Model-Checking Erlang – A Comparison between EtomCRL2 and McErlang

Qiang Guo[1], John Derrick[1], Clara Benac Earle[2], and Lars-Åke Fredlund[2]

[1] Department of Computer Science,
The University of Sheffield,
Regent Court, 211 Portobello, S1 4DP, UK
{Q.Guo,J.Derrick}@dcs.shef.ac.uk
[2] Facultad de Informática,
Universidad Politécnica de Madrid
Boadilla del Monte 28660, Madrid, Spain
{cbenac,fred}@babel.ls.fi.upm.es

Abstract. Model-checking programs is important in the development of a reliable software system. Two approaches might be applied to model-check a system at a source code level. One is to directly apply model-checking algorithm to the programming language; the other to abstract the program source codes into a formal specification, upon which some standard model-checkers can be used to verify system's properties. Both methods have recently been investigated for model-checking the functional programming language Erlang. Correspondingly, two Erlang model-checkers McErlang and Etomcrl2 are developed. This paper evaluates the two model-checkers by applying them to verify a a distributed and concurrent example - telecoms implemented in Erlang/OTP. A number of system's key properties are model-checked with both tool-sets. Advantages and disadvantages upon the uses of Etomcrl2 and McErlang are compared and summarized. Through such a case study, we intend to evaluate the two model-checkers on their effectiveness when verifying distributed and concurrent systems, and propose suggestions for their future work.

Keywords: Erlang, Model Checking, Program Source Code, Etomcrl2, McErlang.

1 Introduction

Model checking has been widely used in system design and verification. It has become a standard technique at both the design level as well as for finite-state hardware components, and much recent research has been concerned with extending its applicability to programming languages. This is increasingly necessary since as the complexity of systems grow, implementations of concurrent and distributed systems sometimes contain fatal errors such as deadlocks, despite the existence of careful designs. One example is demonstrated in the analysis of

L. Bottaci and G. Fraser (Eds.): TAIC PART 2010, LNCS 6303, pp. 23–38, 2010.
© Springer-Verlag Berlin Heidelberg 2010

NASA's Remote Agent Spacecraft Control System [15]. Thus to derive a reliable system, it is essential not only to verify the system design, but also to model check its implementations. This paper reviews and compares the existing techniques that have been applied to model-check applications written in the functional programming language Erlang [7] at a source code level.

There are essentially two approaches to model checking Erlang applications at a source code level. The first is to directly implement verification algorithms to the Erlang programming language; the other to abstract the Erlang programs into a formal specification, upon which a standard model checker can be applied to verify the system's properties. Both methods have advantages and disadvantages. The first approach requires less effort in computing the state space. However, developing verification algorithms is usually hard and time consuming. The second approach verifies the system's properties with the support of some existing tool-sets (model checkers). These tool-sets are usually standard and optimised, and thus efficient to use. However, in order to make use of a model checker, one has to model every aspects of Erlang and the Open Telecom Platform (OTP) components in a formal specification language that is supported by the model checker.

Fredlund *et al.* [8] investigate the first approach and develop a tool-set McErlang to model-check distributed systems written in Erlang. The tool-set McErlang makes use of a standard on-the-fly depth-first model checking algorithm [16] where the properties under evaluation can be represented by Büchi automata. The Erlang tool LTL2Buchi [17] is used to translate a Linear Temporal Logic (LTL) formulas into a Büchi monitor.

Arts *et al.* [2] initiate the strand of the second approach where Erlang and the OTP components are modeled in the process algebra μCRL [10] and verified via the standard model checker CADP [6]. A set of rules is defined to abstract the behaviour of Erlang syntax, OTP *gen_server*, *supervisor* and *gen_fsm* into μCRL. A small tool-set Etomcrl [2] is developed to automate the process of translation. Guo *et al.* extend the work by defining rules to model OTP *gen_fsm* [11] and the Erlang *timeout* events [12] in μCRL. Guo *et al.* [13] further study the ways to transform the existing rules to a set of new rules that is able to model Erlang and the corresponding OTP components in mCRL2 [9]. mCRL2 is a new version of μCRL that is extended with higher-order data-types, standard data-types, multi-actions and local communication. Compared to μCRL, mCRL2 is more applicable in practice. The tool-set Etomcrl is upgraded to Etomcrl2 where mCRL2 is used as the formal specification language for system modeling.

This work evaluates and compares the Erlang model-checker Etomcrl2 and McErlang. A telecoms case study is designed with a server-client infrastructure and implemented making use of Erlang OTP design patterns. A number of system's key properties is verified via Etomcrl2/CADP and McErlang respectively. Experimental results suggest that both model-checkers are effective in verifying the majority of system properties; both are able to distinguish the faulty

implementations from the design. A number of limitations on the uses of the tool-sets are summarised. Etomcrl2 has to make use of a third-party toolset such as CADP to model-check an Erlang application. In order to make Etomcrl2 a mature toolset, modeling of all OTP components in mCRL2 are necessary; McErlang is unable to model-check *timeout* related properties due to it implements neither a discrete nor a real-time semantics. To improve the usability of the toolset, timing scheme needs to be developed. Through such a case study, we intend to provide suggestions for their future work.

