

Model-checking Erlang - A Comparison between EtomCRL2 and McErlang

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

Clara Benac Earle and Lars-Åke Fredlund
Facultad de Informática,
Universidad Politécnica de Madrid
Boadilla del Monte,
28660, Madrid, Spain
{cbenac,fred}@babel.ls.fi.upm.es

September 3, 2010

Background

- Model checking has been widely used in system design and verification. 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 NASA's Remote Agent Spacecraft Control System.
- Thus, to derive a reliable system, it is essential not only to verify the system design, but also to model check its implementations.

Motivation

- There are essentially two approaches to model-checking programs: one is to abstract program sources into a formal specification (for example, a μ CRL specification), upon which the standard model checker such as CADP can be applied for verifying system's properties; the other is to directly implement verification algorithms to programming language.
- Both methods have been investigated in model-checking functional programming language Erlang. Correspondingly, two tools, Etomcrl2 and McErlang are developed to support to process of verifications. This paper reviews and compares these two techniques.
- 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.
- Through such a case study, we intend to evaluate the effectiveness of the two methodologies in system verification, and provide suggestions for the developers of the two Erlang model-checkers in their future work.

Outlines

- Brief introduction to Erlang/OTP with a telecoms example;
- Introductions to Etomcrl2 and McErlang;
- Verifying the telecoms example using Etomcrl2/CADP and McErlang;
- Evaluate and compare Etomcrl2 and McErlang;
- Summary.

Erlang and OTP

- Erlang 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.
- This has caused Erlang 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.

Telecoms: an Illustrative OTP Example

- 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 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). There are four states: *idle*, *connected*, *calling* and *top-up*, where initially, the system is set to the *idle* state.

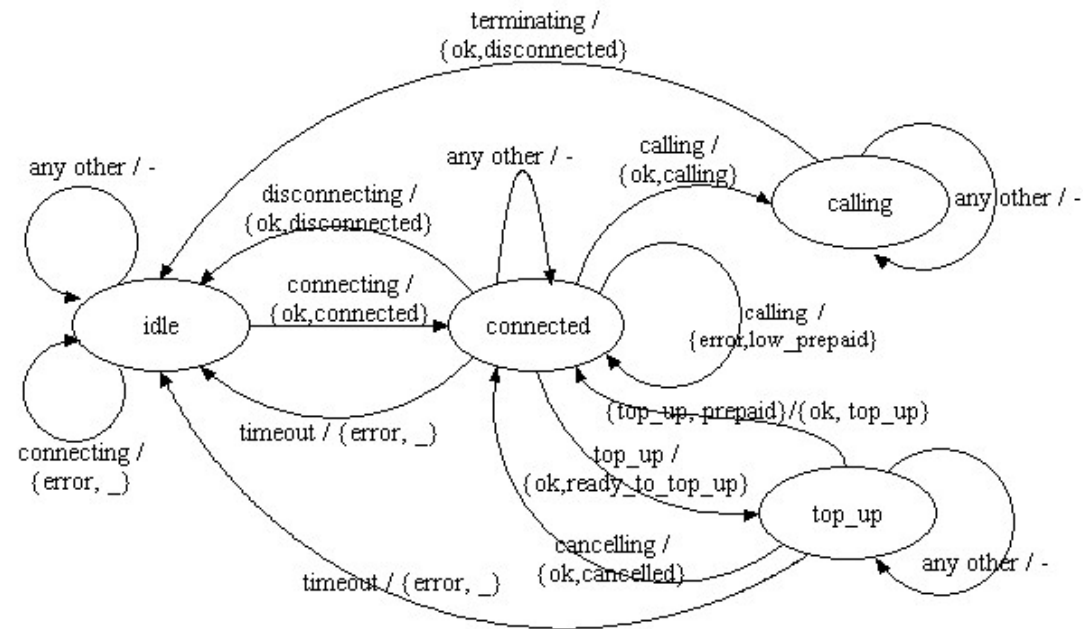


Figure 1: The mobile phone behaviour modeled as an FSM.

- A client FSM has a timing restriction applicable when in the state *connected* or *top_up*.

Telecoms: 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 FSM state function *idle*

```

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} → :

```

% The FSM state function *connected*

```

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.

