eAOP: An Aspect Oriented Programming Framework for Erlang[*]

Ian Cassar[†]
University of Malta and Reykjavik University
Malta and Iceland
ian.cassar.10@um.edu.mt

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

Luca Aceto
Reykjavik University
Iceland
luca@ru.is

Anna Ingólfsdóttir
Reykjavik University
Iceland
annai@ru.is

Abstract

Aspect oriented programming (AOP) is a paradigm ideal for defining cross-cutting concerns within an existing application. Although several AOP frameworks exist for more renowned languages such as Java and C#, little to no frameworks have been developed for actor oriented languages such as Erlang. We thus present eAOP, a new AOP framework specifically designed to instrument actor-oriented constructs in Erlang such as message sends and receives, along with other traditional constructs such as function calls.

CCS Concepts • **Software and its engineering → Frameworks**; **Preprocessors**; *Software design engineering*;

Keywords Erlang, Aspect Oriented Programming, Code instrumentation, Concurrency, Actor Systems

ACM Reference Format:
Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. eAOP: An Aspect Oriented Programming Framework for Erlang. In *Proceedings of 16th ACM SIGPLAN International Workshop on Erlang, Oxford, UK, September 8, 2017 (Erlang'17),* 11 pages.
https://doi.org/10.1145/3123569.3123570

1 Introduction

Aspect Oriented Programming (AOP) [27, 28, 30] is a programming paradigm that provides a clean separation between the *components* [28] and the *aspects* of a system. A component of a system represents a piece of logic that can be encapsulated within a *functional unit* of a programming language (*e.g.,* object, function, method, module *etc.*) in a manner that is well localized, understandable, well-composed and easily accessed. Components usually constitute the building blocks for constructing the functionality of a system, *e.g.,* objects and methods are components of an object oriented language like Java, while actor processes and functions are components of Actor-based languages like Erlang. By contrast, aspects are instances of code logic that is difficult to represent as a unit of the host language. Rather, an aspect signifies program logic that must be composed and coordinated across different components of a program—these properties are said to *cross-cut* [28, 30] different components. For instance, logging, error-handling, data validations, synchronisation of concurrent processes, and the security analysis of memory access patterns are typical examples of aspects that cross-cut system components.

An aspect oriented programming framework thus consists of the *component language* and the *aspect language*, where the former is used to define functional units, *i.e.,* components, while the latter allows for the specification of cross-cutting properties, *i.e.,* aspects. Component languages are usually general-purpose programming languages that are readily available, while aspect languages are developed separately for a specific component language and plugged into it through *code-instrumentation.* Various programming languages from disparate paradigms (*e.g.,* object-oriented, functional *etc.*) can serve as component languages; aspect languages then introduce aspects as supplementary, cross-cutting components that are merged with the designated components *e.g.,* an aspect in a procedural language is invoked whenever a specific procedure is about to be called.

One can find several aspect languages for programming languages that implement a specific programming paradigm. For instance, for object oriented languages like Java and C#, there are AOP frameworks such as AspectJ [24, 30] and SpringAOP [35] for the former, and PostSharp [21] for the latter. These frameworks generally hook aspects to object oriented constructs such as method invocations, object creations, exception handling, *etc.* For procedural languages such as C and C++, there are aspect languages such as AspectC [17] and AspectC++ [36], whereas for functional languages such as Haskell there are implementations like effective-aspects [20].

However, little to no AOP frameworks have been developed specifically for *actor-oriented programming* languages. Erlang [2] is one such programming language that implements the actor-model [1]. In an Erlang program, actors (*i.e.,* concurrent entities that communicate asynchronously through message passing) typically constitute one of the main component forms. Functions are another important component form in Erlang, since the language is also functional. We contend that an ideal AOP framework for Erlang should therefore allow for aspects to be associated with these actor-based components by instrumenting constructs such as message sends and receives, actor spawning and function calls.

[*]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"

AOP has several applications for a variety of diverse areas of software development and maintenance:

- It is used for achieving *structured code optimizations and refactoring* [28]. Optimising functionality sometimes requires exploiting information about the execution context that cross-cuts different components; this requires fusing components together in a way that might disrupt the structure of the original code. Similarly, AOP-based refactoring improves upon existing design patterns and discovers new ones [6, 9, 34] and has been used to implement patterns such as Inversion-of-Control [16, 19].
- Development methodologies such as Monitor Oriented Programming [12, 14, 15] use AOP to *augment functionality* within an existing system. Such methodologies permit systems to be developed incrementally and adapt to fluctuating user requirements by using an onion-layered approach, where the inner-most layer contains the core system functionality, while the outer layers contain secondary functionality such as user access control. In this case, AOP is used to automatically instrument each outer layer on top of the preceding inner layer.
- *Software analysis* may also resort to AOP for validating and verifying software systems. AOP is commonly used to instrument logging and tracing features for debugging purposes. In testing, AOP is also being explored for generating test cases automatically [37], for mocking, and also for assessing the adequacy and code coverage of a test suite [33]. It can also be used for fault injection so as to stress-test a system. In *runtime monitoring* [10, 12, 13, 15, 25], AOP techniques are used extensively to allow a designated runtime analyser to observe system operations and interactions, in order to check this behaviour against a correctness property.

In this paper we investigate whether existing AOP mechanisms are applicable to actor-based languages such as Erlang. We assess whether actor-based constructs constitute valid components that can be fruitfully used for aspect orientation—allowing actor languages to benefit from the aforementioned advantages—or whether these constructs pose any insurmountable challenges. We carry out this investigation by developing a prototype AOP framework called eAOP, specifically designed for instrumenting Erlang code. In what follows, Section 2 provides preliminary material for understanding the concepts behind eAOP, while in Section 3 we introduce the various types of pointcuts and advices that are supported by eAOP. In Section 4 we then explain how the instrumenter performs static analysis in order to inject the appropriate advices at the specified code constructs. In Section 5 we discuss how our framework has been integrated in runtime monitoring and adaptation tools which were used to instrument a number of industry-scale third-party software. Section 6 concludes with a summary of our contributions and a discussion of related and future work.

