

[Electronic Notes in Theoretical Computer Science 271 (2011) 23–40](http://dx.doi.org/10.1016/j.entcs.2011.02.009)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

A Case Study on Verifying a Supervisor Component Using McErlang [6](#_bookmark0)

David Castro[1](#_bookmark0) Victor M. Gulias[2](#_bookmark0)

*MADS Group, Computer Science Department University of A Corun˜a, Spain*

Clara Benac Earle[3](#_bookmark0) Lars-˚Ake Fredlund[4](#_bookmark0)

*Babel Group, School of Computer Science, Universidad Polit´ecnica de Madrid (UPM), Spain*

Samuel Rivas[5](#_bookmark0)

*LambdaStream Servicios Interactivos S.L. Ronda de Outeiro 33 Entlo., A Corun˜a, Spain*

**Abstract**

In this paper we present a work in progress on the formal verification of a process supervisor using the McErlang model checker. The process supervisor is an alternative implementation of the standard supervisor behaviour of Erlang/OTP. This implementation, currently employed at the company LambdaStream, was checked against several safety and liveness properties.

*Keywords:* McErlang, Model Checking, Verification, Supervisor, Erlang

1 Email: [dcastrop@udc.es](mailto:dcastrop@udc.es)

2 Email: [gulias@udc.es](mailto:gulias@udc.es)

3 Email: [cbenac@fi.upm.esc](mailto:cbenac@fi.upm.esc)

4 Email: [lfredlund@fi.upm.es](mailto:lfredlund@fi.upm.es)

5 Email: [samuel.rivas@lambdastream.com](mailto:samuel.rivas@lambdastream.com)

6 This work has been partially supported by the following projects: ProTest (FP7-ICT- 2007-1 215868), DESAFIOS10 (TIN2009-14599-C03-00), PROMETIDOS (P2009/TIC-

1465), AMBITIIONS (MICIN TIN 2010-20959), and XUGA (PGIDIT07TIC005105PR).

1571-0661 © 2011 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

[doi:10.1016/j.entcs.2011.02.009](http://dx.doi.org/10.1016/j.entcs.2011.02.009)

# Introduction

Writing correct concurrent software is a difficult task given the inherit non- deterministic behaviour of such systems. Concurrent declarative languages seem to be an interesting and practical choice for implementing concurrent software due to the absence of side-effects (or at least a reduced number of them). For example, the concurrent functional language Erlang [[5](#_bookmark17)] is being used by several companies worldwide to implement complex concurrent control systems.

A technique which is often used to check that a concurrent program fulfills a set of desirable properties is *model checking* [[9](#_bookmark20)]. In model checking, in theory, all the states of (a model of) a concurrent system are systematically explored to check for safety requirements such as absence of deadlocks and similar critical situations that can cause the system to misbehave. For instance, McErlang [[7](#_bookmark19)], a model checker for programs written in Erlang, directly uses the actual Erlang source code as the model to be analyzed. McErlang has been successfully applied in a number of case studies, such as a Video–on–demand server [[6](#_bookmark18)], leader election protocols [[7](#_bookmark19)], the control software for elevators, and multi-agent systems playing for robotic soccer [[4](#_bookmark13)].

An example of a company using Erlang is LambdaStream S.L., which is interested in improving their software quality as shown by their participation in a European research project on property-based testing (ProTest [7](#_bookmark1) ). Thanks to this project, a fruitful collaboration has been established between develop- ers at LambdaStream and researchers at the University of A Corun˜a and the Universidad Polit´ecnica de Madrid. One example of this collaboration is the experience with verification using McErlang of properties of several LambdaS- tream components, since McErlang model checker is well suited to verify this class of concurrent software components.