The rest of paper is organised as follows: Section 2 introduces Erlang and OTP with a telecoms example; Section 3 reviews the existing Erlang model-checkers Etomcrl2 and McErlang; Section 4 applies Etomcrl2 and McErlang to model-check our telecoms example; Section 5 compares the model checker Etomcrl2 and McErlang; the work is summarised in Section 6.

2 Telecoms: An Illustration of Erlang Application

2.1 An Introduction to Erlang and OTP

Erlang [7] is a concurrent functional language with specific support for the development of distributed,fault-tolerant systems with soft real-time requirements. It was designed from the start to support a concurrency-oriented programming paradigm and large distributed implementations that this supports. The Open Telecom Platform (OTP) is a set of Erlang libraries for building large fault-tolerant distributed applications. With the OTP, Erlang applications can be rapidly developed and deployed across a large variety of hardware platforms, and this has caused it to become increasingly popular, not only within large telecoms companies such as Ericsson, but also with a variety of SMEs in different areas such as Yahoo! Delicious, and the Facebook chat system [7].

To further illustrate how a distributed and concurrent system is constructed using Erlang and OTP, in the following subsections, we demonstrate a telecoms example that is designed and implemented making use of Erlang OTP design patterns.

2.2 The Case Study

The telecoms system is designed using a client-server structure. It configures a number of functional servers (FS) to process clients' requests. Each FS is defined with a capacity that specifies the maximum number of mobile phones to be connected.

A client can communicate with any FSs and perform some functional operations such as *calling* and *top-up*. Each client has an account maintained by the system. In order to make a phone call, a client needs to preset enough money in its account. Before performing any functional operations, a client needs to connect

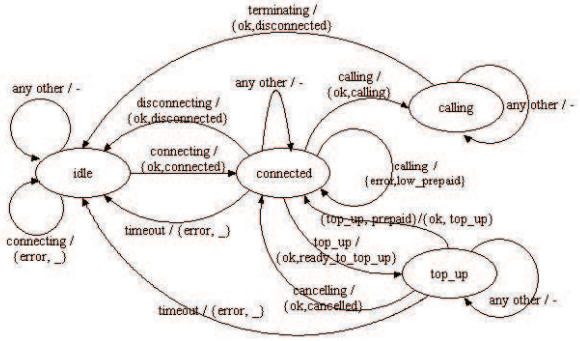

Fig. 1. The mobile phone behaviour modeled as an FSM

to an FS. A client can only be connected to one FS, and if a client has connected to an FS and tries to connect to another FS, the request will be denied.

The behaviour of a client (mobile phone) is modeled as a finite state machine (FSM), and the initial design is shown in Figure 1. There are four states: *idle*, *connected*, *calling* and *top_up*, where initially, the system is set to the *idle* state. The FSM defines the behaviour of a number of operations: *connecting*, *disconnecting*, *calling*, *terminating*, *top_up* and *canceling*. Before performing any operations, a client FSM needs to connect to an FS through sending the *connecting* request.

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

2.3 Erlang Implementations

The telecoms example is implemented, making use of the OTP design patterns as is common practice. The FS is implemented using the Erlang/OTP *gen_server* module. A generic server is implemented by providing a *callback module* where (*callback*) functions are defined to specify the concrete actions such as server state handling and response to messages.

The client behaviour is realized using the OTP *gen_fsm* module. In accordance with the design, four state functions are defined: *idle*, *connected*, *calling* and *top_up*.

The state function *idle* initiates a *connecting* request to an FS. If the FS replies the FSM with {*ok,connected*}, the request is accepted and the connection is set up. The FSM moves to the state *connected*; otherwise, the request is denied and the FSM remains unchanged.

```
idle(AT,{MB,RS,CSs})→                                  :     action:show({MB,already_connected}),
   PT1 = gen_server:                                   :     {next_state,connected,
        call(hd(CSs),{request,AT,MB}),                 :                {MB,RS,CSs},20000};
   case PT1 of                                         :  {error,busy}→
     {error,invalid_mobile}→                           :     action:show({RS,sever_busy}),
        action:show({MB,invalid}),                     :      idle(AT,{MB,RS,
        {next_state,idle,{MB,RS,CSs}};                 :         lists:append(tl(CSs),[hd(CSs)])});
     {ok,connected,CalledFS}→                          :  _Other→
        action:show({MB,connected,CalledFS}),:             action:show({AT,invalid}),}
        {next_state,connected,                         :    {next_state,idle,{MB,RS,CSs}}
                {MB,CalledFS,CSs}, 20000};             :  end.
     {error,already_connected}→                        :
```

