Using McErlang to Verify an Erlang Process Supervision Component[*]

David Castro[1], Clara Benac Earle[2], Lars-Åke Fredlund[2], Victor M. Gulias[1], and Samuel Rivas[3]

[1] MADS Group, Computer Science Department
University of A Coruña, Spain
{dcastrop,gulias}@udc.es

[2] Babel Group, School of Computer Science,
Universidad Politécnica de Madrid (UPM), Spain
{cbenac,lfredlund}@fi.upm.es

[3] LambdaStream Servicios Interactivos S.L.
Ronda de Outeiro 33 Entlo., A Coruña, Spain
samuel.rivas@lambdastream.com

Abstract. We present a case-study in which a tool for model checking programs written in Erlang, McErlang, was used to verify a complex concurrent component. The component is an alternative implementation of the standard supervisor behaviour of Erlang/OTP. This implementation, in use at the company LambdaStream, was checked against several safety and liveness properties. In one case, McErlang found an error.

1 Introduction

Developing reliable concurrent software is a hard task given the inherent non-deterministic nature of concurrent systems. A technique which is often used to check that a concurrent program fulfils a set of desirable properties is model checking [12]. In model checking, in theory, all the states of a concurrent system are systematically explored.

Erlang [6] is a functional programming language that is used by several companies worldwide. One such company is LambdaStream, which is dedicated to improving their software development methodology, as shown by their participation in the European research project ProTest[1]. Thanks to this project, a fruitful collaboration has been established between LambdaStream, the University of A Coruña and the Universidad Politécnica de Madrid. One result of the collaboration is the verification, using the McErlang model checker, of a process supervision component developed by LambdaStream. Although the supervisor has been used in several products, and was well tested, we did find a discrepancy between its documentation and the implementation of the component.

[*] This work has been partially supported by the following projects: ProTest (FP7-ICT-2007-1 215868), DESAFIOS10 (TIN2009-14599-C03-00), and PROMETIDOS (P2009/TIC-1465).

[1] http://www.protest-project.eu/

A process supervision component 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 a critical software component that is used in several products at Lambda-Stream. Fortunately, the McErlang model checker is well suited to verify this class of complex concurrent software components.

McErlang [11] is a model checker for programs written in the Erlang programming language. The model is the Erlang program to be analysed, 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 Erlang byte code. Finally the program is run under the McErlang run-time system, under the control of a verification algorithm, by the normal Erlang byte-code interpreter. The pure computation part of the code, i.e, code with no side effects, 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. Naturally the new run-time system offers easy check pointing (capturing the state of all nodes and processes, and all messages in transit between processes) of the whole program state. One of the chief advantages of using McErlang compared to a traditional testing framework is that the tool provides direct access, and therefore, control, of the runtime system.

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

For non-Erlang programmers the approach has merits too. Erlang is a good *specification* language due to its root in functional programming: higher-order functions enables concise specifications, software components further lifts the abstraction level of specifications, and programs can be treated as data – yielding a convenient meta level. Moreover, Erlang is a good platform for specifying *distributed algorithms* since language features matches common assumptions in algorithms: *isolated* processes communicate using message-passing only, *fault-tolerance* is obtained by using unreliable failure detectors, process *fairness* is built into the language, and *locality* is important – communication guarantees are stronger for intra–node communication than for inter–node communication.

The rest of this paper is organised as follows. A short survey on related work is presented in Section 2, whereas Section 3 introduces Erlang and McErlang. The process supervisor component is described in Section 4 together with a brief discussion of the correctness properties we consider. The formalisation of such properties, and the experimental results of checking them with McErlang, is explained in Section 5. Finally, Section 6 draws conclusion and outlines items for future work.

2 Related Work

Many techniques and tools for verifying concurrent and distributed systems have been developed. SPIN [12] is a well known model checker for distributed software systems, which uses Promela (a C-like language) as the modeling language. The Symbolic Analysis Laboratory (SAL [7]) is a a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of transition systems. Some other model checkers use the source code as modeling language, like Bogor [8] and Java Pathfinder [17]. A key difference between these tools and McErlang is the programming language they analyse, e.g. Java and Erlang. By using Erlang we have all the advantages (and disadvantages) of using a functional programing language for specification and programming (access to higher-order functions, no shared variables, etc.). In contrast to e.g. the Java Pathfinder tool we do not implement a new byte code interpreter (for Beam, the Erlang byte code format). The chief reason for this is that the Erlang byte code format is much less standardised than Java's; indeed there is no documentation available apart from the source code. In McErlang, rather, programs are run by the normal Erlang byte code interpreter, but have been modified to return control to the model checker after a side effect has occurred.

