Trace Analysis of Erlang Programs

Thomas Arts*
IT-university
Box 8718, 402 75 Göteborg, Sweden

Lars-Åke Fredlund
Swedish Institute of Computer Science,
Box 1263, SE-164 29 Kista, Sweden
fred@sics.se

ABSTRACT

The paper reports on an experiment to provide the Erlang programming language with a tool package for convenient trace generation, collection and to support analysis of traces using a set of techniques. Due to the frequent use of state-based software design patterns in Erlang programming we can in many cases recover not only the events from a trace log, but also the program states causing these events. This makes it possible to obtain program models from execution traces. In our work we make use of these program models for program visualization and model checking.

1. INTRODUCTION

The Erlang programming language [2] is at its core a functional programming language extended with features such as processes and asynchronous message passing to cater to the programming of distributed applications. Inside Ericsson Erlang has been used for writing software for telecommunications equipment such as the AXD 301 ATM switch [3] (consisting of over half a million lines of Erlang code [10]).

In practise many aspects of a product such as AXD 301 are, due to the highly complex and distributed nature of its software and hardware architecture, hard to understand for most of the designers working on it. Source code analysis and verification techniques are difficult to apply to the product, partly simply because of the sheer size and complexity of its design. Further, in real life it operates in environments that cannot easily be replicated or simulated during in-lab testing or verification. As such the collection and analysis of runtime information is of critical importance during development, operations and maintenance.

The Erlang language already contains several features that ease runtime data collection: (i) there is an extensive tracing framework (the `erlang:trace/3` and `erlang:trace_pattern/3` functions) that enables tracing of activities such as process communication, function calls, and so on; (ii) typical Erlang software is of a regular nature: it makes use of a number of so called design patterns that provide a standardized means to program recurring patterns in telecommunications software, e.g., finite state machines, client-server communications and event handling.

In our work we focus on tracing the distributed aspects of Erlang software, building upon the trace framework already present in the language. Erlang programmers already are aware of the trace functions `sys:trace` used to obtain traces of the generic behaviours. We implemented a trace capturing facility which roughly provides the same trace data as obtained by using `sys:trace`. The difference is, though, that a prerequisite for using the `sys` module is to start the generic behaviour with the debug option `trace` enabled. We are able to obtain the trace information from a generic behaviour that has been started without this option enabled. Another minor difference is that we record the event together with the present state, whereas the `sys` tracing functions present the updated state after the event.

With our tool traces can automatically be collected from multiple processes executing on multiple hosts in a distributed network. The real contribution of our tool starts after collecting this trace information.

A trace is a (long) sequence of events and process states, e.g., the state of a generic behaviour as passed as an argument to for example the `handle_call` function of the generic server call-back. In most software systems, a certain repetitive behaviour occurs after a while. That means that we see the same sequence of events and states re-occur in the trace. However, because of unique message tags added by the generic behaviours or because of a message counter, the events and states may differ even though we want to consider them equal on a more abstract level.

Our trace package is constructed to visualize re-occurring patterns in the trace. Parts of the trace can be abstracted away in a controlled manner (for instance, ignoring parts of a record structure). As such, different events may be visualized as one and the same abstract event and different states may be mapped to one and the same abstract state (transforming the linear structure of the trace into a graph structure). Such abstractions are written in Erlang itself and a set of standard abstractions are provided.

For analysis of trace data we are currently focusing on the task of extracting a model of the traced components. Essentially such a model is a finite-state graph where the nodes are a collection of the states of the traced processes, and the edges are events that cause a state change in any process. Normally it would be quite hard to recreate such a model

*The author was affiliated with Ericsson when the work described in this article was carried out.

ACM SIGPLAN Erlang Workshop '02 Pittsburg, PA USA

from an executing system because system states are difficult to recreate from a trace. However, due to the predominance of design patterns in Erlang software that explicitly record state information, we can actually obtain very useful state graphs in this manner.

These state graphs can, possibly after using our powerful abstraction mechanism, be directly visualized using graph drawing programs such as daVinci [6] or dot [7]; our software will generate graphs in these formats. In case graphs contain too much information to be amenable to visual inspection they can be exported by our package to external tools such as the model checkers of the CADP (Cæsar/Aldebaran) tool set [5], and checked against correctness properties encoded in the alternation-free modal μ-calculus [8].