In this paper we describe the experience with checking several properties on LambdaStream’s supervisor component using McErlang. A process super- visor is responsible for starting, stopping, and monitoring its child processes. Testing and reproducing errors in such code is intrinsically difficult due to its non-deterministic behaviour, asynchronous communication, timing, etc. At the same time, it is crucial to establish the reliability of that critical software component because it is integrated into several software systems at LambdaS- tream. With the technique described in this paper, it was impossible to fully verify the component. This limitation comes from the *state explosion* prob- lem, which affects all model checking techniques. Although the supervisor has been in use in several products for some time and it was well tested by both

7 <http://www.protest-project.eu/>

the developers and a specific testing team, and it was partially verified, a dis- crepancy between the component specification and the actual implementation of the component was identified.

The remainder of this paper is organized as follows. A brief introduction to Erlang and McErlang is provided in Section [2](#_bookmark2). The description of the process supervisor component is given in Section [3](#_bookmark5) together with a brief description of the properties we have considered. The implementation of such properties and the experimental results of checking them with McErlang is explained in Section [4](#_bookmark7). Finally, some conclusions and lines of future work are discussed in Section [5](#_bookmark12).

# Background

* 1. *The Erlang Programming Language*

Erlang [[1](#_bookmark14), [5](#_bookmark17)] is a concurrent programming language and run-time system. The sequential subset of the language is a dynamically typed functional program- ming language with strict evaluation. Concurrency is achieved by lightweight processes communicating through asynchronous message passing. Erlang has no construct inducing side-effects with the exception of communications among processes [8](#_bookmark3) . Expressions are evaluated eagerly similarly to, for instance, Stan- dard ML.

What makes Erlang different from other functional languages is its support for concurrency and distribution. With Erlang’s primitives for concurrency, it resembles formal calculi such as Milner’s CCS [[10](#_bookmark22)] or Hoare’s CSP [[8](#_bookmark21)]. Because of the absence of side-effects, limited to explicit inter-process com- munications, concurrent programming is far simpler compared to standard imperative languages. An Erlang virtual machine (node, in Erlang terminol- ogy) hosts several Erlang processes running concurrently. Usually, an Erlang node is mapped to an operating system process; an Erlang process is, in fact, a lightweight user-level thread with very little creation and context-switching overheads.

A distributed Erlang application consists of processes running in several nodes, possibly at different (physical or virtual) computers. Even though the initialization and management of each node is platform dependent, the nice feature is that communication among remote processes is equivalent in a distributed framework, though remote communications are less efficient, of course.

8 Strictly speaking this is not true as Erlang has a process registry with side effects, and various libraries offering mutable data structures. The use of such features in Erlang, however, tends to be greatly reduced compared to other programming languages.

Fault-tolerance in Erlang is achieved by linking processes together in order to detect and possibly recover from abnormal process termination. Abnormal termination occurs if, for example, a function tries to access the tail of an empty list. Processes linked to the process that terminated abnormally are notified of the termination, and can thus take corrective action, i.e., possibly restarting the failed process. Process links are always bidirectional but the treatment of process termination notifications may differ between the two parties.

Handling a large number of processes easily turns into an unmanageable task, and therefore Erlang programmers mostly work with higher-level lan- guage components provided by libraries. The OTP (Open Telecom Platform) component library [[12](#_bookmark24)] is by far the most widely used library, offering several behaviours (design patterns in Erlang) that can be instantiated with con- crete callback functions. For example, OTP behaviours include, among oth- ers, generic servers (for client-server communication), finite state machines, or a supervisor component for structuring fault-tolerant systems hierarchi- cally, where a parent supervisor is responsible for supervising and managing its children (work processes).

* 1. *The McErlang model checker*

McErlang [[7](#_bookmark19)] is a tool for model checking Erlang programs. The input to McErlang is the Erlang program we want to verify together with the property of interest. The fact that the program is the model facilitates the use in real-world development.

As the model checking tool is itself implemented in Erlang we benefit from the advantages that a dynamically typed functional programming language of- fers: easy prototyping and experimentation with new verification algorithms, rich executable models that use complex data structures directly programmed in Erlang, the ability to deal with executable models interchangeably as pro- grams (to be executed directly by the Erlang interpreter) and data, etc.

