Verifying Erlang Telecommunication Systems with the Process Algebra μCRL

[1]Qiang Guo, [1]John Derrick and [2]Csaba Hoch

[1]Department of Computer Science,
The University of Sheffield,
Regent Court, 211 Portobello, S1 4DP, UK
{Q.Guo, J.Derrick}@dcs.shef.ac.uk
[2]Faculty of Informatics
Eötvös Loránd Tudományegyetem
Pázmány Péter sétány 1/c., 1117 Budapest, Hungary
hoch@inf.elte.hu

Abstract. Verification is an important process in the development of Erlang systems. A recent strand of work has studied the verification of Erlang applications using the process algebra μCRL. The general idea is that Erlang programs are translated into a μCRL specification, upon which the standard model checkers can be applied for checking the system's properties. In this paper, we pull together some of the existing work and investigate the verification of an Erlang telecommunication system in μCRL. This case study uses a server-client structure and incorporates timing restrictions and is designed and implemented using a number of Erlang/OTP components. We show how this system is translated into a μCRL specification by using the defined rules, after which system properties are checked via the toolset CADP. Through studying the verification of such an application, we aim to validate the effectiveness of the translation rules in an integrated way.

Key words: Erlang, Telecoms case study, Process Algebras, μCRL, Translation, Verification.

1 Introduction

Erlang [1] is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems. It is available under an Open Source Licence from Ericsson, and since its conception its use and development has widened to a number of sectors such as TCP/IP programming, etc.

A key feature of Erlang is the Open Telecom Platform (OTP) architecture where generic components are encapsulated as design patterns, each of which solves a particular class of problem. These patterns include servers, supervisors, finite state machines etc. This makes Erlang an ideal programming language for the development of fault-tolerant systems containing soft real-time requirements.

Verification is an important part of the Erlang system process. Although Erlang has many high-level features, verification can be still non-trivial. A number

of possible approaches have been explored, including the one we investigate here: abstract an Erlang application into a formal model, upon which model checking [9] techniques can be applied. This approach has recently been applied to the verification of Erlang programs and OTP components [2,3,5,7,14,16] where the process algebra μCRL [13] has been used as the formal language upon which verification is carried out.

Arts *et al.* [2,3] initiated this strand of work and proposed rules for translating Erlang syntax and the OTP components *gen_server*, *supervisor* into μCRL. Benac-Earle [5] continued with the work and developed a toolset, *etomcrl*, to automate the process of translation. Guo *et al.* extended the work by proposing a model for the translation of the OTP finite state machine *gen_fsm* [14] and defining rules for coping with Erlang *timeout* events in μCRL [16].

However, rules for the translation of OTP *gen_server*, *supervisor*, *gen_fsm* and Erlang *timeout* have, so far, only been independently evaluated via some small examples, and no work has evaluated these rules in an application where the above components are integrated as a system. One might argue that if all rules are applied in an integrated way, will they show the similar effects for system verification as they demonstrated in the existing work? Moreover, will a state space explosion mean that effective verification is lost?

In this paper, we attempt to look at these questions by investigating the verification of an Erlang telecommunication system in μCRL. A telecommunication system of server-client structure and timing restriction for operation is developed with Erlang/OTP. The system integrates the use of the *supervisor*, *gen_server*, *gen_fsm* components and uses explicit *timeout* events. We show how the system is then translated into a μCRL specification using the proposed translation rules. We then verify a number of system's properties by using the model checker CADP [8] and investigate the changes of state space when the number of clients increases. The experimental results suggest that when being applied in an integrated system, the translation rules show the similar effect for system verification as being applied independently.

The paper is organized as follows: Section 2 describes a telecommunication system that is used as a case study in this paper; Section 3 implements the system with Erlang programming langauge; Section 4 discusses the translation of the telecoms case study into μCRL; Section 5 looks at the system verification using the standard model checker CADP; conclusions are drawn in Section 6.

2 Telecommunication system

In this section we give an overview of our case study, which is implemented in Erlang in Section 3.

2.1 System infrastructure

Our telecoms case study uses a client-server structure, and comprises of a database server (DBS) that is used to maintain all client's data and a number of functional servers (FS) that will process clients' requests. An FS has a capacity and

a user list. The capacity defines the maximum number of clients (mobiles) that can be connected to a server, while, the user list saves all clients (mobiles) that have been connected to this server. The telecoms system illustrated in Figure 1 is designed with one DBS (named as DB), three FSs (named as SVR_1, SVR_2 and SVR_3) and five clients (named as M_1, M_2, M_3, M_4 and M_5).

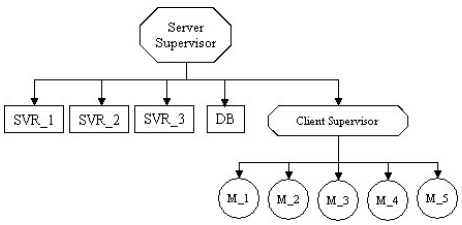

Fig. 1. Telecoms system designed with one DB, three FSs and five clients.

Once the system starts up, an FS can communicate with DBS and any other FSs. A client can communicate with any FSs, and can perform some functional operations such as *calling* and *top-up*. The behaviour of clients is described in section 2.2. Each client has an account maintained in the DBS, and in order to make a phone call, a client needs to save enough money in its account.

Before performing any functional operations, a client needs to connect to an FS. After being connected to an FS, the client's identity is maintained in the FS' user list. A client can only be connected to one FS, and if a client has connected to an FS and tries to connect to another FS, an error message will be returned and the request is denied. When a client disconnects, the appropriate FS cuts off the connection and removes this client from the user list to release the resource. The FS will notify all other FSs about the changes of its clients' state so that they can correctly respond to the client's requests.

