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 Generic Instrumentation Tool 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

Aspect oriented programming (AOP) is an instrumentation mechanism that is generally employed by runtime monitoring frameworks to automatically extract system events and embed monitoring code within an existing system. Although several AOP tools exist for renowned languages such as Java and C#, little to no tools have been developed for actor oriented languages such as Erlang. We present eAOP, an AOP tool specifically designed to instrument actor-oriented constructs in Erlang such as message sends and receives, along with other traditional constructs such as function calls.

1 Introduction

Code instrumentation techniques are often employed by software analysis tools for validating and verifying software systems. In the case of *runtime monitoring* [11, 15] instrumentation mechanisms such as *Aspect Oriented Programming* (AOP) [13, 14] are extensively used to allow a designated runtime analyser to observe system operations and interactions, in order to check this behaviour against some correctness property. It is common practice that runtime monitoring tools convert the given correctness properties into aspect code and then use an AOP framework to perform the required code instrumentation.

In general, AOP frameworks carry out instrumentation through a static analyser known as the *instrumenter*. The instrumenter takes as input a program and an aspect specification consisting of code patterns (*e.g.,* new object declarations, method calls, message sends, *etc.*) known as *pointcuts*, indicating the constructs that should be injected with aspect code known as *advices*. Using source code patterns, a pointcut defines a set of locations in the code where the instrumenter may inject appropriate advice code, thereby producing a possibly modified version of the input system. One can currently find several AOP frameworks that support renowned

*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"

OOP languages such as Java, however, little to no AOP frameworks have been developed specifically for *actor-oriented programming* languages such as Erlang [2].

In this paper we thus present eAOP [7], an AOP framework that is specifically designed for instrumenting Erlang code. Section 2 overviews Erlang and actors, while Section 3 introduces the various types of pointcuts and advices that are supported by our tool. Section 4 discusses how our tool has been used for runtime monitoring purposes. Section 5 concludes with a summary of our contributions and future work.

2 Erlang

Erlang implements the actor model [1]: its main functional components are *actors* – concurrent entities that are uniquely identifiable through unique identifiers (IDs) and interact via asynchronous message passing. Erlang also implements actor-based operations as *first class constructs*. For instance, message send operations are expressed using the dedicated syntax `Pid!Msg`, where `Pid` is the unique identifier (ID) of the destination actor of the message, while `Msg` denotes the message payload that may contain data such as numbers, atoms, strings, tuples, lists and also actor IDs. Message receive operations are defined as a list of *guarded statements*, where guards consist of data structure patterns:

```
receive guard1 -> statement1; ... ; guardN -> statementN end
```

Messages residing in the mailbox of an actor are pattern matched (in order) with every guard, and the continuation statement of the *first guard which matches with the mailbox message* is executed; if none match, the messages remain in the mailbox and the actor blocks (until a new message is received).

Erlang functions can be invoked as `M:F(Args)` where `M` stands for the module in which function `F` is defined, while `Args` represents a list of arguments required by function `F`. In Erlang, the notation `M:F/A` (where `A` is the function arity) uniquely identifies a function.

3 Pointcuts and Advices

Our instrumentation tool, eAOP [7], requires the code of the Erlang system along with the required pointcuts and advices in order to instrument the necessary aspects, as shown in Figure 1. Pointcuts must be defined in a text file suffixed with a `.eaop` extension, and must be provided to the eAOP instrumenter along with the Erlang system. Given these files, the instrumenter produces a version of the system that is instrumented with function calls to advices. Advices in eAOP must be defined as a set of functions in a module called `advices.erl`. Once defined, `advices.erl` must be manually compiled into a `.beam` file using the Erlang compiler. The instrumented system thus amounts to the modified system along with `advices.beam`, as shown in Figure 1. The reader should consult [7] for more detail.

