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

Qiang Guo 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

Abstract

Recent work has looked at how Erlang programs could be model-checked via translation into the process algebra μCRL. Rules for translating Erlang programs and OTP components into μCRL have been defined and investigated. However, in the existing work, no rule is defined for the translation of *timeout* events into μCRL. This could 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 investigate the verification of timed Erlang/OTP components in μCRL. By using an explicit *tick* action in the μCRL specification, a discrete-time timing model is defined to support the translation of timed Erlang functions into μCRL. Two small examples are presented, which demonstrates the applications of the proposed approach.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Verification

Keywords Erlang, OTP, Process algebra μCRL, Timeout events, 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. Since being developed, its use has spread to a variety of sectors such as TCP/IP programming (HTTP, SSL, Email, Instant messaging, etc), web-servers, databases, advanced call control services, banking, 3D-modelling.

A unique 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. Thus, for the development of fault-tolerant systems containing soft real-time requirements, the use of OTP components helps to reduce the complexity of system design while increasing the robustness.

Although Erlang has many high-level features, verification can be still non-trivial. One possible way to verify an Erlang program is to abstract it into a formal model, upon which model checking [11] techniques can be applied. This approach has recently been applied to the verification of Erlang programs and OTP components [2, 3, 5, 7] where the process algebra μCRL [14] has been used as the formal language upon which verification is carried out.

The translation from Erlang to μCRL is performed in two stages. First, a source to source transformation is applied, resulting in Erlang code that is optimised for the verification, but has identical behaviour. Second, this output is translated to μCRL. A toolset, *etomcrl*, has been developed to automate the process of translation of an Erlang program into a μCRL specification.

Erlang/OTP software is usually written according to strict design patterns that make extensive use of software components. Encapsulated in the extensive OTP library are a variety of design patterns, each of which is intended to solve a particular class of problem. Solutions to each such problem come in two parts. The generic part is provided by OTP as a library module and the specific part is implemented by the programmer in Erlang. Typically these specific callback functions embody algorithmic features of the system, whilst the generic components provide for fault tolerance, fault isolation and so forth. Translations of the callback modules of the OTP generic servers and supervisors have been investigated in [2, 3, 6, 7].

In addition to generic servers and supervisors, OTP provides further generic components including finite state machines (FSMs), event handlers, and applications. These considerably simplify the building of systems. In [15], the verification of OTP FSM programs using μCRL has been studied, and a model is proposed to support the translation of an Erlang FSM program into μCRL. In order to define the correct translation, and techniques proposed in [16] are applied to deal with the presence of overlapping patterns in pattern matching.

For some OTP components such as the FSM, timing restrictions might be applied, meaning that the execution of a function must be completed within a defined time period. If the function fails to do so, a *timeout* event is generated and the corresponding *timeout* function will be activated to process the event.

However, 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. By using an explicit *tick* event, a discrete-time timing model is defined to support the translation of timed Erlang functions into μCRL. Two

Erlang'07, October 5, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-675-2/07/0010…$5.00

small examples are presented, which demonstrates the applications of the proposed approach.

The rest of this paper is organized as follows: Section 2 briefly introduces the Erlang programming language and the process algebra μCRL; Section 3 reviews the related work for the translation of Erlang programs and OTP components into μCRL; Section 4 investigates the translation of timed OTP components into μCRL; Section 5 illustrates two small examples; conclusions are finally drawn in Section 6.

2. Preliminaries

2.1 Erlang

Erlang [1] is a functional programming language, and as such an Erlang program consists of a set of modules, each of which define a number of functions. Functions that are accessible from other modules need to be explicitly declared as *export*. A function named *f_name* in the module *module* and with arity *N* is often denoted as *module:f_name/N*.

Erlang is a concurrent programming language, and as such provides a light-weight process model. Several concurrent processes can run in the same virtual machine, each of which being called a *node*. Each process has a unique identifier to address the process and a message queue to store the incoming messages. Erlang has an asynchronous communication mechanism where any process can send (using the `!` operator) a message to any other process of which it happens to know the *process identifier*. Sending is always possible and non-blocking; the message arrives in the unbounded mailbox of the specified process. The latter process can inspect its mailbox by the `receive` statement. A sequence of patterns can be specified to read specific messages from the mailbox. When reading a message, a process is suspended until a matching message arrives or timeout occurs. A distributed system can be constructed by connecting a number of virtual machines.

