UNIVERSITY OF GOTHENBURG

Fault Injection Technique for Evaluating Erlang Test Suites

QUANG HOAT DO
TAIWO DAYO AJAKAIYE

Bachelor of Software Engineering and Management Thesis

Report No. 2009:072
ISSN: 1651-4769

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2009

Fault Injection Technique for Evaluating Erlang Test Suites

Quang Hoat Do
IT University of Göteborg
Software Engineering and Management
Gothenburg, Sweden
quangd@ituniv.se

Taiwo Dayo Ajakaiye
IT University of Göteborg
Software Engineering and Management
Gothenburg, Sweden
mailajaks@yahoo.com

Abstract

In software testing, fault injection involves injecting abnormalities into software programs. This can then be used to evaluate test suites by checking how well they detect those abnormalities. This study involves finding out what typical faults occur in Erlang programs by analyzing data from Erlang/OTP releases, official Erlang reference manual, Erlang bug reports and other related studies. It will also include proposals of how these faults can be injected into Erlang programs based on our Erlang development experience and knowledge. The method adopted in this study involves the implementation of a fault injection tool which was evaluated on the test suite of Erlang/OTP R13B array module. This study contributes knowledge to how fault injection can be used to evaluate Erlang test suites. This in summary involves the following (1) injecting non-trivial faults one at a time into a target Erlang program; these are faults that cannot be detected at compile time, by dialyzer or by a test suite and cover information, and (2) evaluating the program test suite by studying if it can identify the injected fault, and if not why.

Keywords: fault injection, test suites, Erlang typical fault.

1. Introduction

In software testing, a test suite is a collection of test cases that are used to test a software program with the aim of verifying and validating system's behavior in accordance with customer's requirements. A test suite is effective if it can detect present errors; the more errors it can detect the more effective it is. Test suites are specific for individual programs, therefore it is logical to evaluate a test suite on the program it is meant for. When evaluating a test suite on a program, one is confronted with the problem that one doesn't really know whether there are any errors in the code and if so, how many. One way to evaluate a test suite is to monitor whether one can use it to find the same errors in codes as previous ones. One has a list of reported errors from a previous test suite and applies the new test suite to see if these errors can be detected. The strength of this evaluation is that it shows one can find errors in the target program. The weaknesses are in the first place that there might be few errors in the program and that it is hard to say that it is good in finding errors in general. In the second place, one may conclude that it is as good as the previous test suites already in place if one cannot find more errors than previous ones have done.

In order to evaluate test suites on software programs, fault injection technique [1] can be used. In software testing, fault injection involves injecting abnormalities into software programs. This can then be used to evaluate test suites by checking how well they detect those abnormalities. In order to use fault injection properly, one would like to inject faults into the program code that are typical for that kind of code. This differs from programming language to programming language, e.g. in C [44] one can inject faults around pointer dereferencing, whereas for Java [45] that would not make sense. It also differs from one application domain to another, e.g. in a highly concurrent programming domain, one would typically like to inject faults that cause race conditions, whereas in another domain one probably focuses on out-of-bound arrays.

In a study titled *Evaluating Test Suites and Adequacy Criteria Using Simulation-Based Models of Distributed Systems* [2], the authors touched on a testing method based on discrete-event simulations, a fault-based analysis technique for evaluating test suites and adequacy criteria, and a series of case studies that validate the method and technique. Here, the fault-based analysis uses a related form of fault injection technique on the simulation-based specification to provide a fault against which test suites and the criteria that formed them can be evaluated. Many studies [3-21] have also adopted the use of fault injection technique in evaluating computer system dependability, understanding large systems failure, testing distributed object systems, fault injection in distributed systems, evaluation of fault tolerant systems, etc. However, none of these studies [3-21] have addressed how fault injection technique can be used for evaluating Erlang program test suites. Erlang [22] was developed by Ericsson [23] in the early nineties. It is a concurrent functional programming language with specific features for the development of distributed, fault-tolerant systems with soft real-time requirements. Today, Erlang is used in several application domains such as computer telephony, banking, TCP/IP programming (HTTP, SSL, Email, Instant messaging, etc) and 3D-modelling. This study will adopt the use of fault injection technique based on its appropriateness to be used in evaluating software testing suites,

and will present a way in which fault injection can be used to evaluate Erlang testing suites.

The purpose of this qualitative study is to show how Fault Injection Technique can be used to evaluate Erlang software test suites. This will involve finding out what typical faults occur in Erlang programs by analyzing data from Erlang/OTP releases, official Erlang reference manual, Erlang bug reports and related studies. It will also include proposals on how these faults can be injected into Erlang programs based on our Erlang development experience and knowledge. This study will help Erlang testers to write better test suites by presenting Erlang typical faults. It provides the knowledge and knowhow in support of developers who want to develop a fault injection tool for Erlang programs. This study will also help test teams to evaluate how effective their test suites are in terms of how much fault they can detect.

2. Research Method

The goal of the section is to find out how fault injection technique can be used to evaluate test suites. This includes searching various Erlang sources for information about what typical faults exist, and proposing ways in which these faults can be injected into Erlang programs. The two phase qualitative approach illustrated in figure 1 show how this was done. This approach was chosen because there was a need to explore various Erlang resources for information about Erlang typical faults. This is required to inject meaningful faults into Erlang programs which are actually encountered while developing or running Erlang programs. In phase 1, information about Erlang faults were collected and analyzed from 4 data sources which are Erlang/OTP releases, official Erlang reference manual, Erlang bug reports and related studies. The outcome of the analysis was validated and the result was a descriptive list of Erlang typical faults. In phase 2, based on the typical faults gotten from phase 1, and together with our Erlang development experience, we proposed and validated ways in which these faults could be injected into Erlang programs.

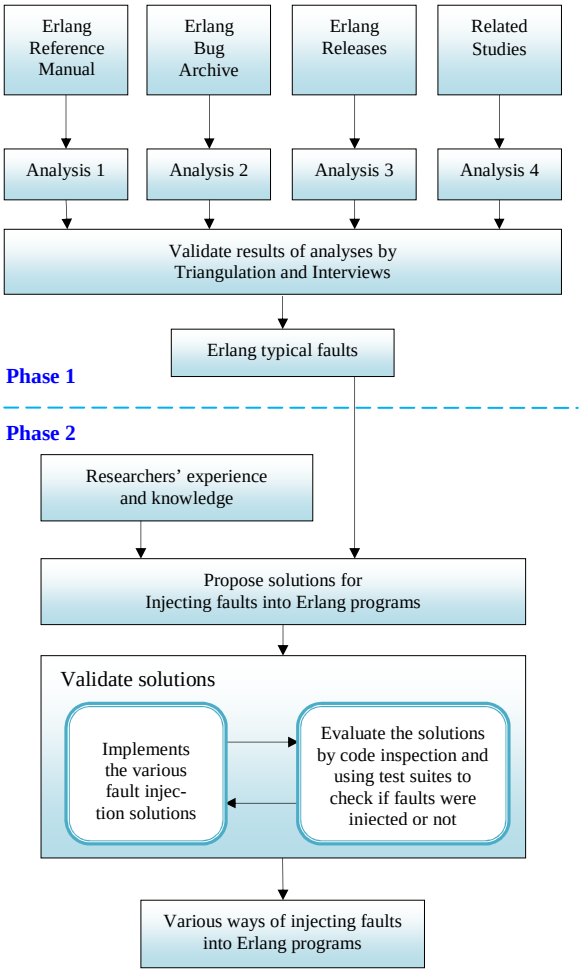

Figure 1 – Research method overview

2.1. Phase 1

The following sections describe how Erlang typical faults were realized by analyzing and validating data collected from the Official Erlang OTP releases, Erlang-bug archives, Erlang reference manual and other related studies (see table 1).

2.1.1. Data Collection

The following table describes the data source, data type (e.g. text, source code etc.), data form (e.g. written text, audio recording etc.) and data collection type (e.g. documents, Interviews etc.) of this phase's data collection.

Table 1 - Data Collection

Data source	*Type of data*	*Data form*	*Collection type*
Official Erlang OTP releases [24]	Erlang Release notes	Written text	Documents
Official Erlang-bug archives: May 2009 to October 2008 [25]	Text and source code	Written text	Documents
Related study [26][27]	Articles	Written text	Documents
Official Erlang Reference Manual Version 5.7.1 [28]	Documentation	Written text	Documents

2.1.2. Analysis

This section describes how the collected data (see section 2.1.1) were analyzed. Separate analyses were carried out on the data obtained from individual data sources. This was because each data source was unique and thus required a different analysis. The aim of these analyses was to find out "what typical faults occur in Erlang programs".

Data source 1

Erlang Reference Manual [28] contains a complete description of the Erlang programming language. This was an ideal place to look for information on Erlang typical faults, because it contained information about what typical faults could occur in Erlang programs.

Analysis 1

The various faults described here and the reasons why they occur were simply collected from the Erlang reference manual.

Data source 2

The Erlang-bug archives contain Erlang/OTP bugs that have been continuously reported since April 2003 [25]. This source was considered because we wanted to see what type of Erlang faults developers encounter while developing Erlang programs. We were able to go through the bug archives from May 2009 to October 2008 based on the available time for this study. Active discussions on Erlang faults between the Erlang/OTP development team and regular Erlang developers also took place here. This provided a medium from which reasons why these faults occur could be easily obtained.