2 Preliminaries

Code instrumentation is an important technique used in several areas of computer science and software engineering, including test coverage analysis [31], runtime monitoring [10, 12, 29], security [7, 32, 35], software development frameworks [5, 14], *etc.* In code instrumentation, the source code, intermediate code, or compiled binary code, is analysed either *statically* [14] at compile-time or *dynamically* [4, 8, 32] at runtime, and modified as necessary to include additional functionality. Static instrumentation tends to be more efficient, since the code is instrumented *once* during a pre-processing phase rather than at runtime for each execution. On the other hand, dynamic instrumentation tends to be more flexible since it allows for code modules to be replaced without requiring instrumentation beforehand.

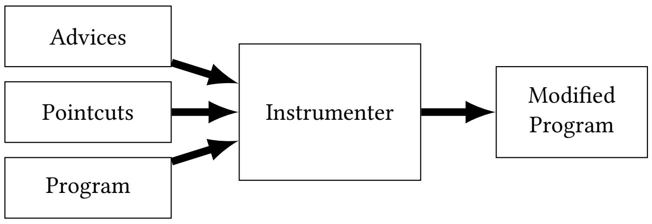

Figure 1. The AOP instrumentation process

Instrumentation is widely used by various software development tools. Compilers use code instrumentation to embed debugging symbols and breakpoints that allow for easier synchronisation between the compiled code and the source code while debugging. Testing tools utilise code instrumentation to measure statement coverage [31] of the designated test cases, while in runtime monitoring, instrumentation is used to inject monitoring code [10, 13, 15, 25, 29] to detect system executions violating some specification.

2.1 AOP Code Instrumentation

AOP [27, 30] employs code instrumentation for injecting aspect code that cross-cuts across multiple components within the code base of an application. In general, AOP frameworks perform code instrumentation by employing a static analyser known as the *instrumenter*. As shown in Figure 1, the instrumenter takes as input a program and an aspect specification consisting in 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.

2.2 Erlang constructs and components

Since Erlang implements the actor model [1], its main functional components are *actors*. 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 that may contain any kind of data including 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 statements thus have the following structure:

```
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*

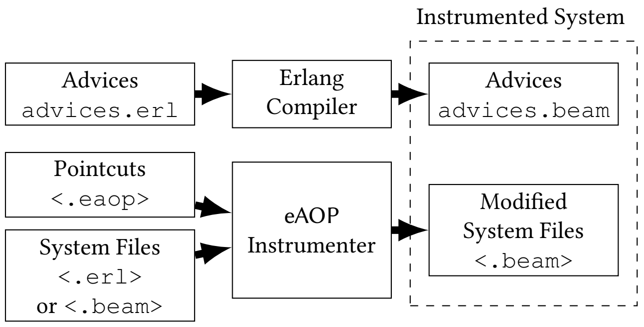

Figure 2. The components of the eAOP framework.

matching guard is executed; if none match, the messages remain in the mailbox and the actor blocks (until a new message is received).

Being also a functional language, Erlang actors execute functions. These 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`; functions may also be invoked locally (*i.e.,* by functions defined in the same module) without specifying the module, *i.e.,* `F(Args)`. In Erlang, the notation `M:F/A` (where `A` is the function arity) uniquely identifies a function. Erlang also encodes a few actor-related operations as *built-in functions* (BIFs) [2]. For instance, an actor can be spawned by calling one of the three functions `erlang:spawn/A`, where the arity `A` varies from 1 to 3. Actors can also be assigned names by registering an atom to their process id by using `erlang:register/2`.

3 Pointcuts and Advices

Our tool, eAOP, 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 2. Pointcuts must be defined in a text file suffixed with a `.eaop` extension, and must be forwarded 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`. Due to dynamic loading of modules in Erlang [2], the advices do not need to be forwarded to the instrumenter. In fact, 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 2.

In the remainder of the section we demonstrate the structure and use of pointcuts and advices in eAOP. Details about the eAOP instrumenter are then given in Section 4.

3.1 Pointcuts for eAOP

Pointcuts allow the user to specify code patterns which 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 must be defined as a *list of code patterns* in the following format:

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

The `<CType>` of a pointcut specifies the *construct type* requiring instrumentation, *i.e.,* the *category* of code operation (*e.g.,* function call, message send, *etc.*) that the instrumenter must look for while traversing the syntax tree of the given program. The module pattern, `<MP>`, and function pattern, `<FP>`, are used to 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 code operation that requires instrumentation. Note that the construct pattern `<CP>` varies depending on the construct type `<CType>`. For instance, if the construct type specifies a function call, the construct pattern must specify the module, function and arity of the function being called. The `<ATypes>` represent a list of *advice types* indicating the way the matched construct should be instrumented: they are discussed in depth in Section 3.2.

The eAOP framework can handle the following 4 construct types (`<CType>`):

`send`: With this type the instrumenter searches for *send operations* (*i.e.,* code excepts of the form `Pid!Msg`) defined in modules and functions whose name pattern matches `<MP>` and `<FP>` *resp.* In this case, pointcut pattern `<CP>` is used to specify a *message pattern*, in order to enable the instrumenter to only instrument send operations that match this pattern.

`'receive'`: *Receive operations* are instrumented whenever they are located within a function `<FP>` and module `<MP>`. A receive operation is only instrumented if one (or more) of its case guards (*i.e.,* the recieve message pattern excerpt `guard` in a construct `guard -> cont`) match the pointcut pattern defined in `<CP>`.

`call`: This type instructs the instrumenter to search for *function calls* (*i.e.,* code excerpts of the form `mod:fun(Args)`), that are being invoked from within a function whose name pattern matches `<FP>` and which must be residing in a module whose name matches `<MP>`. The construct pattern `<CP>` must specify the module, function and arity of the function being called.

`function`: This type is used to instrument *entire function definitions* (not function calls) which 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 object oriented programming, whereby the entire function body is *replaced* by another instrumented function. This is particularly useful in cases where one needs either to add functionality to a function or change it completely. Hence, the modules calling the function need not be instrumented in order to benefit from the new functionality of the instrumented function definition — this is discussed in more detail in Section 3.2.

Example 3.1. The code snippet below provides an Erlang program that implements a simple mathematical server as an Erlang module called `server.erl`. This module defines two functions, namely `start/0` and `math_loop/1`. The latter acts as a server loop that blocks waiting to receive and service mathematical requests (*i.e.,* atom tags `add` and `sub`) sent as messages by external clients. The math loop increments a counter after servicing `add` or `sub` requests. This counter can then be queried by a client by sending a service query message consisting of a tuple containing the atom tag `srv` and its process id. The `start/0` function serves to spawn a fresh server actor that initiates the math server loop by setting the counter to 0. It then proceeds by sending the process id of the newly spawned mathematical server to a registered process identified by the atom `registry`.

