Verification of Timed Erlang/OTP Components Using the Process Algebra μCRL

Qiang Guo and **John Derrick**
Department of Computer Science,
The University of Sheffield,
Regent Court, 211 Portobello Street, S1 4DP, UK
{Q.Guo, J.Derrick}@dcs.shef.ac.uk

October 05, 2007

Sixth ACM SIGPLAN Erlang Workshop, Freiburg, Germany, October 05, 2007

Motivation

- Erlang is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems.
- Although Erlang has many high-level features, verification can still be non-trivial. One approach is to perform an abstraction of an Erlang program into the process algebra μCRL, upon which standard tools such as CADP can be applied.
- This approach has recently been applied to the verification of Erlang programs and OTP components *supervisor*, *generic server* and *finite stat machine (FSM)*.
- No existing work has defined rules for the translation of *timeout* events into μCRL. This could dramatically degrade the usability of the existing work as in some real applications, *timeout* events play a significant role in the system development.
- In this paper, by extending the existing work, we investigated the verification of timed Erlang/OTP components in μCRL.

Outline

- Introduction of the process algebra μCRL;
- A review on the related work;
- Translating timed Erlang/OTP components into μCRL;
- Two case studies;
- Conclusions and future work.

The Process Algebra μCRL

- The process algebra μCRL (micro Common Representation Language) is extended from the process algebra ACP where equational *abstract data types* are integrated into process specification.
- A μCRL specification is comprised of two parts: the data types and the processes.
- Processes are declared using the keyword *proc*.
- A process may contain actions representing elementary activities that can be performed. These actions must be explicitly declared using the keyword *act*.
- Data types are specified as the standard abstract data types, using sorts, functions and axioms. Sorts are declared using the key work *sort*, functions are declared using the keyword *func* and *map* is reserved for additional functions. Axioms are declared using the keyword *rew*, referring to the possibility to use rewriting technology for evaluation of terms.

Related Work - Translating Erlang Syntax

- Arts *et al.* initiated the studies of translating Erlang into μCRL. Benac-Earle developed a toolset *etomcrl* to automate the process of translation.
- Rules for the translation of pattern matching, communication, functions with side-effects, higher-order functions, generic server and supervision tree have been defined respectively;

Related Work - Overlapping in Pattern Matching

- The toolset *etomcrl* translating the pattern matching in a way where overlapping patterns are not considered.
- In Erlang, evaluation of pattern matching works from top to bottom and from left to right.
- The μCRL toolset instantiator does not evaluate rewriting rules in a fixed order.
- If there exists overlapping between patterns, the problem of overlapping in pattern matching occurs, which could lead to the system being represented by a faulty model.
- Benac-Earle discussed a solution to cope with the problem, but did not apply the technique in *etomcrl*.
- Guo *et al.* proposed a different solution, whereby an Erlang program with overlapping patterns is transformed into a counterpart program without overlapping patterns.
- A data structure called the Structure Splitting Tree (SST) is defined and applied for pattern evaluation, and its use guarantees that no overlapping patterns will be introduced to the transformed program.

Related Work - Translating Finite State Machine

- Guo *et al.* studied the translation of the Erlang finite state machine (FSM) design pattern.
- By extending the approach discussed by Arts *et al.*, a model was proposed to support the translation of an Erlang FSM component into μCRL.
- The translation of the Erlang *gen_fsm* module into μCRL is comprised of two parts, *simulating state management* and *translating the state functions*.

Related Work - Translating FSM::Simulating State Management

- A (one place) stack is used in the μCRL specification to simulate the management of FSM states and state data. Actions s_event, r_event and $send\_event$, are defined to save and read data from the stack; $s\_event \mid r\_event = send\_event$.
- The process $write$ is defined to push a data onto the stack while the process $read$ to pop from the stack.
- An Erlang FSM state S_1 is assigned with a μCRL state name s_S_1 ("s_" plus the state name) and a state process fsm_S_1 ("fsm_" plus the state name).
- The *current state* and the *state data* are coded in a tuple with the form of $tuple(state, tuplenil(state\_data))$ and saved in the stack.
- The process $fsm\_init(State{:}Term, Data{:}Term)$ initially pushes $tuple(Init\_State, tuplenil(State\_Data))$ onto the stack. The process $fsm\_next\_state(State{:}Term, Data{:}Term)$ updates the *current state* and the *state data* in the stack.
- The process fsm_next_state receives a command through $r\_command$. When a command is received, the process fsm_state_change is proceeded where the *current state* and the *state data* are read out from the stack. The *current state* determines which state process is about to be activated.

act
s_event, r_event, send_event: Term

comm
s_event | r_event = send_event

proc
write(Val:Term) =
wcallresult (Val)

read(Cmd:Term)=
sum(Val:Term, recallresult(Val).
fsm_S_1(Cmd,element (2,Val))
$\lhd$ is_s_S_1(element(1,Val)) $\rhd$
fsm_S_2(Cmd,element (2,Val))
$\lhd$ is_s_S_2(element(1,Val)) $\rhd$
...
fsm_S_n(Cmd,element(2,Val))
$\lhd$ is_s_S_n(element(1,Val))$\rhd$
delta)

fsm_change_state =
sum(Cmd:Term,
r_event(Cmd).read(Cmd))

fsm_init(S:Term,Data:Term) =
fsm_next_state(S,Data)

fsm_next_state(S:Term,Data:Term) =
wcallresult(tuple(S,tuplenil(Data))).
sum(Cmd:Term,r_command(Cmd).
s_event(Cmd).fsm_change_state)

fsm_S_1(Cmd:Term, Data:Term) =
pre_defined actions ...
fsm_next_state(next_State, new_data)
....
fsm_S_n(Cmd:Term, Data:Term) =
pre_defined actions ...
fsm_next_state(next_State,new_data)

