McErlang User Manual [*]

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

Contents

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

1 Introduction

To get started with using the McErlang model checker we recommend reading the McErlang tutorial first; it can be found both on the McErlang web page and in the source code distribution. For up-to-date information regarding McErlang consult the `https://babel.ls.fi.upm.es/trac/McErlang` web page.

In the following we make a number of simplifying assumptions. First, that you will be running McErlang under the Linux operating system; we have been using Ubuntu to develop and test the tool. The tool may work under other operating systems as well, but we have not tried it out. Moreover we assume that there is an Erlang version installed on your computer. The model checker was developed under Erlang/OTP R13B; it will probably work under other releases of Erlang/OTP but we haven't verified this. The examples assume that you have installed McErlang so that the directory `$MCERLANG/scripts` is on your "command path", and similarly that the command `erl` for starting Erlang can be run without specifying a directory path.

1.1 Getting to know Erlang

Erlang is a functional programming language developed at Ericsson for the implementation of concurrent, distributed, fault-tolerant systems. A system is typically composed of a number of lightweight processes, which may be distributed over physically separated nodes, communicating through asynchronous message passing. On top of the basic language, the OTP library implements commonly used components such as client-server architectures, finite state machines, etc. Providing a model checker for Erlang is especially rewarding since the language is by now being seen as a very capable platform for developing industrial strength distributed applications with excellent fault tolerance characteristics. In contrast to most other Erlang verification attempts, we provide support for virtually the full, rather complex, programming language. The model checker has full Erlang data type support, support for general process communication, node semantics (inter-process communication behave subtly different from intra-process communication), fault detection and fault tolerance, and crucially can verify programs written using the high-level OTP Erlang component library (used by most Erlang programs).

For non-Erlang programmers the approach has merits too. Erlang is a good *general specification* language, due to its root in functional programming (with higher-order functions for writing concise specifications, with software components for lifting the specification abstraction level, can treat program as data – yielding a convenient meta level, . . .).

Moreover Erlang is a good platform for *specifying distributed algorithms* since language features matches common assumptions in distributed algorithms: (a) *isolated* processes that communicate using message-passing, (b) *fault-detection* is achieved using unreliable failure detectors, (c) process *fairness* built into the language, and (d) *locality* is important: communication guarantees are stronger for intra–node communication than inter–node communication.

The user manual assumes a basic knowledge of Erlang. Information on the programming language and its application programming interface can be found on the site www.erlang.org (where Erlang can be downloaded as well) and in a number of concerning Erlang programming which are available or will appear in print shortly:

- *Concurrent Programming in Erlang (2nd edition)* by Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams. Prentice-Hall, 1996.
- *Programming Erlang: Software for a Concurrent World* by Joe Armstrong. Pragmatic Bookshelf, 2007.
- *Erlang Programming* by Francesco Cesarini and Simon Thompson. O'Reilly Media, Inc., 2009.

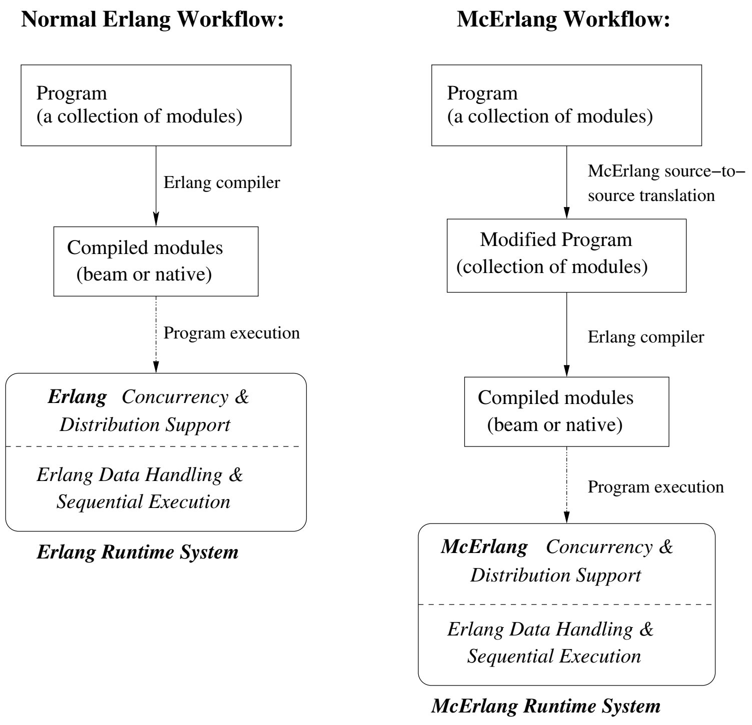

Figure 1: Usage of McErlang

1.2 The McErlang Workflow

Figure 1 illustrates the differences between a normal Erlang workflow (left) and one using the McErlang model checker (right). The model is the Erlang program to be analyzed, which undergoes a source-to-source translation to prepare the program for running under the model checker. Then the normal Erlang compiler translates the program to either Beam byte code (an Erlang byte code language) or directly to native machine code. Finally the program is run under the McErlang run time system, under the control of a verification algorithm, by the normal Erlang bytecode interpreter. The pure computation part of the code, i.e, code with no side effects, including garbage collection, is executed by the normal Erlang run time system. However, the side effect part is executed under the McErlang run time system which is a complete rewrite in Erlang of the basic process creating, scheduling, communication and fault-handling machinery of Erlang (comprising a significant portion of the code of the model checker).

Naturally the new run time system offers easy checkpointing (capturing the state of all nodes and processes, of the message queues of all processes, and all messages in transit between processes), of the whole program state as a feature (impossible to achieve in the normal Erlang run time system due to the physical distribution of processes).

1.3 A Smallish Example

We illustrate the use of the tool with the smallish example below. Two processes are spawned, the first starting an “echo” server that echoes received messages, and the other process invokes the echo server:

```
-module(example).
-export([start/0]).

start() ->
  spawn(fun() ->
            register(echo,self()), echo()
        end),
  spawn(fun() ->
           echo!{msg,self(),'hello␣world'},
           receive
             {echo,Msg} -> Msg
           end
        end).

echo() ->
  receive
    {msg,Client,Msg} ->
      Client!{echo,Msg},
      echo()
  end.
```

Let's run the example under the standard Erlang runtime system:

```
> erlc example.erl
> erl
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] ...

Eshell V5.6.5  (abort with ^G)
1> example:start().
<0.34.0>
2>
```

That worked fine. Let's try it under McErlang instead. To do so we first have to recompile the module using the McErlang compiler:

```
> mcerl_compiler -sources example.erl -output_dir .
```

Then we run it:

```
> mcerl
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] ...

Eshell V5.6.5  (abort with ^G)
1> mce:apply(example,start,[]).
Starting McErlang model checker environment version 1.0 ...
...

Process ... exited because of error: badarg


*** User code generated error
exception error due to reason badarg
Stack trace:
  mcerlang:resolvePid/2
  mcerlang:send/2
  ...

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

Ah, so there was an error. Let's find out more:

