

In the need of a design... reverse engineering Erlang software

Thomas Arts¹ and Cecilia Holmqvist²

¹ IT university in Göteborg
Box 8718, 40275 Gteborg, Sweden,
email `thomas.arts@ituniv.se`

² Ericsson AB
Lindholmospiren 11, 41756 Gteborg, Sweden

1 Introduction

Software development often faces the problem that over time the design documents and the actually implemented code coincide less and less. After fine-tuning the software, adding features, leaving out other features and correcting design errors in the code, not in the documents, the result is a product that can be sold. However, the design documentation is no longer up-to-date.

Time to market is important and the costs that it takes to keep design documentation updated is sometimes thrown into the shade of being out first. Only if one survives, the up-to-date design documents are of any use. Thus, after obtaining a market share, one is forced to update the design documents in order to effectively propose additional features or major software changes.

In this paper we discuss a technique to help reverse engineering a part of an Ericsson product. In the economic crisis in which the product was finalized for the market, the resources for updating the design documents were not available. After being successful in the market, the product has stabilized. Now, there is a strong wish for updated design documents. First of all, for understanding the system better (what is actually going on?). Second, for being able to plan a new implementation from scratch for large parts of the system. Third, the difference between actual behaviour and designed behaviour indicates problems in software parts: the larger the difference between design and actual code, the larger the possibility that errors can be found in that part.

The product we looked at was written in Erlang [1] with rather strict design guidelines. For example, the names used for functions and modules were well reflecting the original design; in the software block *Mobilty Management in UMTS (MMU)* all module names started with `mmu`. These strict guidelines helped us in easily reconstructing a design from traces of the code.

We performed a case study to show the possibility of supporting the reverse engineering attempt with some software tools. These tools strongly depended on the fact that Erlang is the implementation language of the system and that the design guidelines are followed. However, projects with other, but also strict guidelines will be able to use the same tools in order to reverse engineer the code.

Other reverse engineering attempts have been presented by Nyström [6] and by Mohagheghi *et. al.* [7]. Both tools are based on a static analysis of the code, whereas in our approach we assume not to have access to the source code, but only use information available at runtime.

The paper is organized as follows: In Sect. 2 we describe the rough outline of the software we looked at. In Sect. 3 we discuss how we obtained a finite state machine from looking at traces of the running software. The state machine was graphically visualized and manually compared with the original UML design. In Sect. 4 we discuss what differences we were quickly able to find with this technique.

2 Mobility Management in UMTS

In our case-study we concentrate on the Mobility Management in UMTS (MMU), which is a software component, a so called *block*, in one of the systems in a UMTS network.

The component is specified by a set of UML diagrams. A subset of these diagrams consist of state machines that specify the states in which the component can be. This subset resembles much that of a hierarchical state machine, a UML version of statecharts [5]. On the first or top level a state machine with eight states, with names like *ms_attaching*, *ms_connected*, *etc*, is specified (see Fig. 1). On the next level, each of these states is represented as a state machine with several sub-states within this larger state.

Strangely enough, the implementation of the hierarchical state machines is not based upon the *generic finite state machine* behaviour, nor on a specially developed hierarchical state machine behaviour. Without digging into the reason for it, we only state here that the events are implemented as function calls (e.g., `detach_request/6` implements a Detach Request event) and state is implicit in the software. For that reason we cannot use the earlier developed trace visualisation software [2]. Since the application needs to have access to the state now and then, the programmers have added a function call `set_state/2` to store the information of the present state. Whether this is stored in a process dictionary or in a server process, is not important for the rest of the story.

Needless to say that this way of keeping track of the present state is rather error prone. The generic finite state machine behaviour, where state is guaranteed to be updated after every event, is more robust against a programmer that forgets to update a state. In particular, a behaviour can be used to already statically detect such omissions, whereas in the present implementation, one has to find these omissions by running tests.

