CHALMERS

Improving FSM reverse engineering for test development in Erlang

Master of Science Thesis in Software Engineering

PABLO LAMELA SEIJAS

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2011

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Improving FSM reverse engineering for test development in Erlang

PABLO LAMELA SEIJAS

© Pablo Lamela Seijas, May 2011.

Examiner: Thomas Arts

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2011

Abstract

This thesis contributes with a new implementation of an algorithm that infers a finite state machine from samples of a grammar, and with an automated system able to apply this algorithm to aloud the visualization of EUnit test cases, without the need to execute them.

This paper documents the experience derived from the process, as well as the technique used to test this new implementation against an existing one by using a blackbox approach.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Thomas Arts, for his shared knowledge, for the invaluable guiding along the whole project and for always being available despite his tight schedule.

Special thanks to Hans Svensson and Simon Thompson for their contributions, support and ideas. Their experience helped greatly to keep this project on track.

I would also like to thank the professors Victor Gulías, Francesco Cesarini, Antonio Blanco and José María Molinelli, whose teachings have proven so useful during all this time.

And last but not least, I would like to thank my family and friends, for their support and help.

Pablo Lamela Seijas, Göteborg, May 2011

Contents

1 Introduction **1**

2 Erlang QSM **3**

2.1 Initialization 4
2.2 State merging 6
2.3 Additional considerations 8

3 Testing **9**

3.1 QuickCheck testing 9
3.2 Interfacing StateChum 10
3.3 Differences with StateChum 11

4 EUnit parsing **13**

4.1 Possible scenarios 13
4.2 Syntax analysis 14

5 Related Work **17**

6 Conclusion and future research **18**

Bibliography **19**

A Source code for merge module **20**

1

Introduction

ERLANG programmers test their code by using, among other tools, EUnit tests. If we follow test-driven development approach we are encouraged to write those tests before starting the implementation, this way we try to avoid writing unnecessary code[1]. We can see the completeness of that testing by generating a finite state machine (FSM) from the traces produced by our tests [2]. If generated FSM fully models our intuition, then we have written enough tests.

The generation of the FSM is based upon an algorithm to extract regular grammars from samples of a language [3]. This algorithm, (QSM,) takes as input both words in the language and words not in the language and infers a regular grammar from them. The QSM algorithm is implemented in the program StateChum [4]. We can create a model of our software by collecting positive and negative traces and supplying them to StateChum as input words. [5] StateChum will show as output a graph of the FSM that represents the inferred grammar.

The earlier mentioned approach [2] manually translates EUnit tests into positive and negative traces:

For example, if we have the following code:

```
startstop_test() ->
    ?assertMatch(true,start([])),
    ?assertMatch(ok,stop()),
    ?assertMatch(true,start([])),
    ?assertMatch(ok,stop()).
```

We can extract the positive trace: `[start, stop, start, stop]`

If any of that sequences leads to an exception we generate a negative trace instead. Negative traces are necessary to get a complete test-suite and consequently they are also necessary for this method to give a positive result.

If we collect enough traces, we can feed them as input to the program StateChum[4] and it will show us a diagram of an FSM that will represent the expected behaviour of our system. StateChum is an implementation of an algorithm (QSM[3]) which explains how to extract a general regular grammar from samples of a regular language.

Nevertheless, StateChum[4] works with sets of traces, not with EUnit tests. Because of that, we are forced to do the described translation manually. This would not be a problem if we only have a few tests, but it can imply a big amount of work and related errors when we apply the method to a normal sized project.

Our contribution is completing the automation of the whole process by implementing QSM in Erlang, (as alternative to StateChum tool,) and by adding support for the EUnit tests in it. This will also greatly simplify the task of generating FSM from existing projects that already have EUnit tests implemented. We also contribute with a method by which we use QuickCheck to test our tool against StateChum in order to ensure a similar quality in the results.

2

Erlang QSM

The already existent QSM implementation, StateChum, was available to us as a binary. We could not access the source code but we could access two papers written by its developers [6][5]. In these papers they describe how they used an implementation of the QSM algorithm [3] to reverse engineer software. We used the description in those three papers to re-implement this algorithm in Erlang and the binaries to test our implementation, as we explain in the next section.

The algorithm works in terms of regular languages, when using it to reverse-engineer a program, we will consider traces (execution sequences) as words in the language. The QSM algorithm takes two sets of words, one set with words from the language, (valid sequences of events,) and one with words that do not belong to the language, (invalid sequences). With one peculiarity, invalid sequences are invalid strictly because of its last symbol, thus, if we remove the last event from an invalid word, we should get a valid one. We assume this because an invalid sequence is a sequence that produces an exception. There is no point in considering what happens after the exception is thrown, because of this we consider that the event that produces the exception is the last one. From this input consisting in two sets of words, QSM will try to produce the most general automaton that complies with the traces, and this will hopefully give us an idea of the completeness of our tests.