Analysis 2

The reasons for why the bugs reported in these archives occurred, were carefully studied and collected. Referenced modules in the Erlang/OTP releases (see data source 3 bellow) were also studied to get a deeper understand of the root causes of faults that were reported.

Data source 3

Erlang/OTP releases [24] comprises of source code, a release note and documentation. The available releases at the time of conducting this analysis were R10B-0 to R13B. This data source was studied whenever there was a reference to it from the *Erlang bug archive* data source above.

Analysis 3

References from the *bug archives* (see Analysis 2) mostly refer to particular functions or modules within certain releases. The difference between the module/function in the release where the faults were located, and the same module/function in the next release where the faults were fixed, was studied with aim of locating the root cause of the fault.

Data source 4

A couple of related studies have been conducted on distributed and concurrent programs such as Erlang. Mats Cronqvist conducted a study on *Troubleshooting a Large Erlang System* [26]. The system under study here was AXD 301 (a multi-service switch from Ericsson AB), with over a 1000 usage registered as at when the study was conducted. Another study titled *Typing for Reliable Distributed Systems - Recent Advances* [27], touched on using advanced type systems for statically detecting non-trivial programming errors in distributed and concurrent programs. These studies were chosen because they identified several typical faults that occur in Erlang and distributed systems.

Analysis 4

Erlang typical faults such as deadlock, race condition. were presented during the course of carrying out the studies described above. These faults were studied and relevant ones were collected and documented.

2.1.3. Validation

The results of all the analyses conducted on data obtained from the Official Erlang OTP releases, Erlang-bug archives, Erlang reference manual and other related studies (see section 2.1.2) were validated to be Erlang typical faults by conducting *triangulation* and *interviews* (see below). This two strategies of validation were adopted to make the validation process more concrete. The outcome of this validation led to a descriptive list of Erlang typical faults which are presented in the result section of this study (see section 3.1).

Triangulation

Triangulation [29] is a way of validating data collected from different data sources especially when it comes to small exploratory research such as this study. Thus, this method has been adopted based on its suitability. Applied to this study, faults obtained from each data source were validated to be Erlang typical fault by examining other data sources for prove supporting this.

Interviews

Erlang typical faults collected by analyzing the various data sources in section 2.1.2 were also validated by conducting interviews with Erlang developers and researchers. This method of validation was adopted in order to get an input from those that actually program in Erlang and encounter these faults from time to time.

2.2. Phase 2

This phase was part of the steps that would show how fault injection technique can be used to evaluate Erlang testing suites (see figure 1: phase 2). Thus, it built on the result of Phase 1(see section 2.1). This phase contained data collection, analysis and validation. It resulted in solutions on how non-trivial faults can be injected into Erlang programs (see section 3.3).

2.2.1. Data collection

This phase built on the various Erlang typical faults realized from Phase 1 (see section 3.1). Proposing how faults can be injected into Erlang program at run time required familiarity and development experience with the Erlang programming language. Therefore, these typical faults and our Erlang development experience and knowledge served as the data source for this phase.

2.2.2. Analysis

The aim of the analysis conducted here was to find out how the typical faults from Phase 1 can be injected into Erlang programs at run time. Based on our development experience and knowledge of the Erlang Programming language, we proposed solutions to how this can be done. The various Erlang typical faults described in section 3.1 were analyzed, firstly by checking if they could not be detected at compile time, and secondly that they could not be detected or

evaluated by available Erlang tools such as *dialyzer* and *cover* (see below). The reason for carrying out all this checks was that we did not want to inject trivial faults. Thus, solutions to how faults can be injected into Erlang programs would be proposed for non-compile time faults that couldn't be detected or evaluated by *dialyzer* and the information from *cover* analysis. These faults are *failed function clause match, deadlocks, race condition* and *failed case clause match.* A fault can only occur in a program if conditions that cause it to arise are present; for example, the chance of *deadlocks* occurring in an Erlang program with only one process is very rare. With this in mind, solutions were only proposed for the faults that can be validated with the chosen target program. In order for the fault injection solutions to be validated, a fault injection tool was implemented that executed the solutions. This tool was then evaluated on the target program.

Dialyzer

Dialyzer is a static analysis tool that identifies software discrepancies such as type errors, unreachable code etc. in a single Erlang module or applications [30]. Using *dialyzer* as a criterion for screening which faults should or should not be used for fault injection, eliminates trivial faults such as type errors (e.g. *wrong arguments* in section 3.1.5) or unreachable code (e.g. *calling a non existing function* in section 3.1.9).

Cover

Cover is a coverage analysis tool for Erlang programs. It can be used to verify test cases and to make sure that all relevant code is covered. It may also be helpful when looking for bottlenecks in the code [31]. Fault injection is irrelevant if faults are injected in the code areas that are not covered by the available test cases. With these test cases, injecting fault in such areas will never be detected. One of the conditions with fault injection is that, it shouldn't be impossible for test suites to detect the injected faults. However test suites cannot detect faults that are not injected in the part of code they test. Therefore, *cover* is used as a criterion for evaluating where faults should be injected, which in this case are parts of the code covered by available test suites.

The next section describes how the solutions were implemented, what target program was used and how the proposed solutions were validated.

2.2.3. Validation

The solutions provided in the previous section needed to be evaluated on Erlang programs in order to validate their workability. This was done by implementing a Fault Injection Tool (FIT) which executed these solutions. The FIT used Erlang *syntax_tool* [32] to traverse through the target program until it gets to a point in the code where faults can be injected. When using the *syntax_tool*, an Erlang module is transformed into a list of Erlang *syntax_tree*s [33], where each tree represents a part of the module, let's call this list a *module syntax tree*. Elements of this list could be attributes such as module name, exported functions, function definition and other parts which make up the Erlang module. Each *syntax_tree* composes of sub-trees which in turn are *syntax_trees*. Leaf of a *syntax_tree* is defined as the tree whose sub-tree is an empty list. Hence, the way to traverse through an Erlang module is using recursion to traverse deep into each syntax tree's sub trees until its leaves are reached. Faults were injected into an Erlang program by traversing through the module syntax tree until appropriate places for fault injection were found (see section 3.3).

The FIT was evaluated by injecting faults into a target program. The target program in this case was the *array* module [34] from the *Erlang OTP release R13B.* The *array* module was chosen because it came with an official pre-written test suite with 100% code coverage (see Appendix B), and developed using the widely used *Eunit unit testing framework* [35]. After using the FIT to inject faults into the target program, the output code was inspected to determine if the faults were injected or not. The *array* test suite was also evaluated by checking if it could detect the injected fault. During the fault injection and code inspections, several new faults were discovered that could be injected into the module in question. What made these faults interesting was that they couldn't be detected by the available test suite. These faults are presented in section 3.2 while solutions on how they can be injected are presented in section 3.3. The outcome of this process led to a list of validated solutions on how to inject certain faults into Erlang programs (see section 3.3).

3. Results

The aim of this study was to find out how fault injection technique can be used to evaluate Erlang test suites. In other to do this we set out to do two things: (1) find out what typical faults occur in Erlang programs by analyzing data from Erlang/OTP release notes, official Erlang documentation and Erlang bug reports, and (2) propose how these faults can be injected into Erlang programs based on our Erlang development experience and knowledge. This section presents the results of our findings based on the method utilized in section 2.

3.1. Erlang typical faults

The following sections describe Erlang typical faults; they are the validated results of the analysis carried out in Phase 1(section 2.1). These faults have been collected by going through the several different resources.

3.1.1. Failed function clause match

This fault occurs when the pattern of a function's argument does not match any clause within that function [28]. An example of this fault occurring in pro-

grams can be drawn from the bug found in Erlang/OTP R12B-5 by Matt Evans.

'The inets HTTP code does not handle HTTP status code 206 (Partial Content) responses when using streaming. Handling this is required when a server streams only part of a file (i.e., a range) and thus returns 206 rather than 200. Without this fix, on Linux the client would just block and eat 100% of the CPU.'

Official Erlang bug Reports [36]

Taking a closer look at the *inets/src/http_client/httpc_handler.erl* module in Erlang release *R12B-5*, we observed that there was actually no clause handling status code 206 (see below).

```
%% Stream to caller
stream(BodyPart, Request = #request{stream = Self},
    200) when Self == self;Self == {self,once} ->
    httpc_response:send(Request#request.from,
    {Request#request.id, stream, BodyPart}),
    {<<>>, Request};
stream(BodyPart, Request = #request{stream = File-
       name}, 200) when is_list(Filename) ->
    % Stream to file
    case file:open(Filename, [write, raw, append,
                   delayed_write]) of
         {ok, Fd} -> stream(BodyPart,
               Request#request{stream = Fd}, 200);
         {error, Reason} ->
               exit({stream_to_file_failed, Reason})
    end;
stream(BodyPart, Request = #request{stream = Fd},
       200) -> % Stream to file
    case file:write(Fd, BodyPart) of
         ok -> {<<>>, Request};
         {error, Reason} ->
               exit({stream_to_file_failed, Reason})
    end;
stream(BodyPart, Request,_) ->
       % only 200 responses can be streamed
       {BodyPart, Request}.
```