```

Erlang Model-checker: Etomcrl2

- Etomcrl2 is a tool-set that automatically translates the source codes of an Erlang application into an mCRL2 specification, upon which the standard model checker CADP is used to generate a (finite) state space to check the system properties against the designs.
- The process algebra μ CRL (micro Common Representation Language) is an extension of the process algebra ACP. 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 mCRL2 is a new version of μ CRL that is extended with higher-order data-types, standard data-types, multi-actions and local communication.
- Compared to μ CRL, mCRL2 is more applicable in practice.

Erlang Model-checker: McErlang

- McErlang 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 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 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 supports model checking programs against full Linear Temporal Logic (LTL) formulas.
- The LTL2Buchi tool 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.

The Tools in Use

- There are two groups of experiments. In the first group, a number of properties are checked against the implementations 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.

Model-checking Telecoms: Property Verification 1

- 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; 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;
 3. the logics of *timeout* event have been correctly implemented.

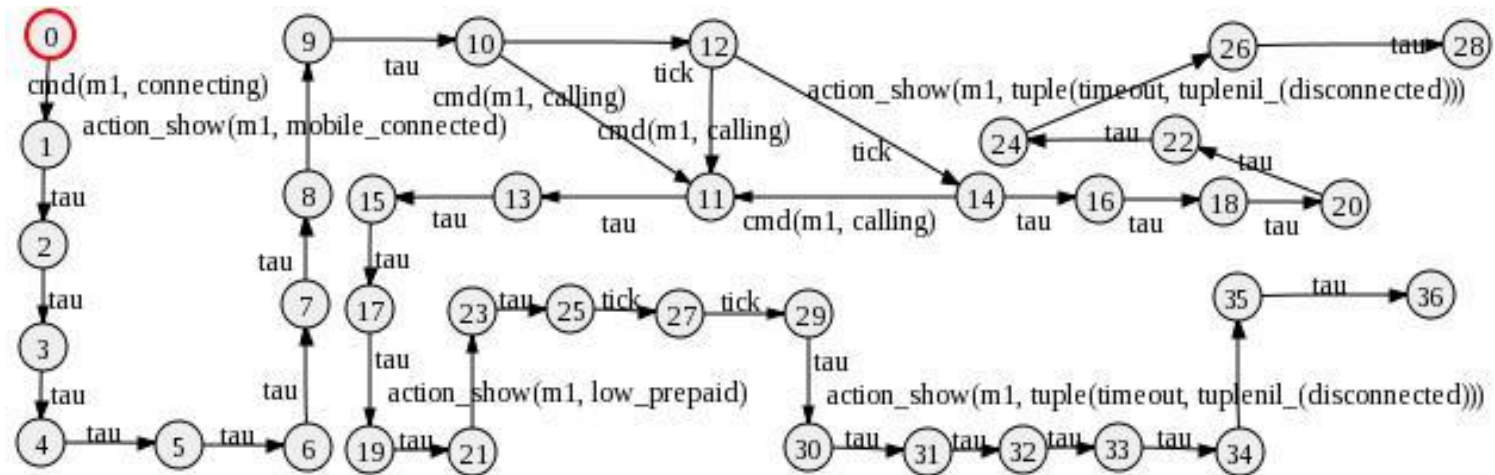


Figure 2: LTS: m_1 tries to make a call with low prepaid and the request is denied.