2.2 OTP

A unique feature of Erlang is the OTP architecture, which is designed to support the construction of fault-tolerant systems containing soft real-time requirements. Each OTP design pattern solves a particular class of problems, and solutions to each such problem come in two parts: the generic part and the specific part. The first is provided as a library module, while, the second is implemented by the programmer where the necessary algorithms are applied. Generic servers, supervisors and finite state machines are three key components which account for over 80% of OTP compliant code.

2.2.1 Generic servers

The *gen_server* module provides a standard set of interface functions for synchronous and asynchronous communication, debugging support, error and timeout handling, and other administrative tasks. 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. When a client wants to synchronously communicate with the server, it calls the standard *gen_server:call* function with a certain message as an argument. If an asynchronous communication is required, the *gen_server:cast* is invoked where no response is expected after a request is sent to the server. A *terminate* function is also defined in the call back module. This function is called by the server when it is about to terminate, which allows the server to do any necessary cleaning up.

2.2.2 Supervisors

Erlang/OTP supports fault-tolerance by using the *supervision tree*, which is a structure where the processes in the internal nodes (supervisors) monitor the processes in the external leafs (children). A supervisor is a process that starts a number of child processes, monitors them, handles termination and stops them on request. The children themselves can also be supervisors, supervising its children in turn.

2.2.3 Finite state machines

The *gen_fsm* module supports the implementation of finite state machines, and these are used extensively in a variety of contexts. A (deterministic) FSM M can be described as a set of relations of the form $State(S) \times Event(E) \rightarrow (Action(A), State(S))$ where S, E and A are finite and nonempty sets of states, events and actions respectively. For an implementation using the *gen_fsm* module, *gen_fsm* is started by calling *start_link(Code)* to register a new *gen_fsm* process.

```
start_link(Code)→
  gen_fsm:start_link({local, fsm_name},
          callback_module_name, Code, [ ]).
```

If the registration succeeds, the new process calls the callback function *callback_module_name*:*init(Code)* where the initial state and the corresponding state data are set.

The state transition rules are written as a number of state functions that conform to the following convention:

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

A state function ends up by returning the name of the next state and an updated state data. *Timer* is an optional element. If *Timer* 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.

2.2.4 Example - a door with code lock

An example of a door with a code lock system is modelled by OTP FSM. The initial design, illustrated in Figure 1, 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.

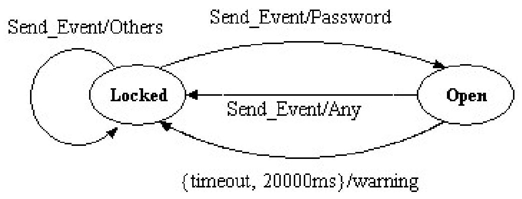


Figure 1. FSM - door with code lock.

The Erlang/OTP implementation is shown in Figure 2 where the function $button$ is defined to simulate the receiving of a password. The action $send\_event$ triggers a state transition where a state function is executed, in this example either *locked* or *open*.

A password is generated from an external action and is evaluated to open the door or not. The timer for the state *open* is set

```
-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 2. The Erlang code for a door with code lock.

to 20,000ms. If the door is switched to *open* and no action is performed within 20,000ms, a *timeout* event is generated, which enables the function *open(timeout,Code)* to process the event.

2.3 The process algebra μCRL

The process algebra μCRL (micro Common Representation Language) [14] 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 (in a way similar to LOTOS [9]).

A μCRL specification comprises two parts: the data types and the processes. Processes are declared using the keyword *proc*, and contains actions representing atomic events that can be performed. These actions must be explicitly declared using the keyword *act*. Data types used in μCRL are specified as the standard abstract data types, using sorts, functions and axioms. Sorts are declared using the keyword *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 the evaluation of terms.

A number of process-algebraic operators are defined in μCRL, these being: sequential composition ($\cdot$), non-deterministic choice (+), parallelism ($\|$) and communication ($|$), encapsulation (∂), hiding (τ), renaming (ρ) and recursive declarations. A conditional expression $true \lhd condition \rhd false$ allows data elements to influence the flow of control in a process, and an alternative quantification operator ($\sum$) provides the possibly infinite choice over some sorts.

In μCRL, parallel processes communicate via the synchronization of actions. The communication in a process definition is described by its communication specification, denoted by the keyword *comm*. This describes which actions may synchronize on the level of the labels of actions. For example, in *comm* $in|out$, each action $in(t_1, ..., t_k)$ can communicate with $out(t'_1, ..., t'_m)$ provided $k = m$ and t_i and t'_i denote the same element for $i = 1, ..., k$.

3. Related work

3.1 Translating Erlang syntax

Translating Erlang into μCRL was initially studied in [3, 5, 6, 7]. A toolset, *etomcrl*, was developed for automating the process of translation. The translation from Erlang to μCRL is performed in two stages. First, a source to source transformation is applied, resulting in Erlang code that is optimized for the verification, but has identical behaviour. Second, this code is translated to μCRL.

The actual translation is quite involved due to particular language features in Erlang. For example, Erlang makes use of higher-order functions, whereas μCRL is 1st order; Erlang is dynamically typed, but μCRL is statically typed; in Erlang communication can take place in a computation, in μCRL it cannot. However, μCRL is sufficiently close that such a translation is feasible, and model checking on it computationally traceable even if the translation is involved.