According to *Hypertext Transfer Protocol - HTTP /1.1* [37], http applications are not required to understand all registered codes but such understanding is desirable. In this case, the status code had not been recognized in *R12B-5 inets/src/http_client/httpc_handler.erl*. This led to a critical fault (blocks and consumes 100% of CPU) occurred in a Linux machine running this application. The reason why this happened by looking at the code above is that, any call received by the *stream* function that doesn't match any previous clause is caught at the shaded clause. A case where the *stream* function is called with status code 206 (e.g. `stream(BodyPart, Request = #request{stream = Fd}, 206 )` will be handled in `stream(BodyPart, Request,_)`. This will result in a wrong behavior because status code 206 should be handled differently or at least as 200 [37]. This fault was noted and fixed in Erlang release R13A, by accepting any status code passed to the stream function and handling status code 200 and 206 the same way. See code below.

```
%% Stream to caller
stream(BodyPart, Request = #request{stream = Self},
    Code) when ((Code == 200) or (Code == 206)) and
    ((Self == self) or (Self == {self,once})) ->
        httpc_response:send(Request#request.from,
         {Request#request.id, stream, BodyPart}),
{<<>>, Request};

stream(BodyPart, Request = #request{stream = Self},
    404) when Self == self; Self == {self, once} ->
       httpc_response:send(Request#request.from,
       {Request#request.id,   stream,   BodyPart}),
       {<<>>, Request};

stream(BodyPart, Request = #request{stream = File-
    name}, Code) when ((Code == 200) or (Code ==
    206)) and is_list(Filename) -> % Stream to file
       case file:open(Filename,[write, raw, append,
                      delayed_write]) of
         {ok, Fd} ->
            stream(BodyPart,Request#request{stream
            = Fd}, 200);
         {error, Reason} ->
            exit({stream_to_file_failed, Reason})
          end;

stream(BodyPart,Request=#request{stream = Fd},Code)
      when ((Code == 200) or (Code == 206)) ->
      % Stream to file
       case file:write(Fd, BodyPart) of
         ok -> {<<>>, Request};
         {error, Reason} ->
              exit({stream_to_file_failed, Reason})
          end;

stream(BodyPart, Request,_) ->
       % only 200 and 206 responses can be streamed
       {BodyPart, Request}.
```

3.1.2. Race condition

This fault occurs when accesses to a shared resource are not properly synchronized [38]. An example of this fault happening in an Erlang program can be taken from a program that was running `lists:foreach(fun erlang:garbage_collect/1, erlang:processes())` every ten minutes [39]. While this program was been tested, some abnormal behaviors such as stuck `gen_server` was discovered [40]. This led to the uncovering of a race condition fault in all R11's and R12's versions of the `smp emulator` [41]. Quoting the Erlang/OTP team, the reason the fault occurred was:

'A process being garbage collected via the `garbage_collect/1` *BIF or the* `check_process_code/2` *BIF didn't handle message receive and resume correctly during the garbage collect. When this occurred, the process returned to the state it had before the garbage collect instead of entering the new state.'*

Rickard Green, Erlang/OTP, Ericsson AB [42]

This shows that any program that runs two or more processes in parallel is capable of experiencing this type of fault if processes sharing or using the same recourses are not properly scheduled and synchronized.

3.1.3. Deadlocks

This fault occurs when two or more processes are waiting for the other to finish [26]. Deadlock was tagged a common fault in Erlang during a study on *Troubleshooting a Large Erlang System* [26]. This study involved a large industrial software project primarily developed in Erlang, where the implementation and testing phases were studied with a focus on programming errors. This project involved around 2.1 million lines of code contributed by about 300 programmers. Another study titled *Typing for Reliable Distributed Systems - Recent Advances* [27], also described deadlock as non-trivial fault in distributed and

concurrent programs such as Erlang. This fault has also been confirmed to be a typical Erlang fault from interviews conducted with several Erlang developers and a researcher. The transcripts from the interviews can be viewed in Appendix A.

3.1.4. Runaway process

Runaway process occurs when a process consumes resources (such as memory or CPU time), without doing any useful work; this is typically the result of a non-terminating loop [26]. Runaway process was tagged a common fault in Erlang during a study on *Troubleshooting a Large Erlang System* [26]. This fault has also been confirmed to be a typical Erlang fault from interviews conducted with several Erlang developers and researchers. See Appendix A for transcripts from the interviews conducted.

3.1.5. Wrong argument

This fault occurs when a function is called with an argument having a wrong data type, or when the argument is badly formed [28]. For example, a call is made to a function that receives a string and converts it to an atom, but a number is passed to it instead such as `list_to_atom(5)`. This fault has been documented as a typical fault in Erlang reference manual and has also been confirmed to be a typical Erlang fault from interviews conducted with several Erlang developers and a researcher (See Appendix A).

3.1.6. Bad argument in arithmetic expression

This fault occurs when an arithmetic expression is provided with wrong operand [28]. For example, an addition between a number and an Erlang atom such as `10 + a` will result in a fault because arithmetic addition can only be made with numeric data types such as *int*, *float*. This fault has been documented as a typical fault in Erlang reference manual and has been experienced in practice based on the interviews conducted with Erlang developers and a researcher. Refer to Appendix A for more on the interviews.

3.1.7. Failed case expression match

This fault occurs when no matching branch is found when evaluating a case expression [28]. For example, the piece of code below will result in a *failed case expression match* because `connect` will not match any of the available branches. This fault has been described as a typical fault in Erlang reference manual and has also been confirmed to be a typical Erlang fault from interviews conducted with several Erlang developers and researchers. See Appendix A for transcripts from the interviews.

Function definition: `f(A) ->`

```
case A of
    reply -> response;
    call ->   answer
end.
```

Function call: `f(connect)`

3.1.8. Failed match expression

This fault occurs when the value from the right hand side of a pattern match expression does not match with the value on the left hand side [28]. For example, the following piece of code `{name, FirstName} = {name, "John", "doe"}` will result in a *failed match expression* fault because the left hand tuple expects a tuple with an atom `name` and any other literal to be matched with it but instead gets a tuple with size three. This fault has been documented as a typical fault in Erlang reference manual and has also been confirmed to be a typical Erlang fault from interviews conducted with several Erlang developers and a researcher. See Appendix A for transcripts from the interviews.

3.1.9. Calling a non-existing function

This fault occurs when a function call is made to a non-existing function [28]. This fault has been documented as a typical fault in Erlang reference manual and has also been confirmed to be a typical Erlang fault from interviews conducted with several Erlang developers and a researcher. See Appendix A for transcripts from the interviews.

3.1.10. System limit

System limit occurs when a system limit has been reached [28]. For example if the maximum process limit of an Erlang program is 1000 as returned by `erlang:system_info(process_limit)`.Then a *system limit fault* will occur if the program tries to create more than 1000 process. Just like *Interviewee 1* said (See Appendix A), this might indeed be quite common in a not configured environment where system resources have not been properly configured and also during machine load. This fault has also been documented as a typical fault in Erlang reference manual.

3.2. Target program's faults

While the *array* module was used as the target program for evaluating the fault injection tool / solutions (see 2.2.3), several faults were discovered. These faults are presented here because this discovery shows another approach in which fault injection can be used to evaluate Erlang test suite. Apart from looking at external resources for typical faults that can be used during fault injection with the aim of evaluating the test suite of the program in question. One can also study the internals of the program for possible faults that can be injected. These faults can then be generalized to the level where they can be injected into other similar programs. The following sections present the generalized faults.

3.2.1. Omitted guard

This fault occurs when a certain guard required for a function to work correctly is missing. An example is a function that does the division between two numbers; there should be a guard to check for division by zero which leads to a fault, such as `when Y =/= 0` in the function below

```
div(X, Y) -> X / Y.
```

3.2.2. Missing Constraint

This fault arises when some constraints required by a function to work correctly is missing. An example is a function that returns the absolute of a number; there should be an *if* statement to handle the case where input is a negative number in the function `abs(X) -> X.` Such an *if* statement could be added to this function as

```
abs(X) ->
if X >= 0 -> X;
   true -> -X
end.
```

3.2.3. Under specification

This fault occurs when there is an extra constraint in a function that limits its accepted inputs. Below is an example of a function that returns the double of a number. The extra constraint `X > 0` is not needed in this case; otherwise the function will not be able to handle negative numbers.

```
double(X) when is_number(X), X > 0 -> X*2.
```

3.2.4. Swapped argument

This fault occurs in a function definition where two of its arguments are in the wrong order. Below is an example of a function that returns the weekday for the input date. The order of arguments `Month` and `Day` is not correct.

```
weekday(Year,Day,Month) ->
case calendar:day_of_the_week(Year,Month,Day) of
        1 -> "Monday";
        2 -> "Tuesday";
        3 -> "Wednesday";
        4 -> "Thursday";
        5 -> "Friday";
        6 -> "Saturday";
        7 -> "Sunday"
end.
```

3.3. Solutions for injecting typical faults into Erlang programs

This section presents the various ways of injecting faults into Erlang programs. It is the validated results from the analysis conducted in *phase 2* of the research method (see section 2.2), which includes both solutions for injecting the validated typical faults and newly discovered faults in the target *array* module.

For each fault, the solution is provided with *Solution* description on how it can be injected into the target program, the *Algorithm* for injecting the fault, an *Example* from the *array* module in the Erlang/OTP, the *Test cases* that test this part of code, the *Output of the test suite* before and after injecting the fault, and the *Meaning of test suite's outputs* that explains the reason for the result from the test cases after injecting fault in comparison to the previous one.

As mentioned in the research method (section 2.2), the solutions for injecting faults into Erlang programs should be non-trivial. This means the programs after fault injection must be compiled normally without any warnings. The fault should also be injected in covered code by checking with *cover* [31] and should not be detected by *dialyzer* [30].

