Document downloaded from:

http://hdl.handle.net/10251/159137

This paper must be cited as:

Pérez-Rubio, S.; Tamarit Muñoz, S. (2019). Enhancing POI Testing Approach through the Use of Additional Information. Lecture Notes in Computer Science. 11285:74-90. https://doi.org/10.1007/978-3-030-16202-3_5

The final publication is available at

https://doi.org/10.1007/978-3-030-16202-3_5

Copyright Springer-Verlag

Additional Information

Enhancing POI testing through the use of additional information *

Sergio Pérez and Salvador Tamarit

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera s/n
E-46022 València, Spain
{serperu,stamarit}@dsic.upv.es

Abstract. Recently, a new approach to perform regression testing has been defined: the point of interest (POI) testing. A POI, in this context, is any expression of a program. The approach receives as input a set of relations between POIs from a version of a program and POIs from another version, and also a sequence of entry points, i.e. test cases. Then, a program instrumentation, an input test case generation and different comparison functions are used to obtain the final report which indicates whether the alternative version of the program behaves as expected, e.g. it produces the same outputs or it uses less CPU/memory. In this paper, we present a method to improve POI testing by including additional context information for a certain type of POIs. Concretely, we use this method to obtain an enhanced tracing of calls. Additionally, it enables new comparison modes and a categorization of unexpected behaviours.

Keywords: code evolution control, automated regression testing, call traces, tracing

1 Introduction

During its useful lifetime, a program might evolve many times. Each evolution is often composed of several changes that produce a new release of the software. Software developers usually define test suites to detect any *unexpected behaviour* (UnB) in new program releases. These suites help developers to notice the errors, but detecting an error is just the beginning of the debugging process.

* This work has been partially supported by MINECO/AEI/FEDER (EU) under grant TIN2016-76843-C4-1-R, and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic). Salvador Tamarit was partially supported by the *Conselleria de Educación, Investigación, Cultura y Deporte de la Generalitat Valenciana* under grant APOSTD/2016/036.

Let us suppose that we are running our test suite and some test fails. Our next step would probably be to start a debugging process in order to find the bug. This process requires our knowledge and cannot be done without it. We are going to start observing intermediate results until the origin of the bug is found. We need to interpret each one of these intermediate results, until we locate the one that makes no sense. However, in this process there is a lot of information that is available, but is not usually exploited by users. All this information comes from previous versions of the code which, in a continuous integration scheme, will be very similar to the latest version or exactly the same.

First of all, the mentioned information can be used to establish relations between the expressions of the last version of the code and its predecesor. Once these relations are created, we can run the failing test using as observation point all the expressions that form these relations. We should run the test on both versions, and observe and compare the results obtained on each of these auto-generated observation points[1]. However, most of the times, the compared values are going to be very similar, so why do not automatise the comparison process as well? These principles (expression relation across versions and comparison of execution results) are the basis of *Point of interest* (POI) testing (briefly described in Section 2).

POI testing tries to help users in comparing the behaviour of their code across versions by observing specific points of a program. A POI can be any expression of the code whose behaviour wants to be observed, e.g. the POI `(module, 5, (var, 'A'), 2)` refers to the second occurrence[2] of variable A in the fifth line of the file `module`. We can establish these relations between points of both programs versions and seach for errors with our defined tests suites but, what happens if the observed points does not reveal any incoherence when running our test suite? or, even worse, what happens if we do not have any test suite? This is also solved in our approach by auto-generating (a lot of) test cases that focus on the observation points (POIs) when deciding what to use as concrete data inside the test. Thus, it enables the comparison of versions without needing a user-defined test-suite or reuses the existing one to create similar ones that explore the behaviour of the POIs for different program executions.

POI testing, as mentioned previously, can be a very powerful tool that eases the debugging process and reuses a lot of information that is available in common repositories, such as Git repositories. However, its first and current definition does not use all the available information when running a test. For this reason, the user needs to do some extra work to locate the source of a bug. In this paper we present a framework that allows to reuse more available information to ease the task of the user in the debugging process (Section 3). As an example, we present a concrete usage of the framework by increasing the information provided by POIs placed at function calls (Section 4) in the context of the functional

[1] Although current implementation of POI testing does not still support auto-generation of observation points, it is a key feature that will be part of next release.

[2] The occurrence argument (2 in the example) can be omitted, selecting in that case the first occurrence of the variable in the indicated line.

programming language Erlang[3] [1]. Let's see Example 1 to better understand the framework and its usage.

Example 1. Consider the code (in any specific language) shown in Figure 1 which represents three lines of a program that show some differences between versions.

```
PREVIOUS        NEW
x = g()         x = h()      → Expression changed, no relation built.
y = i()         y = i()      → Expression unchanged, relation built.
lib.f(x,y)      lib.f(x,y)   → f is in a library (lib) that has completely
                               changed due to some severe refactoring. Re-
                               lation built.
```

Fig. 1: Two different versions of a source code

In the new version, a severe refactoring on function `f` has been done. Suppose that in the new version `h()` computes the value 5, while `g()` computes 4. Then, when running our test suite, an error raises. Then we run POI testing, which builds the relations between code versions as explained in Figure 1.