```
2> mce_erl_debugger:start(mce:result()).
Starting debugger with a stack trace; execution terminated with status:
  user program raised an uncaught exception.
...

stack(@2)> where().
2:

1: process <node0@exodo3,3>:
    run #Fun<example.2.125316937>([])
    process <node0@exodo3,3> died due to reason badarg

0: process <node0@exodo3,1>:
    run function example:start([])
    spawn({#Fun<example.1.27838786>,[]},[]) --> {pid,node0@exodo3,2}
    spawn({#Fun<example.2.125316937>,[]},[]) --> {pid,node0@exodo3,3}
    process <node0@exodo3,1> was terminated
    process <node0@exodo3,1> died due to reason normal
```

Apparently in one execution trace the second process spawned (the one calling the echo server) was run before the echo server itself, and of course upon trying to send a message `echo!{msg,self(),'hello world'}` the `echo` name was not registered.

1.4 Limitations

The translator handles full Erlang, however there are a number of issues involving the translation of the receive construct of Erlang (see Section 2.2 for details). Nevertheless a particular state space exploration algorithm (see Section 3.2.2) may impose additional limitations. Concretely the `mce_alg_safety` and `mce_alg_buechi` model checking algorithms (for checking safety and liveness properties respectively) currently implements neither a real-time nor a discrete-time semantics for Erlang. Instead, a receive statement with a timeout clause

```
receive
  Pattern1 when Guard1 -> Expr1;
  ...
  PatternN when GuarnN -> ExprN
  after Timeout -> Expr
end
```

where `Timeout` is not `infinity`, is considered to have a transition enabled whenever there is no element in the process mailbox that matches any pattern and guard. This treatment can be changed by setting the option (see Section 3.1) `is_infinitely_fast` to `true`; then non-timeout transitions prohibit non-zero timeouts from occurring. This corresponds to the assumption that the system is infinitely fast compared to any timers set.

The simulation algorithm (`mce_alg_simulation`) does implement a real-time semantics.

1.5 Extensions

The main addition to the basic Erlang language is a nondeterministic feature. By calling the function `mce_erl:choice` with a set of function applications as arguments, McErlang will randomly select one of the functions as its continuation.

```
mce_erl:choice(Funs::fun_spec())
   fun_spec() = {Fun::fun(),ArgList::[term()]}
              | {Module::atom(),FunctionName::atom(),ArgList::[term()]}
```

The example below builds a set of continuation alternatives using the list comprehension construct:

```
mce_erl:choice
  ([{vodka,select_movie,[{lookup,X,[Y]}]}
    || X <- [casablanca,jaws,volver],Y <- [1,2]])
```

To execute the nondeterministic construct McErlang will choose randomly one of the continuation alternatives whereas in model checking mode, all alternatives will of course be investigated.

Node and Global Dictionaries In addition to the normal process dictionary McErlang also implements a node dictionary and a global dictionary. These new dictionaries implement the same application interface as the process dictionary, except every function is suffixed with "n" for the node dictionary, and with "g" for the global dictionary. For example, the node dictionary can be accessed using the functions `mcerlang:nerase/0`, `mcerlang:nerase/1`, `mcerlang:nget/0`, `mcerlang:nget/1` and `mcerlang:nput/2`.

These additional dictionaries are useful to, for instance, implement a store that is persistent in case of process crashes (the node dictionary) or both process and node crashes (the global dictionary).

Node Functions New nodes are created automatically in McErlang; there is normally no need to explicitly create them. The `mcerlang:spawn(Fun::fun(),NodeName::atom())` function, for instance, will create a (simulated) node with the name `NodeName` if it doesn't already exist.

Nodes can also be explicitly started, and shutdown using the functions:

```
mcerlang:bringNodeUp(NodeName::atom())
mcerlang:bringNodeDown(NodeName::atom())
```

2 Compiling Erlang code 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.

2.1 Translation Phase

There are essentially two types of source-to-source transformations performed by the translation phase:

1. Programs running under the McErlang model checker need to invoke the McErlang application programming interface (e.g., the module `mcerlang`) instead of the normal Erlang API (e.g., the module `erlang`). Thus the transformation transforms the code so that e.g. instead of calling the function `erlang:send/2` the modified program calls `mcerlang:send/2`.

2. Receive statements cannot be executed directly in McErlang, as internally in McErlang communication between (simulated) processes does not use message passing. In fact, McErlang normally runs in a single process, regardless of how many processes are spawned by the simulated code. During translation receive statements are transformed into an expression which returns a special value (a kind of continuation), signalling the intention to execute a receive statement. To ensure that such special return values are not captured by the environment in which they reside, the transformation must also modify the environment. An example:

   ```
   echo() ->
     receive
       {msg,Pid,Msg} -> ok
     end,
     Pid!{echo,Msg}.
   ```

If we just transform the receive statement we would loose its special return value:

```
echo() ->
  [[receive
    {msg,Pid,Msg} -> ok
  end]],
  Pid!{echo,Msg}.
```

(where [[...]] performs the translation of a receive statement). Instead we (conceptually) embed the translated receive statement in a new let construct:

```
echo() ->
  let X =
    [[receive
      {msg,Pid,Msg} -> ok
    end]]
  in Pid!{echo,Msg}.
```

where the semantics that the argument part of the let construct are executed immediately, and if that part returns a special value, then a special value with the body part as continuation is returned. If the argument part returns a normal expression, then the body part is executed normally.

Thus to complete the transformation we need to know, for every function call `Module:Fun(Arguments)` in the code, whether the called function can ever execute a receive statement (and if so transform it, as well as the calling context, and its calling context, and so on). In other words, the transformation phase needs access to all source code modules comprising a program (including libraries), as it implements a global analysis identifying exactly which function calls can result in a (receive) side effect.

The transformation is implemented on the HiPE Core Erlang format. The standard Erlang compiler generates Core Erlang code, which is subjected to a number of transformations, and finally the standard Erlang compiler is used to generate beam (or native) code from the resulting transformed Core Erlang code.

The mapping of function calls, and information regarding which functions in binary modules (for which no source code is available) has side effects, is defined in a configuration file. This configuration file is read during the compilation phase; the default is to use "`$MCERLANG/configuration/funinfo.txt`".

We will describe the semantics of the transformation configuration file by explaining an excerpt:

```
[
 {gen_server,[{translated_to,mce_erl_gen_server}]},
 {supervisor,[{translated_to,mce_erl_supervisor}]},
 {gen_fsm,[{translated_to,mce_erl_gen_fsm}]},
 {erlang,[{rcv,false}]},
 {{erlang,spawn,4},[rcv,{translated_to,{mcerlang,spawn}}]},
 {{erlang,send,2},[snd,{translated_to,{mcerlang,send}}]},
 ...
]
```

The configuration information is represented as a normal Erlang term, and contains a number of commands, on two basic formats:

```
{module, [attribute1,...,attributeN]}
```

or

```
{{module,functionname,arity}, [attribute1,...,attributeN]}
```

As an example, consider the first line in the above specification:

```
{gen_server,[{translated_to,mce_erl_gen_server}]}
```

This command maps any call to a function in the gen_server module to a corresponding call in the module mce_erl_gen_server. The command {erlang, [{rcv,false}]} expresses that by default no function in the erlang module will ever execute a receive statement. In the next line we override this default by specifying that indeed erlang:spawn/4 (which spawns a function on a specified node) can actually cause a receive statement to be executed, and secondly that it should be mapped to calls to mcerlang:spawn/4 instead. Next the erlang:send/2 function is declared to cause a sending action (see explanation of the compiler option -sends_are_sefs).

Depending on the particular application requirements, it may be a good idea to use the OTP version of a particular module, or use a (simplified) module that we provide. Such "McErlang versions" of normal OTP modules normally reside in "$MCERLANG/lib/erlang/src" and its subdirectories.

2.2 Transformation Issues

Unfortunately the transformation is not fool-proof, the fundamental problem is that the transformation must potentially modify all the source code of an Erlang system. All the code locations (not in a tail position) from which a receive statement may eventually be called must be modified. As there is no Erlang source code for parts of the distribution (because they are implemented in C) we have no choice but to replace those parts with equivalent parts in pure Erlang.

As an example, consider the translation of the following code fragment:

```
lists:foreach(fun (Element) ->
                 receive Command -> {Element,Command} end
               end L).
```

Apart from transforming the receive statement itself, the transformation must potentially also modify the anonymous function in which the receive statement resides. Here this is not necessary, as the receive statement occurs in a tail position. Moreover, the transformation must also modify the lists:foreach function. Obviously we can use the source code of the lists module; however there is a problem. Not all functions provided by the lists module are actually implemented in Erlang. Thus we have no choice but to provide a slightly modified version of the module (mce_erl_lists) with McErlang. By default the modified library module is used, however a user can choose to use the standard module instead (by overriding the mapping from lists to mce_erl_lists in "funinfo.txt"). This may be safe, if no function passed to any function in lists can ever execute a return statement (the static analyser that the transformation phase implements is generally not intelligent enough to discover such possible optimisations).

As we have not analysed all standard Erlang modules for such problems, nor have we provided alternative implementations for all cases needed, there is a risk that a program may exhibit different behaviour running under McErlang than under the normal Erlang runtime system. Concretely, functions may have arguments which contain unexpected special return values, or special return values may be silently ignored.

Another concern is memory efficiency. In a model checker it is generally important to save memory space. Thus some data structures, although highly efficient in terms of execution speed, can waste too much memory space (examples may include the dict module).

2.3 Compilation

To prepare a set of Erlang source code files for use in McErlang (i.e., to map function calls and translate receive statements) and compile the resulting modules into object code the mcerl_compile script is used[1]. If a particular module should not be subjected to McErlang code transformations one can add the compiler directive

```
-language(erlang).
```

anywhere in the source code of the module. The Macro McErlang is defined if a file is compiled using the McErlang compiler. A trivial example:

[1]it calls functions in the mce_erl_compile module; see the EDoc documentation for more details.

```
-ifdef(McErlang).
-define(Output(),io:format("McErlang␣is␣here!~n",[])).
-else.
-define(Output(),io:format("No␣McErlang...~n",[])).
-endif.
```

Note that the `McErlang` macro will be defined if a file is compiled using McErlang, even if no transformations are done because the file contains a `-language(erlang).` directive.

The `mcerl_compile` script accepts the following parameters:

-
  ```
  -sources file_or_directory1.erl
           ...
           file_or_directoryN.erl [options]
  ```

  Specifies that the corresponding files should be compiled. Multiple `-sources` specification are allowed. If a directory is specified, all the Erlang files in that directory are selected for compilation. Options are of two kinds, and only apply to a single `-source` specification:

  - `-recursive` specifies that all subdirectories of the source specifications should be searched, recursively, for Erlang source files as well.
  - `-include_dirs directory1 ... directoryN` specifies that the specified directories are searched for include files by the preprocessor.

- `-libs directory1 ... directoryN`

  Used to add library directories containing alternative implementations of standard Erlang modules. The order in which libraries are added is relevant for searching for files. That is, `directory1` is searched before directory `directory2`. The directory "`$MCERLANG/lib/erlang/src`" is the default location to search for libraries.

- `-output_dir directory`

  Specifies the directory where the resulting .beam files are written (and where for debugging purposes .core files containing HiPE Core Erlang code are written as well). The default is to store .beam files in a directory "`ebin`", which shall be created before compilation.

- `-funinfo configuration_file`

  Used to provide an alternative transformation configuration file (see Section 2.1 above).

- `-verbose [true|false]`

  Used to provide more verbose output. If no argument is specified `true` is assumed. The default is `false`.

- `-unknown_is_rcv [true|false]`

  Controls whether unknown functions are considered to have (receive) side effects, for the purpose of the static analysis (this is safe default option). The default is `true`.

- `-sends_are_sefs [true|false]`

  An experimental option that causes actions that cause communications to be considered side effects as well. Functions that are normally declared as potentially causing communications include `erlang:link/1`, `erlang:monitor/2`, `exit/2`. The default is `false`.

An example make file which compiles all the Erlang source files in the current directory, and stores the beam files in "`../ebin`", and which refers to include files in "`include`", is provided below:

```
sources = $(wildcard *.erl)
beams = $(patsubst %.erl,../ebin/%.beam,$(sources))

$(beams): $(sources)
        mcerl_compiler -output_dir ../ebin \
        -sources \$(sources) \
        -include_dirs include
```

2.4 Customized Erlang/OTP Modules

In the distribution of McErlang we include tailored versions of some of the more commonly used Erlang/OTP libraries. The reason is twofold: sometimes they use features currently not translatable or implementable in McErlang (a case in point is `lists` which contains fragments implemented in C), or whose implementation may be too inefficient in terms of memory use. A user may choose to use these modules or the standard ones (through the mechanism of mapping function calls described in Section 2.1).

Below we list these modules and the status of their current implementation (beware that changes occur frequently, consult the sources for exact status). These files reside in the directory `lib/erlang/src` unless otherwise indicated.

`mce_erl_ets.erl`	Reimplements parts of the `erl_ets` library. The implementation is timewise inefficient compared to the original library.
`mce_erl_gen_event.erl`	Reimplements the major part of the `gen_event` library. Hot code swapping is not supported.
`mce_erl_gen_fsm.erl`	Reimplements the major part of the `gen_fsm` library. Hot code swapping is not supported.
`mce_erl_lists.erl`	An imported version of the `lists` library where the `member` function (and others) is still implemented in Erlang.
`mce_erl_rpc.erl`	Reimplements a small portion of the `rpc` library.
`mce_erl_supervisor.erl`	Reimplements a small portion of the `ev_supervisor` library, essentially only supporting the creation of a process tree from a textual description, but does not support monitoring the created process tree.
`mce_erl_gen_server.erl`	Reimplements the major part of the `gen_server` library. Hot code swapping is not supported.
`mce_erl_gen_server/no_tag/` `mce_erl_gen_server.erl`	Reimplements the major part of the `gen_server` library. Hot code swapping is not supported. Is timewise more efficient for checking generic server systems where clients always use the proper API (`gen_server:call`), and which does not implement fault detection or fault recovery.

3 Running McErlang

To execute a program under McErlang, one of the functions in the `mce` module should be called. A convenient way to start Erlang with all the modules comprising the McErlang tool available is to run the script `mcerl`.

Functions for starting McErlang:

- `mce:start(Conf::conf())`

  Starts McErlang using the configuration specified in `Conf` (see below for details).

- `mce:`**`apply`**`(Module::`**`atom`**`,FunName::`**`atom`**`,Arguments::`**`[`**`term()`**`]`**`)`
  `mce:`**`apply`**`(Module::`**`atom`**`,FunName::`**`atom`**`,Arguments::`**`[`**`term()`**`]`**`,Conf::conf())`

Starts McErlang using the configuration specified in `Conf` – if specified – to check the specified function. If a configuration is not provided, then a default one is used (using a safety checking algorithm, not checking any monitor, . . .).

- `mce:shell()`

  Starts a rudimentary shell for experimenting (using a simulation algorithm) with the McErlang tool.

3.1 Model Checking Configuration

The configuration of a model checking run is specified using a record structure (`mce_opts`). The best way of making the definition of `mce_opts` available to the Erlang shell is through the following command:

```
1> rr(mce:find_mce_opts()).
```

The relevant fields of are enumerated below:

-
  ```
  program = {Module::atom(),FunName::atom(),Arguments::[term()]}
          | {Function::fun(), Arguments::[term()]}
  ```

  Specifies the initial function of the program.

- `algorithm = {Module::atom(),InitArg::term()}`

  Specifies the type of algorithm used for traversing the program state space (see Section 3.2.2 below for details on algorithms). The default algorithm is `mce_alg_safety`.

- `monitor = {Module::atom(),InitArg::term()}`

  Specifies the correctness monitor used to check the behaviour of the program. Monitors are either *safety monitors* which should be checked in every program state, or *büchi monitors* which encode linear temporal logic properties (LTL). The `Ltl2Buchi` tool can be used to translate LTL properties into büchi monitors (see Section 4 for details). Further details on monitors are given in Section 3.3. The default monitor never reports an error.

- `abstraction = {Module::atom(),InitArg::term()}`

  Abstraction implementation (see Section 3.4 for details). The default abstraction preserves all properties of the states and actions to which it is applied.

- `table = {Module::atom(),InitArg::term()}`

  State table implementation (see Section 3.5 for details). The default table hashes on the stored states, but does not store transitions.

- `stack = {Module::atom(),InitArg::term()}`

  Stack implementation (see Section 3.6 for details). The default stack implementation is `mce_stack_list`.

- `scheduler = {Module::atom(),InitArg::term()}`

  A scheduler determines which transitions are taken (in simulation mode, see Section 3.7 for details). The default scheduler (`mce_sched_rnd`) selects randomly a new transition.

There are also a number of fields, which when specified, can change the behaviour of state space exploration algorithms and the McErlang runtime system.

- `sim_external_world = true() | false()`

If this flag is set the model checker will interface with the external world in simulation mode (i.e., will receive messages sent from other processes outside the simulation).

- `pathLimit = int()`

  Limit execution paths to a maximum depth (not set by default).

- `shortest = true() | false()`

  If the flag is set tries to compute the shortest path to failure (default false).

- `terminate = true() | false()`

  If set, the runtime system will randomly terminate processes (default false).

- `notice_exits = true() | false()`

  Warn when a process terminates abnormally due to an uncaught exception (default true).

- `fail_on_exit = true() | false()`

  Stop a model checking run if a process terminates abnormally due to an uncaught exception (default true).

- `is_infinitely_fast = true() | false()`

  Prohibits (non-zero) timeouts (caused by `after` clauses in `receive` statements) from occurring if non-timeout transitions are enabled. This corresponds to the assumption that the system is infinitely fast (default false).

- `sim_actions = true() | false()`

  Print actions that occur during a simulation run (using the debugger or the algorithm `mce_alg_simulation`) (default false).

- `seed = {int(),int(),int()}`

  Specify the initial seed (used by the randomised scheduler).

- `record_actions = true() | false()`

  Record seen actions during a model checking run (default true).

- `random = true() | false()`

  Randomize the order of transitions. This option is typically used when using a safety or liveness model checking algorithm, as the specified scheduler (using `scheduler=`) is not used for such algorithms (default false).

- `time_limit = int()`

  Stops model checking after the time limit on the running time of the verification, specified in seconds, has been reached (default unlimited), with an inconclusive result.

- `rpc = true() | false()`

  Support for the `rpc` module re-implementation in McErlang (default false).

- `small_pids = true() | false()`

  Controls whether McErlang tries to aggressively reuse process identifiers as soon as possible. Such reuse is costly in terms of execution speed, but generally necessary to obtain finite models during model checking. Default true for model checking algorithms, and false for simulation algorithms.

3.2 State Space Traversal Algorithms

An algorithm determines the particular state space exploration strategy used by McErlang. If no algorithm is specified, the default one is `mce_alg_safety`. The result of a model checking run is a "result value" which can be inspected using the functions in the `mce_result` module. The result value is normally stored in the process dictionary under the key `result`. If the result represents an error (a failed monitor, a process died normally due to an exception) the behaviour of the program up to the point of the error can be studied in further detail in the McErlang debugger using the command:

```
mce_erl_debugger:start(mce:result()).
```

3.2.1 Algorithm Termination

A verification may fail if the state space is too large to explore given the amount of memory available, this happens fairly frequently in model checking as most algorithms need to store encountered states in a state table. A possibility to alleviate this problem is to select the `mce_table_bitHash` table module for storing encountered states (see Section 3.5 for details). Another option is to use a bounded state stable, available as the module `mce_table_hash_bounded`.

Even though the state table size may be bounded, the verification algorithms may still run out of memory if the "verification stack", which stores the list of encountered states in a single program trace, grows too long (thus consuming too much memory). This problem can occur if the module is faulty, for instance if an ever increasing counter is included in the program. Such counters are, potentially, process identifiers, references (created using `erlang:make_ref()`, timestamps (i.e., calling `erlang:now()`) and so on. However, McErlang makes sure, when model checking but *not* when simulating code, that process identifiers and references are not represented by ever increasing counters but selected *fresh* (i.e., as the least unused value in some ordering of pids and references). A good method to detect problems with long verification stacks is to regularly output created states and to manually inspect them to see if some part of them look amiss (such printouts may be inserted in a custom monitor for instance). There are other fixes as well: the length of the verification stack may be limited using the `pathLimit=N` option (in the `mce_opts` record), and secondly a bounded stack implementation may be specified (using for instance a directive `stack={mce_stack_bounded,Size}` also in the `mce_opts` record, see Section 3.6 for details).

To help detect such problems the model checking algorithms `mce_alg_safety` and `mce_alg_behaviour` regularly print out statistics regarding the number of states generated, and the length of the verification stack:

```
Path depth at 10000 entries
```

This indicates that the length of the maximum execution path has reached 10000 entries.

