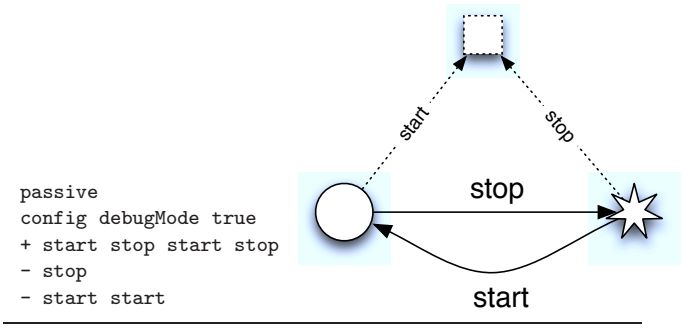


Figure 3. Start/stop behaviour: final model

second state there is a `stop` transition back to the initial state, but no further transitions. The third state is a 'dead state' – denoted by a *square* – and this is the result of a `stop` move from the initial state. The two traces: one negative and one positive, are insufficient to predict what happens when a `start` call is made in the second state.

So, we need to add another negative test case, stating that starting a system that is already running will result in an error. The new FSM derived is shown in Fig. 3, where we now have an extra transition to the error state. This picture describes the complete behaviour of the `start` and `stop` functions and all sequences thereof. Therefore, we are justified in concluding that the set containing one positive and the two negative tests is sufficient for testing the start/stop aspects of the system behaviour.

3.5 Writing negative tests in EUnit

When writing negative tests, we can either choose to specify which exception should occur, or just match on any possible exception. Many testers like the first alternative best, since one also tests whether the code fails for the right reason. However, in our case the reason is not specified, and, by adding it to the test we limit the possibilities in the design.

For example, if we were to decide that an initial `stop` raises an exception with reason `not_running` and were then to decide to implement the server using the standard OTP behaviour `gen_server`, then the error generated by the implementation would be a `noproc` exception rather than `not_running`, and so the negative test would fail. We could change the exception sought, but rather than overspecify the error exception, we choose the second alternative above and match on any possible exception.

```
stop_without_start_test() ->
     ?assertException(_,_,stop()).

start_twice_test_() ->
     {setup,
      fun() -> start([]) end,
      fun(_) -> stop() end,
      fun() -> ?assertException(_,_,start([])) end}.
```

If we stop a non running server, an exception is raised and starting an already running server similarly raises an exception. The reason for writing the last test case as a test generator[2] with set-up code, clean-up code and actual test code is that EUnit raises an exception as soon as the `?assertException` would fail, e.g., when the second start succeeds. In such cases one still wants to clean up and stop the already running server.

Translating EUnit tests to sequences

Although we used StateChum to derive a full set of tests by first supplying the negative sequences and then writing the additional test cases, we still strive after a translation from EUnit tests to these sequences. We extend therefore our translation in such a way that any command sequence in EUnit that ends with an `?assertException` is a negative tests and the translation of these assertions is given by:

$[$`?assertException(`$E1$`,`$E2$`,`Fun`(`$A_1, ..., A_n$`))`$] \rightarrow Fun$

$[\{$`setup,`
`fun() ->` $InitSeq$ `end,`
`fun() ->` $StopSeq$ `end,`
`fun() ->` Seq `end}`$] \rightarrow [InitSeq\ Seq]$

Thus ignoring the cleanup code and assuming at most the last assertion is an exception assertion, which determines the test to be a negative sequence.

Taking the examples given at the start of this subsection we generate the sequences:

```
- stop
- start start
```

as shown in Figure 3.

3.6 Initial implementation: start/stop behaviour

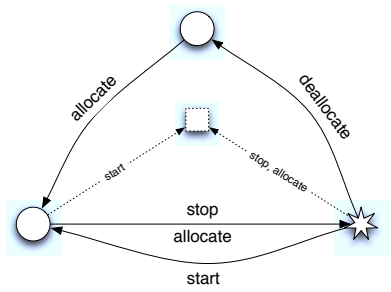
We can now run the tests and all three fail with notifications pointing to the fact that `start` and `stop` are as yet undefined! We now write the code for starting and stopping the server.[3] thus:

```
start(Freqs) ->
   {ok,Pid} =
     gen_server:start({local,?SERVER},
                      ?MODULE,Freqs,[]),
   Pid.

stop() ->
   gen_server:call(?SERVER,stop).

%% callbacks
init(Freqs) ->
    {ok, Freqs}.

handle_call(stop,_From, State) ->
    {stop, normal, ok, State};
handle_call(_Msg,_From,State) ->
    {reply,error,State}.
```

All tests pass and by having seen the correspondence between the test cases and the FSM in Fig. 3, we have strong confidence that adding more tests is superfluous and that we can proceed with

[2] Note the subtle addition of '_' after the function name, which transforms a direct test into a test generator. See [6] for details.

[3] An alternative implementation of the system is provided in Ch. 5 of [7].




```
passive
config debugMode true
+ start stop start stop
- stop
- start start
+ start allocate deallocate allocate stop
- start allocate allocate
```

Figure 4. Single frequency: second model

specifying the tests for the additional functionality of allocating and deallocating frequencies.

3.7 Defining tests for a data-dependent state space

After having defined test cases for starting and stopping the server, we would now like to allocate and deallocate frequencies. Whether or not allocation succeeds depends on the number of frequencies that are available. In other words, depending on how many frequencies we start with and how many allocations we perform, we get different successful and failing test cases.

