Erlang and the McErlang Model Checker

Lars-Åke Fredlund, Clara Benac Earle

Babel research group
Facultad de Informática, Universidad Politécnica de Madrid
Talk Overview

- The **Erlang** programming language
Talk Overview

- The **Erlang** programming language
- **McErlang**: a tool for model checking Erlang programs
  (short intros to model checking and linear temporal logic)
Erlang/OTP History

- Erlang language born in 1983 at Ericsson
- Used inside and outside Ericsson for implementing challenging concurrent and distributed applications
- Application example: High-speed ATM switch developed in Erlang (2 million lines of Erlang code), C code (350,000 lines of code), and 5,000 lines of Java code
- Other examples: parts of Facebook chat written in Erlang (70 million users), CouchDB (integrated in Ubuntu 9.10), users at Amazon, Yahoo, ...
- In Spain: Tuenti, LambdaStream (A Coruña), ...
- Open-source; install from http://www.erlang.org/
Erlang is becoming popular

C and C++ job offers over the last 5 years:

Erlang job offers the last 5 years:
Erlang as a source of inspiration

- Ideas from Erlang are also influencing other programming languages and libraries like Scala, Node.js, Clojure, …
- So lets see the main features…
Erlang/OTP

- Basis: a general purpose functional programming language
- Automatic Garbage Collection
- With lightweight processes (in terms of creation time and memory requirements). Typical software can make use of many thousands of processes; **smp** supported on standard platforms
- Support for fault-tolerance and distributed computation in the programming language!
- Implemented using virtual machine technology. Available on many OS:es (Windows, Linux, Solaris, ...)
- Supported by extensive libraries: **OTP** – open telecom platform – provides design patterns, distributed database, web server, etc
Erlang basis

A simple functional programming language:

- Simple data constructors:
  - integers (2), floats (2.3), atoms (hola), tuples ({2, hola}), and lists ([2, hola], [2|X]), functions, records (#process{label=hola}), bit strings (<<1:1, 0:1>>)
- Call-by-value
- Variables can be assigned once only (Prolog heritage)
- No static type system!
  - That is, expect runtime errors and exceptions
- Similar to a scripting language (python, perl) – why popular?
Example:

```erlang
fac(N) ->
    if
       N == 0 -> 1;
       true -> N*fac(N-1)
    end.
```

Variables begin with a capital (N)
Atoms (symbols) begin with a lowercase letter (fac, true)
Example:

```erlang
define(N) ->
  if
    N == 0 -> 1;
    true -> N*define(N-1)
  end.
```

Variables begin with a capital (N)
Atoms (symbols) begin with a lowercase letter (define, true)

But this also compiles without warning:

```erlang
define(N) ->
  if
    N == 0 -> 1;
    true -> "upm"*define(N-1)
  end.
```
Example:

fac(N) ->
  if
    N == 0 -> 1;
    true -> N*fac(N-1)
  end.

Variables begin with a capital (N)
Atoms (symbols) begin with a lowercase letter (fac, true)

But this also compiles without warning:

fac(N) ->
  if
    N == 0 -> 1;
    true -> "upm"*fac(N-1)
  end.

And this call is permitted (what happens?): fac(0.5)
Concurrency and Communication

- Concurrency and Communication model inspired by the *Actor model* (and earlier Ericsson software/hardware products)
- Processes execute Erlang functions
- No implicit sharing of data (shared variables) between processes
- Two interprocess communication mechanisms exists:
  - processes can send asynchronous messages to each other (*message passing*)
  - processes get notified when a related process dies (*failure detectors*)
Erlang Processes

- Processes execute Erlang functions \( f(\text{Arg}_1, \ldots, \text{Arg}_n) \)
- A process has a unique name, a **process identifier** \( (\text{Pid}) \)
- Messages sent to a process is stored in a **mailbox** \( (M_2, M_1) \)
Erlang Communication and Concurrency Primitives

■ Sending a message to a process:

    Pid!{request, self(), a}

■ Retrieving messages from the process mailbox (queue):

    receive
    {request, RequestPid, Resource} ->
        lock(Resource), RequestPid!ok
    end

■ Creating a new process:

    spawn(fun () -> locker!{request,B} end)

■ A name server assigns symbolic names to processes:

    locker!{request,a}
Retrieving a message from the process mailbox:

```
receive
  pat_1 when g_1 -> expr_1 ;
  ... ;
  pat_n when g_n -> expr_n
  after time -> expr'
end
```

- `pat_1` is matched against the oldest message, and checked against the guard `g_1`. If a match, it is removed from the mailbox and `expr_1` is executed.