```
Generated states 70000; checked states 247709; relation 3.5387
```

The second printout indicates that 70000 distinct program states have been encountered, while the verification run has generated 247709 (non-unique) states in total, for a ratio of 3.5387.

3.2.2 Listing of Available Algorithms

- `mce_alg_simulation`

  This algorithm implements a basic simulation algorithm. In every encountered program state only a single transition is chosen. During the simulation a safety program monitor is checked.

  If you wish to allow your program to communicate with the outside McErlang (accept messages sent by other processes) the `sim_external_world` option in `mce_opts` should be set to true. An example specific scheduler can be specified using the `scheduler` field in `mce_opts`. This algorithm does not require any initial argument.

  Suppose as an example that we want to run the `example:start` example in Section 1.3 using the simulation algorithm:

```
> mcerl
Erlang (BEAM) emulator version 5.6.5 [source] [smp:4] ...

Eshell V5.6.5  (abort with ^G)
1> rr(mce:find_mce_opts()).
[mce_opts]
2> mce:start(#mce_opts{program={example,start,[]},
                       algorithm={mce_alg_simulation,void}}).
Starting McErlang model checker environment version 1.0A ...

Starting mce_alg_simulation(void) algorithm on program
example:start()
with monitor mce_mon_test(ok)

...
Execution terminated normally
Access result using mce:result()
ok
3>
```

Aha, no error this time. This is because we only explored one execution trace. Note that we in the first line read in the record definition of `mce_opts`, so that we are allowed to use the syntax `#mce_opts{program=...}` in the Erlang shell.

- `mce_alg_debugger`

  Runs the debugger on the selected program, see Section 3.8. This algorithm does not require any initial argument.

- `mce_alg_safety`

  Checks the specified monitor, which *must* be of type `safety`, on all program states of the program. Note that the option `sim_external_world` has no effect (the algorithm makes the "closed world assumption", whereby no message ever arrives from outside the specified program). This algorithm does not require any initial argument.

- `mce_alg_buechi`

  Checks the specified monitor, which *must* be of type `buechi`, on the program. Note that the option `sim_external_world` has no effect (the algorithm makes the "closed world assumption", whereby no message ever arrives from outside the specified program). This algorithm does accept an initial argument, `{process_fairness,bool()}`, which specifies that the program should be checked under a process fairness assumption if the `bool()` option is true (the default value of the parameter is true).

- `mce_alg_safety_parallel`

  Checks the specified monitor, which *must* be of type `safety`, on all program states of the program. This algorithm makes use of the symmetric multiprocessing implementation of Erlang to use, potentially, multiple processors or cores to speed up the model checking run. It accepts an initial argument specifying the number of Erlang process schedulers to run (there may be more process schedulers running that the number of processor elements available). An example:

```
mce:start(#mce_opts{program={example,start},
                    algorithm={mce_alg_safety_parallel,4}})
```

checks the `example:start` program using 4 process schedulers. Note that the option `sim_external_world` has no effect (the algorithm makes the "closed world assumption", whereby no message ever arrives from outside the specified program).

- `mce_alg_combine`

  This algorithm provides a method to combine to other state space exploration algorithms. It accepts a parameter which enumerates the two different sub-algorithms. An example:

  ```
  algorithm=
    {mce_alg_combine,
       {#mce_opts{algorithm={mce_alg_simulation,void},
                            scheduler={locker_sched,void}},
        #mce_opts{algorithm={mce_alg_safety,void},
                            monitor={monMutex,[]},
                            table={mce_table_hashWithActions,[]}}}}
  ```

  The example specifies that first a simulation algorithm should be run, using a custom scheduler `locker_sched`, and when the simulation terminates (successfully), the safety model checking algorithm `mce_alg_safety` algorithm should be run.

  In practise this algorithm can be used to throw away an initial segment of the behaviour of a program which is deemed uninteresting, to permit focusing on a latter part. A particular example is to check only a system after when it is has stabilised after the completion of the starting-up phase (perhaps programmed using the OTP supervisor behaviour `gen_supervisor`). It is a method to save memory space.

3.3 Monitors

A monitor is conceptually an observer which examines program states, and program actions, and which is run in lock-step with the the program to verify. A monitor has an internal state.

Upon examining a state, and the program action leading to the state from the previous state, a monitor can either return a set of new monitor states (signalling an acceptable program state), or an error condition signalling that the program state failed the monitor.

There are two types of monitors in McErlang: "safety monitors" which are deterministic (i.e., return only a single next monitor state) and "büchi monitors" which implement Büchi automata. Such monitors may be nondeterministic, and in addition each monitor state is marked either "accepting" or "non-accepting". The `mce_alg_buechi` verification algorithm signals an error if it discovers an infinite loop (through the combined program state and monitor state graph) containing only accepting monitor states.

In the McErlang repository we have included an associated tool, Ltl2Buchi, written by Hans Svensson, which can automatically translate a formula in Linear Temporal Logic (LTL) to a corresponding Büchi automaton. See Section 4 for details.

In the code listing in Figure 2 we show the source code for a simple safety to detect deadlocks. A monitor should be an Erlang/OTP behaviour that conforms to the `mce_behav_monitor` (see `www.erlang.org` for details on behaviours, in reality the only requirement on a McErlang monitor is that it exports three functions: `monitorType/0`, `init/1` and `stateChange/3`. A "Büchi" monitor must also export a function `stateType/1` which marks a state as either `accepting` or `nonaccepting`.

The `stateChange/3` function gets called by the chosen verification algorithm using three parameters:

```
stateChange(ProgramState,MonitorState,VerificationStack)
```

In the example the safety monitor examines the program state, and if all processes are deadlocked, an atom `deadlock` is returned (signalling a monitor failure), and if a non-deadlock process exists a new monitor state `{ok,MonState}` (identical to the old state) is returned.

We can check a general program with the above monitors by specifying it in a `mce_opts` record:

```
-module(mce_mon_deadlock).
-export([init/1,stateChange/3,monitorType/0]).

-include("state.hrl").
-include("process.hrl").
-include("node.hrl").

-behaviour(mce_behav_monitor).