Starting by defining a set of test cases for four frequencies would immediately result in a large number of possible allocation and deallocation scenarios, let alone taking a realistic example of several hundreds of resources. We therefore start by defining the tests for systems with one and two frequencies available and make sure that we get a complete set of tests for each of these, trusting that we can generalise from these to the general case.

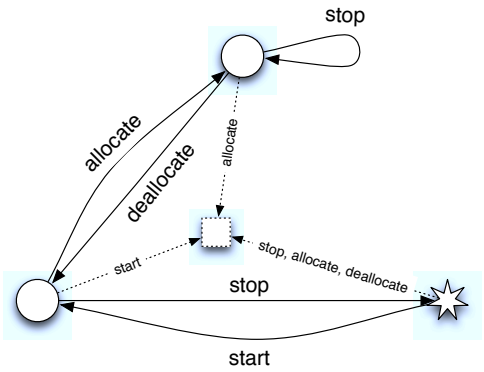
3.8 A single frequency

A typical test case would be to allocate a frequency and then deallocate it. Another typical test case would be to allocate it once more after deallocation. Since the first test case is subsumed in the second one, we only write the second.

```
alloc_dealloc_alloc_test_() ->
  {setup,
      fun() -> start([1]) end,
      fun(_) -> stop() end,
      fun () ->
         ?assertMatch({ok,1}, allocate()),
         ?assertMatch(ok,deallocate(1)),
         ?assertMatch({ok,1}, allocate())
     end
      }.
```

Note that the frequency value 1 used here is arbitrary. (We assume it to be more likely to find an error in the implementation by adding more different scenarios than by trying more different values for the specific frequencies.) This test must allocate the same value twice since there is only one value to be allocated.

We use StateChum again to visualize the FSM, which is equivalent to Fig. 3 with the addition of an arbitrary allocations and deallocations after starting the server. So, we do not capture the fact that it is possible to allocate all available frequencies and that an error is returned in that case. In order to add a general test case for the exhaustion of frequencies, we need to know how many frequencies there are. We propose to get the tests right for one frequency first, then take the two frequency case and see if we can generalise from there.




```
passive
config debugMode true
+ start stop start stop
- stop
- start start
+ start allocate deallocate allocate stop
- start allocate allocate
- deallocate
- allocate
```

Figure 5. Single frequency: third model

Using StateChum we can quickly observe what happens if we add a negative test for allocating two frequencies in case we only have one. The result is shown in Fig. 4 and it is immediately clear that we have to add a few more test cases to make a sensible picture out of this FSM.

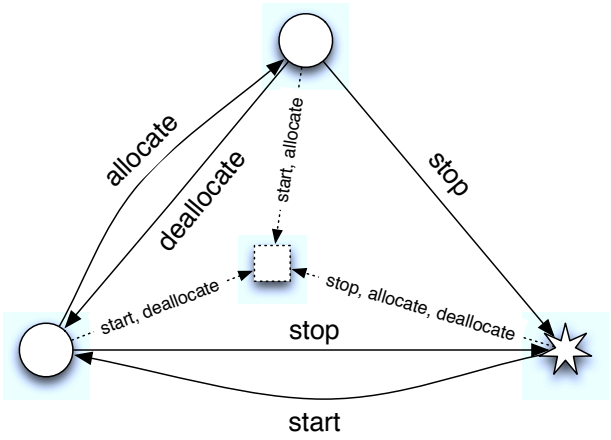
According to Fig. 4, from the initial state we can perform a `deallocate` and then an `allocate`. We need to exclude that possibility by stating that deallocation (and indeed also allocation) can only be done after a `start`; this results in Fig. 5.

In the FSM of Fig. 5 a `start` can only be followed by an `allocate`, which after deallocation allows a new allocation. The only strange part is that one can `stop` indefinitely often after allocation; one would like instead to have a `stop` transition back to the initial state. In fact, it is good to observe this in a visualisation of a state space, since it is domain-dependent whether or not one would allow a server that allocates frequencies to just stop or that one would need to deallocate the frequencies first. In Erlang it is most natural to perform the deallocation as side-effect of stopping. We add a test to ensure that we can start again after stopping with one allocated resource.

The tests[4] added for the server with one frequency are:

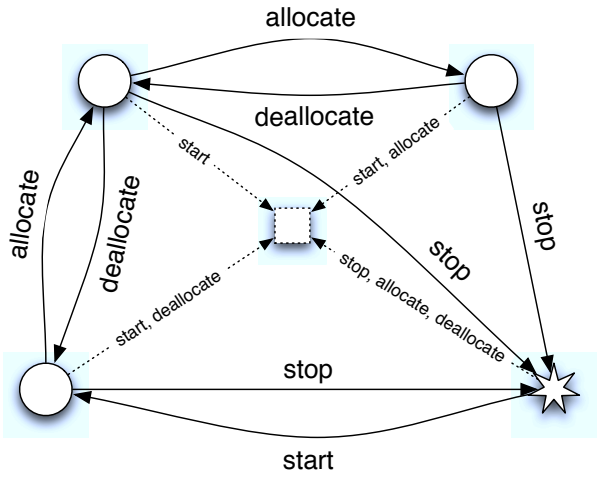
```
allocate_without_start_test() ->
    ?assertException(_,_,allocate()).

deallocate_without_start_test() ->
    ?assertException(_,_,deallocate(1)).

running_server_test_() ->
    {foreach,
     fun() -> start([1]) end,
     fun(_) -> stop() end,
     [fun() ->
       ?assertMatch({ok,1} ,allocate()),
       ?assertMatch(ok,deallocate(1)),
```

