Kalpa Publications in Computing
Volume XXX, 2017, Pages 1–7
RV-CuBES 2017. An International Workshop on
Competitions, Usability, Benchmarks, Evaluation,
and Standardisation for Runtime Verification Tools

A Suite of Monitoring Tools for Erlang *

Ian Cassar[12†], Adrian Francalanza[1], Duncan Paul Attard[12], Luca Aceto[32], and Anna Ingólfsdóttir[2]

[1] University of Malta, Department of Computer Science, Malta
{ian.cassar.10,adrian.francalanza,duncan.attard.01}@um.edu.mt
[2] Reykjavík University, School of Computer Science, Iceland
{ianc,duncanpa17,luca,annai}@ru.is
[3] Gran Sasso Science Institute, L'Aquila, Italy

Abstract

Ensuring formal correctness for actor-based, concurrent systems is a difficult task, primarily because exhaustive, static analysis verification techniques such as model checking quickly run into state-explosion problems. Runtime monitoring techniques such as Runtime Verification and Adaptation circumvent this limitation by verifying the correctness of a program by dynamically analysing its executions. This paper gives an overview of a suite of monitoring tools available for verifying and adapting actor-based Erlang programs.

1 Introduction

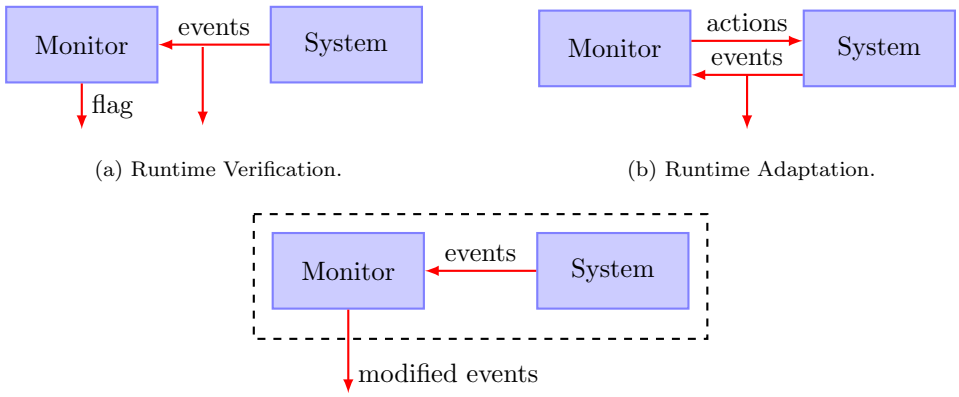
Runtime Monitoring is a lightweight dynamic verification technique in which the correctness of a program is assessed by only analysing a single execution, normally the currently executing one. In most monitoring settings [6, 17, 8, 18] the correctness property is generally specified as a formula in a logic with precise formal semantics, from which a *monitor* is then automatically synthesised. This monitor is essentially the executable software which analyses the runtime execution of a program in relation to the given property. Runtime monitoring constitutes the basis of several other techniques including Runtime Verification, Adaptation and Enforcement.

In Runtime verification (RV) [24, 17] monitors adopt a *passive* role [6, 4] and are exclusively concerned with receiving system events, analysing them, and *detecting* (flagging) violations (or satisfactions) of their respective correctness properties; this is illustrated in Figure 1a. Hence, RV monitors refrain from directly modifying the system's behaviour in any way.

By contrast, monitors in Runtime Adaptation (RA) [9, 22, 8, 21] break this passivity by executing adaptation actions after analysing a particular sequence of system events. As shown in Figure 1b, rather than flagging violations, RA monitors can execute adaptation actions upon detecting an event sequence that denotes incorrect behaviour. The adaptation actions executed

*The work presented in this paper was partly supported by the project "TheoFoMon: Theoretical Foundations for Monitorability" (grant number: 163406-051) of the Icelandic Research Fund.

†The research work disclosed in this publication is partially funded by the ENDEAVOUR Scholarships Scheme. "The scholarship may be part-financed by the European Union — European Social Fund"



(a) Runtime Verification.

(b) Runtime Adaptation.

(c) Runtime Enforcement.

Figure 1: Distinguishing between Runtime Verification, Adaptation and Enforcement

by the RA monitor do not necessarily correct or revert the detected misbehaviour [27, 22]; instead they attempt to mitigate its effect by changing certain aspects of the system as it executes, with the aim of preventing either future occurrences of the same error, or of other errors that may potentially occur as a side-effect of the detected violation. RA may also be used to optimise [1, 22] the system's behaviour based on the information collected by the monitor, *e.g.,* switching off redundant processes when under a small load, or increasing processes and load balancing when under a heavy load.