```
sort
    Term
func
    pid: Natural → Term
    int: Natural → Term
    nil: → Term
    cons: Term # Term → Term
    tuplenil: Term → Term
    tuple: Term # Term → Term
    true: → Term
    false: → Term
```

Figure 3. The translation scheme for Erlang data types.

Because Erlang is dynamically typed it is necessary to define in μCRL a data type *Term* where all data types defined in Erlang are embedded. The translation of the Erlang data types to μCRL is then basically a syntactic conversion of constructors as shown in Figure 3.

Atoms in Erlang are translated to μCRL constructors; *true* and *false* represent the Erlang booleans; *int* is defined for integers; *nil* for the empty list; *cons* for a list with an element (the head) and a rest (the tail); *tuplenil* for a tuple with one element; *tuple* for a tuple with more than one element; and *pid* for process identifiers. For example, a list $[E_1, E_2, ..., E_n]$ is translated to μCRL as $cons(E_1, cons(E_2, cons(..., cons(E_n, nil))))$. A tuple $\{E_1, E_2, ..., E_n\}$ is translated to μCRL as $tuple(E_1, tuple(E_2, ..., tuplenil(E_n)))$.

Variables in Erlang are mapped directly to variables in μCRL. Operators are also translated directly, specified in a μCRL library. For example, *A + B* is mapped to *mcrl_plus(A,B)*, where *mcrl_plus(A,B) = int(plus(term_to_nat(A), term_to_nat(B)))*.

Higher-order functions in an Erlang code are flattened into first-order alternatives. These first-order alternatives are then translated into rewrite rules.

Program transformation is defined to cope with side-effect functions. With a source-to-source transformation, a function with side-effects is either determined as a pure computation or a call to another function with side-effects. *Stacks* are defined in μCRL where *push* and *pop* operations are defined as communication actions. The value of a pure computation is pushed into a stack and is popped when it is called by the function.

3.2 Overlapping in pattern matching

Erlang makes extensive use of pattern matching in its function definitions. The toolset *etomcrl* translates pattern matching in a way where overlapping patterns are not considered. This might induce faults in the μCRL specification in our translation, and we need to use techniques to cope with the occurrence of overlapping patterns.

In Erlang, evaluation of pattern matching works from top to bottom and from left to right. When the first pattern is matched, evaluation terminates after the corresponding clauses are executed. However, 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.

A solution to the problem of overlapping in pattern matching was discussed by Benac-Earle [5], however, the solution was not implemented in the toolset *etomcrl*. Subsequently, Guo *et al.* [16] proposed a different solution, whereby an Erlang program with overlapping patterns is transformed into a counterpart program without overlapping patterns. The rewriting operation rewrites all pattern matching clauses in the original code into some calling functions. A calling function is activated by a guard that is determined by the function *patterns_match*. Function *patterns_match* takes the predicate of the pattern matching clauses and one pattern as arguments and is *true* iff the predicate matches the pattern.

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. The evaluation of an SST is equivalent to the search of nodes in a tree, and thus is of linear complexity.

3.3 Translating OTP components

3.3.1 Translating generic servers

The *gen_server* module defines a Server/Clients structure of Erlang programs. The *gen_server* module is translated into a set of communicated processes in μCRL. Communication between two Erlang processes, which can be asynchronous, is translated via defining two process algebra processes, one of which is a buffer, while the other implements the logic.

The synchronous communication is modelled by the synchronizing actions of process algebra. One action pair is defined to synchronize the sender with the buffer of the receiver, while another action pair to synchronize the active receive in the logic part with the buffer. In this way the asynchronous communication and the Erlang message queue is simulated directly in the μCRL abstraction.

3.3.2 Translating finite state machines

Translation of the Erlang finite state machine (FSM) design pattern was studied in [15]. By extending the approach discussed in [3, 6, 7], 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*.

Simulating state management

A (one place) stack is used in the μCRL specification to simulate the management of FSM states and state data. The translation rules are given in Figure 4, where three actions, s_event, r_event and $send\_event$, are defined respectively. A command, generated from an external action is sent out to some other processes by action s_event. This command is received through action r_event and is used for further processing; $s\_event : r\_event = send\_event$.

An Erlang FSM state is assigned with a μCRL state name ("s_" plus the state name) and a state process ("fsm_" plus the state name). For example, state S_1 is given a μCRL state name