- If there is no match, pattern `pat_2` is tried, and so on...

- If no pattern matches the first message, it is kept in the mailbox and the second oldest message is checked, etc.

- `after` provides a timeout if no message matches any pattern.
Receive Examples

Given a receive statement:

```plaintext
receive
  {inc,X}  ->  X+1;
  Other    ->  error
end

and the queue is a · {inc, 5} what happens?
```
Given a receive statement:

```plaintext
receive
    {inc,X}  ->  X+1;
    Other  ->  error
end
```

and the queue is $a \cdot \{inc, 5\}$ what happens?

Suppose the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?
Receive Examples

- Given a receive statement:
  ```
  receive
  \{inc, X\} \rightarrow X+1;
  \text{Other} \rightarrow \text{error}
  \end
  ```
  and the queue is $a \cdot \{inc, 5\}$ what happens?

- Suppose the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?

- Suppose the receive statement is
  ```
  receive
  \{inc, X\} \rightarrow X+1
  \end
  ```
  and the queue is $a \cdot \{inc, 5\} \cdot b$ what happens?
Given a receive statement:

```plaintext
receive
    {inc,X} -> X+1;
    Other -> error
end
```

and the queue is \( a \cdot \{\text{inc, 5}\} \) what happens?

Suppose the queue is \( a \cdot \{\text{inc, 5}\} \cdot b \) what happens?

Suppose the receive statement is

```plaintext
receive
    {inc,X} -> X+1
end
```

and the queue is \( a \cdot \{\text{inc, 5}\} \cdot b \) what happens?

And if the queue is \( a \cdot b \)?
Communication Guarantees

Messages sent from any process P to any process Q is delivered in order (or P or Q crashes)

![Diagram showing communication guarantees between processes P and Q](image-url)
facserver() ->
    receive
        {request, N, Pid}
    when is_integer(N), N>0, pid(Pid) ->
        spawn(fun () -> Pid!(fac(N)) end),
        facserver()
    end.

A Simple Concurrent Program

```erlang
facserver() ->
  receive
    {request, N, Pid} when is_integer(N), N>0, pid(Pid) ->
      spawn(fun () -> Pid!(fac(N)) end),
      facserver()
  end.

1> spawn(fun () -> facserver() end).
<0.33.0>
```
A Simple Concurrent Program

```erlang
facserver() ->
    receive
        {request, N, Pid}
    when is_integer(N), N>0, pid(Pid) ->
        spawn(fun () -> Pid!(fac(N)) end),
        facserver()
    end.
```

1> `spawn(fun () -> facserver() end).`  
<0.33.0>

2> `X = spawn(fun () -> facserver() end).`  
<0.35.0>
A Simple Concurrent Program

```erlang
facserver() ->
    receive
        {request, N, Pid} when is_integer(N), N>0, pid(Pid) ->
            spawn(fun () -> Pid!(fac(N)) end),
            facserver()
    end.
```

```erlang
1> spawn(fun () -> facserver() end).
<0.33.0>
2> X = spawn(fun () -> facserver() end).
<0.35.0>
3> X!{request,2,self()}.
{request,2,<0.31.0>}
```
A Simple Concurrent Program

```
facserver() ->
    receive
        {request, N, Pid} when is_integer(N), N>0, pid(Pid) ->
            spawn(fun () -> Pid!(fac(N)) end),
            facserver()
    end.
```

1> spawn(fun () -> facserver() end).
<0.33.0>

2> X = spawn(fun () -> facserver() end).
<0.35.0>

3> X!{request,2,self()}.  
{request,2,<0.31.0>}

4> X!{request,4,self()}, receive Y -> Y end.
2
Unavoidably errors happen in distributed systems
Unavoidably errors happen in distributed systems

- hardware (computers) fail
Unavoidably errors happen in distributed systems

- hardware (computers) fail
- network links fail
Unavoidably errors happen in distributed systems

- hardware (computers) fail
- network links fail
- local resources (memory) runs out
Erlang and Errors

- Unavoidably errors happen in distributed systems
  - hardware (computers) fail
  - network links fail
  - local resources (memory) runs out

- Errors happen, good fault-tolerant systems cope with them
Erlang and Errors

- Unavoidably errors happen in distributed systems
  - hardware (computers) fail
  - network links fail
  - local resources (memory) runs out

- Errors happen, good fault-tolerant systems cope with them

- Many Erlang products have high availability goals: 24/7, 99.9999999% of the time for the Ericsson AXD 301 switch (31 ms downtime per year!)
Unavoidably errors happen in distributed systems

- hardware (computers) fail
- network links fail
- local resources (memory) runs out

Errors happen, good fault-tolerant systems cope with them

Many Erlang products have high availability goals: 24/7, 99.9999999% of the time for the Ericsson AXD 301 switch (31 ms downtime per year!)