In Runtime Enforcement (RE) [16, 25, 26] the system behaviour is kept in line with the correctness requirement by anticipating incorrect behaviour and countering it before it actually happens. In RE the monitor is typically designed to act as a *proxy* which wraps around the system and analyses its external interactions (see the dotted-line in Figure 1c). The monitor is thus able to either drop incorrect events generated by the system, or add system events by executing actions on behalf of the system [25, 26]. This contrasts with runtime adaptation, where monitors may allow violations to occur but then execute remedial actions to mitigate the effects of the violation.

Erlang [3, 13] is a functional programming language that implements the *actor model* [2]. Actors are concurrent entities that execute independently and interact with each other through asynchronous message passing. Each actor in a system can be addressed via its unique identifier. Verifying formal correctness for actor-based, concurrent systems is a difficult task, primarily because exhaustive, static analysis verification techniques such as model checking quickly run into state-explosion problems, typically due to the multiple thread interleavings of the system being analysed, and to the open nature of reactive programs that input values dynamically – runtime monitoring is therefore an ideal alternative.

Even though several monitoring tools [14, 23, 15, 20] exists for renowned languages such as Java, the only tools developed specifically for Erlang are detectEr [4, 7] and adaptEr [9, 8]. In this paper we therefore give an overview of these two runtime monitor tools.

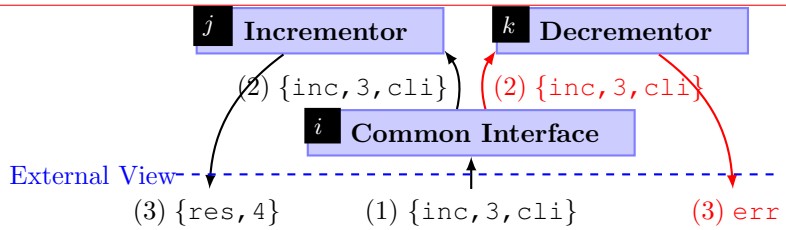


Figure 2: A server actor implementation offering integer increment and decrement services

2 Runtime Verification with detectEr

detectEr [4, 5] is a runtime verification tool that converts properties, expressed using a fragment of Hennessey Milner Logic with recursion (μHML) [18], into *detection monitors* capable of producing verdicts stating whether an Erlang program satisfies or violates a given property. The detectEr RV tool [4, 5] implements the theory explored in [19, 18] which formally defines the runtime behaviour of concurrent RV monitors in terms of an LTS semantics. The authors also identified a maximally expressive monitorable subset of μHML, coined as mHML. This subset allows one to define either safety, or co-safety properties.

Example 2.1. Consider a simple Erlang system consisting of 3 actors as depicted in Figure 2. This server system consists of a front-end *Common Interface* actor with identifier i receiving client requests, a back-end *Incrementor* actor with identifier j, handling integer increment requests, and a back-end *Decrementor* actor k, handing decrement requests. A client sends service requests of the form {*tag*,*arg*,*ret*} to actor i, where *tag* selects the type of service, *arg* carries the service arguments and *ret* specifies the return address for the result (typically the client actor ID). The interface actor forwards the request to one of its back-end servers (depending on the *tag*) whereas the back-end servers process the requests, sending results (or error messages) to *ret*. The tool detectEr allows us to specify mHML properties such as (1), explained below:

```
max Y.[i?{inc,x,y}](([j>y!{res,x+1}]Y) & ([_>y!err]ff))        (1)
```

It is a recursive property (obtained using the maximal fixpoint `max Y....`) requiring that, from an *external* viewpoint, *every* increment request received by actor i, action `i?{inc,x,y}`, is followed by an answer from actor j to the address y carrying $x+1$, action `j>y!{res,x+1}` (recursing through variable `Y`). Note how the address/ID of the recipient of the output action y is learnt dynamically as a freeze variable from the preceding input action. However, increment requests followed by an error message sent from *any* actor back to y, action `_>y!err`, represent a violation (expressed as a falsity, `ff`). detectEr can synthesise a concurrent monitor (consisting of a system of actors) corresponding to (1) and instrument it with an Erlang system [4]. ■

The original implementation for detectEr implemented this theory by synthesising *completely-asynchronous (CA)* [11] monitors that were capable of observing Erlang systems by using the asynchronous tracing mechanism [3] provided by the Erlang Virtual Machine. Although completely-asynchronous monitoring is generally very efficient, it tends to suffer from *late detection*, *i.e.,* a misbehaving actor may be able to execute other actions before the asynchronous monitor detects the misbehaviour. Timely detections are crucial especially when monitoring for safety-critical properties which may require immediate system reparation when violated; achieving this required introducing synchrony in an efficient manner.