In our implementation we take as event any Erlang term. For explaining the algorithm we used atoms as events. Thus, a trace corresponds to a list of atoms, and the input to the algorithm is a tuple of two lists of lists of atoms, (the positive ones first).

In this paper we will use this set as example:

Positive	`[a,b,a]`
	`[b,b,a,b]`
Negative	`[a,b,c,c]`

Which is represented in Erlang by:

```
{[[a,b,a], [b,b,a,b]], [[a,b,c,c]]}
```

So, we have two possitive traces and one negative trace where the first two events of the negative trace also occur as part of a positive trace.

The QSM algorithm roughly consists of two phases[3]. In the first one, called *initialization*, we create a finite state machine with a tree structure (called APTA) that will accept all positive traces and reject all negative ones. In the second phase, *state merging*, we merge nodes of the tree in order to get a smaller finite state machine which is still deterministic and accepts and rejects the same input set, but possibly more.

2.1 Initialization

The Augmented Prefix Tree Acceptor (APTA) is the tree that we will use as our initial FSM. It must necessarily have a tree shape, it must accept all positive traces and reject all negative traces and it must also be deterministic (this is, there cannot be two branches with the same symbol departing from the same node).

For example, from the previous traces we would get the following APTA:

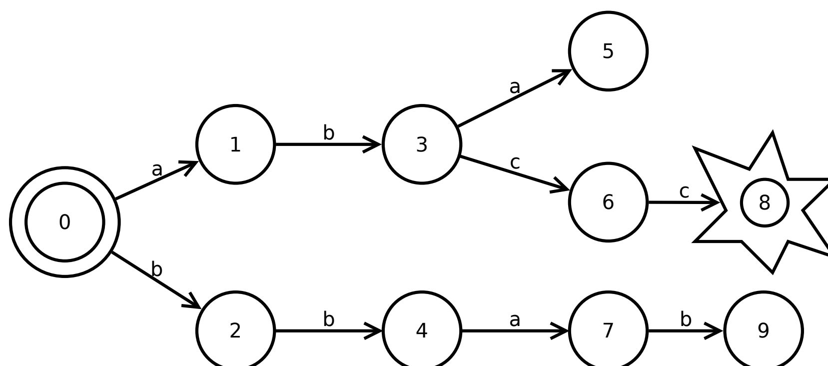


Figure 2.1: Example APTA tree

Where 0 is the initial state and 8 a failing one.

In order to generate this in Erlang we create an initial state with all the traces in it and extract the first event from each trace. Then we create as many states as different events we extracted and divide the rests of the traces between the new states.

In the first iteration of our example we would get:

State	Kind of trace	New trace
1 (from a)	Positive	`[b, a]`
	Negative	`[b, c, c]`
2 (from b)	Positive	`[b, a, b]`

We repeat the process with every new generated state until the states we generate do not contain traces. When we arrive to the end of a positive trace we just remove the trace, but when we fetch the last event of a failing trace we generate a new failing state and check that there are no traces left to expand from that state.

The target is to obtain the automaton as a record with the fields: initial state, alphabet, states, transitions and failing states. The transitions are stored using the labelled transition system (LTS), as a list of tuples in the form: origin, event, destination. In our example the transitions would look like:

```
[{0, a, 1}, {0, b, 2}, {1, b, 3}, {2, b, 4}, ... , {7, b, 9}]
```

We also decided to keep some order in the numbers of the states to simplify the implementation of the *state merging* later, this way the number of a state in a given level would always be smaller than the number of a state in a deeper level. This implied a breadth-first processing which made the functions more complex and added the need to keep a lot of information in the parameters.

This can be seen in one of the lowest level functions of the `bluefringe_apta` module: `expandTrace/2`. The only purpose of this function is to remove the first symbol from a trace and to add to the automaton structure the related information.

But it must remember the kind of the trace, (positive or negative, in case there are no more symbols in it), the last state number granted, the alphabet used (to add new symbols to it), the defined rejection states, and a separate buffer with the already expanded transitions from the current node, (in case our symbol already has a transition).