[4] EUnit allows to combine a few of these tests with the `foreach` primitive instead of `setup`




```
+ start stop start stop
- stop
- start start
+ start allocate deallocate allocate stop
- start allocate allocate
- allocate
- deallocate
+ start allocate stop start
- start deallocate
- start allocate start
```

Figure 6. One frequency: final model and StateChum test set




```
+ start stop start stop
- stop
- start start
+ start allocate allocate deallocate allocate
  deallocate deallocate stop
- start allocate allocate allocate
- allocate
- deallocate
+ start allocate stop start
+ start allocate allocate stop start
- start allocate deallocate deallocate
- start allocate start
- start allocate allocate start
```

Figure 7. Two frequencies: final model and StateChum test set

```
    ?assertMatch({ok,1},allocate())
   end,
  fun() ->
    ?assertMatch({ok,1} ,allocate()),
    ?assertMatch({error,no_frequency} ,allocate())
  end,
  fun() ->
    ?assertMatch({ok,1} ,allocate()),
    ?assertMatch(ok,stop()),
    ?assertMatch(Pid when is_pid(Pid),start([1]))
  end]}.
```

Note that in the above test cases we use domain knowledge to interpret the error value returned from allocation as a negative test case, expressing the condition that starting the server and performing two allocations is impossible. Were we to be given an API for our frequency server that raised an exception for a failing allocation, then the test case would be identified as a negative test case much more easily.

At this point we could conclude, if we were confident that all the transitions shown are as expected. However, the StateChum tool diagnostics for this input are:

```
#Prescribed: 5
#Proscribed: 5
#Unknown: 2
```

This output states that, of the twelve possible transitions in the machine, five make a transition to an accepting state and another five to the dead state: two transitions are as yet undetermined. The two transitions in question are: whether it is possible to deallocate before any allocation, and, whether it is possible to start the machine again after the one frequency is allocated.

We can rule these out with two negative test sequences that come at the end of the complete set of cases listed in Fig. 6 and these generate the machine in that figure. The data might appear to be skewed in favour of the negative tests: there are 7 negative and 3 positive tests. However, noting the prefix-closure property of the positive tests, we can see these three tests as embodying 10 distinct positive test cases, and under this interpretation we have of the same order of positive and negative tests.

Translating EUnit tests to sequences

We need to extend the translation of EUnit tests to the `foreach` construct, which is equivalent to the translation of several `setup` commands. In addition we have to add that an assertion that matches an error produces a negative sequence. As explained, this is somewhat controversial and probably one would like to enforce the design to raise an exception instead.

3.9 Two frequencies

Now we look at the case where there are two frequencies to be allocated, and develop a set of tests along the lines of the one frequency machine in Section 3.8. The set of tests – described in StateChum input format – are shown in Fig. 7.

The greyed-out tests are identical to the previous case, while the other tests are developed by a similar process to that in Section 3.8. Counting distinct prefixes as separate tests, we have 15 positive tests and 8 negative ones. A number of the later tests are included to avoid loops, such as looping on stop behaviour rather than returning

the system to the start state when it is stopped; others are to prevent starting a system that is already running, whatever state it is in.

Note that in the EUnit tests the specific frequency that we allocate and deallocate was not significant when there was just one frequency available. However, now that we have two frequencies to choose from, a choice has to be made about which frequency is to be allocated. Now we have either to specify in our test case how the algorithm implements the choice, or to abstract away from the allocation algorithm. In EUnit tests this difference manifests itself as the difference between the following two test cases. In the first case, the test requires an implementation that takes frequencies from the head of the list:

```
twofreq_server_test_() ->
    {setup,
     fun() -> start([1,2]) end,
     fun(_) -> stop() end,
     fun() ->
       ?assertMatch({ok,1} ,allocate()),
       ?assertMatch({ok,2},allocate()),
       ?assertMatch(ok,deallocate(2)),
       ?assertMatch({ok,2},allocate()),
       ?assertMatch(ok,deallocate(1)),
       ?assertMatch(ok,deallocate(2))
      end}.
```

The alternative is a test that does not enforce any order on the allocation of frequencies:

```
twofreq_server_test_() ->
    {setup,
     fun() -> start([1,2]) end,
     fun(_) -> stop() end,
     fun() ->
       ?assertMatch({ok,F1} ,allocate()),
       ?assertMatch({ok,F2},allocate()),
       ?assertMatch(ok,deallocate(F2)),
       ?assertMatch({ok,F3},allocate()),
       ?assertMatch(ok,deallocate(F1)),
       ?assertMatch(ok,deallocate(F3))
      end}.
```

The latter test seems preferable in a test-driven development process, since it does not over-specify implementation details. Moreover, if the set of frequencies is extended to contain more than two frequencies, the test makes still sense without having to re-evaluate how the choice of frequencies is actually implemented. In this case, it is likely that re-use of frequencies is preferred to assigning as-yet-unused frequencies.

3.10 Data abstraction

With the translation of EUnit tests to sequences for StateChum we abstract from the data in the EUnit test cases. According to the API of the frequency server, the `start` and `deallocate` operations are parameterised by a list of frequencies and the frequency to be deallocated, respectively. These parameters play different roles.