2.2 Client behaviour modelling

The behaviour of a client (mobile) is modeled as a finite state machine (FSM), and the initial design is shown in Figure 2. There are four states: *idle*, *connected*, *calling* and *top_up*, where initially, the system is set to the *idle* state.

The FSM defines the behaviour of a number of operations: *connecting*, *disconnecting*, *calling*, *terminating*, *top_up* and *cancelling*. Before performing any operations, a client FSM needs to connect to an FS through sending the *connecting* request. If the FS replies $\{ok,connected\}$, it indicates that the request is accepted and the connection is set up. The FSM moves to the state *connected*; otherwise, if $\{error,busy\}$ arrives, it suggests that the server has reached it maximum capacity. The client will send the *connecting* request to another FS. The FSM remains in the state *idle*. The client will iteratively send the *connecting* request to each FS until the connection is approved and set up by an FS.

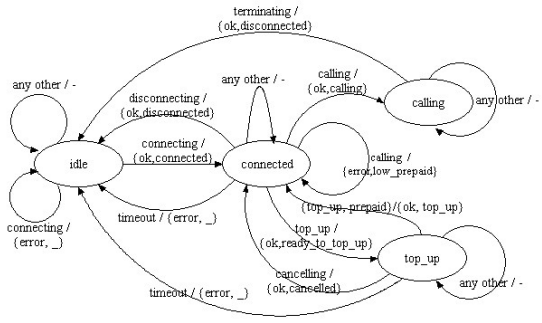

Fig. 2. Client behaviour modelled as an FSM

A client can stop the connection by sending the *disconnecting* request to the FS. Once the *disconnecting* request being received, the FS will cut off the connection and remove the client from its user list to release the resource. A reply message {*ok,disconnected*} will be sent to the client, and upon receiving this reply, the FSM will be reset to the state *idle*.

When in the state *connected*, a client can make a phone call through *calling* operation or top up it account through *top_up* operation. When the *calling* request is sent, if its account has enough money, a client will receive {*ok,calling*} from the appropriate FS, enabling the calling process. The FSM then moves to the state *calling*; otherwise, {*error, low_prepaid*} will be received, asking the client to top up its account. The FSM remains in the state *connected*.

When in the state *calling*, only the *terminating* operation can terminate a calling process. This prevents the calling process from being disrupted by some unintended actions. When the *terminating* request is sent, the DBS will reduce the amount of money from this client's account. The FS then cuts off the client's connection and releases the resource from its user list. Meanwhile, a message {ok,*disconnected*} is sent to the client, and on receiving the reply, the FSM is reset to the state *idle*.

After being connected to an FS, a client can ask to top up its account by sending the *top_up* request. If {*ok, ready_to_top_up*} is received, it indicates that the top up process is accepted by the FS, and the FSM moves to the state *top_up*. The client can then transfer money to its account through {*top_up, Prepaid*} operation where *Prepaid* is the amount of money that is about to be transfered. When the transaction succeeds, the FS replies the client with {*ok, top_up*} and when receiving such a reply, the FSM returns to the state *connected*.

A client FSM has a timing restriction applicable when in states *connected* or *top_up*. Specifically, when the FSM is directed to the state *connected* or *top_up*, a timer will be instantiated which enables the timing process. If, within the predefined time period, no action is performed by the client, a *timeout* event will be generated and sent to the FS. By receiving *timeout* event, the FS cuts

off the connection and releases the resource from its user list. The FSM is then reset to the state *idle*.

3 Erlang Implementation

Erlang is used to implement the telecoms system, making use of the OTP design patterns as is common practice.

3.1 Functional server implementation

The functional server (FS) is implemented using the Erlang/OTP *gen_server* module. A generic server is implemented by providing a *callback module* where (*callback*) functions are defined specifying the concrete actions of the server such as server state handling and response to messages.

In this work, the callback function *handle_call* in the FS module is comprised of two parts. One (using keyword *request*) processes client's requests; the other (using keyword *notify*) deals with the notification sent from FSs:

```
handle_call({request,R,M},Fr,Chs)→                :   case Request of
  {Reply,State}= handle_request(R,M,Chs),         :     connecting →
  {reply,Reply,State};                            :         do_connecting(M,Chs);
handle_call({notify,M,F},Fr,Chs)→                 :     disconnecting →
  {C,N,MList,SVRList} = Chs,                      :         do_disconnecting(M,Chs);
  case F of                                       :     cancelling →
    add →                                         :         do_cancelling(M,Chs);
      {reply,ok,{C,N,                             :     calling →
          MList++[M],SVRList}};                   :         do_calling(M,Chs);
    remove →                                      :     terminating →
      {reply,ok,                                  :         do_terminating(M,Chs);
        {C,N,delete(M,MList),SVRList}};           :     top_up →
  end.                                            :         do_top_up(M,Chs);
handle_request(timeout,M,Chs)→                    :     {top_up, Prepaid} →
  do_timeout(Mobile,Chs);                         :         top_up_tranfer(M,Chs)
handle_request(Request,M,Chs)→                    :   end.
```

The internal variable *Chs* (defined by the Erlang system for saving values) is defined with the form of {*C, N, MList, SVRList*} where *C* defines the FS' capacity, *N* counts for the number of clients that has been connected to this FS, *MList* saves clients that have connected to this FS and *SVRList* saves all FS servers running in the system.