```
1  -module(server).
2  -export([start/0]).
3
4  start() ->
5     SPid = spawn(server, math_loop, [0]),
6     registry ! {math_srv,SPid}.
7
8  math_loop(Count) ->
9     receive
10      {add, Pid, N1, N2} ->
11        Pid ! {res, N1 + N2},
12        math_loop(Count + 1);
13      {sub, Pid, N1, N2} ->
14        Pid ! {res, N1 - N2},
15        math_loop(Count + 1);
16      {srv, Pid} ->
17        Pid ! {sys, Count},
18        math_loop(Count)
19    end.
```

Consider the following pointcut definitions:

```
Pointcut(call,"server", "start", ["erlang",
  "spawn","3"], <ATypes>
)
```

The construct type `call` instructs the instrumenter to search for *function calls* invoking `erlang:spawn/3` from within the `start` function defined in our `server` module. This will thus instrument line 5 in the server module.

```
Pointcut(send, "server", "math_loop", "{res,_}",
   <ATypes>
)
```

We can also instrument *message send operations* using the `send` construct type. The above pointcut notifies the instrumenter to search for send operations that are defined within the `math_loop` function body, situated inside the `"server"` module. In conjunction, a `send` operation is only instrumented if the sent message matches pattern `"{res,_}"`. Hence, the above pointcut leads to instrumenting lines 11 and 14. However, the send operation at line 6 will not be instrumented since the former is defined in the `start` function (and not the `math_loop` as specified). Similarly, since the message sent at line 17 does not consist in a tuple of two elements containing atom `res` as its first element, it is also not instrumented.

```
Pointcut('receive',"server", "math_loop",
   "{add,_,_,_}", <ATypes>
)
```

In the pointcut above we use the `'receive'` construct type to instrument the receive block residing in the function `math_loop`, as this block defines a case guard that matches `"{add,_,_,_}"` *i.e.,* a tuple containing four elements with the first element being the atom `add`. With this pointcut, the instrumenter modifies the receive block declared at line 9.

Finally, with the following pointcut we can instrument the entire `start/0` function definition residing in the `server` module.

```
Pointcut(function, "server", "start", "0",
   <ATypes>
)
```

This allows us to completely replace the function body of the function with our custom code. ■

3.1.1 Wildcards and Regular Expressions

String patterns defined in pointcuts, *i.e.,* `<MP>`, `<FP>`, *etc.,* can also be defined using *regular expressions.* For instance, recall that in Example 3.1, we want to instrument the point where the `start` function creates a fresh actor. In Erlang [2], this can be done using either `spawn`, `spawn_link` or `spawn_monitor`, each of which may vary in arity (between 1 and 3) and may also be called from either the `erlang` or the `proc_lib` module. Using regular expressions we can therefore redefine Example 3.1 into a more generic pointcut:

```
Pointcut(call, "server",
     "start([A-Za-z_]*)", ["(erlang|proc_lib)",
     "spawn(_(link|monitor))?","(1|2|3)"],
     <ATypes>
)
```

In the above pointcut we use standard regular expressions constructs such as: `*` to signify that the pattern `[A-Za-z_]` can match for 0 or more times, `|` to represent a choice between two patterns, and ? to imply that `_(link|monitor)` can occur either once or none at all.

As an alternative to regular expressions, one can also specify *wildcard patterns* by using "_" (some pointcut definitions presented in Example 3.1 already used them). Wildcards can be used to match anything. For instance, instrumenting *every* spawn operation in our module (*i.e.,* not just the one specified in the `start` function) may be achieved by defining the function pattern for our pointcut `<FP>` as:

```
Pointcut(call, "server", "_",
     ["(erlang|proc_lib)",
     "spawn(_(link|monitor))?","(1|2|3)"],
     <ATypes>
)
```

Wildcards are also useful to instrument constructs whose precise information may only be known at runtime. *E.g.,* to instrument a function call defined as `Module:foo()` (where `Module` is a variable) inside a function `bar()` defined in module `sample`, one can use the pattern

```
Pointcut(call, "sample", "bar", ["_", "foo", "0"],
     <ATypes>)
```

3.2 Advices for eAOP

Different pointcuts inject different advices based on 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 serve to advise the instrumenter about how it should instrument the matching code fragment and about which *advice function calls* should be injected. The instrumenter can only inject function calls to five advices functions, namely:

`advices:before_advice/5,`

advices:after_advice/5,
advices:intercept_advice/5,
advices:upon_advice/5, and
advices: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.

Our framework provides the following 5 advice types that can be used to specify which of the above listed advice function calls should be injected by the instrumenter.

- `before`: instructs the instrumenter to inject a call to the `advices:before_advice/5` advice function, *before* executing the respective action.
- `after`: notifies the instrumenter to insert a function call to `advices:after_advice/5`, *after* the instrumented action.
- `intercept`: the instrumenter *replaces* the specified action with a call to `advices:intercept_advice/5`.
- `upon` (`'receive'` only): injects a function invocation call to `advices:upon_advice/5` *after each matching receive case.*
- `override` (`function` only): the entire function body of a matching function definition is *replaced* by a call to `advices:override_advice/5`.

Note that multiple advice types can also be specified to inject multiple advices.

Example 3.2. Consider the following pointcut:

```
Pointcut(call, "server", "start",
    ["erlang","spawn","3"],
    [before,after,intercept]
)
```

This pointcut leads to replacing line 5 of Example 3.1 with a function call to `advices:intercept_advice`, that is preceded by a call to `advices:before_advice` and followed by another fucntion call to `advices:after_advice`. ■

In eAOP, the injected advices must be defined as an Erlang module called `advices.erl`, depicted earlier in Figure 2. This module must be created inside the *same directory* in which the instrumented `beam` files are set to be generated by the instrumenter, and must be compiled using the Erlang compiler. 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. The signature of the advice functions defined in `advices.erl` adheres to the following structure:

```
advices:XXX_advice(Type,Pid,Mod,Fun,Payload)
```

where the 5 function parameters have the following roles:

- `Type` represents the *construct type* as a discriminating atom denoting the function that is being instrumented and may consist of any of the tags `send`, `'receive'`, `call` or `function`.
- `Pid` is the *process id* of the actor in which the instrumented advice is being executed.
- `Mod` specifies to the *module* from which this advice is being called.
- `Fun` identifies the *function* from which this advice function is being invoked.
- `Payload` contains the *data* related to the instrumented Erlang constructs. The payload thus varies according to the discriminating construct type given by the `Type` argument; *e.g.,* for type `send`, at runtime the payload would consist of the actual message being sent by the instrumented send operation and the process id of the recipient actor, *i.e.,* `[Pid,Msg]`.

In what follows, we discuss the five advice functions supported by our eAOP tool. In each case, we explain the Payload parameter (that differs for every type) and give examples of how the respective instrumentation works *wrt.* the code introduced in Example 3.1.

3.2.1 The `before_advice` function

This function is executed *before* the instrumented Erlang action (*i.e.,* function call, message send or receive) is executed. The `Payload` parameter may take one of the following forms:

(*i*) `[]` - the empty list (no payload), in case the type of the instrumented action is a `'receive'` operation.
(*ii*) `[Pid, Msg]` - a list containing the `Pid` of the *recipient actor*, along with the sent message `Msg`, when the type of the instrumented action is a `send` operation.
(*iii*) `[M,F,A]` - a list containing the module `M`, function `F` and arguments `A` when the instrumentation concerns a function `call` operation.

Example 3.3. Consider the three pointcuts below, each defining a `before` advice type for one of the permitted construct types.[1]

```
[Pointcut('receive',"server","math_loop",
   "{add,_,_,_}", [before]),
 Pointcut(send, "server", "math_loop",
    "{res,_}", [before]),
 Pointcut(call, "server", "start",
   [ "erlang","spawn","3"], [before])].
```

The first pointcut specifies that the aspect code should be inserted *before* a `receive` construct that must be defined in `math_loop` and must have one of its guards matching pattern `"{add,_,_,_}"`. This results in the injection of a call to `before_advice` at line 9 in the instrumented code below, before the `receive` operation at line 10, with type `'receive'` and payload `[]`. The second pointcut also injects a `before_advice` at line 12, this time with type `send` and payload containing the process id of the recipient actor and the sent message. Finally, the third pointcut injects a `before_advice`, at line 2, with the type `call`.

```
1  start() ->
2    advices:before_advice(call,self(),
3        server,start,[erlang, spawn,
4        [server,math_loop,[0]]]),
5    SPid = spawn(server, math_loop, [0]),
6    registry ! {math_srv, SPid}.
7
8  math_loop(Count) ->
```

[1]The `function` construct type can only be used in conjunction with the `override` advice type.

```
   advices:before_advice('receive',self(),server,
       math_loop,[]),
   receive
    {add, Pid, N1, N2} ->
     advices:before_advice(send,self(),
      server,math_loop,[Pid, {res,N1+N2}]),
     Pid ! {res,N1+N2};
   ...
   end
 ...
```

The code snippet below shows how a before_advice function is typically defined in advices.erl. In this definition, the payload is inspected using a case statement to differentiate between the instrumented constructs.

```
before_advice(Type,Pid,M,F,Payload) ->
 case Payload of
  [] when Type=='receive' and M==server and F==
      math_loop ->
   print("Before Receive");
  [SentTo,Msg] when Type==send and M==server and
      F==math_loop ->
   print("Before Send, SPid:~p, Msg:~p",[SentTo,
       Msg]);
  [Mod,Func,Args] when Type==call and M==server
      and F==start ->
   print("Before Call, M:~p, F:~p, A:~p",[Mod,
       Func,Args])
 end.
```
■

3.2.2 The after_advice function

This advice is meant to be executed *after* the instrumented Erlang action has executed. The returned payload can be one of the following:

(*i*) [Msg] - the actual received message, when the instrumented action is a 'receive' operation.

(*ii*) [Pid, Msg] - same as in before_advice when the instrumented construct is a send operation.

(*iii*) [M,F,A,R] - the payload arguments are the same as for the before_advice in the case of a function call, except that along with the module name, function name and arguments of the called function, the payload includes the return value R as well.

Example 3.4. Consider the following after pointcuts.

```
[Pointcut('receive', "server","math_loop",
  "{add,_,_,_}", ['after']),
 Pointcut(send, "server", "math_loop",
   "{res,_}", ['after']),
 Pointcut(call, "server", "start",
   ["erlang","spawn","3"], ['after'])].
```

In the resulting instrumented code shown below, a call to the advice function after_advice is injected *after* each specified code construct with the appropriate type and payload.

```
start() ->
 R=SPid=spawn(server, math_loop, [0]),
 advices:after_advice(send,self(),server,start,
     [erlang, spawn, [server,math_loop,[0]], R]),
 registry ! {math_srv,SPid}.

math_loop(Count) ->
 receive
  {add, Pid, N1, N2} ->
   Pid ! {res,N1+N2},
   advices:after_advice(send,self(),
    server,math_loop,[Pid,{res,N1+N2}]),
   math_loop(Count + 1);
   ...
 end,
 advices:after_advice('receive',self(),server,
     math_loop,[Msg]),
 ...
```

The code snippet below shows a typical implementation for the after_advice function similar to that in Example 3.3.

