Erlang Training and Consulting Ltd

Exago

Aniko Nagyne Vig and Atilla Erdodi

Stockholm, 13th Nov 2009

What can we use log files for?

The ideal audit log files capture the essence of the system. Contains a permanent record of the most important key points of the life of the working system.

There are many other tool around to analyse them: mainly to provide statistics, make it more "human readable"

Why Exago is different?

In Exago we have a different goal:

Use the audit logs to test your live system and your logs.

Even after a tested system is deployed and it is in production for years, from time to time there is a need to prove the system behaves correctly in specific cases, or an error needs to be located. The first attempt is to use the log files. We would like to ease this process with Exago.

A solution

Exago: An audit log monitoring tool.

Prototype by Hans Svensson, Chalmers Univ.

Development by Erlang Training and Consulting Ltd.

How to describe the expected behaviour?

- Using Prolog or some proprietary language is a solution, but needs time to learn ...
- Check it against a state machine!
- Simple yet powerful.
- By adjusting the granularity, we can check more complex properties as well ...

Idea

- Reconstruct what happened and create a series abstract commands. During this process the dependencies and connections between the fields of log files can be checked.
- See if the system behaved properly, using some sort of model checking – building an abstract state machine for the system. Every log entry will represent an edge in the state machine.
- Keep the tool general enough to allow configuring different level of detail checks, creating different state machines from the state system logs.

How can we transform the logs to a state machine?

Lines in the log files → … → State machine

- Based on the content of one or more entries will be one edge in the state machine
- Entries in the log files needs to be grouped to form one execution instance of the model

Terminology

- Transaction:
  - A set of events that describe a state change
  - To be abstracted to a single command
- Session:
  - A series of transactions to be checked against a state machine

NOTE:

- These terms depend on the level of detail,
- not necessary mean the same from the system's and tool's perspective.

Available interfaces:

- Erlang record,
- command line,
- GUI built on wxWidgets,
- (planned Emacs and Eclipse integration).

Results so far

- Two industrial case studies, from a different perspective
- Production system for years: *(TPSG)*
  - Took approximately 2 day to create the model
  - After spent several years on testing the system, we still managed to found bugs just by analysing log files!
- Using Exago as a tool of development *(SMSC)*
  - In progress ...
  - Helping quality assurance

To use Exago, you have to:

- Get some log files from the production system,
- Specify relations between the files and their formats,
- Provide abstraction and validation functions for the processing,
- Present an abstract model of the system as a state machine.

Log files

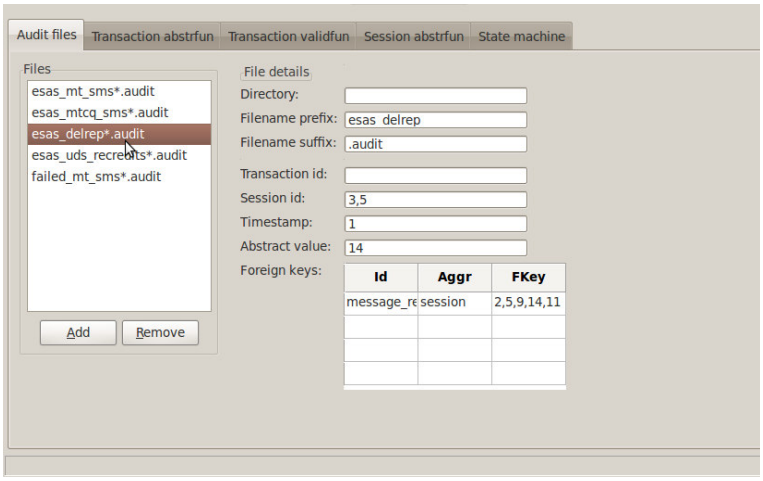
Preconditions:

- must be CSV formatted (plans for supporting other formats as well),
- must provide „enough information”.