The diagram below depicts an encapsulation of how *failed function clause match*, *failed case expression match* (see section 3.1), *omitted guard*, *missing constraint*, *under specification*, and *swapped argument* faults (see section 3.2) will be injected into the

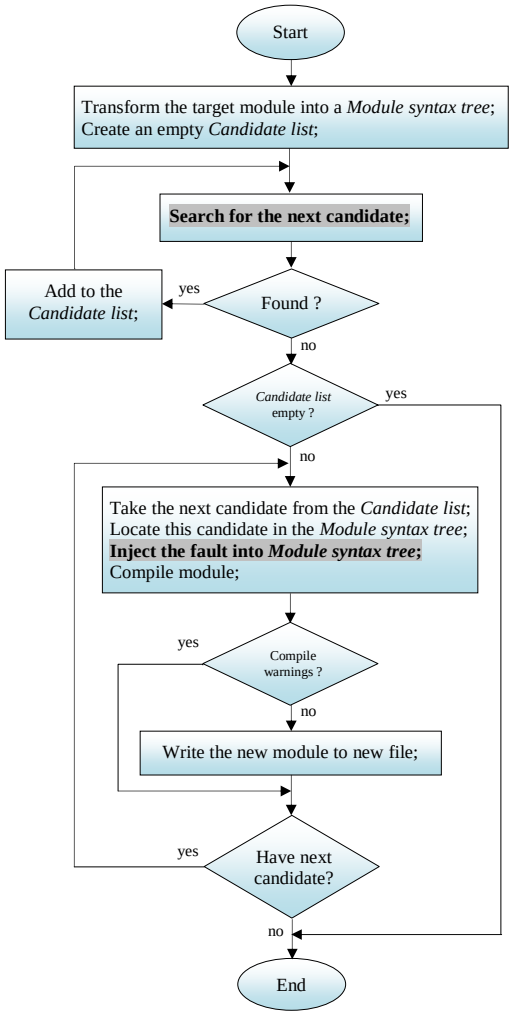

Figure 2 – Fault injection algorithm

target Erlang program (the *array* module). *Module syntax tree* is a list of *syntax_tree*s (see section 2.2.3). *Candidate* is a *syntax_tree* in the *Module syntax tree* where a particular type of fault can be injected. For example, it is a function when the fault to be injected is *failed function clause match* or an *if* statement when the fault to be injected is *missing constraint*. *Candidate list* is a list of *Candidate*s gotten from going through the *Module syntax tree*. The highlighted parts are unique for each fault injection solution and will be described in more detail under the following sections.

3.3.1. Failed function clause match

Solution

This fault is injected by removing the last function clause from a function with at least two function clauses. It is typical in Erlang that the last clause should be the one that handles all other remaining cases. Removing this will create more severe fault, which should be detected by a good test suite.

Algorithm

The algorithm for injecting this fault follows the one described in Figure 2. The highlighted parts in the figure should be replaced as in the table below.

Original parts	*Replaced parts*
Search for the next candidate;	Search for a function with at least two function clauses;
Inject the fault into *Module syntax tree*	Inject the fault by removing the last function clause;

Example

The function in the *array* module prior to injecting the *failed function clause match* fault looked like below:

```
new_1([fixed | Options], Size, _, Default) ->
    new_1(Options, Size, true, Default);
new_1([{fixed, Fixed} | Options], Size, _, Default)
  when is_boolean(Fixed) ->
    new_1(Options, Size, Fixed, Default);
new_1([{default,Default} | Options],Size,Fixed,_) ->
    new_1(Options, Size, Fixed, Default);
new_1([{size, Size} | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([Size | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([], Size, Fixed, Default) ->
    new(Size, Fixed, Default);
new_1(_Options, _Size, _Fixed, _Default) ->
                    erlang:error(badarg).
```

After injecting the fault, the highlighted function clause was removed and this function looks like:

```
new_1([fixed | Options], Size, _, Default) ->
    new_1(Options, Size, true, Default);
new_1([{fixed, Fixed} | Options], Size, _, Default)
  when is_boolean(Fixed) ->
    new_1(Options, Size, Fixed, Default);
new_1([{default, Default} | Options], Size, Fixed,
_) ->
    new_1(Options, Size, Fixed, Default);
new_1([{size, Size} | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([Size | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([], Size, Fixed, Default) ->
    new(Size, Fixed, Default).
```

Test cases

Below are some of the test cases included in the *array* module test suite. These test cases test that the function handles the task performed by the removed clause.

```
?_test(new(10)),
?_assert(new(fixed) =:= new(0)),
?_assert(new(10) =:= new([{size,0}, {size,5},
                          {size,10}])),
?_assert(17 =:= array:size(new(17))),
?_assert(is_array(new(10))),
?_test(set(9, 17, new(10))),
?_assert([undefined] =:= to_list(new(1))),
?_assert([] =:= sparse_to_list(new(1))),
?_assert([{0,undefined},{1,undefined}] =:=
                                 to_orddict(new(2))),
```

Output of test suite before injecting fault

```
All 284 tests passed.
```

Output of test suite after injecting fault

```
Failed: 41.  Skipped: 0.  Passed: 243.
```

Meaning of test suite's outputs

The injected fault was easily detected by the test suite because there were test cases covering it.

3.3.2. Failed case clause match

Solution

This fault is injected by removing the last case clause from a `case` statement with at least two case clauses. It is typical in Erlang that the last case clause usually handles the remaining cases. Removing this will create more severe fault, which should be detected by a good test suite.

Algorithm

The algorithm for injecting this fault follows the one described in Figure 2. The highlighted parts in the figure should be replaced as in the table below.

Original parts	*Replaced parts*
Search for the next candidate;	Search for a *case* statement with at least two case clauses;
Inject the fault into *Module syntax tree*	Inject the fault by removing the last case clause;

Example

The function in *the array module* prior to injecting the *failed case clause match* fault looked like below:

```
sparse_push_tuple(0, _D, _T, L) ->  L;
sparse_push_tuple(N, D, T, L) ->
  case element(N, T) of
       D -> sparse_push_tuple(N - 1, D, T, L);
       E -> sparse_push_tuple(N - 1, D, T, [E | L])
  end.
```

After injecting the fault, the highlighted clause was removed and the function looked like:

```
sparse_push_tuple(0, _D, _T, L) ->  L;
sparse_push_tuple(N, D, T, L) ->
  case element(N, T) of
       D -> sparse_push_tuple(N - 1, D, T, L)
  end.
```

Test cases

Below are the test cases included in the *array* module test suite which test the part of code where the fault was injected. The target function `sparse_push_tuple/4` was called by function `sparse_to_list/1`.

```
sparse_to_list_test_() ->
    N0 = ?LEAFSIZE,
    [?_assert([] =:= sparse_to_list(new())),
     ?_assert([] =:= sparse_to_list(new(1))),
     ?_assert([] =:= sparse_to_list(new(1,
                                  {default, 0}))),
     ?_assert([] =:= sparse_to_list(new(2))),
     ?_assert([] =:= sparse_to_list(new(2,
                                  {default, 0}))),
     ?_assert([] =:= sparse_to_list(new(N0,
                                   {default,0}))),
     ?_assert([] =:= sparse_to_list(new(N0+1,
                                   {default,1}))),
     ?_assert([] =:= sparse_to_list(new(N0+2,
                                   {default,2}))),
     ?_assert([] =:= sparse_to_list(new(666,
                                   {default,6}))),
     ?_assert([1,2,3] =:= sparse_to_list(set(2,3,
                          set(1,2,set(0,1,new()))))),
     ?_assert([3,2,1] =:= sparse_to_list(set(0,3,
                         set(1,2,set(2,1,new()))))),
     ?_assert([0,1] =:= sparse_to_list(set(N0-1,1,
                                  set(0,0,new())))),
     ?_assert([0,1] =:= sparse_to_list(set(N0,1,
                                  set(0,0,new())))),
```

```
    ?_assert([0,1] =:= sparse_to_list(set(N0+1,1,
                                set(0,0,new())))),
    ?_assert([0,1,2] =:= sparse_to_list(
     set(N0*10+1,2,set(N0*2+1,1,set(0,0,new()))))),
    ?_assertError(badarg, sparse_to_list(
                                      no_array))].
```

Output of test suite before injecting fault

```
All 284 tests passed.
```

Output of test suite after injecting fault

```
Failed: 6.  Skipped: 0.  Passed: 278.
```

Meaning of test suite's outputs

The injected fault was easily detected by the test cases. This means the test suite is effective enough in detecting the injected fault.

3.3.3. Omitted guard

Solution

This fault is injected by removing the `when` guard from a function clause of a function containing at least one guard. Even though this is particular in the *array* module, this solution can be applied to any other Erlang programs that using guard.

Algorithm

The algorithm for injecting this fault follows the one described in Figure 2. The highlighted parts in the figure should be replaced as in the table below.

