Verifying Erlang/OTP Components in μCRL

Qiang Guo

Department of Computer Science,
The University of Sheffield,
Regent Court, 211 Portobello Street, S1 4DP, UK
Q.Guo@dcs.shef.ac.uk

Abstract. 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 (existing) approach is to perform an abstraction of an Erlang program into the process algebra μCRL, upon which standard verification tools can be applied. In this paper we extend this work and propose a model that supports the translation of an OTP finite state machine design pattern into a μCRL specification. Then a standard toolset such as CADP can be applied in order to check properties that should hold for the system under development. Two small examples are presented, which experimentally show how the proposed model assists in model checking Erlang OTP components in μCRL.

Key words: Erlang, OTP, process algebra, μCRL, Verification

1 Introduction

Model checking [8] has been widely used in system design and verification. The advantage of using model checking based techniques for system verification is that, when a fault is detected, model checker can generate a counter example given as a trace. These traces are useful since they help the system designer to understand the reasons that cause the occurrence of failures and provide clues for fixing the problem.

Model checking can be applied in two ways. One way, in combination with a model checker, is to use a formal specification language such as a process algebra [15], to obtain a correct specification. The specification is then used to develop an implementation in a programming language such as Erlang [1]. The other way uses the program code as a starting point and abstracts it into a form suitable for use by a model checker, and this requires an interpretation mechanism to support the translation of the programming language into the formal specification language used by the model checker.

Recently this second approach has been applied to the verification of Erlang programs and OTP components [2,3,6,10]. Here the process algebra μCRL [13] has been used as the formal language upon which verification is carried out. A toolset, *etomcrl*, has been developed to automate the process of translation of an Erlang program into a μCRL specification. The translation from Erlang to μCRL is performed in two stages, where in the first, a source to source transformation is applied, resulting in Erlang code that is optimised for the verification, but has identical behaviour. Then second, this output is translated to μCRL.

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. The *etomcrl* translation tool currently produces translations of the callback modules of the OTP generic servers and supervisors.

In addition to generic servers and supervisors, OTP provides further generic components including finite state machines, event handlers, and applications. These considerably simplify the building of systems. In this paper we extend the above approach to develop a model that supports the translation of OTP finite state machines (FSMs) into μCRL.

To do so, the Erlang state function in the FSM is translated into two parts in μCRL, one of which defines a μCRL state-process that can be called or synchronized by some other processes, while, the other consists of a series of μCRL state functions. The set of sequences of actions defined in an Erlang state function are translated into a set of pre-defined action sets in μCRL, each of which is uniquely indexed by an integer. A μCRL state-process starts by calling its μCRL state function. The function returns an index number that determines which pre-defined action set needs to be performed. we use a simple stack to simulate the management of FSM states and data. In order to define the correct translation we use techniques proposed in [16] which are needed to deal with the presence of overlapping patterns in pattern matching.

The rest of this paper is organized as follows: Section 2 introduces the Erlang programming language; Section 3 describes the process algebra μCRL; Section 4 reviews the related work for the translation of Erlang programs into μCRL; Section 5 investigates the translation of Erlang FSM programs into μCRL; Section 6 evaluates the proposed model with two case studies; conclusions are finally drawn in Section 7.

2 Erlang and OTP

The programming language Erlang [1] is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems. Since being developed, it has been used to implement some substantial business critical applications such as the Ericsson AXD 301 high capacity ATM switch [4]. Erlang is available under an Open Source licence from Ericsson, and its use has spread to a variety of sectors. Applications include TCP/IP programming (HTTP, SSL, Email, Instant messaging, etc), web-servers, databases, advanced call control services, banking, 3D-modelling.

Erlang 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 lightweight 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.

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. Its use has been very successful since Erlang/OTP software is usually written according to strict design patterns that make extensive use of software components. Each design patterns solves a particular class of problem, and solutions to each such problem come in two parts: the generic part is provided as a library module and the specific part is implemented by the programmer. The specific callback functions implement the necessary algorithm, and fault tolerance, fault isolation etc is provided by the generic component. The following briefly reviews generic servers, supervisors and finite state machines - the three key components which account for around 80% of OTP compliant code.