To do this we use the function `extractTrace/1` that joins the information about the next trace to expand in one tuple.

```
extractTrace({StateNum, [FailingTrace|Rest], []}) ->
    {{neg, fun addToFail/2, FailingTrace, StateNum},
     {StateNum, Rest, []}};
extractTrace({StateNum, FailingTraces,
             [AcceptanceTrace|Rest]}) ->
    {{pos, fun addToAccept/2, AcceptanceTrace, StateNum},
     {StateNum, FailingTraces, Rest}}.
```

Then we pass this tuple as the first parameter of the function `expandTrace/2`, together with the buffer of already expanded transitions from the current state and the record Agd, which contains all the information described above.

```
%% lastState, foundAlphabet, foundTransitions, rejectionStates
-record(agd, {lastSt = 0, alph = [], tr = [], rSt = []}).
```

And the function will update both the Agd and the buffer.

```
% @spec (TraceInfo, {Agd, Buffer})
% TraceInfo (returned by extractTrace)
expandTrace({pos, _, [], _}, {Agd, Buffer}) -> {Agd, Buffer};
expandTrace({neg, _, [], StateNum}, {Agd, Buffer}) ->
    {addFailingState(StateNum, Agd), Buffer};
expandTrace({_, Add, [H|_] = Trace, StateNum}, {Agd, Buffer}) ->
    case searchMatchInBuffer(H, Buffer) of
        no_match ->
            NewState = Agd#agd.lastSt + 1,
            NewAgd = addSymbol(H,
                        addTransition({StateNum, H, NewState},
                                Agd#agd{lastSt = NewState})),
            NewBufferEntry = {H, NewState, [], []},
            {NewAgd, [Add(Trace, NewBufferEntry)|Buffer]};
        {Match, RestOfBuffer} -> {Agd,
                                [Add(Trace, Match)|RestOfBuffer]}
    end.
```

2.2 State merging

Now we generalize the FSM by merging states. To merge two states we just move the transitions from one state to the other. For example, if we merge in the APTA from Fig. 2.1 the states 1 and 2, by calling the Erlang function `merge(APTA, 1, 2)` (see Appendix A), first we get the intermediate tree shown in Fig. 2.2 and after removing non-determinism we get the final result shown in Fig. 2.3.

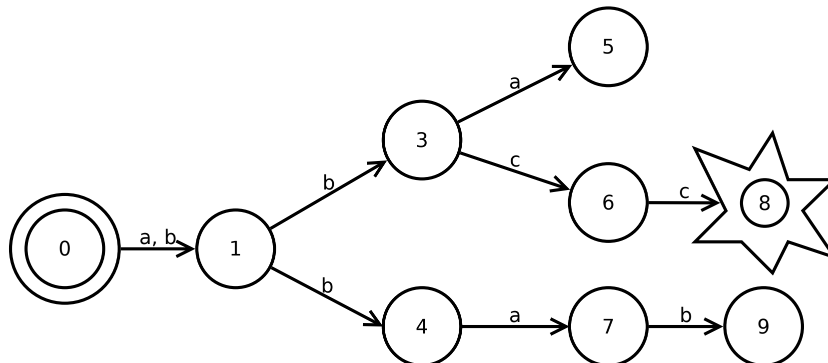


Figure 2.2: First state of merging nodes 1 and 2

We can see that, in this first step, non-determinism appears in node 1 with the symbol `b`. To solve that determinism we continue merging now 3 with 4 and finally 5 with 7. After that we would get:

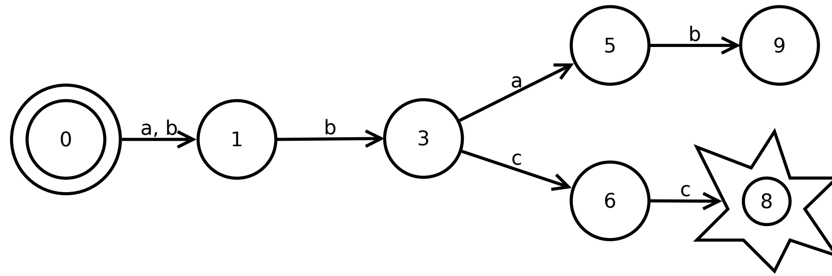


Figure 2.3: Result of merging nodes 1 and 2

After merging, we check that the original traces are still valid. If any trace is lost we undo the merge. In our implementation this is done by throwing an exception which interrupts the merging when this happens.

To decide which nodes should be merged first we use the strategy blue-fringe. This strategy consists in considering two zones of the FSM. The red zone has the nodes that cannot be reduced and the blue zone has the immediate neighbours, which will be used as candidates to merge with the red zone.

We start setting the initial state as red and its neighbours as blue.