The component consists of several Erlang modules. The main module is called `mmumoc`, which serves as an interface for all other modules in the component. All communication from other components or subsystems to this component goes via the `mmumoc` module, which means that at least one interface function in this module is called for every external signal. The functions in the `mmumoc` module depend on functions in the other modules (of which the names all start with

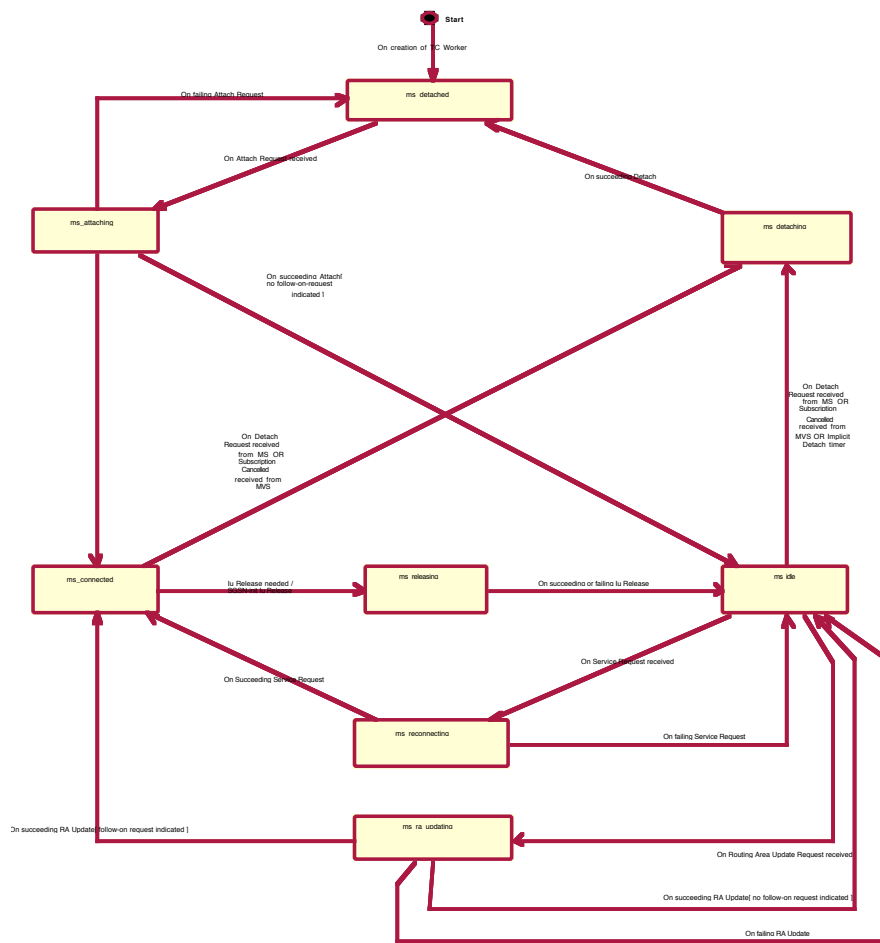


Fig. 1. Top level state machine

mmu). An incoming Detach Request can for example cause over twenty different functions in five 'mmu' modules to be called, many of which are called several times, before the state is updated.

3 Runtime software analysis

The MMU component consists of too many lines of code to simply understand what the software is doing by looking at the source code. Even with the design at hand it is far too costly, if not impossible, to statically analyze the code, i.e., determining the behaviour of the code by carefully studying the source code. Therefore, traditionally, a manual runtime analysis is performed.

Erlang has a trace function that allows a person to select whatever function he/she is interested in. Calls to these functions are then monitored in a running system. Runtime analysis uses this trace function. If one wants to analyse the MMU component, one monitors all calls to functions in this component, i.e., where the module name starts with `mmu`. A set of stimuli is provided to the system and the resulting list, a so called *trace*, of function calls (including time stamp, arguments, return value, etc) is stored in a file created by the `disk_log` module.

Before we started our work, these trace files were converted to a textual format and manually analyzed. Traces can differ in length, but we looked at files of about 9MB, containing about 15 thousand entries. Loaded into emacs, it is rather easy to find each occurrence of `mmumoc:set_state/2`. However, we immediately wanted to have a tool for quickly extracting these calls from the trace. Note that one could also choose to only monitor calls to this particular function with the trace functions. However, we are not only interested in the possible state transitions. If we find a state transition that differs from the one in the specification, we also want to understand what has happened. The other functions in the trace are necessary to get that understanding. Since we might miss the particular behaviour when we run the software again, it is important to have enough information in the trace to determine what happened in case of an unexpected state transition.

