McErlang: a Tutorial *

Clara Benac Earle and Lars-Åke Fredlund
Babel group, LSIIS, Facultad de Informática,
Universidad Politécnica de Madrid, Spain
email: {cbenac,lfredlund}@fi.upm.es

November 22, 2011

1 Prerequisites

We assume that you will be running McErlang under the Linux operating system; we have been using Ubuntu 8.10 to develop and test the tool. Moreover we assume that Erlang is already installed on your computer and that the command erl for starting Erlang can be run directly without specifying a directory path. The model checker was developed under Erlang/OTP R12B-5; it will probably work under other releases of Erlang/OTP as well but we haven't verified this.

We recommend that you have the User Manual at hand to help you understanding this tutorial.

2 Installing McErlang

For the development of McErlang source code, documentation and examples we are using the Subversion revision control system. This means that there is a central repository where the code is located. To get your own local copy of the repository type the following command in a terminal window.

```
$ svn checkout https://babel.ls.fi.upm.es/svn/McErlang/trunk McErlang
```

This command will create a local copy of all the McErlang code and examples. To compile McErlang it normally suffices to execute "./configure; make" in the main McErlang directory. In the following we assume that McErlang is installed in the directory ~/McErlang.

```
$ cd McErlang; ./configure; make
```

The following examples assume that you have put the directory McErlang/scripts in the "command path" of your shell. Consult the manual for your command shell if you are not sure how to do this. As an example, if you are running a bash shell, the following piece of text can be added to the file .profile:

```
# set PATH so it includes the directory with McErlang scripts
if [ -d ~/McErlang/scripts ] ; then
    PATH=~/McErlang/scripts:"${PATH}"
fi
```

*This work has been partially supported by the FP7-ICT-2007-1 Objective 1.2. IST number 215868

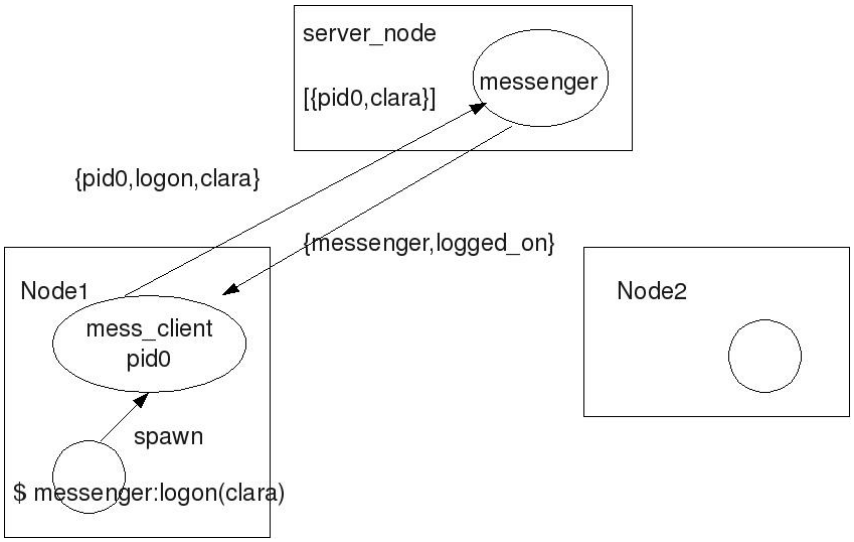

Figure 1: User clara sends a logon message to the messenger server

3 The Messenger example

To illustrate the use of McErlang we consider the messenger example from the "Getting started with Erlang" document in the Erlang/OTP R12B documentation[1]. To begin working with the example, locate the source code for the messenger example (**messenger.erl**) in the McErlang/examples/Simple_messenger directory.

The messenger is a program which allows users to log in on different nodes and send simple messages to each other. The messenger allows "clients" to connect to a central server, specifying their identities and locations that are then stored in the state of the server. Thus a user won't need to know the name of the Erlang node where a user is located to send a message to that user, only his identity.

Let us consider the example with three nodes depicted in Fig. 1, the node server_node where the messenger server is already running and two other nodes, Node1 and Node2. If a process in Node1 calls the messenger:logon(clara) command the result of this call is the spawning of a process registered as mess_client that will send a message to the messenger server containing the name of the user and the pid of the mess_client. The messenger will then look into its state and if it is empty it will add the new user to the UsersList and send a confirmation message to the mess_cient.