Generic servers and supervisors The Erlang/OTP supports a generic implementation of a server by providing the *gen_server* module which 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.

When developing concurrent and distributed systems, a commonly accepted assumption is that any Erlang process may unexpectedly terminate due to some failures. 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 leaves (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 their children in turn.

Finite state machines The Erlang/OTP architecture supports the implementation of finite state machines by providing the *gen_fsm* module, 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. If M is in state $s \in S$ and receives event $e \in E$, action $a \in A$ is performed, moving M to a state $s' \in S$. For an implementation using the *gen_fsm* module, *gen_fsm* is started by calling *start_link(Code)*:

$$\begin{aligned}
&start\_link(Code) \rightarrow \\
&\quad gen\_fsm : start\_link(\{local, fsm\_name\}, \\
&\qquad\qquad callback\_module\_name, Code, [\,]).
\end{aligned}$$

$\{local, fsm\_name\}$ implies that the FSM is locally registered as *fsm_name*; *callback_module_name* is the name of the callback module where the callback functions are located; $Code$ is a term that is passed to the callback function *init*; the last argument, [ ], is a list of options. If the registration succeeds, the new *gen_fsm* process calls the callback function *callback_module_name*:*init*($Code$). This function is expected to return $\{ok, StateName, StateData\}$ where $StateName$ saves the name of initial state and $StateData$ the corresponding state data.

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

$$\begin{aligned}
&\text{StateName(Event, StateData)} \rightarrow \\
&\quad \text{... code for actions ...;} \\
&\quad \{\text{next\_state,StateName}',\text{StateData}',\text{Timer}\}.
\end{aligned}$$

Having performed all pre-defined actions, the state function returns a tuple that contains the name of the next state, $StateName'$, and an updated state data, $StateData'$. $StateName'$ is updated as the new current state by the *gen_fsm* module. $Timer$ is an optional element, if it is set to a value, a timer is instantiated, and a *timeout* event will be generated when the time-up occurs.

The function *send_event* is defined to trigger a transition. When *send_event* is executed, the *gen_fsm* module automatically calls the *current state* function.

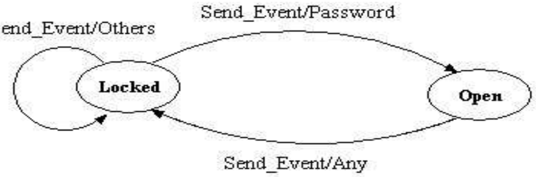


Fig. 1. FSM - door with code lock.

```
-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};
         _Wrong →
             action:display_message(),
             {next_state, locked, Code}.

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

Fig. 2. The Erlang code for a door with code lock.

Example - a door with code lock The initial design for a door with a code lock is illustrated in Figure 1, and 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.

The Erlang/OTP implementation of the system 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 generated from an external action is evaluated, and if the door is in the state *locked* and the received password is correct, the door will be opened through action *send_event*. Otherwise, if the password is not correct, the door remains locked. When the door is in the state *open* and action *send_event* is performed, the door will be locked, regardless of the password received.

3 The process algebra μCRL

The process algebra μCRL (micro Common Representation Language) [13] is an extension of the process algebra ACP [14], 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).

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 keywords *func* and *map*. 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 the 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 $out(t'_1, ..., t'_k)$ provided $k = m$ and t_1, t'_1 denote the same element for $i = 1, ..., k$.

As an example, consider the specification of a stack in μCRL given in Figure 3. The stack, initially defined in [3] for coping with side_effect functions, defines six actions, these being *rcallvalue*, *wcallresult*, *push_callstack*, *rcallresult*, *wcallvalue* and *pop_callstack*; *rcallvalue* $|$ *wcallresult* $=$ *push_callstack* and *rcallresult* $|$ *wcallvalue* $=$ *pop_callstack*. The action *rcallvalue* pushes a value to stack, while, the action *rcallresult* pops up the top value from stack.

An interleave relation with the process $CallStack$ needs to be defined for those processes that will exchange data with the stack. To save a $Value$, a process needs to perform $wcallresult(Value)$ first, which leads to the synchronization between this process and process $CallStack$. The action $sum(Value : Term,\ rcallresult(Value))$ is consequently performed, which pushes the value into the stack. To read a value, a process needs to perform $sum(Value : Term,\ rcallresult(Value))$ where *wcallvalue* is performed to pop up the top value from stack and assign it to $Value$.

4 Related work

As discussed in the introduction, Benac Earle *et al.* [2,3,6,10] have studied the translation of Erlang programs into μCRL and developed a toolset, *etomcrl*, for automating the process of translation.