Figure 1: Rules for translating state processes.

```
act
  s_event, r_event, send_event: Term

comm
  s_event | r_event = send_event

proc
  write(Val:Term) =
    wcallresult (Val)

  read(Cmd:Term)=
    sum(Val:Term, recallresult(Val).
      fsm_S1(Cmd,element (2,Val))
        ◁ is_s_S1(element(1,Val)) ▷
          fsm_S2(Cmd,element (2,Val))
            ◁ is_s_S2(element(1,Val)) ▷
                ...
                  fsm_Sn(Cmd,element(2,Val))
                    ◁ is_s_Sn(element(1,Val))▷
                                  delta)

  fsm_change_state =
    sum(Cmd:Term,
      r_event(Cmd).read(Cmd))

  fsm_init(S:Term,Data:Term) =
    fsm_next_state(S,Data)

  fsm_next_state(S:Term,Data:Term) =
    wcallresult(tuple(S,tuplenil(Data))).
      sum(Cmd:Term,r_command(Cmd).
            s_event(Cmd).fsm_change_state)

  fsm_S1(Cmd:Term, Data:Term) =
    pre_defined actions ...
      fsm_next_state(next_State, new_data)
      ....
  fsm_Sn(Cmd:Term, Data:Term) =
    pre_defined actions ...
      fsm_next_state(next_State,new_data)
```

Figure 2: Rules for translating state processes.

act
s_event, r_event, send_event: Term

comm
s_event | r_event = send_event

proc
write(Val:Term) =
wcallresult (Val)

read(Cmd:Term)=
sum(Val:Term, recallresult(Val).
fsm_S_1(Cmd,element (2,Val))
$\lhd$ is_s_S_1(element(1,Val)) $\rhd$
fsm_S_2(Cmd,element (2,Val))
$\lhd$ is_s_S_2(element(1,Val)) $\rhd$
...
fsm_S_n(Cmd,element(2,Val))
$\lhd$ is_s_S_n(element(1,Val))$\rhd$
delta)

fsm_change_state =
sum(Cmd:Term,
r_event(Cmd).read(Cmd))

fsm_init(S:Term,Data:Term) =
fsm_next_state(S,Data)

fsm_next_state(S:Term,Data:Term) =
wcallresult(tuple(S,tuplenil(Data))).
sum(Cmd:Term,r_command(Cmd).
s_event(Cmd).fsm_change_state)

fsm_S_1(Cmd:Term, Data:Term) =
pre_defined actions ...
fsm_next_state(next_State, new_data)
....
fsm_S_n(Cmd:Term, Data:Term) =
pre_defined actions ...
fsm_next_state(next_State,new_data)

Figure 3: Rules for translating state processes.

```
act
  s_event, r_event, send_event: Term

comm
  s_event | r_event = send_event

proc
  write(Val:Term) =
    wcallresult (Val)

  read(Cmd:Term)=
    sum(Val:Term, recallresult(Val).
      fsm_S1(Cmd,element (2,Val))
        ◁ is_s_S1(element(1,Val)) ▷
          fsm_S2(Cmd,element (2,Val))
            ◁ is_s_S2(element(1,Val)) ▷
                ...
                  fsm_Sn(Cmd,element(2,Val))
                    ◁ is_s_Sn(element(1,Val))▷
                                  delta)

  fsm_change_state =
    sum(Cmd:Term,
      r_event(Cmd).read(Cmd))

  fsm_init(S:Term,Data:Term) =
    fsm_next_state(S,Data)

  fsm_next_state(S:Term,Data:Term) =
    wcallresult(tuple(S,tuplenil(Data))).
      sum(Cmd:Term,r_command(Cmd).
            s_event(Cmd).fsm_change_state)

  fsm_S1(Cmd:Term, Data:Term) =
    pre_defined actions ...
      fsm_next_state(next_State, new_data)
      ....
  fsm_Sn(Cmd:Term, Data:Term) =
    pre_defined actions ...
      fsm_next_state(next_State,new_data)
```

Figure 4: Rules for translating state processes.

```
act
  s_event, r_event, send_event: Term

comm
  s_event | r_event = send_event

proc
  write(Val:Term) =
    wcallresult (Val)

  read(Cmd:Term)=
    sum(Val:Term, recallresult(Val).
      fsm_S1(Cmd,element (2,Val))
        ◁ is_s_S1(element(1,Val)) ▷
          fsm_S2(Cmd,element (2,Val))
            ◁ is_s_S2(element(1,Val)) ▷
                ...
                  fsm_Sn(Cmd,element(2,Val))
                    ◁ is_s_Sn(element(1,Val))▷
                                delta)

  fsm_change_state =
    sum(Cmd:Term,
      r_event(Cmd).read(Cmd))

  fsm_init(S:Term,Data:Term) =
    fsm_next_state(S,Data)

  fsm_next_state(S:Term,Data:Term) =
    wcallresult(tuple(S,tuplenil(Data))).
      sum(Cmd:Term,r_command(Cmd).
            s_event(Cmd).fsm_change_state)

  fsm_S1(Cmd:Term, Data:Term) =
    pre_defined actions ...
      fsm_next_state(next_State, new_data)
      ....
  fsm_Sn(Cmd:Term, Data:Term) =
    pre_defined actions ...
      fsm_next_state(next_State,new_data)
```

Figure 5: Rules for translating state processes.

act
s_event, r_event, send_event: Term

comm
s_event | r_event = send_event

proc
write(Val:Term) =
wcallresult (Val)

read(Cmd:Term)=
sum(Val:Term, recallresult(Val).
fsm_S_1(Cmd,element (2,Val))
$\lhd$ is_s_S_1(element(1,Val)) $\rhd$
fsm_S_2(Cmd,element (2,Val))
$\lhd$ is_s_S_2(element(1,Val)) $\rhd$
...
fsm_S_n(Cmd,element(2,Val))
$\lhd$ is_s_S_n(element(1,Val))$\rhd$
delta)

fsm_change_state =
sum(Cmd:Term,
r_event(Cmd).read(Cmd))

fsm_init(S:Term,Data:Term) =
fsm_next_state(S,Data)

fsm_next_state(S:Term,Data:Term) =
wcallresult(tuple(S,tuplenil(Data))).
sum(Cmd:Term,r_command(Cmd).
s_event(Cmd).fsm_change_state)

fsm_S_1(Cmd:Term, Data:Term) =
pre_defined actions ...
fsm_next_state(next_State, new_data)
....
fsm_S_n(Cmd:Term, Data:Term) =
pre_defined actions ...
fsm_next_state(next_State,new_data)

Figure 6: Rules for translating state processes.