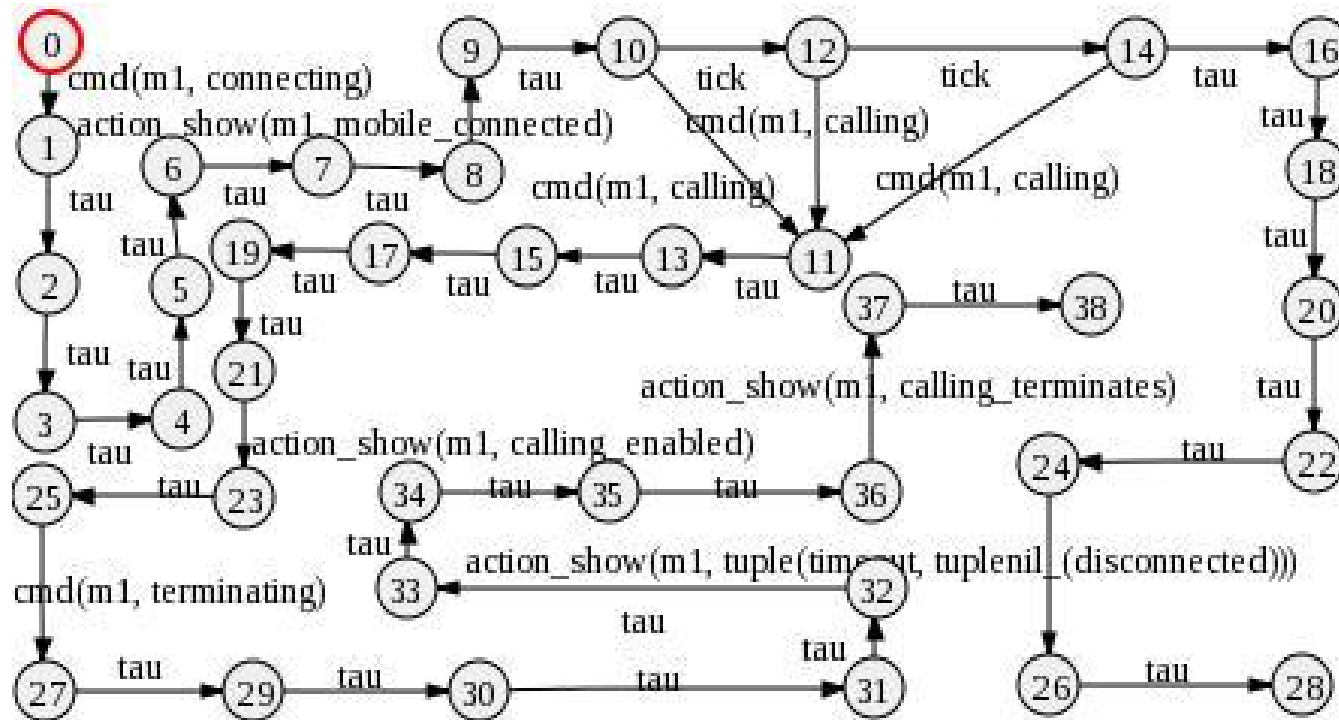


Figure 3: LTS: m_1 tries to make a call with enough prepaid and the request is enabled.

- 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:

$[\text{not}(\text{action_show}(m_1, \text{mobile_connected}))^* . \text{action_show}(m_1, \text{calling_enabled}))] \text{ false}$

- 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:

$[\text{true}^* . \text{action_show}(m_1, \text{mobile_connected})^*]$
 $\langle \text{not}(\text{'tick.tick'})^* . \text{action_show}(m_1, \text{tuple}(\text{timeout}, \text{tuplenil}(\text{disconnected}))) \rangle \text{ false}$

- 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_ert: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.

Model-checking Telecoms: Property Verification 2

- 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.