```
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)))
```

Fig. 3. The syntax of μCRL stack.

4.1 Translating Erlang programs into μCRL

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 tractable even if the translation is involved.

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 4.

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 $tuple(E_1, tuple(E_2, ..., tuplenil(E_n)))$.

```
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
```

Fig. 4. The translation scheme for Erlang data types.

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 programs 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.

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.

4.2 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.

The problem of overlapping in pattern matching was studied in [16]. An approach was proposed where 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 searching of nodes in a tree, and thus is of linear complexity.

After an Erlang program has been translated into a μCRL specification, one can check the system properties by using some existing tools such as CADP [7]. 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 [9]. 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. We very briefly illustrate the approach in Section 6 where a few simple properties are defined for our running examples.

5 Translating Erlang/OTP FSMs into μCRL

This section investigates the translation of the OTP FSM design pattern into μCRL.

5.1 Simulating state management

When translating an Erlang FSM program into μCRL, the first thing one needs to consider is how to maintain the FSM states and data. In particular, a scheme needs to be defined to store and update the *current state* and the *state data* in μCRL. Normally a global variable would be used to perform such a task, however, μCRL does not support the use of global variables. Thus we use a (one place) stack for simulating the management of states and data as it has well been defined in μCRL. Alternatively, one might define some other mechanics such as data buffer for state and datum management.

```
act
 s_event, r_event, send_event: Term

comm
 s_event | r_event = send_event

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

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

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

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

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

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

Fig. 5. Rules for translating state processes.

The translation rules are defined in Figure 5, 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 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 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. Rules for the translation of Erlang state functions are discussed in Section 5.2.

Having performed all actions, a state process ends up by calling the process *fsm_next_state*(*next_state*, *new_data*), updating the *current state* and the *state data* in stack. The process *CommandList*($CmdList : Term$) is defined to simulate the behaviour of the external actions. A list of commands is initialized in $CmdList$, where commands in the list define the logic for verification. The process *fsm_next_state* will synchronize with $CommandList$ through the actions *r_command* and *s_command*, *r_command* | *s_command* = *cmd*. Each time, *fsm_next_state* reads the head of $CmdList$, and communication terminates when $CmdList$ is empty.

5.2 Translating the state functions

An Erlang state function may consist a list of branches, each of which defines a sequence of actions to be performed. A branch is usually guarded by a pattern, and only when the function arguments match the pattern of its guards, can a branch be selected for execution. Thus in the door locking example above, the state function *locked* defines a number of actions (*do_unlock, display_message*) which are selected depending on the value of the password inputted.

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

A: Matching.

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

B: Guards.

Fig. 6. Guarded Erlang programs.

In general there are two ways in which such pattern matching can be defined, and Figure 6 illustrates an example where the program in Figure 6-A is written using pattern matching, while, in Figure 6-B, with a set of guards. When

N matches P_i, the action sequence *action*(i) is enabled. In general, overlapping might exist between patterns P_i and P_j, and only the first matched action sequence *action*(i) will be performed.