```
act
  s_event, r_event, send_event: Term
  s_command, r_command, cmm_command: Term

comm
  s_event | r_event = send_event
  s_command | r_command = cmm_command

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

Figure 4. Rules for translating Erlang FSM state processes.

s_S_1 and a state process fsm_S_1. 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 stack used for managing states and data is defined in a way where only one element can be read/written. This ensures that only one *current state* is available.

The process $write$ is defined to push the *current state* and the *state data* onto the stack while a process called $read$ is used to pop the *current state* and the *state data* from the stack. The process $fsm\_init(State{:}Term,\ Data{:}Term)$ is defined to initially push $tuple(Init\_State,\ tuplenil(State\_Data))$ 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 will receive commands through the action $r\_\ command$. The action $r\_command$ communicates with the action $s\_command$ which is externally performed. When a command is received, the process $fsm\_\ state\_change$, guarded by the action s_event, is enabled. It passes the command to the process $read$ where the *current state* and the *state data* are read from stack. The *current state* determines which state process is about to be activated.

A state process fsm_S_i starts by calling its μCRL state function $S_i(Command : Term, Data : Term)$. Function S_i returns a tuple with the form of $tuple(next\_state, tuple(new\_data, tuplenil(index)))$ where $next\_state$ shows the next state; new_data the updated state data. The $index$ saves an index number for the sequence of actions to be selected.

Translating state functions

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 5, 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 $actions(i)$.

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

Figure 5. An Erlang FSM state function with pattern matching.

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 in [16] are applied. Specifically, pattern matching clauses in the program are replaced by a series of case functions. These case functions are guarded by the $patterns\_match$ function that takes the predicate of pattern matching clauses and one pattern as arguments, then if the predicate matches the pattern, function *patterns_match* returns *true*; otherwise, *false*, and this eliminates the overlapping between patterns and ensures that the index returned by the μCRL state function is deterministic and unique. Figure 6 illustrates the translation rules for the Erlang state function shown in Figure 5.

When the state process fsm_S_i starts, it first calls the μCRL state function $S_i(Cmd, Data)$. S_i returns an index number i that determines which action sequence $action(i)$ is about to be performed. The process fsm_S_i ends up by calling process fsm_next_state, updating the *current state* and the *state data* in the stack.

3.4 Model checking Erlang with 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 [10]. The toolset CADP provides a number of tools for system behaviour checking. It includes an interactive graphical simulator, a tool for the visualization of labelled transition systems (LTSs), several tools for computing bisimulations and a model checker.

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 [19]. 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.

```
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 6. Translation rules for Erlang state function.

4. Translating timed Erlang/OTP components into μCRL

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

4.1 Defining a timer in μCRL

Two approaches might be considered to extend the existing work for coping with *timeout* events. The first is to use a timed extension to μCRL, while, the second to define a *timer* in the untimed μCRL. A timed version of μCRL is defined in [13] where time is incorporated as an abstract data type. However, without further modification, a timed μCRL cannot be used by some of existing tools. Moreover, timed μCRL is incompatible with liberalization (translating a specification into the intermediate format) and partial order reduction [13].

This work considers the use of the second approach. A *timer* and an explicit timing action *tick* are incorporated in the μCRL