The function *handle_request(Request,Mobile,Chs)* is defined where a list of *do* functions is called to process client's requests. The function *do_connecting* is defined to set up connection between the FS and a client. It first examines whether the FS reaches its maximum capacity. If the FS is full, {*error, busy*} will be returned to the client; otherwise, the connection is set up. The client is then registered in *MList* and *N* is increased by 1. The function *do_disconnecting* is defined to disconnect a client from the FS. When the *disconnecting* request arrives, the FS cuts off the connection and then, by calling the function *notify_servers*, notifies other FSs (saved in *SVRList*) to release the resource (removes the client from *MList*).

The *do_calling* function monitors the calling process. When the *calling* request is received, the function reads the client's data from the DBS and checks whether it has enough money for making a call. If so, the calling process is enabled; otherwise, {*error, low_prepaid*} is replied, asking the client to top up its account. When the client finishes calling, it sends the *terminating* request to the FS. Upon receiving the request, the FS enables the *do_terminating* function to subtract amount of money from the client's account, and then cut off the connection through *disconnecting* operation. The function *do_top_up* and *top_up_tranfer* are defined to top up client's account. When the *top_up* request is received, the FS enables the process by replying the client with {*ok,ready_to_top_up*}. Once the client's money is received, the *top_up_tranfer* function is enabled to complete the transaction.

3.2 Client implementation

The client behaviour is implemented using the OTP *gen_fsm* module, and the state transition rules are defined conforming to the following convention:

StateName(Event, StateData) →
... code for actions ...;
{*next_state,StateName′,StateData′,Timer*}.

where the state function returns a tuple that contains the name of the next state, $StateName'$, and an updated state data, $StateData'$. *Timer* is an optional element, if it is set to a value, a timer is instantiated, and a *timeout* event will be generated when the time-up occurs. The function *send_event* is defined to trigger a transition. When *send_event* is executed, the *gen_fsm* module automatically calls the *current state* function.

In accordance with the design given above, four state functions are defined in the client module: *idle*, *connected*, *calling* and *top_up*. The state function *idle* initiates a *connecting* request to the FS *SVR*. If the FS *SVR* replies the FSM with {*ok,connected*}, the request is accepted and the connection is set up. The FSM moves to the state *connected*; otherwise, the request is denied and the FSM remains unchanged.

```
idle([Act,SVR],{M,_RSVR,SVRList})→          :       {error,busy}→
  case member(SVR,SVRList) of               :            display(server,busy),
    true →                                  :            {next_state,idle,
      F=gen_server:call(SVR,{request,Act,M}),:                {M,nil,SVRList}}
      case F of                             :    end;
        {ok,connected}→                     :  false →
          display(connected),               :    display(server,invalid)
          {next_state,connected,            :    {next_state,idle,{M,nil,SVRList}}
            {M,SVR,SVRList},20000};         :end.
```

Once the client is connected to an FS, an event will trigger the state function *connected*, which evaluates the request and then makes decisions for the consequent actions. For example, if a *calling* request is made, the function will call the FS to evaluate the client's state. If the client has enough money in its account, {*ok,calling*} will be returned to approve the calling process, and upon receiving the reply, the FSM moves to the state *calling*.

```
connected(timeout,{M,SVR,SVRList})→          :          display(client,calling),
  gen_server:call(SVR,{request,timeout,M}),  :      {next_state,calling,
  display(M,timeout),                        :        {M,SVR,SVRList}};
  {next_state,idle,{M,nil,SVRList}};         :      {error,low_prepaid}→
connected([Act,_SVR],{M,SVR,SVRList})→       :          display(low_prepaid),
  case Act==terminating of                   :          {next_state,connected,
    true →                                   :             {M,SVR,SVRList},20000};
      display(action,invalid),               :      {ok,ready_to_top_up}→
      {next_state,connected,                 :          display(ready_to_top_up),
           {M,SVR,SVRList},20000};           :          {next_state,top_up,
    false →                                  :             {M,SVR,SVRList},20000};
      F=gen_server:call(SVR,{request,Act,M}):      _Other →
      case F of                              :          display(action,invalid),
        {ok,disconnected}→                   :          {next_state,connected,
          display(disconnected),             :             {M,SVR,SVRList},20000}
          {next_state,idle,                  :    end
               {M,SVR,SVRList}};             : end.
        {ok,calling}→                        :
```

When in the state *calling*, only the *terminating* action can stop the calling process. This prevents the calling process from being disrupted by any unintended actions.

```
calling([Act,_SVR],{M,SVR,SVRList})→         :              {M,nil,SVRList}};
  case Act of                                :     false →
    terminating →                            :       display(server,invalid),
      gen_server:call(SVR,{request,Act,M}),: :       {next_state,calling,
      display(call,terminating),             :             {M,SVR,SVRList}}
      {next_state,idle,                      : end.
```

When being connected to an FS, the client can ask to top up its account by sending the *top_up* request to the FS. If {*ok,ready_to_top_up*} is replied, the top up process is enabled, and the FSM moves to the state *top_up*. An action will trigger the state function *top_up* to either start the transaction by {*top_up, Prepaid*} operation (*Prepaid* is the amount of money the client is about to transfer), or cancel the process by sending the *cancelling* request.

```
top_up(timeout,{M,SVR,SVRList})→             :  {ok,cancelled} →
  gen_server:call(SVR,{request,timeout,M}),  :      display(top_up,cancelled),
  display(M,timeout),                        :      {next_state,connected,
  {next_state,idle,{M,nil,SVRList}};         :           {M,SVR,SVRList},20000};
top_up([Act,_SVR],{M,SVR,SVRList})→          :  _Other →
  case gen_server:call(SVR,{request,Act,M}) of:      display(action,invalid),
    {ok,top_up} →                            :      {next_state,top_up,
      display(top_up,ok),                    :          {M,SVR,SVRList},20000}
      {next_state,connected,                 : end.
          {M,SVR,SVRList},20000};            :
```

When the FSM moves to the state *connected* and *top_up*, a timer is initiated. The timer is set to 20,000ms. If within the time period, no action is performed, a *timeout* event will be generated and sent to the FS. The FSM is reset to the state *idle*. A function *command* is defined to simulate the receiving of external actions. It calls *gen_server:send_event* to triggers the state functions.