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 and guards occur in the function, the pre-defined action sets are uniquely indexed with a set of integers. For example, in Figure 6, the set of action sequences {*actions*(1), ..., *actions*(n)} is indexed with an integer set {1, ..., n} where integer i identifies the pre-defined action set *actions*(i).

The selection of a μCRL state function for execution is determined by the pattern of function arguments. By the end, the 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 7 illustrates an example for the state functions shown in Figure 6.

```
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)...)
```

Fig. 7. Translation rules for Erlang state function.

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.

6 Case studies

To illustrate the approach we present two case studies, one of which is a door with code lock system, while, the other a coffee machine system. As discussed in Section 2, *gen_fsm*:*send_event* is often called through some external actions. Therefore, before starting a simulation process, a sequence of actions needs to be initialized in the process *CommandList* to simulate the external behaviour.

6.1 A door with code lock

Consider the example given in Section 2. In the simulation, the system code is set to *abc*. The function *button* is defined to input a password.

Following the rules defined in Section 5, the OTP component is translated into μCRL, and the resultant μCRL specification is listed in the appendix. A sequence of external actions $[\{abb\}, \{abc\}]$ is initialized in the μCRL specification, stating that two passwords, *abb* and *abc*, are consecutively inputted. The LTSs derived from CADP are shown in Figure 8 where Figure 8-A lists all actions, while, Figure 8-B hides the actions *push_callstack* and *pop_callstack* as internal actions.

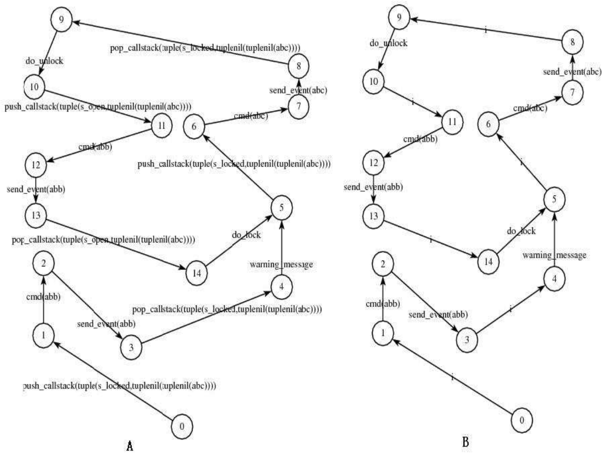

9
pop_callstack(tuple(s_locked,tuplenil(tuplenil(abc))))
do_unlock
8
10
send_event(abc)
push_callstack(tuple(s_open,tuplenil(tuplenil(abc))))
7
11
6
cmd(abc)
cmd(abb)
12
push_callstack(tuple(s_locked,tuplenil(tuplenil(abc))))
send_event(abb)
13
5
pop_callstack(tuple(s_open,tuplenil(tuplenil(abc))))
do_lock
2
14
warning_message
4
cmd(abb)
pop_callstack(tuple(s_locked,tuplenil(tuplenil(abc))))
send_event(abb)
1
3
push_callstack(tuple(s_locked,tuplenil(tuplenil(abc))))
0
A
9
i
do_unlock
8
send_event(abc)
10
7
i
cmd(abc)
11
6
cmd(abb)
i
12
send_event(abb)
5
13
i
do_lock
14
warning_message
2
4
cmd(abb)
send_event(abb)
i
1
3
i
0
B

Fig. 8. LTSs derived from the door with code lock system.

From the LTSs it can be seen that, initially, the system pushes the state *s_locked* and the code *abc* onto the stack. This simulates the *start_link* function in the Erlang program where the initial state and the system code are set to *s_locked* and *abc* respectively. When the action *send_event* is performed, the state *s_locked*, saved in the stack, is read out. The state *s_locked* determines process *fsm_locked* is about to be activated. This simulates the process that the current state function is executed when *gen_fsm* : *send_event* is invoked. Since the first password is not correct, $abb \neq abc$, a warning message is given and the door remains locked. After *abc* is received, the door is opened and the state *s_open* is pushed onto the stack.

