ELARVA: A Monitoring tool for Erlang

Christian Colombo, Adrian Francalanza, and Rudolph Gatt

Department of Computer Science, University of Malta, Malta.

Abstract. The LARVA monitoring tool has been successfully applied to a number of industrial Java systems, providing extra assurance of behaviour correctness. Given the increased interest in concurrent programming, we propose ELARVA, an adaptation of LARVA for monitoring programs written in Erlang, an established industry-strength concurrent language. Object-oriented LARVA constructs have been translated to process-oriented setting, and the synchronous LARVA monitoring semantics was altered to an asynchronous interpretation. We argue how this loosely-coupled runtime verification architecture still permits monitors to actuate recovery actions.

1 Introduction

Ensuring correctness in highly concurrent systems, through either testing or model checking, is problematic because it is difficult to test for all possible behaviour interleavings. A case in point is code written in Erlang [1], an established industry-strength functional concurrent language used mainly in the Telecoms industry. Ensuring correctness in this language is made even harder by the fact that: (i) Erlang is not statically type-checked, preventing developers from filtering out certain errors at compile time; and (ii) Erlang supports hot-code swapping *i.e.,* modules can be replaced on-the-fly, increasing the set of possible outcomes of a system execution.

Runtime Verification (RV) is a promising approach towards ensuring Erlang software correctness as it provides a disciplined methodology for conducting the runtime checks necessary in the absence of static guarantees. Importantly, the approach does not suffer from coverage and state explosion issues associated with standard verification techniques for concurrency.

LARVA is a runtime monitoring tool targeting the correctness of Java code [2] enabling one to: (i) specify system properties with recovery actions (in case of violation) in terms of automata-based specifications, (ii) compile the properties into Java monitors, and (iii) instrument the monitors at byte-code level using techniques from Aspect-Oriented Programming (AOP) [5]. Through aspects, synthesised monitors are then automatically updated with events from the execution of the monitored system, triggering corrective actions where necessary, providing extra reassurance as to the correctness of the monitored software behaviour.

LARVA supports modular property specification in a number of ways. For instance (i) each object can be verified by a separate monitor through a mechanism of monitor parametrisation, and (ii) properties can be decomposed into sub-properties that can communicate with one another through channels. Erlang's actor-based concurrency

model [4], which circumvents any shared memory through the use of message passing, is consistent with such a modular approach, making LARVA a sensible starting point for a monitoring framework for Erlang.
Porting LARVA to Erlang is however non-trivial, because: (i) Erlang does not have AOP support, the mechanism used by LARVA for monitor instrumentation; and (ii) Erlang is process-oriented whereas Java is object-oriented. In the rest of this proposal, we present ELARVA, an adaptation of LARVA to Erlang, giving an overview of how we tackled these issues and outline how we have evaluated our tool.

2 Solution Overview

In the absence of any AOP support, ELARVA employs Erlang's tracing facility for instrumentation; this makes monitoring asynchronous, which alters the nature of recovery actions. Moreover, adapting monitor parameterising constructs such as `foreach` to processes accentuated a shortcoming in LARVA's broadcast interpretation of channel communication, the extensive use of which made inter-monitor communication unwieldy; in ELARVA, channel communication was thus given a point-to-point interpretation.

2.1 Eliciting Events Asynchronously

Erlang's tracing mechanism enables us to hook on to Erlang's VM and receive the relevant events as messages to a *singleton* tracer process [1]. Monitors are set up as processes executing in parallel with the monitored system, where the tracer acts as a demultiplexer, reading the trace received and sending parts of it to the relevant monitors in non-blocking fashion, as shown in Fig. 1 (left). This setup has a number of advantages: (i) the system and the monitor can be running on separate machines, reducing the monitoring cost to that of tracing (ii) as opposed to LARVA, we can monitor a live system via ELARVA without having to stop the system and trace-compile it (iii) no errors are introduced in the monitored system as a result of instrumentation.
However, the non-blocking nature of Erlang message passing makes ELARVA's trace-based monitoring *asynchronous*, possibly detecting violations late. In general, this complicates a monitor's assessment of which sub-systems where effected by the violation. However, in the case of Erlang, adverse effects emanating from a violation can be confined since (i) processes do not share memories (ii) code is typically written in fail-fast fashion *i.e.*, processes fail as soon as anything abnormal is encountered and (iii) process dependencies can be explicitly delineated through mechanisms such as process linking [1], which propagates the failure to linked processes. Erlang process failure detection then allows the monitor localise the affected sub-system and take appropriate action. In fact, Erlang programs successfully achieve fault tolerance using these same mechanisms, through code patterns such as the Supervisor behaviour [1].