```
sort                                var
    Timer                               t:Timer
func                                    n:Natural
    off:→ Timer                     rew
    on: Natural → Timer                 expire(off)=F
map                                     expire(on(n))=eq(0,n)
    pred: Timer → Timer                 pred(on(n))=on(pred(n))
    expired: Timer → bool               pred(off)=off
    set: Timer # Natural → Timer        set(t,n)=on(n)
    reset: Timer → Timer                reset(t)=off
```

Figure 7. The syntax of a μCRL timer

specification. A *timer* [8], syntactically illustrated in Figure 7, 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*.

As is common with other approaches to modelling time explicitly in a process algebra, an explicit timing action *tick* is defined. During the timing, the timer is periodically evaluated by the μCRL function *expired*. If the value of the timing does not come to zero, before calling the μCRL function *pred* to count down the timer by one unit, the action *tick* is performed once, standing for the passing of one time elapse unit; otherwise, a *timeout* event is generated.

4.2 Translating timed Erlang functions

For an Erlang function with timing restrictions, execution of the function needs to be completed within the defined time period. If the Erlang function fails to do so, a *timeout* event will be generated and the corresponding *timeout* function is invoked to process the event.

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*, while, the other copes with *count down*. When the timing begins, the *timing* process will be called. The *timing* process will either call another process (the execution of this function is completed within the defined time period) 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.

For example, the state function *locked* of the Erlang FSM program shown in Figure 2 has a timing restriction of 20,000ms. If the door is switched to *open* and no action is performed within 20,000ms, a *timeout* event is generated, which will activate the function *open(timeout,Code)* to first give a warning message and then close the door. The Erlang state function is translated into μCRL as shown in Figure 8. The process *fsm_locked* calls the process *timing_locked* by the end of execution, instantiating and initiating a timer. The *timing_locked* will either move to the process *fsm_next_state* driven by an action within the defined time, or count the timer down by one unit (one *tick* action is performed). A *time_out* event will be generated once time-up occurs, leading to the *warning_messages* and *do_lock* actions being performed.

```
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 8. Translating rules for timed Erlang functions.