Pointcuts for eAOP *Pointcuts* allow the user to specify code patterns that are consulted by the instrumenter while conducting a depth first search traversal of the syntax tree of the parsed program. During this traversal, whenever a node in the syntax tree matches a pointcut definition, the appropriate advices are embedded into the syntax tree. Pointcuts are defined using the following format:

```
Pointcut(<CType>, <MP>, <FP>, <CP>, <ATypes>)
```

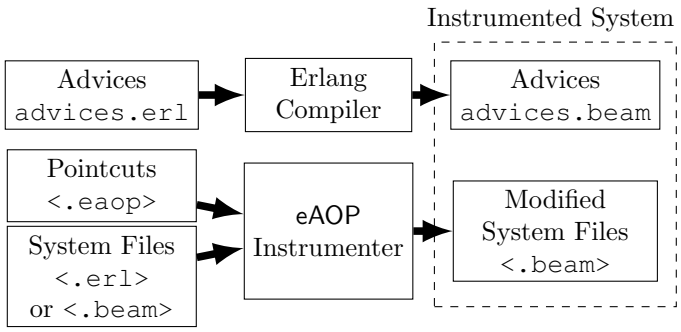


Figure 1: The components of the eAOP instrumentation tool.

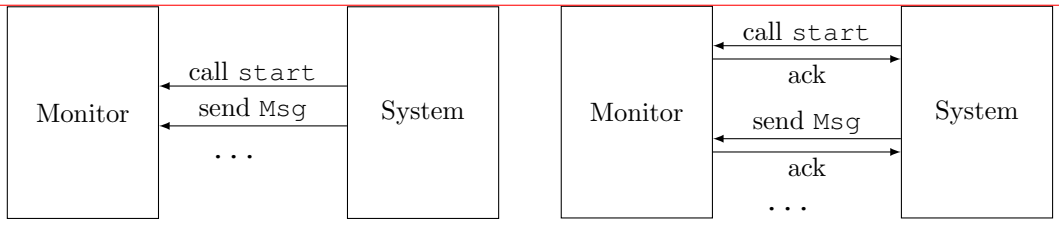
The <CType> specifies the *construct type*, *i.e.,* the *type* of code operation (*e.g.,* function call, message send, *etc.*) that the instrumenter must look for while analysing the given program. The module pattern, <MP>, and function pattern, <FP>, determine the module and function in which the instrumenter must look for the required code operations, while the construct pattern, <CP>, is used to describe the construct that requires instrumentation (where <CP> varies depending on <CType>). eAOP can handle the following *four* construct types:

- **`send`, `receive`** and **`call`:** With these types the instrumenter *resp.* searches for *send operations* (`Pid!Msg`), *receive blocks* (`receive` [...] `end`), and *function calls* (`mod:fun(Args)`) that are defined in modules and functions whose name pattern matches <MP> and <FP> *resp.* Since <CP> varies according to the <CType>, if for instance <CType> is `send`, pattern <CP> must specify a *message pattern*, thereby restricting the instrumenter to instrument only send operations that send messages matching this pattern.

- **`function`:** This type is used to instrument *entire function definitions* (not function calls) that reside in modules matching <MP>, whose function name and arity match patterns <FP> and <CP> *resp.* This construct type is used to mimic method overriding typical of OOP, whereby the entire function body is *replaced* by another instrumented function.

Advices for eAOP Different pointcuts inject different advices based on the advice types defined in <ATypes>. Advice types are defined as a *list of directives* indicating which advices the instrumenter should inject upon matching a syntax tree node with a pointcut definition. Advice types dictate how the instrumenter should instrument the matching code fragment by specifying which *advice function calls* should be injected. The instrumenter can only inject function calls to five advices functions, namely: `before_advice/5`, `after_advice/5`, `intercept_advice/5`, `upon_advice/5`, and `override_advice/5`. These advice functions must be manually defined to include the code that needs to be executed whenever a specific instrumented code construct runs.

In total, eAOP provides *five* advice types that can be used to specify which of the aforementioned advice function calls should be injected by the instrumenter. Primarily, our tool supports *three augmentative* advice types, namely `before`, `after` and `upon`, which instruct the instrumenter to *resp.* augment a function call to `before_advice` *before* executing the respective action, a function call to `after_advice` *after* the instrumented action, and a call to `upon_advice` *after each matching receive guard*. eAOP also supports *two refactoring* advice types, namely, `intercept` in which the instrumenter *replaces* the specified action with a call



(a) Asynchronous (b) Synchronous

Figure 2: Instrumentation for Asynchronous and Synchronous Monitoring

to `intercept_advice`, and `override` where the entire function body of a matching function definition is *replaced* by a call to `override_advice`.

As depicted in Figure 1, the injected advice functions must be defined in the `advices.erl` module which must be created inside the *same directory* in which the instrumented `beam` files are set to be generated by the instrumenter. The advices follow a standard order for their arguments. At runtime, the advice arguments will contain the necessary data for *identifying the instrumented part of the code*. When defining advices, one must therefore inspect the data in these parameters (*e.g.,* using the Erlang `case` statement) and perform the necessary aspect logic for the respective cases (an example of how this can be done is given in Section 4).