Exago specific:

- we specify a set of log files at a time using wildcards

Log files



Exago - configuration

Giving the details of individual log files:

- Directory inside the given base one
- Filename prefix and suffix
- Transaction and session identifiers
- Abstract value
- Foreign keys

Important Values for the analysis

- Timestamp
- Session id
- Transaction id
- Abstract value

Specify the values for the tool:

- Describe them using a list of integers denoting the positions in the CSV log files
- eg.: [3,11,5]

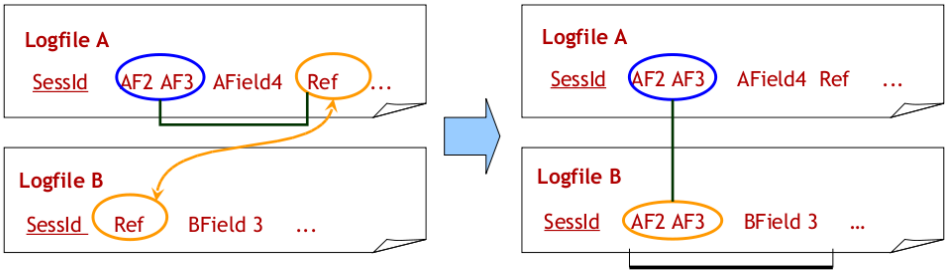
Relations

Configuration for logfile A:

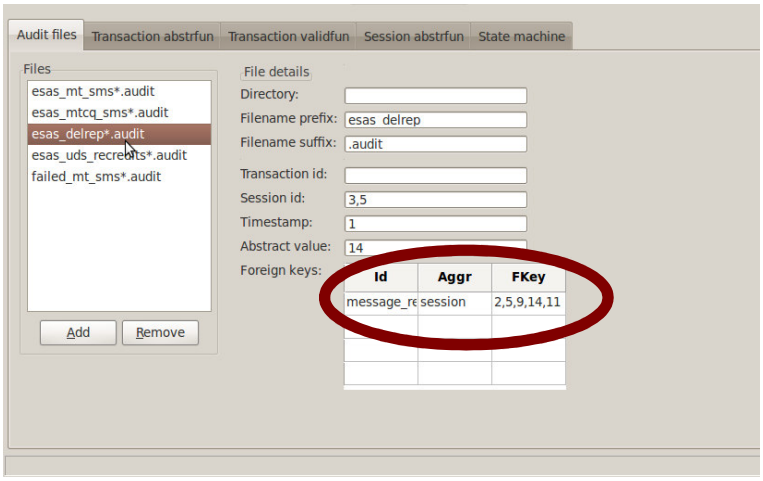
```
{ …
  session_id = [1],
  fkeys = [{ref_id, [2,3], [5]}]
  ... }
```

Configuration for logfile B:

```
{ …
  session_id = [1],
  abstract_value = [{ref_id, [2]}, 3]
  ... }
```



Relations



Abstraction

Parsed and resolved log entries

→ *aggregate and abstract* →

Transactions

→ *filter* →

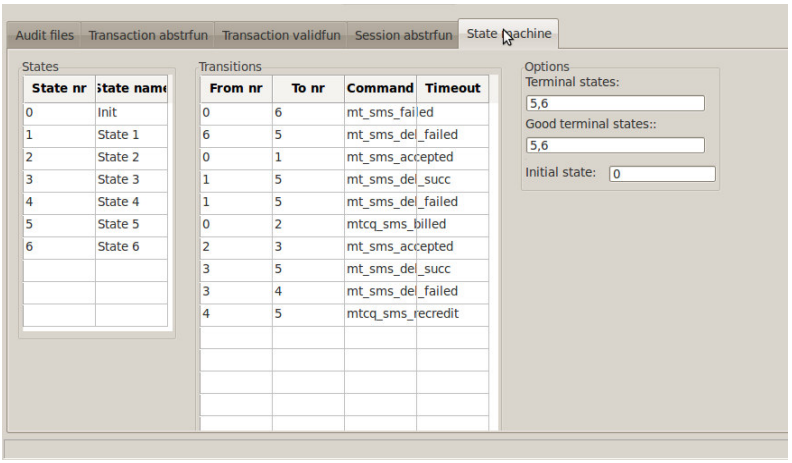
Filtered transactions

→ *abstract and group* →

Sessions

NOTE: Transaction abstraction and validation are optional

State machine



Output

- Generated HTML output,
- Errors and warnings are listed,
- Failed sessions are listed, failed command highlighted,
- Generated state machine with graphviz,
  with highlighted terminal state.

What happens after the analysing the results?