Original parts	*Replaced parts*
Search for the next candidate;	Search for a function containing at least a `when` guard;
Inject the fault into *Module syntax tree*	Inject the fault by removing a `when` guard in the function;

Example

The function in the *array* module prior to injecting the *omitted guard* fault looked like below:

```
new_1([fixed | Options], Size, _, Default) ->
    new_1(Options, Size, true, Default);
new_1([{fixed, Fixed} | Options], Size, _, Default)
  when is_boolean(Fixed) ->
    new_1(Options, Size, Fixed, Default);
new_1([{default, Default} | Options], Size, Fixed,
_) ->
    new_1(Options, Size, Fixed, Default);
new_1([{size, Size} | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([Size | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([], Size, Fixed, Default) ->
    new(Size, Fixed, Default).
new_1(_Options, _Size, _Fixed, _Default) ->
                    erlang:error(badarg).
```

After injecting the fault, the highlighted guard was removed and this function looked like:

```
new_1([fixed | Options], Size, _, Default) ->
    new_1(Options, Size, true, Default);
new_1([{fixed, Fixed} | Options], Size, _, Default)
->
    new_1(Options, Size, Fixed, Default);
new_1([{default, Default} | Options], Size, Fixed,
_) ->
    new_1(Options, Size, Fixed, Default);
new_1([{size, Size} | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([Size | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([], Size, Fixed, Default) ->
    new(Size, Fixed, Default).
new_1(_Options, _Size, _Fixed, _Default) ->
                    erlang:error(badarg).
```

Test cases

Below are the test cases included in the *array* module test suite. These test cases test the function clause contains the removed guard.

```
?_test(new({fixed,true})),
?_test(new({fixed,false})),
?_test(new([{size,100},{fixed,false},
                              {default,undefined}])),
?_assert(new() =:= new([{size,0},
              {default,undefined},{fixed,false}])),
?_assert(new() =:= new(0, {fixed,false})),
?_assert(new(10, []) =:= new(10,
             [{default,undefined},{fixed,true}])),
?_assertMatch(#array{size=N0,max=N0,elements=N0},
                               new(N0, {fixed,false})),
?_assertMatch(#array{size=N01,max=N1,elements=N1},
                              new(N01, {fixed,false})),
?_assertMatch(#array{size=N1,max=N1,elements=N1},
                               new(N1, {fixed,false})),
?_assertMatch(#array{size=N11,max=N2,elements=N2},
                              new(N11, {fixed,false})),
?_assertMatch(#array{size=N2, max=N2, default=42,
 elements=N2},new(N2,[{fixed,false},{default,42}])),
?_assert(is_array(new(10, {fixed,false})))
?_assertNot(is_fix(new({fixed,false}))),
?_assertNot(is_fix(new(10, {fixed,false}))),
?_assert(is_fix(new({fixed,true}))),
?_assert(is_fix(new(10, {fixed,true}))),
?_assert(is_fix(fix(new({fixed,false})))),
?_assertError(badarg, set(10, 17, fix(new(10,
                                    {fixed,false})))),
?_assert(new(17, {fixed,false}) =:= relax(new(17))),
?_assert(new(100, {fixed,false}) =:=
              relax(fix(new(100, {fixed,false})))),
?_assert(array:size(resize(array:set(99, 0, new(10,
                          {fixed,false})))) =:= 100),
?_assert(sparse_size(array:set(99, 0, new(10,
                           {fixed,false}))) =:= 100),
```

Output of test suite before injecting fault

```
All 284 tests passed.
```

Output of test suite after injecting fault

```
All 284 tests passed.
```

Meaning of test suite's outputs

The outputs show that the injected fault was not detected by the test suite. The reason is either the test suite is not effective enough and/or there is some problem with the code. Examining the test suite confirms that there wasn't any negative test case for this function clause, i.e. test case with one of the inputs is `{fixed, Any}` while `Any` is anything other than `true` or `false`. An example of a test case which covers this and that could be included in the test suite is `?_assertError(badarg,new({fixed,any}))`. However, a closer look at the code reveals that the removed guard `when is_boolean(Fixed)` in this case is unnecessary code. In other words, this is an over-specification phenomenon where in this case the programmer was not 100% sure that the second argument of the tuple `{fixed, Value}` is always a Boolean value. In the *array* module, an array is created with either function `new/0`, `new/1` or `new/2`, which will call function `new_0/3` where the array size is either fixed or not. This will in turn call function `new_1/4` with the `Fixed` input as either `{fixed, true}` or `{fixed, false}`.

3.3.4. Missing Constraint

Solution

This fault is injected by replacing the *if* statement with one of its clauses. Even though this is specific to the *array* module, this solution can be applied to any other Erlang program that uses an *if* statement.

Algorithm

The algorithm for injecting this fault follows the one described in Figure 2. The highlighted parts in the figure should be replaced as in the table below.

Original parts	*Replaced parts*
Search for the next candidate;	Search for an *if* statement with at least two clauses;
Inject the fault into *Module syntax tree*	Inject the fault by replacing the *if* statement with one of its clauses;

Example

The function in the *array* module prior to injecting the *missing constraint* fault looked like below:

```
resize(Size,#array{size = N,max = M,elements = E}=A)
  when is_integer(Size), Size >= 0 ->
    if Size > N ->
           {E1, M1} = grow(Size-1, E,
                           if M > 0 -> M;
                              true -> find_max(N-
1, ?LEAFSIZE)
                           end),
           A#array{size = Size,
                   max = if M > 0 -> M1;
                            true -> M
                         end,
                   elements = E1};
       Size < N ->
           A#array{size = Size};
       true ->
           A
    end;
resize(_Size, _) ->
    erlang:error(badarg).
```

After injecting the fault, the highlighted code was removed and the function looked like:

```
resize(Size, #array{size = N, max = M, elements =
E}=A)
  when is_integer(Size), Size >= 0 ->
    {E1, M1} = grow(Size-1, E,
                    if M > 0 -> M;
                       true -> find_max(N-1, ?LEAF-
SIZE)
                    end),
    A#array{size = Size,
            max = if M > 0 -> M1;
                     true -> M
                  end,
            elements = E1};
resize(_Size, _) ->
    erlang:error(badarg).
```

Test cases

Below are the test cases included in the *array* module test suite. These test cases test the function that contains the replaced *if* statement.

```
resize_test_() ->
 [?_assert(resize(0, new()) =:= new()),
  ?_assert(resize(99, new(99)) =:= new(99)),
  ?_assert(resize(99,   relax(new(99)))   =:=   re-
lax(new(99))),
  ?_assert(is_fix(resize(100, new(10)))),
  ?_assertNot(is_fix(resize(100, relax(new(10))))),

  ?_assert(array:size(resize(100, new())) =:= 100),
  ?_assert(array:size(resize(0, new(100))) =:= 0),
  ?_assert(array:size(resize(99, new(10))) =:= 99),
  ?_assert(array:size(resize(99,   new(1000)))   =:=
99),

  ?_assertError(badarg, set(99, 17, new(10))),
  ?_test(set(99, 17, resize(100, new(10)))),
  ?_assertError(badarg,  set(100,  17,  resize(100,
new(10)))),

  ?_assert(array:size(resize(new())) =:= 0),
  ?_assert(array:size(resize(new(8))) =:= 0),
  ?_assert(array:size(resize(array:set(7,          0,
new()))) =:= 8),
  ?_assert(array:size(resize(array:set(7,          0,
new(10)))) =:= 8),
  ?_assert(array:size(resize(array:set(99,         0,
new(10,{fixed,false})))) =:= 100),
  ?_assert(array:size(resize(array:set(7, undefined,
new()))) =:= 0),

?_assert(array:size(resize(array:from_list([1,2,3,un
defined]))) =:= 3),

?_assert(array:size(resize(array:from_orddict([{3,0}
,{17,0},{99,undefined}])))
                =:= 18),
  ?_assertError(badarg, resize(foo, bad_argument))].
```

Output of test suite before injecting fault

```
All 284 tests passed.
```

Output of test suite after injecting fault

```
All 284 tests passed.
```

Meaning of test suite's outputs

The outputs show that the injected fault was not detected by the test suite. The reason is that either the test suite is not sufficient and/or there is some problem with the code. Examining the code exposes an "over-implementation" phenomenon in the code. In this case, the second and the last clause of the above *if* statement are not needed. The first clause already covers the second and the third ones. The new array size is always set, even when new size equals the current one. In addition, the `max` and `elements` attributes were implemented in a way that they are only changed when the new array size is greater than both the current one and the current `max` value.

While studying the test cases, it showed that only the array size was tested when the array was resized. Thus there isn't any test case testing the `max` and `elements` attributes when resizing the array with a different size. Such test cases can be written as below.

```
?_assert((resize(5,new(15,
          [{fixed,false}])))#array.max =:= (new(15,
                      [{fixed,false}]))#array.max),
?_assert((resize(5,new(15,
      [{fixed,false}])))#array.elements =:= (new(15,
                  [{fixed,false}]))#array.elements),
?_assert((resize(101,new(15,
             [{fixed,false}])))#array.max =:= 1000),
?_assert((resize(101,new(15,
         [{fixed,false}])))#array.elements =:= 1000)
```

3.3.5. Under specification

Solution

This fault is injected by adding to the `when` guard one more constraint that limits the accepted input of a function. Even though this is specific to the *array* module, this solution can be applied to any other Erlang program that does comparison with a guard.

Algorithm

The algorithm for injecting this fault follows the one described in Figure 2. The highlighted parts in the figure should be replaced as in the table below.

