Verification of Timed Erlang/OTP Components Using the Process Algebra $\mu$CRL

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 $\mu$CRL, 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 $\mu$CRL. 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 $\mu$CRL.
Outline

- Introduction of the process algebra $\mu$CRL;
- A review on the related work;
- Translating timed Erlang/OTP components into $\mu$CRL;
- Two case studies;
- Conclusions and future work.
The Process Algebra $\mu$CRL

- The process algebra $\mu$CRL (micro Common Representation Language) is extended from the process algebra ACP where equational abstract data types are integrated into process specification.
- A $\mu$CRL 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 $\mu$CRL. 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 $\mu$CRL 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 \( \mu \)CRL 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 \( \mu \)CRL 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 \( \text{tuple(state, tuplenil(state_data))} \) and saved in the stack.
- The process \( fsm\_init(State:\text{Term}, Data:\text{Term}) \) initially pushes \( \text{tuple(Init\_State, tuplenil(State\_Data))} \) onto the stack. The process \( fsm\_next\_state(State:\text{Term}, Data:\text{Term}) \) updates the current state and the state data in the stack.
- The process \( fsm\_next\_state \) receives a command through \( r\_\text{command} \). When a command is received, the process \( fsm\_\text{state\_change} \) 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.
\begin{figure}[h]
\centering
\begin{verbatim}
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))
    \(\triangleleft\) is_s_S1(element(1,Val)) \(\triangleright\)
    fsm_S2(Cmd,element (2,Val))
    \(\triangleleft\) is_s_S2(element(1,Val)) \(\triangleright\)
    ...
    fsm_Sn(Cmd,element(2,Val))
    \(\triangleleft\) is_s_Sn(element(1,Val)) \(\triangleright\)
    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) =
  fsm_next_state(next_State,new_data)

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

\end{verbatim}
\caption{Rules for translating state processes.}
\end{figure}
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_next_state(next_State,new_data)

pre_defined actions ...
\begin{verbatim}
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))
                              \langle is_s_S_1(element(1, Val)) \rangle
    fsm_S_2(Cmd, element(2, Val))
                              \langle is_s_S_2(element(1, Val)) \rangle
    ...                     \langle is_s_S_n(element(1, Val)) \rangle \rangle
    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)
\end{verbatim}

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

  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_next_state(next_State, new_data)

Figure 4: Rules for translating state processes.
Figure 5: Rules for translating state processes.
\begin{align*}
\text{act} & \quad s_{\text{event}}, r_{\text{event}}, \text{send}_{\text{event}} : \text{Term} \\
\text{comm} & \quad s_{\text{event}} \mid r_{\text{event}} = \text{send}_{\text{event}} \\
\text{proc} & \quad \text{write}(\text{Val} : \text{Term}) = \text{wcallresult}(\text{Val})
\end{align*}

\begin{align*}
& \text{read}(\text{Cmd} : \text{Term}) = \\
& \quad \text{sum}(\text{Val} : \text{Term}, \text{recallresult}(\text{Val}). \\
& \quad \text{fsm}_S(\text{Cmd}, \text{element}(2, \text{Val})) \text{ if } \text{is}_S(\text{element}(1, \text{Val})) \\
& \quad \text{fsm}_S(\text{Cmd}, \text{element}(2, \text{Val})) \text{ if } \text{is}_S(\text{element}(1, \text{Val})) \\
& \quad \ldots \\
& \quad \text{fsm}_S(\text{Cmd}, \text{element}(2, \text{Val})) \text{ if } \text{is}_S(\text{element}(1, \text{Val})) \\
& \quad \text{delta})
\end{align*}

\begin{align*}
\text{fsm}_{\text{change}}_{\text{state}} = \\
& \text{sum}(\text{Cmd} : \text{Term}, \\
& \quad r_{\text{event}}(\text{Cmd}).\text{read}(\text{Cmd}))
\end{align*}

\begin{align*}
\text{fsm}_{\text{init}}(\text{S} : \text{Term}, \text{Data} : \text{Term}) = \\
& \text{fsm}_{\text{next}}_{\text{state}}(\text{S}, \text{Data})
\end{align*}