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

stateChange(State,MonState,_) ->
  case is_deadlocked(State) of
    true -> deadlock;
    false -> {ok, MonState}
  end.

is_deadlocked(State) ->
  State#state.ether =:= [] andalso
  case mce_erl:allProcesses(State) of
     [] -> false;
     Processes ->
       case mce_utils:find
            (fun (P) -> P#process.status =/= blocked end,
             Processes) of
         {ok, _} -> false;
         no -> true
       end
  end.

monitorType() -> safety.
```

Figure 2: The `mce_mon_nondeadlock` monitor

```
mce:start(#mce_opts{program={example,start,[]},
                    monitor={mce_mon_nondeadlock,void}}).
```

3.3.1 Observational Power of Monitors

The above example only inspects the program state parameter to the `stateChange` function. The application programming interface for examining program states is stable with regards to examining invariant state components (process mailboxes, process status, node names, process dictionaries, etc) but there is currently no well-defined mechanism for retrieving the value of program variables from a program state[2].

An alternative is to instead inspect the actions a program causes. The third parameter to `stateChange` is the verification stack, which contains the actions of the programs (if the field `record_actions` in `mce_opts` is not false). The actions that the program performed after leaving the program state, and before entering the program state now checked is returned as a list by the following code fragment:

```
  {Entry,_} = mce_behav_stackOps:pop(VerificationStack),
  Entry#stackEntry.actions.
```

The McErlang modules `mce_erl` and `mce_erl_actions` contains application interfaces for examining program actions.

A convenient means to determine if a program has performed a certain action is to instrument said program with synthetic probe actions, and during model checking determine if these synthetic actions are present. A program can cause a probe action by calling the `mce_erl:probe(Label::term(),Term::term())` or `mce_erl:probe(Label::term())` functions (where `Label` and `Term` are general terms).

As an example we show a fragment of a client that allocates resources by sending a request message to a (`gen_server`), and then releasing the resource by sending a new message:

```
  gen_server:call(Locker, request),
  mce_erl:probe(inUse),
  gen_server:call(Locker, release)
```

The property we want to establish is mutual exclusion, i.e., that there are not two clients simultaneously accessing the shared resource. To do this we instrument the client to perform a probe action whenever a resource has been granted, and before it is has been released.

We can then check for the existence of a "mutual exclusion failure" using the following safety monitor fragment:

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

stateChange(_State,init,Stack) ->
  case has_probe_action(actions(Stack)) of
    {true, inUse} -> {ok, inUse};
    no -> {ok, init}
  end;
stateChange(_State,inUse,Stack) ->
  case has_probe_action(actions(Stack)) of
    {true, release} -> {ok, init};
    {true, inUse} -> {inUse, twice};
    no -> {ok, inUse}
  end.

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

[2]This is still possible in many cases, but we prefer not to document the mechanisms here as they are fragile causing the retrieval of variable values to break if changes are made to the translation strategy of McErlang. We plan to rectify this problem in a new version of McErlang.

```
    void -> [];
    Other -> Other
  end.

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

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.

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. Internally they are stored in the global system dictionary (accessible using the McErlang API functions `gget/1`, et.c.).

A probe state is asserted using the functions

```
mce_erl:probe_state(Label::term())
mce_erl:probe_state(Label::term,Term::term)
```

A probe state must be explicitly retracted using the function

```
mce_erl:del_probe_state(Label::term())
```

when it no longer should hold.

We can test for the existence of a probe state using the function

```
mce_erl:has_probe_state(Label::term(),ProgramState::mcErlangState()) ->
  bool()
```

and the term corresponding to a probe state label for an enabled probe state can be retrieved using the function:

```
mce_erl:get_probe_state(Label::term(),ProgramState::mcErlangState()) ->
  term()
```

3.3.2 Some Available Monitors

Most monitors are rather application specific, but there are some generic ones implemented. Note also that in some situations it may not be necessary to specify any monitor at all; the default monitor is the always true one, but since the default setting for the `fail_on_exit` flag of `mce_opts` is true, the model checker will verify that no process is ever abnormally terminated.

Below we show the tuples naming the monitors, and their respective arguments (to be specified in the `monitor` field of `mce_opts`).

- `{mce_mon_queue, MaxQueueSize::int()}`

  Checks that all queues contain at most `MaxQueueSize` elements.

- `{mce_mon_deadlock, Any::any()}`

  Checks that there is at least one non-deadlocked process.

3.4 Abstractions

Abstractions are functions that abstract the state space, or actions. The model checking algorithms (e.g., `mce_alg_safety`) uses such an abstraction to compute a "normal form" for a state before checking whether a newly generated state has been previously visited (checking for membership in the state table).

As an example we show the code for the `mce_abs_hash` abstraction module:

```
-module(mce_abs_hash).
-export([init/1, abstract_actions/2,abstract_state/2]).
-behaviour(abstraction).

init(Size) -> {ok,Size}.
abstract_actions(A,Size) -> {A,Size}.
abstract_state(State,Size) -> {erlang:phash2(State,Size),Size}.
```

Such abstraction modules should conform to the `mce_abs_hash` behaviour, i.e., export an `init` function, a state abstraction function `abstract_state` and an action abstraction function `abstract_actions`. An abstraction module can use a private state, which is passed along as a parameter and is kept using the verification algorithm.

Concretely the `mce_abs_hash` module above implements a "hash abstraction", whereby the whole program state is mapped to a hash value (an integer). The size of the hash table is sent along as the private data to the abstraction functions.

3.5 Tables

Tables records encountered program states (typically using a hash table). The following tables are implemented:

- `mce_table_hash`

  implements a hash table using an Erlang `ets` table.

- `mce_table_hashWithActions`

  implements a hash table using an Erlang `dict` data structure, also storing actions between states.

- `mce_table_bitHash`

  implements a hash table where states are hashed into an integer value, and there is no collision handling. That the actual states themselves are not stored in the table, only a bit indicating whether the table entry contains an element or not. Note that this algorithm is not currently available under Windows as it relies on a HiPE private function for efficiency, which is not yet implemented on that operating system.

- `mce_table_hash_bounded`

  implements a hash table with a bounded maximum size. If during verification the size of the hash table has to be increased, verification stops and McErlang returns an inconclusive result.

3.6 Stacks

A stack is used to store verification stack, i.e., the program states (together with an associated monitor state – a *verification state*) encountered on the run from the initial verification state to the current one.

Available stacks:

- `mce_stack_list`

  An unbounded list based stack implementation.

- `mce_stack_bounded`

  An bounded stack implementation, which accepts as parameter the bound. If the stack overflows, the most recent entries overwrites the oldest ones. That is, it can be specified as follows in the `mce_opts` record:

```
-module(locker_sched).
-export([init/1,choose/4,willCommit/1]).

-behaviour(mce_behav_scheduler).

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

willCommit(_) -> true.

choose(Transitions,SchedState,Monitor,Conf) ->
  FilteredTransitions =
    lists:filter
      (fun reqFilter/1,
       lists:map
       (fun (T) -> mce_erl_opsem:commit(T,Monitor,Conf) end, Transitions)),

  case length(FilteredTransitions) of
    N when N>0 ->
      SelectedNumber = random:uniform(N),
      {ok,{lists:nth(SelectedNumber,FilteredTransitions),SchedState}};
    0 -> no_transitions
  end.

reqFilter({Actions,_}) -> not(lists:any(fun send2locker/1, Actions)).

send2locker(Action) ->
  case mce_erl_actions:is_send(Action) of
    true -> mce_erl_actions:get_send_pid(Action)=:=locker;
    false -> false
  end.
```

Figure 3: A custom scheduler

```
#mce_opts{stack={mce_stack_bounded,Bound::int()},...}
```

- `mce_stack_onePlace`

  A bounded stack with just one place.