4 Translating our case study into μCRL

In this section we describe the verification methodology used in this project. It uses the process algebra μCRL (micro Common Representation Language) [13] which is an extension of the process algebra ACP [4], where equational *abstract data types* have been integrated into the process specification to enable the specification of both data and process behaviour. We assume the reader is familiar with μCRL.

4.1 Pre-processing

Before the translation begins, the Erlang input is pre-processed which transforms the Erlang code into an optimized format, but has identical behaviour. For example, Erlang makes extensive use of pattern matching in its function definitions, and overlapping between patterns could lead to the system being represented by a faulty model in μCRL. This work transforms Erlang programs using the techniques discussed in [15] where pattern matching clauses in a function are replaced with a series of calling functions, each of which being guarded by the function *patterns_match*.

For example, the function *handle_request* is transformed as shown above. A data structure, called a *Structure Splitting Tree (SST)* [15], is applied for pattern evaluation, and the use of such an SST for pattern evaluation guarantees the elimination of overlapping between patterns in the transformed program.

```
handle_request(R,M,Chs) →                : hr_case_6(true,R,M,Chs,Vars) →
  hr_case_1(eval:pattern_match([R],      :   do_top_up(M,Chs,Vars);
      [connecting]),R,M,Chs,[]).         : hr_case_6(false,R,M,Chs,Vars) →
                                         :   hr_case_7(eval:pattern_match([R],
hr_case_1(true,R,M,Chs,Vars) →           :       [{top_up,Prepaid}]),R,M,
  do_connecting(M,Chs,Vars);             :           Chs,Vars++[Prepaid]).
hr_case_1(false,R,M,Chs,Vars) →          : hr_case_7(true,R,M,Chs,Vars) →
  hr_case_2(eval:pattern_match([R],      :   top_up_transfer(M,Chs,Vars);
      [disconnecting]),R,M,Chs,Vars).    : hr_case_7(false,R,M,Chs,Vars) →
...                                      :   {error, action}
```

4.2 Translating the server component

Erlang performs synchronous and asynchronous communications using the generic server primitives *gen_sever:call / handle_call* and *gen_sever:cast / handle_cast* respectively. One then has to model both synchronous and a synchronous communication in μ CRL, and to do so we use a *Server_Buffer* process (described in [5]). A data type *GSBuffer* is defined to contain the data for the process *Server_Buffer*. The actions *gen_server_call* and *gscall* are defined to write a message to the buffer while, *gshall* and *handle_call* to read a message from the buffer.

The database server is translated into a process *SDB*, and it maintains a number of clients, each in the form {*CName,Prepaid,State*} where *CName* is the client's name, *Prepaid* shows the amount of money saved in the client's account and *State* indicates whether the client is connected to an FS or not. The actions *server_read_db* and *db_send_data* are defined to read a client's data out from the process, *server_read_db* | *db_send_data* = *read_db*. The actions *servr_update_db* and *db_ack_request* are used to update the client's data in the process, *servr_update_db* | *db_ack_request* = *update_db*.

The functional server (FS) is translated into a process *server*. The process contains a server ID *SVRID*, a capacity *C* and a client list *CLs*. It uses the action *handle_call* to receive the client's requests: when a request is received, the *server* process first checks whether the request is made for itself. If so, the process calls the process *handle_request* to tackle the request; otherwise, its returns without changing anything.

```
proc server(SVRID:Term,C:Term,CLs:Term) =
  sum(SVR:Term,sum(Request:Term,sum(Client:Ter,
      handle_call(SVR,Request,Client).
      handle_request(SVR,Client,Request,C,CLs))))
        ◁ eq(SVRID,SVR) ▷ server(SVRID,C,CLs)
```

For each client's request, a request process is defined. Once the *handle_request* is enabled, it selects the corresponding request process. The selected process first performs all pre-defined actions and then replies the client with a message through the action *gen_server_reply*. For example, when the *connecting* request is received, the process *handle_request_ connecting* is activated. It first checks whether the server reaches its maximum capacity. If the server is full, the process goes back to the process *server* without changing anything; otherwise, if the client has not been connected to a server before, the process sets up the connection and replies the client with {*ok, connected*}.

```
handle_request_connecting(SVRID:Term,CL:Term,C:Term,CLs:Term) =
  sum(Vals:Term, server_read_db(Vals).
    (gen_server_reply(SVRID,tuple(error,tuplenil(unregister_user)),CL).
     server(SVRID,C,CLs)
        ◁ is_nil(find_client(CL,Vals)) ▷
          (gen_server_reply(SVRID,tuple(error,tuplenil(already_connected)),CL).
           server(SVRID,C,CLs)
              ◁ eq(find_client(CL,Vals),CL) ▷
                (server_update_db(CL,find_client(CL,Vals),CL),true).
                 gen_server_reply(SVRID,tuple(ok,tuplenil(connected)),CL).
                 server(SVRID,C,list_append(CL,CLs))
        ◁ mcrl_less(list_number(CLs),C) ▷
            gen_server_reply(SVRID,tuple(error,tuplenil(busy)),CL).
            server(SVRID,C,CLS)))))
```

The process *handle_timeout* is defined to deal with *timeout* event. Once the *timeout* event is generated from a client, the process *handle_timeout* will be activated. It cuts off the connection between the server and the client and removes the client from its user list to release the resource.