In order to use Erlang programs as models, McErlang undergoes a source- to-source transformation to prepare the program for running under the model checker. Then, the actual Erlang compiler translates the program to Erlang byte code. Finally the program runs under the McErlang run-time system, under the control of a verification algorithm, using the regular Erlang byte- code interpreter. The pure computation part of the code (i.e, code with no side effects) and memory management (including garbage collection), is carried out 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. Naturally, the new run-time system offers easy check pointing (capturing the state of all nodes and processes, of the message mailboxes of all processes, and all messages in transit between processes) of the whole program state as a feature.

The steps required to perform model checking with McErlang are the fol- lowing:

1. **Create the model.** Use the Erlang program directly. McErlang pro- vides support for virtually the full language, full data type support, sup- port for general process communication, node semantics (inter-process communication behaves in a subtly different way from intra-process com- munication), fault detection and fault tolerance, and crucially can verify programs written using the high-level OTP Erlang component library used by most Erlang programs.
2. **Formulate correctness properties.** Write down the properties to ver- ify as either a *safety monitor* or a formula in linear temporal logic (LTL). A safety monitor is used to verify safety properties (*something bad never happens*), by keeping an internal state to check the properties at each state the program to verify reaches. Given a program and a monitor, McErlang runs them in lockstep letting the monitor analyze each new program state generated. If the property does not hold, a counterexample (an execution trace) is generated. Monitors are capable of observing both the shape of the system (e.g., which processes are alive) and significant

system events (e.g., messages sent between processes).

Some properties cannot be expressed with safety monitors, for example, liveness properties (*something good eventually happens*). In McErlang one can express such properties in Lineal Temporal Logic (LTL), and use the LTL2Buchi tool [[11](#_bookmark23)] to automatically translate an LTL formula into a Bu¨chi automaton [[3](#_bookmark15)].

1. **Generate scenarios.** Model checking is typically a memory intensive operation, and frequently completely verifying a large system may require more computer memory than is available. As an alternative to verifying the whole system at once we specify a sizeable number of smaller sys- tem *scenarios* instead, specifying both system configuration and process communications, such that each scenario is small enough to admit com- plete verification. As an example, if the system to verify is a client/server architecture, we verify a sizeable number of client–server configurations varying the number of clients, and varying the requests that clients issue to the servers.

# Case Study: Supervisor Process

* 1. *The Notion of Supervision*

A supervisor is a process in charge of starting, stopping and monitoring a set of children (processes). Basically whenever a child process terminates the supervisor should restart it, i.e., spawn a new process executing the task of the terminated child. A supervisor typically supervises not only process workers, but also other supervisors, defining a hierarchical structure as shown in the example of Fig. [1](#_bookmark6).

Fig. 1. A supervision tree.

Worker1

Worker3

Worker2

Worker4

Worker5

Supervisor3

Supervisor2

Supervisor1

There are several strategies of how a supervisor process should handle the event of an abnormal termination of a child. The two main policies are:

* one for one: If a child process terminates, only that process is restarted.
* one for all: If a child process terminates, all other child processes are ter- minated and then all children, including the originally terminated process, are restarted.

There is usually a mechanism to prevent a situation where a process re- peatedly terminates for the same reason, only to be immediately restarted again. This mechanism involves a maximum *restart intensity*. When a child reaches this restart intensity, the supervisor terminates all the child processes and then itself.

As the supervision tree is hierarchical, when a lower-level supervisor termi- nates, then the next higher level supervisor can take corrective action. That is, either restarting the terminated supervisor (and its subsystem of processes), or terminating itself if the error can not be handled on that level of the su- pervision tree.

In addition to the restart policy, the creation and termination order of children must be also specified in the supervisor.

* 1. *Supervisor Implementation*

There is a standard implementation of the supervisor behaviour provided by the open source distribution of Erlang/OTP. In this case study, we used an implementation of a supervisor process slightly different from the OTP imple- mentation. This variant (from now on *nos supervisor* ) was implemented by the company LambdaStream because they wanted a more configurable super- visor to easily integrate into its products. In particular, different restarting policies are allowed for each child node in nos supervisor.