This means that, until comparing the values computed by each call to `f`, we would not know that the call is uncovering the UnB. Our first reaction would be to blame function `f` and start the debugging process for `f`, comparing completely different versions of the same algorithm, that stands a hard and arduous task. The framework presented in this paper can be used to add to our POI testing the knowledge of the call arguments. These arguments, are integrated as another expression in the plain POI testing, so they can be used when comparing values computed on the POIs across versions.

2 POI testing

In POI testing, (i) the programmer identifies a POI and a set of *entry points*. Then, by using some automatic test case generation technique, (ii) the approach automatically generates a test suite which tries to cover all possible paths that reach the POI (trying also to produce execution paths that evaluate this POI several times). Therefore, in POI testing, the *input of a test case* (ITC) is defined as a call to a particular function (defined as an entry point) with some specific arguments. On the other hand, the output of a test case is the sequence of values that the POIs are evaluated to for an ITC. For the sake of simplicity, in the rest of the paper we use the term *traces* to refer to these sequences of values. Next, (iii) the test suite is used to automatically check whether the behaviour of the program remains unchanged across new versions[4]. Finally, (iv) the user is provided with a report about the success or failure of these test cases.

[3] More information about Erlang and how we implemented a POI tester for this language are further discussed in Section 4.

[4] Steps (ii) and (iii) could also be executed in parallel. Here, for the sake of simplicity, we only consider the sequential execution of these steps.

Note that this technique allows the definition of multiple POIs [9]. With this feature, users can trace several (and maybe unrelated) functionalities in a single run. Additionally, users can strengthen the quality of their test suite by checking behaviour preservation in more than one point. Finally, this feature is required in those cases where a POI in one version is associated with more than one POI in another version (e.g., when a POI is associated with two or more POIs due to a refactoring or a removal of duplicated code).

An example of a POI tester is the tool `SecEr` (*Software Evolution Control for Erlang*), which is publicly available at `https://github.com/mistupv/secer`. The analyses performed by `SecEr` are transparent to the user. The only task in `SecEr` that requires user intervention is identifying suitable POIs in both old and new versions of a program. `SecEr` allows to define test configuration files to ease all this process and also to make it reusable. The interested readers are referred to [9] where they will find an extensive discussion about the similarities with other tools and how to deal with concurrency.

```
     Program P1            Program P2

1  main(X, Y) ⇒         main(X, Y) ⇒
2     A = X + Y,           S = diff(X, Y),
3     D = X - Y,           A = add(X, Y),
4     RETURN A * D.        RETURN A * S.
5                        add(X, Y) ⇒
6                          RETURN X + Y.
7                        diff(Y, X) ⇒
8                          RETURN X - Y.
```

Fig. 2: Two versions of the same program

In the following, we introduce the approach with the help of the example presented in Figure 2. There are three main parts in POI testing:

- **Inputs:** POI testing requires at least two parameters to be able to operate: a sequence of *POI relations* and a set of *entry points*. Additionally, POI testing can also be run with some specific comparison and report functions. The comparison and report functions are explained within the internals and the outputs of the approach, respectively.
  - *POI relations* connect POIs from two different versions of the same program. A POI relation is represented by a set of pairs. Each pair contains two POIs, one of each version of the program. For instance, the POI relation `((`P_1`, 3, (var, D)), (`P_2`, 2, (var, S), 1))` defines a relation between the two POIs contained at lines 3 and 2 respectively, in Figure 2. This POI relation indicates that the variable `D` in line 3 has been renamed to `S`, and moved to line 2.
  - *Entry points* determine the beginning of the execution. They are represented by a set of function names with their arity. For example, the set `{main/2}` defines an entry point for the approach in the example of Figure 2, i.e. all the ITCs generated for the approach are calls to function `main/2`. Concretely, this function requires the generation of specific

arguments which are not provided by the user. This is further discussed in the internals of the approach.

– **Internals:** There are three main stages of the approach. First, how traces are built, then how they are compared, and finally how new ITCs are generated.
  - *Trace building.* The basis of POI testing is the tracing of some POIs during the evaluation of a concrete ITC. For this reason, it is needed a way to create and collect these traces. This process is performed by means of a program instrumentation (like the one defined in [9] for Erlang). The instrumentation builds tuples of the form $(POI, value)$ and sends them to a trace server which collects and sorts all the tuples, producing the final trace. For example, when we run the program P_1 with the ITC `main(4,2)`, the trace obtained for the previously defined POI `(P₁, 3, (var, D))` would be `{2}`[5].
  - *Trace comparison.* Once traces are generated and stored, the next step is to compare them in order to infer if there is any UnB[6]. In the case that some UnB is found, it shows the corresponding UnB type with and an associated UnB report. There are several ways of comparing traces. The most relevant techniques to compare multi-POI traces are described in [9], where the authors distinguish between two main forms of comparison: (i) the traces are compared as a whole or (ii) the traces are compared independently for each POI. For example, consider both versions of the program shown in Figure 2. Consider also the set of POI relations

```
{((P1, 3, (var, D)), (P2, 2, (var, S))),
 ((P1, 2, (var, A)), (P2, 3, (var, A)))}
```

represented from now on as `{(P1_D, P2_S), (P1_A, P2_A)}`, and the ITC `main(5,1)`. When running this ITC, the generated traces are

```
P1: {(P1_A, 6), (P1_D, 4)}
P2: {(P2_S, -4), (P2_A, 6)}
```