Related Work - Translating FSM::Translating State Functions

```
S_i(N) →
  case N of
    P_1 →
      actions(1);
    P_2 →
      actions(2);
        ...
    P_n →
      actions(n)
  end.

S_i(N) when N is of P_1 →
      actions(1);
S_i(N) when N is of P_2 →
      actions(2);
    ...
S_i(N) when N is of P_n →
      actions(n).
```

Figure 7: Guarded Erlang programs.

- The translation of an Erlang state function into μCRL starts by splitting the function into two parts, one of which defines a series of μCRL state functions while the other a set of action sequences.
- Every set of action sequences is translated into a pre-defined action set in μCRL. According to the order that patterns occur in the Erlang function, the pre-defined action sets are uniquely indexed with a set of integers.
- For example, in Figure 7, the set of action sequences $\{actions(1)$, ..., $actions(n)\}$ is indexed with an integer set {1, ..., n} where integer i identifies the pre-defined action set $actions(i)$.

- The selection of a μCRL state function for execution is determined by the pattern of function arguments.
- By the end, the μCRL function returns a tuple with the form of $tuple(next\_state, tuple(new\_data, tuplenil(index)))$ where $next\_state$ returns the next state, new_data the updated state data and $index$ the index of the action sequence that needs to be performed.
- To eliminate any potential overlapping between patterns, techniques proposed by Guo *et al.* are applied.
- Figure 8 illustrates the translation rules for the Erlang state function shown in Figure 7.

rew

S_i(Args) =
 S_i_case_0(patterns_match(Args,P_1),Args)
S_i_case_0(true,Args) =
  tuple(S_j,tuple(Data,tuplenil(1)))
S_i_case_0(false,Args) =
  S_i_case_1(patterns_match(Args,P_2),Args)
S_i_case_1(true,Args) =
  tuple(S_k,tuple(Data,tuplenil(2)))
  ...
S_i_case_(n-1)(true,Args) =
  tuple(S_u,tuple(Data,tuplenil(n-1)))
S_i_case_(n-1)(false,Args) =
  S_i_case_n(patterns_match(Args,P_n),Args)
S_i_case_n(true,Args) =
  tuple(S_v,tuple(Data,tuplenil(n)))

proc

fsm_S_i(Cmd:Term,Data:Term) =
 actions(1).
 fsm_next_state(element(1,S_i(Cmd,Data)),
      element(2,S_i(Cmd,Data)))
  $\triangleleft$ element(3,S_i(Cmd,Data))=1 $\triangleright$
   (actions(2).
    fsm_next_state(element(1,S_i(Cmd,
      Data)),element(2,S_i(Cmd,Data)))
     $\triangleleft$ element(3,S_i(Cmd,Data))=2 $\triangleright$
       ...
   (actions(n).
    fsm_next_state(element(1,S_i(Cmd,
      Data)),element(2,S_i(Cmd,Data)))
     $\triangleleft$ element(3,S_i(Cmd,Data))=n $\triangleright$
                                  delta)...)

Figure 8: Translation rules for Erlang state function.

rew

S_i(Args) =
 S_i_case_0(patterns_match(Args,P_1),Args)
S_i_case_0(true,Args) =
  tuple(S_j,tuple(Data,tuplenil(1)))
S_i_case_0(false,Args) =
  S_i_case_1(patterns_match(Args,P_2),Args)
S_i_case_1(true,Args) =
  tuple(S_k,tuple(Data,tuplenil(2)))
  ...
S_i_case_(n-1)(true,Args) =
  tuple(S_u,tuple(Data,tuplenil(n-1)))
S_i_case_(n-1)(false,Args) =
  S_i_case_n(patterns_match(Args,P_n),Args)
S_i_case_n(true,Args) =
  tuple(S_v,tuple(Data,tuplenil(n)))

proc

fsm_S_i(Cmd:Term,Data:Term) =
 actions(1).
 fsm_next_state(element(1,S_i(Cmd,Data)),
      element(2,S_i(Cmd,Data)))
  $\triangleleft$ element(3,S_i(Cmd,Data))=1 $\triangleright$
   (actions(2).
    fsm_next_state(element(1,S_i(Cmd,
      Data)),element(2,S_i(Cmd,Data)))
     $\triangleleft$ element(3,S_i(Cmd,Data))=2 $\triangleright$
       ...
    (actions(n).
     fsm_next_state(element(1,S_i(Cmd,
      Data)),element(2,S_i(Cmd,Data)))
      $\triangleleft$ element(3,S_i(Cmd,Data))=n $\triangleright$
                                  delta)...)