Following the specification for a child, a child process is spawned by a supervisor process. In the nos supervisor a child specification is a tuple:

*{*Id, StartFun, RestartFun, RestartStrategy, RestartIntensity, Shutdown, Options*}*

* Id is a name that is used to identify the child specification internally by the supervisor.
* StartFun defines the function call used to start the child.
* RestartFun defines the function call to restart the child.
* RestartStrategy. When a child process crashes/dies, if its restart strategy is child, only this child will be restarted. If its restart strategy is all, all the children will be stopped in reverse start order, and they will be restarted (including the offending one) in start order.
* RestartIntensity: either a tuple *{*MaxR,MaxT,Finally*}* or infinity:
* *{*MaxR,MaxT,Finally*}*. If a child is restarted MaxR times in MaxT or less seconds, the Finally action is triggered:

Finally==kill sup. The supervisor shuts down its living children in reverse start order, and then it terminates.

Finally==stop child. The offending child is not restarted. Remaining children continue restarting normally.

* infinity. If a child terminates, the supervisor will always try to restart

it.

* Shutdown. The shutdown strategy is the same as in the OTP supervisor.
* brutal kill. Supervisor kills processes, i.e., a child process P will be unconditionally terminated by the supervisor sending an exit(P,kill) message.
* infinity. The supervisor will inform the child process that it should

terminate and then wait to receive an exit signal from the child signaling

that it indeed has terminated.

* Timeout (integer). As in infinity but if no exit signal is received from the child within the specified time, the child process is unconditionally terminated.
* Options is a list of options where an option can be either insistent restart meaning that restart function failures are treated as process crashes instead of restart errors, or *{*notify,pid()*}* which makes the supervisor send no- tification messages to the specified process identifier.
  1. *Properties of the nos supervisor*

By reading the nos supervisor documentation we came up with many inter- esting properties to verify for its implementation. As a trivial example, we wanted to check that a child that has terminated is actually restarted by its supervisor. We also focused on properties regarding the different restart inten- sities and the shutdown strategies, checking all combinations of those options. For example, we formulated and checked a property that states that if there is any child specification with the Finally action of the restart intensity set to kill sup and this child reaches the maximum restart intensity, the supervisor applies the Shutdown strategy to its children in reverse start order and then it finishes. In the following section we describe how some of these properties were verified using McErlang.

# Verification using McErlang

To verify the nos supervisor described in the previous section using McErlang we follow the approach described in Sec. [2.2](#_bookmark4), i.e., we create a verifiable model, we formulate a number of correctness properties, and then we generate a set of scenarios.

* 1. *Create the Model*

Obtaining the model for this case-study is straightforward since McErlang can use the actual source code as model. In this section, we explain the only change to the source code of the nos supervisor that was needed to obtain a verifiable model. To measure the restart intensity, the nos supervisor needs to determine the number of restarts that took place within a certain time interval. As McErlang currently implements neither real-time nor discrete-time model checking algorithms we were forced to abstract away from the time interval.

add\_restart (# child\_spec {

restart\_intensity = infinity

}) -> [];

add\_restart (# child\_spec {

restart\_intensity = { MaxR , MaxT , Finally }, state = ChildState }) ->

Restarts = Child State # child\_state . restarts , Now = now () ,

check\_restarts ( MaxR

, Finally

, filter\_restarts ( MaxT

,[ Now | Restarts ] )).

As we can see above, the nos supervisor stores the restart time of each process in a list, which is then filtered using the difference between the time of the current restart and the maximum time specified by the restart intensity. If the resulting list is greater than the allowed number of restarts, then the Finally action is triggered. If not, the filtered list with the new restart time is returned:

filter\_restarts ( MaxT , [ H | Restarts ]) -> F = fun ( Restart ) ->

difference ( Restart , H) < MaxT end ,

[ H | lists : takewhile ( F, Restarts )].

check\_restarts ( MaxR , Finally , Restarts ) -> case length ( Restarts ) > MaxR of

true -> Finally ; false -> Restarts

end .