If they are compared as a whole, an UnB is raised reporting that a trace from `P2_A` was expected, but a trace of `P2_S` was found instead. In this comparison, the elements of both traces must be generated in the same order for both executions. On the other hand, when they are compared separately, independent traces are generated for each POI relation. Therefore, for this example, there is an UnB found when comparing the traces of the `(P1_D, P2_S)` POI relation (i.e., value 4 was expected but value -4 was found). There is an optional input parameter that allows users to define their own comparison functions. This function

[5] Note that a trace consists of a sequence of values since a variable can be evaluated several times. Each evaluation is represented by an element of the sequence.

[6] The observed UnBs are represented and identified using literals, e.g. the atom `greater` could be used to represent an UnB that occurs when an expression is evaluated to a greater value in the new version than in the old one. The UnB representations are defined during the comparison process as it is then when the UnBs are found.

should receive two traces and must return either *true* or a tuple of the form (`UnB_Type`, `UnB_Report`), where `UnB_Type` is a label representing the UnB and `UnB_Report` is the message shown when this type of UnB is reported.

- *ITC generation.* POI testing starts by generating an ITC for each entry point. However, in order to reinforce the obtained UnB report, each time an ITC is run, new ITCs may be derived from it according to the comparison result. This way, if an UnB is found when running a particular ITC, we can generate new ITCs based on it, so that they are more conducive to generate the same or other UnBs. In [9], an ITC generation based on the mutation of the arguments of the ITC is described. Alternative generators can be used, but they should take into account the result of the trace comparison in order to obtain better results for POI testing.

– **Outputs:** The output of POI testing consists of a collection of ITCs together with the result of their trace comparisons. When none of the ITCs evaluated have generated an UnB, users are informed of the successful result by adding also some additional information such as the number of ITCs evaluated. On the contrary, when one or more UnBs have been observed, users get a report that can be configured in several ways. For example, the program of Figure 2 when using the POI relation `{(P₁_D, P₂_S)}` would result in a report similar to the one shown in Figure 3.

```
Generated test cases: 259
Mismatching test cases: 234 (90.35%)
*** Detected Error ***
Call: main(5,1)
Error Type: Unexpected trace value
POI: (P1_D) trace: [4]
POI: (P2_S) trace: [-4]
```

Fig. 3: Example of an error report

3 Enhancing POI testing with additional information

This section introduces a general overview of the enhancement that we have defined for POI testing. In order to include this enhancement, extensions have been incorporated in some of the stages of the approach introduced in Section 2. Concretely, the trace building stage, the trace comparison stage, and the output stage. Therefore, Section 4 is a special case of the general methodology explained here.

3.1 Augmenting traces with additional information

POI testing uses POI traces to check whether some UnBs exist across several program versions. We represent each element of this trace as a tuple $(POI, value)$. In this paper, we propose an extension where some additional information is attached to each *trace element*. Thus, we have extended the concept of trace

element to a triplet $(POI, value, ai)$ where ai is a mapping function containing any kind of information we add about the program context when tracing a POI.

Then, a concrete implementation of an enhanced POI testing should be able to construct these trace element triplets and handle them during the whole process. As mentioned in Section 2, the trace elements are sent when the instrumented code is executed. Then, they are collected and stored by the tracer which produces the final trace. Thus, the task of any concrete enhancement proposed for POI testing is to build these triplets, storing in ai all the desired execution information (e.g. the arguments of a function call), and manage its usage in the trace comparison and UnB report stages. In other words, for each enhancement we need to modify the code instrumentation and add some extra tracing functionality.

3.2 Using augmented traces to compare program behaviour

POI testing allows using any comparison function. This feature gives users a complete freedom to configure the testing and/or debugging process in the best way according to their needs.

In order to maximize the customization level of the comparison, users can define their own comparison functions. Each comparison function is defined by a set of connected functions that cover the different aspects of the comparison. Thus, the input part of the approach is extended with two extra functions. The connection between these functions is illustrated in Figure 4. The outer black box represents the general comparison function (it receives the whole traces of both executions as parameters, `TO` and `TN`), while the gray and white boxes represent the new mentioned functions, that are used inside the general comparison function. Their behaviour is explained below[7].

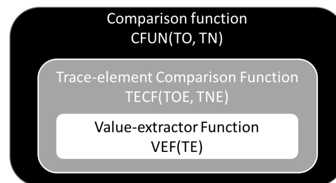

Fig. 4: Comparison function structure

– **Value-extractor function (`VEF`)**: This function works at the trace element level. Its target is to extract for any trace element only the parts that the user wants to compare. For example, function:

```
VEF(POI, value, ai) ⇒ RETURN (POI, value, ai(args))
```

extracts from a trace element only the arguments information related to call POIs and ignores the rest of additional information[8].