The Erlang philosophy is to do error detection and recovery, but not everywhere in the code, only in certain places
Erlang and Errors

- Unavoidably errors happen in distributed systems
  - hardware (computers) fail
  - network links fail
  - local resources (memory) runs out

- Errors happen, good fault-tolerant systems cope with them

- Many Erlang products have high availability goals: 24/7, 99.9999999% of the time for the Ericsson AXD 301 switch (31 ms downtime per year!)

- The Erlang philosophy is to do error detection and recovery, but not everywhere in the code, only in certain places

- Higher-level Erlang components offer convenient handling of errors
Error handling example:

\[
g(Y) \rightarrow
\]
\[
  X = f(Y),
\]
\[
  \text{case } X \text{ of}
\]
\[
  \{ \text{ok}, \text{Result} \} \rightarrow \text{Result};
\]
\[
  \text{reallyBadError} \rightarrow 0 \% \text{ May crash because of ...}
\]
\[
  \text{end.}
\]
Error handling example:

```erlang
\[ g(Y) \rightarrow X = f(Y),
    \text{case } X \text{ of}
            \{ok, Result\} \rightarrow Result;
            \text{reallyBadError} \rightarrow 0 \% \text{May crash because of ...}
    \text{end.}
\]

instead one usually writes

```erlang
\[ g(Y) \rightarrow \{ok, Result\} = f(Y), Result.\]
Error handling example:

```erlang
  g(Y) ->
      X = f(Y),
      case X of
        {ok, Result} -> Result;
        reallyBadError -> 0 % May crash because of ...
      end.
```

Instead one usually writes

```erlang
  g(Y) ->
      {ok, Result} = f(Y), Result.
```

The local process will crash; another process is responsible from recovering (restaring the crashed process)
Error handling example:

```erlang

g(Y) ->
    X = f(Y),
    case X of
        {ok, Result} -> Result;
        reallyBadError -> 0  %% May crash because of ...
    end.

``` instead one usually writes

```erlang

g(Y) ->
    {ok, Result} = f(Y), Result.

``` The local process will crash; another process is responsible from recovering (restaring the crashed process)

Error detection and recovery is localised to special processes, to special parts of the code (aspect oriented programming)
Exceptions are generated at runtime due to:

- type mismatches (10 * "upm")
- failed pattern matches, processes crashing, ...

Exceptions caused by an expression $e$ may be recovered inside a process using the construct `try e catch m end`

Example:

```
try
  g(Y)
catch
  Error -> 0
end
```
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the \texttt{link(pid)} function call

- Example:
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the \texttt{link(pid)} function call

- Example:

  Initially we have a system of 3 independent processes:

  \begin{align*}
  &\hspace{1cm} \text{\texttt{P2}} \\
  &\hspace{1cm} \text{\texttt{P1}} \quad \text{\texttt{P3}}
  \end{align*}
Within a set of processes, via bidirectional process links set up using the \texttt{link(pid)} function call

Example:

Result of executing \texttt{link(P1)} in \texttt{P2}:
Within a set of processes, via bidirectional process links set up using the `link(pid)` function call.

Example:

Result of executing `link(P1)` and `link(P3)` in `P2`:
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the `link(pid)` function call

- Example:

  Result of executing `link(P1)` and `link(P3)` in `P2`:

  ![Diagram](link(P1) and link(P3) in P2)

- If `P2` dies abnormally then `P1` and `P3` can choose to die
- If `P1` dies abnormally then `P2` can choose to die as well
Error Detection and Recovery: process level

- Within a set of processes, via bidirectional process links set up using the `link(pid)` function call

- Example:

  Result of executing `link(P1)` and `link(P3)` in `P2`:

  ![Diagram](image)

  - If `P2` dies abnormally then `P1` and `P3` can choose to die.
  - If `P1` dies abnormally then `P2` can choose to die as well.

  - Alternatively when `P2` dies both `P1` and `P3` receives a message concerning the termination.
What is Erlang suitable for?