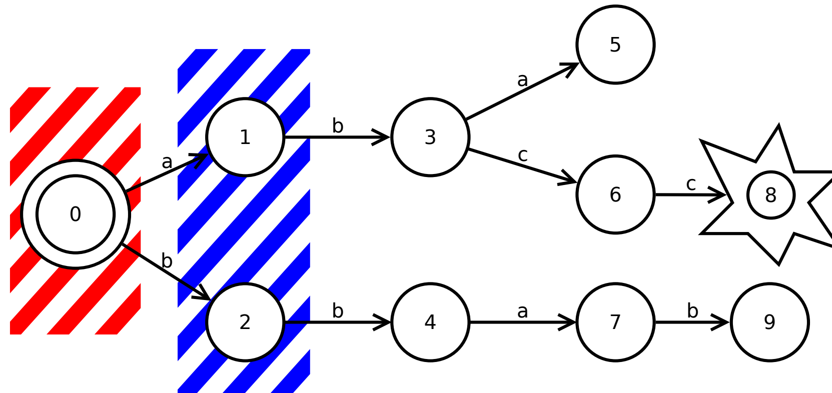


Figure 2.4: Apta before merging with red and blue zones

In each step we compute the score for every possible pair of candidates to merge, (pairs consisting on one node from the red zone and one node from the blue zone). The score for a pair of candidates is given by the number of extra merges that we would be forced to carry out in order to restore determinism after hypothetically merging the nodes of that pair.

Pair	(0, 1)	(0, 2)
Score	1	3

Two states are incompatible if one of them is a failing state, and the other is not. If a pair of candidates is incompatible or if it forces us to make an incompatible merge (in order to restore determinism) its score will be -1.

We must also check that all the positive traces are accepted and all the negative traces are rejected before actually committing any merge.

If a blue node cannot be merged with any of the red nodes, it becomes red and the blue zone is updated accordingly to match all the immediate neighbours of the new red zone.

The process ends when the whole FSM is red and we wrap up by merging all the failing states in one. This last merging should not produce indeterminism since there should not be transitions starting in any failing state.

2.3 Additional considerations

QSM was also initially designed to be interactive, here we only focus in the non-interactive version. The main difference is that the interactive version is intended to generate sample traces during the merging process and to ask the user if those are valid, in order to avoid over-generalization. Nevertheless, the same results can be achieved with the passive implementation. The user only has to add new tests to the input and to run the algorithm again.

Having the QSM algorithm implemented in Erlang gives us a better integration with the test code than the one we could get with other languages, (as java, in the case of StateChum,) since for example it allows us to use arbitrary Erlang terms as traces, which can be useful for some purposes as we explain in a later section.

After finishing the implementation, we have also found that it executes faster most of the time, since it does not need to start an extra virtual machine for this sole purpose.

3

Testing

In order to improve the reliability of such a new implementation we have used QuickCheck to test it against StateChum [4] because we wanted it to behave the same as this already well tested implementation.
We test that for a given set of positive and negative traces, our implementation and StateChum give similar state machines. In other words, we use StateChum as an oracle for our model based testing.

3.1 QuickCheck testing

To test our implementation of QSM we first used a property that should hold after the execution of the algorithm. The positive traces that where provided as input must be accepted by the output automaton. And the negative traces must be rejected exactly in its last symbol, this is, they must drive us precisely from the initial state to the failing one.

Implementing just this property in QuickCheck would be simple if we only generated completely random sets of traces. But, if we do it this way, most of the generated sets of traces that we would get, would lead to meaningless automatons that would have nothing to do with the ones we would find in a real scenario. To solve this issue we decided to generate random automatons instead, and then walk through them randomly. This implementation could result in unreachable states, but this is not a problem since the input to the algorithms is just the traces and they will not contain such states.

```
automata() ->
  ?LET({States,Events}, {set(elements([a,b,c,d,e,f,g])),
                         non_empty(set(elements([x,w,y,z])))},
       ?LET(Trs, transitions(Events,States),
       {[init,bad] ++ States,init,bad,Events,Trs})).
```

An automaton in the testing module is represented by a tuple in the form: {list of states, initial state (`init`), failing state (`bad`), list of events, list of transitions}. And each transition is as usual another tuple: {origin, event, destination}.

```
transitions(Events, States) ->
    ?LET(Trs, [{init,elements(Events),oneof([bad | States])} |
               normal_transitions([init|States], Events,
                                  [bad,init|States])],
         determinize(Trs)).
```

In the `transitions` function we make sure that we get at least one transition so that later we can generate at least one trace.

In order to make valid and useful automatons we just have to treat the initial state (`init`) and the failing state (`bad`) independently and to make sure that we do not generate transitions that start in the failing state.

In the first place we though that forcing the resulting automaton to be deterministic would not be required because the resulting traces could always be translated into a deterministic APTA tree. But the experience of testing the algorithm showed us an exception to this idea: the non-determinism can lead us to have one symbol that would drive us to both a normal and a failing state, (e. g. we could get the trace `[x, y]` as both positive and negative).

We decided not to use this kind of input as negative tests because this validation would be almost equal to the validation the algorithm itself does, so it would be just like repeating code.