Our traces can be rather large, in principle even several gigabytes. For performance reasons, we do not want to read this binary trace in memory to convert it in a list and then remove most of the unnecessary elements (only very few function calls are calls to the `set_state` function). By using the `disk_log:chunk` function, we would be so much restricted to 64 bytes, that we decided to implement our own log reader. In an eager language we program a lazy file reading, which we make flexible to the kind of filtering by having a filter function as argument.

```
read(FileName,Filter) ->
  {ok,FileDescr} = file:open(FileName,[read, raw, binary]),
  Terms = unpack(FileDescr,Filter,[]),
  file:close(FileDescr),
  Terms.
```

```

unpack(FileDescr,Filter,Terms) ->
  case file:read(FileDescr,5) of
    {ok,<<B1,B2,B3,Size:16>>} ->
      {ok,BTerm} = file:read(FileDescr,Size),
      Term = binary_to_term(BTerm),
      case Filter(Term) of
        true ->
          unpack(FileDescr,Filter,[Term|Terms]);
        false ->
          unpack(FileDescr,Filter,Terms)
      end;
    eof ->
      lists:reverse(Terms)
  end.

```

The result of the `unpack` function is a list of terms. To every term in the log file, we have applied a filter function, returning either *true* or *false*. Only those terms for which *true* was returned, are left in the list.

The filter function is typically something like, *either the term is a mmumoc: set_state/2, or a an event originating external to the MMU component*. Thus, it is a logical combination of well determined terms. This can be implemented very flexible by defining small filters for the well determined terms and combinators to combine them. For example, the filters to determine state and events from outside, can be defined as:

```

state_mmu() ->
  fun({trace_ts,Pid,call,{mmumoc,set_state,[S,SS]},Caller,TS}) ->
    true;
  (_) ->
    false
  end.

outside_mmu() ->
  fun({trace_ts,Pid,call,{mmumoc,F,A},{CM,CF,CA},TS}) ->
    string:substr(atom_to_list(CM),1,4)=[$_,$m,$u,$_];
  (_) ->
    false
  end.

```

A beautiful way of writing a filter would be `or(state_mmu(),outside_mmu())`, or a more complex case to obtain all calls that are not inside the MMU component `and(call(),not(inside_mmu()))`. Since the logical operators cannot be overloaded in Erlang, we use a different name for them and define them as follows (cf. [3]):

```

filter_or(F1,F2) ->

```

```

fun(T) ->
  case F1(T) of
    true ->
      true;
    false ->
      F2(T)
  end
end.

filter_and(F1,F2) ->
  fun(T) ->
    case F1(T) of
      true ->
        F2(T);
      false ->
        false
    end
  end.

filter_not(F) ->
  fun(T) ->
    not F(T)
  end.

```

This gives us an easy ‘language’ for quickly defining what terms one wants to keep in the trace.

Thus, for standard trace from the Mobility Management component we constructed with hardly any effort a small program to select the items of the trace we are interested in. After filtering the states and events from the trace, we introduced a second pass over the (much shorter) list to label the terms either as state or as event. A pure syntactic criteria is used to determine which is which. A textual or graphical representation of this reduced trace is already a great help in the analysis, but we can do more.

Recall that the specification was a hierarchical state machine. Hence we obtain too many states when we consider states with different substates as different states in our trace. Instead of the above second pass to label certain terms as state and certain terms as event, we use that pass to not only label the states and events, but also rename them with a given function. This function maps states with different substates, represented in the trace by a list with two elements, to one and the same atom. It can also be used to keep the substate information for one particular state, and rename all other states and substates to **external**. In that way, one selects all activity inside a state machine on the second level, i.e., the state machine inside a state.

```

rename_state(Rename) ->
  fun({trace_ts,Pid,call,{mmumoc,set_state,[S,SS]},Caller,TS}) ->
    Rename({state,[S,SS]});

```

```

    (Term) ->
      Term
    end.

firstlevel() ->
  fun({state,[S,SS]}) -> {state,S} end.

secondlevel(States) ->
  fun({state,[S,SS]}) ->
    case (States==all) or lists:member(S,States) of
      true ->
        {state,[S,SS]};
      false ->
        {state,?StartState}
    end
  end.
end.