The solution was not to store concrete times when restarting, but merely recording that a restart occurred (using the symbol now), until the length of the list containing restart indications is equal to the maximum number of restarts. The last two lines of add restart are thus replaced by the following code fragment:

New Restarts = case length ( Restarts ) >= MaxR of true -> Restarts ;

false -> [ now | Restarts ] end ,

check\_restarts ( MaxR , Finally , NewRestarts ).

It no longer makes sense to filter the list, but instead the model checker must consider two possibilities: whether these restarts happened within the maximum time or not. McErlang offers the possibility to express such a non-deterministic choice using a function call to mce erl:choice giving the branches as a list of anonymous functions as shown below, and during model checking both alternatives will be explored:

check\_restarts ( MaxR , Finally , Restarts ) -> case length ( Restarts ) >= MaxR of

true ->

mce\_erl : choice ([ fun () -> Finally end

, fun () -> Restarts end

]);

false ->

Restarts

end .

* 1. *Formulating Correctness Properties*

As explained in section [2.2](#_bookmark4), in McErlang correctness properties can be specified either using a safety monitor, which run in parallel with the system to verify and observe its actions, or as a formula of linear temporal logic. In practice, most of the relevant nos supervisor correctness properties can be expressed as safety monitors.

In McErlang a safety monitor is implemented as an Erlang behaviour; a module implementing a safety monitor must implement the following callback functions:

* an init function to initialize the monitor (which should return the initial state of the monitor).
* a stateChange function: when the model checker computes a new system state, this function is called with the following arguments: (i) the new system state, (ii) the current state of the monitor, and (iii) the sequence of system actions that occurred during the computation of the new system state. The stateChange function should determine whether the new system state, and the system actions, are acceptable given the current state of the monitor. If so, the function should return a new monitor state, otherwise a failure reason.
  + 1. *An Example Safety Monitor*

In this section, a safety monitor which checks some properties for the nos supervisor is shown. The relevant properties of interest are:

* + - * a child is spawned only if it has died
      * no process is killed by the nos supervisor without having a sibling process that has terminated too many times (so that it should not be restarted again).

To be able to determine whether a process spawning or a kill action caused by the nos supervisor is allowed, the monitor has to keep track of the state of all its children. Thus, the state of the monitor is a record keeping track of the status of all child processes under supervision:

- record ( monitorState ,

*% List of dead or never started children*

*% ordered by the crash time*

{ deadProcesses = []

*% children killed by supervisor*

, killedProcesses = []

*% spawned children currently alive*

, spawnedProcesses = []

*% number of restarts for children*

, supervisorChildren = []

, supervisorPid = undefined

}).

The main function of the monitor is depicted below:

state Change ( \_, MonState , Actions ) ->

...

case interpret\_action ( Action ) of

{ died , Pid , normal } ->

{ ok , Monstate };

{ died , Pid , Reason } -> died ( Pid , Monstate );

{ spawn , SpawnInfo , Spawned Pid } -> case Spawn Info of

{ supervisor , Intensities }

-> set Children ( Intensities , Monstate );

{ worker , WorkerName }

-> spawned ({ WorkerName , SpawnedPid }

, Monstate )

end ;

{ killedby , SourcePid , Killed Pid } -> killed ( KilledPid , Monstate );

\_ ->

{ ok , Monstate }

end

...

where the function interpret action splits concrete program actions into different abstract categories, and for each one it checks whether that action is acceptable in the current monitor state. The action *{*died, Pid, normal*}*, for instance, represents the fact that a process has terminated normally (and

so should not be restarted). If the abstract action represent a new child process being spawned, the function spawned is called:

spawned ({ WorkerName , Pid }, State ) -> DeadProcs =

State # programState . deadProcesses , SpawnedProcs =

State # programState . spawnedProcesses ,

case lists : member ( WorkerName , Dead Procs ) of true ->

State # programState

{ dead Processes = lists : delete ( Worker Name

, DeadProcs ), spawnedProcesses = [{ Worker Name , Pid }

| SpawnedProcs ]};

false ->

throw (" Already . spawned . worker ")

end .