- The list parameter is the `start` value for the particular run of the server, and it can be any legitimate integer list; of course its size will constrain the behaviour of the system, but the call to `start` is bound to succeed if and only if the system is not already running. This (pre-)condition is encapsulated in the structure of the FSMs seen in Figs. 6 and 7.
- On the other hand, the parameter to `deallocate` is assumed to be a frequency that is already allocated. This condition is not something that can be modelled in the FSM without 'hard wiring' the set of frequencies into the FSM itself. Supposing that there are n frequencies available, this would give rise to some 2^n states, each one representing a different subset of the n states having been allocated.

So, we can safely abstract in our EUnit tests from the specific frequency that is returned by `allocate`, i.e. we do not need to know the exact allocation algorithm. But, we cannot easily abstract from the specific frequency that is passed to `deallocate`; that frequency has to be remembered in our test case. Therefore, the abstraction

```
- start allocate deallocate deallocate
```

is only a valid abstraction if both deallocations refer to the same frequency. This means that the translation from EUnit test cases to sequences that we have developed fails in some cases, such as this:

```
twofreq_server_test_() ->
    {setup,
     fun() -> start([1,2]) end,
     fun(_) -> stop() end,
     fun() ->
       ?assertMatch({ok,F1} ,allocate()),
       ?assertException(_,_,deallocate(3-F1))
      end}.
```

One solution would be the 'hard-wiring' of the frequencies discussed earlier, which would involve two allocation operations, `allocate1` and `allocate2` and two deallocation operations, `deallocate1` and `deallocate2`. However the state machine resulting from that approach suffers from an exponential state explosion (as described earlier).

Instead we use another abstraction. We can 'loosen' our model, so that `deallocate(N)` can be applied whether or not `N` has been allocated or not. A problem with this is that this makes the FSM non-deterministic, since in the case that `N` is not already allocated the result of the transition will be that the set of available states is unchanged.

We can then interpret an exception for a deallocation as a possibility in a positive sequence, which is similar to changing the API for the `deallocate` function so that

```
-spec deallocate(integer()) -> ok | error.
```

with the `error` result indicating that no actual reallocation has taken place, because the argument frequency was not allocated. We can then distinguish the normal termination and error termination by translating the EUnit

```
?assertMatch(error,deallocate(...))
```

into a `failDA` operation. This would restore determinism in the model. Taking this approach, we add the `failDA` transition and the following test cases to those in Fig. 7

```
+ start failDA stop
+ start failDA failDA stop
- start allocate failDA allocate allocate
- start allocate allocate failDA
- failDA
```

and obtain the state machine of Fig. 8. However, the translation of EUnit tests to sequences needs to be adapted to treat certain error cases as part of a positive sequence and others as making the test case negative. This requires the user to specify the differences and therefore this method is not entirely satisfactory if full automation is the goal.

As shown in Fig. 8, this resolves all the `failDA` operations, which are only permissible when zero or one frequencies have been allocated. The labels on the transitions to the 'dead' state have been elided for readability in the figure.

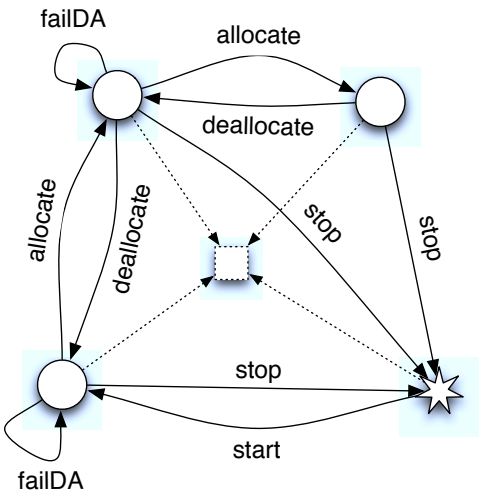


Figure 8. Two frequencies with failed deallocation, `failDA`

3.11 One, two, many ...

A pattern is emerging in Figs. 6 and 7: an FSM to model the server with n frequencies will have $n+3$ states: an initial state, a dead state and $n+1$ states representing the different numbers of frequencies that have been allocated.

We contend that the case of allocation from a set of two frequencies should be sufficient to test the general case, since it allows us to examine the case of allocation and deallocation when some frequencies have been allocated and some not.

Of course, it is possible for an implementation to have special case 'Easter egg'[5] behaviour for particular collections of frequencies, but any finite set of tests will be vulnerable to this. So, making the assumption that our implementation is generic in the frequency set we repeat our contention. Probably a careful tester would extend the model to contain three resources in order to be able to test re-use of a frequency in the middle, but it seems a large investment to go any further than that. We have already to define 17 EUnit tests to capture the behaviour of two frequencies and 20 to capture the behaviour of three frequencies. This corresponds to about 100 lines of Erlang test code for an implementation that is itself smaller than that.

If one is interested in testing even more possible combinations of allocating and deallocating resources, one would rather generate a large number of random combinations for a random collection of frequencies. We can do precisely this by using the QuickCheck finite state machine library to generate the test cases.

4. QuickCheck finite state machine

QuickCheck comes with a library (`eqc_fsm`) for specifying test cases as finite state machines. Given a few callback functions for this state machine, the QuickCheck machinery is able to generate and run test cases that are sequences generated from these callback functions.