We can then use a toolset such as CADP to verifying design properties of the system. For instance, to check "without receiving a correct password "*abc*", the door cannot be opened", the property can be formulated as:

[not ("cmd(abc)")* . "do_unlock"] false,

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

<true*. "cmd(abb)" . ("pop_calls(tuple(s_locked,tuplenil (tuplenil(abc))))")* .
"warning_message"> true,

stating that when an incorrect password "*abb*" is received and the current state is *s_lokced*, the action *warning_message* will be fairly performed. Thus once we have a specification in μCRL, applying model-checking approaches is standard.

However, the example given in this section is simple and the system is comparatively easy to be verified. In the next sub-section, a more complicated system is designed to further evaluate the proposed model.

6.2 Coffee machine

A coffee machine has three states, these being, *selection*, *payment* and *remove*. State *selection* allows a buyer to choose the type of drink, while, state *payment* displays the price of a selected drink and requires payment for the drink; after enough coins being paid, 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 select a type of drink at a machine, pays for it and takes a cup after the drink is ready. A buyer can also cancel the current transaction where the pre-paid coins will be returned.

Figure 9 illustrates the FSM design of the coffee machine. The program initially sets the current state to *selection*.

The OTP component is then translated into μCRL, and 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 change if more coins have been paid for the drink, or gives back the pre-paid coins if the transaction is cancelled.

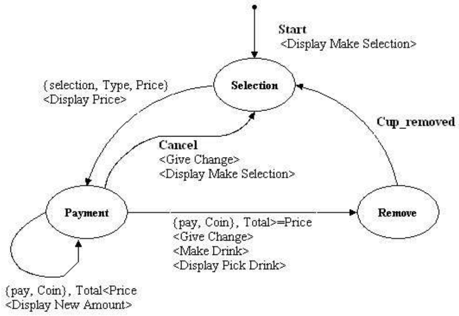


Fig. 9. FSM - coffee machine.

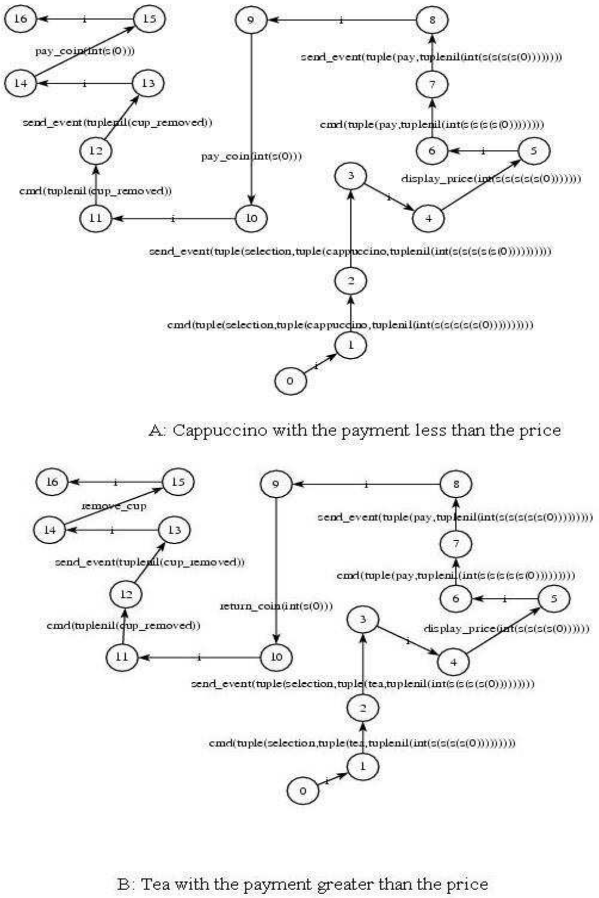
Before verifying the system's properties, a set of verification tasks is required, each of which consists of a set of commands to simulates the process on buying a drink. Two sequences of external actions are constructed. The first simulates "selecting *cappuccino* (£5 for a cup), paying £4 and then 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}]*. They are then initialized in the process *CommandList* respectively.