The spawned functions checks that the (name of the) newly spawned child function is not already spawned, and removes the new child from the list of dead processes and then adds it to the list of spawned ones. If the child is already spawned, an exception is thrown which notifies McErlang that the monitor has found an error.

Similarly, when a child dies abnormally the died function is called, which increments the restart counter for the child. This counter is checked when the nos supervisor kills a child to ensure that a sufficient number of restarts have occurred for some sibling of the killed process.

It is interesting to note the way in which we formulate correctness prop- erties using safety monitors. Instead of specifying them in a temporal logic, what is done is to develop a set of simplified models of the system, and then check whether the behaviour of the real component corresponds to these mod- els. The use of the same language for regular development and for specifying safety monitors eases the adoption of this technique by the developers.

* + 1. *Checking Liveness Properties*

Some properties, i.e., liveness properties that express claims regarding eventual behaviour, cannot be implemented as safety monitors. In this case, we can instead formulate the property in LTL and use the automatic translator from LTL formulae to Bu¨chi monitors. For example, in the case study an interesting property to check is that always that a child terminates abnormally, it gets eventually restarted. This property is expressed as:

always (P => eventually Q)

where P and Q are predicates stating that *a child terminates abnormally* and *a dead child gets restarted*, respectively. Such predicates are specified in Er- lang, and the correspondence between the names in the LTL formula and the concrete implementation must be given to McErlang. As an example of a predicate, this function implements the predicate that states that *a child terminates abnormally* :

child\_terminated ( \_, Actions , \_) -> lists : any

(fun ( A) -> try

died == mce\_erl\_actions : type (A),

normal =/= mce\_erl\_actions : get\_died\_reason ( A) catch \_: \_ -> false end

end , Actions ).

As can be seen, a predicate is a function which receives three arguments. The first is the program state, the second are the actions which triggered the state change, and the third is some private state. If any action is an action corresponding to a process dying (died action), with a Reason different than normal, this means that a child terminated abnormally.

Note, however, that this property holds only for a nos supervisor whose children have restart intensity infinity; otherwise, after a sufficient number of deaths, the child would not be restarted. To check more complex features,

we need to keep a private state, and pass it along between the state predicates (thus corresponding to a monitor state in the generated Bu¨chi automaton).

* + 1. *Modular Safety Monitors*

Instead of trying to check all the desirable properties of the system using a single complex safety monitor, it is advisable to define several safety monitors, each checking a particular property of the system. This approach reduces the risk of checking the system with incorrect safety monitors, as the resulting monitors individually are much easier to understand and write. Moreover, a large part of a safety monitor code is reusable when writing a new one. That is, the strategy in which concrete actions are translated into abstract ones and the way the monitor state is updated upon process deaths, spawn and kills, is completely generic. What changes from one safety monitor (property) to another is how the actions are interpreted – i.e., when the occurrence of an action signal is interpreted as an error.

These are some of the safety monitors used in the verification of the su- pervisor component:

1. **A supervisor will always try to restart a child, until one reaches the maximum restart intensity.** Applicable for checking child speci- fications with restart intensity different than infinity, and with kill sup finally action.
2. **When a child reaches its maximum restart intensity, living work- ers are killed in reverse start order and then it terminates.** Appli- cable for child specifications with restart intensity different than infinity, and with kill sup finally action.
3. **When a child reaches its maximum restart intensity, living work- ers are stopped in reverse start order.** Applicable for child specifi- cation with restart intensity different than infinity with kill sup finally action and infinity as shutdown strategy.
4. **If a child has infinity as shutdown strategy, the supervisor never kills it.**
5. **If a child has an integer as shutdown strategy, a supervisor never kills a process before it has tried to stop it.**
6. **When the shutdown strategy for a child is** stop child**, when the supervisor stops, the workers which were not respawned had reached their maximum restart intensity.**
7. **For the** all **restart strategy, when a child dies, the supervisor kills alive children in reverse starting order.**
8. **For the** all **restart strategy, the workers are restarted in start order after killing all alive children.**
   1. *Veriﬁcation Scenarios*