4.3 Coping with synchronization

The *gen_fsm* module defines a function *sync_send_event(FSMRef, Event, Timer)* to provide synchronized communications between FSMs where *FSMRef* points to the Erlang FSM process to which an event is about to send. *Timer* is an optional element. 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. Thus, translation rules defined in subsection 4.2 can be applied. Figure 9 illustrates the translation scheme.

```
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 9. Coping with synchronization communication between FSMs.

The function is translated into two processes in μCRL. The process *sync_send_event* reads a command from the command list and then sends it off through action *s_command*[1]. Once a command is sent off, a timer is instantiated and initialized in the process *sync_send_event_timing*. The process expects to read a reply message through action *sync_read*. If the μCRL FSM process does not

[1] Action *s_command* is defined in Figure 4

return the message within the defined time period, a *timeout* event is generated and the program is terminated.

5. Experiments

Two small examples are presented in this section to illustrate the applications of the proposed approach.

5.1 A door with code lock

A door with a code lock system, illustrated in Figure 1, is explained in Section 2.2.4. The system is simulated with the Eralng/OTP FSM, and the implementation is shown in Figure 2. The program is translated into μCRL by applying the rules defined in [3, 15] and Section 4. In the simulation, the system code is set to *abc*. A sequence of external actions $[\{abb\}, \{abc\}]$ is initialized in the μCRL specification, stating that two passwords, abb and abc, are consecutively inputted.

Once the Erlang program is translated into μCRL, a standard toolset such as CADP [10] can be applied to check the system properties. The toolset CADP provides a tool for generating the labelled transition systems (LTSs) and a model checker for checking system behaviour. Figure 10 demonstrates the LTS for a door with code system derived from the μCRL specification.

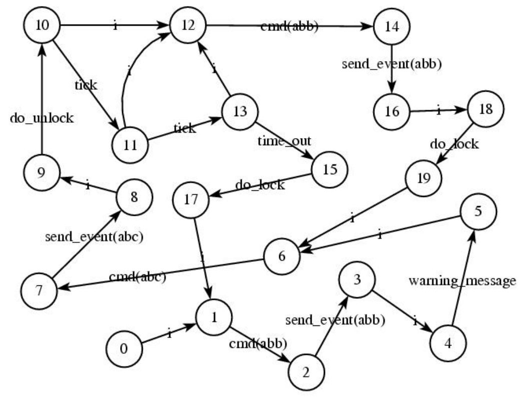


Figure 10. LTS derived from the door with code lock system.

Design properties of the system can then be verified with the CADP model checker. The system properties should be formalized in the regular alternation-free μ-calculus (a fragment of the modal μ-calculus) [19]. Thus once we have a specification in μCRL, applying model-checking approaches is standard.

For example, in this experiment, a property involving in *timeout* event can be formulated as:

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

stating that "Without being delayed for 20,000ms (two *tick* actions are performed), *time_out* event cannot be generated". 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).

Another property with timing issue one might wish to check can be formulated as:

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

stating that "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". Since in the original design, this property is not defined, when checking with the CADP, the toolset should return *false*.

The system design properties can be formulated with a set of μ-calculus logic expresses and saved in a SVL file. Through evaluating the SVL file, the process of verification can be automatically accomplished within a couple of seconds.

The example studied in this subsection is comparatively simple and is easy to be verified. A more complicated system, coffee machine system is presented in the next subsection to further illustrate the application of the proposed approach.

5.2 Coffee machine

A coffee machine has three states, these being, *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 11. The program initially sets the current state to $selection$. A timing restriction of 20,000ms is set to the state function *payment*.

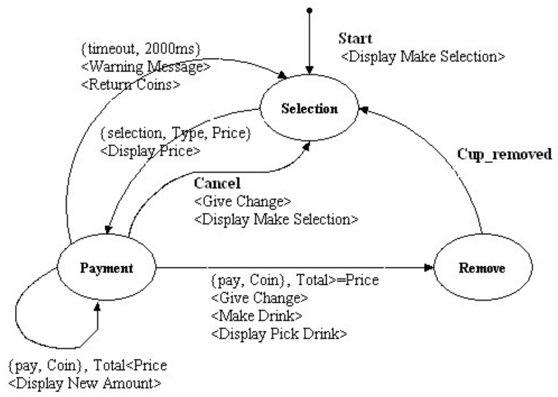
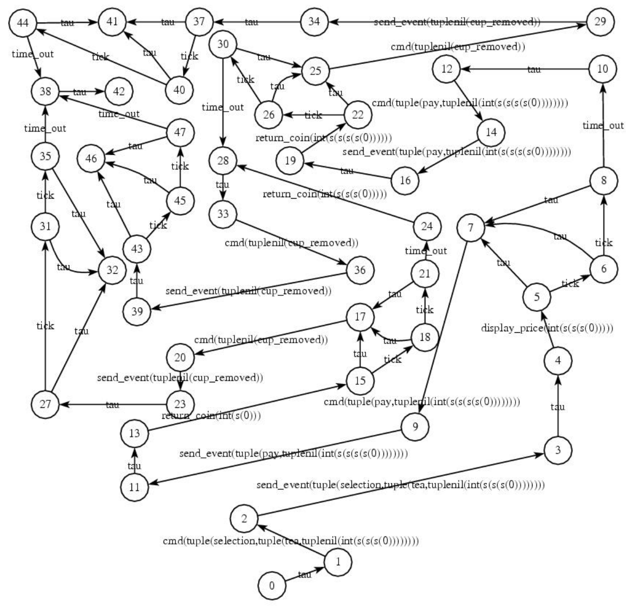


Figure 11. FSM - coffee machine.

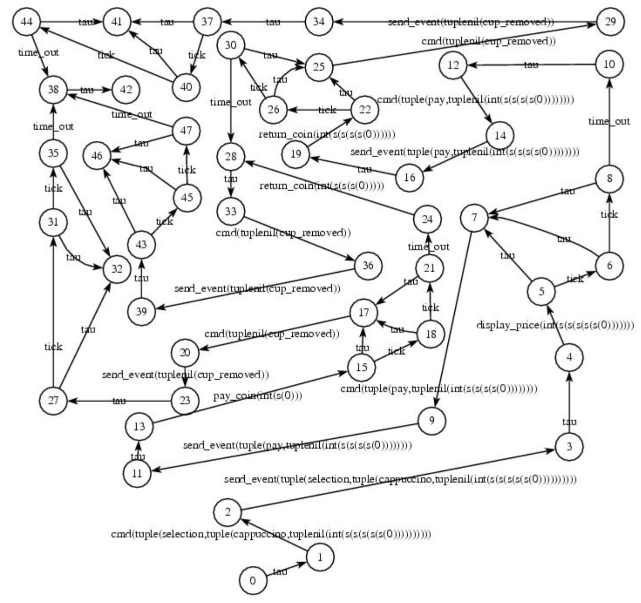
Four actions *display_price*, *pay_coin*, *return_coin* and *remove_cup* are defined in the μCRL specification where *display_price* displays the price for a selected drink; *pay_coin* requires a buyer to pay coins for the drink; *return_coin* returns the changes if more coins have been paid for the drink, or gives back the pre-paid coins if the transaction is cancelled. In the μCRL specification, the time elapse unit is defined as 10,000ms which states that a *tick* action in the LTS stands for the passing of 10,000ms.