Since the work on trace analysis spans several fields there are a large number of partially related works, especially in the field of software visualisation. A number of works focus on recreating the underlying protocol structures (state machines) of programs from collected trace sets. Ammons et al. [1], for instance, use machine learning techniques to discover from traces the protocol specifications that programs must adhere to when interacting with a certain application interface and a similar effort is reported in [4].

Efforts are under way to support limited model checking directly on the collected traces, without first generating a state graph model.

2. GENERIC BEHAVIOURS

The tracing package aims to trace the significant actions of a set of (possibly distributed) Erlang processes in order to be able to visualize and reason about inter-process communications and events. The end goal is of course to increase the understanding of cause-effect relationships among these processes. The process actions that thus needs to be traced are, for instance, process creation and termination, process communications and receptions and stores to the process global storage (though these are not common since Erlang is a functional programming language).

Much of the popularity of Erlang is due to the excellent libraries supplied with the language itself [9]. These offer an assortment of services and Erlang design patterns, ranging from a distributed database (Mnesia) to support for a generic client-server architecture which is commonly used in Ericsson's telecommunications applications. The design patterns, or generic behaviours, are of special interest to us, as the additional structure they impose on Erlang code helps to recover more of the underlying structure of the traced Erlang application from one of its traces.

The trace package currently has specialized support for tracing two such design patterns: `gen_server` and `gen_fsm`.

3. DESIGN AND IMPLEMENTATION OF A TRACE PACKAGE

The trace package can be broken down into the following separate analysis steps: (i) collection of raw trace data, (ii) abstraction of raw trace data, (iii) translation of abstracted trace data into a graph structure, (iv) presentation and, (v) analysis. We will examine each step in turn.

3.1 Collection of Raw Trace Data

Raw trace data is collected using the built-in trace functionality of Erlang. This offers, roughly, the ability to trace a set of significant events for a process such as, sending and receiving messages, spawning of new processes, calls to functions for which tracing is enabled, API calls, etc.

Tracing using our package is initiated through calling a function `gen_trace:dump/3` with a list of names of processes (possibly distributed onto different nodes) to trace. The processes should implement a generic server or generic finite state machine. First, the function call `gen_server:loop` or `gen_fsm:loop` is traced to obtain the name of the call-back module that implements the behaviour. This module is one of the argument of the loop function. Given this module name, the call-back functions in the behaviour can be traced. We trace all functions that are obtained by the *behaviour*`:info` function. For finite state machines we dynamically obtain the names of the states during the tracing process. Note that the name of the state is only part of what we actually call the state. Our notion of state for a finite state machine consists of both the name of the state and the data component at the moment that the process is in that state.

Thus, for the generic server we trace calls of the server (calls to the functions `handle_call` or `handle_cast`). Similarly for a finite state machine we trace all the calls to the state functions (calls of the form `module:statename` where `module` is the call-back module implementing the state machine and `statename` implements a state function), and for event handling code calls to the function `handle_event` are traced.

The `gen_trace:dump/3` function has as additional arguments a file name to write the trace in and a list of options, such as to whether tracing should be limited to a certain time interval, limited to a maximal set of relevant events, or continue until all traced processes have died.

On every node we create a process that receives all trace events. These processes all forward these events to a central monitoring process. Creating the trace processes on every node is necessary for the Erlang trace functions to work.

When a traced event occurs, the Erlang runtime system sends a copy of the event, with information about the process causing the event, via our trace process, to the monitoring process created by us. Raw trace data events are logged by the monitoring process using the efficient logging facility of Erlang/OTP.

3.2 Interpretation of Raw Trace Data

The result of the event tracing activity is a raw event log where events in the form of, for example, calls to the call-back functions of the behaviours of different processes are intermingled.

The task of the interpretation phase is to recapture the state notion for those processes that implement generic servers or generic finite state machines. We do this by refining, step-by-step, the trace log to record also the states of processes. The state of a process implementing a generic component is observable indirectly through the arguments to the function calls of the call-back module. Concretely, for the generic server software pattern, we observe the calls to the `handle_call`, `handle_cast`, and `handle_info` functions. The arguments are the request to service, the pid of the initiating process (in case of the `call`, and *the current state of the generic server process.* In the refined trace log the current state of the process (the last argument function calls we trace). Then the message, i.e. the first argument of the func-

tion call (the event causing a possible state change), is stored in an internal data structure, together with all future actions of the process until another call-back function is called. At this point all the actions between the state changing events have been recorded, and thus the resulting state can be written to the new trace log. The result is a refined trace log, where states of components are recorded together with the sequences of actions causing the state changes, and where interleaved events from other processes have been moved forward or backward in the trace log. We clarify this by the following example:

```
<0.49.0>: handle_cast({closed,1,2},{open,2,[1]})
<0.59.0>: handle_call(...)
...
<0.49.0>: gen_fsm:send_event(<0.53.0>,{move,down})
...
<0.49.0>: handle_call({approaching,1,1},<0.53.0>,
                      {moving,2,[1]})
```