System Instrumentation In order to instrument an Erlang system, one needs to provide the instrumenter with a `.eaop` text file containing the list of pointcuts, along with the system's source `.erl` files or its `.beam` files (compiled in debug mode).

To instrument an Erlang system with eAOP, one can invoke the instrumenter by calling `eaop:instrument(SrcDirs, ConfigDirs)`, where `SrcDirs` refers to a *list of directories* containing either the system `.erl` (source) files or `.beam` files that require instrumentation. `ConfigDirs` refers to a list of directories containing the `.eaop` pointcut specification files.

4 Applying eAOP for monitoring

In [4], we used eAOP as part of a toolchain making up a *runtime monitoring* framework called DetectEr 2.0[1]. This monitoring framework generates actors that monitor for a given specification, and instruments the system under scrutiny to enable it to report trace events to the generated monitoring actors.

Using the code instrumentation mechanism provided by eAOP, DetectEr 2.0 is able to implement multiple monitoring mechanisms. The instrumentation used by the tool varies depending on the chosen monitoring mechanism, namely *synchronous*, *asynchronous* or *hybrid* monitoring [8]. In synchronous monitoring, the system is forced to *wait* until the monitor finishes processing the reported trace event before it can proceed with its execution, while in asynchronous monitoring the system proceeds immediately. Hybrid monitoring constitutes a mixture of the two mechanisms. Although asynchronous monitoring was shown to be more efficient in [4], the inclusion of synchronous monitoring was crucial to allow for timely detections [8].

As shown in Figure 2a, to implement asynchronous monitoring with AOP, the instrumentation code is used to asynchronously deliver an event notification as a message to the monitor

[1]DetectEr 2.0 is open source and downloadable from https://bitbucket.org/casian/detecter2.0.

before the specified action executes. Implementing synchronous monitoring, however, requires the instrumentation of a *handshake protocol* as depicted in Figure 2b. In this case, the instrumented code sends the event notification and then blocks the system actor rather than resuming execution immediately; the blocked actor is thus forced to wait until it receives an acknowledgement message from the monitor signalling it to continue with its execution.

DetectEr 2.0 instruments this monitoring protocol by *automatically generating the required pointcuts and advices* from the given specification, and then uses eAOP to perform the required instrumentation. For instance, to synchronously monitor for send operations in which the message being sent has the form `{add,N1,N2}` and which are defined within a function `mod:foo(Args)`, the tool generates the following pointcut:

```
[Pointcut(send,"mod","foo","{add,_,_}",[before])].
```

along with the following advice function definition:

```
1 before_advice(Type,Sender,M,F,Payload) ->
2   case Payload of
3     [SentTo,Msg={add,_,_}] when Type==send and M==mod and F==foo ->
4       detecter!{trace,send,Sender,SentTo,Msg},
5       receive
6         ack -> ok
7       end,
8   end.
```

The generated pointcut instructs the instrumenter to inject a function call to `before_advice` *before* every send operation in which the message being sent matches the pattern `"{add,_,_}"`, *i.e.,* the message consists of a tuple containing the atom `add` along with two other elements. The instrumenter also generates the `advices.erl` module containing the implementation for `before_advice`. The generated advice definition immediately applies a case statement (lines 2 and 3) to inspect the contents of the payload so as to ensure that a trace message is only produced when the instrumented action is a `send` operation that is defined in `mod:foo` which is sending a message that matches `"{add,_,_}"`. In such case, the generated advice reports the operation by sending a trace message (line 4) to the monitoring process (registered as the designated name `detecter`) and then blocks via an injected `receive` waiting for an `ack` message (lines 5-7). In the case of asynchronous monitoring, the advice code omits the receive block thereby allowing the instrumented actor to proceed immediately.

eAOP was also embedded in a framework called AdaptEr[2] to automate *runtime adaptation* monitors for Erlang systems [5, 6]. AdaptEr was developed as an extension to DetectEr 2.0, and can selectively apply adaptation actions to specific actors so as to either *rectify* the effects of a detected misbehaviour (*e.g.,* restart a misbehaving actor) or else to improve the system based on the current state of the system (*e.g.,* by terminating idle/redundant actor processes).