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

```
connected(timeout,{MB,RS,CSs})→                        :       action:show({MB,calling_enabled}),
   gen_server:call(RS,{request,timeout,MB}),:              {next_state,calling,
   action:show({MB,timeout}),                          :              {MB,RS,CSs}};
   {next_state,idle,{MB,nil,CSs}};                     :    {error,low_prepaid}→
connected(AT,{MB,RS,CSs})→                             :       action:show({MB,low_prepaid}),
   case AT==terminating of                             :       {next_state,connected,
     true →                                            :              {MB,RS,CSs},20000};
       action:show({AT,invalid}),                      :     {ok,ready_to_top_up}→
       {next_state,connected,                          :       action:show({MB,ready_to_top_up}),
            {MB,RS,CSs},20000};                        :       {next_state,top_up,
     false →                                           :              {MB,RS,CSs},20000};
       Flag=gen_server:call(RS,{request,AT,MB}):           _Other →
       case Flag of                                    :       action:show({MB,invalid}),
         {ok,disconnected}→                            :       {next_state,connected,
           action:show({MB,disconnected}),             :              {MB,RS,CSs},20000}
           {next_state,idle, {MB,RS,CSs}};             :   end
         {ok,calling_enabled}→                         : end.
```

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

```
calling(AT,{MB,RS,CSs})→                               :              {MB,nil,CSs}};
   case AT of                                          :    _Other →
      terminating →                                    :       action:show({AT,invalid}),
         gen_server:call(RS,{request,AT,MB}),:            {next_state,calling,
         action:show({MB,call_terminating}),           :              {MB,RS,CSs}}
         {next_state,idle,                             : end.
```

When being connected to an FS, the client can ask to top up its account by sending the *top_up* request to the FS. If {*ok,ready_to_top_up*} is replied, the top up process is enabled, and the FSM moves to the state *top_up*. An action will

trigger the state function *top_up* to either start the transaction by {*top_up, Prepaid*} operation (*Prepaid* is the amount of money the client is about to transfer), or cancel the process by sending the *canceling* request.

```
top_up(timeout,{MB,RS,CSs})→                    :  {ok,cancelled} →
  gen_server:call(RS,{request,timeout,MB}),:           action:show({MB,top_up_cancelled}),
  action:show(MB,timeout),                   :         {next_state,connected,
  {next_state,idle,{MB,nil,CSs}};            :               {MB,RS,CSs},20000};
top_up(AT,{MB,RS,CSs})→                          :    _Other →
  case gen_server:call(SVR,{request,AT,MB}) of:          action:show({AT,invalid}),
    {ok,top_up} →                            :         {next_state,top_up,
      action:show({MB,top_up_ready}),        :               {MB,RS,CSs},20000}
      {next_state,connected,                 : end.
           {MB,RS,CSs},20000};               :
```

When an FSM moves to the state *connected* and *top_up*, a timer is initiated. The timer is set to 20,000ms. If within the time period, no action is performed, a *timeout* event will be generated and sent to the FS. The FSM is reset to the state *idle*.

3 Erlang Model-Checking Tool-Sets

This section reviews the existing Erlang model checker Etomcrl2 and McErlang.

3.1 Etomcrl2

Etomcrl2 [13] is a tool-set that automatically translates the source codes of an Erlang application into an mCRL2 [9] specification, upon which the standard model checker CADP [6] is used to generate a (finite) state space to check the system properties against the designs. The process algebra μCRL (micro Common Representation Language) [10] is an extension of the process algebra ACP [3]. It was developed with equational *abstract data types* being integrated into the process specification, which enables the specification of both data and process behaviour. The language [1]mCRL2 is a new version of μCRL that is extended with higher-order data-types, standard data-types, multi-actions and local communication. Compared to μCRL, mCRL2 is more applicable in practice.

The tool-set Etomcrl2 is comprised of three functional modules: the *Preprocess* module, the *mCRL2 translation* module and the *mCRL2 initialization* module. These functional modules work together to automatically translate the source codes of an Erlang application into an mCRL2 specification, upon which the standard model checker CADP is used to check system's properties.

3.2 McErlang

McErlang [8] is a tool-set that is developed to model-check Erlang programs, particularly concurrent applications. The main idea behind McErlang is to re-use as

[1] Instead of μCRL2, the newly release language is named as mCRL2.

much of a normal Erlang programming language implementation as possible, but adding a model checking capability. To do so, the tool-set replaces the part of the Erlang runtime system that implements concurrency and message passing without modifying the runtime system for the evaluation of sequential executions.

The tool-set takes an Erlang function as its input. This function specifies the entry of the Erlang application under verification, a call-back module (written in Erlang) that defines the behavioural safety property[2] to be checked (called the *monitor*), and the algorithm used to check the property. When a property is checked with McErlang, the tool-set either returns a positive reply, confirming that property holds, or a negative one with a counterexample (a trace leading to the problem state).