```
after_advice(Type,Pid,M,F,Payload) ->
 case Payload of
  [Msg] when Type=='receive' and M==server and
      F==math_loop ->
    print("After Receive, Msg:~p",[Msg]);
  [Dest,Msg] when Type==send and M==server and
      F==math_loop ->
    print("After Send,SPid:~p,Msg:~p",[Dest,Msg]);
  [Mod,Func,Args] when Type==call and M==server
      and F==start ->
    print("After Call, M:~p, F:~p, Arguments:~p",
          [Mod,Func,Args])
 end.
```
■

3.2.3 The intercept_advice function

This advice is executed *instead of* the instrumented action. For call and send types, the payload has the same format as in advices:before_advice, but varies in the case of 'receive' operations as shown below:

(*i*) [Fun] - in the case of 'receive', the payload consists of an *anonymous* function that takes *no* input arguments. When executed, this function executes the intercepted receive operation.

Example 3.5. Consider the intercept advice type pointcut definitions below, similar to those in Example 3.3 and Example 3.4.

```
[Pointcut('receive', "server","math_loop",
   {add,_,_,_}, [intercept]),
 Pointcut(send, "server", "math_loop",
   {res,_}, [intercept]),
 Pointcut(call, "server", "start",
   ["erlang","spawn","3"], [intercept])].
```

As shown below, the intercept_advice function calls *replace* the code excerpts identified by the pointcuts. For instance, the third pointcut causes the instrumenter to replace the spawn operation on line 2 with a function call to intercept_advice of type call along with the module, function and arguments of the replaced spawn construct.

```
1 start() ->
2  SPid = advices:intercept_advice(call, self(),
       server, start, [erlang, spawn, [server,
       math_loop,[0]]),
  registry ! {math_srv,SPid}.

math_loop(Count) ->
 IFunc = fun() ->
  receive
   {add, Pid, N1, N2} ->
    advices:intercept_advice(send,self(),
     server,math_loop,[Pid, {res,N1+N2}]),
   ...
 end,
 advices:intercept_advice('receive',self(),
     server,math_loop,[IFunc]),
 ...
```

The snippet below shows a definition for the `intercept_advice` function. In particular, note that the case for the `'receive'` type shows how the intercepted receive operation, stored in `IFunc`, can be invoked as a normal function.

```
intercept_advice(Type,Pid,M,F,Payload) ->
 case Payload of
  [IFunc] when Type=='receive' and M==server and
      F==math_loop ->
  print("Applying Intercepted Receive"),
  IFunc();
  ...
 end.
```

■

3.2.4 The upon_advice function

This advice can only be used for `'receive'` pointcuts, since it is called after a receive guard is matched. Recall that a `receive` construct may contain multiple pattern-matching guarded clauses as discussed in Section 2. The AOP weaves `upon_advice` for each guarded expression whose guard matches the pointcut pattern (*i.e.*, the continuation excerpt `guard` in a construct `guard -> cont`). At runtime, only *one* `receive` guarded expression is triggered, at which point the necessary pattern-matched data of the event is known and can thus be forwarded as payload to the instrumented advice.

Example 3.6. The two pointcuts below specify receive construct types with an upon advice type.

```
[Pointcut('receive',"server","math_loop",
    "{add,_,_,_}", [upon]),
Pointcut('receive',"server","math_loop",
    "{srv,_}", [upon])].
```

As shown below, a call to `upon_advice` is injected after the receive guards that match patterns `"{add,_,_,_}"` and `"{srv,_}"`. The advices are thus injected at lines 4 and 10 after the guard cases `"{add,Pid,N1,N2}"` and `"{srv,Pid}"`.

```
1  math_loop() ->
2    receive
3     Msg = {add,Pid,N1,N2} ->
4       advices:upon_advice('receive',self(),server,
            math_loop,[Msg]),
5       Pid ! {res, N1 + N2},
6       math_loop(Count + 1);
7     Msg = {sub,Pid,N1,N2} ->
8       Pid ! {res, N1 - N2}, ...
9     Msg = {srv,Pid} ->
10      advices:upon_advice('receive',self(),server,
            math_loop,[Msg]),
11      Pid ! {sys, Count},
12      math_loop(Count)
13   end.
14 ...
```

In the `advices.erl` file, `upon_advice` functions are defined using a structure that is very similar to that one seen earlier for the functions `before_advice`, `after_advice` and `intercept_advice`. ■

3.2.5 The override_advice function

This can be used *exclusively* for `function` pointcuts. Whenever a function pointer matches the name of a function definition, its function body is *replaced* by a function call to `override_advice`. A new/fresh function definition is then created using the body of the matched function definition. The overridden function is renamed, using its original name suffixed by `"_@"`, *i.e.*, its name becomes `<Function>_@` where `<Function>` is its original name. The `Payload` consists of a list `[M,F,A]` where `M`, `F` and `A` respectively refer the module, function and arguments of the overridden function. These can be used within the `override_advice` to call the base function by calling `"apply(M,F,A)"`.

Example 3.7. The pointcut below specifies that the `start` function should be overridden by the instrumented code.

```
[Pointcut(function,"server","start","0",
    [override]
)].
```

As shown in the code below, the original `start` function is *renamed* to `start_@`, and a new `start` function is introduced. Moreover the body of the original `start` function is *replaced* entirely by a call to `override_advice`. This advice is invoked with arguments `function` as the type, and `[server,start_@,[]]` as payload. Note that the payload provides the necessary information for invoking the overridden function from within the advice definition.

```
start() ->
  advices:override_advice(function,self(),
    server,start,[server,start_@,[]]).

start_@() ->
  SPid = spawn(server, math_loop, [0]),
  registry ! {math_srv,SPid}.
```

The sample code below defines an `override_advice` function that prints the name of the overridden function before reapplying the overridden function by calling Erlang's built in function `apply(Mod,Func,Args)`.