Erlang's implementation of the actor model limits inter-process communication to asynchronous message passing. For instance, one of AdaptEr's adaptation actions allows the monitor to empty the mailbox contents of a system actor after this performs a specific sequence of actions. Since Erlang strictly forbids actors from directly modifying the mailbox contents of another actor, this adaptation had to be encoded using an instrumented protocol by means of which the monitor delivers the required adaptation using message passing.

As illustrated in Figure 3, the instrumented protocol builds on the synchronous monitoring protocol outlined in Figure 2b, by forcing the system actor to block after forwarding the trace event to the monitor. The blocked actor must then wait for either an acknowledgement message

[2] AdaptEr is open-source and downloadable from https://bitbucket.org/casian/adapter.

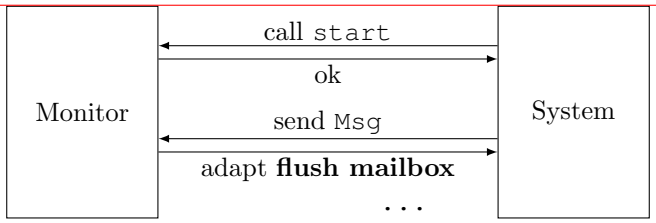


Figure 3: Instrumentation for Runtime Adaptation

from the monitor allowing it to resume execution, or else for an *adaptation message*. As shown in the code excerpt below (lines 5-10), upon receiving an adaptation message, the instrumentation code interprets the message and forces the actor to execute the requested adaptation action. Using our eAOP tool, this instrumented protocol permitted the monitor to deliver (intrusive) adaptation actions through message passing.

```
1  before_advice(Type,Sender,M,F,Payload) ->
2   case Payload of
3    [SentTo,Msg={add,N1,N2}] when Type==send and M==mod and F==foo ->
4     detecter!{trace,send,Sender,SentTo,Msg},
5     receive
6       ack -> ok;
7       flush -> adapter:flush();
8       restart -> adapter:restart();
9       ...
10    end,
11  end.
```

Both DetectEr 2.0 and AdaptEr were used to instrument and monitor industry-scale applications including Yaws and Ranch [4, 3]. Recent work [6] also showed how AdaptEr can be used to patch a mitigation for the *Directory Traversal* vulnerability that was found in Yaws 1.89 [12], that makes it vulnerable to *dot-dot-slash attacks*.

5 Conclusion

In this paper we have presented eAOP[3], an AOP framework for instrumenting Erlang programs. This tool was specifically designed to effectively instrument the actor-based functionality that is native to Erlang. eAOP proved to be an essential asset to allow for achieving synchronous monitoring [3, 4, 8] and Runtime Adaptation [5, 6] in Erlang programs.

As future work, we plan to design a more expressive language that allows for defining advices along with the pointcuts, *i.e.,* directly within the `.eaop` specification file. We also plan to include other advice types including types related to exception handling that allow for injecting code at specific *try-catch* cases.

Related Work Several AOP tools [14, 10, 9] exist for other more renowned programming languages. However, to our knowledge, ErlAop[4] is the only other AOP framework for Erlang

[3] eAOP is open-source and downloadable from https://github.com/casian/eaop.
[4] Accessible from http://erlaop.sourceforge.net/.

that exists apart from eAOP. ErlAop, however, lacked a number of important features that are essential for monitoring tools. For instance, ErlAop only permits the instrumentation of function calls, whereas eAOP permits the instrumentation of message sends and receives and is incapable of instrumenting compiled `.beam` files.

References

[1] Gul Agha. Actors: a Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence. *MIT Press*, 11(12):12, 1986.

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

[3] 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.

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

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

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

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

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

[9] Ismael Figueroa, Nicolas Tabareau, and Éric Tanter. *Effective Aspects: A Typed Monadic Embedding of Pointcuts and Advice*, pages 145–192. Springer, 2014.

[10] Gael Fraiteur. A Thread-Safe Extension to Object-Oriented Programming. Technical report, PostSharp Technologies.

[11] 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.

[12] Alejandro Hernandez. Yaws 1.89: Directory Traversal Vulnerability. Available online at `www.exploit-db.com/exploits/15371/`, 2010. Accessed on 24/5/2017.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*, pages 220–242. Springer, 1997.

[14] Ramnivas Laddad. *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press, 2003.

[15] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.