After generating a random automaton, we just do a series of random walks through the automaton (starting at the initial state), give them as input to the algorithm, and check that the output automaton complies with the input traces.

This helped us to find some misunderstandings in the implementation. For example: we thought in the first place that just by taking care of not merging a normal state and a failing one, the collapsed automaton would properly accept or reject all the input traces, but this was proved to be false when we run this test. We fixed it by checking the traces with each merge. But even though this test is useful, it would still pass if we had only implemented the APTA generation, and consequently we are not actually testing that the merging system and the blue-fringe implementation work properly. In the next section we solve this by testing our implementation against an existing one.

3.2 Interfacing StateChum

Specifically, we wanted to know if our implementation generated minimal automatons. We could not find an alternative way to check if the resulting automaton was in fact minimal, since that is the actual purpose of the QSM algorithm. But we did know StateChum, an already tested implementation, that does give a minimal automaton. So we checked instead that our implementation gave similar results as it. In order to do

this, we first wrote an interface to StateChum that would allow us to provide lists of atoms as input, and then parse the resulting automaton to an Erlang entity. This could be done thanks to the text mode of StateChum.

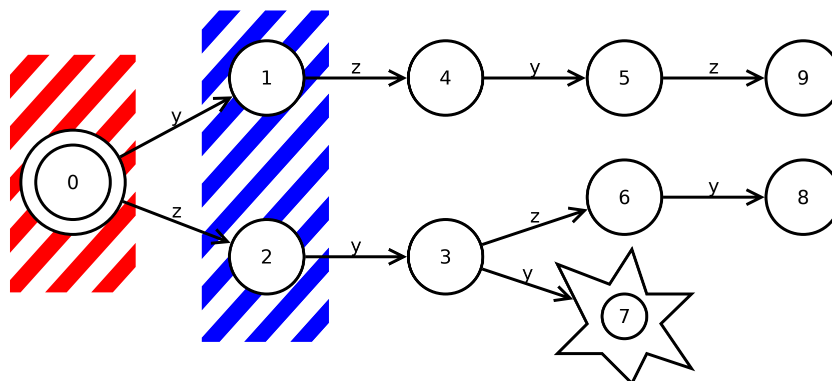
StateChum's text mode provides a parsable description of the transitions of the output automaton. Nevertheless, this output does not specify the initial state or transitions that end in a failing state, or the failing state at all. Because of this, we could not check that the automatons were equivalent, so we just compared the number of non-failing states.

This simple check allowed us to realize about some misunderstanding about the semantics of the algorithm. For example: blue states need to be transformed to red immediately after they become impossible to merge with the red ones, instead of trying to merge all possible blue states first.

Another example is that in the original pseudo-code there are instructions of the kind `for all X` where the list `X` grows while execution is inside the loop. After testing we discovered that this changes should be taken into account by performing extra iterations in the end.

3.3 Differences with StateChum

We needed to fix a few errors in our QSM implementation, but even after that, we found out that in some cases the results were still different despite being both correct according to the QSM specification. For example, from the traces: `{[[y,z,y,z], [z,y,z,y]], [[z,y,y]]}`. We would get the following APTA tree:

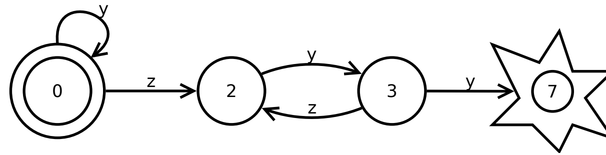


Then we apply the bluefringe strategy and compute the scores for all possible combinations:

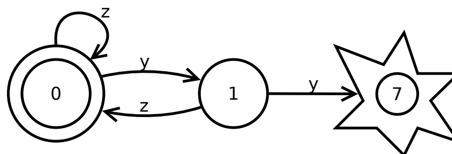
Pair	(0, 1)	(0, 2)
Score	3	3

One obvious question arises: If several pairs of nodes have the same score, which pair shall we merge first?

If we choose the pair (0, 1), (as StateChum apparently does in this case,) we will form a loop with the `y` symbol in the state 0 and continuing with the algorithm we will end up with an irreducible automaton like this, (the state numeration may differ):



On the other hand, if we chose the pair (0, 2), (as our implementation does in this case,) we will form a loop with the `z` symbol in the state 0 and we will get this slightly smaller, also irreducible automaton:



Given this choice in implementation, one question now arises, is any of the choices better than the other? In order to find out, we generated a large number of inputs and tested the two algorithms on these inputs and compared the size of the resulting automatons. This is done in the QuickCheck property `prop_statistic/0`. It uses the function `collect()` to see how many times our implementation produces a smaller automaton than StateChum, how many times they match, and how many times occurs the opposite. The result for 1000 iterations was:

```
OK, passed 1000 tests
89% draw
5% sc
5% qsm
true
```

Being `sc` that the output from StateChum was smaller, `qsm` that the output from our tool was smaller and `draw` that both outputs had the same size.