Figure 9: Translation rules for Erlang state function.

```
rew
 S_i(Args) =
  S_i_case_0(patterns_match(Args,P_1),Args)
 S_i_case_0(true,Args) =
    tuple(S_j,tuple(Data,tuplenil(1)))
 S_i_case_0(false,Args) =
    S_i_case_1(patterns_match(Args,P_2),Args)
 S_i_case_1(true,Args) =
    tuple(S_k,tuple(Data,tuplenil(2)))
     ...
 S_i_case_(n-1)(true,Args) =
    tuple(S_u,tuple(Data,tuplenil(n-1)))
 S_i_case_(n-1)(false,Args) =
    S_i_case_n(patterns_match(Args,P_n),Args)
 S_i_case_n(true,Args) =
    tuple(S_v,tuple(Data,tuplenil(n)))
```

```
proc
 fsm_S_i(Cmd:Term,Data:Term) =
  actions(1).
  fsm_next_state(element(1,S_i(Cmd,Data)),
        element(2,S_i(Cmd,Data)))
   ◁ element(3,S_i(Cmd,Data))=1 ▷
    (actions(2).
     fsm_next_state(element(1,S_i(Cmd,
          Data)),element(2,S_i(Cmd,Data)))
      ◁ element(3,S_i(Cmd,Data))=2 ▷
           ...
     (actions(n).
      fsm_next_state(element(1,S_i(Cmd,
        Data)),element(2,S_i(Cmd,Data)))
       ◁ element(3,S_i(Cmd,Data))=n ▷
                                  delta)...)
```

Figure 10: Translation rules for Erlang state function.

```
rew
 S_i(Args) =
  S_i_case_0(patterns_match(Args,P_1),Args)
 S_i_case_0(true,Args) =
    tuple(S_j,tuple(Data,tuplenil(1)))
 S_i_case_0(false,Args) =
    S_i_case_1(patterns_match(Args,P_2),Args)
 S_i_case_1(true,Args) =
    tuple(S_k,tuple(Data,tuplenil(2)))
     ...
 S_i_case_(n-1)(true,Args) =
    tuple(S_u,tuple(Data,tuplenil(n-1)))
 S_i_case_(n-1)(false,Args) =
    S_i_case_n(patterns_match(Args,P_n),Args)
 S_i_case_n(true,Args) =
    tuple(S_v,tuple(Data,tuplenil(n)))
```

```
proc
  fsm_S_i(Cmd:Term,Data:Term) =
   actions(1).
   fsm_next_state(element(1,S_i(Cmd,Data)),
          element(2,S_i(Cmd,Data)))
    ◁ element(3,S_i(Cmd,Data))=1 ▷
     (actions(2).
      fsm_next_state(element(1,S_i(Cmd,
           Data)),element(2,S_i(Cmd,Data)))
       ◁ element(3,S_i(Cmd,Data))=2 ▷
            ...
      (actions(n).
       fsm_next_state(element(1,S_i(Cmd,
          Data)),element(2,S_i(Cmd,Data)))
        ◁ element(3,S_i(Cmd,Data))=n ▷
                                  delta)...)
```

Figure 11: Translation rules for Erlang state function.

Related Work - Model Checking Erlang in μCRL

- Once an Erlang program is translated into a μCRL specification, one can check the system properties by using some existing tools such as CADP.
- Properties one wishes to check with the CADP model checker are formalized in the regular alternation-free μ-calculus (a fragment of the modal μ-calculus), a first-order logic with modalities, and least and greatest fixed point operators.
- Automation for property checking can be achieved by using the Script Verification Language (SVL). SVL provides a high-level interface to all CADP tools, which enables an easy description and execution of complex performance studies.

Translating Timed Erlang/OTP Components

- All existing work so far translates Erlang programs without taking *timeout* events into account.
- This could degrade the usability of the existing work.
- In some real applications, *timeout* events play a significant role in the OTP design.
- This paper studies the translation of *timeout* events into μCRL and defines rules to support the translation of timed Erlang functions into μCRL.

Translating Timed Components - Definition of a Timer in μCRL

- Two approaches might be considered to extend the existing work for coping with *timeout* events; (I) use a timed extension to μCRL; (II) define a *timer* in the untimed μCRL.
- A timed version of μCRL was defined in by Groote *et al.* but has limitations in the applications.
- This work considers the second approach. A *timer* and an explicit timing action *tick* are incorporated in the μCRL specification.
- A *timer*, has two states, *on* and *off*. The μCRL function *set(t:Timer, x:Natural)* instantiates a timer with an integer while the μCRL function *expired(t:Timer)* evaluates whether the time-up occurs.
- A timer periodically calls the μCRL function *pred(t:Timer)* to count down the value of timing. To do so, the μCRL function *pred(x:Natural)* with an integer as its argument needs to be defined before the definition of *timer*.
- The μCRL function *pred(x:Natural)* determines the time elapse unit for a *timer*, namely, *pred(x)* $=$ *x - time_elapse_unit*. For example, if the time elapse unit is defined as 1, then *pred(3)* $=$ *2* and *pred(2)* $=$ *1*.

- Figure 12 illustrates the μCRL syntax.

sort
Timer
func
off:⟶ Timer
on: Natural ⟶ Timer
map
pred: Timer ⟶ Timer
expired: Timer ⟶ bool
set: Timer # Natural ⟶ Timer
reset: Timer ⟶ Timer

var
t:Timer
n:Natural
rew
expire(off)=F
expire(on(n))=eq(0,n)
pred(on(n))=on(pred(n))
pred(off)=off
set(t,n)=on(n)
reset(t)=off

Figure 12: The syntax of a μCRL timer

Translating Timed Components - Translating Timed Functions

- To incorporate *timeout* events in the translation, for those Erlang functions with timing restrictions, two processes are defined in μCRL, one of which deals with *timing*; the other copes with *count down*.
- When the timing begins, the *timing* process will be called. The *timing* process will either call another process or activate the *count down* process (counts down the timer by one unit).
- When going through the *count down* process, the timer is evaluated.
- If the timer does not expire, a *tick* action will be performed once, stating the passing of one time elapse unit. Afterwards, the *timing* process is called;
- otherwise, a *timeout* event will be generated and the corresponding *timeout* process is enabled to process the event.