\begin{align*}
\text{fsm}_{\text{next}}_{\text{state}}(\text{S} : \text{Term}, \text{Data} : \text{Term}) = \\
& \text{wcallresult}(\text{tuple}(\text{S}, \text{tuplenil}(\text{Data}))). \\
& \text{sum}(\text{Cmd} : \text{Term}, r_{\text{command}}(\text{Cmd}). \\
& \quad \text{send}_{\text{event}}(\text{S}, \text{Data})).
\end{align*}

\begin{align*}
\text{fsm}_{\text{S}_1}(\text{Cmd} : \text{Term}, \text{Data} : \text{Term}) = \\
& \text{pre-defined actions} \\
& \text{fsm}_{\text{next}}_{\text{state}}(\text{next}_\text{State}, \text{new}_\text{data}) \\
& \ldots \\
\text{fsm}_{\text{S}_n}(\text{Cmd} : \text{Term}, \text{Data} : \text{Term}) = \\
& \text{pre-defined actions} \\
& \text{fsm}_{\text{next}}_{\text{state}}(\text{next}_\text{State}, \text{new}_\text{data})
\end{align*}

Figure 6: Rules for translating state processes.
Related Work - Translating FSM::Translating State Functions

\[
S_i(N) \rightarrow \\
\text{case } N \text{ of} \quad S_i(N) \text{ when } N \text{ is of } P_1 \rightarrow \\
P_1 \rightarrow \quad \text{actions}(1); \; S_i(N) \text{ when } N \text{ is of } P_2 \rightarrow \\
\text{actions}(2); \quad \ldots \; S_i(N) \text{ when } N \text{ is of } P_n \rightarrow \\
P_n \rightarrow \quad \text{actions}(n) \\
\text{end.}
\]

Figure 7: Guarded Erlang programs.

- The translation of an Erlang state function into \( \mu \text{CRL} \) starts by splitting the function into two parts, one of which defines a series of \( \mu \text{CRL} \) state functions while the other a set of action sequences.
- Every set of action sequences is translated into a pre-defined action set in \( \mu \text{CRL} \). 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 \( \{\text{actions}(1), \ldots, \text{actions}(n)\} \) is indexed with an integer set \( \{1, \ldots, n\} \) where integer \( i \) identifies the pre-defined action set \( \text{actions}(i) \).
The selection of a $\mu$CRL state function for execution is determined by the pattern of function arguments.

By the end, the $\mu$CRL function returns a tuple with the form of $\text{tuple}(\text{next
data}, \text{tuplenil}(\text{index}))$ where $\text{next\ndata}$ returns the next state, $\text{new\ndata}$ the updated state data and $\text{index}$ 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
\[ S_i(\text{Args}) = \]
\[ S_i\_case\_0(\text{patterns}\_\text{match}(\text{Args}, P_1), \text{Args}) \]
\[ S_i\_case\_0(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_j, \text{tuple}(\text{Data}, \text{tuplenil}(1))) \]
\[ S_i\_case\_0(\text{false}, \text{Args}) = \]
\[ S_i\_case\_1(\text{patterns}\_\text{match}(\text{Args}, P_2), \text{Args}) \]
\[ S_i\_case\_1(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_k, \text{tuple}(\text{Data}, \text{tuplenil}(2))) \]
\[ ... \]
\[ S_i\_case\_n(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_u, \text{tuple}(\text{Data}, \text{tuplenil}(n-1))) \]
\[ S_i\_case\_n(\text{false}, \text{Args}) = \]
\[ S_i\_case\_n(\text{patterns}\_\text{match}(\text{Args}, P_n), \text{Args}) \]
\[ S_i\_case\_n(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_v, \text{tuple}(\text{Data}, \text{tuplenil}(n))) \]