- In optimal case the result show up no problems, so the system and logging is verified.
- If any problem was found, the reason needs to be find, it was in the logging procedure, or in the functional part of the system.
  - Further knowledge can be involved by not just using the tools, but support engineers or developers
  - Further detailed debugging using on-line tracing on the specific problem on the live system by onviso for example
- The found and fixed issue can result in a new unit or system test case, and can be used to test with continuous integration if the tool was used during the development phase

Pros and cons

Main advantages:

- Don't need access to the live system
- Can analyse errors without reproducing it,
  just check the log entries at the time the failure occured

Disadvantages:

- Limited usability: logging must meet certain requirements
- The information is limited in the audit log files

Erlang Training and Consulting Ltd

Exago Exercises

Anikó Nagyné Víg and Atilla Erdődi

Stockholm, 13 Nov 2009

The “classic” elevator example

Modifications:

- Events are captured to file for analysis
- Unique id is generated for each event

Deciding the model

We abstract out the buttons, the command queue and only care about *what* happened but not *why*.

Exercise 1: Define the model in Exago

0. Generate some log files

- Start the elevator example with tracing:
  - util:start_trace(1,3,3).
- Call an elevator to floor 2
- Send elevator 2 to floor 2
- Send elevator 3 to floor 3

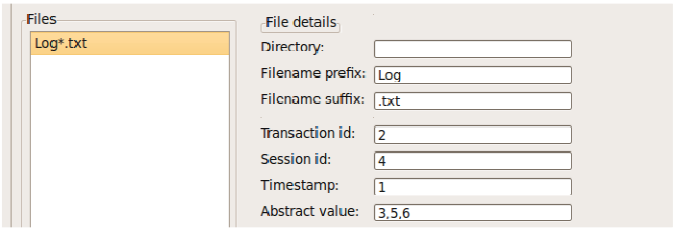
Log file format

1. Define the log file in the tool

- Only one file:
  - Log.txt (or as set in the app file)
- Format:
  - Timestamp, EventId, Event, ElevatorId, [Parameters]

Log file format

- ElevatorId → Session Id
  - We don't have *real* sessions in this application
- EventId → Transaction Id



Transaction abstraction

2. Define the abstraction functions

- Simple case: one event per transaction
- Trivial transaction abstraction function

```
1  fun(_TrId, [{Timestamp, SessionId, AbstrVal, File} | _]) ->
2      {Timestamp, AbstrVal}
3  end.
```


Transaction filtering

- Filtering out the events
  we don't take into account in our model

```
1  fun({_Timestamp, AbstrVal}) ->
2      case AbstrVal of
3          {"reset", "closed", _Floor} -> true;
4          {"open", none, none} -> true;
5          {"close", none, none} -> true;
6          {"approaching", _Floor, none} -> true;
7          {"stopped_at", _Floor, none} -> true;
8          _ -> false
9      end
10 end.
```


Session abstraction

- Creating the abstract commands

```
1  fun(Trs) ->
2      lists:map(fun({Timestamp, AbstrVal}) ->
3                    {Timestamp,
4                    case AbstrVal of
5                        {"reset", "closed", Floor} ->
6                            list_to_atom("reset_to_" ++ Floor);
7                        {"open", none, none} ->
8                            open;
9                        {"close", none, none} ->
10                           close;
11                       {"approaching", Floor, none} ->
12                           list_to_atom("approaching_" ++ Floor);
13                       {"stopped_at", Floor, none} ->
14                           list_to_atom("stopped_at_" ++ Floor);
15                       _Other ->
16                           _Other
17                   end}
18     end, Trs)
19 end.
```


State machine

3. Define the state machine

- 10 states for 3 elevators and 3 floors

States

Nr.	State name
0	Init
1	Closed at floor 1
2	Open at floor 1
3	Approaching floor 1
4	Closed at floor 2
5	Open at floor 2
6	Approaching floor 2
7	Closed at floor 3
8	Open at floor 3
9	Approaching floor 3

Transitions

From nr	To nr	Command	Timeout
0	1	reset_to_1	
0	4	reset_to_2	
0	7	reset_to_3	
1	2	open	
2	1	close	
3	1	stopped_at_1	
6	4	stopped_at_2	
4	5	open	
5	4	close	
4	3	approaching_1	
4	9	approaching_3	

Options

Terminal states: 0,1,2,3,4,5,6,7,8,9

Good terminal states:: 0,1,2,3,4,5,6,7,8,9

Initial state: 0

Interpreting the results

4. Run the tool and analyse the results

Take a look at one of the failed sessions:

```
{"2009-11-11 14:22:22:0089135","reset_to_1"}
{"2009-11-11 14:22:27:0713588","approaching_1"}
{"2009-11-11 14:22:29:0345272","approaching_2"}
{"2009-11-11 14:22:29:0956680","stopped_at_3"}
{"2009-11-11 14:22:29:0957266","open"}
{"2009-11-11 14:22:30:0960854","close"}
{"2009-11-11 14:22:31:0985493","approaching_3"}
{"2009-11-11 14:22:33:0621364","approaching_2"}
 ...
```