[7] All functions presented in Section 3 are written in pseudocode. In a particular implementation, all functions should be implemented in the target language.

[8] We use the notation *ai(key)* to refer to access some specific information previously stored in the ai mapping. In this case *ai(args)* represent the arguments of a POI placed in a function call.

- **Trace element comparison function (`TECF`)**: In order to allow users to check UnBs in different ways and not only a plain equality function, i.e. operator `==`, we add a comparison function for each pair of trace elements. This function iteratively receives pairs of trace elements contained in the whole traces, and it is the one in charge of comparing them. In order to compare two trace elements, it uses the `VEF` function to extract their values, perform a defined comparison, and, if an UnB is found, it returns an UnB type notifying about it. For example, function:

```
TECF(TOE, TNE) ⇒
    CASE compare(VEF(TOE),VEF(TNE)) OF
        gt → RETURN true
        eq → RETURN same
        lt → RETURN downgrade
    ENDCASE
```

  is an example of a `TECF`, where function `compare/2` is used to check whether a reduction in some performance indicator is obtained. Then, when it is not obtained, either a `same` or a `downgrade` UnB type is returned.

3.3 Reporting customized error types

When an UnB is detected, a specific report should be generated, i.e. a message should be provided to users. In order to further define these error messages, we have added to the approach a way to specify how the POI tester should react to a particular UnB. In this case, we have added an extra input parameter called *unexpected-behaviour report mapping* (`UnBRM`). We use a mapping, because it allows easy redefinitions and additions of UnBs reports. The mapping returns a function for a given UnB type. The returned function should build a string that will be the report message. For example, expression `UnBRM(downgrade)` may return a function similar to the one shown in Figure 5, where a custom message is shown[9] when an UnB of this type is found during the execution.

```
DOWNGRADE(TEO, TEN, History) ⇒
   RETURN "There has been a downgrade in the new version"
```

Fig. 5: Example of a function providing a customized error message

3.4 POI testing configurations

There are several ways of using the additional information stored in the trace elements, and all these modes are defined by the added resources introduced in the previous subsections (`TECF`, `VEF` and `UnBRM`). In this section, we show three different modes which will be more useful for users.

[9] The function presented in Figure 5, does not make use of parameters `TEO`, `TEN` and `History` to define the message content. However, more complex functions that treat the information stored in these parameters can be defined to obtain a more elaborated message, like the one shown in Figure 7.

– **Additional information is not used during comparison (NUAI).** In this mode, the traced values are the only data used when comparing the trace elements. This is the mode that should be used when the additional information is expected to vary due to the differences between program versions or simply due to the type of data it contains. Additionally, this mode can also be used to ease the comparison process. This mode will use a value-extractor function like `VEF(POI, V, AI) ⇒ RETURN (POI, V)` or a particular variant, where additional information is simply ignored. According on how additional information is used, we have identified three submodes.
  • **Additional information is only used to define UnB types (NUAI-T).** The additional information is only used to define new types of UnBs, but it will not appear in the UnB report. This mode is really convenient in such cases where the additional information is too complex or large, so it will not give a significative feedback to the user. In such cases, when an UnB is found, it is interesting to define a new type of UnB (e.g., `diff_value_same_args` can represent those UnBs where the values of two POIs placed at function calls differ when their call arguments are the same). The `TECF` is the one that should be defined to use this mode. For example, Figure 6 shows a `TECF` which distinguishes between those unexpected values for call POIs where the arguments are the same and those where the arguments are different[10]. Categorizing different types

```
TECF(TOE, TNE) ⇒
    IF VEF(TOE) == VEF(TNE) THEN
        RETURN true
    ELSE
        IF get_ai(TOE)(args) == get_ai(TNE)(args) THEN
            RETURN diff_value_same_args
        ELSE
            RETURN diff_value_diff_args
        ENDIF
    ENDIF
```

Fig. 6: TECF which returns different UnB types

of UnBs has several benefits in POI testing. First of all, these types can be considered in the ITC generation as a criterion to decide whether an ITC should be mutated or not. In the example above, if we do not distinguish between `diff_value_same_args` and `diff_value_diff_args`, once a `diff_value` type has been mutated it can have less chances of being selected to be mutated again. However, with this distinction, each one is treated separately, so that both mutate as separated entities. Additionally, if the final report is enriched with several UnB types, users have more feedback that can help while finding the source of the UnB.
  • **Additional information is only used in the UnB report (NUAI-R).** If we consider that the additional information is not representative enough to categorize new types of UnBs, we can use its data only in the

[10] Function `get_ai` is defined as `get_ai(POI,value,ai) ⇒ RETURN ai`.