The LTSs, derived from the CADP, are shown in Figure 10. Figure 10-A shows that the system initially pushes *s_selection* onto the stack. Once *cappuccino* is selected, its price is displayed. When a buyer pays less coins (£4) than the price (£5), the machine stays in *payment*, asking for the rest of payment (£1). Figure 10-B shows that, after *tea* (£4 for a cup) is selected and more coins (£5) are paid, the machine will prepare the drink and returns the change (£1). When the drink is taken away, the machine moves back to *selection*.

System properties can then be verified by the CADP model checker. For example, to check the property "After *cappuccino* is selected, its price will be displayed.", the property can be formulated as:

```
[true*. "cmd(tuple(selection,tuple(cappuccino,tuplenil (5))))" . (not "display_price(5)")*]
 <true* . "display_price(5)"> true
```

Similarly, to check the properties "When *cappuccino* is selected and £4 has been paid, if the rest of payment £1 is not paid, the drink cannot be taken away.", and "When *tea* is selected and £5 has been paid, before the drink being taken away, change must be returned.", we formulated them as (respectively):


16
i
15
pay_coin(int(s(0)))
14
i
13
send_event(tuplenil(cup_removed))
12
cmd(tuplenil(cup_removed))
11
i
10
pay_coin(int(s(0)))
9
i
8
send_event(tuple(pay,tuplenil(int(s(s(s(s(0))))))))
7
cmd(tuple(pay,tuplenil(int(s(s(s(s(0))))))))
6
i
5
display_price(int(s(s(s(s(s(0)))))))
3
i
4
send_event(tuple(selection,tuple(cappuccino,tuplenil(int(s(s(s(s(s(0))))))))))
2
cmd(tuple(selection,tuple(cappuccino,tuplenil(int(s(s(s(s(s(0))))))))))
1
i
0
16
i
15
remove_cup
14
i
13
send_event(tuplenil(cup_removed))
12
cmd(tuplenil(cup_removed))
11
i
10
return_coin(int(s(0)))
9
i
8
send_event(tuple(pay,tuplenil(int(s(s(s(s(s(0)))))))))
7
cmd(tuple(pay,tuplenil(int(s(s(s(s(s(0)))))))))
6
i
5
display_price(int(s(s(s(s(0))))))
3
i
4
send_event(tuple(selection,tuple(tea,tuplenil(int(s(s(s(s(0)))))))))
2
cmd(tuple(selection,tuple(tea,tuplenil(int(s(s(s(s(0)))))))))
1
i
0

A: Cappuccino with the payment less than the price

B: Tea with the payment greater than the price

Fig. 10. LTSs derived from the coffee_machine system.

[true* . (‘cmd(tuple(selection,tuple(cappuccino,tuplenil (5)))) . *’ and ‘cmd(tuple(pay, tuplenil(4))). *’). (not “pay_coin(1)”)* . “cmd(tuplenil(cup_removed))”] false

[true* . (‘cmd(tuple(selection,tuple(tea,tuplenil(4)))) . *’ and ‘cmd(tuple(pay, tuplenil(5))). *’). (not “return_coin(1)”)* . “cmd(tuplenil(cup_removed))”] false

We applied the translation approach to a faulty implementation to evaluate the model's capability for fault detection. In stead of using *payment* $\geq$ *price*, the faulty Erlang program implements the logic *payment* $>$ *price* for selling a drink. The faulty Erlang program is then translated into μCRL.

A sequence of actions, $[\{selection, cappuccino, 5\}, \{pay, 5\}, \{cup\_remove\}]$ is constructed to simulate the external behaviour of “paying exactly £5 for a cup of cappuccino (£5 for a cup)". The LTS derived from the CADP toolset is shown in Figure 11. It can be seen the machine requires additional £0 for the drink, even though enough money has been paid.