Further work on the tool, therefore, included extensions that allow the specifier to select between different monitor instrumentation techniques, namely, *completely-asynchronous monitoring (CA)*, *synchronous instrumentation monitoring (SMSI)*, *asynchronous monitoring with checkpoints (AMC)* and *synchronous detection monitoring (AMSD)* (as defined in [11]) – this extended version is known as DetectEr 2.0 [7]. To implement this variety of monitoring modalities, the tracing mechanism was replaced by code instrumentation that was achieved through an aspect-oriented programming framework for Erlang called eAOP [10]. eAOP[1] allowed for instrumenting the system with a custom tracing protocol that, apart from reporting events to the monitor as asynchronous messages, it is also able to force certain system components to block waiting for the monitor's feedback, thereby achieving synchrony.

CA in DetectEr 2.0 is implemented by completely omitting the requirement for concurrent system components to wait for the monitor's feedback. By contrast, SMSI is achieved by forcing each system component to block after *every* reported event. AMC and AMSD (referred to as *hybrid* in DetectEr 2.0) are achieved through an extension to the specification language that introduces *synchronous necessities* and *synchronous verdicts*. Synchronous necessities are used to force the instrumented component to wait for feedback whenever the event described in the necessity is reported to the monitor. AMC is thus achieved via synchronous necessities that serve as checkpoints that allow the lagging monitor to catch up and synchronise with the system. Synchronous verdicts are used for achieving AMSD, since they only force system components to synchronously report events that may lead to a violation.

In the following example we give an overview of how one can make slight alterations to the property given in Example 2.1 in order to apply different monitor instrumentation techniques.

Example 2.2. Unless specified otherwise, detectEr converts properties such as (1) into a completely-asynchronous monitor. By adding the directive `--{sync}` at the beginning of the RV script, the property is converted into a synchronous monitor which applies SMSI.

```
max Y.[|i?{inc,x,y}|]((([j>y!{res,x+1}] Y) & ([_>y!err]ff))        (2)
max Y.[i?{inc,x,y}](([j>y!{res,x+1}] Y) & ([_>y!err] sff))        (3)
```

Minor modifications to (1) are required to achieve AMC and AMSD. By synthesising property (2) we obtain a monitor implementing AMC by specifying `[|i?{inc,x,y}|]` instead of `[i?{inc,x,y}]`, where `[| − |]` denotes a checkpoint in which the monitor's execution synchronises with that of the monitored system. With (3) we obtain an AMSD monitor since we specify the synchronous violation verdict `sff` instead of `ff`. This ensures that the monitor synchronises with the system upon the occurrence of a violating event, thereby guaranteeing the timely detection of the violation. ■

3 Runtime Adaptation with adaptEr

The synchronisation protocol introduced in DetectEr 2.0 was further extended to allow for *adaptation actions* to be effectively applied to specific Erlang actors in a timely manner. Unlike detection monitors in detectEr, adaptation monitors do not just detect and flag violations, but are also capable of reacting to the detection by applying adaptation actions. Adaptation actions are rectifying actions (such as restarting or terminating misbehaving actors) in order to mitigate the effects incurred by a detected violation. This extension led to the creation of a Runtime Adaptation tool called adaptEr [8, 9].

[1] The eAOP framework is open-source and accessible from https://github.com/casian/eaop.

Example 3.1. The RA tool adaptEr *extends* properties such as (1) with *adaptation actions* to be taken by the monitor once a violation is detected, as shown in property (4).

```
max Y.[i?{inc,x,y}]>(
    ([j>y!{res,x+1}]rel(i).Y)   &
    ([z>y!err]> restart(i).flush(z).rel(i,z).Y)
)
```
(4)

In this property, the specifier presumes that the error (which may arise after a number of correct interactions) is caused by the interface actor i (as shown in Figure 2, where an `inc` request is erroneously forwarded to the decrementor actor k) — one may, for instance, have prior knowledge that actor i is a newly-installed, untested component. The monitor thus immediately blocks the execution of actor i (using the blocking derivative `[-]>`) and depending on whether the incrementor j produces a correctly incremented result (*i.e.,* `[j>y!{res,x+1}]`), the monitor releases the actor (using `rel(i).`), otherwise it restarts actor i using adaptation `restart(i).`.

