Towards a Specification-Based Correctness of Erlang Systems Through Asynchronous Monitoring

Christian Colombo
Dept. of Computer Science
University of Malta
christian.colombo@um.edu.mt

Adrian Francalanza
Dept. of Computer Science
University of Malta
adrian.francalanza@um.edu.mt

ABSTRACT

We argue that runtime verification, in the guise of monitor-oriented programming, is a natural way how to ensure correctness in dynamically-typed, concurrent languages such as Erlang. Our proposal involves a methodology for marrying correctness runtime checking with the fail-fast approach in Erlang, exploiting the rich failure-handling mechanisms of the language. This allows us to smoothly integrate correctness analysis within existing language code practices.

1. INTRODUCTION

Erlang [1] is an established, industry strength, concurrent language for constructing distributed, fault-tolerant scalable systems. It is used mainly in the Telecoms industry, an area where program correctness is of paramount importance. Ensuring correctness in Erlang is however non trivial mainly because the programs constructed are inherently non-deterministic: every execution of a concurrent set of processes may yield a different interleaving, potentially resulting in different behaviour. Unfortunately, exhaustive methods such as model checking suffer from state explosion, particularly when applied to concurrent code with multiple execution paths.

Runtime Verification (RV) is an appealing compromise towards ensuring Erlang code correctness, as it circumvents this problem by verifying only the current program execution. RV usually employs a *monitor* which executes alongside existing code so as to analyse the execution of the monitored program at runtime. It can sometimes take the guise of Monitoring-Oriented Programming (MOP) [4], a paradigm which advocates separation of concerns by delegating *cross-cutting concerns* to monitors.

In this paper we argue that an MOP approach is particularly relevant to programming languages such as Erlang, where several correctness checks cannot but occur at runtime due to inherent language features such as dynamic variable typing, remote execution of code and on-the-fly code modification, *i.e.,* hot code swapping. These runtime checks often enhance expressivity (as opposed to static checks) and allow for rapid prototyping of software. However, they tend to clutter the code making it hard to understand and maintain and, if misused, may lead to defensive programming. We argue that these runtime checks are perfectly eligible to be treated as cross-cutting concerns: using monitors to handle such checks would relieve the main code from the clutter and enable the reuse of monitors across the whole system.

Monitoring is not new to Erlang. In order to build fault-tolerant code, Erlang common code practices advocate for the *fail-fast* design pattern: code is structured in such a way that errors in a process are not handled by the process itself; instead the process is left to fail as a result of such errors, and its abnormal termination is then detected by *monitoring (supervising) processes* linked to the failing process; once this is detected, supervising monitors can take the necessary actions, ranging from the launching of other processes replacing the failed process, to failing themselves and letting other processes handle their failure. In most cases, this asynchronous way of handling errors suffices to ensure fault-tolerant behaviour in Erlang, since every process has its own local memory and the effect of erroneous behaviour can be readily localised and isolated.

While basic Erlang superisor-style monitoring detects process failures, correctness violations are often specification-dependent, behav-ioural errors that do not necessarily lead to a process failure. Thus, when a behavioural error occurs, the monitor might never be notified and the correctness violation (which constitutes a software failure) goes unnoticed, breaking the fail-fast convention. This fact complicates the task of building Erlang code that is *fault-tolerant* with respect to specification-violating errors.

In this paper, we rectify this problem by proposing a mechanism that packages a monitor-oriented approach to correctness as a fail-fast code pattern, reducing RV monitoring in Erlang to a fault-tolerance problem. We elaborate more on the proposed approach in Section 3 and show case it through an elevator example in Section 4. This is preceeded by a brief outline of the main Erlang mechanism used in Section 2. Section 5 concludes.

2. BASIC CONCEPTS IN ERLANG PROGRAMMING

Erlang is a functional language based on the *actor model* of concurrency[6] whereby processes, *i.e.,* actors, identified by a unique ID, own a message queue, *i.e., mailbox*, and a set of local immutable variables, *i.e.,* local memory. Inter-process communication occurs through message passing: a process communicates with another process in two phases by first sending an asynchronous message to the other process's mailbox, uniquely identified by the respective process ID; the receiving process can then select which messages to read by pattern matching the messages held in its mailbox queue.