2.2 Parametrised Properties and Channel Communication

In ELARVA, the `foreach` construct was adapted to be parameterised by processes (as opposed to objects), so that a separate monitor could be replicated for every process

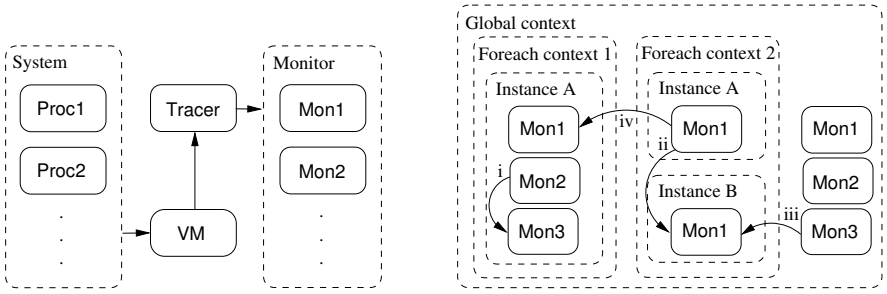

Fig. 1. Monitoring architecture in ELARVA *(left)*, Possible communication configurations *(right)*.

spawned. Modularly decomposing and replicating monitors in this way simplifies specifications since each monitor can focus on one process instance only, communicating with other monitors whenever necessary, as opposed to having one monolithic monitor monitoring multiple processes.

This specification approach is also in line with Erlang code practices, which advocate for the structuring of programs into as many small shortlived processes as possible.[1] However, it quickly became apparent that the existing LARVA communication mechanism, based on broadcasts, created bottlenecks in settings with extensive use of monitor decomposition *i.e.,* smaller communicating sub-monitors that are replicated on a per-process basis. As depicted in Fig. 1 (right), this was rectified in ELARVA by allowing monitors to select the destination of their communication from the following: (i) across monitors of the same instance, (ii) across monitors of different instances, (iii) from global to `foreach` context, and (iv) across `foreach` contexts.

3 Case Study

We consider a hospital management system where patients can place requests for medical reports and medical reports are issued once all the doctors concerned give their approval; note that patient requests are handled concurrently. Each patient and each doctor are modelled as a process and interact with a "main office" central process we refer to as the "hub" for short, see Fig. 2 (left).

Despite its simplicity, a number of correctness properties can be identified over this system such as: (i) A patient receives a report only if a request has been placed earlier, (ii) A patient never receives the medical report of another patient, (iii) A report received by a patient must be approved by at least two doctors overseeing that patient.

In what follows we give an intuition of how we monitor the third property outlined for this hospital system. A monitor is defined *foreach* patient process and *foreach* doctor process, depicted by dotted boxes in Fig. 2 (right). The following are the steps involved in a medical report request/response, lead by the labelled solid edges in Fig. 2 (right): (i) the patient, Pat1, requests a report, and the patient monitor, MP1 detects it (through

[1] Often referred to as concurrency-oriented programming, this allows the virtual machine to better apportion computation amongst the multiple processing units on a multicore machine [1].

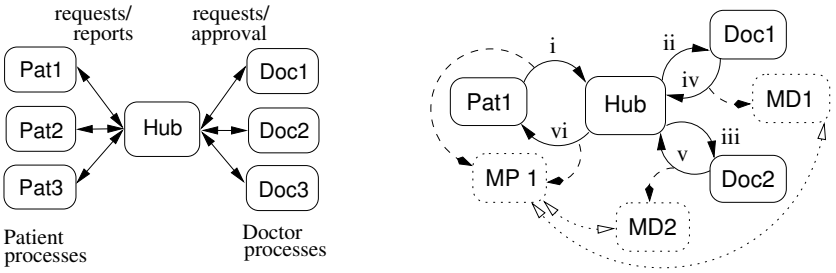

Fig. 2. The hospital management system setup *(left)*, with monitors *(right)*.

tracing, denoted by a *dashed edge*), (ii/iii) the hub forwards the report request to the doctors, Doc1 and Doc2 (no monitor activity), (iv/v) the doctors reply to the request by either approving or rejecting the report and the respective doctor monitors, MD1 and MD2, detect this (denoted by the respective dashed edge), (vi) the patient receives the report and the patient monitor detects it (see respective dashed edge). At this point, the patient monitor communicates with the doctor monitors (denoted by the respective *dotted edges*) to ensure that the report received was indeed authorised by the respective doctors.[2] If no two approving doctors are found then the monitor can conclude that the property was violated and an action is taken, possibly restarting the patient with the unauthorised report.