Here we present an approach to generate QuickCheck state machine specifications from EUnit tests in contrast to the more common manual generation from informal specifications of the software under test. The advantage of using QuickCheck, as we will see in this section, is that with little extra effort, we get many new tests cases that actually test meaningful sequences, not covered by the EUnit tests.

As demonstrated in the previous section, we use StateChum to generate a state machine from the EUnit tests in order to obtain states names and their transitions. We may potentially improve the state machine by adding tests cases, but, as explained before, the data part restricts us to test cases with only little data involved. Now we translate the obtained state machine in a QuickCheck specification.

4.1 Sequence of calls

Each state obtained by StateChum is translated into a unary Erlang callback function[6] that returns a list with possible next states and the transitions thereto. For example, the state machine described in Fig. 3 has 3 states; state names are randomly chosen by the tool, but manually translated into something meaningful, say *init*, *started* and *error*. From *init* there are two possible transitions and in the QuickCheck library `eqc_fsm`, this is specified thus:

```
init(_) ->
  [ {started,{call,?MODULE,start,[nat()]}},
    {error,{call,?MODULE,stop,[]}}].

started(_) ->
  [ {error,{call,?MODULE,start,[nat()]}},
    {init,{call,?MODULE,stop,[]}}].

error(_) ->
   [].
```

Note that Fig. 3 has no arguments for the functions; this information is present in the EUnit tests, but not in the abstracted state machine. Therefore, we have to retrieve it from the EUnit tests. At this moment we may realize that starting the server with an empty list and a list with only one element has been a completely arbitrary choice in our EUnit tests. In fact, we would like to start the server with an arbitrary, positive number of frequencies `nat()`.

Each transition is encoded as a tuple with first argument the name of the next state and as second argument a symbolic call to an Erlang function, in this case the `start` and `stop` functions in the module (`?MODULE`) we define our specification in, which differs from the implementation module `frequency.erl`. The reason for a local version of the start and stop function is that we expect these to potentially raise an exception and similar to the `assertException` in EUnit we have to notify QuickCheck that exceptions may be valid. Moreover, we use a maximum number of frequencies to compute the list with consecutive sequences in the start function.

```
start(Freqs) ->
    catch frequency:start(lists:seq(1,Freqs)).

stop() ->
    catch frequency:stop().
```

In the EUnit tests, the return values of the calls to start and stop are checked in the assertions. These assertions translate into postconditions in the QuickCheck specification. Postconditions are callbacks with five arguments: a `From` state, a `To` state, the data in `From` state, the symbolic call and the result of that call. Thus, we check that indeed the positive calls return the right value and that whenever we enter the error state, it was because of a call that raised an exception.

```
postcondition(init,started,_,{call,_,start,_},R) ->
    is_pid(R);
```

[5] Some hidden message or feature, coded as a surprise in software and other artefacts.

[6] The argument of the state is the state data.

```
postcondition(started,init,_,{call,_,stop,_},R) ->
    R == ok;
postcondition(_From,error,_,{call,_,_,_},R) ->
    case R of
       {'EXIT',_} -> true;
       _ -> false
    end.
```

Finally, we need to write a QuickCheck property to run the test cases. First an arbitrary sequence of start and stop commands is created using the state machine description and then that sequence is evaluated. In order to make sure that we start in a known state (even if a previous test has failed), we both stop the frequency server at the beginning and end of each test, relying on the catch when the server is not running.

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

4.2 Adding state data

The advantage of running many different sequences of starting and stopping the server may not be so obvious for this example. The real benefit of using a QuickCheck state machine specification shows when the state data is used to represent the allocated frequencies.

We choose to use the state machine from Fig. 7 as our starting point. In the state *started* we should add a transition to a state in which one frequency is allocated. From that new state, we create a transition to yet another one where two frequencies are allocated, etc. Of course, the state names have to be generalised and we use QuickCheck's support for parametrized states, i.e. each state is represented by a tuple of which the first argument is the state name and the second argument is a parameter, the number of allocated frequencies in our case.

Note that the state machine in Fig. 7 was obtained from tests with two frequencies and is in fact an abstraction of tests with two allocations. We would like to generalise this to an arbitrary number of frequencies, but start with setting a maximum of 2 for the moment.

```
-define(MAX,2).
```

We introduce a record to represent an abstraction of the state of the frequency server: the free frequencies and the used frequencies.

```
-record(freq,{used=[], free=[]}).
```

We rename the state *started* into `allocated` and add appropriate transitions. We fix the maximum number of allocations to 2 and deallocation of frequencies that have not been allocated is smoothly added as a transition.

```
init(_) ->
    [ {{allocated,0},{call,?MODULE,start,[?MAX]}},
      {error,{call,?MODULE,stop,[]}}
     ].

allocated(N,S) ->
    [ {error,{call,?MODULE,start,[nat()]}} ] ++
    [ {{allocated,N+1},{call,?MODULE,allocate,[]}}
                      || N < ?MAX] ++
    [ {error,{call,?MODULE,allocate,[]}}
                      || N == ?MAX] ++
    [ {{allocated,N-1},{call,?MODULE,deallocate,
                        [elements(S#freq.used)]}}
                      || N > 0] ++
    [ {init,{call,?MODULE,stop,[]}}].

error(_) ->
    [].
```

The list comprehensions are used to lazily compute the state parameter and only include the alternatives that are valid for that particular state. Starting an already started server may take any argument, hence no `?MAX` there but an arbitrarty positive number.