Whenever an error occurs (*i.e.,* `[z>y!err]>`), the monitor also blocks any actor z that produces the error and then empties its mailbox —which may contain more erroneously forwarded messages—through adaptation `flush(z).` (the actor to be purged is determined at runtime, where z is bound to identifier k from the previous action `[z>y!err]>`). Importantly, note that in the above execution (where k is the actor sending the error message), actor j's execution is *not affected* by any adaptation action taken. After adapting the monitored system according to the detected erroneous behaviour using `restart(i).` and `flush(z).`, the monitor also releases the blocked actors allowing the adapted actors to proceed their execution. Blocking an actor before adapting it is crucial to ensure timely mitigation. This guarantees that errors do not propagate and thus prevents other consequent errors from occurring.

4 Conclusion

In this paper we have discussed detectEr[2], a runtime verification tool designed for detecting the violation or satisfaction of monitorable μHML properties while monitoring an Erlang program using a variety of monitoring instrumentation techniques. We also gave an overview of how this tool has been evolved into a runtime adaptation tool called adaptEr[3] which synthesises monitors capable of applying mitigating actions in order to rectify a detected misbehaviour.

Ongoing and Future Work. We are currently in the process of adding support for synthesising distributed monitors with detectEr, that are capable of migrating from one node to another depending on the load on the current node. We are also working on introducing the adaptation functionality provided by adaptEr in the setting of session types; further details can be found in [12]. Moreover, we are developing enforcEr, a novel runtime enforcement tool capable of suppressing and inserting Erlang messages that are exchanged amongst a number of actors.

References

[1] An architectural blueprint for autonomic computing. Technical report, IBM, 2005.

[2]Both detectEr and DetectEr 2.0 are open source and downloadable from https://bitbucket.org/duncanatt/detecter-lite and https://bitbucket.org/casian/detecter2.0 *resp.*

[3]The tool adaptEr is also open source and available from https://bitbucket.org/casian/adapter.

[2] Gul A. Agha, Prasannaa Thati, and Reza Ziaei. Actors: A model for reasoning about open distributed systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing - An Object Oriented Approach*, chapter 8, pages 155–176. Cambridge University Press, New York, NY, USA, 2001.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. A runtime monitoring tool for actor-based systems. *Behavioural Types: from Theory to Tools.*, 2017.

[5] Duncan Paul Attard and Adrian Francalanza. *A Monitoring Tool for a Branching-Time Logic*, pages 473–481. Springer International Publishing, Cham, 2016.

[6] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

[7] Ian Cassar and Adrian Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. *arXiv:1502.03514*, 2015.

[8] Ian Cassar and Adrian Francalanza. Runtime adaptation for actor systems. In *RV*, pages 38–54. Springer, 2015.

[9] Ian Cassar and Adrian Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *IFM*, pages 176–192. Springer, 2016.

[10] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. eAOP: An Aspect Oriented Programming Framework for Erlang. In *Erlang*, ACM SIGPLAN, 2017.

[11] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In *PrePost2017*, pages 15–28, 2017.

[12] Ian Cassar, Adrian Francalanza, Claudio Antares Mezzina, and Emilio Tuosto. Reliability and fault-tolerance by choreographic design. In *PrePost2017*, pages 69–80, 2017.

[13] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[14] Feng Chen and Grigore Roşu. *Java-MOP: A Monitoring Oriented Programming Environment for Java*, pages 546–550. 2005.

[15] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded java. In *ECOOP*, pages 546–569. Springer, 2009.

[16] Ylis Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223, June 2011.

[17] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In *RV*, pages 8–29, 2017.

[18] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *RV*, pages 71–86, 2015.

[19] Adrian Francalanza and Aldrin Seychell. Synthesising correct concurrent runtime monitors. *FMSD*, 46(3):226–261, 2015.

[20] Klaus Havelund and Grigore Roşu. An Overview of the RV tool Java PathExplorer. *FMSD*, 24(2):189–215, 2004.

[21] Gabriela Jacques-Silva, Buğra Gedik, Rohit Wagle, Kun-Lung Wu, and Vibhore Kumar. Building user-defined runtime adaptation routines for stream processing applications. *Proc. VLDB Endow.*, 5(12):1826–1837, August 2012.

[22] Stephen Kell. A survey of practical software adaptation techniques. *J.UCS*, 14(13):2110–2157, 2008.

[23] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *FMSD*, 24(2):129–155, 2004.

[24] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of*

Logic and Algebraic Programming, 78(5):293–303, 2009.

[25] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *IJIS*, 4(1):2–16, 2005.

[26] Jay Ligatti and Srikar Reddy. *A Theory of Runtime Enforcement, with Results*, pages 87–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[27] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *ICSE*, pages 80–90, Piscataway, NJ, USA, 2012. IEEE Press.