4.3 Translating the client component

The client was initially modelled as an FSM and implemented using the OTP *gen_fsm*. This is then translated into μCRL using techniques defined in [14], where the translation process is comprised of two parts, *simulating state management* (SSM) and *state function translation* (SFT).

In this work, a (one place) stack is used to perform the SSM which is modified by using a global variable (GV) process. A GV process contains a list of indexed GVs, where each GV is of the format {*VName, Val*} where the *VName* gives the variable's name and the *Val* the value. A GV process with three GVs, V_1, V_2 and V_3, is defined as follows:

```
proc
  GVs(V1:Term,V2:Term,Var3:Term) =
    sum(V:Term,receive_val(V).
        (GVs(V,V2,V3) ◁ eq(element(int(1),V),element(int(1),V1)) ▷
            (GVs(V1,V,V3) ◁ eq(element(int(1),V),element(int(1),V2)) ▷
                (GVs(V1,V2,V) ◁ eq(element(int(1),V),element(int(1),V3)) ▷ delta))))
  +
  send_val(V1,V2,V3).GVs(V1,V2,V3)
```

A GV can be read out through the actions *read_val* *send_val*, *write_val* | *receive_val* = *write*, and be modified through the actions *write_val* / *receive_val*, *write_val* | *receive_val* = *write*. We use a GV to stand for a client where *VName* and *Val* are used to save the client FSM's *current state* and the *state data* respectively. We found that, by applying such a modification, the state space is largely reduced.

The process *receive_cmd* is defined to receive commands generated from the external actions. For each client, a unique ID *CLID* is associated with all its FSM processes. Once a command is received, the process *receive_cmd* calls the process *read_clients*. According to the *CLID*, the designated FSM's *current* state and the *state data* are read out. The corresponding state process is then selected for performing all defined actions. Once the execution of the state process finishes, the FSM moves to the process *fsm_update_state* to update the *current* state and the *state data*.

For example, when the FSM is in the state *idle* and the *connecting* command is received, the state process *fsm_idle* is activated. The process *fsm_idle* sends the request to an FS and then waits for reply. If the FS returns *busy*, the process will calls for another FS; otherwise, the connection is set up. The process then calls for the process *fsm_update_state* to update the state *connected* as the current state. Thus we have the following:

```
fsm_idle(CLID:Term,Data:Term,Cmd:Term,SList:Term,SMList:Term) =
   gen_server_call(hd(SList),Cmd,ClID).
   wait_for_reply(hd(SList),CLID,Data,Cmd,SList,SMList)

wait_for_reply(SVRID:Term,CLID:Term,Data:Term,Cmd:Term,) =
  sum(S:Term,sum(R:Term,sum(CL:Term,
     gen_server_replied(S,R,CL).
     ((client_info(S,R,CL).
       (fsm_idle(CLID,Data,Cmd,SMList,SMList)
          ◁ is_nil(SList) ▷ fsm_idle(CLID,Data,Cmd,tl(SMList),SMList))
            ◁ is_busy(element(int(2),R)) ▷
              (fsm_update_state(CLID,connected,Data,SList,SMList,true,false)
                  ◁ is_connected(element(int(2),R))▷
                    fsm_update_state(CLID,idle,Data,SList,SMList,false,false,2))))
         ◁ eq(CL,CLID) ▷ wait_for_reply(SVRID,CLID,Data,Cmd)))))
```

The process *fsm_update_state* is parameterized with two arguments, *FT* and *FTM*. The *FT* determines whether the updated current state has timing restrictions on it; the *FTM* decides whether the process will be terminated due to some unexpected events. If the newly updated current state process has timing restrictions, the *FT* will be set to *true*, which enables the process *fsm_timing* to count down the time. If, within the predefined time period, no external action is performed, a *timeout* event will be

generated and sent to the FS. Afterwards, the process is terminated by setting the *FTM* to *true*. Thus we have the following output from the translation process:

```
fsm_update_state(CLID:Term,SNext:Term,Data:Term,Cmd:Term,
                 SList:Term,SMList:Term,FT:Term,FTM:Term,TR:Nat) =
    write_val(tuple(CLID,tuplenil(tuple(SNext,tuplenil(Data))))).
    (delta ◁ eq(FTM,true) ▷
        (fsm_timing(CLID,SNext,Data,SList,SMList,on(TR))
            ◁eq(FT,true)▷ receive_cmd(SList,SMList)))
```

The process *fsm_timing* and the process *count_down* are parameterized with a *timer* [16]. By using an explicit *tick* action in the process *count_down*, we apply a discrete-time timing model to support the translation of *timeout* event. When the process *fsm_timing* is called at the first time, the timer t is initiated and initialized. The process will either call for the process *count_down* to start the timing process or the *receive_cmd* process to continue with another external command.

```
fsm_timing(CLID:Term,SNext:Term,Data:Term,SList:Term,SMList:Term,t:Timer) =
    count_down(CLID,SNext,Data,SList,t) + receive_cmd(SList,SMList)
```

When the *count_down* process is activated, it checks whether the *timer* expires (using the function *expire(t:Timer)*). If not, the process will first perform the *tick* action once, standing for the passing of one time unit. The process then moves back to the process *fsm_timing*, counting down the timer t by one unit (*pred(t)*); otherwise, if the timer expires, the process *fsm_update_state* is called, with the next state *SNext* being reset to *idle* and the *FTM* to *true*.

```
count_down(CLID:Term,SNext:Term,Data:Term,SList:Term,SMList:Term,t:Timer) =
    tick.fsm_timing(CLID,SNext,Data,SList,SMList,pred(t))
        ◁ not(expire(t)) ▷
            gen_server_call(hd(SList),timeout,CLID).
            fsm_update_state(CLID,idle,nil,nil,SList,SMList,false,true)
```

4.4 System translation

By considering the translation of server section and client section together, the system is translated into a complete μCRL specification. In the specification, every server and client are initialized with a unique client ID. For each client, the process *client_cmds(CLID, CmdList)* is applied to initialize a list of external actions where *CLID* indicates the client's ID and *CmdList* saves the sequence of commands. A client receives a command through the action *r_cmd(CLID, Cmd)*.