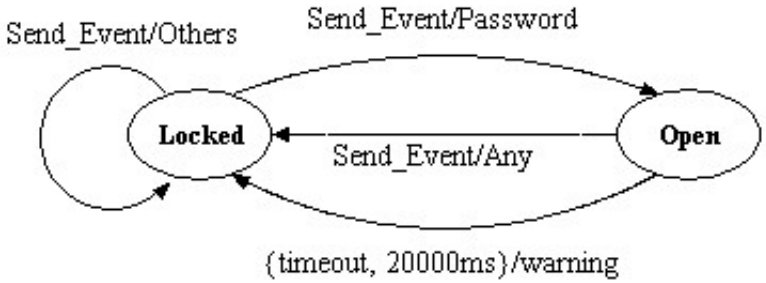

Figure 13: FSM - Door with Code Lock:Design.

- Figure 13 and 14 illustrates an example - A door with code lock. The door consists of two states, *locked* and *open*, and a system code for opening the door.
- Initially, the door is set to *locked* while the code is set to a word.
- The door switches between states, driven by an external event.
- A timing restriction is applied for the system. If the door is switched to the *open* state and no action is performed within a defined period, a *timeout* event is generated, which activates the *timeout* function to close the door.

```
-module(fsm_door).
-export([start_link/1, button/1, init/ 1]).
-export([locked/2, open/ 2]).

  start_link(Code) →
     gen_fsm:start_link(local, fsm_door,
                        fsm_door, Code,[]).

  init(Code) →
     {ok, locked, Code}.

  button(Password) →
     gen_fsm:send_event(fsm_door,
              {button, Password}).
```

```
locked({button, Password}, Code) →
  case Password of
        Code →
           action:do_unlock(),
           {next_state, open, Code, 20000};
        _Wrong →
           action:display_message(),
           {next_state, locked, Code}.

open({button, Password}, Code) →
  action:do_lock(),
  {next_state, locked,Code};
open(timeout, Code) →
  action:display_message(),
  action:do_lock(),
  {next_state, locked, Code}.
```

Figure 14: The Erlang code for a door with code lock.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
     do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
               warning_message .
               fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
     count_down_locked(Cmd,Data,t) +
     fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
     tick . locked_timing(Cmd,Data,pred(t))
        ◁ not(expired(t)) ▷
               time_out . warning_message .
               fsm_next_state(s_locked,tuplenil(abc))
```

Figure 15: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
     do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
               warning_message .
               fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
     count_down_locked(Cmd,Data,t) +
     fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
     tick . locked_timing(Cmd,Data,pred(t))
        ◁ not(expired(t)) ▷
               time_out . warning_message .
               fsm_next_state(s_locked,tuplenil(abc))
```

Figure 16: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
     do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
               warning_message .
               fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
     count_down_locked(Cmd,Data,t) +
     fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
     tick . locked_timing(Cmd,Data,pred(t))
        ◁ not(expired(t)) ▷
               time_out . warning_message .
               fsm_next_state(s_locked,tuplenil(abc))
```

Figure 17: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
      do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
                warning_message .
                fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
      count_down_locked(Cmd,Data,t) +
      fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
      tick . locked_timing(Cmd,Data,pred(t))
          ◁ not(expired(t)) ▷
                time_out . warning_message .
                fsm_next_state(s_locked,tuplenil(abc))
```

Figure 18: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
     do_unlock . locked_timing(Cmd,Data, on(x))
         ◁ term_to_bool(equal(element(int(1),element(int(3),
                                  locked(Cmd,Data))),int(1))) ▷
               warning_message .
               fsm_next_state(element(int(1),locked(Cmd,Data)),
                                  element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
     count_down_locked(Cmd,Data,t) +
     fsm_next_state(element(int(1),locked(Cmd,Data)),
                           element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
     tick . locked_timing(Cmd,Data,pred(t))
        ◁ not(expired(t)) ▷
               time_out . warning_message .
               fsm_next_state(s_locked,tuplenil(abc))
```

Figure 19: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
      do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
                warning_message .
                fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
      count_down_locked(Cmd,Data,t) +
      fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
      tick . locked_timing(Cmd,Data,pred(t))
          ◁ not(expired(t)) ▷
                time_out . warning_message .
                fsm_next_state(s_locked,tuplenil(abc))
```

Figure 20: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
     do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
               warning_message .
               fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
     count_down_locked(Cmd,Data,t) +
     fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
     tick . locked_timing(Cmd,Data,pred(t))
        ◁ not(expired(t)) ▷
               time_out . warning_message .
               fsm_next_state(s_locked,tuplenil(abc))
```

Figure 21: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
     do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
               warning_message .
               fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
     count_down_locked(Cmd,Data,t) +
     fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
     tick . locked_timing(Cmd,Data,pred(t))
        ◁ not(expired(t)) ▷
               time_out . warning_message .
               fsm_next_state(s_locked,tuplenil(abc))
```

Figure 22: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
      do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
                warning_message .
                fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
      count_down_locked(Cmd,Data,t) +
      fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
      tick . locked_timing(Cmd,Data,pred(t))
          ◁ not(expired(t)) ▷
                time_out . warning_message .
                fsm_next_state(s_locked,tuplenil(abc))
```

Figure 23: Translating rules for timed Erlang functions.