For Erlang the *etomcrl* toolset [2] already provides a model checking capability. In comparison, however, the *etomcrl* toolset supports only a smaller subset of Erlang, for instance lacking the concept of direct process communication, distribution and fault tolerance (i.e. nodes, processes, links, monitors, . . .). Other verification tools for Erlang include Huch's abstract interpretation model checker [13] which uses abstract interpretations to reduce the size of the state space. Another work addressing the verification of Erlang programs is the "*Verification of Erlang Programs*" project [9] which uses theorem proving technology. Furthermore there is a QuickCheck tool (originally for the Haskell programming language) for Erlang [3], which however primarily focuses on testing software rather than formal verification.

3 Background

3.1 Erlang Programming Language

Erlang [1, 6] is a concurrent programming language and run-time system. The sequential subset of the language is a dynamically typed functional programming 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[2]. Expressions are evaluated eagerly similarly to, for instance, Standard ML.

[2] 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.

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 [14]. Because of the absence of side-effects, which are limited to explicit inter-process communications, concurrent programming is far simpler compared to standard imperative languages.

An Erlang virtual machine (node, in Erlang terminology) 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 low creation and context-switching overheads.

A distributed Erlang application consists of processes running in several nodes, possibly on different computers. Even though the initialisation and management of each node is platform dependent, the nice feature is that communication among remote processes is semantically equivalent in a distributed framework – though remote communications are less efficient, of course.

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 language components provided by libraries. The OTP component library [16] is by far the most widely used library, offering various design patterns such as a generic server component (for client-server communication), a finite state machine component, TCP/IP communication, and a supervisor component for the structuring of fault-tolerant systems. In Erlang these design patterns are called behaviours, and they provide functionality in a fashion similar to inheritance in object-oriented programming or interfaces in Java. A number of callback functions must be defined for each behaviour to work.

In this paper we focus on verifying a process supervisor similar to the one in the OTP library. Such a process supervisor is used by Erlang applications to structure processes hierarchically in a tree structure, where a parent supervisor is responsible for supervising and managing its children (work processes).

3.2 McErlang, a Model Checker for Erlang

McErlang [11] is a tool for model checking programs written in Erlang which has been successfully applied in a number of case studies, for example: a Video–on–demand server [10], leader election protocols [11], the control software for elevators, and multi-agent systems playing for robotic soccer [4].

The steps needed to perform model checking with McErlang are the following:

Create the model. McErlang provides 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 behaves in a subtly different way from intra-node 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.

Formulate correctness properties. Write down the correctness properties to verify as either safety monitors or as formulas in linear temporal logic (LTL).

A safety monitor is used to verify safety properties (*something bad never happens*). In McErlang a safety monitor is an Erlang function, with an internal state, that given a program state determines whether the program state violates some correctness property. Given a program and a monitor, McErlang runs them in lockstep letting the monitor investigate each new program state generated. If the property does not hold, a counterexample (an execution trace) is generated.

Some properties cannot be expressed with a safety monitor, for example, liveness properties (*something good eventually happens*). In McErlang one can express such properties in LTL, and use the LTL2Buchi tool [15] to automatically translate LTL formulas into Büchi automata [5].

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). In the case study we for instance specify monitors that observe the termination of a process, and the creation of new processes.

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

4 Case Study: A Supervisor

A supervisor is a process responsible for starting, stopping and monitoring a set of children (processes). Basically whenever a child process terminates, the supervisor should restart the child, i.e., spawn a new process executing the task of the terminated child.

In Fig. 1 a typical supervision tree is depicted. Note that the tree is multi-level, a supervisor may have other supervisors as children.

The two main policies for how a restart is performed 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 terminated and then all children, including the originally terminated process, are restarted.

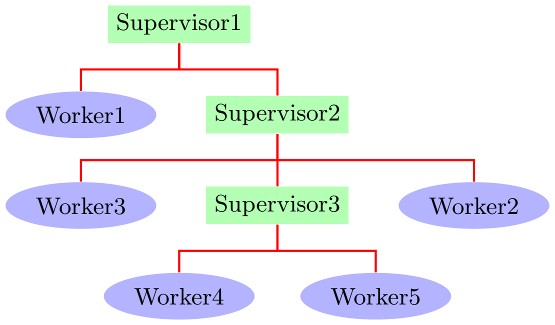

Fig. 1. A supervision tree.

There is a mechanism to prevent a situation where a process repeatedly 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 terminates, then the next higher level supervisor can take corrective action. That is, either restarting the terminated supervisor (and thus indirectly its children), or terminating itself.