Lets now assume that both a user clara and a user fred in fig. 2 are logged on and clara wants to send a message to fred. Again the message will be sent from the mess_client running in Node1 to the messenger server which, after checking that both clara and fred are logged on, will forward the message to the mess_client running in Node1.

Upon a logoff, the mess_client will send a message to the messenger server that will produce the deletion of the user that wanted to log off from the UsersList maintained by the messenger server.

4 Executing the messenger example inside McErlang

To run the messenger example inside McErlang we need to implement the start up of the system, and simulate the actions of the (simulated) users of the messenger service. This test-case for using the messenger service is what we call a scenario. Random scenarios can, for instance, be generated by using the Quickcheck tool (this possibility is not in the scope of this tutorial).

[1] Section 3.5 in `http://erlang.org/doc/getting_started/part_frame.html`.

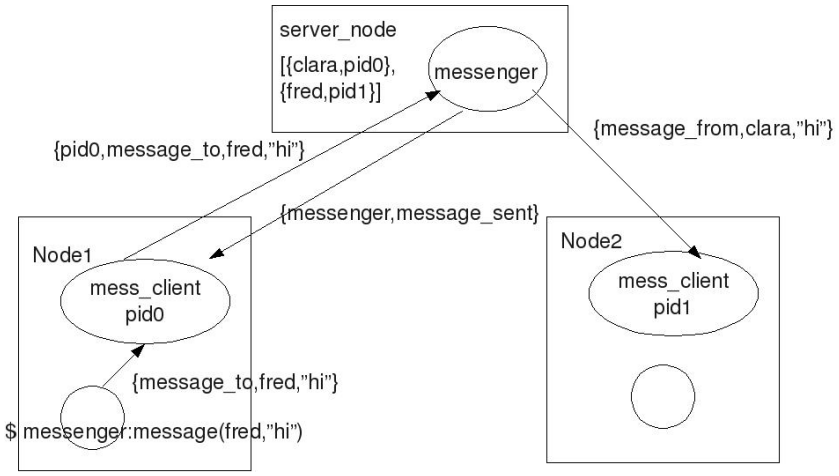

Figure 2: User clara sends a message to user fred

4.1 Writing a scenario

In the file **scenario.erl**, in McErlang/examples/Simple_messenger directory, we have defined a function named "start" which should be invoked with a list of commands representing the commands that each user sends to the messenger. The start function first spawns a process executing the module:start_server function on the node named "server"; the result is that the messenger process (the server) is created. Then a process is created for each simulated user, running on separate nodes. These user processes send commands to the messenger process, which forwards them to users on other nodes (in case the command is a request to send a message to another user).

```
-module(scenario).
-export([start/1,start_clients/2,execute_commands/1]).

start(Commands) ->
    spawn(server_node,messenger,start_server,[]),
    start_clients(Commands,1).

start_clients([],N) -> ok;
start_clients([Commands|Rest],N) ->
    Node = list_to_atom("n"++integer_to_list(N)),
    spawn(Node,?MODULE,execute_commands,[Commands]),
    start_clients(Rest,N+1).

execute_commands(Commands) ->
    lists:foreach
    (fun (Command) ->
         case Command of
             {logon, Client} ->
                 messenger:logon(Client);
             {message,Receiver,Msg} ->
                 messenger:message(Receiver,Msg);
             logoff ->
                 messenger:logoff()
         end
     end, Commands).
```

4.2 Compiling the Messenger Example using McErlang

To execute Erlang code under the McErlang model checker a translation step is necessary, whereby Erlang code is translated into modified Erlang code suitable for running under the control of the McErlang model checker (which has its proper runtime system for managing processes, communication and nodes). Thereafter, a normal Erlang compilation translates the modified source code into beam (or native) object files.

To compile the Messenger example type "make" in the McErlang/examples/Simple_messenger directory.

```
~/McErlang/examples/Simple_messenger$ make
../../scripts/mcerl_compiler -sources messenger.erl monSendLogon.erl \
                             scenario.erl test.erl testl.erl tests.erl
```

The code resulting from the source-to-source translation and the beam files are stored in the ebin directory. In the ebin directory, if we type "mcerl", an erlang shell is started.

```
~/McErlang/examples/Simple_messenger$ cd ebin/
~/McErlang/examples/Simple_messenger/ebin$ ../../../scripts/mcerl
Erlang (BEAM) emulator version 5.6.5 [source] [smp:4] [async-threads:0] [hipe] [kerne

Eshell V5.6.5  (abort with ^G)
1>
```

From this shell we can invoke the verification functions implemented in the **run.erl** file in the McErlang/examples/Simple_messenger directory. We see some examples in the following sections.