```
override_advice(Type,Pid,M,F,Payload) ->
 case Payload of
  [Mod,Func,Args] when Type==function and M==
      server and F==start ->
    print("Function ~p was overridden.", [Func]),
    print("Reapplying base function ~p.",[Func]),
    apply(Mod,Func,Args);
 end.
```

■

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; see Figure 3`(i)`. Compiled beam files can only be processed if they have been *compiled in debug mode*: `.beam` files compiled in debug mode are embedded with debug information from which the syntax tree of the compiled module can easily be extracted.

Once these files are provided, the instrumenter proceeds to compile the given source files (`.erl`) using Erlang's compilation function `compile:file/3`. As for the compiled `.beam` files, the embedded abstract code is first extracted via `beam_lib:chunks/2` and then re-compiled using `compile:forms/2`—both functions are provided by Erlang libraries [2]. To initiate the AOP weaving process, both compilation functions are executed using the `parse_transform` option to specify that the compiler must invoke our weaver module during compilation.

Once the syntax tree of a module is obtained via this method, the instrumenter applies a *depth first search* traversal during which every node is pattern matched with the specified pointcut patterns. Whenever a node matches a pointcut pattern, the parse tree is modified according to the type of advices that need to be injected. For instance, in Figure 3 `(ii)`, the matching node is *replaced* by the instrumented code whenever an `intercept` advice type is specified, while in the case of `before` and `after` advice types, function calls to the respective advices are instrumented *before* and *after* the matching node *resp.* Finally, the modified tree is compiled into the required `.beam` file which amounts to the instrumented module.

Remark. *Due to the lack of documentation provided regarding the abstract forms of the tree, modifying the abstract syntax tree proved to be a delicate task: the structure of the syntax tree was mostly understood through reverse engineering. Since extensive changes to the syntax tree can easily lead to the introduction of errors, we opted to minimize the number of changes by limiting the instrumented code to just function call insertions to the respective advice functions instead of inlining the full advice code.* ■

4.1 Instrumenting a System with eAOP

To instrument an Erlang system with eAOP one needs to use one of the following methods, exported by the `eaop` module:

(*i*) `eaop:instrument(SrcDirs, ConfDirs)`: where argument `SrcDirs` refers to a *list of directories* containing either the system `.erl` (source) files or `.beam` files, compiled in debug mode, that require instrumentation. `ConfDirs` refers to a list of directories containing the pointcut definitions specified in `.eaop` files.

(*ii*) `eaop:instrument(SrcDirs, ConfDirs, Options)`: same as (*i*), but also accepts a *list of compiling options* identical to the compiling options definable for `compile:file` (see [18]). The most useful option is the `{outdir,DestDir}` option which specifies the destination directory (`DestDir`) in which the instrumented beam files will be generated. Along with the standard options, our instrumenter accepts the following additional options:

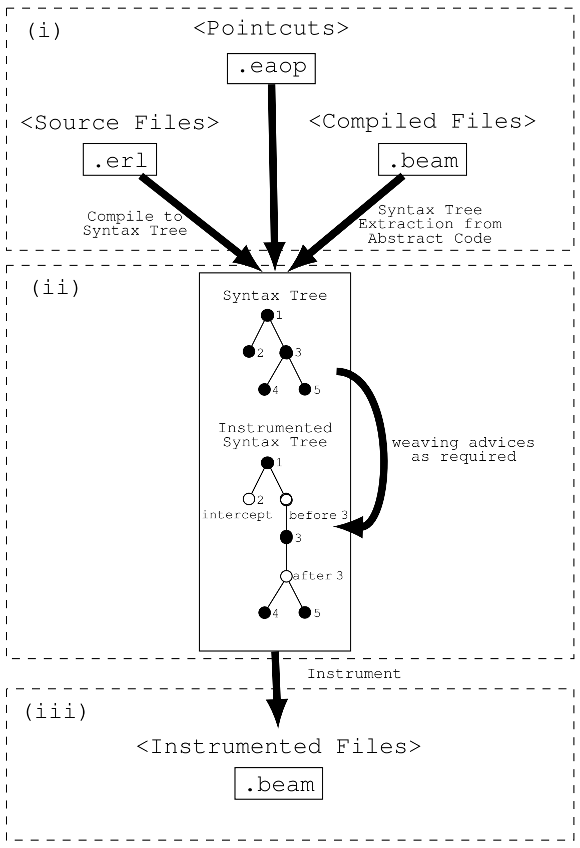

Figure 3. The instrumentation process.

- `output_instrumented_src`: A flag instructing the instrumenter to *output the source files* of the instrumented system. This allows for a better understanding of how the system has been modified by the instrumenter, and it also eases the process of debugging instrumented systems.
- `{outsrcdir,DestDir}`: This option serves to specify the destination directory where the instrumented *source files* are to be created. Unless this option is specified, the source files are generated in the same output directory as specified by the `{outdir,DestDir}` option.
- `gen_advice_template`: This flag alerts the instrumenter to search for `advices.erl` in the output directory. If this is not found, it creates a file containing a *template* for specifying the required advices — this file must then be modified accordingly. If the file already exists, the instrumenter leaves it untouched.
- `compile_advices`: This flag notifies the instrumenter to search for and recompile `advices.erl`. If this is not found, a warning is issued notifying the user that this file is missing.

(*iii*) `eaop:instrument_srcs(SrcFiles, ConfigDirs, Options)`: same as (*ii*) but argument `SrcFiles` requires a list of paths leading to system source `.erl` files.

(*iv*) `eaop:instrument_beams(BeamFiles, ConfigDirs, Options)`: same as (*iii*) but argument `BeamFiles` accepts a list of paths leading to system `.beam` files, compiled in debug mode, instead of its source files.

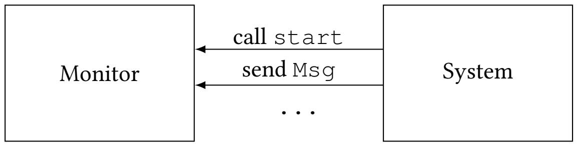

Figure 4. Instrumentation for Asynchronous Monitoring

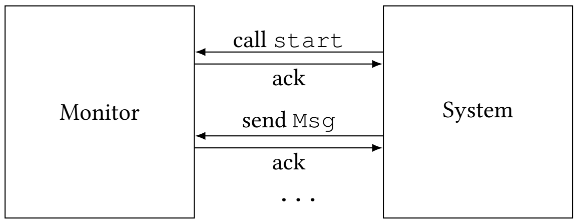

Figure 5. Instrumentation for Synchronous Monitoring

5 Applications of eAOP

The instrumentation of the running example in Section 3 may easily be extended to produce execution logs that can be then used for debugging purposes. Moreover, by intercepting code constructs and overriding functions in eAOP, one can easily attain code refactoring as discussed in Section 1. In what follows, we overview how eAOP can be used to augment program functionality (via adaptation), and to carry out runtime software analysis (via monitoring).

In [10], Cassar *et. al* used eAOP as part of a toolchain making up a *runtime monitoring* framework called DetectEr 2.0[2]. This prototype monitoring tool generates monitoring actors from a given specification defined using a program logic called recHML [23]. This monitoring tool also requires instrumenting the system under scrutiny, in such a way that allows the instrumented system to be able to report trace events to the autonomously executing monitoring actors using conventional message passing.

The instrumentation used by the tool varies depending on the chosen monitoring mechanism, namely *synchronous*, *asynchronous* or *hybrid* monitoring. In synchronous monitoring, the system is forced to *wait* until the monitor finishes processesing the reported trace event before it can proceed with its execution, while in asynchronous monitoring the system does not block whenever a trace event is reported to the monitor. Hybrid monitoring constitutes a mixture of the two mechanisms. Although asynchronous monitoring was shown to be more efficient in [10], the inclusion of synchronous monitoring was crucial so as to allow the monitor to detect erroneous behaviour *before* the offending actors are allowed to proceed with their execution, thus preventing further misbehaviour from these actors.

Although asynchronous monitoring can be implemented using Erlang's native tracing mechanism (as was done already in [3, 22]) this mechanism cannot be used for synchronous and hybrid monitoring. Therefore instrumentation was employed. As shown in Figure 4, to implement asynchronous monitoring with AOP, the instrumentation code is used to asynchronously deliver an event notification as a message to the monitor before the specified action executes. On the other hand, implementing synchronous monitoring requires the instrumentation of a *handshake protocol*, as depicted in Figure 5. 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 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 the eAOP framework to perform the required instrumentation. For instance, to synchronously monitor the send operations in the `math_loop` function (defined in Example 3.1), the tool generates the pointcut