```
proc
  fsm_locked(Cmd:Term,Data:Term,x:Natural) =
     do_unlock . locked_timing(Cmd,Data, on(x))
          ◁ term_to_bool(equal(element(int(1),element(int(3),
                                    locked(Cmd,Data))),int(1))) ▷
               warning_message .
               fsm_next_state(element(int(1),locked(Cmd,Data)),
                                    element(int(2),locked(Cmd,Data)))

  locked_timing(Cmd:Term,Data:Term,t:Timer) =
     count_down_locked(Cmd,Data,t) +
     fsm_next_state(element(int(1),locked(Cmd,Data)),
                          element(int(2),locked(Cmd,Data)))

  count_down_locked(Cmd:Term,Data:Term,t:Timer)
     tick . locked_timing(Cmd,Data,pred(t))
        ◁ not(expired(t)) ▷
               time_out . warning_message .
               fsm_next_state(s_locked,tuplenil(abc))
```

Figure 24: Translating rules for timed Erlang functions.

Translating Timed Components - Coping with Synchronization

- The *gen_fsm* module defines a function *sync_send_event(FSMRef, Event, Timer)* to provide synchronized communications between FSMs
- When *Timer* is set and an event is sent out, a reply message is expected to arrive within the defined time period; otherwise, a timeout will occur and the Erlang program will be terminated.
- The function *sync_send_event* itself is a timed Erlang function and can be translated by using the defined rules.
- The function is translated into two processes in μCRL. The *sync_send_event* reads a command from the command list and then sends it off through action *s_command*. Afterwards, a timer is instantiated and initialized in the process *sync_send_event_timing*.
- The process expects to received a *reply* through action *sync_read*. If the μCRL FSM process does not return the message within the defined time period, a *timeout* event is generated and the program is terminated.

```
act
   sync_read, sync_send, sync_event : Term
comm
   sync_read | sync_send = sync_event
proc
   sync_send_event(CmdList:Term,x:Natural) =
         s_command(head(CmdList)) .
         sync_send_event_timing(CmdList,x,on(x))

   sync_send_event_timing(CmdList:Term,x:Natural,t:Timer) =
         count_down__sync_send_event(Cmd,x,t) +
         sum(Reply:Term, sync_read(Reply).
                        sync_send_event(tail(CmdList), x))