proc
\[ \text{fsm}_S_i(\text{Cmd}: \text{Term}, \text{Data}: \text{Term}) = \]
\[ \text{actions}(1). \]
\[ \text{fsm}\_\text{next}\_\text{state}(\text{element}(1, S_i(\text{Cmd}, \text{Data})), \]
\[ \text{element}(2, S_i(\text{Cmd}, \text{Data}))) \]
\[ \langle \text{element}(3, S_i(\text{Cmd}, \text{Data})) = 1 \rangle \]
\[ \text{(actions}(2). \]
\[ \text{fsm}\_\text{next}\_\text{state}(\text{element}(1, S_i(\text{Cmd}, \]
\[ \text{Data})), \text{element}(2, S_i(\text{Cmd}, \text{Data}))) \]
\[ \langle \text{element}(3, S_i(\text{Cmd}, \text{Data})) = 2 \rangle \]
\[ ... \]
\[ \text{(actions}(n). \]
\[ \text{fsm}\_\text{next}\_\text{state}(\text{element}(1, S_i(\text{Cmd}, \]
\[ \text{Data})), \text{element}(2, S_i(\text{Cmd}, \text{Data}))) \]
\[ \langle \text{element}(3, S_i(\text{Cmd}, \text{Data})) = n \rangle \]
\[ \text{delta})... \]

Figure 8: Translation rules for Erlang state function.
rew

\[ S_i(\text{Args}) = \]
\[ S_i\_case\_0(\text{patterns}\_\text{match}(\text{Args}, P_1), \text{Args}) \]
\[ S_i\_case\_0(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_j, \text{tuple}(	ext{Data}, \text{tuplenil}(1))) \]
\[ S_i\_case\_0(\text{false}, \text{Args}) = \]
\[ S_i\_case\_1(\text{patterns}\_\text{match}(\text{Args}, P_2), \text{Args}) \]
\[ S_i\_case\_1(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_k, \text{tuple}(	ext{Data}, \text{tuplenil}(2))) \]
\[ ... \]
\[ S_i\_case\_n(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_u, \text{tuple}(	ext{Data}, \text{tuplenil}(n-1))) \]
\[ S_i\_case\_n(\text{false}, \text{Args}) = \]
\[ S_i\_case\_n(\text{patterns}\_\text{match}(\text{Args}, P_n), \text{Args}) \]
\[ S_i\_case\_n(\text{true}, \text{Args}) = \]
\[ \text{tuple}(S_v, \text{tuple}(	ext{Data}, \text{tuplenil}(n))) \]

Figure 9: Translation rules for Erlang state function.
\[
\begin{align*}
\text{rew} & \quad S_i(\text{Args}) = \\
& \quad S_{i\_\text{case\_0}}(\text{patterns\_match}(\text{Args}, P_1), \text{Args}) \\
& \quad S_{i\_\text{case\_0}}(\text{false}, \text{Args}) = \\
& \quad S_{i\_\text{case\_1}}(\text{patterns\_match}(\text{Args}, P_2), \text{Args}) \\
& \quad S_{i\_\text{case\_1}}(\text{true}, \text{Args}) = \\
& \quad \text{tuple}(S_j, \text{tuple}(\text{Data}, \text{tuplenil}(1))) \\
& \quad S_{i\_\text{case\_0}}(\text{true}, \text{Args}) = \\
& \quad S_{i\_\text{case\_1}}(\text{patterns\_match}(\text{Args}, P_2), \text{Args}) \\
& \quad S_{i\_\text{case\_1}}(\text{true}, \text{Args}) = \\
& \quad \text{tuple}(S_k, \text{tuple}(\text{Data}, \text{tuplenil}(2))) \\
& \quad \ldots \\
& \quad S_{i\_\text{case\_n}}(\text{true}, \text{Args}) = \\
& \quad \text{tuple}(S_v, \text{tuple}(\text{Data}, \text{tuplenil}(n))) \\
\end{align*}
\]