reports. This is a less intrusive way of using the additional information, but still a useful way to obtain richer feedback in the final report of each UnB. We should add to the `UnBRM` a new function associated to each UnB type that could benefit from the stored additional information. For example, in the UnBRM, we can store the function of Figure 7 associated with the default error key `diff_value`. In this example, each time we find a difference in the results of two call POIs, we also get feedback on their arguments.

```
DIFF_VALUE({POI1, value1, ai1}, {POI2, value2, ai2}, History) ⇒
    RETURN "Value for POI1 (value1) and for POI2 (value2) differ.
           Their call arguments were:
             ai1(args)
             ai2(args)"
```

Fig. 7: Example of UnBRM error function associated to error key `diff_value`

  - **Additional information is used to categorize and report UnBs (NUAI-TR).** This submode takes the advantages of both previous submodes. It also involves specific trace-element comparison functions and additions in the UnB report mapping.

- **Additional information is used during comparison (UAI).** This mode is the one that gives a major relevance to the additional information. By using this mode, the value and the additional information is compared as a whole. This means that, for instance, even if the compared values are the same, when any pair of elements of the additional information differs, the ITC is reported to be generating an UnB. This mode is very convenient to uncover some UnBs earlier. It can also be used for performance checking, e.g. the values of the trace elements are equal but a performance indicator included in the additional information is revealing some downgrade. This mode uses a VEF function and a TECF which takes into account all or some parts of the additional information. The amount of information that is finally used to build the UnB reports is left to user's choice.
- **The additional information is not attached to the trace element, instead considered as an independent one (AIT).** Finally, in this completely different mode, the additional information is considered as a separated entity and constitutes a single trace element as the ones that are generated for the POIs. This mode can be similar to plain POI testing, however it requires a special instrumentation (to send the new trace elements), tracing (to receive and store the new trace elements) and maybe some special comparison functions (to take into account their singularities). This mode is very convenient in such cases where the additional information can be directly used to uncover an UnB, avoiding in this way the comparison of several subcomputations. For instance, if we place a POI in a call, and the call parameters are compared before comparing the call result, all intermediate trace elements are not compared. Additionally, this mode can be combined in such a way that other additional information is attached to these special trace elements forming a hybrid mode suitable for some specific scenarios.

	Program P	*Program P'*
1	`main(X, Y) ⇒`	`main(X, Y) ⇒`
2	`...`	`...`
3	`foo(X),`	`foo(X),`
4	`...`	`...`

Fig. 8: Two versions of a program with a call to the `foo` function

4 Enhancement by using improved call tracing

In this section we explain how call tracing has been improved and how the enhanced call traces have been incorporated to POI testing for Erlang.

Erlang [1] is a concurrent, functional programming language. It implements the actor model approach for concurrency, and allows concurrency based on message passing. Although one of the main features of Erlang is its potential for distributed programming, in this paper we have just focused on the non-distributed part. The sequential subset of the Erlang language supports eager evaluation and dynamic typing. In our case, the order of evaluation (eager or lazy) is not relevant for the approach, because POI testing is interested in the values of the evaluation, not in the order the expressions are evaluated. On the other hand, the dynamic typing supposed an important drawback for the ITC generation due to all possible types that needed to be considered in the generation process. This problem does not exist in static typed languages, as we know exactly the types of the value we need to generate. Additionally, all the Erlang libraries prepared to perform a live instrumentation of the code, make it a suitable language for implementing POI testing.

In this section, we also present a use case, using our tool (`SecEr`) that implements the POI testing methodology. In the use case, we compare the results provided by both the initial and the enhanced version of the tool.

4.1 Motivation

As it is usual in testing, after finding an UnB, we still have to find its source to fix it, i.e. we have to start a debugging process. Unfortunately, POI testing is not an exception. Consider the programs P and P' shown in Figure 8, and both `foo` calls at line 3 as related POIs presenting an UnB. The information provided by plain POI testing for this scenario is shown in Figure 9. This report contains a list with the set of values each function call is evaluated to, but without any context information (e.g., the parameters of each call, the Erlang dictionary of the process, etc.). Thus, it might be difficult to determine whether the source of the error is located in the arguments of the call or inside the function called.

In the new approach, the POIs placed at function calls will be treated in a special way. This special treatment augments the information given in the report, allowing us to speficy where the source of the UnB can be found. This is the main benefit of using an enhanced tracing for calls.

```
*** Detected Error ***
Call: main(6,9)
Error Type: Unexpected trace value
POI: (P, (call, foo), line 3) trace: [12]
POI: (P', (call, foo), line 3) trace: [7]
```

Fig. 9: Report returned for programs *P* and *P'*

4.2 Implementation details of the call tracing

When we place a POI in a call, we are saying that we are interested in comparing the result of this call, so the standard behaviour of a POI tester is to trace only these values. In this work, we want to create an enhanced trace where not only the result of the call, but also its arguments, are traced. Therefore, this enhancement adds to the additional information mapping a new element whose key is `ca` and whose value is a list that contains the call arguments.