```
[Pointcut(send,sever,math_loop,_)].
```

and its corresponding advice

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

The generated advice reports the operation by sending a trace message 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.

The eAOP framework was also used to automate *runtime adaptation* monitors for Erlang systems [11, 12], using a tool called AdaptEr[3]. AdaptEr was developed as an extension to DetectEr 2.0, and was capable of applying adaptation actions to specific actors without affecting the execution of the other actors. Adaptations can be used 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 which the monitor delivers the required adaptation using message passing.

As illustrated in Figure 6, the instrumented protocol builds on the synchronous monitoring protocol introduced in DetectEr 2.0 [10], by forcing the system actor to block after forwarding the trace event to the monitor. The blocked actor would then wait for either an acknowledgement message 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

[2]This runtime monitoring tool is open source and downloadable from https://bitbucket.org/casian/detecter2.0

[3]DetectEr is an open-source runtime adaptation prototype tool downloadable from https://bitbucket.org/casian/adapter

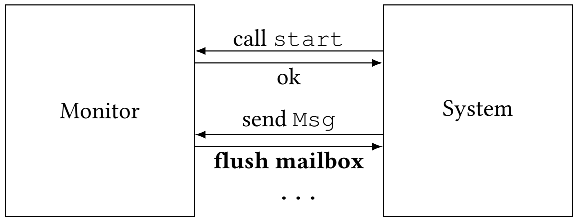

Figure 6. Instrumentation for Runtime Adaptation

actor to execute the requested adaptation action. Using our eAOP framework, this instrumented protocol permitted the monitor to deliver (intrusive) adaptation actions through message passing.