Roughly the trace log records the activities of an elevator application. Process `<0.49.0>` implements a generic server behaviour that determines the scheduling logic of the elevator. The first trace event in the log corresponds to a message sent from the process controlling elevator 1 (a finite state machine) that its door has just been closed at floor two. Since the elevator has a request to proceed to floor 1 (this is recorded as the list `[1]` in the state argument of the `handle_cast` function), the scheduler should order the elevator process to start moving (line four in the trace log). However, before this happens, a call to another traced process `<0.59.0>` occurs in the trace log. Finally another message from elevator 1 is received, informing that the elevator is approaching floor 1.

In the refined trace log the states of the traced process have been recorded together with the event causing each state change, and the list of resulting actions of the process. Any actions by other processes are moved before or after such a big step transition of the traced component. The resulting trace is shown in the example below.

```
Process: <0.49.0>
Events: ???
State: {open,2,[1]}
```

⋮

```
Process: <0.49.0>
Events: handle_cast {closed,1,2} from <0.53.0>,
        gen_fsm:send_event(<0.53.0>, {move,down})
State: {moving,2,[1]}
```

⋮

```
Process: <0.49.0>
Events: handle_call {approaching,1,1} from <0.53.0>
State: ???
```

⋮

To reorder trace events in this manner would not normally be a sound operation, if by sound we understood to mean that an event causing another event must always occur before the second event in the trace log. Here the soundness of this reordering relies on the observation that the standard components (such as generic servers) read their messages in a first-in-first-out manner and are not sensitive to reception of messages when they are computing a response to an event, i.e., exactly the same value will be computed by the generic server for the reply and the next state. Furthermore communication in Erlang is asynchronous and performed using (in theory) perfect unbounded message channels that cannot reorder or loose messages.

3.3 Abstraction of raw trace data

Having recaptured the states of traced components the next step in trace analysis is to interpret, and possibly abstract, the result. Here, with interpretation we mean the operation of assigning meaning to the traced state data, and the traced actions causing or caused by state changes. Interpretation of data might, for example, mean to translate compactly represented data, e.g. for reasons of efficiency, to a more easily understandable representation.

As an example we could want to translate the event from an elevator to the scheduler `{closed,1,2}` to clarify that the first integer is the elevator number and the second integer is the present floor: "elevator 1 is on floor 2 with doors closed". Seeing this sentence in the trace might be more instructive.

At the same time, we might also be faced with a system in which we are not interested in some part of the data. In those cases we want to abstract from this data and ignore the part that we are not interested in. For example, we might want to ignore the actual number of the elevator in the events (in particular if we only have one elevator).

The difference between interpretation and abstraction is that the abstraction removes data and therefore states and events that are different before abstraction become equal after abstraction. This is important for our visualization later on, since equal abstract states appear only once in the graph constructed from the trace.

Clearly the interpretation and abstraction operations require user assistance since most such abstractions will be rather application and/or analysis specific. The abstractions are given as simple functions with fixed names, put in a call-back module for the newly defined `gen_trace` behaviour. Simply put, the generic trace module expects two functions to be defined in the call-back module, one that transforms the state into an abstract state and one that transforms an event in an abstract event; two functions from arbitrary Erlang terms to Erlang terms. We have chosen, however, to make the abstraction functions a bit more powerful by demanding a second argument. This argument is the history of the abstractions seen so far. It is a term that is user defined and can be seen as the internal state of the abstraction process.