In order to obtain the improved call traces, we have to define a way for sending, receiving and merging those traces. The main idea is to send the argument traces before actually performing the call and its result just after that. Thus, we should define how the code instrumentation is extended to create these enhanced trace elements, and also we should define how the tracer deals with them.

$$
\mathtt{e}(\overline{\mathtt{e_i}}) \Rightarrow \begin{array}{l}\mathtt{begin}\\ \quad \mathtt{fv_{ref}} = \mathtt{make_ref}(),\\ \quad [\mathtt{fv_v}|\overline{\mathtt{fv_{v_i}}}] = [\mathtt{e}|\overline{\mathtt{e_i}}],\\ \quad \mathtt{tracer!}\{\mathtt{add_i}, \mathtt{POI}, \mathtt{fv_{ref}}, \mathtt{fv_v}\},\\ \quad \overline{\mathtt{tracer!}\{\mathtt{add_i}, \mathtt{POI}, \mathtt{fv_{ref}}, \mathtt{fv_i}\},}\\ \quad \mathtt{fv} = \mathtt{fv_v}(\overline{\mathtt{fv_{v_i}}}),\\ \quad \mathtt{tracer!}\{\mathtt{add}, \mathtt{POI}, \mathtt{fv_{ref}}, \mathtt{fv}\},\\ \quad \mathtt{fv}\\ \mathtt{end}\end{array}
$$

(a) Instrumentation rule for call tracing

```
1  tracer({Stack, Trace}) ->
2      receive
3          {add_i, POI, Ref, V} ->
4              tracer({[{Ref, V} | Stack], Trace});
5          {add, POI, Ref, V} ->
6              {CalleeArgs, NStack} =
7                  remove_same_ref(Ref, Stack),
8              tracer({NStack,
9                  [{POI, V, store(ca, CalleeArgs)}
10                  | Trace]});
11         {add, POI, V} ->
12             tracer({Stack, [{POI, V} | Trace]})
13     end.
```

(b) Simplified tracing server

Fig. 10: Elements of our proposed call tracer enhancement in Erlang

The sending process is done thanks to a program instrumentation that enables this double tracing for the call in two steps, i.e. arguments before performing the call and the result just after the call. We show in Figure 10a how this instrumentation can be done for the programming language used, i.e. Erlang.

When the code instrumentation process finds a call, i.e. $\mathtt{e}(\overline{\mathtt{e_i}})$, the expression is then replaced by the block expression (`begin-end`) on the right-hand side. This instrumentation (i) creates a set of auxiliar free variables[11] to store all the evaluated context of the call (callee and arguments), (ii) sends to the tracer of these defined variables, (iii) performs the actual call using the value of the callee

[11] All free variables used in the rule are represented as $\mathtt{fv}_*$. Each one of these free variables is unique and different to all the original variables of the module.

and the arguments, (iv) sends the result of the call to the trace server, and (v) return the result of the call to make the block return the expected result.

All the information sent while running the instrumented code is received and merged by the tracer. In Erlang, the tracer is a server which is continuously receiving trace elements until the end of the execution or until a timeout is raised. Figure 10b shows a simplification of the Erlang function `tracer/1` which is in charge of this tracing process. The server state is a tuple containing: 1) a stack, where the callee and arguments are stored in the order they are received, and 2) the trace generated so far. Its body is a receive expression with three clauses: the first one is for the information sent by function calls' callees and arguments, the second one is for the result of the function call, and the third one is for the rest of trace elements, i.e. those that do not come from a function call. When a callee or an argument value is received, it is simply stacked. When the call result is received, all its arguments, which are at the top of the stack, are unstacked (function `remove_same_ref/2`), and stored in the additional information of the call trace element. Finally, the rest of trace elements are simply added to the current trace with an empty additional information structure.

4.3 Using the enhanced call tracing to compare traces

In this section, we describe the specific requirements needed to use the implemented enhancement of Section 4.2. To this end, we use two executions of our POI tester `SecEr`, one execution previous to our proposed enhancement and another one after it. In concrete, we compare two versions of an Erlang program that aligns columns of a string with multiple lines. The code of both versions is shown in Figure 11. While `align_columns_ok.erl` version code is implemented using line 19, the `align_columns.erl` version replaces that line of code with line 20. Due to space limitations, the implementation of some trusted methods is omitted, and some changes have been done. Both full program versions are part of the benchmarks used in EDD (Erlang Declarative Debugger) [3], and their originals can be found at `https://github.com/tamarit/edd/tree/master/examples/align_columns`.

These programs export a function with zero parameters, that can be considered a unit case. Our selected entry pointwill be function `align_left/0`. In order to test both `SecEr` implementations, we need to define also a POI relation. Our POI relation, defined in our configuration file is shown in Figure 12.

Additionally, for enhanced POI testing, a configuration mode must be selected among the ones shown in Section 3.4. In our case, we have selected **NUAI-TR** mode, providing the Erlang implementation of the `TECF` function shown in Figure 6 and an `UnBRM` that reports also the callee and the args of a function call as part of the report. The configuration proposed is represented with the function `config/0`[12] represented by line 5 in Figure 12. Once all the configuration requirements are fulfilled, we have run `SecEr` obtaining the results shown in Figure 13.