   count_down_sync_send_event(Cmd:Term,x:Natural,t:Timer)
         tick . sync_send_event_timing(Cmd,x,pred(t))
                  ◁ not(expired(t)) ▷
                           time_out . delta
```

Figure 25: Coping with synchronization communication between FSMs.

Experiment I - A door with Code Lock::Deriving LTS

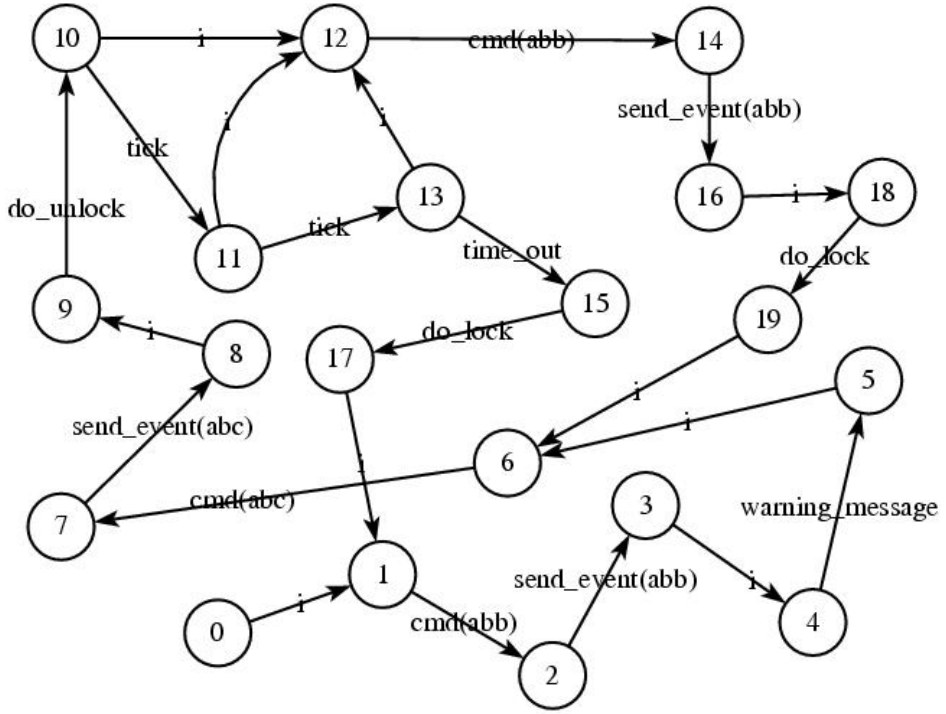

Figure 26: LTS derived from the door with code lock system.

Experiment I - A door with Code Lock::Model Checking

- The system properties should be formalized in the regular alternation-free μ-calculus.
- For example, to check "Without being delayed for 20,000ms (two *tick* actions are performed), *time_out* event cannot be generated", the property can be formulated as:

[true * . "do_unlock" . (not 'tick . tick') . "time_out"] false

- Since this property has been defined in the original design, when applying the CADP model checker, the toolset should return *true* if the Erlang program is correctly implemented (at least in terms of this property).
- To check "When the action *do_unlock* is performed, there exists a transition such that the internal action can still be performed when the time-up occurs", it can be formulated as:

[true * . "do_unlock" *] <'tick . tick . tick' . "i"> true

- The property is not defined the original design, the model checker should return *false*.

Experiment II - Coffee Machine::System Design

- A coffee machine has three states, *selection*, *payment* and *remove*.
- The state *selection* allows a buyer to choose the type of drink, while, the state *payment* displays the price of a selected drink and requires payment for the drink.
- When a buyer selects a drink, if within the defined time period, no action is performed, the machine will reset to the *selection* state.
- When in the state *payment*, if the buyer does not pay enough coins in the defined period, the machine will return all pre-paid coins and reset to the *selection* state; otherwise, the machine goes to the state *remove* where the drink is prepared and the change is returned.
- Four types of drink are sold: *tea*, *cappuccino*, *americano* and *espresso*.
- A buyer can complete the purchase of a drink within the defined time period, or cancel the current transaction to claim back all pre-paid coins.
- The designs of the system is shown in Figure 27. The program initially sets the current state to $selection$. A timing restriction of 20,000ms is set to the state function *payment*.

