Verifying Fault-Tolerant Erlang Programs

Clara Benac Earle
Universidad Carlos III
Madrid, Spain

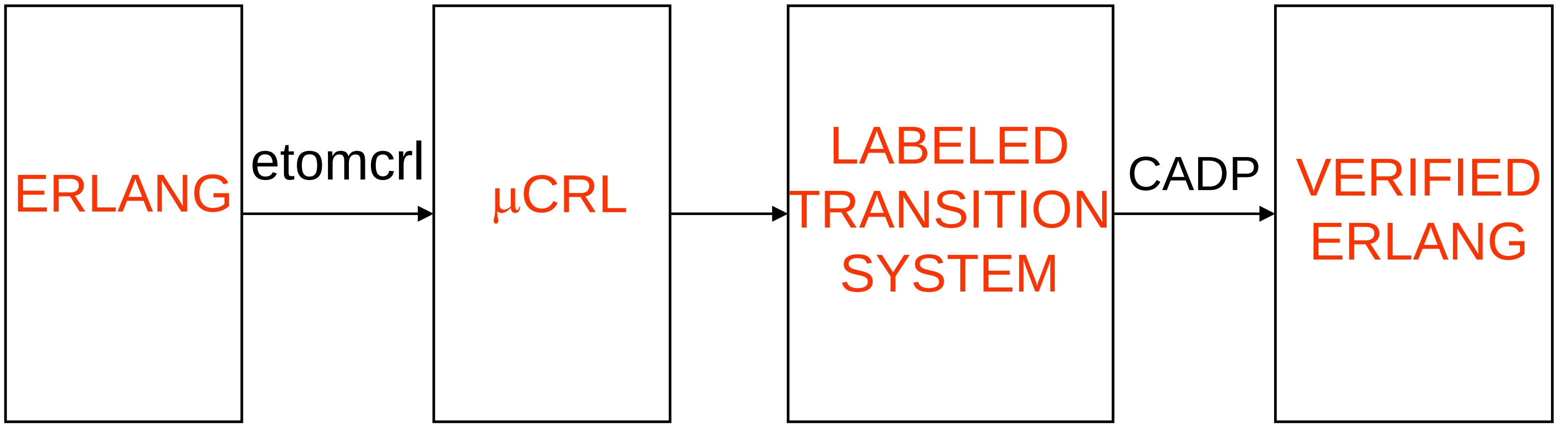
Lars-Åke Fredlund
Universidad Politécnica
Madrid, Spain

John Derrick
University of Sheffield
England, UK

Outline of the talk

- Previous work:
  - T. Arts, C. Benac Earle, J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. on Software Tools for technology Transfer.* Vol. 5, pp 205-220, 2004.
  - C. Benac Earle. Model checking the interaction of Erlang components. PhD thesis, University of Kent, Canterbury, 2005.
- Extension for handling fault-tolerance
- Conclusions, future work

Verification: methodology



Translating an Erlang subset

- Functional part
  - Data types: atoms, numbers and pids
  - Variables and patterns
  - Expressions: data types, variables, lists, tuples and records
  - Functions, including higher-order functions
- Modules

Translating an Erlang subset

Processes and concurrency

- We handle Erlang Behaviours!
  - Not Erlang send and receive but:
    - Generic server behaviour (gen_server): client-server applications
    - Supervisor behaviour

Translation target: µCRL

µCRL is a process algebra with data

- Different types of data are described using sorts
- Functions over sorts are given by rewrite rules
- Processes use synchronous communication

Translation scheme

- Separation of side-effect-free functions and functions with side-effects
- SEF functions are translated into a set of rewrite rules and SE functions are translated into μCRL processes.
- Message queues are translated into μCRL processes

Etomcrl: the translation tool

- Input: Erlang code that uses the generic server component for communication between processes and the supervisor component for starting child processes

- Output: A μCRL specification initialized with the processes started by the supervisor component