From this we can conclude that, despite the decision does affect the size of the resulting automaton, our decision of choosing a random one, (not actually random but whichever was more efficient to extract in that situation) outputs approximately the same number of automatons bigger and smaller than StateChum. And approximately nine out of ten times, they have the same size with both implementations. So, after this results, we accepted this implementation choice as valid.

4

EUnit parsing

EUnit [7] is the Erlang unit test framework in the style of JUnit[8], HUnit[9], etc. In this framework one can specify unit tests and check that their results are correct by using some preprocessor macros provided by the EUnit library. By using this tool we can later run all the unit tests at once and get a summary of those results. EUnit modules contain a series of functions that can in principle contain any Erlang expression, this expressions are evaluated when the tests are run. Because of this, the tests can have arbitrarily complex structures, which makes it difficult, (and in some cases impossible,) to analyse them statically. That occurs because this functions usually make calls to the code, so we could in principle get the traces by collecting those calls while running the tests. But this is not always possible since we may not have the implementation, and thus, the tests may not be possible to execute.

Depending on the use case for our tool, we would like to have both static analysis as well as dynamic analysis and combinations thereof. Nevertheless, we may need static analysis if we use the tool in a test driven development approach, where the tests are developed before the code is written. In that case, we cannot run the tests to obtain the traces. Even if some tests can be run, new tests may be added that require static analysis.

4.1 Possible scenarios

From the syntactic point of view, the tests in EUnit are given by normal Erlang functions with a name that end with the sufix `test` or `test_` and they must contain some of the EUnit macros like `?assertMatch()` or `?assertException()` which are defined in the header file `eunit/include/eunit.hrl` from the EUnit distribution.

Execution flow may vary in terms of the returned values from other functions. This other functions may be in the target code (the code we want to test) itself and, thus, they may not even be defined yet.

On the other hand, in some cases we may find that all the external functions are surrounded by macros like `?assertMatch()` or `?assertException()`, and in those cases we would know the expected return value.

This gives us three possible solutions:

1. Dynamically running the whole tests and collecting the traces. (This is similar to the reverse engineer approach StateChum was designed for)

2. Dynamically running the part of the tests that is in the same module and inferring the returning values from the EUnit macros when possible.

3. Statically parsing the EUnit code and trying to infer the execution flow when possible.

None of the three would work for all the desirable cases. Here, we chose to implement the last one, which achieves the maximum independence from the tested code.

4.2 Syntax analysis

EUnit can be considered a domain specfic language for testing. By a large set of macros, the test notion is expanded in Erlang code for running the tests and comapring their results. We do not need to expand the macros in the same way. In fact, this expansion makes the analysis harder, since we use the semantics of the macros (the domain specific language) to be ale to determine the possible traces.

We want to parse the EUnit file without the expansion of the macros. Erlang offers a parser, but that requires pre-processing, which expands the macros.

Therefore, we replace the EUnit macro expansion by our own macro expansion, and then analyze the resulting code. In fact, we use EUNIT_HRL macro. In the original EUnit macro definition, this EUNIT_HRL macro is used to disable the other EUnit macros when defined. So, if defined, we can use our own macro expansions, if not defined, we use EUnit's macro expansion.

Our macro definitions are very simple, and consist basically in a tuple with an atom and the code inside the macro.

```
-define('_assertMatch'(P1, Trace), Trace).
-define(assertMatch(P1, Trace), Trace).

-define('_assertError'(P1, Trace),
{fsm_eunit_parser_negative, Trace}).

-define(assertError(P1, Trace),
{fsm_eunit_parser_negative, Trace}).

-define('_assertExit'(P1, Trace),
```

```
{fsm_eunit_parser_negative, Trace}).

-define(assertExit(P1, Trace),
{fsm_eunit_parser_negative, Trace}).

-define('_assertException'(P1, P2, Trace),
{fsm_eunit_parser_negative, Trace}).

-define(assertException(P1, P2, Trace),
{fsm_eunit_parser_negative, Trace}).

-define('_assertThrow'(P1, Trace),
{fsm_eunit_parser_negative, Trace}).

-define(assertThrow(P1, Trace),
{fsm_eunit_parser_negative, Trace}).

-define(EUNIT_HRL, true).
```

After removing the EUnit macros we look at the syntax tree and parse the result. We analyse the syntax tree recursively using pattern matching and carrying two lists (one for positive traces and one for negative traces). The tuples that represent EUnit macros are recognized by the pattern matching and the name of the function inside them is added to one list or the other depending on the tuple.