The deallocation functions depends on the state data. As an argument to `deallocate` we supply an arbitrary element of the list `S#freq.used`.

In order to successfully test these cases, QuickCheck need to know more about the state data. This is achieved by defining callback functions that operate on the data.

The state data gets modified by the `next_state_data` callback function, which takes five arguments. The first argument is the state from which the transition originates and the second argument the state that the transition leads to. The third argument is the state data, i.e., the record that we defined above. The fourth argument is the (symbolic) result of the evaluation of the symbolic call in the last argument.

```
next_state_data(_,_,S,V,{call,_,start,[Max]}) ->
    S#freq{used=[], free=lists:seq(1,Max)};
next_state_data(_,_,S,V,{call,_,allocate,[]}) ->
    case S#freq.free == [] of
         true -> S;
         false ->
            S#freq{used=S#freq.used++[V],
                   free=S#freq.free--[V]}
    end;
next_state_data(_,_,S,V,
                {call,_,deallocate,[Freq]}) ->
    S#freq{used=S#freq.used--[Freq],
           free=S#freq.free++[Freq]};
next_state_data(_,_,S,V,{call,_,stop,[]}) ->
    S#freq{used=[], free=[]}.
```

In this way, we know which frequencies are allocated and which are free. Note that if all frequencies are allocated, then an allocation will result in an error and the state stays unchanged.

Similar to the start and stop command before, we add local commands for allocation and deallocation. This time we use the local function to modify the return value, since our model is cleaner when we get a frequency returned from `allocate`:

```
allocate() ->
    case frequency:allocate() of
         {ok,Freq} -> Freq;
         Error -> Error
    end.

deallocate(Freq) ->
    frequency:deallocate(Freq).
```

Finally, we add postconditions for allocation and deallocation to complete our QuickCheck specification.

```
postcondition(_,_,S,{call,_,allocate,[]},R) ->
    case R of
        {error,no_frequency} ->
          S#freq.free == [];
        F when is_integer(F) ->
          lists:member(F,S#freq.free)
    end;
```

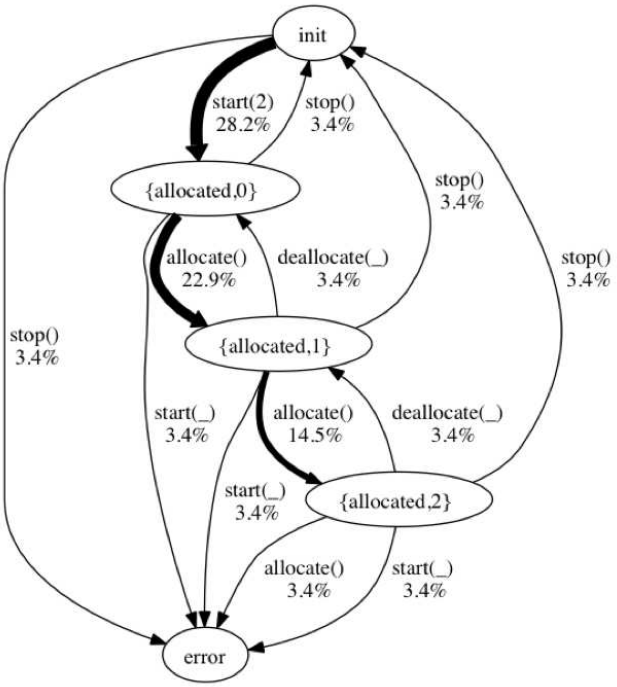


Figure 9. Visualization of QuickCheck specification

```
postcondition(_,_,S,{call,_,deallocate,[Freq]},R) ->
    R == ok;
```

This specification can be used to generate many different sequences of calls to start, allocate, deallocate and stop. QuickCheck can compute a fair distribution for the occurrences of the commands, such that we increase the likelihood to obtain sequences that indeed allocate all available resources instead of just starting and stopping the server all the time. In a visualization of the QuickCheck state machine the weights for each transition are provided as a percentage (see Fig. 9).

A typical example generated with this state machine could be a test case like:

```
{set,{var,1},{call,frequency_eqc,start,[2]}},
{set,{var,2},{call,frequency_eqc,stop,[]}},
{set,{var,3},{call,frequency_eqc,start,[2]}},
{set,{var,4},{call,frequency_eqc,stop,[]}},
{set,{var,5},{call,frequency_eqc,start,[2]}},
{set,{var,6},{call,frequency_eqc,allocate,[]}},
{set,{var,7},{call,frequency_eqc,allocate,[]}},
{set,{var,8},{call,frequency_eqc,deallocate,
                                        [{var,6}]}},
{set,{var,9},{call,frequency_eqc,allocate,[]}}
```

4.3 Additional error transitions

There is still a subtle difference between the QuickCheck state machine in Fig. 9 and the state machine obtained from EUnit tests in Fig. 7, viz. a number of transitions to the error state are missing.

For example, the deallocation in the state with zero allocated frequencies leads to the error state in Fig. 7. We have neglected this case in our specification, but we can add it by adding one more transition to the state defining callback function `allocated`:

```
[ {error,{call,?MODULE,deallocate,
       [elements(S#freq.free)]}} || N == 0] ++
```

Note that we must pick a frequency from the free frequencies, since none is in use yet. Alternatively, we could take any arbitrary frequency using `nat()`.