Erlang coding practices discourage defensive programming [3], which manifestist itself through code that tries to anticipate eventualities of error conditions and handle these anomalies locally, typically through constructs such as `try-catch` blocks. Instead, process code should focus on handling only correct cases and simply crash, *i.e., fail-fast*, when unexpected computation occurs. This fail-fast approach carries a number of advantages all of which improve the understanding and management of concurrent Erlang code:

(*i*) it keeps the program logic simple by separating error recovery from normal code;

(*ii*) it standardises the way crashes and recoveries are handled and;

(*iii*) it simplifies the task of detecting the error and localising the source of the error.

A manifestation of the fail-fast pattern (in Erlang) is often attained by organising processes using the *linking* and *exit-trapping* mechanisms [3]. In Erlang, whenever a process fails, the system generates an *exit-message* (containing the reason why the process failed) and sends it to other processes, notifying them about the process failure. The processes listening for such exit messages are defined through process linking: spawned processes can be linked together (in uni/bidirectional fashion), thus registering themselves to the listening of exit messages from linked processes. Once an exit message is delivered, there are two possible outcomes. If a linked process is set to *trap* these exits, it receives this message in its mailbox, which allows for failure detection once the message is read[1]; otherwise, a non-trapping process fails as well, propagating again the exit message to its linked processes. Erlang also allows processes to explicitly send exit messages to other processes (without dying themselves), which effectively acts as a *kill* directive to the non-exit-trapping receiving process, effectively forcing them to terminate abnormally.

This code organisation is so prevalent that its operation has been standardised as an OTP behaviour, *i.e.,* an Erlang higher-order library module, called the Supervisor behaviour [7]. Code organisation taking advantage of this behavior can be depicted as in see Figure 1. There, the exit-trapping process monitoring other processes for their abnormal termination, referred to as the *supervisor* process from now on

[1]Message selection through pattern matching permits the prioritisation of exit messages.

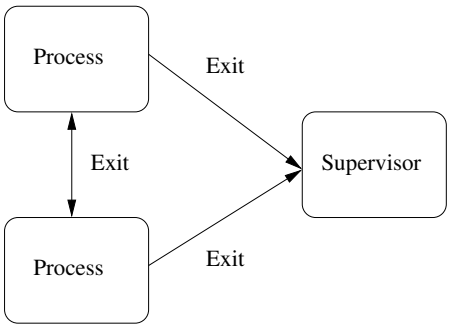

Figure 1: Standard Erlang supervisor code arrangement.

is linked (unidirectionally) to the processes it is supervising and set to trap exit messages; the supervised processes themselves may be linked amongst themselves, but crucially are not set to trap exit-messages. Thus, if one the supervised processes fails, the linked processes automatically recieve exit-messages. However, whereas other supervised processes fail and terminate as well (as a form of termination join), the supervisor process can trap the message and take corrective action.

3. PROPOSED APPROACH

Correctness violations are often specification-dependent, behavioural errors that do not necessarily lead to a process failure. As a result, rudimentary monitor instrumentation through Erlang built-in mechanisms would not conform to the fail-fast practice. This complicates the task of building Erlang code that is *fault-tolerant* with respect to specification-violating errors.

We propose a monitor-oriented programming approach to ensure program correctness in Erlang that adheres to the fail-fast pattern, thereby allowing for a smoother integration within existing language practices. The resulting process organisation is depicted in Figure 2, employing the supervisor setup discussed earlier in Section 2. In addition, correctness criteria (possibly described through an external specification logic) is synthesized as an Erlang process that we refer to as the *verifier*. The role of the verifier is to act as a secondary/internal monitor analysing the behaviour of the program.

Once a correctness violation by the monitored program is detected, the verifier *externally* induces the monitored process to fail, by sending it a kill exit-message detailing the violation. Assuming that the program is not set to trap exits, this will cause it to terminate abnormally[2] thereby exposing the specification-based correctness violations.

Crucially, from the point of view of the supervisor process overseeing the process, the synthesised verifier acting in the background becomes part of the supervised program itself,

[2]If the program consists of a number of processes, they can all be linked to one another in non-exit-trapping fashion whereby sending an exit signal to any one of them will cause it to fail and, as a result, propagated the exit message to the rest of them automatically.