\[
\begin{align*}
\text{proc} & \quad \text{fsm}_S_i(\text{Cmd}: \text{Term}, \text{Data}: \text{Term}) = \\
& \quad \text{actions}(1). \\
& \quad \text{fsm\_next\_state}(\text{element}(1, \text{S}_i(\text{Cmd}, \text{Data})), \\
& \quad \text{element}(2, \text{S}_i(\text{Cmd}, \text{Data}))) \\
& \quad \triangleleft \text{element}(3, \text{S}_i(\text{Cmd}, \text{Data}))=1 \triangleright \\
& \quad (\text{actions}(2). \\
& \quad \text{fsm\_next\_state}(\text{element}(1, \text{S}_i(\text{Cmd}, \\
& \quad \text{Data})), \text{element}(2, \text{S}_i(\text{Cmd}, \text{Data}))) \\
& \quad \triangleleft \text{element}(3, \text{S}_i(\text{Cmd}, \text{Data}))=2 \triangleright \\
& \quad \ldots \\
& \quad (\text{actions}(n). \\
& \quad \text{fsm\_next\_state}(\text{element}(1, \text{S}_i(\text{Cmd}, \\
& \quad \text{Data})), \text{element}(2, \text{S}_i(\text{Cmd}, \text{Data}))) \\
& \quad \triangleleft \text{element}(3, \text{S}_i(\text{Cmd}, \text{Data}))=n \triangleright \\
& \quad \text{delta}) \ldots)
\end{align*}
\]

**Figure 10:** Translation rules for Erlang state function.

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

\[ S_i(\text{Args}) = \]
\[ S_{i \cdot \text{case}_0}(\text{patterns_match}(\text{Args}, P_1), \text{Args}) \]
\[ S_{i \cdot \text{case}_0}(\text{true, Args}) = \]
\[ \text{tuple}(S_j, \text{tuple}(\text{Data}, \text{tuplenil}(1))) \]
\[ S_{i \cdot \text{case}_0}(\text{false, Args}) = \]
\[ S_{i \cdot \text{case}_1}(\text{patterns_match}(\text{Args}, P_2), \text{Args}) \]
\[ S_{i \cdot \text{case}_1}(\text{true, Args}) = \]
\[ \text{tuple}(S_k, \text{tuple}(\text{Data}, \text{tuplenil}(2))) \]
\[ \ldots \]
\[ S_{i \cdot \text{case}_{(n-1)}}(\text{true, Args}) = \]
\[ \text{tuple}(S_u, \text{tuple}(\text{Data}, \text{tuplenil}(n-1))) \]
\[ S_{i \cdot \text{case}_{(n-1)}}(\text{false, Args}) = \]
\[ S_{i \cdot \text{case}_n}(\text{patterns_match}(\text{Args}, P_n), \text{Args}) \]
\[ S_{i \cdot \text{case}_n}(\text{true, Args}) = \]
\[ \text{tuple}(S_v, \text{tuple}(\text{Data}, \text{tuplenil}(n))) \]

proc

\[ \text{fsm}_S_i(\text{Cmd: Term}, \text{Data: Term}) = \]
\[ \text{actions}(1). \]
\[ \text{fsm}\_\text{next}\_\text{state}(\text{element}(1, S_i(\text{Cmd}, \text{Data})), \]
\[ \text{element}(2, S_i(\text{Cmd}, \text{Data}))) \]
\[ \triangleleft \text{element}(3, S_i(\text{Cmd}, \text{Data})), \text{ Cmd, Data}) = 1 \triangleright \]
\[ (\text{actions}(2). \]
\[ \text{fsm}\_\text{next}\_\text{state}(\text{element}(1, S_i(\text{Cmd}, \]
\[ \text{Data})), \text{element}(2, S_i(\text{Cmd}, \text{Data}))) \]
\[ \triangleleft \text{element}(3, S_i(\text{Cmd}, \text{Data})), \text{Cmd, Data}) = 2 \triangleright \]
\[ \ldots \]
\[ (\text{actions}(n). \]
\[ \text{fsm}\_\text{next}\_\text{state}(\text{element}(1, S_i(\text{Cmd}, \]
\[ \text{Data})), \text{element}(2, S_i(\text{Cmd}, \text{Data}))) \]
\[ \triangleleft \text{element}(3, S_i(\text{Cmd}, \text{Data})), \text{Cmd, Data}) = n \triangleright \]
\[ \text{delta} \ldots ) \]

Figure 11: Translation rules for Erlang state function.
Related Work - Model Checking Erlang in $\mu$CRL

- Once an Erlang program is translated into a $\mu$CRL 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 $\mu$-calculus (a fragment of the modal $\mu$-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 $\mu$CRL and defines rules to support the translation of timed Erlang functions into $\mu$CRL.
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 $\mu$CRL syntax.