To verify the nos supervisor against the above correctness properties using McErlang, a number of scenarios were designed manually, although it should not be too difficult to generate them automatically (e.g., using QuickCheck-

/Erlang [[2](#_bookmark16)]). To create this scenarios, a test worker nos test worker has been implemented which only keeps track of its starting time. To simulate process crashes, we enable an option in the McErlang model checker which, non-deterministically, kills any process in any state. We select a subset of the running processes for termination by executing:

mcerlang:process flag(do terminate, true)

in any process that is a candidate for termination (in the case study, the worker processes). Thus, a worker processes can non-deterministically die at any moment, which will cause the nos supervisor to be informed (and hopefully take action).

As an example of a concrete scenario, to check the liveness property *if a child dies then eventually the nos supervisor will restart the child* the simplest adequate scenario would be a nos supervisor which spawns a child with restart strategy child (the rest of options are not relevant).

{ worker1 ,

{ nos\_test\_worker , start\_link ,[ worker1 , foo ]},

{ nos\_test\_worker , restart\_link ,{ worker1 , foo }}, child , infinity , brutal\_kill , []

}

We identified the relevant scenarios as a nos supervisor with, at least one

(two or three depending on the property) child processes. All the child pro- cesses in a scenario have the same specification for simplicity.

Each scenario was chosen to verify properties related to the behaviour of the nos supervisor related to some of its options. For example, if we want to verify that the nos supervisor evaluates a Finally action correctly when a child reaches its maximum restart intensity, we only require a restart intensity different than infinity, but we must know if we should expect a exit(kill) or a exit(shutdown) (which shutdown strategy is being applied).

* 1. *The McErlang Debugger*

What can be done if a property fails? McErlang provides a tool for explor- ing counterexamples, manually exploring the state space, *.. .*: the McErlang

debugger.

When running the model checker with safety monitor 2, McErlang returns a counterexample, indicating that this property has failed:

\*\*\* Property violation detected at minimum depth 13

\*\*\* Monitor failed monitor error:

{failed\_monitor

,"Processes not killed in reverse start order"} Stack depth 13 entries;

state table contains 44446 states.

Access result using get(result)

To see the counterexample type "mce\_erl\_debugger:start(get(result)). "

...

As can be seen, the length of the trace which leads to an error is shown, as well as the concrete scenario of the failure. In the program trace given by the debugger, we realized that the kill order was not the proposed in the specification. The debugger also allows us to manually explore the state space to analyze other paths, and help us to locate the error (under which circumstances does this error arise).

* 1. *Experimental Results*

The safety monitors previously described were checked on a set of scenar- ios, constructed as described in Sect. [4.3](#_bookmark9). We present here some measures taken in a Intel(R) Core(TM)2 Quad at 2.33GHz with 4GB of RAM mem- ory. Figure [2](#_bookmark11) shows the number of states explored (Fig. [2a](#_bookmark10)) by the model checker and the time required (Fig. [2b](#_bookmark10)) to check the safety monitor (i) given in Sect. [4.2.3](#_bookmark8), for a scenario with restart strategy child, a restart inten-

sity of *{*1,1,kill sup*}*, and different number of workers. The state space

may vary depending on scenario parameters (using the same number of work-

ers). For example, the number of states explored by the model checker using

*{*1,1,kill sup*}*, *{*2,1,kill sup*}* and *{*3,1,kill sup*}* were 6617, 13656 and 26712, respectively.