The types of the functions in the generic trace call-back modules are:

```
+type init() -> history().

+type abstract_state(state :: term(),history()) ->
          {abs_state :: term(),history()}.

+type abstract_event(event :: term(),history()) ->
          {abs_event :: term(),history()}.

+type abstract_eventseq([term()],history()) ->
          {[term()],history()}.
```

The `init` function is called at the moment the abstraction process starts and determines the initial history. Depending on the kind of abstraction, the user could either just pass a

constant as history all the time (not using the power of this mechanism), or determine a transformation that depends on the history. For example, one could initialize the history with the number zero and increase the counter for every new state. Whenever the counter reaches a hundred, say, the state would be abstracted to the Erlang atom `and_so_on` for any next state. In our elevator example, a more useful thing might be to use the number to reflect, for example, on which floor the elevator came from. State abstraction for an elevator going down may then differ from an abstraction of the state when the elevator is going up (one could hide information on doors opening and closing for elevators going down, but present this when they go up).

An even more sophisticated use of the history is a kind of property checking. Assume that the elevator software contains a bug, viz. sometimes the doors do not open after pressing the button on the fourth floor. The history can be used to contain one of the three situations: `idle`, `button4_pressed`, and `has_opened_at4`. Whenever the event of opening the door at the fourth floor occurs, the history is changed to `has_opened_at4`, whenever the button is pressed on the fourth floor (while the elevator was not open), the history is changed to `button4_pressed`. Now, for every event of passing the fourth floor, i.e. approaching floor three or floor five, it is checked whether the history contains `button4_pressed`. If so, the error might have occurred and the state which would normally be abstracted to `moving` would now be abstracted to `moving_erroneous`. This enables to easily find the erroneous event sequences in your trace. This is all relatively easy to program in the call-back module as a few simple Erlang functions.

The functions `abstract_state` and `abstract_event` transform a state and event to an abstract state and abstract event respectively. The function `abstract_eventseq` is used as a filter over abstract events. This might be useful if a state transition is always caused by a certain sequence of events, but one only wants to record a few of them in the abstracted trace.

A number of standard abstractions are available. One example is an abstraction that extracts particular components from an Erlang tuple, another abstracts arbitrary terms by removing all subterms occurring below some depth in the original term.

As an example of a generic trace call-back module we present a module that assumes the state to consist of a tuple. It presents only the parts of the tuple that has changed from one state to the other. Thus, if the state is {`sched_elevator,1,<0.53.0>,closed,2,[]`} and the next state is {`sched_elevator,1,<0.53.0>,moving,2,[1]`}, then this abstraction will only show {`_,_,_,moving,_,[1]`} as the second state. All events are left untouched.

```
-module(abs_tuple_diff).

-behaviour(gen_trace).

-export([init/0,
         abstract_state/2,
         abstract_event/2]).

init() ->
  [].

abstract_state(State,History) ->
  NewState =
    difference(tuple_to_list(State), History),
  {list_to_tuple(NewState),NewState}.

abstract_event(Event,History) ->
  {Event,History}.

%%%%%%%% local functions %%%%%%%%%

difference([],_) ->
  [];
difference([E|Es],[]) ->
  [E|Es];
difference([E|Es],[F|Fs]) ->
  case E==F of
       true ->
         ['_'|difference(Es,Fs)];
       false ->
         [E|difference(Es,Fs)]
  end.
```

Assuming that the `gen_trace:dump/3` function has been used to trace a process and that this trace was saved in the file *mytrace.dump*. This trace can be abstracted[1] by the function `gen_trace:abstract(abs_tuple_diff,"mytrace.dump",[])`. The last argument of this function is a list of options on how to structure the trace data and which tool to use to present the data.

3.4 Structuring trace data

The next step in the analysis of trace data is to organise the structure in which the states and the traced data are to be interpreted. There are currently three such interpretations:

trace The first preserves the trace data structure which is essentially a linear trace of states and actions, where a state is a pair of a process identifier and an Erlang term, and actions are arbitrary Erlang terms. The resulting structure is a sequence of pairs where the first component is the list of events triggering and caused by a state change, and the second component is the combined state of all traced processes.

This interpretation is a re-grouping of the events, such that only events that are connected with a certain state change are associated with this change.

graph The second interpretation tries to identify regular behaviours in the code by constructing a state-graph model whereby identical abstract states in the linear trace are collapsed into the same graph node. Thus, after abstraction, states that where different before may become equal. In our elevator example we could abstract from the floor on which the elevator is and which floors it still has to serve. In that way, the states would only depend on the elevator number and the physical state: `moving`, `open`, and `closed`. In the graph there is no difference between the state where elevator one has the doors open on floor five, and the state where the elevator has the doors open on floor two. Therefore, the events from the elevator that the doors close

[1] Note that the same trace can be abstracted in several ways, using different call-back modules.

on floor two and that the doors close on floor five are both edges in the graph from the same node.

msc The third interpretation emphasizes the concurrent and distributed nature of the trace data by partitioning the linear trace data into a message sequence structure (typically illustrated in a message sequence chart) whereby the states of different processes are separated, and where events have direction. Events or actions are caused by a certain process (or the environment – i.e., untraced processes), and are sent to other traced processes or to the environment.