A client sends a request to an FS through the action *gen_server_call(SVRID,Cmd, CLID)* where *SVRID* indicates the target server ID while *CLID* the sender's ID. A client receives a reply through the action *gen_server_replied(SVRID,Reply,CLID)* where *SVRID* shows from which server the reply comes and *CLID* indicates to which client the reply is sent.

When receiving a request, an FS process compares its ID with the received *SVRID* to examine whether the request is made for the server itself. If so, the request is accepted and the consequential actions will be performed; otherwise, the FS process ignores the request and returns without changing anything. Similarly, when receiving a reply, a client process compares its ID with the received *CLID* to check whether the message

is replied to the client itself. If so, the client process performs the actions extracted from the reply; otherwise, the process ignores the message. Through ID checking, a peer-to-peer communication structure is defined in the μCRL specification.

We have now reached a point whereby the design, as implemented in Erlang/OTP has been translated (in fact, abstracted) to a μ CRL specification, and we now described how properties of the initial design can be checked on this model.

5 Verifying the telecommunication system with μCRL

In this section, a number of system properties are abstracted and verified. In our experiments, the property under verification (PUV) is devised in a way where the behaviour of FS(s), the behaviour of client(s) and the communication between the FS and the client are considered as an integrated whole. Thus, instead of focusing on particular individual components, the properties we are concerned with in this case study are defined across the whole system.

5.1 Property verification

The system used for simulation is constructed as shown in Figure 1 where three functional servers (svr_1, svr_2 and svr_3) and five clients (m_1, m_2, m_3, m_4 and m_5) are used. We initialize the capacity of every server to 1. The clients m_1, m_3, m_5 are preset with £1 in their accounts, while m_2, m_4 with £0. We define that the minimum cost of making a phone call to be £1. The timer for the functions with timing restriction is set to 20,000ms, and we define the passing of one time unit as 10,000ms, represented by one *tick* action. As discussed in Section 3.2, the *gen_fsm:send_event* is often called through external actions. Therefore, before starting a simulation process, for each client FSM, a sequence of actions needs to be initialized in the process *Client_Cmds* to simulate the external behaviour.

We first devise two experiments to verify the system's client-server property. In the first experiment, the client m_1 attempts to make a phone call while m_2, m_3, m_4 and m_5 are idle; in the second, the client m_2 tries to make a phone call while m_1, m_3, m_4 and m_5 are idle. Thus, for both these initial experiments, only one client is active. Through these two experiments we want to check whether the FS(s) and the client act as defined in design, and whether the communication between the FS and the client is correctly running.

The commands for the two experiments are coded in the list *Cmd = cons(connecting, cons(calling, nil))* and initialized in the process *client_cmds* respectively. The Labelled Transition Systems (LTSs) derived from the toolset CADP [8] are shown in Figure 3 and 4. Here, we hide the actions *update_db* and *read_db* as internal actions, denoted by i in the LTSs.

Verification of the properties can be performed by using the model checker CADP, where the system properties are formalized by a set of temporal logic formulae. For example, in the first experiment, to check "without being connected to svr_1, m_1 cannot make a phone call.". This property can be formalized as:

[not(client_info(m_1, connected, svr_1))*. client_info(m_1, calling, svr_1))] false

Similarly, to check "when m_1 is connected to svr_1, without delaying enough time (two *tick* actions being consecutively performed), a *timeout* event cannot be generated.", the property is formalized as:

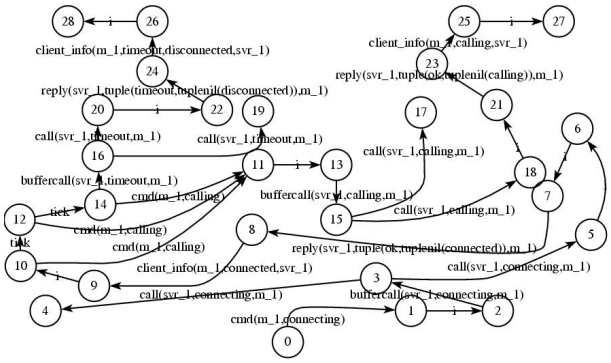

Fig. 3. LTS: The client m_1 makes a phone call.

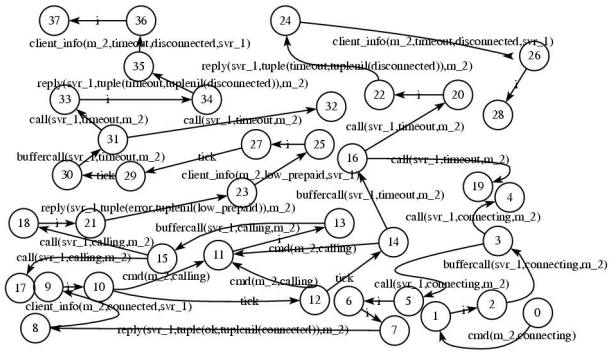

Fig. 4. LTS: The client m_2 makes a phone call.