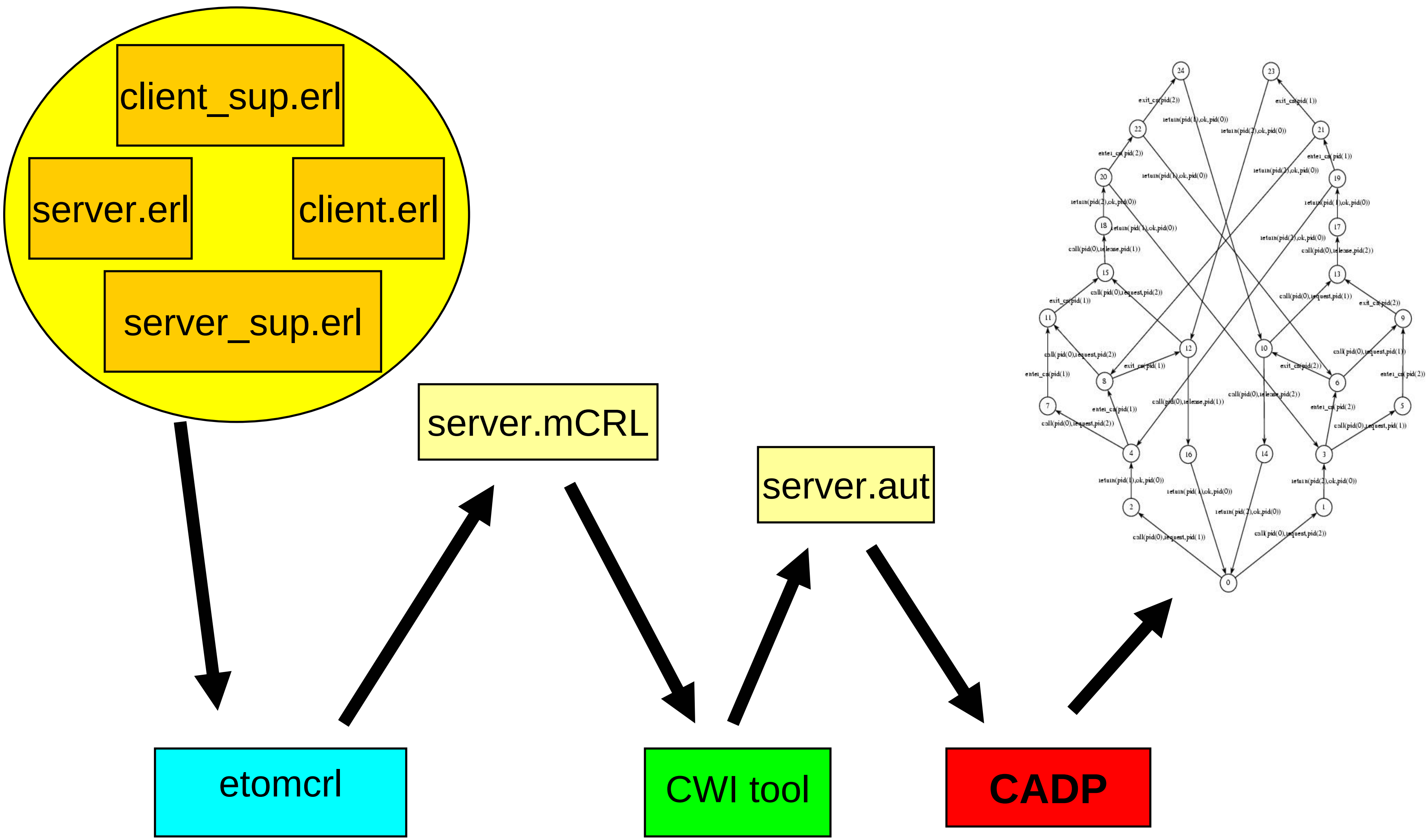
example

```
-module(client).

start_link(Server) ->
 {ok,spawn_link(loop,[Server])}.

loop(Server) ->
  gen_server:call(Server,request),
  enter_critical(self()),
  exit_critical(self()),
  gen_server:call(Server,release),
  loop(Server).
```