- Generally intended for long-running programs
- Processes with state, that perform concurrent (and maybe distributed) activities
- Typical is to have a continuously running system (24/7)
- Programs need to be fault-tolerant
- So hardware is typically replicated as well – because hardware invariably fail – and thus we have a need for distributed programming (addressing physically isolated processors)
Distributed Erlang

- Processes run on nodes (computers) in a network

- Distribution is (mostly) transparent
  - No syntactic difference between inter-node or intra-node process communication
  - Communication link failure or node failures are interpreted as process failures (detected using linking)
Distributed Erlang

- Processes run on nodes (computers) in a network

- Distribution is (mostly) transparent
  - No syntactic difference between inter-node or intra-node process communication
  - Communication link failure or node failures are interpreted as process failures (detected using linking)
  - Compare with Java: no references to objects which are difficult to communicate in messages (copy?)
  - The only references are process identifiers which have the same meaning at both sending and receiving process
Erlang Programming Styles

- Using only the basic communication primitives (send/receive) makes for messy code – everybody invents their own style and repeats lots of code for every program.

- A standard way is needed to:
  - a standard way to handle process start, termination and restarts
  - to handle code upgrading
  - and maybe more structured communication patterns: who communicates with whom, in what role?...

- For Erlang one generally uses the design patterns and the framework of the **OTP library – Open Telecom Platform** – as an infrastructure.
OTP components

- **Application**
  - provides bigger building blocks like a database (Mnesia), a web server, and interfaces to other languages and formats (Java, XML)

- **Supervisor**
  - used to start and bring down a set of processes, and to manage processes when errors occur

- **Generic Server**
  - provides a client–server communication facility

- **Event Handling**
  - for reporting system events to interested processes

- **Finite State Machine**
  - provides a component facilitating the programming of finite state machines in Erlang
Applications are often structured as *supervision trees*, consisting of *supervisors* and *workers*.

A supervisor starts child processes, monitors them, handles termination and stops them on request.

The actions of the supervisor are described in a declarative fashion (as a text description).

A child process may itself be a supervisor.
Supervision Dynamics

When a child process C1 dies (due to an error condition), its supervisor S3 is notified and can elect to:

- do nothing
- itself die (in turn notifying its supervisor S)
- restart the child process (and maybe its siblings)
- kill all the sibling processes (C2,C3) of the dead process
When a child process C1 dies (due to an error condition), its supervisor S3 is notified and can elect to:

- do nothing
- itself die (in turn notifying its supervisor S)
- restart the child process (and maybe its siblings)
- kill all the sibling processes (C2, C3) of the dead process

One can control the frequency of restarts, and the maximum number of restarts to attempt – it is no good having a process continuing to restart and crash
The Generic Server Component

- **gen_server** is *the* most used component in Erlang systems
- Provides a standard way to implement a server process, and interface code for clients to access the server
- The client–server model has a central server, and an arbitrary number of clients:
Clients make *requests* to the server, who optionally *replies*.

A server has a state, which is preserved between requests.

A generic server is implemented by providing a callback module specifying the concrete actions of the server (server state handling, and response to messages).
Part 2: Verifying Erlang Programs
Debugging/Verifying Erlang Programs: Tools

- **Dialyzer** – type checking by static analysis (necessary to minimize type errors at runtime)

- **Testing:** [QuickCheck](http://www.quiviq.com) - a testing tool for Erlang

- **Model checking** – our tool **McErlang**
  ([https://babel.ls.fi.upm.es/trac/McErlang/](https://babel.ls.fi.upm.es/trac/McErlang/))
Why is (random) testing of concurrent programs difficult?
Consider the state space of a small program:
Random testing explores **one** path through the program:
Testing Concurrent Programs

With repeated tests the coverage improves:
Testing Concurrent Programs

A lot of testing later (note the states not visited):
Model checking can guarantee that all states are visited, without revisiting states
- **Construct** an abstract **model** of the behaviour of the program, usually a finite state transition graph

- A node represents a **Program state** \((x = 0, y = 3)\)
- **Graph edges** represent computation steps from one program state to another
**Model Checking: Basics**

- **Construct** an abstract **model** of the behaviour of the program, usually a finite state transition graph

  ![Diagram](image.png)

  - A node represents a **Program state** \((x = 0, y = 3)\)
  - **Graph edges** represent computation steps from one program state to another

- **Check** the abstract model against some description of desirable/undesirable model properties usually specified in a **temporal logic**:  
  \(\text{Always } x \geq 0\)
Usually applied to **reactive systems**
(systems that continuously react to stimuli)
Model Checking