```

It turned out that we had to apply this software to several traces, selecting the function `set_state` and events from outside the MMU. We first ran the software for a while with different stimuli, resulting in several traces stored in files. The files (i.e., traces) are named after the use-cases we used, e.g., *connectUMTS* for connecting a mobile phone to a UMTS network. Every of these traces was then turned into a set of new traces by applying the `secondlevel` function above with exactly one of the toplevel states in the set `States`. In other words, for every substate machine in the hierarchy we created a different trace in which all substate transitions are visible, but all other states are mapped to one and the same external state. In such a way, we created traces like *connectUMTS-ms_attaching*, *connectUMTS-ms_reconnecting*, etc.

These traces contain all state transitions in one of the substate machines given in the specification with respect to one specific use-case. Since the traces cover a set of events, some transitions are clearly covered more than once. By using several use-cases we also ensured that we got several possible scenarios in one substate machine. Now we can compare those transitions with the UML design. In order to do that quickly, we use a graphical visualisation tool that generates a picture of the state machine, which is extremely easy to compare with the UML picture.

3.1 Structuring the data

Given a trace with states and events clearly labeled, we can easily construct a list of vertices and a list of edges of a graph that represents a state machine. First, all states and events are numbered in such a way that different states or events get different numbers, but states or events that occur several times in the trace, all get the number of their first occurrence. Second, by traversing the numbered trace, which is a list of pairs with an integer and a state or event, edges are formed by the pair of the two integers of each two consecutive entries.

Note that in this way, events are made into states in the visualisation. Eventually we will visualize states and events with different colours, to distinguish them. Thus, in the actual visualisation we could even pretend the event to be a label on an arrow by choosing the right form of arrows drawn. The reason to take them as states in the graph data structure is to be able to detect repetition of events before a state transition. Certain events may occur an arbitrary number of times before a state transition is noticed. This normally means that the state transition is not caused by the event. In order to undo the visualization from those possible incorrect causalities, we write events as states.

For example, the trace *connectUMTS* in which only the `ms_connected` state is preserved contains 1142 entries, looking like:

```
[{state,outside},...,{state,outside},
 {state,[ms_connected,sub_idle]},
 {event,get_rai},
 {state,outside},...,{state,outside},
 {state,[ms_connected,sub_idle]},
 {event,get_rai},
 {state,outside},...,{state,outside},
 {state,[ms_connected,sub_idle]},
 {state,[ms_connected,sub_idle]},
 {state,outside},...,{state,outside},
 {event,get_rai},
 ...]
```

The state `outside` occurs many times, since all states not in the substate machine of `ms_connected` fall under that category. There are only two other states present in this trace, viz. `[ms_connected,sub_idle]` and an exceptional occurrence of `[ms_connected,sub_att_failed_no_auth_iu_releasing]`. There is one event, viz. `get_rai`. The vertices in the graph that is constructed are the following:

```
[{1,{event,get_rai}},
 {2,{state,[start,outside]}}},
 {3,{state,[ms_connected,sub_idle]}}},
 {4,{state,[ms_connected,
 sub_att_failed_no_auth_iu_releasing]}}]
```

There are also only a few edges in this graph

```
[{2,2}, {2,3}, {3,1}, {1,2}, {3,2},
 {2,1}, {3,4}, {4,2}, {3,3}, {1,1}]
```

showing that one really gains in representation by the mapping of a trace to a graph. A rather long trace, which takes some effort to understand is in two steps quickly reduced to a small graph that is easy to understand. In such a small graph as above, one need not even visualize the edges to see that several strange

things occur, like setting the state to `[ms_connected,sub_idle]` twice, without any observed event occurring inbetween.

The attentive reader might wonder why state transitions seem possible without an event inbetween. These events do occur, but are either not traced or filtered away in the earlier trace process. We can regard them as unnamed events.