The `next_state_data` function can stay as is since we jump to the error state and no more transitions are allowed from there, hence the specific state of the server is not important. The postcondition has, of course, to be adapted, since a transition to the error state should be caused by an exception:

```
postcondition(_,To,S,{call,_,deallocate,[Freq]},R)
                                  when To =/= error ->
    R == ok;
```

Tests generated from this specification expect an exception raised when we deallocate after starting the server. We need to add a catch in the local function of `deallocate` as well. However, when we run the tests against our frequency implementation, we obtain immediate feedback from QuickCheck that the postconditions of this deallocation is falsified. In other words, our implementation follows the specification and indeed always had `deallocate` return `ok`.

Inspecting the EUnit test cases shows that indeed we never test starting the server and then deallocating. The transition in Fig. 7 was added because of insufficient information. In fact, one can argue that the transition should not be there at all, but that incorrect deallocations are either not allowed, which should be guaranteed by the clients of the server, or that the specification of the API should be enriched with a possible error result for `deallocate`.

Rather would we now add a transition that deallocation of free resources should have no effect. This can be done by adding another transition to the state machine:

```
[ {{allocated,N},{call,?MODULE,failDA,
                       [elements(S#freq.free)]}}] ++
```

We use `failDA` instead of `deallocate` to avoid getting ambiguous transitions in the state machine. QuickCheck cannot compute good test case distribution when the model is ambiguous.

The `failDA` function is simply calling the deallocation in the implementation module. The next state function for `failDA` leaves the state untouched and the postcondition checks that an `ok` is returned. When running QuickCheck with this property we found an error in our implementation, since we expected the clients to obey the rule that they would not release the same frequency twice and always added a released frequency to the list of available frequencies. This gave a list with duplicates in the newly constructed test cases and the postcondition for `allocate` found the mismatch by checking that the given frequency is indeed free.

4.4 Increasing number of frequencies

With the definition of the macro `MAX` we can now easily create a state machine that tests sequences that have 4 frequencies and all possible combinations of allocations and deallocations. The only thing to do is to recompile the code with a larger constant. But, testing with a small number of frequencies thoroughly may reveal more faults than when testing a larger number of frequencies in a less exhaustive manner.

The QuickCheck specification is about 100 lines of code, which is similar to an exhaustive EUnit test suite, but it covers a wider range of tests. For larger, more realistic example, the size of the QuickCheck specification tends to grow less fast than an EUnit test suite does.

5. Related work

In this section we examine related work in test-driven development, grammar inference and testing methodologies.

Test-driven development

As we mentioned in Section 2, Beck's [4] answers a number of frequently-asked questions. In replying to *"How many tests should you write?"* he provides a simple example of a function to classify triangles: this elicits an answer inspired by equivalence partitioning. No state-based systems are discussed. The question *"How do you know if you have good tests?"* relates to the quality of individual tests, rather than the effect of the collection as a whole.

Fowler advocates mutation testing as a mechanism for assessing the adequacy of a set of tests [21]. Astels [3] in discussing TDD for Java also advocates mutation testing with Jester [13], as well as code coverage analysis with Clover [9] and NoUnit [16].

Of course, these methods can only be used when there is an implementation to hand. In the context of TDD there is a circularity to this, since the implementation has been developed specifically to meet the set of tests. By contrast, our method gives feedback on the test set independently of any implementation.

Random testing

Random testing for functional programs was first introduced by QuickCheck for Haskell [8] and more recently developed for Erlang [1]. It has also inspired related systems for Scheme, Standard ML, Perl, Python, Ruby, Java, Scala, F# and JavaScript.

QuickCheck testing is based on the statement of logical *properties* which are then tested for random inputs generated in a type-based manner. Simple logical statements of properties suffice for functional behaviour; state based systems are tested by driving them from an FSM which gives an abstract model of the system.

Fuzz testing or fuzzing [20] is a related technique used particularly with protocol testing, an area where QuickCheck FSMs can also be used. Fuzzing is a "brute force" approach, typically generating inputs at random, rather than having their generation being guided by a model such as an FSM. Fuzzing is perceived, however, as a mechanism providing a high benefit:cost ratio.

A comprehensive overview of other approaches to random testing is given in Pacheto's thesis [17]. Pacheto's thesis also examines ways that random testing can be 'directed' with extra tests being generated as a consequence of examining the results of already executed test cases.

Inference and testing

There is a substantial literature on inferring higher-level structures from trace or event-based data. Among the earliest is Cook and Wolf's [10] which infers an FSM from event-based trace data. More recent work by Artzi *et. al.* [2] uses those techniques to general legal test inputs – that is legal sequences of calls to APIs – to OO programs, again based on execution traces; this paper also provides a useful overview of other work in this area. Walkinshaw and others [24] use the Daikon tool [12] as part of an interactive process of model elicitation.

Daikon implements invariant inference, and has been extended to the DySy tool [11] which augments the Daikon approach based on test set execution with dynamic symbolic execution. Xie and Notkin [26] infer specifications from test case executions, and based on this develop further test cases.

Our approach differs from these in being based on the test cases themselves rather than on their execution: it can therefore be used independently of any implementation.

The Wrangler refactoring tool for Erlang [25] provides clone detection and elimination facilities [15], and in the latest release (0.8.8) implements the facility to transform a cloned test into a QuickCheck property, thus generalising the range of possible tests of the system.