- Usually applied to **reactive systems**
  (systems that continuously react to stimuli)

- Advantages: automatic push button technology
  (algorithms can decide, with decent complexity, whether a model satisfies a property)
Model Checking

- Usually applied to **reactive systems**
  (systems that continuously react to stimuli)

- Advantages: automatic push button technology
  (algorithms can decide, with decent complexity, whether a model satisfies a property)

- Disadvantages:
  - Models can be difficult and time consuming to construct
  - Doesn’t scale well to larger programs (the model of the behaviour of the program becomes too big – the well known **state explosion problem**
The McErlang model checker: Design Goals

- Reduce the gap between program and verifiable model (the Erlang program is the model)
- Write correctness properties in Erlang (and linear temporal logic)
- Implement verification methods that permit partial checking when state spaces are too big – on-the-fly checking and using Holzmann’s bitspace algorithms
- Implement the model checker in a parametric fashion (easy to plug-in new algorithms, new abstractions, …)
To be able to visit all the states of an Erlang program we need the capability to take a snapshot of the Erlang system.

A snapshot/program state is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, ...
To be able to visit all the states of an Erlang program we need the capability to take a snapshot of the Erlang system.

- A snapshot/program state is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, ...

Save the snapshot to memory and forget about it for a while.

Later continue the execution from the snapshot.
The McErlang approach to model checking

- The lazy solution: just execute the Erlang program to verify in the normal Erlang interpreter
- And extract the system state (processes, queues, function contexts) from the Erlang runtime system
The McErlang approach to model checking

- The lazy solution: just execute the Erlang program to verify in the normal Erlang interpreter

- And extract the system state (processes, queues, function contexts) from the Erlang runtime system

- Too messy! We have developed a new runtime system for the process part, and still use the old runtime system to execute code with no side effects
Correctness Properties: Temporal logic

■ Pnueli 1977: added discrete and linear time operators to propositional logic, to be able to specify properties of reactive systems

■ Program meaning (semantics):

◆ a program state $s$ maps the program variables to values
◆ a run of the program is an infinite sequence of program states $(s_0, s_1, s_2, \ldots)$ from an initial state $s_0$
◆ for a terminating system simply add a self-loop in the terminating state to yield an infinite run
◆ the semantics of a program $p$ is its set of runs, $\|p\|$ 
◆ If the program is nondeterministic (or accepts input) there will be more than one run of the program
Consider the following simple shared variable program:

\[
\begin{align*}
\text{if } x > 0 \text{ then } x := x - 1 & \quad \text{|| \quad if } x < 3 \text{ then } x := x + 1 \\
\text{where } S1 \quad \text{||} \quad S2 \text{ runs the atomic statements } S1 \text{ and } S2 \text{ in parallel}
\end{align*}
\]
Consider the following simple shared variable program:

\[
\text{if } x > 0 \text{ then } x := x - 1 \quad || \quad \text{if } x < 3 \text{ then } x := x + 1
\]

where \( S_1 || S_2 \) runs the atomic statements \( S_1 \) and \( S_2 \) in parallel.

Its runs starting from the state \( x = 0 \) is the infinite set:

\[
\begin{cases}
\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \cdot \ldots \\
\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \ldots \\
\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \ldots \\
\ldots
\end{cases}
\]
Program runs

We can also depict the runs

\[
\begin{aligned}
\{ &\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 0 \rangle \cdot \ldots \\
&\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 1 \rangle \cdot \ldots \\
&\langle x = 0 \rangle \cdot \langle x = 1 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \ldots \\
&\ldots \\
\} 
\end{aligned}
\]

as a state graph:

```
x=0 -----> x=1 -----> x=2 -----> x=3
```

\[x=0\]

\[x=1\]

\[x=2\]

\[x=3\]
Temporal logic operators

Classical linear temporal operators (defined over runs):

- **Always** \( \phi \)
  \( \phi \) holds in all future states of the run

- **Eventually** \( \phi \)
  \( \phi \) holds in some future state of the run

- **Next** \( \phi \)
  \( \phi \) holds in the next state

- **\( \phi_1 \ Until \ phi_2 \)**
  \( \phi_1 \) holds in all states until \( \phi_2 \) holds

- And the normal ones: negation \( \neg \phi \), implication \( \phi_1 \supset \phi_2 \), …
Temporal logic state propositions

These provide basic statements about program states

- For Pnueli’s shared variable language: $x > 0, x < y, \text{even}(z)$, ...