McErlang also supports model checking programs against full Linear Temporal Logic (LTL) formulas. The LTL2Buchi tool [17] is used to translate an LTL formulas into a Büchi monitor, which are then checked using a standard on-the-fly depth-first model checking algorithm [16].

4 Illustrative Example – The Tools in Use

There are two groups of experiments. In the first group, a number of properties are checked against the implementationd and, in the second, two types of faulty implementations are constructed to examine the capability of the model-checkers on fault detection.

To instantiate the simulation process, we configure the system with three FSs (svr_1, svr_2 and svr_3) and five clients (m_1, m_2, m_3, m_4 and m_5). The capacity of every FS is set to 1 and the minimal cost for making a call is set to £2. Here, we define that, when the system is modeled with an mCRL2 specification (using Etomcrl2), the passing of one time unit is specified as 10,000ms, represented by one *tick* action.

4.1 Property Verification

We first devise two experiments to verify the properties on making a call. In the first experiment, the client m_1 attempts to make a phone call with its account being preset with £1; while, in the second, m_1 tries to make a call with its account being preset with £3. In both experiments, all other clients are idle. Through these two experiments, we intend to check (1) whether the communication between FS(s) and the client(s) is running correctly; (2) whether the logics of making a call extracted from the behaviour of the FS(s) and the client(s) comply with their designs; and (3) the logics of *timeout* event have been correctly implemented.

These properties are first verified using Etomcrl2 and CADP. The Labeled Transition Systems (LTSs) derived from the experiments are shown in Figure 2 and Figure 3 respectively. It can be seen that, in both experiments,both LTSs present the logics that comply with the designs.

[2] A safety property expresses that nothing bad ever happens, which can be expressed as “always(not P)” in linear temporal logic (where “P” is the bad event).

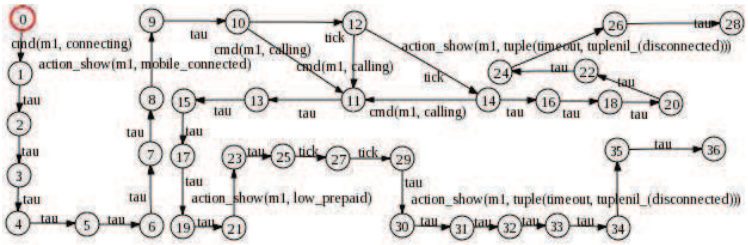

Fig. 2. LTS: m_1 tries to make a call with low prepaid and the request is denied

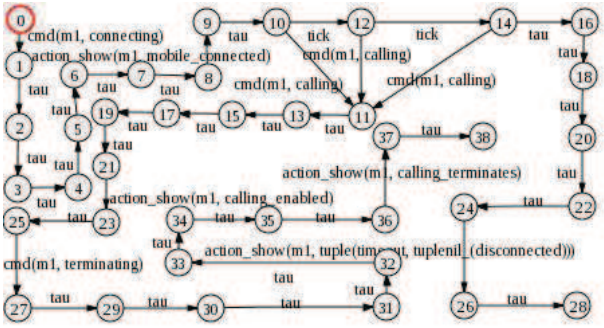

Fig. 3. LTS: m_1 tries to make a call with enough prepaid and the request is enabled

The system properties can be formalized with a set of LTL formulas. For example, in the above experiments, the property "without being connected to an FS, m_1 cannot make a phone call." is formalized as:

[not(action_show(m_1, mobile_connected))*. action_show(m_1, calling_enabled))] false

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

[true*. action_show(m_1, mobile_connected)*]
<not('tick.tick')*. action_show(m_1, tuple(timeout,tuplenil(disconnected)))> false

By applying the formulas to CADP, verification of the system properties can be automated.

The above properties are then verified using McErlang. Since McErlang is not capable of checking the *timeout* event, we will only examine the properties of communication between FS(s) and the client(s) and the logics of making a call. Before the experiments start, a number of transition labels has been inserted to the system's source codes using McErlang *mce_erl:probe* function. McErlang provides the ability to visualize LTSs using the graphviz set of drawing tools. In the following experiments, however, we will only report the verification results.