3.7 Schedulers

A scheduler determines the scheduling policy during a simulation run (using the `mce_alg_simulation` algorithm). Figure 3 illustrates a particular implementation.

The scheduler forbids any action which is the result of sending a message to a process with the registered name `locker`. This scheduler is used in the example on page 15 to steer a simulation algorithm, which is followed by running a model checking algorithm (`mce_alg_safety`). The idea is that the simulation part should explore the initial startup of the system, forbidding any action that involves actually requesting resources from the `locker` process. Once there is no schedulable transition (except the ones involving sends to the locker) the simulation phase ends, and model checking begins.

3.8 The Debugger: Examining Counterexamples

The result of a McErlang run is normally saved in the process dictionary under the key `result`. In case a counterexample has been found the run can be examined further using the McErlang debugger which can be started using the function call `mcerlang:start(mce:result())`. The debugger is also useful for exploring the execution of a program in a step-by-step fashion.

The debugger is used to examine the execution stack of the failed program, using the set of commands enumerated below. Conceptually the user can move forwards and backwards in the stack to examine its content, single step through the transitions of a program, etc.

3.8.1 Starting the Debugger

The debugger can be started in a number of ways. If a counterexample has been produced, and is stored in the process dictionary under the key `result`, the normal manner is to invoke the debugger using the `start` function applied to `mce:result()`.

-
  ```
  mce_erl_debugger:start(Object::debugger_object())
            debugger_object() = stack() | mce_result()
  ```

  The result of a model checking run has the type `mce_result()`; such a result may contain an execution stack (of type `stack()`) which is needed to run the debugger.

It is not necessary to generate a counterexample before starting the debugger, another method to begin a debugging session is to specify the algorithm `mce_alg_debugger` in the `mce_opts` structure:

```
mce:start
  (#mce_opts
   {program={Module,Program,Args},
    algorithm={mce_alg_debugger,void}}).
```

In addition the `mce:debug/N` function calls start the debugger directly (by specifying the `mce_alg_debugger` algorithm).

3.8.2 Debugger Commands

These are commands available in the command loop of the model checker, in addition to the normal Erlang functions.

- `int()`
  `choose(N::int())`

  Chooses the corresponding transition to compute the next state.

- `step()`
  `step(N::int())`

  Executes `N` computation steps.

- `run()`

  Runs the program until it halts.

- `printTransitions()`

  Shows all transitions from the current state.

- `printState()`
  `printEther()`
  `printNodeNames()`
  `printProcessNames()`
  `printNode(NodeName::`**`atom`**`())`
  `printProcess(Pid::`**`pid`**`())`

  Prints different components of the current state.

- `showExecution()`

  Show the complete run starting from initial state to the final state.

- `where()`
  `where(N::int())`

  Shows the execution context (the last `N` stack frames).

- `back()`
  `back(N::int())`

  Moves the "stack pointer" towards the beginning of the program execution.

- `forward()`
  `forward(N::int())`

  Moves the "stack pointer" towards the end of the program execution.

- `goto(N::int())`

  Moves the stack pointer to stack frame `N`.

- `quit()`

  Leaves the debugger.

3.9 The Shell: Experimenting with McErlang

McErlang implements a rudimentary shell which can be used to experiment with running code under McErlang. The shell is invoked using the function `mce:shell()`.

The shell supports an online mapping of Erlang API functions to McErlang ones. For example, the expression `self()!hello` will be mapped to `mcerlang:send(mcerlang:self(),hello)`. Moreover it is possible to execute a `receive` expression.

As an example, the following shell interaction works as expected, although internally the McErlang API is used instead of the Erlang one:

```
1> mce:shell().
...
McErlang::node0@sadrach> self().
{pid,node0@sadrach,1}

McErlang::node0@sadrach> self()!hello.
hello

McErlang::node0@sadrach> receive X -> {msg,X} end.
{msg,hello}

McErlang::node0@sadrach>
```

Arbitrary functions can be executed in the shell, logically executing of a particular node. To switch to executing on a different node the function `connect_to_node(NodeName::atom())` can be executed.

Limitations For technical reasons it is currently not possible to provide anonymous functions as arguments to `erlang:spawn`.

4 Ltl2Buchi: Translation LTL Properties to Büchi Automatons

The Ltl2Buchi tool, due to Hans Svensson at the IT University of Gothenburg, is used to translate LTL formulas into Büchi automata.

These are commands available in the command loop of the model checker, in addition to the normal Erlang functions.

The following functions are available in the module `mce_ltl_parse` for translating LTL formulas into Büchi automatons:

- `string(S::string()) -> ltl()`

  Parses an LTL formula.

- `ltl_string2module(S::string(),FileName::string()) -> atom()`

  Parses an LTL formula, converts the LTL formula into a Büchi automaton, and stores the automaton in `FileName`. Returns the name of the generated module.

- `ltl_string2module_and_load(S::string(),ModuleName::atom()) -> atom()`

  Parses an LTL formula, converts the LTL formula into a Büchi automaton, and stores the automaton in a temporary location. Returns the name of the generated module (`ModuleName`).

- `ltl2module(L::ltl(),FileName::string()) -> atom()`

  Converts the LTL formula into a Büchi automaton, and stores the automaton in `FileName`. Returns the name of the generated module.

- `ltl2module_and_load(S::string(),ModuleName::atom()) -> atom()`

  Converts the LTL formula into a Büchi automaton, and stores the automaton in a temporary location. Returns the name of the generated module (`ModuleName`).

4.1 Writing LTL Properties

LTL properties are composed using the module `ltl`. The following combinators are available:

```
ltl() = ltrue()
      | lfalse()
      | prop(proposition())
      | land(Phi1::ltl(), Phi2::ltl())
      | lor(Phi1::ltl(), Phi2::ltl())
      | lnot(Phi::ltl())
      | implication(Phi1::ltl(), Phi2::ltl())
      | equivalent(Phi1::ltl(), Phi2::ltl())
      | next(Phi::ltl())
      | always(Phi:ltl())
      | eventually(Phi::ltl())
      | until(Psi::ltl(),Phi::ltl())
      | release(Psi::ltl(),Phi::ltl())

proposition() = fun atom():atom()/int()
              | {atom(),atom()}
              | {var,atom()}
```

Note that we permit variables, denoted by a tuple `{var,atom()}`, in the specification of a proposition. Such variables may be instantiated at runtime.