- For Erlang: $\text{Pid}!\{\text{request, a}\}$
  (a request message is sent to some process)
A program $p$ satisfies a formula $\phi$ when all the runs of the program are satisfied by the formula.

The logic is linear because it doesn’t talk about the branching structure of the state graph of the program (what is set of possible next states of the program).

So called branching time logics (CTL, $\mu$-calculus) do consider the branching structure of the state graph of the program.
Consider the atomic parallel program

$$\text{if } x > 0 \text{ then } x := x - 1 \mid \mid \text{if } x < 3 \text{ then } x := x + 1$$

with the starting state $\langle x = 3 \rangle$ and the state graph
Consider the atomic parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \parallel \text{ if } x < 3 \text{ then } x := x + 1
\]

with the starting state \( \langle x = 3 \rangle \) and the state graph

■ Does \( \text{Always } x \geq 0 \) hold?
Consider the atomic parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \parallel \text{ if } x < 3 \text{ then } x := x + 1
\]

with the starting state \( \langle x = 3 \rangle \) and the state graph

- Does \textit{Always} \( x \geq 0 \) hold?
- \textbf{Yes}; if \( x = 0 \) then the guard prevents further decrease
Consider the atomic parallel program

\[
\text{if } x > 0 \text{ then } x := x - 1 \parallel \text{ if } x < 3 \text{ then } x := x + 1
\]

with the starting state \( \langle x = 3 \rangle \) and the state graph

- **Does** *Always* \( x \geq 0 \) *hold?*
  - **Yes**; if \( x = 0 \) then the guard prevents further decrease

- **Does** *Always* \( (x = 3 \supset \text{Eventually } x = 0) \) *hold?*
Temporal logic – examples

Consider the atomic parallel program

\[ \text{if } x > 0 \text{ then } x := x - 1 \quad || \quad \text{if } x < 3 \text{ then } x := x + 1 \]

with the starting state \( \langle x = 3 \rangle \) and the state graph

- Does \( \text{Always } x \geq 0 \) hold?
  - Yes; if \( x = 0 \) then the guard prevents further decrease

- Does \( \text{Always } (x = 3 \supset \text{Eventually } x = 0) \) hold?
  - No; there is a run \( \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdot \langle x = 3 \rangle \cdot \langle x = 2 \rangle \cdots \)
General temporal logic patterns
A safety property expresses that something bad $\phi$ never happens:

$$Always \neg \phi$$
General temporal logic patterns

- A **safety property** expresses that something bad – $\phi$ – never happens:
  \[
  \text{Always } \neg \phi
  \]

- A **liveness property** expresses that something good – $\phi$ – eventually happens:
  \[
  \text{Eventually } \phi
  \]
General temporal logic patterns

- **A safety property** expresses that something bad – $\phi$ – never happens:
  
  $$Always \neg \phi$$

- **A liveness property** expresses that something good – $\phi$ – eventually happens:
  
  $$Eventually \phi$$

- **Fairness assumptions** are used to rule out abnormal program behaviours; $\phi$ eventually holds under the assumption that $\psi$ doesn’t always hold:
  
  $$(\neg Always \psi) \implies (Eventually \phi)$$
How to check LTL properties on programs?

- LTL formulas are translated into Büchi automata

\[ \text{Always}(\text{req} \supset \text{Next}(\neg \text{abort} \text{ Until release})) \]

- A combined program and automaton state graph is generated by executing the program in lock-step with the automaton

- When a new program state is generated, the automaton computes a new automaton state (by inspecting the program state)
LTL checking: correctness condition

For checking safety properties – *Always* ¬ \( \phi \) – every program state should be inspected once
LTL checking: correctness condition

For checking safety properties – *Always* $\neg \phi$ – every program state should be inspected once.
LTL checking: correctness condition

For checking safety properties – *Always* $\neg \phi$ – every program state should be inspected once

To prove a program *incorrect*, it may not be necessary to explore the whole state space of the program.
LTL checking: intuition

For a liveness property – Eventually $\phi$ – to hold, there can be no loop in the combined state graph where something bad happens.
For a liveness property – \( \text{Eventually } \phi \) – to hold, there can be no loop in the combined state graph where something bad happens.
LTL checking: intuition

For a liveness property – *Eventually* $\phi$ – to hold, there can be no loop in the combined state graph where something bad happens.