3.5 Presentation

The aim of the abstraction functions in the call-back module is to hide irrelevant data. If one chooses to structure the data in a graph, as described above, then abstract states that are equal form the same node in the graph. Sometimes one would like to hide certain information, but abstracting it away causes the graph to be different than expected. For example, if we use the abstraction module that abstracts tuples in such manner that only differences between tuples remain, then in our example there would not be a difference between the state of the scheduler when elevator one is moving from floor 3 to floor 1 or from floor 2 to floor 1 (provided that it opened the doors on both floors). This might very well be a wanted abstraction. One might be interested in monitoring the elevator when moving and opening and closing doors, independent of the floor the elevator happens to visit. However, if the aim is to reduce the detail in the final presentation of the graph, but where the states have to be considered different, then abstraction is not the right tool.

Similar to the abstraction functions for state and event, one might add functions for the presentation of the state and events in the call-back module:

```
typeset_state(abs_state :: term(), history()) ->
    {string(),history()}.

typeset_event(abs_event :: term(), history()) ->
    {string(),history()}.
```

Different from the abstraction functions these presentation functions and return a string instead of an arbitrary Erlang term. The functions are applied after abstraction, but if no abstraction functions are supplied in the call-back module, they are applied directly on the states and events.

In case no presentation functions are supplied in the call-back module, the Erlang `io:format` function is used to generate a string for visualization.

One of the major motivations for this work was to enhance the understanding of component-based Erlang code. To achieve this we provide translators from the structured trace data to visualizing tools such as daVinci [6] and dot [7]. As an example we extracted a typical state-graph model from runtime data of the lift example by monitoring both the lift scheduler process (a generic server) and the lift controller process (a finite state machine) and depicted the combined state space using dot in Figure 1.

While the picture may appear complex at first, to the programmer of the lift application it nicely illustrates the dynamics of the communications between the processes, and the management of external events. To make visualization more accessible we use dot to generate graphs in the SVG format (the Scalable Vector Graphics format of the W3C organization), which can be displayed in most web browsers.

3.6 Analysis

Often it is infeasible to obtain realistic program models for software verification from existing code, simply because unless drastic abstractions are made, the models will be much too large. Moreover, abstractions that preserve the essentials of the program behaviour are not easy to design. Further, because of the limitations of model checking, one is typically forced to limit the study of the abstracted model to certain scenarios, since normally only closed systems (where the environment is explicitly modelled) can be considered.

The trace package can automatically generate program models, using the graph model structure of trace data, which can be later checked using state-of-the-art model checkers. There are both drawbacks and advantages in this. First, of course, coverage of the total state space of the underlying program model will be limited by the actual interactions of the environment with the traced program. On the other hand constructing artificial, realistic, environments can sometimes be prohibitively difficult. To generate reasonable sized program models abstractions are of course required. Here our emphasis on tracing standard program components, with a clearly defined notion of a program state, can greatly help in defining the right abstractions, as does having access to visualization tools.

The trace package can output graphs, as well as linear trace data, in the Aldebaran format of the CADP tool set [5].

State information is preserved, although this is strictly not supported by the format, so that error traces resulting from failed model checking attempts can be directly related to the generated program model and visualized using the daVinci graph visualization tool.

We have experimented with model checking the generated program models using the model checker for the alternation-free modal μ-calculus supplied with the tool set. Initial experiments are positive, although for the target audience (Erlang programmers) it could prove highly beneficial to support Erlang syntax in recognizing states and actions, instead of using regular expressions over strings as the CADP tool set does.

4. EXPERIENCES

Our tracing tool was tested in two major examples: the elevator control software used in an Erlang introductory course and on the AXD 301 Distributed Application Controller (rcmDAC for short).