Original parts	*Replaced parts*
Search for the next candidate;	Search for a function containing at least a comparison guard (e.g. N > 100);
Inject the fault into *Module syntax tree*	Inject the fault by adding to the comparison guard one more constraint that limits the accepted input;

Example

The function in the *array* module prior to injecting the *under specification* fault looked like below:

```
new(Size, Options) when is_integer(Size), Size >= 0
->
   new_0(Options, Size, true);
new(_, _) ->
   erlang:error(badarg).
```

After injecting the fault, the highlighted constraint was added and the function looked like:

```
new(Size, Options) when is_integer(Size), Size >= 0,
Size =< 1000 ->
   new_0(Options, Size, true);
new(_, _) ->
   erlang:error(badarg).
```

Test cases

Below is the test case included in the *array* module test suite. This is the only test case that tests the target function clause.

```
-define(LEAFSIZE, 10).
-define(NODESIZE, ?LEAFSIZE).

N0 = ?LEAFSIZE,
N1 = ?NODESIZE*N0,
N2 = ?NODESIZE*N1,

?_assertMatch(#array{size=N2, max=N2,
                     default=42,elements=N2},
  new(N2, [{fixed,false},{default,42}])),
```

Output of test suite before injecting fault

```
   All 284 tests passed.
```

Output of test suite after injecting fault

```
   All 284 tests passed.
```

Meaning of test suite's outputs

The outputs show that the injected fault was not detected by the test suite. This is because there is no test case that verifies an array can be created with a size more than 1000.

3.3.6. Swapped arguments

Solution

This fault is injected by swapping two arguments of a function containing more than one argument. As a minimum, one of the arguments must be unused, i.e. it starts with the "_" sign. Even though this is specific to the *array* module, this solution can be applied to any other Erlang programs that contain a function clause with unused arguments.

Algorithm

The algorithm for injecting this fault follows the one described in Figure 2. The highlighted parts in the figure should be replaced as in the table below.

Original parts	*Replaced parts*
Search for the next candidate;	Search for a function with at least two arguments where one of them must be unused;
Inject the fault into *Module syntax tree*	Inject the fault by swapping the unused argument with any other one;

Example

The function in the *array* module prior to injecting the *swapped argument* fault looked like below:

```
new_1([fixed | Options], Size, _, Default) ->
    new_1(Options, Size, true, Default);
new_1([{fixed, Fixed} | Options], Size, _, Default)
  when is_boolean(Fixed) ->
    new_1(Options, Size, Fixed, Default);
new_1([{default,Default} | Options],Size,Fixed,_) ->
    new_1(Options, Size, Fixed, Default);
new_1([{size, Size} | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([Size | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([], Size, Fixed, Default) ->
   new(Size, Fixed, Default);
new_1(_Options, _Size, _Fixed, _Default) ->
    erlang:error(badarg).
```

After injecting the fault, the highlighted arguments were swapped and the function looked like:

```
new_1([fixed | Options], Size, _, Default) ->
    new_1(Options, Size, true, Default);
new_1([{fixed, Fixed} | Options], Size, _, Default)
  when is_boolean(Fixed) ->
    new_1(Options, Size, Fixed, Default);
new_1([{default,  Default}  |  Options],  Size,  _,
  Fixed) ->
    new_1(Options, Size, Fixed, Default);
new_1([{size, Size} | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([Size | Options], _, _, Default)
  when is_integer(Size), Size >= 0 ->
    new_1(Options, Size, true, Default);
new_1([], Size, Fixed, Default) ->
   new(Size, Fixed, Default);
new_1(_Options, _Size, _Fixed, _Default) ->
    erlang:error(badarg).
```

Test cases

Below are some of the test cases included in the *array* module test suite. These test cases test the function clause contains the swapped arguments.

```
?_test(new({default,undefined})),
?_test(new([{size,100},{fixed,false},{default,undefi
ned}])),
?_test(new([100,fixed,{default,0}])),
?_assert(new(10,     [])     =:=     new(10,     [{de-
fault,undefined},{fixed,true}])),
?_assertError(badarg, new([{default,0} | fixed])),
?_assertMatch(#array{size=N2,       max=N2,       de-
fault=42,elements=N2},
                    new(N2,
[{fixed,false},{default,42}])),
?_assert(4711 =:= default(new({default,4711}))),
?_assert(0 =:= default(new(10, {default,0}))),
?_assert(array:get(0, new(1,{default,0})) =:= 0),
?_assert(array:get(0,  reset(0,  new({default,42})))
=:= 42),
?_assert(array:get(0,     reset(0,     set(0,     17,
new({default,42})))) =:= 42),
```

Output of test suite before injecting fault

```
All 284 tests passed.
```

Output of test suite after injecting fault

```
All 284 tests passed.
```

Meaning of test suite's outputs

The outputs show that the injected fault was not detected by the test suite. The reason is that either the test suite is not effective enough and/or there is some problem with the code. Examining the test suite gave an expression that the following test case was missed:

```
?_assert(new([{default, 5}, 20, fixed]) =:=
                    new([20 ,fixed ,{default, 5}])).
```

However, a closer look at the code reveals that the Boolean variable Fixed was implemented to accept any value rather than just `true` or `false`. That code was written as:

```
if Fixed -> 0;
   true -> E
end,
```

Although according to the local specification, variable `Fixed` can only be `true` or `false`, it would also be better to write the code to accept only these values. This is proved by having the test result of 48 test cases failed with the replaced code when the fault was injected. Such a code can be written as:

```
case Fixed of
     true -> 0;
     false -> E
end,
```

4. Discussion

The aim of this study is to show how fault injection technique can be used to evaluate Erlang test suites. A qualitative approach with data collection, analysis and validation phases was adopted. We will discuss certain reasons behind some decisions that were made, some issues that occurred along the way, how things could have been done differently and so on. We will also touch on some interesting points and findings in the results of this study.

4.1. Approach

Several data sources were used during data collection, these includes the official Erlang OTP releases, Erlang-bug archives, Erlang reference manual and related studies (see table 1). However the original plan was to gather data from only the official Erlang OTP release R10B – 0 to R13B by comparing the source codes of all neighboring releases (e.g. R11B-0 and R10B-10) with the aim of locating what bug fixes were found or fixed from previous releases. This was one way of gathering Erlang faults, but we had several difficulties while using this approach. One problem was that the information available in the release notes on what bug fixes were made, was not detailed enough to relate to what piece of code or module it occurred in. This left room for a lot of uncertainty on the root cause of documented bugs. An example of this lack of detail can be seen below:

```
--- asn1-1.6.10 ----------------------------------

    OTP-7953  The anonymous part of the decode that
splits the ASN1 TLV into Tag Value tuples has been
optimized.

    OTP-7954  A faulty receive case that catched all
messages in the initialization of the driver has
been removed, the initialization has been restruc-
tured.
```

R13B Release note [43]

Another problem was that Erlang consist of a number of applications. Hence, one must have some familiarity with all the applications in each release in order to easily locate where the bugs occurred based on the insufficient information available in the release notes. This will take much longer time than the period of ten weeks used for this study. Therefore, focusing on more data sources made it easier to gather Erlang typical faults especially since there were already some studies in this field and also some official documentation on Erlang typical faults available (see section 2.1.2).

The solutions that were proposed on how to inject faults into Erlang programs were validated by building a fault injection tool that implemented those solutions (2.2.3). This not only certified the solutions as valid but also showed how fault injection can be automated. This automation is particularly useful when it comes to using fault injection with larger programs that have many lines of codes. It is also useful because it can be reused on several Erlang programs. This way of validating was however costly for this study since one has to develop a tool which requires a reasonable amount of development time and the technical knowhow.

The purpose of this study was to understand how *fault injection* technique can be used to evaluate Erlang test suites. This was approached by first finding out what typical faults occur in Erlang programs by analyzing data from Erlang sources such as Erlang/OTP release notes, official Erlang documentation and Erlang bug reports and other related studies. Proposals were then made on how non trivial faults within them could be injected into Erlang programs. This approach has produced meaningful results and has been successful in this study. However, there are some drawbacks when it comes to looking for faults that can be used for fault injection. Majority of the typical faults gathered were eventually not used for fault injection (see section 3.3). This was because most of them were either trivial or not suitable for fault injection, and thus were not part of those used to evaluate Erlang test suites at the end. A better approach would have been gathering not just Erlang typical faults, but faults that are ideal for fault injection from the very beginning.

It is interesting to see that all the typical faults collected (see section 3.1) were actually detected by the test suite of the *array* module while the faults discovered when working with the program (see section 3.2) went undetected. This shows, according to this study,

that an effective way to inject faults which might be missed by the test suite is by having internal knowledge of the target program. Even though these undetected faults have been generalized to the point where they can be injected into other related Erlang programs, it still remains uncertain whether they will not be easily detected.

Having this in mind, another approach that can be used in carrying out fault injection, is by manually injecting faults into a target Erlang program. These faults can then be generalized to the level where they can be injected into other similar programs. This process of manually injecting faults can be automated by using a fault injection tool. Automating the process makes it a lot easier and less time consuming when injecting faults into many different other programs. It also reduces the risk of incorrect fault injection due to human error. This approach also has its drawback as the generalization made here, are less suitable for programs that are not similar to the target one. For example, if the target program is not database oriented, then it might be difficult to inject faults which are typical in database oriented programs. Thus, selecting different target programs from different domains might be a good idea when gathering faults that will be generalized. This will make the generalization applicable to a wider range of different Erlang programs

4.2. Typical Faults

The reason for injecting typical faults is that they are faults that can be found in Erlang programs and there is a high probability of it occurring during and after the development time of the program. An example of this is the *failed function clause match* discovered in Erlang/OTP release that was written by experienced developers (see section 3.1).

4.3. Fault Injection Solutions

In order for a fault to occur in an Erlang program, conditions that cause the fault to arise must be present in that program. For example a *deadlock* fault cannot occur in a program that runs on just one process. This finding means that the choice of what typical faults that can be injected into a program depends on how the program is constructed.