<table>
<thead>
<tr>
<th>sort</th>
<th>var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>t:Timer</td>
</tr>
<tr>
<td>Timer</td>
<td>n:Natural</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>func</th>
<th>rew</th>
</tr>
</thead>
<tbody>
<tr>
<td>off: → Timer</td>
<td>expire(off) = F</td>
</tr>
<tr>
<td>on: Natural → Timer</td>
<td>expire(on(n)) = eq(0,n)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>map</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pred: Timer → Timer</td>
<td>pred(on(n)) = on(pred(n))</td>
</tr>
<tr>
<td>expired: Timer → bool</td>
<td>pred(off) = off</td>
</tr>
<tr>
<td>set: Timer $\neq$ Natural → Timer</td>
<td>set(t,n) = on(n)</td>
</tr>
<tr>
<td>reset: Timer → Timer</td>
<td>reset(t) = off</td>
</tr>
</tbody>
</table>

**Figure 12:** The syntax of a $\mu$CRL 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 $\mu$CRL, 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, \textit{locked} and \textit{open}, and a system code for opening the door.
- Initially, the door is set to \textit{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 \textit{open} state and no action is performed within a defined period, a \textit{timeout} event is generated, which activates the \textit{timeout} function to close the door.
-module(fsm_door).
-export([start_link/1, button/1, init/1, 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.
Figure 16: Translating rules for timed Erlang functions.
\[ \text{proc} \]
\[
\text{fsm\_locked}(\text{Cmd}: \text{Term}, \text{Data}: \text{Term}, x: \text{Natural}) = \\
\quad \text{do\_unlock} . \text{locked\_timing}(\text{Cmd}, \text{Data}, \text{on}(x))
\]
\[
\quad \triangleleft \text{term\_to\_bool}((\text{equal}(\text{element}(\text{int}(1), \text{element}(\text{int}(3), \\
\quad \text{locked}(\text{Cmd}, \text{Data}))), \text{int}(1)))) \triangleright
\]
\[
\quad \text{warning\_message} . \\
\quad \text{fsm\_next\_state}(\text{element}(\text{int}(1), \text{locked}(\text{Cmd}, \text{Data})), \\
\quad \text{element}(\text{int}(2), \text{locked}(\text{Cmd}, \text{Data})))
\]
\[
\text{locked\_timing}(\text{Cmd}: \text{Term}, \text{Data}: \text{Term}, t: \text{Timer}) = \\
\quad \text{count\_down\_locked}(\text{Cmd}, \text{Data}, t) + \\
\quad \text{fsm\_next\_state}(\text{element}(\text{int}(1), \text{locked}(\text{Cmd}, \text{Data})), \\
\quad \text{element}(\text{int}(2), \text{locked}(\text{Cmd}, \text{Data})))
\]
\[
\text{count\_down\_locked}(\text{Cmd}: \text{Term}, \text{Data}: \text{Term}, t: \text{Timer})
\]
\[
\quad \text{tick} . \text{locked\_timing}(\text{Cmd}, \text{Data}, \text{pred}(t))
\]
\[
\quad \triangleleft \text{not}(\text{expired}(t)) \triangleright
\]
\[
\quad \text{time\_out} . \text{warning\_message} . \\
\quad \text{fsm\_next\_state}(\_\text{locked}, \text{tuple}(\text{nil}(\text{abc})))
\]