4.1 Supervisor Implementation

There is a standard implementation of the supervisor behaviour provided by the open source distribution of Erlang/OTP (Open Telecom Platform) [6]. It is implemented as a behaviour, i. e., a library implementing a design pattern which provides functionality in a fashion similar to inheritance in object-oriented programming or interfaces in Java.

In this case study we verified an implementation of the Erlang supervisor behaviour slightly different from the OTP implementation. This variant was implemented by the company LambdaStream because they wanted a more configurable supervisor. In particular, in their supervisor (from now on nos_supervisor) different restarting policies are allowed in each child node.

In this section we will focus on the implementation of the nos_supervisor. A child process is spawned by a supervisor process providing a specification for the child. In the nos_supervisor a child specification is a tuple:

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

where

– `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`:
  - {`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 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.
  - `integer()`. The supervisor will inform the child process that it should terminate and then wait to receive an exit signal from the child signalling that it indeed has terminated. If no exit signal is received within the specified time, the child process is unconditionally terminated.
  - `infinity`. As in `integer()` but without timeout to give the subtree ample time to shutdown.
- `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 notification messages to the specified process identifier (pid).

4.2 Relevant Properties of the nos_supervisor

By reading the nos_supervisor documentation we came up with many interesting properties to verify for the nos_supervisor. As a trivial example, we want to check that a terminated child is actually restarted by its supervisor. We also focused on properties regarding the different restart intensities 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 restart intensity set to `kill_sup` and the `Finally` action reaches the maximum restart intensity, the supervisor applies the `Shutdown` strategy to its children in reverse start order and then finishes. In the following section we describe how some of these properties were verified using McErlang.

5 Checking Properties with McErlang

To verify the nos_supervisor described in the previous section using McErlang we follow the approach described in Sec. 3.2, i.e., we create a verifiable model, we formulate a number of correctness properties, and we generate a set of scenarios.

5.1 Create the Model

Obtaining the model of this case-study with McErlang was straightforward. 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.

The original code of the nos_supervisor deals with restarts (and time) in the following way:

```
add_restart(#child_spec{restart_intensity = infinity}) -> [];
add_restart(#child_spec{restart_intensity = {MaxR, MaxT, Finally},
                        state = ChildState}) ->
  Restarts = ChildState#child_state.restarts,
  check_restarts(MaxR,Finally,filter_restarts(MaxT,[now()|Restarts])).
```

As we can see below, 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 length of 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 we adopted was not to store concrete time when a restart took place, but merely recording the fact 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 code fragment below:

```
  NewRestarts = 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; instead the model checker must consider two possibilities: whether these restarts happened within the maximum time or not. In McErlang such a non-deterministic choice can be expressed using a function call to `mce_erl:choice` with a list of anonymous functions (continuations) as argument. During model checking all 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.
```

5.2 Formulating Correctness Properties

As explained in section 3.2, in McErlang one can specify correctness properties 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 (LTL).

In practise the majority of the nos_supervisor correctness properties we are interested in checking 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 call-back functions:

- an `init` function to initialize the monitor (which should return the initial state of the monitor).
- a `stateChange` function which is called, after the model checker has computed a new system state, with the following arguments: the new system state, the current state of the monitor, and 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.

An Example Safety Monitor. As an example we show below a safety monitor, which checks a number of behavioural properties for a nos_supervisor controlling a set of child processes:

- no child is spawned twice (without first dying)
- 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 the processes managed by the nos_supervisor. Thus the state of the monitor is a record keeping track of the status of all child processes under supervision:

```
-record(monitorState,
    { deadProcesses = []       % List of dead or never started children
                               % ordered by the crash time
    , killedProcesses = []     % children killed by supervisor
    , spawnedProcesses = []    % spawned children currently alive
    , supervisorChildren = [] % number of restarts for children
    , supervisorPid = undefined}).
```