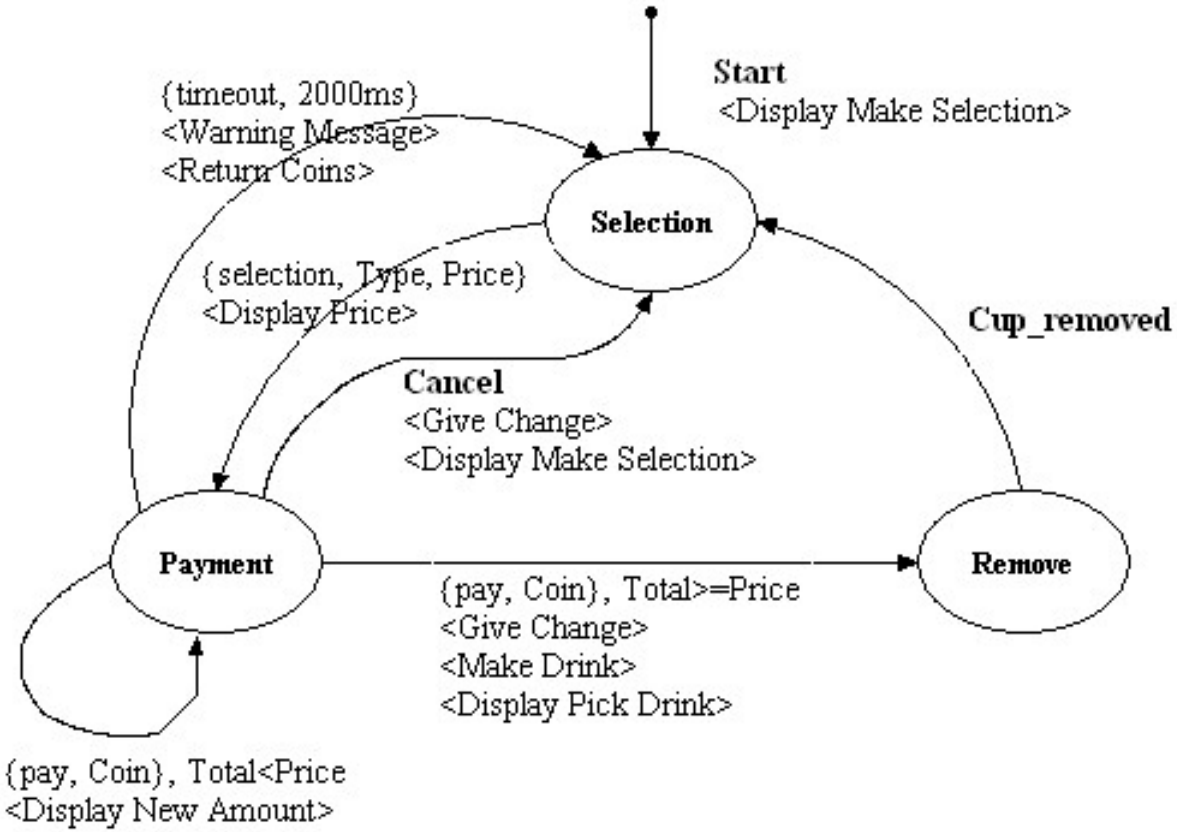

Figure 27: FSM - coffee machine.

Experiment II - Coffee Machine::Deriving LTS

- To initiate the process of buying drinks, two sequences of external actions are constructed.
- The first simulates "selecting *cappuccino* ($\pounds 5$ for a cup), paying $\pounds 4$ and then trying to take the drink away";
- The second simulates "selecting *tea* ($\pounds 4$ for a cup), paying $\pounds 5$ and then taking the drink away".
- The sequences are coded in the lists $[\{selection, cappuccino, 5\}, \{pay, 4\}, \{cup\_remove\}]$ and $[\{selection, tea, 4\}, \{pay, 5\}, \{cup\_remove\}]$, and are initialized in the μCRL specification respectively.

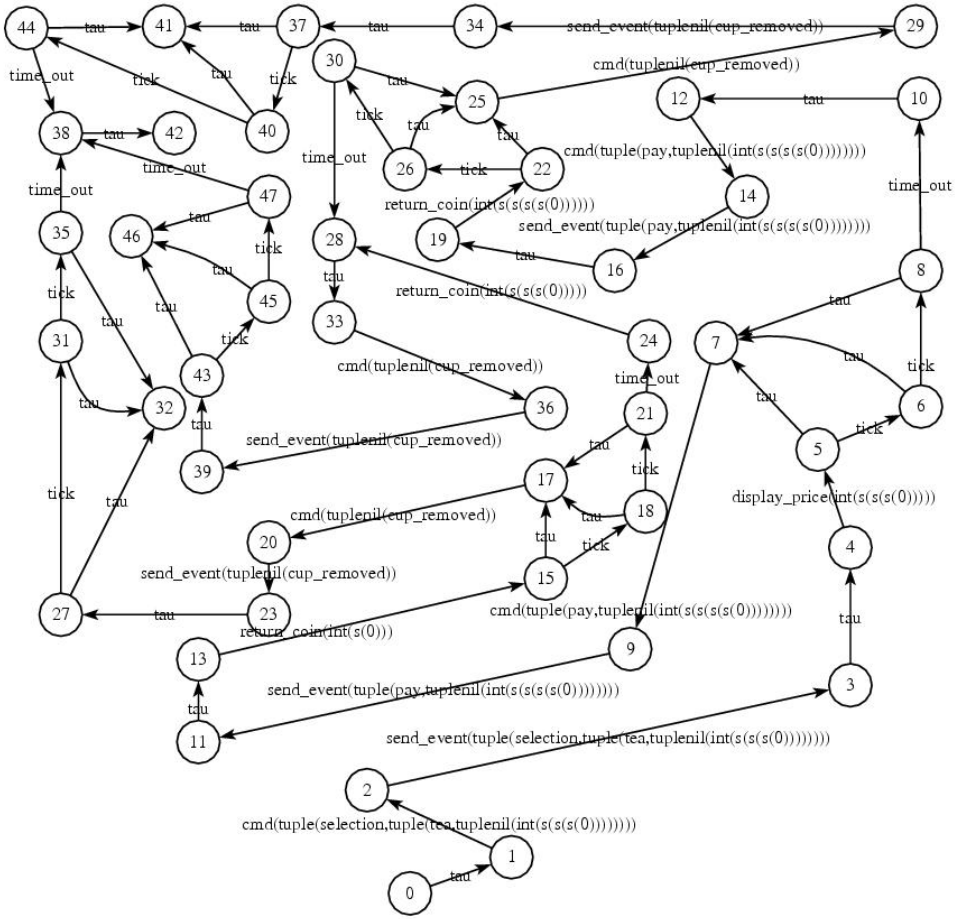

Figure 28: LTS: Buying tea with the payment higher than the price.

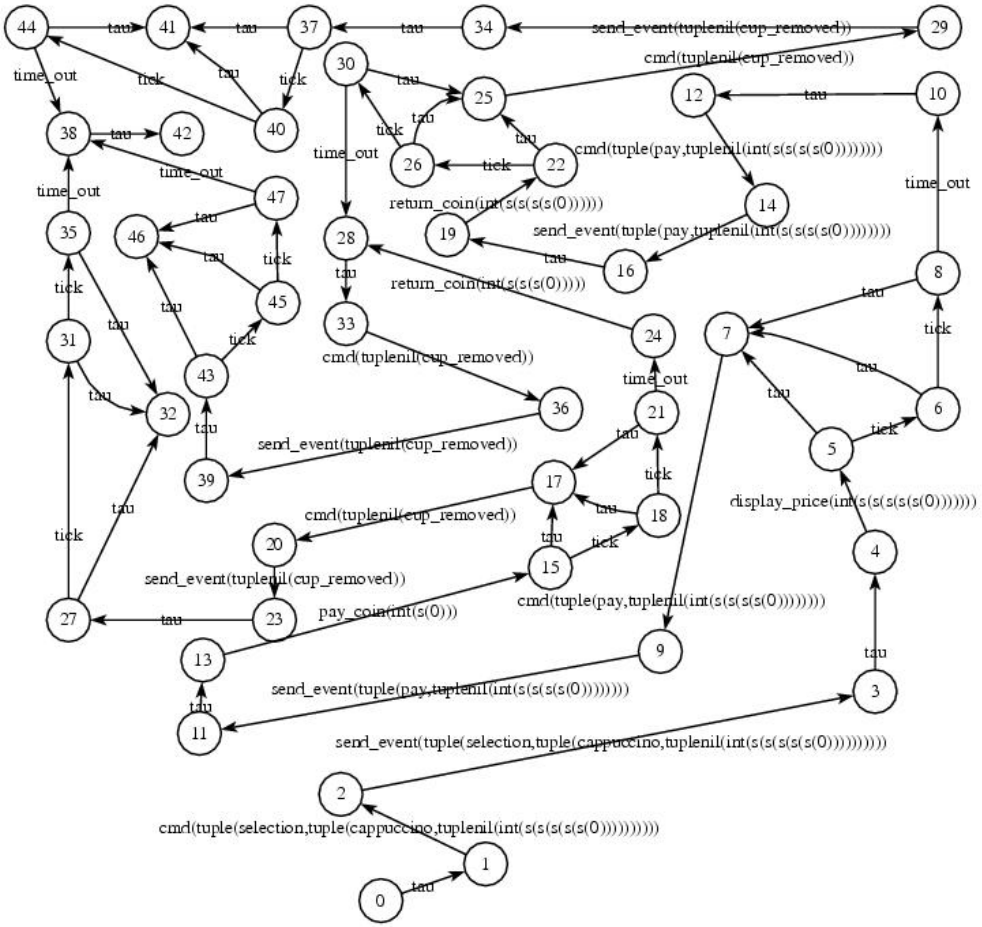

Figure 29: LTS: Buying cappuccino with the payment less than the price.

Experiment II - Coffee Machine::Model Checking

- Model checking using the CADP can then be applied.
- To check "There exists *time_out* events when buying cappuccino and, when the *time_out* is performed, all pre-paid coins should be returned", the property can be formalized as:

<true * . "cmd(tuple(selection,tuple(cappuccino,
tuplenil(int(5)))))" * . (not "time_out") * .
"cmd(tuple(pay,tuplenil(int(4))))" *><true * .
"time_out" *. "return_coin(int(4))"> true

- To check "After a drink is selected and partially paid, without time delay, the pre-paid coins cannot be returned", the property is formalized as:

[(not "time_out") * .
("return_coin(int(4))" or
"return_coin(int(4))")] false

Conclusions and future work

- In this paper, by extending the existing work, we investigated the verification of timed Erlang/OTP components with the process algebra μCRL.
- By using an explicit *tick* event, a discrete-time timing model is defined to support the translation of timed Erlang functions into μCRL.
- We demonstrated the applications of the proposed approach with two small examples. These examples are first modelled by Erlang/OTP FSM with timing restrictions, and then translated into μCRL according to the proposed schema.
- System properties were also verified by using the standard toolset CADP.
- All LTSs presented in this paper were derived through manually translating Erlang FSM programs into a μCRL specification. We are currently upgrading the toolset *etomcrl* where the translation of *timeout* events will be incorporated.

Thank you!

Comments & Questions?