Figure 17: Translating rules for timed Erlang functions.
\begin{verbatim}
proc
  fsmLocked(Cmd:Term,Data:Term,x:Natural) =
    do_unlock . locked_timing(Cmd,Data, on(x))
    \(\triangleq\) term_to_bool(equal(element(int(1),element(int(3),
        locked(Cmd,Data))),int(1))) \(\triangleright\)
    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))
    \(\triangleq\) not(expired(t)) \(\triangleright\)
    time_out . warning_message .
    fsm_next_state(s_locked,tuplenil(abc))
\end{verbatim}

Figure 18: Translating rules for timed Erlang functions.
\[
\text{proc}
\]
\[
\text{fsm\_locked(Cmd:Term,Data:Term,x:Natural) = do\_unlock . locked\_timing(Cmd,Data, on(x))} \\
\text{\quad \triangleright term\_to\_bool(equal(element(int(1),element(int(3), locked(Cmd,Data))),int(1))) } \\
\text{\quad \triangleright warning\_message .} \\
\text{fsm\_next\_state(element(int(1),locked(Cmd,Data)),} \\
\text{\quad element(int(2),locked(Cmd,Data)))}
\]
\[
\text{locked\_timing(Cmd:Term,Data:Term,t:Timer) = count\_down\_locked(Cmd,Data,t) +} \\
\text{fsm\_next\_state(element(int(1),locked(Cmd,Data)),} \\
\text{\quad element(int(2),locked(Cmd,Data)))}
\]
\[
\text{count\_down\_locked(Cmd:Term,Data:Term,t:Timer)} \\
\text{\quad tick . locked\_timing(Cmd,Data,pred(t))} \\
\text{\quad \triangleright not(expired(t)) \triangleright} \\
\text{\quad\quad time\_out . warning\_message .} \\
\text{\quad\quad fsm\_next\_state(s\_locked,tuplenil(abc))}
\]

Figure 19: Translating rules for timed Erlang functions.
Figure 20: Translating rules for timed Erlang functions.
**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))
```
proc
    fsm_unlocked(Cmd:Term,Data:Term,x:Natural) =
        do_unlock . locked_tamining(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_tamining(Cmd:Term,Data:Term,t:Timer) =
        count_down_unlocked(Cmd,Data,t) +
        fsm_next_state(element(int(1),locked(Cmd,Data)),
            element(int(2),locked(Cmd,Data)))
    count_down_unlocked(Cmd:Term,Data:Term,t:Timer)
        tick . locked_tamining(Cmd,Data,pred(t))
        < not(expired(t)) ▷
            time_out . warning_message .
            fsm_next_state(s_unlocked,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

fsms_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 $\mu$CRL. 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 $\mu$CRL 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 $\mu$-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 (£5 for a cup), paying £4 and then trying to take the drink away”;
- The second simulates “selecting tea (£4 for a cup), paying £5 and then taking the drink away”.
- The sequences are coded in the lists \([\{\text{selection, cappuccino, 5}\}, \{\text{pay, 4}\}, \{\text{cup\_remove}\}]\) and \([\{\text{selection, tea, 4}\}, \{\text{pay, 5}\}, \{\text{cup\_remove}\}]\), and are initialized in the \(\mu\text{CRL}\) 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.
• 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:

\[
\langle \text{true} \quad \circ \quad \text{cmd(tuple(selection,tuple(cappuccino, tuplenil(int(5)))))} \quad \circ \quad \text{not “time_out”} \quad \circ \quad \text{cmd(tuple(pay,tuplenil(int(4)))))} \quad \rangle \langle \text{true} \quad \circ \quad \text{time_out} \quad \circ \quad \text{return_coin(int(4))} \quad \rangle
\]

• 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:

\[
[\text{(not “time_out”)} \quad \circ \quad \text{“return_coin(int(4))” or “return_coin(int(4))”}] \quad \text{false}
\]
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 $\mu$CRL.
- By using an explicit \textit{tick} event, a discrete-time timing model is defined to support the translation of timed Erlang functions into $\mu$CRL.
- 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 $\mu$CRL 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 $\mu$CRL specification. We are currently upgrading the toolset \textit{etomcrl} where the translation of \textit{timeout} events will be incorporated.
Thank you!

Comments & Questions?