Interpreting the results

The log files claims the elevator tries to approach floor 1...

```
{"2009-11-11 14:22:22:0089135","reset_to_1"}
{"2009-11-11 14:22:27:0713588","approaching_1"}
{"2009-11-11 14:22:29:0345272","approaching_2"}
{"2009-11-11 14:22:29:0956680","stopped_at_3"}
{"2009-11-11 14:22:29:0957266","open"}
{"2009-11-11 14:22:30:0960854","close"}
{"2009-11-11 14:22:31:0985493","approaching_3"}
{"2009-11-11 14:22:33:0621364","approaching_2"}
{"2009-11-11 14:22:34:0232767","stopped_at_1"}
{"2009-11-11 14:22:34:0233367","open"}
{"2009-11-11 14:22:35:0237494","close"}
{"2009-11-11 14:22:36:0261430","approaching_1"}
{"2009-11-11 14:22:37:0892879","approaching_2"}
{"2009-11-11 14:22:38:0504843","stopped_at_3"} ...
```


Interpreting the results

… but we are already on floor 1!

{"2009-11-11 14:22:22:0089135","reset_to_1"}
{"2009-11-11 14:22:27:0713588","approaching_1"}
{"2009-11-11 14:22:29:0345272","approaching_2"}
{"2009-11-11 14:22:29:0956680","stopped_at_3"}
{"2009-11-11 14:22:29:0957266","open"}
{"2009-11-11 14:22:30:0960854","close"}
{"2009-11-11 14:22:31:0985493","approaching_3"}
{"2009-11-11 14:22:33:0621364","approaching_2"}
{"2009-11-11 14:22:34:0232767","stopped_at_1"}
{"2009-11-11 14:22:34:0233367","open"}
{"2009-11-11 14:22:35:0237494","close"}
{"2009-11-11 14:22:36:0261430","approaching_1"}
{"2009-11-11 14:22:37:0892879","approaching_2"}
{"2009-11-11 14:22:38:0504843","stopped_at_3"} ...

Finding the cause

The abstract commands do not show the the cause of the error in this case, neither the experienced “symptoms”, but clearly indicates that the elevator does not match even with this basic state machine.

Finding the cause

However, we can see there is something wrong when the elevator leaves the floor.

Examining the code

The bug is in the elevator.erl:

```
moving({approaching, Floor}, {ENo, NewFloor}) -> %% Oops...
   sys_event:approaching(ENo, NewFloor),
   case scheduler:approaching(ENo, NewFloor) of
{ok, stop} →
   sys_event:stopping(ENo),
   {next_state, stopping, {ENo, Floor}};
{ok, continue} →
   {next_state, moving, {ENo, Floor}};
_Other ->
   sys_event:stopping(ENo),
   {next_state, stopping, {ENo, Floor}}
   end;
```

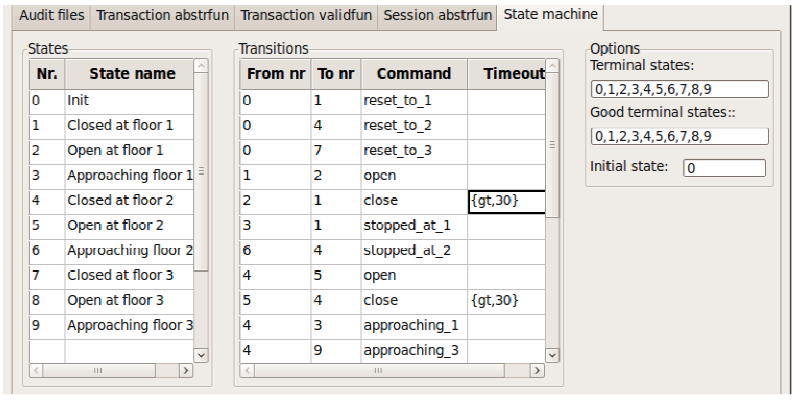

Exercise 2: add time constraints

The elevator door should remain opened
for at least 3 seconds

<=>

At least 3 second should pass between
state „open” and state „closed”

Exercise 2: solution



Thank you for your attention!

Any questions?