[12] In our implementation, the Erlang module `secer_api` provides a list of implemented functions to easily select any execution mode.

```
-module (align_columns_ok). / -module (align_columns).
-export([align_left/0]).

align_left()-> align_columns(left).
align_columns(Alignment) ->
    Lines = ["Weak$people$revenge",
             "Strong$people$forgive",
             "Intelligent$people$ignore"],
    Words = [ string:tokens(Line, "$") || Line <- Lines ],
    Words_length  = lists:foldl( fun max_length/2, [], Words),
    [prepare_line(Words_line, Words_length, Alignment) || Words_line <- Words].

max_length(Words_of_a_line, Acc_maxlength) -> ...% Trusted method
adjust_list(L, Desired_length, Elem) -> ... % Trusted method

prepare_line(Words_line, Words_length, Alignment) ->
    All_words = adjust_list(Words_line, length(Words_length), ""),
    Zipped = lists:zip(All_words, Words_length),
    [ apply(string, Alignment, [Word, Length + 1, $\s]) % align_columns_ok
    [ apply(string, Alignment, [Word, Length - 1, $\s]) % align_columns
      || {Word, Length} <- Zipped ].
```

Fig. 11: Align columns program versions

```
poiOld() -> {'align_columns_ok.erl', 19, call}.
poiNew() -> {'align_columns.erl', 20, call}.
rel() -> [{poiOld(),poiNew()}].
funs() -> "[align_left/0]".
config() -> secer_api:nuai_tr_config(mytecf(),ubrm()).
```

Fig. 12: `SecEr` configuration file

Figure 13a shows the information provided by plain POI testing implemented by `SecEr`. When analysing this information, we can notice that there is not enough feedback to decide whether the bug comes from the arguments or from the function call. We do not even know the function we are calling when the UnB was raised because it is passed as an argument to function `prepare_line/3`. On the other hand, Figure 13b represents the execution of `SecEr` after implementing the enhanced call tracing. It can be seen that the source of the error is clearly the list in the third argument of the call `apply`, where the second element differs for both programs. Thus, we can conclude that the error comes from the expression calculating this argument. Furthermore, we also can observe that the called function in the `apply` call is the function `string:left/3`, information that was not provided by the previous `SecEr` version. This is just a single example, more examples of use cases dealing with some real-life programs (e.g. `https://github.com/mistupv/secer/benchmarks/rebar`) can be found at our Github repository.

5 Related work

The orchestrated survey of methodologies for automated software test case generation [2] identifies five techniques to automatically generate test cases. POI testing could be included in the class of *adaptive random technique as a variant*

```
$ ./secer -pois "test_align:rel()"
          -funs "test_align:funs()"
          -to 5


Function: align_left/0
----------------------------
Generated test cases: 1
Mismatching test cases: 1 (100.0%)
  Error Types:
    + different_value => 1 Errors
        Example call: align_left()

------ Detected Error ------
Call: align_left()
Error Type: different_value
- - - - - - - - - - - - - - -
POI: {'align_columns_ok.erl',19,call,1}
  Trace:
    ["Weak        "]
POI: {'align_columns.erl',20,call,1}
  Trace:
    ["Weak      "]
```

(a) SecEr without enhancements

```
$ ./secer -pois "test_align:rel()"
          -funs "test_align:funs()"
          -to 5
          -config "test_align:config()"

Function: align_left/0
----------------------------
Generated test cases: 1
Mismatching test cases: 1 (100.0%)
  Error Types:
    + different_value_different_args => 1 Errors
        Example call: align_left()

------ Detected Error ------
Call: align_left()
Error Type: different_value_different_args
- - - - - - - - - - - - - - -
POI: {'align_columns_ok.erl',19,call,1}
  Trace:
    ["Weak        "]
  Call POI Info:
    Callee: apply
    Args: [string,left,["Weak",12,32]]
POI: {'align_columns.erl',20,call,1}
  Trace:
    ["Weak      "]
  Call POI Info:
    Callee: apply
    Args: [string,left,["Weak",10,32]]
```

(b) SecEr with call tracing enhancement

Fig. 13: Comparison of SecEr executions

of random testing. Within this class, the authors identify five approaches. POI testing mutation approach of the test input shares several similarities with some approaches like selection of best candidate as next test case or exclusion. According to a survey on test amplification [5], which identifies four categories that classify all the work done in the field, our work could be included in the category named *amplification by synthesizing (new tests with respect to changes)*. Inside this category, our technique falls into the "other approaches" subcategory.

There are other approaches that use traces to compare program versions, like the ones based on program spectra [11]. Different program spectra have been proposed (branch, execution trace or data dependence spectra), but value spectra [16] is the most similar to our call trace enhancement. In particular, value trace spectra record the sequence of the user-function executions traversed as a program executes. After the spectra recording, spectra comparison techniques are used to find value spectra differences that expose internal behavioral deviations inside the black box. However, the spectrum is generated for all the user-defined functions while in our approach users decide which functions should be compared. Additionally, POI testing allows a more flexible use of these call traces. Finally, the motivation and also some techniques of the enhanced call traces are similar to the ones of *algorithmic debugging* [12]. In fact, this approach has been successfully applied to Erlang [3].