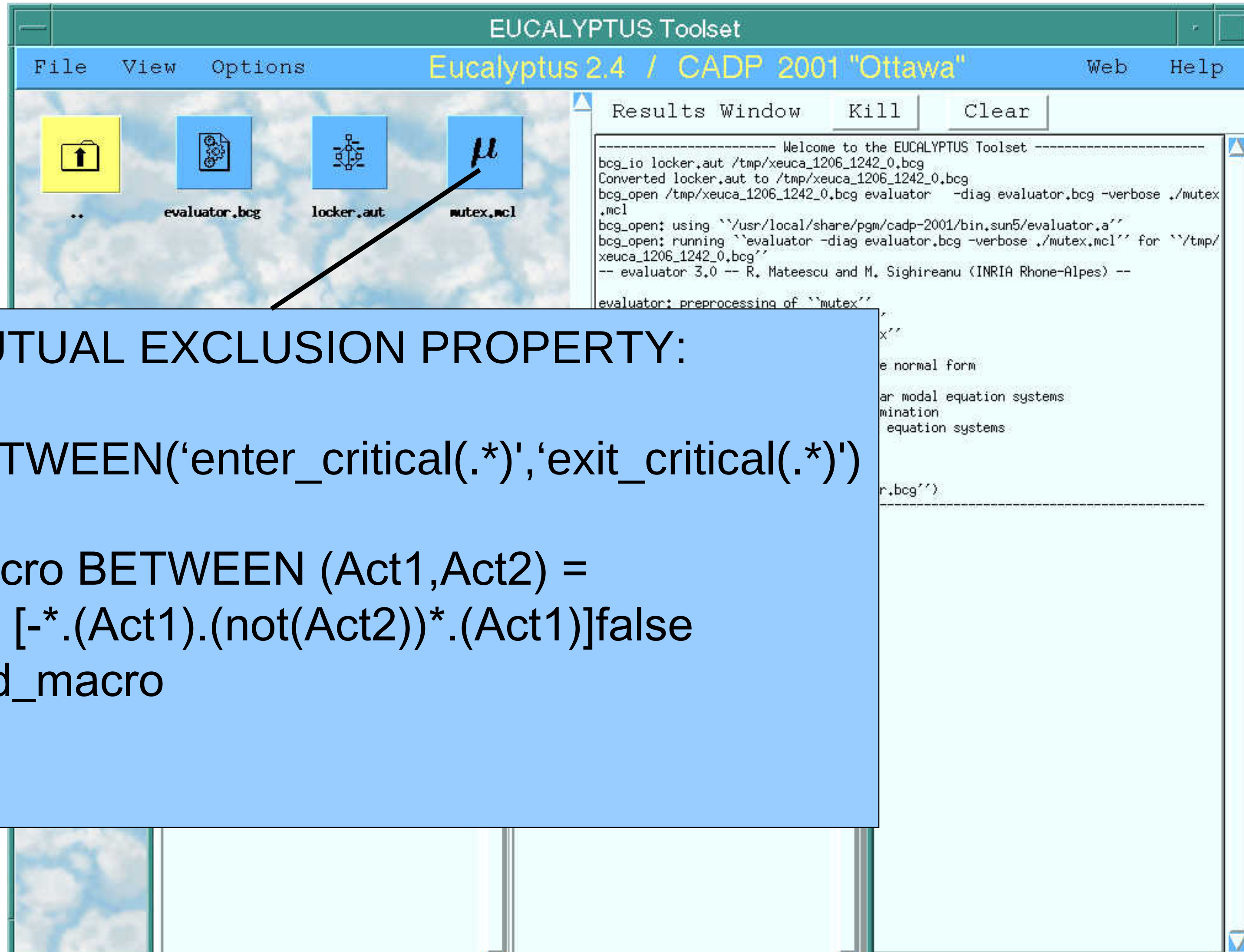
example

```
start_link() ->
    gen_server:start_link(server,[],[]).

init([]) ->
    {ok,[]}.

handle_call(request,Client,Pending)->
    case Pending of
       [] ->
           {reply, ok, [Client]};
      _ ->
           {noreply, Pending ++ [Client]}
    end;

handle_call(release, Client, [_|Pending]) ->
    case Pending of
       [] ->
           {reply, done, []};
      _  ->
           gen_server:reply(hd(Pending), ok),
           {reply, done, Pending}
    end.
```