Not all the EUnit tests are supposed to be executed as is but they can also contain "generators", mainly, this is, code that when executed, it will result in the actual functions that do the tests. For example:

```
startstop_test_() ->
    fun () ->
      ?assertMatch(Pid when is_pid(Pid),start([])),
      ?assertMatch(ok,stop()),
      ?assertMatch(Pid when is_pid(Pid),start([1])),
      ?assertMatch(ok,stop()) end.
```

We can recognize this tests because their names end with and underscore (more specifically they end with `test_`, in comparison to the normal tests that end with `test`)

For simplicity we assume that the functions only contain a list of calls surrounded by EUnit macros or either calls to other functions like them. In this static analysis we do not consider variable assignments either.

Nevertheless, we do consider the special tuples `foreach` and `setup` as well as the equivalent `foreach_` and `setup_`, (that take generators as content,) as long as their contents follow the principles explained before. In the case of `foreach`, each of the

elements of the list will be considered as a different test and, thus, it will produce a different trace.

If the EUnit module complies with this format, our tool will extract a tuple with the two lists of traces (positive and negative) in the format `{module, function, [arg1, arg2, ...]}`, this can be easily mapped if we do not need all that information.

Thus, for the example above, after compiling and analysis, we obtain the sequence:

```
{[[{frequency,start,[[]]},
   {frequency,stop,[]},
   {frequency,start,[[1]]},
   {frequency,stop,[]}]],
 []}
```

It is added to the first list of the tuple since this trace is positive. The module name in this case is extracted from a `import` macro in the header of the file:

```
-import(frequency,[start/1, stop/0, allocate/0, deallocate/1,init/0]).
```

5

Related Work

This whole paper is based on a previous investigation by Thomas Arts and Simon Thompson which is documented in their paper [2], where an important part of the processes used here are described and future possible uses for them are mentioned.

We have already mentioned Statechum[4], this application by Neil Walkinshaw and Kirill Bogdanov, implements the whole QSM algorithm including the interactive version. They have also documented their investigation in the papers [6] and [5]

Also, Hans Svensson, in his paper [10], tested an implementation of büchi-automata by comparing it to existing implementations in a similar way, i.e., generating random input with QuickCheck and comparing the output of both implementations.

6

Conclusion and future research

In this paper we have presented: one new implementation of the QSM algorithm in Erlang, one technique to test such an implementation using an oracle as a black box, and one approach to the extraction of traces from EUnit tests by means of static analysis.

This contributions, together, provide a unified way to visualize EUnit test cases and to generate templates that can in the future be used to generate QuickCheck state machines as proposed by Thomas Arts and Simon Thompson [2], and that will eventually allow migration of EUnit tests to QuickCheck properties in an automatic way.

The QSM algorithm is sufficiently fast for the small sets of test cases in an EUnit test suite. However, for very large test suites, one may see performance problems. Therefore, it could be interesting to create a concurrent implementation of the QSM algorithm. It should be much easier to parallalize our Erlang implementation than it would be to parallelize the corresponding Java implementation, due to the superior support of concurrency in Erlang.

Bibliography

[1] K. Beck, Test-driven development: by example, Addison-Wesley Professional, 2003.

[2] 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.

[3] P. Dupont, B. Lambeau, C. Damas, A. van Lamsweerde, The QSM algorithm and its application to software behavior model induction, Applied Artificial Intelligence 22 (1) (2008) 77–115.

[4] Statechum.
URL `http://statechum.sourceforge.net/`

[5] N. Walkinshaw, K. Bogdanov, M. Holcombe, S. Salahuddin, Reverse engineering state machines by interactive grammar inference, in: wcre, IEEE Computer Society, 2007, pp. 209–218.

[6] N. Walkinshaw, K. Bogdanov, Inferring finite-state models with temporal constraints.

[7] Eunit user's guide.
URL `http://www.erlang.org/doc/apps/eunit/chapter.html`

[8] Junit.
URL `http://junit.sourceforge.net/`

[9] Hunit.
URL `http://hunit.sourceforge.net/`

[10] H. Svensson, Implementing an ltl-to-büchi translator in erlang: a protest experience report, in: Proceedings of the 8th ACM SIGPLAN workshop on ERLANG, ACM, 2009, pp. 63–70.