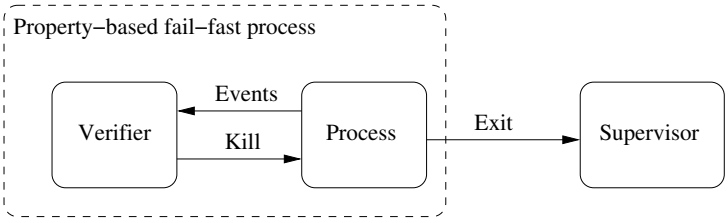

Figure 2: The fail-fast RV approach proposed.

i.e., the dotted lines in Figure 2, and correctness-violation errors can be handled in uniform fail-fast fashion by the supervisor, just like any other failure-generating error. This also means that supervising code, which is often considered to be higher-priority, last-resort code, is unaffected by our proposal, which facilitates the adoption of our approach within existing Erlang practices.

4. CASE STUDY

We consider a simple elevator system where a user requests the elevator at a particular floor by pressing the respective floor button; once the elevator arrives, the user (enters the elevator and) presses the button of the destination floor so that the elevator progresses to the requested floor.

A possible elevator implementation is one that can only service one user request at a time; any button pressed while servicing a request is ignored. Thus a single variable suffices to keep track of the current floor being served, which can only be set if no requests are being serviced at the time, as shown below:

```
case event of
  {press,n}
     -> case req of
         nil -> _req = n
         _   -> _req = req
       end,
  {serve,n}
     -> _req = nil
end;
```

A button press, *i.e.,* when an event received is of the form `{press,n}`, sets the variable `req` to the requested floor number, `n`, whereas servicing a floor (the elevator opens the door), *i.e.,* when a received event is of the form `{serve,n}`, resets `req` to `nil` indicating that no other request is being serviced; note that the variable is only modified if it is `nil`.

This code snippet is not programmed defensively; if *event* does not match either of the alternatives, the code crashes. During correct system behaviour, no other events should be expected and therefore we should not provide a catch-all clause which resolves the issue. Nevertheless, the elevator program is supervised by a designated process, as in Figure 2, so that if the elevator system crashes, it is restarted so as to avoid having people trapped inside the elevator.

An alternative, albeit more intricate, elevator implementation is one that can handle multiple requests at a time. The respective code snippet is shown below:

```
case event of
  {press,n}
     -> [n | List],
  {serve,n}
     -> lists:delete(n,List)
end;
```

It maintains a list of all floors which are awaiting service, adding a floor upon a key press and removing a floor upon servicing. However, the enhanced functionality increases the potential for subtle bugs that are hard to detect before deployment. For instance this new code does not check whether the list already contains the floor number being added, which would result in the multiple servicing of a floor (one for each key press). This implementation would thus violate a correctness criteria stating that "*multiple requests for the same floor should only be serviced once*"; for an elevator system with two floors, the respective property checker can be expressed as the state automaton in Figure 4.

Our methodology exposes this incorrect behaviour by executing the elevator program in parallel with a synthesised version of the property described in Figure 4 in an arrangement akin to Figure 2. This forces program termination upon a violation detection which, in turn, allows the respective supervisor to take the necessary corrective measures. For instance, once the more elaborate elevator controller terminates abnormally, the supervisor can spawn the tried-and-tested simple elevator controller (servicing one user at a time) as the *limp-home* code, while the new controller is being fixed. This is shown in Figure 4.

We have preliminary implementations manifesting this methodology using a tool called ELARVA [5], which synthesises automata-based property specifications, described in terms of a LARVA script, into an Erlang verifier monitor. Behaviour analysis can carried out through the in-built tracing mechanism provided by the Erlang Virtual Machine, which sends messages to the verifier's mailbox containing the events of interest carried out by the program, as specified by the property.

5. CONCLUDING REMARKS AND FUTURE WORK

In Erlang, where scalability, reliability and maintainability are vital, the fail-fast approach is the order of the day. Unfortunately, specification violations do not always lead to failure. By incorporating a specification-synthesised monitor, such violations can automatically be detected and transformed into failures which are in turn handled by existing mechanisms in Erlang.