We have chosen to inject non-trivial Erlang typical faults from the ones described in section 3.1. Thus, typical faults such as *wrong argument* were not injected because they could be easily detected and evaluated by already available tools such as static analyzers (e.g. *dialyzer*) and coverage tools (e.g. *cover*). Faults discovered while evaluating the fault injection tool on the target *array* module were also injected (see 3.2). This was however generalized so that they can also be injected into other similar Erlang programs.

5. Conclusion

Fault injection is a technique that involves injecting abnormalities into software programs [1]. This can then be used to evaluate test suites by checking how well they detect those abnormalities. Test suite is a set of test cases created to test a particular program with the purpose of finding faults that exist in that program. A test suite is effective if it is able to detect errors that exist in its target program. The more errors it detects, the more effective it is. This study showed how fault injection can be used to evaluate Erlang test suites. This was done by (1) injecting non-trivial faults one at a time into a target Erlang program, these are faults that cannot be detected at compile time, by *dialyzer* or by a test suite and *cover* information, and (2) evaluating the program test suite by studying if it can identify the injected fault, and if not why.

We applied fault injection on the *array* module in the *Erlang OTP release R13B*, and evaluated its prewritten test suite. The evaluation was carried out by injecting six non trivial faults, one at a time and checking if they can be detected by the test suite. Out of the six faults injected, two were detected by the test suite while four went undetected. A thorough study of the code where the faults were injected and the test cases covering those revealed two things: some missing test cases and some program code in need of improvement. However, the overall evaluation showed that the evaluated test suite was effective enough in detecting faults in the target *array* module.

One very important part of fault injection is having the right fault to inject into the target program. Nevertheless, it is not possible to know the right faults to inject for every individual program; therefore it is necessary to inject as many faults as possible. We have been able to come up with some typical Erlang fault during the course of this study. However, there is still need to explore more resources for more faults which can be used for fault injection. Further research could focus more on this.

6. Acknowledgement

We would like to say thanks to our supervisor, Thomas Arts for the reviews and support we got during the course of the study.

7. References

[1] Jeffery M. Voas and Gary McGraw, 1998, *Software fault injection: inoculating programs against errors*, New York: Wiley

[2] Matthew J. Rutherford et al., 2008, *Evaluating Test Suites and Adequacy Criteria Using Simulation-Based Models of Distributed Systems*, IEEE Transactions on Software Engineering

[3] Mei-Chen Hsueh et al.,1997, *Fault Injection - Techniques and tools*, IEEE Computer Society Press Los Alamitos, CA, USA

[4] Sébastien Tixeuil et al., 2005, *A language-driven tool for fault injection in distributed systems*, Grid Computing Workshop

[5] Sudipto Ghosh, 2001, *Fault Injection Testing for Distributed Object Systems*, TOOLS39, IEEE Computer Society Washington, DC, USA
[6] Chillarege, R. Bowen, N.S., 1989, *Understanding large system failures-a fault injection experiment*, IBM - NY
[7] Jean Arlat et al., 1992, *Fault Injection and Dependability Evaluation of Fault-Tolerant Systems*, Technical Report: LAAS-CNRS#91260, University of Bologna
[8] Nik Looker. et al., 2005, *A Comparison of Network Level Fault Injection with Code Insertion*, IEEE Computer Society Washington, DC, USA
[9] Scott Dawson et al., 1995, *A software fault injection tool on real-time Mach*, IEEE Computer Society Washington, DC, USA
[10] Seungjae Han et al., 1995, *DOCTOR: An integrated software fault injection environment for distributed real-time systems*, IEEE Computer Society Washington, DC, USA
[11] Sébastien Tixeuil et al, 2006, *An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids*, 2nd CoreGRID Workshop on GRID and Peer to Peer Systems Architecture, Paris
[12] Michel Cukier et al, 1999, *Fault Injection Based on a Partial View of the Global State of a Distributed System*, IEEE Computer Society Washington, DC, USA
[13] Jeffrey Voas, 1997, *Fault Injection for the Masses*, IEEE Computer Society Press Los Alamitos, CA, USA
[14] Jolo V. Carreira et al., 1999, *Fault Injection Spot-Checks Computer System Dependability*, IEEE Spectrum
[15] Jeffrey A. Clark and Dhiraj K. Pradhan , 1995, *Fault injection: A method for validating computer-system dependability*, IEEE Computer Society
[16] Douglas M. Blough and Tatsuhiro Torii, 1997, *Fault-Injection-Based Testing Of Fault-Tolerant Algorithms In Message-Passing Parallel Computers*, IEEE Computer Society Washington, DC, USA
[17] Ghani A. Kanawati et al, 1995, *FERRARI: A Flexible Software-Based Fault and Error Injection System*, IEEE Computer Society Washington, DC, USA
[18] Timothy K. Tsai and Ravishankar K. Iyer, 1995, *FTAPE: A Fault Injection Tool to Measure Fault Tolerance*, National Aeronautics and Space Administration, Washington, D.C
[19] Scott Dawson et al., 1996, *ORCHESTRA: A probing and fault injection environment for testing protocol implementations*, Computer Performance and Dependability Symposium, 1996., Proceedings of IEEE International
[20] Thomas M. Galla et al, 2004, *Software Implemented Fault Injection for Safety-Critical Distributed Systems by Means of Mobile Agents*, Proceedings of the 37th Hawaii International Conference on System Sciences, IEEE Computer Society Washington, DC, USA
[21] Scott Dawson et al, 1996, *Testing of fault-tolerant and real-time distributed systems via protocol fault injection*, Proceedings of FTCS-26, IEEE Computer Society Washington, DC, USA
[22] Erlang/OTP, 2009, http://erlang.org/index.html, last accessed 2009-04-22
[23] Ericsson, 2009, http://www.ericsson.com/ last accessed 2009-06-08
[24] Erlang/OTP releases, 2009, http://erlang.org/download.html, last accessed 2009-04-22
[25] Official Erlang bugs archives, 2009, http://www.erlang.org/pipermail/erlang-bugs/, last accessed 2009-05-19
[26] Mats Cronqvist, 2004, *Troubleshooting a Large Erlang System*, Erlang'04, ACM New York, NY, USA.
[27] Paweł T. W., 2005, *Typing for Reliable Distributed Systems - Recent Advances*, DSN-2005 IEEE Workshop on Dependable Software - Tools and Methods, Yokohama, Japan.
[28] Official Erlang Reference Manual, 2009, http://erlang.org/doc/, last accessed 2009-05-19.
[29] John W. Creswell, 2008, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, Sage Publications Inc
[30] Dialyzer, 2009, http://erlang.org/doc/apps/dialyzer/dialyzer_chapter.html,last accessed 2009-05-31
[31] Cover, 2009, http://www.erlang.org/doc/apps/tools/part_frame.html last accessed 2009-06-08
[32] Erlang Syntax Tool, 2009, http://www.erlang.org/doc/apps/syntax_tools/index.html last accessed 2009-05-21.
[33] Erlang Syntax Tree, 2009, http://www.erlang.org/doc/apps/syntax_tools/index.html last accessed 2009-06-24.
[34] The array module, 2009, http://erlang.org/doc/man/array.html, last accessed 2009-07-02
[35] Eunit, 2009, http://www.erlang.org/doc/apps/eunit/part_frame.html last accessed 2009-05-21

[36] Erlang Bug Report, 2009 http://www.erlang.org/pipermail/erlang-bugs/2009-February/001216.html last accessed 2009-06-02
[37] HTTP/1.1, 2009, http://www.w3.org/Protocols/rfc2616/rfc2616.html last accessed 2009-05-21.
[38] Robert H. B. Netzer and Barton P. Miller, 1992, *What are race conditions?: Some issues and formalizations*, ACM New York, NY, USA.
[39] Erlang Bug Report, 2009, http://www.erlang.org/pipermail/erlang-bugs/2009-January/001159.html last accessed 2009-06-07
[40] Erlang Bug Report, 2009, http://www.erlang.org/pipermail/erlang-bugs/2009-January/001158.html last accessed 2009-06-07
[41] Erlang Bug Report, 2009, http://www.erlang.org/pipermail/erlang-bugs/2009-January/001168.html last accessed 2009-06-07
[42] Erlang/OTP, 2009, http://www.erlang.org/download/patches/otp_src_R12B-5_OTP-7738.readme, last accessed 2009-06-07
[43] R13B Release note, 2009, http://www.erlang.org/download/otp_src_R13B.readme , last accessed 2009-06-24
[44] Brian W.Kernighan and Dennis M. Ritchie, 1978, *The C Programming Language, 1st edition*, Englewood Cliffs, NJ: Prentice Hall
[45] Java, 2009, http://www.java.com/en/download/whatis_java.jsp , last accessed 2009-07-02

Appendix A – Interviews

This interview has been conducted separately with 3 experienced Erlang developers and an academic researcher within the field of Erlang. We haven't requested for their names to be published in this article, thus their names will be given as interviewee 1, 2, 3 and 4. The interview question was not open because we were not trying to explore the problem area (what are Erlang typical faults) but rather to validate findings that we already have on Erlang typical faults.

The question was "*Are the following faults (I – XVIII) Erlang typical faults*?"