The main function of the monitor is depicted below:

```
stateChange(_,MonState,Actions) -> ...
  case interpret_action(Action) of
    {died, Pid, normal} ->  {ok,Monstate};
    {died, Pid, Reason} ->  died(Pid, Monstate);
    {killedby, SourcePid, KilledPid} -> killed(KilledPid, Monstate);
    {spawn, SpawnInfo, SpawnedPid} ->
      case SpawnInfo of
        {supervisor, Intensities}
          -> setChildren(Intensities,Monstate);
        {worker, WorkerName}
          -> spawned({WorkerName, SpawnedPid}, Monstate)
      end;
     _ -> {ok,Monstate}
   end ...
```

where the function `interpret_action` partitions concrete program actions into different abstract categories, and for each 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` below is called.

```
spawned({WN, Pid}, State) ->
  DPs = State#programState.deadProcesses,
  SPs = State#programState.spawnedProcesses,
  case lists:member(WN, DPs) of
    true ->
      State#programState{deadProcesses = lists:delete(WN,DPs),
                         spawnedProcesses = [{WN,Pid} | SPs]};
    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 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 encountered 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 manner in which we formulate correctness properties using safety monitors. Instead of specifying them in a temporal logic, we develop a set of simplified "models" of the nos_supervisor system, and then check whether the behaviour of the real nos_supervisor corresponds to our "models".

Checking Liveness Properties. However, some properties, i.e., liveness properties that express claims regarding eventually behaviour, cannot be implemented as safety monitors. In this case we can instead formulate the property in linear temporal logic (LTL) and use the automatic translator from LTL formulae to Büchi monitors.

We have checked using McErlang that the following LTL formula holds:

```
always "a␣child␣terminated␣abnormally" =>  eventually "the␣child␣is␣restarted"
```

Such predicates are specified in Erlang. As an example, the function below implements the predicate `"a child terminated abnormally"`:

```
child_terminated(_, Actions, _) ->
  lists:any
  (fun (Action) ->
     try
       died == mce_erl_actions:type(Action),
       normal =/= mce_erl_actions:get_died_reason(Action),
     catch _:_ -> false end
   end, Actions).
```

As we can see, 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 a 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[3].

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 Büchi automaton).

Verified Safety Monitors. Instead of trying to check all desirable properties of the nos_supervisor using one very complex safety monitor, we define a set of safety monitors, each checking a particular property of the nos_supervisor. This approach, we believe, reduces the risk of checking incorrect safety monitors, as the resulting monitors individually are much easier to understand and write.

Moreover, a very big portion of the code for a safety monitor is reusable when writing a new one. That is, the manner in which we translate concrete actions into abstract ones, and the way the monitor state is updated upon process deaths, spawn and kills, is completely generic. What changes is how we interpret the actions – i.e., when does the occurrence of an action signal an error.

Below we enumerate the different safety monitors we use in verification, indicating for which particular type of nos_supervisor specification the monitors are targeted:

1. A supervisor will always try to restart a child, until one reaches the maximum restart intensity. Applicable for checking child specifications with restart intensity different than infinity, and with `kill_sup Finally` action.
2. When a child reaches its maximum restart intensity, living workers are killed in reverse start order and then it terminates. Applicable 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 workers are stopped in reverse start order. Applicable for child specification with restart intensity different than infinity with `kill_sup Finally` action and infinity as shutdown strategy.

[3] Here we assume that all the processes in the system are managed by the nos_supervisor.

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.

5.3 Verification Scenarios

To verify the nos_supervisor against the above correctness properties using McErlang, we specified a number of scenarios by hand, although it should not be too difficult to generate them automatically (e.g., using QuickCheck/Erlang [3]).

To create these scenarios, we implemented a test worker `nos_test_worker` which does nothing but keep track of the number of times it has been restarted. Moreover 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 the function call

```
mcerlang:process_flag(do_terminate, true)
```

in any process that is a candidate for termination (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). Killing any process in any state is one example where, compared to testing, we benefit from the fact that McErlang provides direct access to the runtime system.

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). Thus, the child specification in this case will be the following:

```
{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 process. All the child processes 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 which shutdown strategy is being applied.

5.4 Experimental Results

We present here some quantitative experimental results, measured on a computer with an Intel(R) Core Quad processor at 2.33GHz with 4GB of RAM memory, of verifying the correctness properties against a scenario. Figure 2 shows the size of the state space (Fig. 5.4), and the time required to check safety monitor 1 (Fig. 5.4), for a scenario with restart strategy `child`, a restart intensity of {`1,1,kill_sup`}, and a variable number of workers.

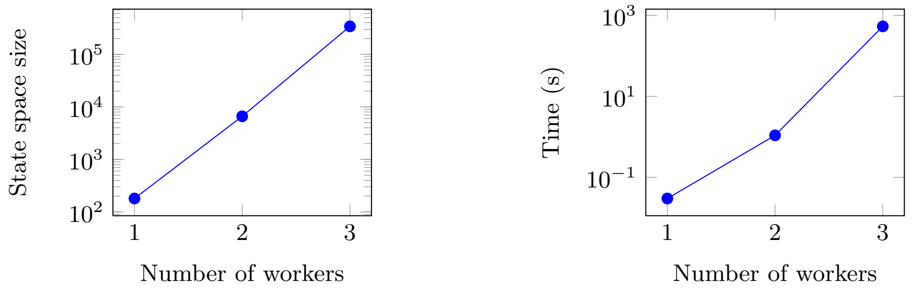

Fig. 2. Size of state space, and verification time required.