6. Conclusions and Future Work

We have shown the value of extracting the finite state machine implicit in a set of EUnit tests not only for understanding the adequacy of the tests developed as a part of the process of test-driven development but also in defining a QuickCheck FSM which can be used for property-based testing of the system under test. In doing this we noted a number of points.

- The negative tests – that is those that lead to an `error` value of some sort, raise an exception or cause another form of error – are as important as the positive tests in delimiting the correct behaviour of the system implicit in the tests. This is due in part to the nature of the extraction algorithm [23] but is also due to the fact that without these tests there would be no explicit bounds on the permissible behaviour.
- We assume that we can extract the call sequences within tests by static examination of the test code. This is not unreasonable since many test cases consist of straight line code, particularly for the state-based systems that we examine here.
- Some aspects of the process can be automated with ease, including the extraction of the function call sequences and the naive conversion of an FSM into QuickCheck notation. Others require manual intervention, including the choice of data values for the 'small' states and the choice of state data for the QuickCheck FSM.
- Given that the model we develop is an abstraction of the actual system, it is natural for non-determinism to creep into the model. This can be resolved by renaming some of the transitions to avoid non-determinism. The old and new transitions can then be seen as having pre-conditions which will be explicit in the QuickCheck model.

The next step for us to take is to refine the process described here into a procedure which automates as much as possible of the FSM development. This will allow QuickCheck properties for state-based systems to be extracted from tests in a semi-automated but user-guided way.

We would like to acknowledge the support of the European Commission for this through the FP7 Collaborative project *ProTest* [18], grant number 215868.

References

[1] T. Arts *et. al.* Testing Telecoms Software with Quviq QuickCheck In Proceedings of the Fifth ACM SIGPLAN Erlang Workshop, ACM Press, 2006.

[2] S. Artzi *et. al.* Finding the Needles in the Haystack: Generating Legal Test Inputs for Object-Oriented Programs. In *M-TOOS 2006: 1st Workshop on Model-Based Testing and Object-Oriented Systems*, 2006.

[3] D. Astels. *Test-driven Development: A Practical Guide.* Prentice Hall, 2003.

[4] K. Beck. *Test-driven Development: By Example.* Addison-Wesley, 2002.

[5] R. Carlsson EUnit - a Lightweight Unit Testing Framework for Erlang. In *Proceedings of the fifth ACM SIGPLAN Erlang Workshop*, ACM Press, 2006.

[6] R. Carlsson and M. Rémond. *EUnit - a Lightweight Unit Testing Framework for Erlang.* `http://svn.process-one.net/contribs/trunk/eunit/doc/overview-summary.html`, last accessed 07-06-2010.

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

[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, ACM Press, 2000.

[9] Clover. *Clover: Java code coverage & test optimization.* `http://www.atlassian.com/software/clover/`, last accessed 07-06-2010.

[10] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. In *ACM Transactions on Software Engineering and Methodology*, 7, 1998.

[11] C. Csallner *et. al.* DySy: Dynamic Symbolic Execution for Invariant Inference. In *ICSE08*, ACM Press, 2008.

[12] M. D. Ernst *et. al.* The Daikon system for dynamic detection of likely invariants. In *ACM Transactions on Software Engineering and Methodology*, 69, 2007.

[13] E. R. Harold. *Test your tests with Jester.* `http://www.ibm.com/developerworks/library/j-jester/`, last accessed 07-06-2010.

[14] Hughes, J. QuickCheck Testing for Fun and Profit. In: 9th Int. Symp. on Practical Aspects of Declarative Languages, Springer (2007)

[15] H. Li and S. Thompson. Similar Code Detection and Elimination for Erlang Programs. In *12th Int. Symp. on Practical Aspects of Declarative Languages*, Springer LNCS 5937, 2010.

[16] NoUnit. `http://nounit.sourceforge.net/`, last accessed 07-06-2010.

[17] C. Pacheco *Directed Random Testing*. Ph.D. thesis. MIT Department of Electrical Engineering and Computer Science, 2009.

[18] ProTest. `http://www.protest project.eu/`, last accessed 07-06-2010.

[19] StateChum. `http://statechum.sourceforge.net/`, last accessed 07-06-2010.

[20] M. Sutton, A. Greene, P. Amini *Fuzzing: Brute Force Vulnerability Discovery*, Addison Wesley, 2007.

[21] B. Venners. *Test-Driven Development: A Conversation with Martin Fowler, Part V.* `http://www.artima.com/intv/testdrivenP.html`, last accessed 07-06-2010.

[22] E. Vidal. Grammatical inference: An introductory survey. In *Grammatical Inference and Applications*, LNCS 862, Springer, 1994.

[23] N. Walkinshaw *et. al.* Reverse-Engineering State Machines by Interactive Grammar Inference. In *14th IEEE Working Conference on Reverse Engineering (WCRE'07)*, IEEE Press, 2007.

[24] N. Walkinshaw *et. al.* Iterative Refinement of Reverse-Engineered Models by Model-Based Testing. In *FM'09*, volume 5850 of Lecture Notes in Computer Science, Springer, 2009.

[25] Wrangler. `http://www.cs.kent.ac.uk/projects/wrangler/`, last accessed 07-06-2010.

[26] T. Xie and D. Notkin. Mutually Enhancing Test Generation and Specification Inference. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, LNCS Vol. 2931, Springer, 2003.