Fault I

Race condition - This fault occurs when accesses to the shared resource are not properly synchronized.	
Answers	
Interviewee 1	Race conditions are very common as soon as you try to do anything which involves concurrency. I would however think that inserting race conditions is quite hard, since you would need to identify them to be able to provoke them deliberately. If you've identified them, it should be possible to fix them. You can however change timing aspects during runtime I guess.
Interviewee 2	Yes, I found this problem usually early stage of a bigger project, or adding new features to a complex system. It was quite rare, if the design was good before.
Interviewee 3	Yes, this is a rather common and important type of error. This type of error is very hard to find in unit-tests and often shows up late in the development process. But I fail to see how that could relate to fault injection!?
Interviewee 4	Happens occasionally, can be difficult to find as it can seem intermittent.

Fault II

Deadlocks - This fault occurs when two or more processes are waiting for the other to finish.	
Answers	
Interviewee 1	Yes and no. I wouldn't say that Deadlocks are common in any Erlang system written by an experienced Erlang programmer. I've however experienced deadlocks when interacting with databases, trying to dispatch table locknig requests over OS threads, to avoid locking the Emulator. A verry common newbee mistake would be to go a gen_server:call(self(), whatever) inside any callback function, but this is very quickly identified and usually not repeated.
Interviewee 2	Yes, I had this problem a few times, in bigger projects it is usually time consuming to debug the reason.
Interviewee 3	Yes
Interviewee 4	Happens occasionally, but normally easy to find & correct.

Fault III

Runaway process - This fault occurs when a process consumes resources (such as memory or CPU time), without doing any useful work. Typically this is the result of a non-terminating loop.	
Answers	
Interviewee 1	I haven't seen this too much to be honest, but I've hard quite recently about this happening in one of our production systems :) In this case it was a badly formulated guard. Thing fibonacci without checking if input data is negative.
Interviewee 2	Yes, this happens sometimes, an other example not to terminate unused listeners (processes only waiting for input messages and forward them after some work).
Interviewee 3	Yes
Interviewee 4	Happens occasionally, but normally easy to find & correct.

Fault IV

Wrong arguments - This fault occurs when a function is called with wrong data type of the argument, or the argument is badly formed.	
Answers	
Interviewee 1	Yes, extremely common.
Interviewee 2	Yes, this is one of the most common problems when extending an already existing code. I usually make this error when writing a big code part, through several modules, and I forget to update the return values of a function at function call from the other module.
Interviewee 3	Not very often
Interviewee 4	Common enough. I mostly do it when using functions with nested arguments, lists of tagged tuples that contain lists of...

Fault V

Bad argument in arithmetic expression - This fault occurs when an arithmetic expression is provided with bad arguments.	
Answers	
Interviewee 1	Quite common. Good example is timouts, which can usually be an integer or the atom infinity.
Interviewee 2	No, for me usually this is not a typical error, but this can depend on the code written.
Interviewee 3	No
Interviewee 4	This happens regularly but is normally found very quickly if in the local module. It can go undetected if it's used in a library function that doesn't use guards.

Fault VI

Failed match expression - This fault occurs when result from the right hand side of a pattern matches expression does not match with pattern of the left one.	
Answers	
Interviewee 1	Yes this is quite common. Mostly during development or testing though.
Interviewee 2	Yes, one of the most typical error. Especially after extending existing code, when the right hand side is a result of a function call, what changed.
Interviewee 3	Yes, mostly because one has changed the format of a record or tuple.
Interviewee 4	The "badmatch", probably the most common basic error I've seen.

Fault VII

Failed function clause match - This fault occurs when argument's pattern of a function call does not match any clause of that function.	
Answers	
Interviewee 1	Yes this is quite common, but also the easiest to debug, since there is very much information available :)
Interviewee 2	Yes, really typical error, very common.
Interviewee 3	Yes
Interviewee 4	This occurs regularly enough. Mostly when calling modules from other applications or library functions.

Fault IX

Failed case expression match - This fault occurs when no matching branch is found when evaluating a case expression.	
Answers	
Interviewee 1	Yes, quite common, unless ppl. tend to use an Other clause in the end.
Interviewee 2	Yes, really typical error, very common.
Interviewee 3	No, its a trivial code-coverage problem
Interviewee 4	This occurs regularly enough but most designers have a catch-all default case at the end of their statements.

Fault X

Failed if expression match - This fault occurs when none of the guards in an if expression evaluated to true.	
Answers	
Interviewee 1	Less common, probably since the statement itself is less common. It is often used as "if this is tue do that, otherwise nothing, so there is usally a true -> ok clase in the end...
Interviewee 2	Yes, typical error, but not very common.
Interviewee 3	No, it's a trivial code-coverage problem
Interviewee 4	Don't think I've seen this one. Most designers handle the catch-all 'else' with some default behaviour.

Fault XI

Failed try expression match - This fault occurs when no matching branch is found when evaluating a try expression.	
Answers	
Interviewee 1	Not very common. But then we don't use try very much.
Interviewee 2	Yes, typical error, common one.
Interviewee 3	No, it's a trivial code-coverage problem
Interviewee 4	I've seen this occasionally.

Fault XII

Calling a non-existing function - This fault occurs when a function call is made to a non-existing function.	
Answers	
Interviewee 1	Quite common as a result of a typo. Can be caught easily with testcases / dialyzer though.
Interviewee 2	Yes, it happened a few times, usually not during new development, but extending old codebase.
Interviewee 3	Yes, but it is an easy to find problem and an easy to fix problem, thus it is not very interesting from a fault perspective.
Interviewee 4	I've done this when coding but normally find it very quickly.

Fault XIII

Faulty fun - This fault occurs when there is something wrong with a fun.	
Answers	
Interviewee 1	No, not really.
Interviewee 2	Yes, it happens, but very rare.
Interviewee 3	Too vague
Interviewee 4	That description might be a little vague, if I ever found a problem in a fun I'd probably classify it as a case clause, wrong arguments or whatever other heading it might fall under. The fact that it's in a fun isn't the root cause.

Fault XIV

Wrong number of arguments applied to a fun - This fault occurs when wrong number of arguments is applied to a fun.	
Answers	
Interviewee 1	No not really.
Interviewee 2	Yes, it happened, but very rare.
Interviewee 3	No, it's a trivial code-coverage problem
Interviewee 4	I've seen this delivered in systems long after it should have been found. Depending on how the function behaves its' not as easy to test for as it first appears. Normally occurs when an API has changed.

Fault XV

Time out value - This fault occurs when the timeout value in a receive..after expression is evaluated to something else than an integer or infinity.

Answers	
Interviewee 1	Not so common, but it is possible to have a negative value if you decrement a timeout in a loop, which would give an error.
Interviewee 2	Yes, it happened, but very rare.
Interviewee 3	Again, it's a trivial code-coverage problem
Interviewee 4	I've never seen this one, but I'm sure it happens ;-)

Fault XVI

Unavailable process - This fault occurs when trying to link to a non-existing process.	
Answers	
Interviewee 1	No, not very common. We usually spawn_link anyway. Or add monitors.
Interviewee 2	Yes, it happened, usually in big systems, running the system. During some non-expected rare scenarios, after some failover, some process still try to link a non-existing one.
Interviewee 3	Linking to a non-existing process cannot be considered an error, it is something that normally happens in a fault-tolerant system.
Interviewee 4	I've seen this good few times. Normally happens when one process has crashed or a start-up sequence isn't right.

Fault XVII

Evaluating a throw outside a catch - This fault occurs when trying to evaluate a throw outside a catch.	
Answers	
Interviewee 1	Never seen :)
Interviewee 2	Yes, very rarely, but happened.
Interviewee 3	It's a trivial code-coverage problem
Interviewee 4	I never use throws in my code unless I absolutely have to so I've not seen this one before. Might not recognise it in someone else's code as a result.

Fault XVIII

System limit - This fault occurs when a system limit has been reached.	
Answers	
Interviewee 1	Yes, this is quite common in a not configured environment.
Interviewee 2	Yes, usually during the first (load) testing the erlang system different system limits are reached, it happens during later (load) tests, but not so frequent.
Interviewee 3	No
Interviewee 4	Yes, I've seen this under load a few times. I've often wondered how to handle it in SW, how to reliably detect that the machine is under load and how best to reject new jobs.

Appendix B – Coverage for the array module

Below is the output of the coverage analysis conducted on the target array module. Cover coverage analysis tool was used to analyze if the array module test suite covers all code parts and lines. The output shows that all code parts and lines are covered.

```
Eshell V5.7.1  (abort with ^G)
1> cover:compile(array).
{ok,array}
2> eunit:test(array).
  All 284 tests passed.
ok
3> cover:analyze(array, coverage, line).
{ok,[{{array,0},{0,1}},
     {{array,184},{1,0}},
     {{array,228},{1,0}},
     {{array,249},{1,0}},
     {{array,251},{1,0}},
     {{array,254},{1,0}},
     {{array,256},{1,0}},
     {{array,259},{1,0}},
     {{array,262},{1,0}},
     {{array,264},{1,0}},
     {{array,267},{1,0}},
     {{array,270},{1,0}},
     {{array,272},{1,0}},
     {{array,274},{1,0}},
     {{array,277},{1,0}},
     {{array,278},{1,0}},
     {{array,279},{1,0}},
     {{array,281},{1,0}},
     {{array,286},{1,0}},
     {{array,288},{1,0}},
     {{array,301},{1,0}},
     {{array,303},{1,0}},
     {{array,315},{1,0}},
     {{array,316},{1,...}},
     {{array,...},{...}},
     {{...},...},
     {...}|...]}
4> cover:analyze(array, coverage, module).
{ok,{array,{658,1}}}
5>
```