In the Erlang introduction course a set of modules is provided that implement the control software for a number of elevators, comprising around 1700 lines of Erlang code. A graphical user interface shows the building with elevators on the screen, such that changes in the software are visible to the students. The code is implemented according to the standard generic behaviours, like the generic server and generic finite state machine, with full functionality, fault tolerance and debugging support. Several errors are on purpose left in the code and the students have to find and correct the mistakes. A major obstacle to solving this task is to understand the dynamics of the software. We believe that the trace package can greatly help in this. By tracing the elevator scheduler or the finite state system representing the states of a single elevator, a student will get a better

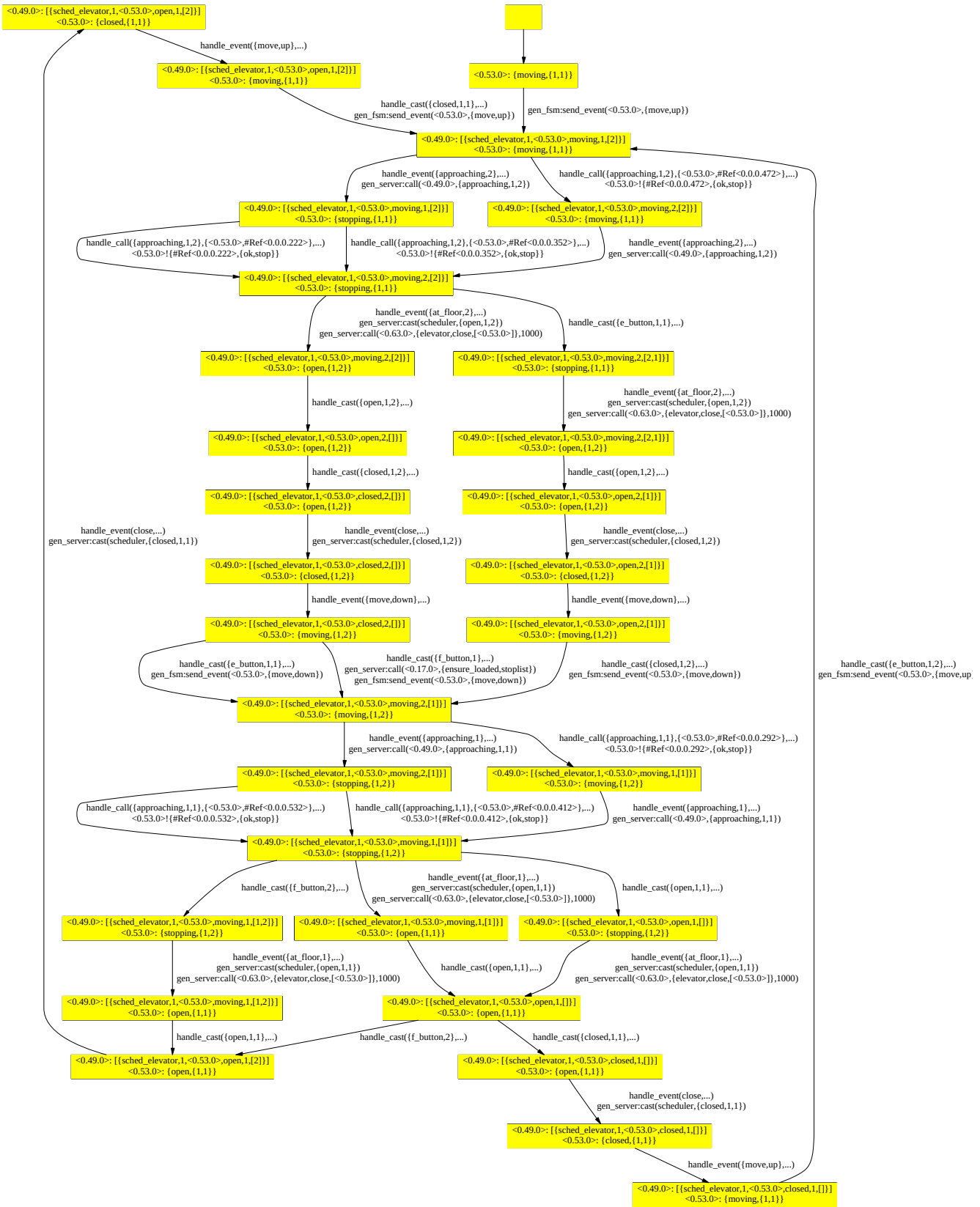

Figure 1: The Lift Example Visualized Using dot

understanding of the actual code. Using abstractions on the traces results in extremely intuitive visualizations of the state changes in the processes. A trace of the elevator, for example, can be depicted as a graph with nodes with numbers in them (representing the floor the elevator is at) and two arrows, one labelled **up** and one labelled **down**, pointing to a node with larger number and smaller number in it. Of course, the bottom and top node only have one arrow.