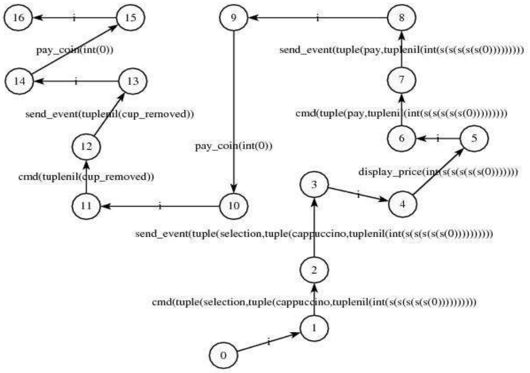

16
15
i
9
8
i
pay_coin(int(0))
send_event(tuple(pay,tuplenil(int(s(s(s(s(s(0))))))))))
14
13
i
7
send_event(tuplenil(cup_removed))
cmd(tuple(pay,tuplenil(int(s(s(s(s(s(0))))))))))
12
pay_coin(int(0))
6
i
5
cmd(tuplenil(cup_removed))
display_price(int(s(s(s(s(s(0)))))))
3
i
11
i
10
4
send_event(tuple(selection,tuple(cappuccino,tuplenil(int(s(s(s(s(s(0)))))))))))
2
cmd(tuple(selection,tuple(cappuccino,tuplenil(int(s(s(s(s(s(0)))))))))))
1
i
0

Fig. 11. LTSs derived from the faulty Erlang program.

We then checked the derived model against the property:

["cmd(tuple(selection,tuple(cappuccino,tuplenil(5))))"* . "cmd(tuple(pay,tuplenil(5)))"* .
(not "remove_cup")*] <true* . "remove_cup"> true

stating that, when cappuccino is selected and after £5 has been paid, the drink will be prepared. Using this property the CADP model checker can correctly distinguish the correct and faulty implementations based upon the design we wish to check against.

7 Conclusions and future work

In this paper we have extended work on model checking Erlang in μCRL. The principal aim of the work is to define rules that will translate Erlang/OTP programs (assumed to be correctly implemented) into a μCRL specification, and then to verify properties that the system should hold with standard toolsets such as CADP. We have extended previous work by investigated the model checking of Erlang/OTP Finite State Machine components in the process algebra μCRL. Specifically, a model was proposed to support the translation of an Erlang FSM design pattern into a μCRL specification, where a stack is defined in μCRL to simulate the management of the FSM states and the up-to-date state data.

The particular challenge is not the writing of a FSM in a process algebra, which is, of course, trivial, but the correct translation of how Erlang treats and defines FSMs, and the parameters with which it can be invoked. Furthermore, the translation needs to be faithful to the translation of other OTP components, that is, maintain the same design philosophy, and specifically the level of abstraction of the mapping from Erlang to μCRL.

Here, the state function defined in the Erlang FSM is translated into two parts in μCRL, one of which defines a μCRL state-process that can be called or synchronised by some other μCRL processes, while, the other defines a series of μCRL state functions determined by the patterns defined in the Erlang state function. A sequence of actions defined in an Erlang state function and guarded by a pattern is translated into a pre-defined action set in μCRL indexed with a unique integer number. A μCRL state-process will receive an index number from a μCRL state function that determines which pre-defined action set will be triggered.

Two small examples illustrate the proposed model, one of which looked at a door with code lock system while the other studied a coffee machine system. Both systems were modelled by Erlang/OTP *gen_fsm* design pattern first, and then translated into a μCRL specification. By using a model checker such as CADP, properties can be verified which represent an abstraction over the original Erlang code.

The algorithm presented performs an abstraction of the Erlang code, and is currently being implemented and integrated into the *etomcrl* toolset so that complex OTP designs involving generic servers, FSMs etc can be translated. There are a number of issues that we have not had space to discuss here. One is correctness of the translation, which is involved as it depends on verification against a semantics of Erlang. Such issues of correctness of the approach are discussed in [5]. The other issue is that the model discussed in this paper does not define rules for the translation of *timeout* events. However, in some real applications, *timeout* events in a FSM play a significant role in the OTP design, and there are two approaches to extending the work we have presented here. The first is to use a timed extension to μCRL (which exist, but have limited tool support), the second is to incorporate explicit *tick* events in the untimed μCRL. We have recently experimented successfully with the second approach, and again the translation produces tractable μCRL specifications.