5 Debugging the messenger example

We can use the McErlang debugger to explore the execution of the messenger example step by step. The debug() function in the **run.erl** module is used to start a debugging session. To debug the messenger example we only need to specify the mce_alg_debugger algorithm, and the program to debug, here the scenario module with a list containing the parameters used by the start function. The list contains the commands sent by the user clara (logon, send the message "hola" to fred and logoff) and the commands sent by the user fred (logon).

```
debug() ->
  mce:start
     (#mce_opts
      {program={scenario,start,
                [[[{logon,clara},{message,fred,"hola"},logoff],
                 [{logon,fred}]]]},
       algorithm={mce_alg_debugger,void}}).
```

To start the debugging session we type run:debug() in the erlang shell.

```
McErlang/src/McErlang/examples/Simple_messenger/ebin$ ../../../scripts/mcerl
Erlang (BEAM) emulator version 5.6.4 [source] [smp:2] [async-threads:0] [hipe] [kerne

Eshell V5.6.4  (abort with ^G)
1> run:debug().
Starting McErlang model checker environment version 1.0 ...

Starting mce_alg_debugger(void) algorithm on program
scenario:start([[[{logon,clara},{message,fred,"hola"},logoff],[{logon,fred}]])
```

```
with monitor mce_mon_test(ok)

...

At stack frame 0: transitions:

1: process <node0@esther,1>:
    run function scenario:start([[[{logon,clara},{message,fred,"hola"},logoff],
                          [{logon,fred}]]])


stack(@0)>
```

Note that for readability some printouts have been omitted in the example.

The debugger shows all the possible transitions from the current state (here the initial state) to the next state. In this example, the only possible transition is to execute the scenario:start function in the process running on the node0@esther node. Thus, we choose "1.".

```
stack(@0)> 1.

At stack frame 1: transitions:

1: node server_node:
    receive signal {spawn,{messenger,start_server,[]},no,{pid,node0@esther,1}}
   from node node0@esther
```

Now the only possible transition is for the server_node node to receive the message spawn from the process with pid 1 located in node0@esther.

```
stack(@1)> 1.

At stack frame 2: transitions:

1: process <server_node,2>:
    run function messenger:start_server([])

2: node node0@esther:
    receive signal {message,{pid,node0@esther,1},
                           {hasSpawned,{pid,server_node,2}}}
   from node server_node
```

Now there are two possible transitions: to execute the messenger:start_server function or to receive a message at the node node0@esther containing the pid of the process that has been spawned.

6 Model Checking the Messenger example

McErlang is a model checker for programs written in Erlang. The idea is to replace the part of the standard Erlang runtime system that concerns distribution, concurrency and communication with a new runtime system which simulates processes inside the model checker, and which offers easy access to the program state.

A transition system is a digraph that represents the behaviour of a system. The labeled transition system generated by the model checker comprises two elements:

- system states which record the state of all nodes, all processes, all message queues, etc of the program to model check. Such states are *stable* in the sense that all processes in a state are either waiting in a receive statement to receive a message, or have just been spawned.

- transitions or computation steps between a source state and a destination state. A transition is labeled by a sequence of actions that represent selecting one process in the source state which is ready to run, and letting it execute until it is again waiting in a receive statement. The actions that label the transition are the side effects caused by the execution of the process (i.e., sending a message to another process, linking to another process, ...).

Thus the model checker uses an interleaving semantics to execute Erlang programs.

McErlang provides a number of different ways to formulate correctness properties for checking on a given program. Correctness properties are given in the form of a monitor/automaton that is executed in parallel with the program, checking for errors along the execution path. Monitors are either safety monitors or büchi monitors which encode linear temporal logic properties (LTL). Given a program and such an automaton, McErlang will run them in lockstep letting the automaton investigate each new program state generated. If the property does not hold, a counterexample (an execution trace) is generated.

We show two examples of using safety monitor in the following section and in sect. 8. In sect. 7 we show examples of using büchi monitors.

6.1 Model Checking a simple property

We can use the mce_alg_safety algorithm to check that a property holds on all the states of a program. We define the function safety in the run.erl file.

```
safety() ->
  mce:start
     (#ev_opts
      {program={scenario,start,
                [[[{logon,clara},{message,fred,"hola"},logoff],
                 [{logon,fred}]]]},
       algorithm={mce_alg_safety,void}}).
```

Because we do not specify a monitor, the default monitor mce_mon_test is used, which is always true. However, the model checker will check that no process terminates abnormally.