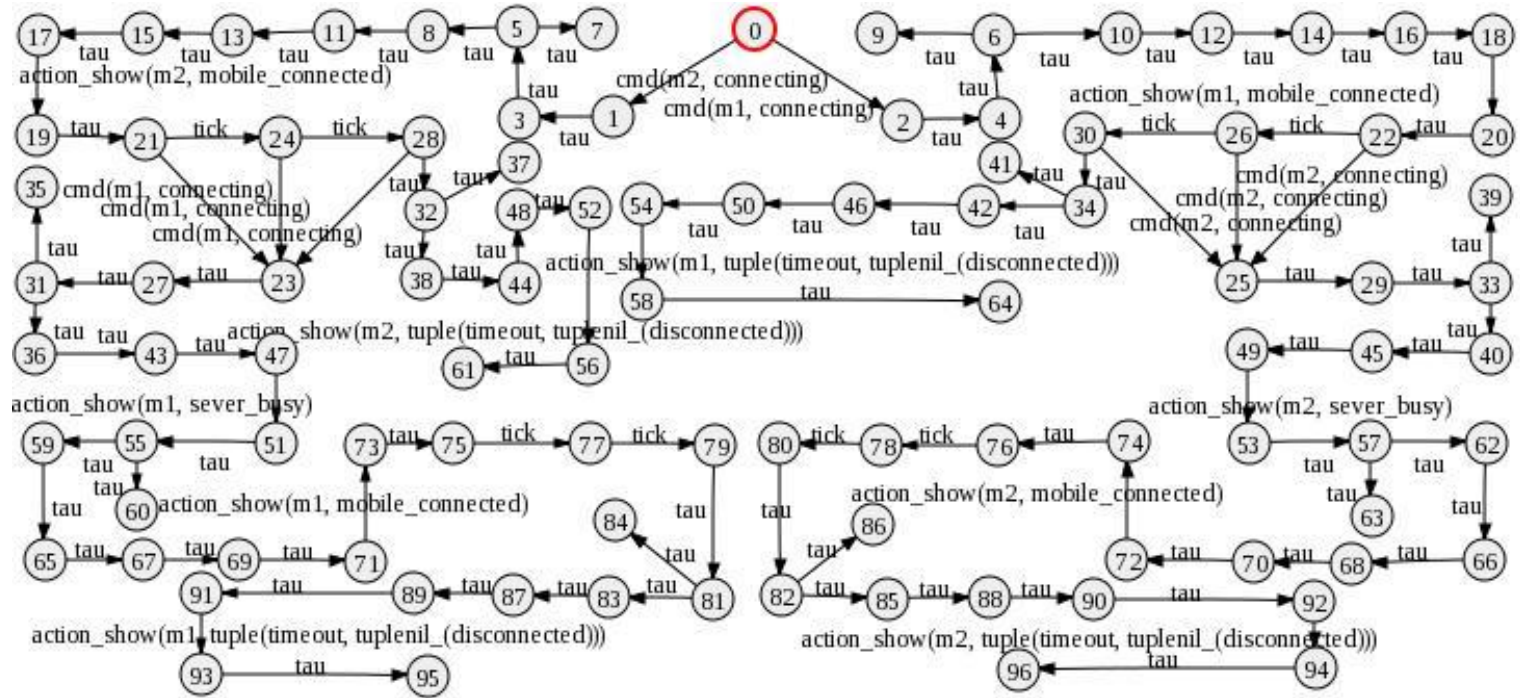


Figure 4: LTS: m_1 and m_2 request to connect to an FS simultaneously with the capacity of svr_1 is set to 1.

- 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:

$$\langle \text{true}^*. \text{action_show}(m_1, \text{mobile_connected})^*. \text{cmd}(m_2, \text{connecting})^*. \text{action_show}(m_2, \text{server_busy}) \rangle \text{true}$$

- Another property we want to check is formalized as:

$$\langle \text{true}^*. \text{cmd}(m_2, \text{connecting})^*. \text{action_show}(m_2, \text{server_busy})^*. \text{cmd}(m_2, \text{connecting})^*. \text{action_show}(m_2, \text{mobile_connected}) \rangle \text{true}$$

- 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.

Model-checking Telecoms: Fault Detection 1

- This experiment is designed to detect 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 fault is detected by Etomcrl2 and McErlang. Compared to McErlang, Eromcrl2 is more difficult to locate the error in the original code.

Model-checking Telecoms: Fault Detection 2

- This experiment is designed to detect a configuration error;
- 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.
- 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.

Etomcrl2 vs McErlang - Effectiveness in System Verification

- 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.
- Before the process of verification starts, Etomcrl2 generates a complete state space and uses the state space throughout any stages of system verification; McErlang applies on-the-fly to dynamically generate a small/partial that is sufficient to check a property under evaluation.

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: State spaces and the times used for their generations.

- Etomcrl2 generates fewer states than McErlang does.
- McErlang delivers answers faster than Etomcrl2. This is due to the fact that McErlang applies on-the-fly techniques for system verification;
- 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.

Etomcrl2 vs McErlang - Usability

- Etomcrl2 has to take use of a third-part model-checker such as CADP to perform model-checking. This makes the process upon the verification of an Erlang application (using Etomcrl2 and CADP) a standard model-checking process.
- Limitation: every aspect of Erlang and OTP components has to be modeled in mCRL2.
- McErlang is an independent model-checker that uses on-the-fly techniques for model-checking. McErlang re-implements a number of model checking algorithms while Etomcrl2 reuses an already available mature implementation in mCRL2 and CADP;
- In general, Etomcrl2 would be faster than McErlang but early experiments do not show such a slow-down of McErlang compared to Etomcrl2. If improvements are made to mCRL2 or CADP, Etomcrl2 would benefit too without having to write new code.
- McErlang is particularly developed for model-checking Erlang applications, where system properties must be described partly in Erlang.

Summary

- This paper evaluates and compares the Erlang model-checker Etomcrl2 and McErlang by applying them to verify a telecoms case study.
- 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.
- A number of limitations are summarised. 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.
- McErlang is not capable of verifying some properties related to timing and as such, it is an item for future work to extend McErlang with an implementation of a timed semantics.
- 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.