Finding the abstractions for the events and states in order to obtain these pictures is rather easy given the raw trace. As a side effect of finding them, the students learn quickly what the data structures in the code represent and as such they learn to understand the details of the software and are able to fix problems.

Software written for educational purposes is one thing, the reality is often more complicated. We asked the developers of the AXD 301 ATM switch to identify one of the more difficult and less understood processes in their system, for us to evaluate the trace package on. They came up with the Distributed Application Controller (rcmDAC). The AXD 301 consists of a number of Erlang nodes. On each node rcmDAC is one of the first processes that is started. It controls the loading of most of the applications on the node. If a node fails, another rcmDAC is responsible for loading the required applications somewhere else. The rcmDAC processes exchange messages to select a leader process, to coordinate the moving of applications. This leader election protocol is re-executed whenever the elected leader process dies (e.g. because the node fails). The rcmDAC processes notify each other regarding which applications are loaded on each node, and further communicate with several other processes that depend on the fact that certain applications are loaded.

To trace the rcmDAC processes we restarted an AXD 301 with four nodes three times and collected all relevant trace data during the ten minute startup of the switch. This resulted in much smaller traces (about 300 events per process) than the designers of the software had originally expected, indicating that they identified complex behaviour with a lot of events and state changes. The fact that the software was meant to capture a lot of possible situations, did not necessarily mean that many of these situations occurred during startup. Analyzing the trace further we found that the state of the rcmDAC consists of a record with seventeen fields. Without any abstractions the visualized startup graph was ugly and not very enlightening.

Because of the complexity of the state, we could abstract in several ways, clarifying different aspects of the same trace. For example, we used visualization to verify that one of the fields (*request to load*) in the record 'followed' another field (*acknowledge loaded*), i.e. always received the same value a few events after the first field got this value. In the same way we provided a few more abstractions that help the designers to understand and more properly document the startup procedure of the AXD.

5. CONCLUSIONS

We implemented a tracing tool that collects trace data from running systems. The tool is based on the tracing primitives in the Erlang runtime system, which allow tracing of e.g. messages and function calls without re-compiling code. This is a great advantage when obtaining trace data from complex systems comprising both hardware and software, where there is a great reluctance to code modifications.

In our work we focused in particular on tracing code that has been written according to certain design principles, to enable an easy recovery of the state of the processes. An abstraction mechanism was defined on traces, which maps concrete Erlang states and events to abstract one. This abstraction mechanism has shown in experiments to be easy to use even by inexperienced Erlang programmers.

One particular interpretation of the trace data is the graph obtained by collapsing identical trace states. This presentation of the data has, because of the predominance of state in Erlang components, proved to provide valuable insight into the source code of the traced application, and has been shown to be beneficial for documenting purposes.

The success of the method of tracing software in this way comes with the right choice of abstraction functions. Our experience is that designers of a system already have an accurate picture of which the important parameters of a complex state (or action) are, but often fail to appreciate the full dynamics of a system. Further, since they have no problems with programming Erlang, they are the right people to write these abstraction functions.

The trace package is currently primarily used for visualization and documentation purposes, but we hope to be able to use it more extensively for checking correctness properties of the software by means of traces in the future.

6. REFERENCES

[1] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In In *Proc. POPL'02*, 2002.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition)*. Prentice-Hall International (UK) Ltd., 1996.

[3] S. Blau and J. Rooth. AXD 301 - a new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.

[4] J. Cook and A. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[5] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, 1996.

[6] M. Fröhlich and M. Werner. The graph visualization system daVinci – a user interface for applications. Technical Report 5/94, Department of Computer Science; Universitt Bremen, 1994.

[7] 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.

[8] D. Kozen. Results on the propositional μ-calculus. *Theoretical Comput. Sci.*, **27**:333–354, 1983.

[9] S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.

[10] U. Wiger. Four-fold increase in productivity and quality – industrial-strength functional programming in telecom-class products. In *Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems (FEmSYS 2001)*, 2001.