Let us execute the safety function.

```
Eshell V5.6.4  (abort with ^G)
1> run:safety().
Starting McErlang model checker environment...
Starting mce_alg_safety(void) algorithm on program
 scenario:start([[{logon,clara},{message,fred,"hola"},logoff],[{logon,fred}]])
with monitor mce_mon_test(ok)

*** Run ending. 2036 states explored, stored states 836.


Execution terminated normally
Access result using mce:result()
ok
```

Apparently no process terminated abnormally.

7 Model checking using Büchi monitors

The alternative we will consider here is to express the desired property in Lineal Temporal Logic (LTL), and to use the LTL2Buchi tool developed by Hans Svensson and distributed with McErlang to automatically translate an LTL formulae into a Büchi monitor (see User Manual).

We have model checked several interesting safety and liveness properties of the messenger example. Some of them can be found in the run.erl file. One such property can be informally expressed as follows: if a user who is logged on sends a message to another user who is also logged on then the recipient of the message will eventually receive the message. As mentioned before, McErlang allows access to the program states and the actions between states. We explain in the following sections how this information can be used to write properties.

7.1 Using probe actions to write properties

One possible and more precise description of the aforesaid property is the following:

> *if user1 does not send a message m to user2 until user2 is logged on, then if user1 does send a message m to user2 then eventually user2 receives the message m.*

The formalization in LTL of the above formula can be found in the "message_received1()" function in the **run.erl** file.

```
"not P until Q => (eventually P => eventually R)"
```

The predicates P, Q and R have the following meaning in the concrete scenario where a user clara that first sends a logon message to the messenger server, then sends the message "hi" to the user fred and finally sends a logoff message, and a user fred that sends a logon message and a logoff message:

- P: clara sends message "hi" to fred
- Q: fred is logged on
- R: fred receives the message "hi" from clara

Linear Temporal Logic is defined over program runs: $P\ until\ Q$ holds for a program run if at every state of the run the P predicate holds, until a state in the program run is encountered where the Q predicate holds (and Q must hold for some state on the run). $eventually\ R$ holds for a program run if the predicate R holds at some program state in the run. Normal logical implication is denoted by the "⇒" symbol.

For simplicity and modularity the property and the predicates present in the property are considered separately. To write the predicates or basic facts in the formula (user1 sends a message m to user2, etc.) McErlang allows access to the program states and the sequence of actions labelling transitions between states. These predicates can be written directly in Erlang, using all the expressiveness of the language. For example, a predicate stating that a user is logged on can be implemented as a function "logon" that returns true when an action corresponding to the user being logged on is found labelling a transition. The process of searching for the desired action is simplified if we annotate the program with what we call "probe actions", which serve to make the internal state of a program visible to the model checker in a simple fashion.

In the messenger example we have annotated the program code with probe actions that are referred to in the predicates. For example, the following probe action has been added in messenger.erl to the client function for expressing that a user is logged on:

```
client(Server_Node, Name) ->
  {messenger, Server_Node}!{self(), logon, Name},
  await_result(),
  mce_erl:probe(logon, Name),
  client(Server_Node).
```

From the example we can see that a probe action, as created using the mce_erl:probe function, has two arguments, corresponding to a label naming the particular probe, and an arbitrary Erlang term as probe argument.

The code implementing predicates (**basicPredicates.erl**) is in the McErlang/examples/Simple_messenger directory. For example, the "logon" predicate is implemented as the "logon" function which provided a user name as argument, defines an anonymous function that returns true if its second argument is a sequence of actions containing a logon probe action corresponding to a logon by the named user:

```
logon(Name) ->
  fun (_,Actions,_) ->
    lists:any
    (fun (Action) ->
       try
         logon =
           mce_erl_actions:
             get_probe_label(Action),
         Name =
           mce_erl_actions:
             get_probe_term(Action),
         true
       catch _:_ -> false end
     end, Actions)
  end.
```

Similarly, probe actions and predicates have been written for the other predicates appearing in property (1).