First, we will check the connection relation between client m_1 and the FSs. The property is defined as "without being connected to an FS, the functional operation *calling* performed by m_1 is invalid" and constructed in McErlang as shown:

```
property1_1() →
  mce:start(#mce_opts
    {program = {action,startSimulation,[[{[m_1,m_2,m_3,m_4,m_5],[1,2,3,4,5]},
                                          [svr_1,svr_2,svr_3],2,3]]},
     monitor = {mce_ltl_parse:ltl_string2module_and_load(
                  "always(((not P) and Q) ⇒ eventually R)",messenger_mon),
                  {void,[{'P',basicPredicates:show_message({m_1,mobile_connected})},
                         {'Q',basicPredicates:receive_cmd({calling,m_1})},
                         {'R',basicPredicates:show_message({m_1,action_invalid})}]}},
          algorithm = {mce_alg_buechi,void}}).
```

We then evaluate "after m_1 is connected to an FS and tries to make a phone call, the request will be denied with a reply *low_prepaid*". The property is defined in a verification run as:

```
property1_2() →
  mce:start(#mce_opts
    {program = {action,startSimulation,[[{[m_1,m_2,m_3,m_4,m_5],[1,2,3,4,5]},
                                          [svr_1,svr_2,svr_3],2,3]]},
     monitor = {mce_ltl_parse:ltl_string2module_and_load(
                  "always(P and Q) ⇒ eventually R)",messenger_mon),
                  {void,[{'P',basicPredicates:show_message({m_1,mobile_connected})},
                         {'Q',basicPredicates:receive_cmd({calling,m_1})},
                         {'R',basicPredicates:show_message({m_1,low_prepaid})}]}},
                  algorithm = {mce_alg_buechi,void}}).
```

After running the checks of these two properties in McErlang, the tool-set returns "Execution terminated normally.", with total 1377 and 18201 states being explored respectively. The experimental results imply that both properties are held in the implementation.

Next, we construct an experiment to examine the system's behaviour where more than one clients are active. Two clients m_1 and m_2 request to connect to a FS simultaneously. Since the capacity of the FS is set to 1, according to the design, when an FS, for example svr_1, accepts the request of a client, say m_1, it should reply the other m_2 with *server_busy*; the client m_2 should afterwards request a connection to another FS, say svr_2.

The property is first checked using Etomcrl2 and CADP. The LTS derived from the experiment is illustrated in Figure 4. The graph is symmetric and shows that if m_1 is firstly connected to an FS and m_2 requests to connect to the same FS, m_2 will receive a reply *server_busy*. After trying a different FS, m_2 is connected to the FS; or, if m_2 is firstly connected to an FS and m_1 requests to connect to the same FS, m_1 will receive a reply *server_busy*. After trying a different FS, m_1 is connected to the FS. The logics extracted from the LTS comply with the system design.

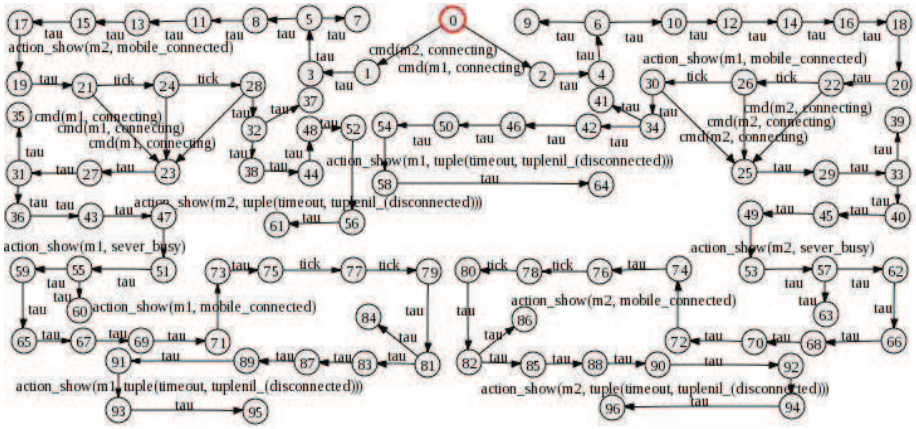

Fig. 4. LTS: m_1 and m_2 request to connect to an FS simultaneously with the capacity of svr_1 is set to 1

A number of properties can then be automatically verified via CADP. For example, to check "when m_1 is connected to an FS and m_2 requests to connect to the same FS, m_1 will receive reply *server_busy*.". The property is formalized as:

<true*. action_show(m_1, mobile_connected) *. cmd(m_2, connecting) *. action_show(m_2, server_busy)> true

Another property we want to check is formalized as:

<true*. cmd(m_2, connecting) *. action_show(m_2, server_busy) *. cmd(m_2, connecting) *. action_show(m_2, mobile_connected)> true

stating that "when m_2 requests to connect to an FS and receives the reply of *server_busy*, it will request to connect to another FS and its request will be accepted."