The graph formatting is performed by a special open source program called 'dot' [4]. The vertices and edges are exported in textual format and dot is generating a nice looking picture as shown in Fig. 2 (many formats, including postscript and scaled vector graphics are supported).

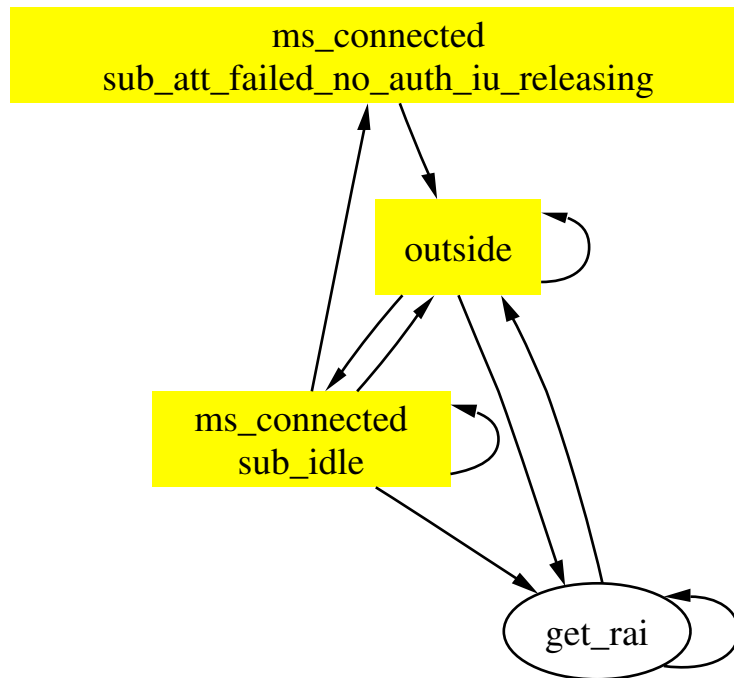


Fig. 2. Visualisation of obtained `ms_connected` substate machine

With some effort one could certainly also find out in which format state machines are presented in the UML tools that Ericsson is using. In that way, one could easily generate pictures of the state machines that have identical layout to the designed state machines. However, our shown visualisation is already so close to the design, that it is really easy to compare designed and obtained state machine.

4 Comparing Design and Reality

With the visualization of several traces at hand, it is easy to compare the UML design and the state machines obtained from a trace. It turns out that during the development, the code has really diverted a lot from the actual design. Most generated substate machine contains some states and some transitions that are not present in the design.

In a few hours, many issues are written down. Sometimes the differences between real behaviour and design is so different, that the number of states in common is less than the number in which they differ. The whole design started from use-cases and some of these use-cases have found their way in the code without the state machines in the design being updated.

e even find that some state machines are not modeled at all, i.e., there is no design available.

All together, we can conclude that we quickly (a few hours) detect and administer a number of major differences. After that, other traces visualize the same machines and we only find some minor differences.

If we would have had access to the encoding of the UML pictures, we would have been able to write a small tool to compare state machines, therewith being even faster in our comparison (visually highlighting the difference). However, even with the present way of comparing design and actual code, we are many times faster than by using a manual comparison of design and a text version of the trace information.

References

- [1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition)*. Prentice-Hall International (UK) Ltd., 1996.
- [2] T. Arts and L-Å. Fredlund. Trace Analysis of Erlang Programs. In Proc. of *ACM Sigplan Erlang Workshop*, Pittsburgh, USA, 2002.
- [3] T. Arts, K. Claessen, H. Svensson. Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang. In Proc. of *FATES 2004*, Linz, Austria, 2004.
- [4] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.
- [5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231-274, 1987.
- [6] J. Nyström and Bengt Jonsson. A Tool for Extracting the Process Structure of Erlang Applications. Erlang User Conference, Stockholm 2001. Also at the Erlang Workshop, Florens, Italy, 2001.
- [7] P. Mohagheghi, J.P. Nyttun, Selo, and W. Najib. MDA and Integration of Legacy Systems: An Industrial Case Study. In Proc. of *Workshop on Model Driven Architecture: Foundations and Applications, MDFA'03*. University of Twente, Enschede, The Netherlands. 26-27 June 2003.