To model check property (1) on a concrete scenario we use the function message_received in run.erl.

```
Eshell V5.6.5  (abort with ^G)
1> run:message_received1().
Starting McErlang model checker environment version 1.0 ...

Starting mce_alg_buechi(void) algorithm on program
scenario:start([[{logon,clara},{message,fred,"hi"},logoff],[{logon,fred},logoff]])
with monitor messenger_mon({void,[{'P',#Fun<basicPredicates.5.8045620>},
                                  {'Q',#Fun<basicPredicates.1.30432014>},
                                  {'R',#Fun<basicPredicates.6.70178892>}]})

...

Access result using mce:result()
To see the counterexample type "mce_erl_debugger:start(mce:result()). "
ok
2>
```

If we look at the counterexample we should be able to see that the property did not hold because fred could logoff before receiving the message.

One option to address the problem found is to generate only test cases where fred never logs out. However, we prefer to instead rewrite the property to handle the situation when fred logs out.

Thus we modify the property (1) as follows:

> *if user1 does not send a message m to user2 until user2 is logged on, then if user1 does send a message m to user2 then eventually user2 receives the message m from user1, or user2 is logged off.*

The resulting LTL formula is the following:

```
"not P until Q => (eventually P => eventually (R or S))"
```

where S represents the predicate “fred is logged off”. This property is used in the message_received2 function in the **run.erl** file with the same scenario.

```
Eshell V5.6.5  (abort with ^G)
1> run:message_received2().
```

```
Starting McErlang model checker environment version 1.0 ...

Starting mce_alg_buechi(void) algorithm on program
scenario:start([[{logon,clara},{message,fred,"hi"},logoff],[{logon,fred},logoff]])
with monitor messenger_mon({void,[{'P',#Fun<basicPredicates.5.8045620>},
                                  {'Q',#Fun<basicPredicates.1.30432014>},
                                  {'R',#Fun<basicPredicates.6.70178892>},
                                  {'S',#Fun<basicPredicates.2.72475960>}]})

...

*** Run ending. 1004 states explored, stored states 1922.

Execution terminated normally
Access result using mce:result()
ok
2>
```

The property (2) has been checked against several scenarios, returning always a positive result.

7.2 Using probe states to write properties

Working with probe actions in LTL formulas can sometimes be difficult, as we have manually "remember" the occurrence of important actions in the formula. In formulas (1) and (2) above, this was accomplished using the until formula.

Instead of using probe actions we can use so called "probe states". In contrast to probe actions, which are enabled in a single transition step only, such probes are persistent from the point in the execution of the program when they are enabled, until they are explicitly deleted.

As an example we instrument the login code of the server in messenger.erl to record, using the function mce_erl: probe_state , the fact that a user has logged in:

```
%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
  %% check if logged on anywhere else
  case lists:keymember(Name, 2, User_List) of
    true ->
      From!{messenger,stop,
            user_exists_at_other_node},
      User_List;
    false ->
      From!{messenger, logged_on},
      mce_erl:probe_state({logged_on,Name}),
      [{From, Name}|User_List]
    end.
```

Similarly we delete the probe state using the function mce_erl: del_probe_state when a user logs out:

```
%%% Server deletes a user from the user list
server_logoff(From,User_List) ->
  case lists:keysearch(From,1,User_List) of
    {value,{From,Name}} ->
      mce_erl:del_probe_state
        ({logged_on,Name});
    false ->
      ok
```

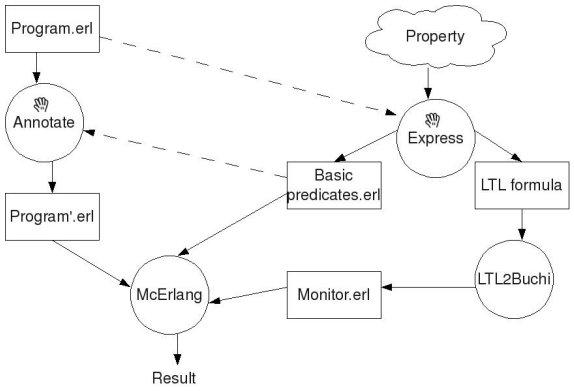

Figure 3: verification schema

```
  end,
  lists:keydelete(From,1,User_List).
```

We can test for the existence of a probe state using the function mce_erl: has_probe_state as exemplified in the function logged_on in the basicPredicates.erl file which checks if a user is logged on:

```
logged_on(Name) ->
  fun (State,_,_) ->
    mce_erl:has_probe_state
      ({logged_on,Name},State)
  end.
```

This predicate will be abbreviated as "t" below.