A

Source code for merge module

```
%%%-------------------------------------------------------------------
%%% File    : merge.erl
%%% Author  : Pablo Lamela Seijas <lamela@student.chalmers.se>
%%% Description : Implements functions to merge states.
%%%
%%% Created : 7 Nov 2010
%%%-------------------------------------------------------------------
-module(bluefringe_merge).
-include("../include/automata.hrl").

%% API
-export([merge/3, number_of_merges/3]).

%%====================================================================
%% API
%%====================================================================
%%--------------------------------------------------------------------
%% Function: merge(Automata, OptimizedExtraInfo, State1, State2)
%% Description: Merges states in both automata and optimized extra
%%   info. Doesn't remove the eliminated states from the lists.
%%--------------------------------------------------------------------

merge(Automata, St1, St2) ->
    {Result, _} = merge_list(Automata, [{St1, St2}], 1),
    Result.
```

```
number_of_merges(Automata, St1, St2) ->
    case (catch merge_list(Automata, [{St1, St2}], 1)) of
        {_, Number} -> Number;
        incompatible -> -1
    end.

%%======================================================================
%% Internal functions
%%======================================================================

% Merges the list of pairs and the new pairs needed to make it
% deterministic. Updates OptimizedExtraInfo.
merge_list(Automata, [{St1, St2}|Tail], Number) ->
    {NewAutomata, NewMerges} = merge_one(Automata, St1, St2),
    merge_list(NewAutomata, NewMerges++Tail, Number + 1);
merge_list(Automata, [], Number) -> {Automata, Number}.

del_from_list(List, Element) ->
    lists:filter(fun (X) -> X =/= Element end, List).

is_in_list(List, Element) ->
    lists:any(fun (X) -> X =:= Element end, List).

% Merges one pair updating EI. Returns the list of new pairs needed
% to make it deterministic (return is not a valid, will be valid after
% the list of new pairs to merge is merged...)
merge_one(Automata, St1, St2) ->
    case {is_in_list(Automata#fa.st, St1),
          is_in_list(Automata#fa.st, St2)} of
        {true, true} ->
            checkIfIncompatible(Automata#fa.fSt, St1, St2),
            {NewTr, NewMerges} = merge_on_trList(Automata#fa.tr,
                                                 St1, St2),
            {Automata#fa{tr = NewTr,
                         st = del_from_list(Automata#fa.st, St2),
                         fSt = del_from_list(Automata#fa.fSt, St2)},
             NewMerges};
        _ -> {Automata, []}
    end.

checkIfIncompatible(List, St1, St2) ->
    case lists:subtract([St1, St2], List) of
        [_] -> throw(incompatible);
```

```
        _ -> ok
    end.


% Merges one pair not updating EI. Returns the list of new pairs
% needed to make it deterministic
merge_on_trList(Tr, St1, St2) ->
    merge_on_trList([], Tr, St1, St2, [], []).
merge_on_trList(DoneL, [], _St1, _St2, Final, NewMerges) ->
    {Final++DoneL, NewMerges};
merge_on_trList(DoneL, [{Ori, Tra, Dest}|Tail],
                        St1, St2, Final, NewMerges)
    when ((Ori == St1) or (Ori == St2) or
         (Dest == St1) or (Dest == St2)) ->
    {NewFinal, NewNewMerge} =
        merge_one_tr(St1, St2, {Ori, Tra, Dest}, Final),
    merge_on_trList(DoneL, Tail, St1, St2, NewFinal,
                    lists:merge(NewNewMerge, NewMerges));
merge_on_trList(DoneL, [{_, _, _} = Head|Tail],
                St1, St2, Final, NewMerges) ->
        merge_on_trList([Head|DoneL], Tail,
                        St1, St2, Final, NewMerges).

replace(Or, De, A) ->
    case A of
        Or -> De;
        Other -> Other
    end.

replace_3t(Or, De, {A, B, C}) ->
    A2 = replace(Or, De, A),
    C2 = replace(Or, De, C),
    {A2, B, C2}.

merge_one_tr(St1, St2, {_, _, _} = Tupla, Final) ->
    merge_one_tr(replace_3t(St2, St1, Tupla), Final, []).
merge_one_tr({_, _, _} = Tupla, [], DoneF) ->
    {[Tupla | DoneF], []};
merge_one_tr({Ori, Tra, Dest} = Tupla,
    [{Ori, Tra, Dest}|Tail], DoneF) -> {[Tupla|Tail]++DoneF, []};
merge_one_tr({Ori, Tra, Dest1} = T1, [{Ori, Tra, Dest2}|Tail], DoneF)
    when Dest1 < Dest2 -> {[T1|(Tail++DoneF)], [{Dest1,Dest2}]};
merge_one_tr({Ori, Tra, Dest1}, [{Ori, Tra, Dest2} = T2|Tail], DoneF)
```

```
    when Dest2 < Dest1 -> {[T2|(Tail++DoneF)], [{Dest2,Dest1}]};
merge_one_tr({_, _, _} = Tupla1, [Tupla2|Tail], DoneF) ->
    merge_one_tr(Tupla1, Tail, [Tupla2|DoneF]).
```