As expected with model checking techniques, the state space grows expo- nentially making difficult to check large scenarios using the actual implemen- tation as a model. Nevertheless, even though nos supervisor behaviour has been deployed in a number of LambdaStream products and no truly critical errors were expected to be found, surprisingly, one discrepancy between the specification in the component documentation and the actual implementation of the nos supervisor was found. McErlang returned a counterexample for

106

105

States explored

104

103

103

102

Time (s)

101

100

10*−*1

1 2 3

Number of workers

1. States explored

1 2 3

Number of workers

1. Time required

Fig. 2. States explored and time required (restart strategy=child, restart inten- sity=*{*1,1,kill sup*}*).

the following combination of scenario and property *If any children with spec- iﬁcation kill sup and Finally reaches the maximum restart intensity, then the supervisor only kills a child after all the “younger” children (those that have been started after this child) are not running (stopped, killed, crashed, dead, .. .)* . The counterexample returned by McErlang was analyzed using the McErlang debugger and it turned out that an “older” worker was killed before a “younger” worker, i.e., they were not killed in reverse start order as it was explicitly stated in the documentation. Although this discrepancy does not seem relevant, it may cause a misbehaviour in some scenarios. For example, if mutually dependent workers need to perform some actions before stopping in a specified order.

# Conclusions

In this paper we have explored the verification of safety and liveness properties using the McErlang tool on a process supervisor deployed as part of several real-world products. Thanks to this verification, we have improved the compo- nent’s reliability, not only because a slight discrepancy between specification and implementation was identified but also because component specification became much more precise. From this verification effort we can conclude that model checking is a valuable technique for analyzing concurrent programs, and that McErlang can be succesfully applied to industrial software.

The methodology we have followed consists of three steps (a) creating a model for the component, (b) expressing and implementing the properties of interest, and (c) creating scenarios where the properties were checked. Creat-

ing the model is a straightforward task as McErlang can use the source code as a model with minimal changes. In the case study, the only required change was to abstract from the timing aspects of the supervisor, into a non-deterministic choice, as currently McErlang implements neither real-time nor discrete-time model checking algorithms. Even though we have been able to verify most as- pects of the supervisor without considering exact timing, an important aspect for future work is to add support for real-time model checking algorithms to McErlang.

The properties of interest were extracted from the component documen- tation, and from informal discussion with the developers of the supervisor component. Most of the properties were formulated as safety monitors, writ- ten in Erlang, which observe the actions of the supervisor component as it manages a set of children, and signaling an error if the supervisor issues an incorrect command. In other words, we have defined a set of simplified mod- els, and check that the real supervisor has the same behaviour as the models (up to the abstraction level of the monitor).

To fight the inevitable state explosion problem of model checking we apply the McErlang model checker not to a large monolithic scenario, but define and check a large number of smaller scenarios (varying, for example, the number of work processes). Even though the verification is partial, as there is no way we can check every possible scenario, we discovered a discrepancy between the documentation and the actual implementation of the component in a small scenario comprised of one supervisor and three children.

A significant advantage of the approach to model checking taken in McEr- lang, compared to other model checkers, is that there is no need for learning a new specification language since Erlang is both used as a programming lan- guage and as a specification language. This is achievable, in part, due to the inherent power of a functional programming language which is sufficiently expressive to be used both for programming and for writing more abstract specifications.

This technique can be used to check properties of larger systems, although a set of small scenarios must always be chosen.

Still obviously there is much room for improvement. The learning curve to be able to use McErlang effectively, and to formulate correctness properties, is currently too steep. In next release of McErlang improvements are planned regarding the usability of the tool, to provide, for example, better information on error causes, guidance in selecting appropriate verification options, and a simplified API for the formulation of correctness properties.

# References

1. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
2. T. Arts and J. Hughes. QuickCheck for Erlang. In *Proceedings of the 2003 Erlang User Conference (EUC)*, 2003.
3. J. B¨chi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science 1960*, pages 1–11. Stanford University Press, 1960.
4. C. Benac Earle, L. Fredlund, J. Iglesias, and A. Ledezma. Verifying robocup teams. *Lecture* *Notes in Computer Science*, 5348/2009:34–48, 2009.
5. F. Cesarini and S. Thompson. *Erlang Programming*. O’Reilly Media, 2009.
6. L. Fredlund and J. S´anchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.
7. L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9):125–136, 2007.
8. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
9. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
10. R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
11. H. Svensson. Implementing an LTL-to-Bu¨chi translator in Erlang. In *Proceedings of the 2009 ACM SIGPLAN Erlang Workshop*, 2009.
12. S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.