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", while, the second simulates "selecting *tea* (£4 for a cup), paying £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.



send_event(tuplenil(cup_removed))
cmd(tuplenil(cup_removed))
cmd(tuple(pay,tuplenil(int(s(s(s(s(s(0))))))))
return_coin(int(s(s(s(0))))))
send_event(tuple(pay,tuplenil(int(s(s(s(s(s(0))))))))
return_coin(int(s(s(s(0)))))
time_out
tick
tau
display_price(int(s(s(s(0)))))
cmd(tuple(pay,tuplenil(int(s(s(s(s(0))))))))
return_coin(int(s(0)))
send_event(tuple(selection,tuple(tea,tuplenil(int(s(s(s(0))))))))
cmd(tuple(selection,tuple(tea,tuplenil(int(s(s(s(0))))))))

Figure 12. LTS: Buying tea with the payment higher than the price.



send_event(tuplenil(cup_removed))
cmd(tuplenil(cup_removed))
cmd(tuple(pay,tuplenil(int(s(s(s(s(0))))))))
return_coin(int(s(s(s(0))))))
send_event(tuple(pay,tuplenil(int(s(s(s(s(0))))))))
return_coin(int(s(s(s(0)))))
time_out
tick
tau
display_price(int(s(s(s(s(s(0)))))))
pay_coin(int(s(0)))
send_event(tuple(selection,tuple(cappuccino,tuplenil(int(s(s(s(s(s(0))))))))))
cmd(tuple(selection,tuple(cappuccino,tuplenil(int(s(s(s(s(s(0))))))))))

Figure 13. LTS: Buying cappuccino with the payment less than the price.

The LTSs can then be derived through the use of CADP. They are presented in Figure 12 and Figure 13 respectively. Figure 12 shows the LTS on buying a cup of tea with the payment higher than the price, while, Figure 13 demonstrates the LTS on buying a cup of cappuccino with the payment less than the price.

Model checking using the CADP model checker can then be applied. Properties involving in timing issues can be formalized as discussed. For example, 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(s(s(s(s(s(0))))))))))" * . (not "time_out") * . "cmd(tuple(pay,tuplenil(int(s(s(s(s(0))))))))" *><true * . "time_out" *. "return_coin(int(s(s(s(0)))))"> true

Similarly, 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(s(s(s(0)))))" or
"return_coin(int(s(s(s(s(0))))))")] false

6. Conclusions and future work

Erlang is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems. Generic components encapsulated as design patterns are provided by the Open Telecom Platform (OTP) library. Although Erlang has many high-level features, verification is still non-trivial. One possible approach is to perform an abstraction of an Erlang program into the process algebra μCRL, upon which standard verification tools can be applied.

Previous work has investigated the verification of Erlang programs and OTP components with μCRL. Rules that supports the translation of Erlang syntax and OTP supervisor, generic server and finite state machines are proposed and studied respectively. However, in the existing work, no rule is defined to deal with *timeout* event. This could dramatically degrade the usability of the existing work since in some real applications *timeout* events play a significant role in the system design. For example, in the telecommunication protocols, a sender often requires an acknowledgement within a defined time period after a message is sent out. Failing to do so leads to the assumption of data losing.

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. This inclusion of explicit *tick* events is, of course, not new and has been investigated as a means to include timings in process algebras such as CSP [17] and LOTOS (which owe much to toolsets for timed automata such as UPPAAL [20]).

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.

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] S. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In Manfred Broy and Alexandre V. Zamulin, editors, *5th Andrei Ershov International Conference on Perspectives of System Informatics PSI'2003*, volume 2890 of *LNCS*, pages 178–192. Springer-Verlag, July 2003.

[9] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1987.

[10] CADP. *http://www.inrialpes.fr/vasy/cadp/*.

[11] E. Clarke, O. Grumberg, and D. Long. *Model Checking*. MIT Press, 1999.

[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. The syntax and sematics of timed μCRL. In *SEN R9709, CWI, Amsterdam*, 1997.

[14] 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.

[15] 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.

[16] 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.

[17] C. A. R. Hoare. *Communicating Sequential Processes (Prentice-Hall International Series in Computer Science)*. Prentice Hall, April 1985.

[18] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999.

[19] D. Kozen. Results on the propositional μ-calculus. *Theortical Computer Science*, 27:333–354, 1983.

[20] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

A. μCRL specification for a door with code lock system