```
before_advice(Type,Sender,M,F,Payload) ->
 case Payload of
  [SentTo,Msg] when Type==send and M==server
      and F==math_loop ->
   detecter!{trace,send,Sender,SentTo,Msg},
   receive
     ack -> ok;
     flush -> adapter:flush();
     restart -> adapter:restart();
     ...
   end,
 end.
```

Both DetectEr 2.0 and AdaptEr were used to instrument monitoring systems via eAOP for several industry-scale applications such as Yaws and Ranch [3, 10]. Recent work in [12] also showed how AdaptEr can be used to create a quick patch for mitigating the effects of the *Directory Traversal* vulnerability that was found in Yaws 1.89 [26] and which made it vulnerable to *dot-dot-slash attacks*.

6 Conclusion

We investigated the applicability of AOP mechanisms for Erlang, an archetypal actor-based language. Via our prototype tool, eAOP, the running example of Section 3 and the case studies in Section 5 we are able to show that (*i*) actor-based constructs lend themselves well to AOP techniques and that (*ii*) aspect oriented programming can be used effectively for actor-based software development. A further contribution of our work is the AOP tool itself[4] that works for both Erlang source and intermediary code.

6.1 Related Work

Several forms of instrumentation exist other than AOP. For instance, DynamoRIO [4, 8] is a dynamic binary instrumentation [32] framework that provides an API for achieving runtime code manipulation. Such manipulation includes the insertion of trampolines *i.e.,* callback functions that are called when something specific is executed, and other assembly level instructions. Dynamic binary instrumentation is most popular in the field of security as it is generally used to strengthen software against vulnerabilities that can be exploited for malicious purposes. Although this framework is highly efficient, it does not provide any type of abstraction. In fact, the user is required to have deep knowledge of the hardware architecture and the processor's instruction set.

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

To our knowledge, ErlAop[5] is the only other AOP framework for Erlang that exists apart from eAOP. ErlAop, however, lacked a number of important features used in the monitoring tools discussed in Section 5. For instance, ErlAop does not provide any functionality similar to our `upon_advice` for instrumenting individual message receive cases. The use of the `upon_advice` proved to be crucial for injecting monitoring code after specific individual receive cases. Moreover, ErlAop does not provide any form of function overriding functionality similar to our `override_advice`. Also, ErlAop only permits the instrumentation of function calls, whereas eAOP permits the instrumentation of message sends and receives. ErlAop is also incapable of outputting the modified (instrumented) code and does not work on compiled `.beam` files. ErlAop, however, provides around advices, which are equivalent to our `intercept` advices.

Although eAOP targets specifically the Erlang language, it is equipped with the standard pointcut types (*i.e.,* `before`, `after`, *etc.*) found in mature AOP frameworks such as AspectJ [30] for Java and PostSharp [21] for C#; this gives eAOP a comparable expressive power.

6.2 Future Work

As future work, we plan to add several other features to our framework that could improve its expressiveness and flexibility. For instance, currently the required advices must always be defined within a source file called `advices.erl`. This introduces two limitations, namely, (*i*) that the user must manually define advices and employ case statements accordingly, and (*ii*) that merging multiple instrumented systems requires manually merging their `advices.erl` file into a single file so as to avoid naming conflicts of modules.

To address (*i*), we thus plan to allow the user to define advices along with the pointcuts, *i.e.,* directly within the `.eaop` specification file. Upon processing the specification file, in order to cater for (*ii*), the instrumenter should then generate a uniquely named module (*e.g.,* advices_(md5 hash).erl) for hosting the required advice functions. This addition will, however, require a more expressive pointcut language to allow for advices to be defined within the pointcut specification.

We also plan to include other advice types including types related to exception handling such as `upon_throw`, `upon_error` and `upon_exit`. With these advice types, the user can instruct the instrumenter to inject code at specific *try-catch cases*. This feature was not given priority because Erlang code practices generally discourage the use of traditional exception handling, due to Erlang's let-it-fail philosophy. There are, however, cases where exceptions and local error handling are useful.

References

[1] G. Agha. *An overview of actor languages.* ACM, 1986.
[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
[3] D. P. Attard, I. Cassar, A. Francalanza, L. Aceto, and A. Ingolfsdottir. A runtime monitoring tool for actor-based systems. *Behavioural Types: from Theory to Tools.*, 2017.
[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM*, 35:1–12, 2000.
[5] A. Bawiskar, P. Sawant, V. Kankate, and B. Meshram. Spring Framework: A Companion to JavaEE. *IJCEM*, 1:41–49, 2012.

[5]Accessible from http://erlaop.sourceforge.net/

[6] G. A. T. Bayona. Aspect oriented programming meets design patterns. Master's thesis, University of Manchester, Faculty of Engineering and Physical Sciences, 2013.
[7] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *IEEE/ACM Symposium on Code Generation and Optimization*, pages 213–223. IEEE, 2011.
[8] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. *ACM*, 47:133–144, 2012.
[9] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing design patterns: A scalability study of aop. In *AOSD*, pages 109–121. ACM, 2006.
[10] I. Cassar and A. Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. *arXiv:1502.03514*, 2015.
[11] I. Cassar and A. Francalanza. Runtime adaptation for actor systems. In *RV*, pages 38–54. Springer, 2015.
[12] I. Cassar and A. Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *IFM*, pages 176–192. Springer, 2016.
[13] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. A Survey of Runtime Monitoring Instrumentation Techniques. In *PrePost*, EPTCS, 2017. (To appear).
[14] F. Chen and G. Roşu. MOP: An efficient and generic Runtime Verification framework. In *ACM SIGPLAN Notices 2007*, volume 42, pages 569–588.
[15] F. Chen and G. Roşu. *Java-MOP: A Monitoring Oriented Programming Environment for Java*, pages 546–550. 2005.
[16] S. Chiba and R. Ishikawa. Aspect-oriented programming beyond dependency injection. *ECOOP*, pages 121–143, 2005.
[17] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ACM SIGSOFT*, volume 26, pages 88–98. ACM, 2001.
[18] Ericsson. Compiler reference manual. Online. http://erlang.org/doc/man/compile.html#file-2.
[19] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, Oct. 1997.
[20] I. Figueroa, N. Tabareau, and É. Tanter. *Effective Aspects: A Typed Monadic Embedding of Pointcuts and Advice*, pages 145–192. Springer, 2014.
[21] G. Fraiteur. A Thread-Safe Extension to Object-Oriented Programming. Technical report, PostSharp Technologies.
[22] A. Francalanza and A. Seychell. Synthesising correct concurrent runtime monitors. *FMSD*, 46(3):226–261, 2015.
[23] A. Francalanza, L. Aceto, and A. Ingólfsdóttir. On verifying hennessy-milner logic with recursion at runtime. In *RV*, pages 71–86, 2015.
[24] K. Gregor, H. Erik, H. Jim, K. Mik, P. Jeffrey, and G. G. William. An Overview of AspectJ. In *ECOOP*, 2001.
[25] K. Havelund and G. Roşu. An Overview of the RV tool Java PathExplorer. *FMSD*, 24(2):189–215, 2004.
[26] A. Hernandez. Yaws 1.89: Directory Traversal Vulnerability. Available online at www.exploit-db.com/exploits/15371/, 2010. Accessed on 24/5/2017.
[27] I. Jacobson and P.-W. Ng. *Aspect-oriented software development with use cases*. Addison-Wesley Professional, 2004.
[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*, pages 220–242. Springer, 1997.
[29] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *FMSD*, 24(2):129–155, 2004.
[30] R. Laddad. *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press, 2003.
[31] Mountainminds GmbH & Co. KG. EclEmma - Java Code Coverage for Eclipse. Online. www.eclemma.org/.
[32] N. Nethercote. *Dynamic binary analysis and instrumentation*. PhD thesis, University of Cambridge, 2004.
[33] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. AOSD, pages 181–191. ACM, 2005.
[34] C. Sant'Anna, A. Garcia, U. Kulesza, C. Lucena, and A. v. Staa. Design patterns as aspects: A quantitative assessment. *JBCS*, 10(2):42–55, 2004.
[35] K. Sirbi and P. J. Kulkarni. Stronger enforcement of security using AOP and Spring AOP. *arXiv:1006.4550*, 2010.
[36] O. Spinczyk, A. Gal, and W. SchrÃűder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *CRPIT '02*, pages 53–60, 2002.
[37] G. Xu, Z. Yang, H. Huang, Q. Chen, L. Chen, and F. Xu. JAOUT: Automated generation of Aspect-oriented unit test. In *APSEC*, pages 374–381, Nov 2004.