There is also a LTL parser which given a string argument, accepts the following language:

$$
\begin{array}{rcl}
\mathit{formula} & ::= & (\mathit{formula}) \\
 & | & \mathit{proposition} \\
 & | & \mathit{formula}\ \texttt{until}\ \mathit{formula} \\
 & | & \texttt{always}\ \mathit{formula} \\
 & | & \texttt{next}\ \mathit{formula} \\
 & | & \texttt{eventually}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{or}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{and}\ \mathit{formula} \\
 & | & \texttt{not}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{release}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{implies}\ \mathit{formula} \\
 & | & \mathit{formula}\ \texttt{equivalent}\ \mathit{formula} \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
 & & \\
\mathit{proposition} & ::= & \texttt{fun}\ \mathit{atom}\texttt{:}\mathit{atom}\texttt{/}\mathit{integer} \\
 & | & \texttt{\{}\mathit{atom}\texttt{,}\ \mathit{atom}\texttt{\}} \\
 & | & \mathit{var}
\end{array}
$$

As in normal Erlang, a variable `var` begins with an uppercase letter. The binding power of the LTL operators range, from looser to tighter, `implies`, `equivalent`, `and`, `or`, `until`, `release`, `eventually`, `always`, `next`, `not`. Apart from writing the operators in textual form (e.g., `always`), the standard abbreviations and alternatives are available (e.g., `G`,`[]`).

4.2 Interfacing with Büchi Automatons

As an example we parse a LTL formula, convert into a Büchi automaton, and load the generated module `ltl_test` into the Erlang runtime system using the following code fragment:

```
Module =
  ltl_string2module_and_load
  ("always (P => eventually Q)",
   ltl_test).
```

The resulting module is used as an argument in the `monitor` field in the `mce_opts` record.

The generated büchi monitor is parametric on the variables that occur in the LTL formula (as propositional predicates). These parameters are instantiated using the argument in the specification of the `monitor` field, in the `mce_opts` record. Concretely, if an LTL formula f contains no variables then the corresponding module m (containing the Büchi automaton that is the result of translating f) should be specified as: `monitor={`m`,`*private*`}` where *private* is the initial "private" state of the proposition predicates (see below for explanation).

In case the generated monitor contains variables the `monitor=` specification accepts as argument a tuple containing two elements: the first is private data that predicate functions may initialise and use as they see fit, and the second is a list containing binary tuples with a predicate variable as the first element and the function implementing the predicate as the second element.

The specification of the `ltl_test` monitor above serves as a concrete example:

```
monitor=
  {ltl_test,                  %% Monitor name
  {void,                      %% Predicate function private data
   [{'P',fun p_mod:p_fun/3},
    {'Q',fun q_mod:q_fun/3}]}} %% Variable mapping
```

We remap the `P` variable to a call to the function `p_mod:p_fun/3` (and vice versa for the `Q` variable). The order in which the variable mappings are specified does not matter.

A function predicate is called by McErlang using three arguments:

```
predfun(ProgramState,Actions,PrivateState::term()) ->
   true | false | {true, term()}
```

where `ProgramState` is the current program state, `Actions` are the actions from the previous state, and `PrivateState` is the current private state. A function predicate may return either `true`, signalling that the predicate holds in the program state, `false` signalling that it doesn't, and `{true, NewPrivateState}` signalling that the predicate holds, and replacing the old private state with the second element of the returned tuple.

Note that the use of a private state for passing information between formula predicates is a rather fragile mechanism, but if used wisely it adds expressive power. Consider the following example of the specification of a resource usage scheme. We want to verify that a request for the use of a resource is always eventually followed by a release of the said resource. In LTL we can specify the property as follows, using the `ltl` module combinator functions:

```
ltl:always
  (ltl:implication
   (ltl:prop(p),
    ltl:next(ltl:eventually(ltl:prop(q)))))
```

or more symbolically, $G(p \Rightarrow X(F\ q))$.

The two predicate functions corresponding to p and q are coded as `does_request` and `does_release` below. They assume that the code of the client has been annotated with two probe actions: a release action with the atom `release` as label and a request action with the atom `request` as label.

```
does_request(_,Actions,_) ->
  case mce_utils:findret
    (fun (Action) ->
         try
           true = mce_erl_actions:is_probe(Action),
           request = mce_erl_actions:get_probe_label(Action),
           {ok,mce_erl_actions:get_source(Action)}
         catch _:_ -> false end
     end, Actions) of
    {ok, Result} -> {true, Result};
    _ -> false
  end.

does_release(_,Actions,ClientPid) ->
  lists:any
    (fun (Action) ->
         try
           true = mce_erl_actions:is_probe(Action),
           release = mce_erl_actions:get_probe_label(Action),
           ClientPid = mce_erl_actions:get_source(Action),
           true
         catch _:_ -> false end
     end, Actions).
```

The first function searches for probe actions labelled with `request`, and if one is found, returns the process identifier of the client process issuing the probe action as private date. The second function searches for probe actions labelled with `release` and which moreover was issued by the same client process as the `request` (checked via the `ClientPid` parameter).

Supposing that the translated property is stored in the file `always_eventually`, the above predicate functions are located in the module `predfuns`, and the program is started using the function `resourcemanager:start()` then we can invoke a model checking run as follows:

```
mce:start(#mce_opts{program={resourcemanager,start,[]},
                    monitor={always_eventually,
                             {void,
                              {{predfuns,does_request},
                               {predfuns,does_release}}}},
                    algorithm={mce_alg_buechi,void}}).
```

5 A McErlang Application

The McErlang application provides an optional interface to McErlang configuration, and it keeps the configuration in its state in order to easily run several different model checking runs with the same configuration. This is feature is most notably used when invoking McErlang from QuickCheck[3].

The application can be started using either `mce_app:start/0` or the more formal `application:start(mcerlang)`.

6 A QuickCheck–McErlang Interface: Experimental Feature

The QuickCheck–McErlang interface is distributed with QuickCheck and provides the capability to use McErlang to verify QuickCheck parallel and sequential state machines. To start a verification run using the interface first the McErlang application must be started. An example QuickCheck property which uses McErlang as the verification engine is shown below:

```
prop_parallel_mcerlang() ->
  ?FORALL
  (PCmds,
   parallel_commands(?MODULE),
   ?MCERLANG
   ([...], % Names of modules in verified program
    {H,AB,Res},
    begin
      ... % Program setup
      run_parallel_commands(?MODULE,PCmds)
    end,
   ?WHENFAIL
   (io:format("Sequential: ~p\nParallel: ~p\nRes: ~p\n",
    [H,AB,Res]),
   Res == ok))).
```

For more information concerning the interface see the tutorial `eqc_mcerlang_tutorial.pdf` in the `doc` directory of the McErlang distribution.

[3]see `http://www.quviq.com`

7 Module Information

This section contains an enumeration of a number of modules in the McErlang source code that implement important functions:

`mce`	Contains functions to start the execution of McErlang
`mce_actions`	Action API (not Erlang specific)
`mce_result`	Handling of model checking results
`mce_erl_actions`	Action API (Erlang specific)
`mce_erl_compile`	Transforming Erlang code for running under McErlang
`mce_erl_debugger`	McErlang debugger
`mce_erl`	Functions for constructing and deconstructing McErlang terms

Consult the EDoc based documentation of McErlang for more details on these modules and their exported functions.