The property is then verified using McErlang. The above two properties are configured as:

```
property2_1() →
  mce:start(#mce_opts
     {program = {action,startSimulation,[[{[m_1,m_2,m_3,m_4,m_5],[1,2,3,4,5]},
                                [svr_1,svr_2,svr_3],1,3]]},
     monitor = {mce_ltl_parse:ltl_string2module_and_load(
                    "always((O and P) and Q) ⇒ eventually R)",messenger_mon),
                    {void,[{'O',basicPredicates:receive_cmd({connecting,m_1})},
                           {'P',basicPredicates:show_message({m_1,mobile_connected})},
                           {'Q',basicPredicates:receive_cmd({connecting,m_2})},
                           {'R',basicPredicates:show_message({m_2,server_busy})}]}},
                    algorithm = {mce_alg_buechi,void}}).
```

```
property2_2() →
  mce:start(#mce_opts
     {program = {action,startSimulation,[[{[m_1,m_2,m_3,m_4,m_5],[1,2,3,4,5]},
                                          [svr_1,svr_2,svr_3],1,3]]},
     monitor = {mce_ltl_parse:ltl_string2module_and_load(
                    "always(R and Q) ⇒ eventually P)",messenger_mon),
                    {void,[{'P',basicPredicates:show_message({m_2,mobile_connected})},
                           {'Q',basicPredicates:receive_cmd({connected,m_2})},
                           {'R',basicPredicates:show_message({m_2,server_busy})}]}},
                    algorithm = {mce_alg_buechi,void}}).
```

After the two properties being checked in McErlang, the tool-set returns "Execution terminated normally." with 11412 states being explored. The properties are concluded to be held in the implementation.

4.2 Fault Detection

This subsection evaluates the capabilities of Etomcrl2 and McErlang on fault detection. Two types of faulty implementations are devised, one of which is designed with a coding fault while the other a configuration error.

A. Detecting a Coding Error. The telecoms system takes use of a number of FSs. These FSs should be configured in a list [svr_1, ..., svr_(k-1), svr_k]. A faulty implementation is devised where the FS list is coded in the format of [svr_1, ..., svr_(k-1)|svr_k]. Such a coding pattern is syntactically legal and will not cause any errors or exceptions in the state of compiling. However, the injected fault could give rise to a serious problem since, when trying to connect to an FS, instead of **svr_k**, a client may send the request to the list [**svr_k**]. [**svr_k**] is not recognised as an FS entity, which could make the telecoms system crashed.

The faulty implementation is then model-checked using Etomcrl2 and McErlang respectively. The fault is immediately captured when the implementation is compiled by McErlang to derive the core files for model-checking. Thanks to the fact that McErlang implements the Erlang semantics directly on the model checker, the location of the fault in the code and the interleaving of the actions that caused the error are layed out clearly, which provides clues to fix it.

The fault is also detected by Etomcrl2 and CADP. After the implementation source codes are translated into an mCRL2 specification, CADP is used to verify the system's properties. It is discovered that there exists *deadlock* in the generated state space. By examining the execution traces, it is concluded that the *deadlock* is induced by the fact that clients send requests to the FS [**svr_k**]. Compared to McErlang, when using Etomcrl2 to debug Erlang programs, it is more difficult to locate the error in the original code.

B. Detecting a Configuration Error. A configuration error is devised in this section. Here, we define the telecoms is constructed with two FSs (svr_1 and svr_2) and four clients (m_1, m_2, m_3 and m_4) where four clients simultaneously request a connection to an FS. Both svr_1 and svr_2 are meant to be designed with a capacity of 2, and we assume that one (say svr_2) by mistakenly implemented with a capacity of 1. This could cause serious problems as one

client will iteratively make a request to connect to the system without knowing whether he/she will ever get through.

One way to detect such a problem is to check whether the four clients are successfully connected to the FSs. Since the system is designed with the capacity of 4, all four clients should have connected to an FS. Thus, for each client m_i, $i \in \{1, 2, 3, 4\}$, The properties can be defined as "when client m_i sends *connecting* request to the system, its request will be fairly accepted by an FS (svr_1 or svr_2)". The properties are constructed in Etomcrl2 and McErlang as shown:

```
[true*. "cmd(m_i, connecting)" *]
(<true* "action_show(m_i, connected)"> or
<true* "action_show(m_i, connected)">) true
```

```
property3() →
  mce:start(#mce_opts
     {program = {action,startSimulation,[[{[m_1,m_2,m_3,m_4,m_5],[1,2,3,4,5]},
                                          [svr_1,svr_2,svr_3],1,3]]},
     monitor = {mce_ltl_parse:ltl_string2module_and_load(
                    "always(R and Q) ⇒ eventually P)",messenger_mon),
                    {void,[{'P',basicPredicates:receive_cmd({connecting,m_i})},
                           {'Q',basicPredicates:show_message({m_i,mobile_connected})}]}},
                    algorithm = {mce_alg_buechi,void}}).
```

Using these properties, Etomcrl2/CADP and McErlang can correctly distinguish the correct and faulty implementations based upon the design we wish to check against.

5 Comparisons between Etomcrl2 and McErlang

This section makes a comparison between Etomcrl2 and McErlang.