```
sort
  Timer

func
  off: → Timer
  on: Natural → Timer

map
  pred: Timer → Timer
  expire: Timer → Bool

var
  t: Timer
  n: Natural

rew
  expire(off) = F
  expire(on(n)) = eq(0,n)
  pred(on(n)) = on(pred(n))
  pred(off) = off

%———————————————————————

sort
  TermStack

func
  empty: → TermStack
  push: Term # TermStack → TermStack

map
  is_top: Term # TermStack → Bool
  is_empty: TermStack → Bool
  pop: TermStack → TermStack
  top: TermStack → Term
  eq: TermStack # TermStack → Bool

var
  S1,S2: TermStack
  T1,T2: Term

rew
  is_top(T1,empty) = F
  is_top(T1,push(T2,S1)) = eq(T1,T2)
  is_empty(empty) = T
  is_empty(push(T1,S1)) = F
  pop(push(T1,S1)) = S1
  top(push(T1,S1)) = T1
  eq(empty,S2) = is_empty(S2)
  eq(push(T1,S1),S2) = and(is_top(T1,S2),eq(S1,pop(S2)))

act
  rcallvalue,wcallresult,push_callstack: Term
  rcallresult,wcallvalue,pop_callstack: Term

comm
  rcallvalue | wcallresult = push_callstack
  rcallresult | wcallvalue = pop_callstack

proc
  CallStack(S:TermStack) =
    sum(Value:Term,
     rcallvalue(Value).CallStack(push(Value,S))) +
        (delta ◁ is_empty(S) ▷
          wcallvalue(top(S)).CallStack(pop(S)))

%——————————————————————————

act
  s_event, r_event, send_event: Term
  s_command, r_command, cmd: Term
  do_lock
  do_unlock
  warning_message
  tick
  time_out

comm
  s_event | r_event = send_event
  s_command | r_command = cmd

map
  patterns_matching: Term # Term → Term
  locked: Term # Term → Term
  open: Term # Term → Term
  locked_case_0_0: Term # Term # Term → Term
  locked_case_0_1: Term # Term # Term → Term

var
  Command, Data: Term
  Pattern1, Pattern2: Term

rew
  locked(Command, Data) =
      locked_case_0_0(patterns_matching(Command,
              element(int(1),Data)),Command,Data)
  locked_case_0_0(true, Command, Data) =
      tuple(s_open,tuple(Data,tuplenil(tuplenil(int(1)))))
  locked_case_0_0(false, Command, Data) =
      locked_case_0_1(patterns_matching(do_not_care,
              do_not_care),Command,Data)
  locked_case_0_1(true, Command, Data) =
      tuple(s_locked, tuple(Data,tuplenil(tuplenil(int(2)))))
  open(Command, Data) =
      tuple(s_locked, tuple(Data,tuplenil(tuplenil(int(1)))))
  patterns_matching(Pattern1, Pattern2) =
      equal(Pattern1,Pattern2)

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

  read(Command:Term) =
    sum(Val:Term, rcallresult(Val).
      (fsm_locked(Command,element(int(2),Val),on(2))
        ◁ is_s_locked(element(int(1),Val))▷
          (fsm_open(Command,element(int(2),Val),on(2))
            ◁ is_s_open(element(int(1),Val)) ▷ delta)))

  fsm_locked(Command:Term,Data:Term,t:Timer) =
    do_unlock.locked_timing(Command,Data,t)
      ◁ term_to_bool(equal(element(int(1),element(int(3),
          locked(Command,Data))),int(1))) ▷
              warning_message.
              fsm_next_state(element(int(1),
                locked(Command,Data)),
                    element(int(2),locked(Command,Data)))

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

  count_down_locked(Command:Term,Data:Term,t:Timer) =
    tick.
    locked_timing(Command,Data,pred(t))
        ◁ not(expire(t)) ▷
            time_out.
            do_lock.
            fsm_next_state(s_locked,tuplenil(abc))

  fsm_open(Command:Term,Data:Term,t:Timer) =
    do_lock.fsm_next_state(s_locked, tuplenil(abc))

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

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

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

  fsm_command(Command:Term, CmdSet:Term) =
    s_command(hd(CmdSet)).
    fsm_command(tl(CmdSet), CmdSet)
        ◁ is_nil(Command) ▷
            s_command(hd(Command)).
            fsm_command(tl(Command), CmdSet)

init
  encap(s_command,r_command,
    fsm_command(nil, cons(abb,cons(abc, nil))) ||
      hide(push_callstack,pop_callstack,
        encap(rcallvalue,wcallvalue,rcallresult,wcallresult,
            s_event,r_event,CallStack(empty) ||
              fsm_init(s_locked,tuplenil(abc)) ||
                  fsm_change_state)))
```