[true*. client_info(m_1, connected, svr_1)*]
<not('tick.tick')*. client_info(m_1, timeout, disconnected, svr_1)> false

In the second experiment, to check "when m_2 is connected to svr_1, if m_2 has not preset enough money in its account, the calling process cannot be accepted.", the property is formalized as:

<true*. client_info(m_1, connected, svr_1) *. client_info(m_2, low_prepaid, svr_1) *. client_info(m_1, calling, svr_1)> false

Next, we construct an experiment to examine the system's behaviour where more than one clients are active. Two clients m_1 and m_2 request to connect to a server simultaneously. Since the capacity of the FS is set to 1, according to the design, when an FS, for example svr_1, accepts the request of a client, say m_1, it should reply the other m_2 with *busy*; the client m_2 should afterwards request a connection to svr_2. Similar to the previous experiments, we want to check the behaviour of FSs and the clients in an integrated way, but use more complicated system structure. Figure 5 illustrates the derived LTS. Here, the actions *call*, *buffercall* and *reply* are hidden as internal actions as well.

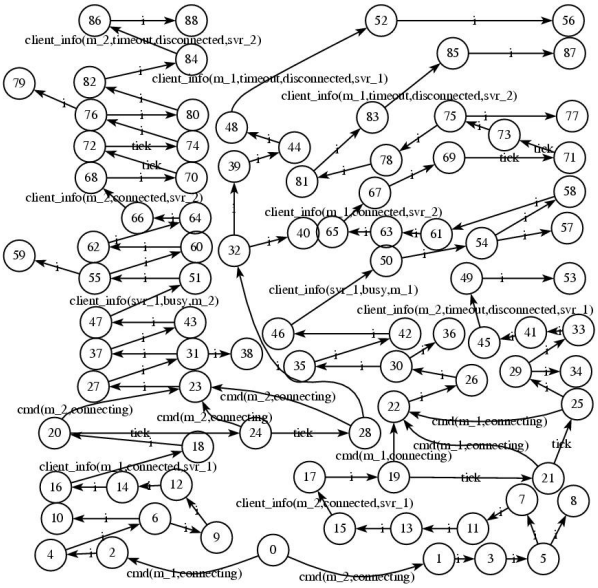

Fig. 5. m_1 and m_2 make requests to connect.

A number of properties can then be automatically verified via CADP. For example, to check "when m_1 is connected to svr_1 and m_2 requests to svr_1, svr_1 will reply m_2 with *busy*.". The property is formalized as:

<true*. client_info(m_1, connected, svr_1) *. cmd(m_2, connecting) *.
client_info(m_2, busy, svr_1)> true

Another property we want to check is formalized as:

<true*. cmd(m_2, connecting) *. client_info(m_2, busy, svr_1) *.
cmd(m_2, connecting) *. client_info(m_2, connected, svr_2)> true

stating that "when m_2 requests to connect to svr_1 and receives the reply of *busy*, it will request to connect to svr_2 and its request will be accepted by svr_2."

We also devise an experiment to show how the methodology can be used for fault detection. A system with two FSs (svr_1 and svr_2) and four clients (m_1, m_2, m_3 and m_4) is constructed, where four clients simultaneously request a connection to an FS. Both svr_1 and svr_2 are meant to be designed with a capacity of 2, and we assume that one (say svr_2) by mistakenly implemented with a capacity of 1. This could cause serious problems as one client will iteratively make a request to connect to the system without knowing whether he/she will ever get through.

The erroneous implementation is then translated into a μCRL specification from which we derive its LTS, however, since it has a total of 354 states and 407 transitions it cannot be clearly presented here. As usual we use the toolset CADP to verify the properties.

One way to detect such a problem is to check whether the four clients are successfully connected to the FSs. Since the system is designed with the capacity of 4, all four

clients should have connected to an FS. Thus, for each client, we define the following property:

[true*. "cmd(m_i, connecting)" *]
(<true* "client_info(m_i, connected, svr_1))"> or
<true* "client_info(m_i, connected, svr_2))">) true

stating "when client m_i sends connecting request to the system, its request should be either accepted by $sver\_1$ or by $sver\_2$". Using these properties, the CADP model checker can correctly distinguish the correct and faulty implementations based upon the design we wish to check against.

5.2 State space investigation

In addition to system wide property checking, we were interested in whether the integrated system had a tractable state space as the size of its components grew, thus we also investigated the state space generated from the μCRL specification by using the toolset CADP.

Clients	States	Transitions
1	39	40
2	413	456
3	4381	5055
4	4845	5681
5	5309	6307

Table 1. One FS with capacity of 5.

Clients	States	Transitions
1	49	50
2	867	932
3	12307	14073
4	13449	15917
5	14591	17761

Table 2. Three FSs with capacity of 1,2 and 3 respectively.

We first construct a system where only one FS is applied. The FS' capacity is set to 5. A number of clients simultaneously request a phone connection. Before the simulation starts, all clients have preset enough money in their account. We incrementally increase the number of clients from 1 to 5, and Table 1 illustrates the changes of the state space that result. It can be seen that when the second and third client are connected to the FS, the state space increases rapidly: by a factor of almost 10. However, after this the subsequent increases level off, and the size is increased by roughly 20% when one new client is added to the system.

The same phenomenon is noticed as well when we apply three FSs to the system. The server capacities are set to 1, 2 and 3 respectively, and the resultant state space is shown in Table 2. This seems to suggest that, with the number of clients being increased, the state space will fairly reach a saturated point where the state space is slowly increased by a stable pace. We are currently investigating whether this is a general phenomenon or one peculiar to this particular example.