We can now reformulate the second property above, removing the until operator and one eventually operator with the *always* operator, which holds if its argument holds over the whole execution trace:

```
"always ((P and T) => eventually (R or not T))"
```

Note that since T is a state predicate we can safely negate it to compute its logical negation ("the user is not logged on") whereas the negation of the action predicate $notQ$ in properties (1) and (2) only expresses that the "logon action is not present in the current transition" (but it may have occurred earlier in the execution of the program).

7.3 Verification methodology

A schema of the verification methodology used in this example is shown in Fig. 3. To summarize, the steps we have followed to check that a program verifies a property are the following:

- Express the property in a suitable formalism, in this case, an LTL formula with some predicates. This is done by hand.
- Use the LTL2Buchi tool to translate the LTL property to a buchi automaton implemented as a monitor
- Annotate, by hand, the program with probe actions and/or probe states[2]
- Invoke the McErlang model checker with the annotated program, the predicates, and the monitor.

The result provided by the model checker will be either a positive result if the property holds or a counterexample (an execution trace) if the property does not hold.

[2]Strictly speaking it is not necessary to annotate a program using probe actions or probe states. McErlang can in principle compute the information directly from inspecting the system state. However, this is often not convenient as there is no way to gain access to program data using variable names (variable names are not preserved by the compilation process).

8 Model Checking a Simple Safety Property

An alternative approach to formulating the property in LTL and using the LTL2Büchi translator is to implement a monitor/automaton directly

Let us check a simple property: that a user never logs on. We formulate this property as a safety property, "something bad never happens", in this case the "bad" thing is if a user logs on. The property is encoded as the monSendLogon monitor given below. But first we modify the client code to emit a synthetic "probe" action after having sent off a login message to the server:

```
client(Server_Node, Name) ->
    {messenger, Server_Node} ! {self(), logon, Name},
    mce_erl:probe(logon),
    await_result(),
    client(Server_Node).
```

To implement a correctness property we implement a new module monSendLogon that provides a stateChange function, which is called by McErlang whenever a new program state is encountered. When a logon action is found, the stateChange function returns a tuple, instead of returning a new monitor state, thus indicating an error to the model checker. Note that instead of using a synthetic probe action we could have simply checked for the existence of a send action with a logon message inside.

```
-module(monSendLogon).
-export([init/1,stateChange/3,monitorType/0]).
-include("stackEntry.hrl").

-behaviour(mce_behav_monitor).

monitorType() ->
  safety.

init(State) ->
  {ok,State}.

stateChange(_,MonState,Stack) ->
  case has_probe_action(actions(Stack)) of
    {true, logon} -> {somebody,logon};
    no -> {ok,MonState}
  end.

actions(Stack) ->
  {Entry,_} = mce_behav_stackOps:pop(Stack),
  Entry#stackEntry.actions.

has_probe_action(Actions) ->
   mce_utils:findret(fun mce_erl:match_probe_label/1, Actions).
```

To invoke the model checker with this property we have defined the following function.

```
logon() ->
  mce:start
    (#ev_opts
     {program={scenario,start,
               [[[{logon,clara},{message,fred,"hola"},logoff],
                [{logon,fred}]]]},
      monitor={monSendLogon,[]},
      algorithm={mce_alg_safety,void}}).
```

The result of the execution is given below:

```
Eshell V5.6.4  (abort with ^G)
1> run:logon().
Starting McErlang model checker environment...
Starting mce_alg_safety(void) algorithm on program
 scenario:start([[{logon,clara},{message,fred,"hola"},logoff],[{logon,fred}]])
with monitor monSendLogon([])


***Monitor failed***
monitor error:
  {somebody,logon}

Access result using mce:result()
To see the counterexample type "mce_erl_debugger:start(mce:result()). "
```

We can now access the counter example:

```
2> mce_erl_debugger:start(mce:result()).
Starting debugger with a stack trace
Execution terminated with status:
  monitor failure.

...

stack(@13)> where().
13:



12: process <n1,2> "registered as mess_client":
    receive {messenger,logged_on}
      in function await_result near line 169 of "messenger.erl"
        called by client near line 147 of "messenger.erl"
        with active data clara, server_node
        sent by node n1 in stack frame 11
    *** PROBE logon:clara ***



11: node server_node:
    received signal {signal,server_node,{message,{pid,n1,2},{messenger,logged_on}}} f
        sent by process server_node in stack frame 10

stack(@13)>
```

The `where()` command displays the latest actions of the process. In stack frame 12 we can see that a message is sent to the messenger server, and next the probe action occurs.