To prove a program *incorrect*, it may not be necessary to explore the whole state space of the program.
Install Erlang first

Then download McErlang from
https://babel.ls.fi.upm.es/trac/McErlang/

Runs on Linux, Windows, ...
Two processes are spawned, the first starts an “echo” server that echoes received messages, and the second invokes the echo server:

```erlang
-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.
```
Example under normal Erlang

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

```erlang
> 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.
First have to recompile the module using the McErlang compiler:

```
> mcerl_compiler -sources example.erl -output_dir .
```
Example under McErlang

First have to recompile the module using the McErlang compiler:

> mcerl_compiler -sources example.erl -output_dir .

Then we run it:

> erl
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

Stack trace:
  mcerlang:resolvePid/2
  mcerlang:send/2
  ...
Investigating the Error

An error! Let’s find out more using the McErlang debugger:

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

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

1: process <node0,3>:
   run #Fun<example.2.125>([])
   process <node0,3> died due to reason badarg

0: process <node0,1>:
   run function example:start([])
   spawn({#Fun<example.1.278>,[],[]} --> <node0,2>
   spawn({#Fun<example.2.125>,[],[]} --> <node0,3>
   process <node0,1> was terminated
   process <node0,1> died due to reason normal
Error Cause

- Apparently in one program run the second process spawned (the one calling the echo server) was run before the echo server itself.

- Then upon trying to send a message

\[\text{echo!\{msg, } \text{self}()\text{, }'\text{hello}_\text{world}' \}\]

the echo name was obviously not registered, so the program crashed.
We study the control software for a set of elevators

Used to be part of an Erlang/OTP training course from Ericsson
Elevator Control Software

- Static code complexity: around 1670 lines of code (uses several OTP behaviours: supervisor, gen_fsm, ...)
Elevator Control Software

- Static code complexity: around 1670 lines of code (uses several OTP behaviours: supervisor, gen_fsm, ...)
- Dynamic complexity: around 10 processes (for two elevators)

We had to modify around 10 lines to model check this example
Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?
Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*
Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- No runtime exceptions
- An elevator only stops at a floor after receiving an order to go to that floor
Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*

- *An elevator only stops at a floor after receiving an order to go to that floor*

- ...
How to formulate a property like: “an elevator only stops at a floor after receiving an order to go to that floor”? 
How to formulate a property like: “an elevator only stops at a floor after receiving an order to go to that floor”?

We can borrow an idea from runtime monitoring: we write a monitor/safety automaton that detects when the above property is violated.
Formulating Correctness Properties

- How to formulate a property like: “an elevator only stops at a floor after receiving an order to go to that floor”?

- We can borrow an idea from runtime monitoring: we write a monitor/safety automaton that detects when the above property is violated

- Seen from another viewpoint we have created a model for the elevator system

- The model only describes a small subset of the behaviour of the elevator – fine, it is what models are supposed to do

- So we have to write more monitors and properties…”
What does a safety automaton do?

- It runs in parallel (lock-step) with the program
What does a safety automaton do?

- It runs in parallel (lock-step) with the program.
- Has an internal state, which can be updated when the program does a *significant* action (or something happens – a *button press*).
What does a safety automaton do?

- It runs in parallel (lock-step) with the program.
- Has an internal state, which can be updated when the program does a significant action (or something happens – a button press).
- The monitor should signal an error if an action happens in an incorrect state.
Which elevator events do the monitor need to react to?
Which elevator events do the monitor need to react to?

- Button presses in the elevator
Which elevator events do the monitor need to react to?

- Button presses in the elevator
- Button presses at each floor
Which elevator events do the monitor need to react to?

- Button presses in the elevator
- Button presses at each floor
- The arrival of the elevator at a floor
State and Correctness Check

- What is the state of the monitor?
What is the state of the monitor?

A data structure that remembers orders to go to a certain floor
State and Correctness Check

- What is the state of the monitor?

  A data structure that remembers orders to go to a certain floor

- What is the correctness check?
State and Correctness Check

- **What is the state of the monitor?**

  A data structure that remembers orders to go to a certain floor

- **What is the correctness check?**

  When the elevator arrives at a floor, the order to do so is in the monitor state
Safety Automata

- Safety automata is a subclass of automata which users can program directly in Erlang.

- Concretely, to implement a safety automaton a McErlang user should provide a function

  \[
  \text{stateChange} \left( \text{ProgramState}, \text{AutomatonState}, \text{Action} \right) \rightarrow \ldots \{\text{ok, NewAutomatonState}\}. \]

  which is automatically called by McErlang when a program changes its state.

- The automaton can inspect the current program state, its own state, and the side effects (actions) in the last computation step.

- The automaton either returns a new automaton state (success), or signals an error.
What can automata observe?

- **Program actions** such as e.g. sending or receiving a message
- **Program state** such as e.g. contents of process mailboxes, name of registered processes
- Indirectly the values of some program variables (can be somewhat difficult to access)
- Programs can be instrumented with special “probe actions”, that are easy to detect in monitors
- Programs can be instrumented too with special “probe states”, which are persistent (actions are transient)
Correctness property spec:

```prolog
stateChange(_, FloorReqs, Action) ->
    case Action of
    {f_button, Floor} ->
        ordsets:add_element(Floor, FloorReqs);
    {e_button, Elevator, Floor} ->
        ordsets:add_element(Floor, FloorReqs);
    {stopped_at, Elevator, Floor} ->
        case ordsets:is_element(Floor, FloorReqs) of
            true -> FloorReqs;
            false -> throw({bad_stop, Elevator, Floor})
        end;
    _ -> FloorReqs
    end.
```

Uses ordered sets (`ordsets`) to store the set of floor orders (the state of the monitor)
Scenarios

- Ok, so we have a program, and a correctness property, what is missing?
Scenarios

- Ok, so we have a program, and a correctness property, what is missing?
- Hmm... we have to specify the environment under which we check the program, i.e., the sequences of buttons the elevator users press
Scenarios

- Ok, so we have a program, and a correctness property, what is missing?

- Hmm... we have to specify the environment under which we check the program, i.e., the sequences of buttons the elevator users press

- Instead of specifying one big scenario with a really big state space, we generate a number of smaller scenarios, similar to test cases:
  - Floor button 1 pressed
  - Floor button 2 pressed, Elevator button 1 pressed
  - Elevator button 2 pressed, Floor button 2 pressed, Floor button 2 pressed, ...
Scenarios

- Ok, so we have a program, and a correctness property, what is missing?

- Hmm... we have to specify the environment under which we check the program, i.e., the sequences of buttons the elevator users press.

- Instead of specifying one big scenario with a really big state space, we generate a number of smaller scenarios, similar to test cases:
  - Floor button 1 pressed
  - Floor button 2 pressed, Elevator button 1 pressed
  - Elevator button 2 pressed, Floor button 2 pressed, Floor button 2 pressed, ...

- But since we are model checking every scenario is fully explored.
More Correctness Properties

- Refining the floor correctness property:

  An elevator only stops at a floor after receiving an order to go to that floor, if no elevator has already met the request

  (implemented as a monitor that keeps a set of floor requests; visited floors are removed from the set)
The floor correctness property is a safety property

(nothing bad ever happens)
Other Correctness Properties

- The floor correctness property is a safety property
  \((\text{nothing bad ever happens})\)

- A Liveness property:
  \(\text{If there is a request to go to some floor, eventually some elevator will stop there}\)
Other Correctness Properties

- The floor correctness property is a safety property
  *(nothing bad ever happens)*

- A Liveness property:
  *If there is a request to go to some floor, eventually some elevator will stop there*

- In temporal logic:
  
  ```
  always
  (fun go_to_floor/3) =>
  next (eventually (fun stopped_at_floor/3))
  ```

- The state predicate `fun go_to_floor/3` is satisfied when an elevator has received an order to go to a floor

- The state predicate `fun stopped_at_floor/3` is satisfied when an elevator stops at a floor
A Pragmatic Testing-Like Approach to Model Checking

- We strive to reduce the effort in creating a model from a program (we support almost full Erlang).

- When programs are too complex to fully verify, model checking becomes a form of controlled testing:
  - The amount of memory and time available to verify a program can be controlled (a verification attempt can be inconclusive).
  - Randomized (wrt. state space exploration order) verification algorithms are available (thus repeating a verification run can explore new parts of the state space).
  - Randomized state storage data structures are available (Holzmann’s bitspace algorithms).

- Instead of building complex program environments a program is checked under a set of much simpler program environments.
Self Study

- **Install McErlang**
  
  [https://babel.ls.fi.upm.es/trac/McErlang/](https://babel.ls.fi.upm.es/trac/McErlang/)

- **The file**
  
  [https://babel.ls.fi.upm.es/trac/McErlang/attachment/wiki/midTermWorkshop/exercises.txt](https://babel.ls.fi.upm.es/trac/McErlang/attachment/wiki/midTermWorkshop/exercises.txt)

  contains instructions

- **See the directory examples in the McErlang distribution for the lift example source code**