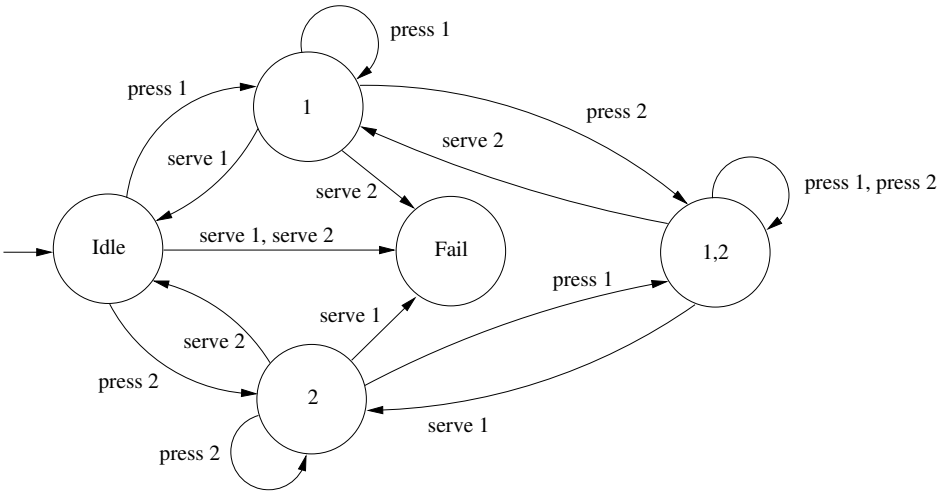

Figure 3: The elevator property shown as a state automaton.

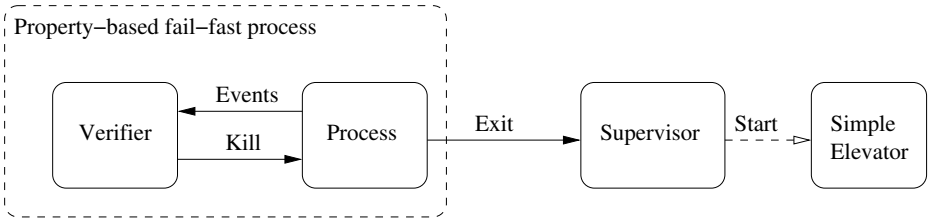

Figure 4: Updated elevator case study configuration.

The form of monitoring employed by our methodology thus far is often termed as *asynchronous monitoring*[3], which is renowned for being unobtrusive [2]. At the same time, it is often found to be inadequate for taking corrective measures upon violation detection since it may be difficult to determine the extent of the error's effects when violations are detected late. Nevertheless, in the case of Erlang, this problem is alleviated since each process has its own local memory. Furthermore, interdependencies among processes (as a result of message passing) can be explicitly identified through process linking so that if one process is terminated abnormally, the others get either notified or else terminated. Thus, although delayed violation detection might still allow unwanted actions to occur after an error, our approach promises a *best-effort* fail-fast architecture.

We have so far shown how to synthesise a verifier from the specification where the only corrective action allowed is the forced termination of the violating program. As a next step, we intend to extend this work by allowing violation reparations to be specified as part of the correctness specification, possibly allowing supervisors to be synthesised from the specification as well. Another future direction is that of automating the linking and exit-trapping settings associated with our monitoring setup, so as to ensure that all dependent processes are killed when a process has violated a specification.

[3]Assuming the asynchronous monitor is executed on (parallel) auxiliary resources, it becomes equivalent to offline monitoring.

6. REFERENCES

[1] J. Armstrong. *Programming Erlang - Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[2] H. Barringer, A. Groce, K. Havelund, and M. Smith. An entry point for formal methods: Specification and analysis of event logs. In *Formal Methods in Aerospace (FMA)*, 2009.

[3] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly, 2009.

[4] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Workshop on Runtime Verification (RV'03)*, volume 89(2) of *ENTCS*, pages 108 – 127, 2003.

[5] C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for erlang. In *RV: International Conference on Runtime Verification*, pages 370–374. Springer, 2011.

[6] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245. Morgan Kaufmann, 1973.

[7] M. Logan, E. Merritt, and R. Carlsson. *Erlang and OTP in Action*. Manning, 2010.