6 Conclusions and future work

Verification is an important process in the development of Erlang applications. This paper contributes to the recent strand of work which has studied the verification of Erlang

applications using the process algebra μCRL. The basic methodology in this approach is for an Erlang application to be translated (abstracted) into a μCRL specification, upon which the standard model checker CADP can be applied.

The study of how best to translate Erlang into μCRL contains many open research issues and is still in its early stage. Recent results had shown how components such as *supervisor*, *gen_server*, *gen_fsm* and the Erlang *timeout* event could be translated into μCRL, but the translation rules for each component had been evaluated independently. At FORTE'07 we defined the rules for translating *gen_fsm* into μCRL, and evaluated the rules with two case studies. This was extended in [16] by defining the rules for coping with Erlang *timeout* events and evaluated the work with some case studies. The experimental results show quite a promising effect for system verification.

However, no work had investigated the translation rules in an application where the *gen_server*, *supervisor*, *gen_fsm* and *timeout* events were incorporated in an integrated system. This forces us to face a challenge. If we apply all rules in an integrated way, will these rules show the similar effects for system verification as they independently demonstrated? Moreover, will a state space explosion mean that effective verification is lost? These questions are important to us since we want to make sure (or at least be confident) that all the defined rules can work in an integrated way, or that this requirement could be achieved through modifying some rules before we looked at the translation of other OTP components (such as *applications*).

In this paper, we have attempted to look at these questions by investigating the verification of an Erlang telecommunication system in μCRL. The system integrates the use of the *supervisor*, *gen_server*, *gen_fsm* components and uses explicit *timeout* events. We have shown how the system is translated into a μCRL specification using the proposed translation rules, and verified a number of system properties by using CADP and investigated the changes of state space when the number of clients increases. In our experiments, a property under verification (PUV) is defined in a way where the behaviour of the functional servers (FSs), the behavior of the clients and the communication between the FSs and the clients should be verified simultaneously. Thus, each PUV looks at a property in the view of complete system. A faulty implementation was also used to test the capability of fault detection, and based upon the design the faulty implementation was correctly distinguished by CADP.

The experimental results suggest that when being applied in an integrated system, the translation rules show the similar effect for system verification as being applied independently. The study of the changes in state space suggests, with the number of clients being increased, the state space is slowly increased by a stable pace. All experimental evidence gives us confidence that we are working in the correct direction, and thus we can continue with the study of some other OTP components.

There remains much to be done. Work continues on the automation of the translation of additional OTP components, as does work on verifying the correctness of the translation against the Erlang semantics [10,11]. This latter aspect remains a challenging task, for the full semantics for distributed nodes in an Erlang application can have, semantically, some very subtle behaviour, as discussed in, for example, [18].

Acknowledgements

This work is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/C525000/1. We would like to thank the developers of the tool sets of μCRL and CADP for permitting the use of tools for system verification.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. T. Arts, C. Benac-Earle, and J. Derrick. Verifying Erlang code: a resource locker case-study. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Formal Methods Europe: Getting IT Right, Copenhagen, Denmark*, volume 2391 of *LNCS*, pages 184–203. Springer-Verlag, July 2002.
3. T. Arts, C. Benac-Earle, and Juan José Sánchez Penas. Translating Erlang to μCRL. In *The Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 135–144. IEEE Computer Society, June 2004.
4. J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
5. C. Benac-Earle. *Model checking the interaction of Erlang components*. PhD thesis, The University of Kent, Canterbury, Department of Computer Science, 2006.
6. C. Benac-Earle and Lars-Åke Fredlund. Verification of Language Based Fault-Tolerance. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *EUROCAST*, pages 140–149. Springer-Verlag, February 2005.
7. C. Benac-Earle, Lars-Åke Fredlund, and J. Derrick. Verifying Fault-Tolerant Erlang Programs. In K. Sagonas and J. Armstrong, editors, *Proceedings of ACM SigPlan Erlang 2005 Workshop*, pages 26–34. ACM Press, September 2005.
8. CADP. *http://www.inrialpes.fr/vasy/cadp/*.
9. E. Clarke, O. Grumberg, and D. Long. *Model Checking*. MIT Press, 1999.
10. Lars-Åke Fredlund. Towards a sematics for Erlang. In *Foundatins of Mobile Computation: A Post-Conference Satellite Workshop of FST and TCS 99*, 1999.
11. Lars-Åke Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Roral Institute of Technology, Stockholm, Sweden, 2001.
12. Lars-Åke Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer*, 4(4):405–420, August 2003.
13. J. F. Groote and A. Ponse. The syntax and sematics of μCRL. In *Algebra of Communicating Processes 1994, Workshop in Computing*, pages 26–62, 1995.
14. Q. Guo. Verifying Erlang/OTP Components in μCRL. In John Derrick and Jüri Vain, editors, *FORTE'07*, volume 4574 of *LNCS*, pages 227–246. Springer-Verlag, June 2007.
15. Q. Guo and J. Derrick. Eliminating overlapping of pattern matching when verifying Erlang programs in μCRL. In *12th International Erlang User Conference (EUC'06), Stockholm, Sweden*, 2006.
16. Q. Guo and J. Derrick. Verification of Timed Erlang/OTP Components Using the Process Algebra μCRL. In Simon Thompson and Lars-Ake Fredlund, editors, *6th ACM SIGPLAN Erlang Workshop*, pages 55–64. ACM Press, October 2007.
17. F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999.
18. H. Svensson and Lars-Åke Fredlund. A More Accurate Semantics for Distributed Erlang. In Simon Thompson and Lars-Ake Fredlund, editors, *6th ACM SIGPLAN Erlang Workshop*, pages 43–54. ACM Press, October 2007.