Verification



Model Checking Software



MUTUAL EXCLUSION PROPERTY:

BETWEEN(‘enter_critical(.*)',‘exit_critical(.*)')

macro BETWEEN (Act1,Act2) =
    [-*.(Act1).(not(Act2))*.(Act1)]false
end_macro

Fault-tolerance in Erlang

- Establish links between processes
- If a process *A* terminates abnormally, a signal is sent to all linked processes, which will terminate abnormally or will receive the message in their mailbox
- Supervisor component

Example of fault-tolerant code

```
init([]) ->
    process_flag(trap_exit,true),
    {ok,[]}.

handle_call(request,{ClientPid,Tag},Pending)->
    link(ClientPid)
    …
handle_info({'EXIT',ClientPid,Reason},Pending) ->
    NewPending = remove(ClientPid,Pending),
    case available(ClientPid,Pending) of
       true ->
           gen_server:reply(hd(NewPending), ok),
           {noreply,NewPending};
       _  ->
           {noreply,NewPending}
    end.
```

Fault-tolerance: translation

- Translate fault handling code (handle_info)
- Extend the translation from Erlang to μCRL to include the possibilites of faults
  - Add μCRL code corresponding to the crashing of a client

Adding crashing points

- Between issuing a generic server call and receiving the reply
- After receiving the reply from the server
- After issuing a generic server cast if there was at least one generic server call to the same server before

Mutual Exclusion

BETWEEN(a1,a2,a3) = [-*.a1.(¬a2)*.a3]false

MUTEX() =
BETWEEN(´enter_critical(.*)´,´exit_critical(.*)´.enter_critical(.*)´)

Counter-example

“call(server,request,C1)”

“reply(C1,ok,server)”

“enter_critical(C1)”

“info(server,{EXIT,C1,EXIT})”

“call(server,request,C2)”

“reply(C2,ok,server)”

“enter_critical(C2)”

FT_MUTEX

FT_BETWEEN(a1,a2,a3,a4) =
[-*.a1.(¬a2 V a3)*.a4]false

FT_MUTEX()=
FT_BETWEEN(´enter_critical(.*)´,´exit_critical(.*)´,´info(.*)´,´enter_critical(.*)´)

Another example

```
handle_info({´EXIT´,ClientPid,Reason},Pending) ->
   NewPending = remove(ClientPid,Pending),
   case NewPending == [] of
       false ->
         gen_server:reply(hd(NewPending),ok),
         {noreply,NewPending};
       _->
         {noreply,[]}
   end.
```

Counter-example

“call(server,request,C1)”
“reply(C1,ok,server)”
“call(server,request,C3)”
“info(server,{EXIT,C3,EXIT})”
“enter_critical(C1)”
“exit_critical(C1)”
“reply(C1,ok,server)”
“call(server,request,C2)”
“call(server,release,C1)”
“reply(C2,ok,server)”
“enter_critical(C2)”
“reply(C1,done,server)”
“enter_critical(C1)”

Conclusions

- Checking fault tolerance is hard
- In Erlang it is easier, because of
  - Language support for fault tolerance (links)
  - High-level components reduces the number of program locations where failures have to be handled
- As a consequence the state spaces we generate automatically are relatively small, and thus checkable
- The verification method is general, and reusable for a class of fault-tolerant Erlang client-server programs

Future Work

- Extending the tool
- Supporting other design patterns, including user-defined behaviours
- Equivalence Checking

- Download etomcrl from http://etomcrl.sourceforge.net