5.1 Effectiveness in System Verification

The experimental results suggest that both Etomcrl2/CADP and McErlang are effective in verifying the system properties. In terms of fault detection, both model-checkers are able to isolate the faults from the faulty implementations and provide clues to fix them. However, McErlang is unable to verify properties related to *timeout* event, since it implements neither a discrete nor a real-time semantics for Erlang program. This could decrease its applicability to some examples for classes of systems where exact timing is crucial for correctness. Etomcrl2 introduces a discrete clock into the mCRL2 specification, which makes it possible to simulate the timing process.

Etomcrl2 takes use of a static/fixed state space for system verification, that is, before the process of verification starts, the tool-set generates a complete state space and uses the state space throughout any stages of system verification; while, McErlang applies on-the-fly to dynamically generate a small/partial state space for a property under evaluation, that is, when a property is about to be checked, the tool-set generates a partial state space that is sufficient to check the property's correctness.

Both Etomcrl2 and McErlang can be applied to check for the presence of *deadlock* in the scenarios, where the size of system components incrementally grows. It can be seen that, when McErlang is used for *deadlock* checking and the telecoms system is free from *deadlock*, McErlang generates a complete state space. All experiments are run in a desktop of DELL OPTIPLEX 760 (Memory: 2.0Gib, Processor: Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz). The numbers of states and the times used to generate such state spaces are illustrated in Table 1 where "<" stands for *less than*.

Table 1. State spaces and the times used for their generations

Clients	States (E2Crl)	Times (E2Crl)	States (McErl)	Times (McErl)
1	20	21 sec	38	$<$ 5 sec
2	77	23 sec	214	$<$ 5 sec
3	286	32 sec	5163	$<$ 5 sec
4	1217	172 sec	543358	46 sec
5	6176	2747 sec	1801308	2385 sec

Table 1 shows (1) Etomcrl2 generates fewer states than McErlang does. McErlang defines function calls as state changes. Transitions depicted in McErlang are interpreted as function calls. By contrast, Etomcrl2 highly abstracts Erlang and OTP behaviour in mCRL2, where modeling rules are used to concisely encapsulate the executions of Erlang and OTP functions with a set of actions. Compared to McErlang, Etomcrl2 can use fewer states to model some system functionalities; (2) McErlang delivers answers faster than Etomcrl2. This is due to the fact that McErlang applies on-the-fly techniques for system verification, and (3) when the complexity of the system under investigation arrives at a certain degree, both model-checkers come to a bottle-neck and become less efficient in the generation of state spaces. These problem is currently being addressed in McErlang by defining partial order reductions that decrease the number of states.

5.2 Usability

Etomcrl2 is a tool-set that has to take use of a third-part model-checker such as CADP to perform model-checking. CADP is a standard model-checker, and it allows Linear Temporal Logic (LTL) formulas to be directly used for modeling system behaviour. This makes the process upon the verification of an Erlang application (using Etomcrl2 and CADP) a standard model-checking process. By defining the system's properties in a set of LTL formula, one can easily model-check the Erlang application, without further learning about Erlang.

The limitation upon the use of Etomcrl2 is that every aspect of Erlang and OTP components has to be modeled in mCRL2. So far, Etomcrl2 has included the abstract schema for Erlang syntax, OTP modules *gen_server*, *supervisor*, *gen_fsm*, and *timeout* event. To make Etomcrl2 a more comprehensive tool-set, abstract rules for other OTP components such as *event* need to be developed.

By contrast, it is a much easier task to add support for new OTP components in McErlang as such components can be written directly in Erlang.

Moreover, when using Etomcrl2 it is usually more difficult to identify the reason for an error, and location in the source code where it was introduced, than when using McErlang. This is because for Etomcrl2 errors are discovered in the mCRL2 specification *generated* from the Erlang program rather than in the original Erlang program itself.

In comparison, McErlang is an independent model-checker that directly applies model-checking algorithms in system verification. McErlang uses on-the-fly techniques for checking a system property. This has the potential for making McErlang a faster model-checker for some verification problems. When examining a system property, McErlang produces a tree of executions and allows the executions to be performed step by step. This provides a means to track and analyse the execution traces from the property under evaluation.

There exists some potencial limitations on the use of McErlang. McErlang reimplements a number of model checking algorithms whereas Etomcrl2 reuses an already available mature implementation in mCRL2 and CADP of a set of model checking algorithms. This brings the advantage that the developers of Etomcrl2 can focus only on the problem of translating Erlang to the specification language of the model checker. In general one would expect that as a result of using a mature model checker, Etomcrl2 would be faster than McErlang[3]. Early experiments reported here do not show such a slow-down of McErlang compared to Etomcrl2, but if improvements are made to mCRL2 or CADP, the Etomcrl2 tool would benefit too without having to write new code. Besides, McErlang is particularly developed for model-checking Erlang applications, where system properties must be described partly in Erlang. This is in fact an advantage since facilitates the use of McErlang for Erlang programmers.