As expected, since the nos_supervisor behaviour has been deployed in a number of LambdaStream products that have been working in production for years, no truly critical errors were found. Surprisingly though, one discrepancy between the documentation and the implementation of the nos_supervisor was detected. During a model checking run, McErlang returned a counterexample for the following property: "If any children with specification `kill_sup` and `Finally` reaches the maximum restart intensity and has `brutal_kill` shutdown strategy, 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, ...)". From inspecting the counter example in the McErlang debugger it became apparent 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. Discussions with the developers revealed that the error was in the documentation. Such discrepancies between the documentation and the actual behaviour are a serious violation of the less astonishment principle. They can mislead developers to wrong implementations, wasting precious time and enabling potential bugs to slip in the resulting code.

6 Conclusions

In this paper we have explored the formal verification of a process supervision component written in Erlang, using the McErlang model checking tool. The methodology used for verification is rather standard: we create a model for the component, express and implement the correctness properties of interest, and check the model against the properties as constrained by a set of scenarios. Our approach is an improvement upon a normal model checking work-flow in that constructing a model from the component is mostly trivial as the source language of the model checker is virtually full Erlang. Moreover as Erlang is used also to formalise the correctness properties there is no need for the user to learn a special purpose 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.

Creating the model from the source code of the nos_supervisor component was a straightforward task. The single change needed 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 aspects of the supervisor without considering exact timing, an important item for future work is to add support for real-time model checking algorithms to McErlang.

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

Finally, to combat the inevitable state explosion problem of model checking we applied the McErlang model checker not to a large monolithic scenario, but have defined and checked a large number of smaller scenarios (varying for example the number of work processes, etc). Even though the verification is partial, as there is no way we can check every possible scenario, we discovered an error in a small scenario comprising one supervisor and three children.

From this verification effort we can conclude that model checking is a valuable technique for analysing concurrent programs, and that McErlang is capable of analysis and finding bugs in industrial software.

Compared to e.g. using random testing, model checking can provide firm guarantees that smaller or mid-sized scenarios are fully verified. For larger scenarios and systems, neither random testing nor model checking will be capable of a full verification. It is an interesting area for future research to quantify how statistically important is the capability of not re-testing system states that model checking provides, compared to the tendency of random testing to be able to execute a test (scenario) quicker than a model checker. Still, compared to testing, with McErlang we have the advantage of full and uncomplicated control of the execution environment of a scenario, i.e., the Erlang runtime system. In the case study we used this feature to specify that a worker process should be subjected to abrupt termination at any moment, regardless of its state, simply by changing a pre-existing verification parameter in the model checker. For testing we would have had to modify either the source code of the worker process itself, or add a new process to send a termination message to each worker. In general, by controlling the runtime system, we can often write less complicated scenarios.

We expect the same advantages of using a McErlang based verification methodology to be applicable in the verification of other distributed programs. In particular, to verify the standard OTP supervisor should be a straightforward undertaking using the same approach as in this paper.

Still there is room for improvement. The learning curve for using McErlang effectively is currently too steep. Thus we are currently focusing on improving the usability of McErlang, to provide, for example, better visualising of errors, guidance in selecting appropriate verification options, and a simplified API for expressing correctness properties.

References

1. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
2. T. Arts, C. Benac Earle, and J. Sánchez Penas. Translating Erlang to mucrl. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004)*. IEEE Computer Society Press, June 2004.
3. T. Arts and J. Hughes. QuickCheck for Erlang. In *Proceedings of the 2003 Erlang User Conference (EUC)*, 2003.
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. J. Büchi. 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.
6. F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.
7. L. de Moura, S. Owre, and N. Shankar. *The SAL language manual*.
8. M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *CAV*, pages 148–152, 2005.
9. L. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):405 – 420, Aug 2003.
10. L. Fredlund and J. Sánchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.
11. L.-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9):125–136, 2007.
12. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
13. F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, 1999.
14. R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
15. H. Svensson. Implementing an LTL-to-Büchi translator in Erlang. In *Proceedings of the 2009 ACM SIGPLAN Erlang Workshop*, 2009.
16. S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.
17. W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker, 2000. In Proc. of Post-CAV Workshop on Advances in Verification.