Most of the efforts in regression testing research have been put in regression testing minimization, selection, and prioritization [17], although among practitioners it does not seem to be the most important issue [6]. In fact, in the particular case of the Erlang language, most of the works in the area are focused on this specific task [13,15]. We can find other works in Erlang that share similar goals but more are focused on checking whether applying a refactoring rule will yield to a semantics-preserving new code [10].

With respect to tracing, there are multiple approximations similar to the POI testing's. In Erlang's standard libraries, there are two tracing modules. Both are able to trace function calls and process related events (spawn, send, receive, etc.). One of these modules is oriented to trace the processes of a single Erlang node [7], allowing for the definition of filters to function calls, e.g., with names of the function to be traced. The second module is oriented to distributed system tracing [8] and the output trace of all the nodes can be formatted in many different ways. M. Cronqvist [4] presented a tool named redbug where a call stack trace is added to the function call tracing, making possible to trace both the result and the call stack. A. Till [14] implemented erlyberly, a debugging tool with a Java GUI able to trace the previously defined features but also giving the possibility to add breakpoints and trace other features such as exceptions thrown or incomplete calls. All these tools are accurate to trace specific features of the program, but none of them is able to trace the value of an arbitrary point of it. In our approach, we can trace both the already defined features and a point of the program regardless of its position.

6 Conclusions

We have presented a common framework to enhance POI testing with the addition of new information. This new information enriches the approach, allowing users to get better UnB reports and to define new UnB types. These new UnB types benefit some of the internal processes of the approach, e.g. the ITC generation. These additions need new ways to send and store this additional information and also new comparison modes. In this paper, an enhancement have been proposed by augmenting call traces. This enhancement has its particularities, but it follows the common framework presented in Section 3.

This work opens a way to extensions of POI testing. Following the same common framework described in this paper, we can easily include different additional information. This additional data can be other functional data, i.e. similar data to richer call traces. One interesting enhancement is to store a snapshot of the current environment for each POI, so more contextual information is available to find the UnB source. We could also store the followed conditional paths, so it could be used to improve the coverage during the ITC generation. At the same time, we are studying the use of some special additional information that enables the *mocking* of values. The idea is to execute again an ITC that leads to an UnB, but when the value that uncovers the UnB is found, replace it by the value computed by a correct version of the program. This will allow the technique to

find further errors using the same ITC. The same idea can be applied when an internal call is a previously executed ITC in order to avoid its recomputation. Finally, we plan to define extensions of the approach that study non-functional data, e.g. CPU or memory usage. After some preliminary work, we have concluded that the common framework presented in this paper represents a very natural way to operate with such kind of data.

References

1. Erlang. Available at: `http://www.erlang.org/`, 1986.
2. S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.
3. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. EDD: A declarative debugger for sequential Erlang programs. *20th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, volume 8413 of *LNCS*, pages 581–586. Springer, April 2014.
4. M. Cronqvist. *redbug.* Available at: `https://github.com/massemanet/redbug`,2017.
5. B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry. The Emerging Field of Test Amplification: A Survey. *CoRR*, abs/1705.10692, 2017.
6. E. Engström and P. Runeson. A Qualitative Survey of Regression Testing Practices. In M. A. Babar, M. Vierimaa, and M. Oivo, editors, *Product-Focused Software Process Improvement, 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings*, volume 6156 of *Lecture Notes in Business Information Processing*, pages 3–16. Springer, 2010.
7. Ericsson AB. dbg. Available at: `http://erlang.org/doc/man/dbg.html`, 2017.
8. Ericsson AB. Trace tool builder. Available at: `http://erlang.org/doc/apps/observer/ttb_ug.html`, 2017.
9. D. Insa, S. Pérez, J. Silva, and S. Tamarit. Behaviour Preservation across Code Versions in Erlang. *Scientific Programming*, vol. 2018, pages 1–42, 2018.
10. E. Jumpertz. Using QuickCheck and semantic analysis to verify correctness of Erlang refactoring transformations; Master's thesis, Radboud University Nijmegen, 2010.
11. T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, Proceedings*, volume 1301 of *LNCS*, pages 432–449. Springer, 1997.
12. E. Y. Shapiro. *Algorithmic program debugging.* MIT Press, April 1982.
13. R. Taylor, M. Hall, K. Bogdanov, and J. Derrick. Using behaviour inference to optimise regression test sets. In *IFIP International Conference on Testing Software and Systems*, pages 184–199. Springer, 2012.
14. A. Till. erlyberly. Available at: `https://github.com/andytill/erlyberly`, 2017.
15. I. B. M. Tóth and Z. Horváth. Reduction of regression tests for Erlang based on impact analysis. 2013.
16. T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Trans. Software Eng.*, 31(10):869–883, 2005.
17. S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, 2012.