The monitors depicted in Fig. 2 can be setup through the following larva scripts specifying monitor automata, where transitions (backslash-separated triples consisting of an event detection, a condition, and an action to be executed if the condition is satisfied) take the property from one state to the next. Using the following event definitions:

```
EVENTS { RecRep = RECEIVE {backend_response,{Pat,Id,Diagnosis}}
         Ack    = CHANNEL {doc_response,{Pat,Id,Res}}
         AskAck = CHANNEL {mon_commm,{Doc,Id,Time}}
         RepRes = SEND {doc_response,{Doc,Res,Pat,Id}} }
```

For each patient process we need to detect report receipt, communicate with doctor monitors to confirm the approval of the report, receive an acknowledgement from the doctor monitors (indicating approval (yes), rejection (no) or indifference (ok)), and checks whether a violation has occurred. The *FOREACH* construct below automatically applies the monitoring logic to each patient joining the hospital system: upon the receipt of a report (*RecRep* event), the patient monitor sends a channel communication to all doctors, indicating the identifier and the timestamp of the report involved; the patient monitor then listens for *Ack* events and decides whether a violation has occurred.

```
FOREACH {patient,newPatient,[_]} { TRANSITIONS {
  start-> wait [RecRep\\cnt=numDocs,cntYes=0,{channel,{foreach,doctor,{'AskAck',{_,Id,Time}}}}]
  wait -> wait [Ack\Res=="ok" and cnt>1\cnt--]
  wait -> wait [Ack\Res=="yes" and cntYes<1\cntYes++]
  wait -> ok   [Ack\Res=="yes" and cntYes>0\] } }
  wait -> violation [Ack\Res=="no" or (Res=="ok" and cnt==1)\] } }
```

[2] Since monitoring is asynchronous, detection for either (iv/v) may not have happened by the time the doctor monitors receive the patient monitor communications *i.e.,* a race condition. Hence when a patient monitor detects the received patient report, it communicates with the relevant doctor monitors to ensure that the report has been approved by at least two of the doctors. Before the doctor monitors reply to the patient monitors, they are forced to make the necessary trace detections, thus reaching a synchronisation point with the patient monitor.

Below, the *FOREACH* construct allows us to dynamically launch a doctor monitor for every doctor joining the system, so as to detect report response events and communicate them to the corresponding patient monitors. The doctor monitor goes through all events which occurred before the report timestamp. Upon detecting a response (*RepRes* event) or the lack of it, the doctor monitor replies to the specific patient monitor (*Pat*) that requested the information.

```
FOREACH {doctor,newDoctor,[_]} { TRANSITIONS {
  idle -> detect [AskAck\\]
  detect -> detect [Event\eventTime<Time\]
  detect -> idle [Event\eventTime>Time\{channel,{foreach,patient,{'Ack',{Pat,Id,"ok"}}}}]
  detect -> idle [RepRes\\{channel,{foreach,patient,{'Ack',{Pat,Id,Res}}}}] } }
```

4 Evaluation

ELARVA was compared to Exago [3], an offline property-based Erlang monitoring tool. Both were successfully used to specify correctness properties of the hospital management system (introduced in the previous section). However, Exago necessitated the inclusion of substantial Erlang code-chunks; this blurred the distinction between code and specification logic and introduced the possibility of inserting further errors through the code chunks. By contrast, ELARVA was able to specify the properties using the tool logic (called DATES), the translation of which was automated by the monitor compiler. Another disadvantage of Exago was that it is an offline tool, which exclude the possibilities of applying reparatory actions in case of violations.

5 Conclusion

Through ELARVA, we have extended the LARVA tool and provided a minimally intrusive runtime monitoring framework to monitor expressive properties on Erlang code, whereby the limited intrusiveness makes it more palatable to potential adopters from the Erlang community. We aim to improve ELARVA by (i) investigating means of eliciting system events in a decentralised fashion, unlike the present centralised tracing mechanism relying on the Erlang VM and (ii) supporting distribution, so as to enable monitoring across machines.

References

1. J. Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, 2007.
2. C. Colombo, G. J. Pace, and G. Schneider. Larva — Safer Monitoring of Real-Time Java Programs (tool paper). In *SEFM*, pages 33–37. IEEE, November 2009.
3. A. Erdödi. Exago: Property monitoring via log file analysis. (Presented at the Erlang User Group Meeting, London, 2010).
4. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245. Morgan Kaufmann, 1973.
5. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.