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. Thanks also go to my supervisor, John Derrick, for his help with this work.

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 Alexander Lindsay, editors, *In Proc. 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. *Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 135–144, 2004.
4. J. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund. AXD 301: A new generation ATM switching system. *Computer Networks*, 31:559–582, 1999.
5. C. Benac Earle. *Model check the interaction of Erlang components.* PhD thesis, The University of Kent, Canterbury, Department of Computer Science, 2006.
6. Lars-Ake Fredlund C. Benac Earle 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, 2005.
7. CADP. *http://www.inrialpes.fr/vasy/cadp/.*
8. E. Clarke, O. Grumberg, and D. Long. *Model Checking.* MIT Press, 1999.
9. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
10. C. Benac Earle and Lars-Ake Fredlund. Verification of Language Based Fault-Tolerance. In *EUROCAST*, pages 140–149, 2005.
11. F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999.
12. Lars-Ake 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.*, 4:405–420, 2003.
13. J. F. Groote and A.Ponse. The syntax and sematics of μCRL. In *Algebra of Communicating Processes 1994, Workshop in Computing*, pages 26–62, 1995.
14. J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. Report P9008, University of Amsterdam, 1990.
15. J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.
16. Q. Guo and J. Derrick. Eliminating overlapping of pattern matching when verifying Erlang programs in μCRL. In *the 12th International Erlang User Conference (EUC'06), Stockholm, Sweden*, 2006.

Appendix: The μCRL specification for code lock door

sort
  Term

func
  s_locked, s_open, abc, abb: -> Term

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

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, LoopData: Term
  Pattern1, Pattern2: Term

rew
  locked(Command, LoopData) =
     locked_case_0_0(patterns_matching(Command, element(int (1),LoopData)),
Command, LoopData)
  locked_case_0_0(true, Command, LoopData) =
      tuple(s_open, tuple(LoopData, tuplenil(tuplenil(int(1)))))
  locked_case_0_0(false, Command, LoopData) =
        locked_case_0_1(patterns_matching(Command, do_not _care), Command,
LoopData)
  locked_case_0_1(true, Command, LoopData) =
      tuple(s_locked, tuple(LoopData,tuplenil(tuplenil(int(2)))))
  open(Command, LoopData) =
      tuple(s_locked, tuple(LoopData,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))
              ◁ is_s_locked(element(int(1),Val)) ▷
                    ( fsm_open(Command,element(int(2),Val))
                          ◁is_s_open(element(int(1),Val)) ▷ delta)))
  fsm_locked(Command:Term,LoopData:Term) =
      ( do_unlock.
        fsm_next_state(element(int(1),locked(Command,LoopData)),
                              element(int(2),locked (Command,LoopData))))
                    ◁term_to_bool(equal(element(int(1),element(int(3),
                            locked(Command,LoopData))),int(1)))▷
                            ( warning_message.
                  fsm_next_state(element(int (1),locked(Command,LoopData)),
                                  element(int (2),locked(Command,LoopData)))
                              ◁ term_to_bool(equal (element(int(1),element(int(3),
                                         locked (Command,LoopData))),int(2)))
▷ delta)
  fsm_open(Command:Term,LoopData:Term) =
     do_lock.
     fsm_next_state(element(int(1),open(Command,LoopData)),
                        element(int(2),open (Command,LoopData)))
          ◁ term_to_bool(equal(element(int(1),element (int(3),
                        open(Command,LoopData))),int(1))) ▷ delta
  fsm_change_state =
     sum(Command:Term,r_event(Command).read(Command))

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

  fsm_next_state(S:Term, LoopData:Term) =
     wcallresult(tuple(S,tuplenil(LoopData))).
     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)))
```