6 Summary

Model-checking programs is important in the development of a reliable software system. This paper evaluates and compares the Erlang model-checker Etomcrl2 and McErlang by applying them to verify a telecoms case study. The telecoms is designed with a server-client infrastructure and is implemented making use of the OTP components *gen_server*, *supervisor* and *gen_fsm* and the *timeout* event. A number of system's key properties are outlined and verified by using Etomcrl2 and McErlang. Experimental results show both model-checkers are effective in verifying the majority of these properties. In terms of fault detection, both model-checkers are able to distinguish the devised faulty implementations from the design. Early benchmark results indicate that McErlang delivers answers quicker than Etomcrl2.

We have compared the two model-checkers with their usabilities. A number of limitations on the uses of the tool-sets are summarised. Through such a case

[3] Especially since mCRL2 is largely implemented using C++ compared to McErlang which is implemented using Erlang.

study, we propose suggestions for both toolsets in their future work. Etomcrl2 has to make use of a third-party toolset such as CADP to model-check an Erlang application. This requires every aspect of Erlang and OTP components to be modeled in mCRL2. To make Etomcrl2 a mature model-checker, it is necessary to develop abstract rules for the other OTP components such as *event*.

McErlang is not capable of verifying some properties related to timing, since it implements neither a discrete nor a real-time semantics for Erlang program. As such, it is an item for future work to extend McErlang with an implementation of a timed semantics. On the other hand, there are classes of systems and properties which require real-time model checking algorithms too, for which the discrete clock implementation in Etomcrl2 is not sufficient.

It has been noticed that, when the complexity of the system under investigation arrives at a certain degree, both model-checkers come to a bottle-neck and become less efficient in the generation of state spaces. To overcome such a problem, for both tool-sets, more work need to be carried out. In particular, there is some work in progress in using some partial order reductions in McErlang.

Acknowledgements

This work was funded by the FP7 project *ProTest*, number 215868: www.protest-project.eu.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice-Hall, Englewood Cliffs (1996)
2. Arts, T., Benac-Earle, C., Penas, J.J.S.: Translating Erlang to μCRL. In: Kishinevsky, M., Darondeau, P. (eds.) 4th International Conference on Application of Concurrency to System Design, pp. 135–144. IEEE Computer Society, Los Alamitos (June 2004)
3. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press, Cambridge (1990)
4. Benac-Earle, C., Fredlund, L.-Å.: Verification of Language Based Fault-Tolerance. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2005. LNCS, vol. 3643, pp. 140–149. Springer, Heidelberg (2005)
5. Benac-Earle, C., Fredlund, L.-Å., Derrick, J.: Verifying Fault-Tolerant Erlang Programs. In: Sagonas, K., Armstrong, J. (eds.) Proceedings of ACM SigPlan Erlang 2005 Workshop, pp. 26–34. ACM Press, New York (September 2005)
6. CADP, `http://www.inrialpes.fr/vasy/cadp/`
7. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media, Sebastopol (2009)
8. Fredlund, L., Svensson, H.: McErlang: a Model Checker for a Distributed Functional Programming Language. In: Hinze, R., Ramsey, N. (eds.) 12th ACM SIGPLAN International conference on functional programming (ICFP 2007), pp. 978–1–59593–815–2 (2007)
9. Groote, J.F., Mathijssena, A., van Weerdenburga, M., Usenkoa, Y.: From μCRL to mCRL2. Electronic Notes in Theoretical Computer Science 162, 191–196 (2006)

10. Groote, J.F., Ponse, A.: The syntax and sematics of μCRL. In: Ponse, A., Verhoef, C., van Vlijmen, S. (eds.) Algebra of Communicating Processes 1994, Workshop in Computing, pp. 26–62 (1995)
11. Guo, Q.: Verifying Erlang/OTP Components in μCRL. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 227–246. Springer, Heidelberg (2007)
12. Guo, Q., Derrick, J.: Verification of Timed Erlang/OTP Components Using the Process Algebra μCRL. In: Thompson, S., Fredlund, L.-Å. (eds.) 6th ACM SIGPLAN Erlang Workshop, pp. 55–64. ACM Press, New York (2007)
13. Guo, Q., Derrick, J.: Formally Based Tool Support for Model Checking Erlang Applications. International Journal on Software Tools for Technology Transfer (2010) (under review)
14. Guo, Q., Derrick, J., Hoch, C.: Verifying Erlang Telecommunication Systems with the Process Algebra μCRL. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 201–217. Springer, Heidelberg (2008)
15. Havelund, K., Lowry, M., Penix, J.: Formal Analysis of a Space-Craft Controller Using SPIN. IEEE Transactions on Software Engineering 27(8), 749–765 (2001)
16. Holzmann, H.: Design and Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs (1991)
17. Svensson, H.: Implementing an LTL-to-Büchi translator in Erlang: a protest experience report. In: 8th ACM SIGPLAN Erlang Workshop, pp. 63–70. ACM Press, New York (September 2009)