Hindawi
Scientific Programming
Volume 2018, Article ID 9251762, 42 pages
https://doi.org/10.1155/2018/9251762

Research Article

Behaviour Preservation across Code Versions in Erlang

David Insa, Sergio Pérez, Josep Silva, and Salvador Tamarit

Universitat Politècnica de València, Camí de Vera s/n, E-46022 València, Spain

Correspondence should be addressed to Josep Silva; jsilva@dsic.upv.es

Received 2 October 2017; Revised 13 March 2018; Accepted 31 March 2018; Published 13 June 2018

Academic Editor: Can Özturan

In any alive and nontrivial program, the source code naturally evolves along the lifecycle for many reasons such as the implementation of new functionality, the optimization of a bottleneck, or the refactoring of an obscure function. Frequently, these code changes affect various different functions and modules, so it can be difficult to know whether the correct behaviour of the previous version has been preserved in the new version. In this paper, we face this problem in the context of the Erlang language, where most developers rely on a previously defined test suite to check the behaviour preservation. We propose an alternative approach to automatically obtain a test suite that specifically focusses on comparing the old and new versions of the code. Our test case generation is directed by a sophisticated combination of several already existing tools such as TypEr, CutEr, and PropEr; and it introduces novel ideas such as allowing the programmer to choose one or more expressions of interest that must preserve the behaviour, or the recording of the sequences of values to which those expressions are evaluated. All the presented work has been implemented in an open-source tool that is publicly available on GitHub.

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. There are multiple ways of control so that these changes do not modify the behaviour of any part of the program that was already correct. Most of the companies rely on *regression testing* [1, 2] to ensure that a desired behaviour of the original program is kept in the new version, but there exist other alternatives such as the static inference of the impact of changes [3–6].

Even when a program is perfectly working and it fulfils all its functional requirements, sometimes we still need to improve parts of it. There are several reasons why a released program needs to be modified, for instance, improving the maintainability or efficiency, or for other reasons such as obfuscation, security improvement, parallelization, distribution, platform changes, and hardware changes. In the context of scientific programming, it is common to change an algorithm several times until certain performance requirements are met. During each iteration, the code often naturally becomes more complex and, thus, more difficult to understand and debug. Although regression testing should be ideally done after each change, in real projects, the methodology is really different. As reported in [7], only 10% of the companies do regression testing daily. This means that when an error is detected, it can be hidden after a large number of subsequent changes. The authors also claim that this long-term regression testing is mainly due to the lack of time and resources.

Programmers that want to check whether the semantics of the original program remain unchanged in the new version usually create a test suite. There are several tools that can help in all of this process. For instance, Travis CI can be easily integrated in a GitHub repository so that each time a pull request is performed, the test suite is launched. We present here an alternative and complementary approach that creates an automatic test suite to do regression testing: (i) an alternative approach because it can work as a standalone program without the need for other techniques (therefore, our technique can check the evolution of the code even if no test suite has been defined) and (ii) a complementary approach because it can also be used to complement other techniques, providing major reliability in the assurance of behaviour preservation.

More sophisticated techniques, but with similar purpose, have been recently announced like Ubisoft's system [8] that is able to predict programmer errors beforehand. It is quite illustrative that a game-developer company was the first one in presenting a project like this one. The complex algorithms used to simulate physical environments and AI behaviours need several iterations in order to improve their performance. It is in one of those iterations that some regression faults can be introduced.

In the context of debugging, programmers often use breakpoints to observe the values of an expression during an execution. Unfortunately, this feature is not currently available in testing, even though it would be useful to easily focus the test cases on one specific point without modifying the source code (as it happens when using assertions) or adding more code (as it happens in unit testing). In this paper, we introduce the ability to specify *points of interest* (POIs) in the context of testing. A POI can be any expression in the code (e.g., a function call), meaning that we want to check the behaviour of that expression. Although they handle similar concepts, our POIs are not exactly like breakpoints, since their purpose is different. Breakpoints are used to indicate where the computation should stop, so the user can inspect variable values or control statements. In contrast, a POI defines an expression whose sequence of evaluations to values must be recorded, so that we can check the behaviour preservation (by value comparison) after the execution. In particular, note that placing a breakpoint inside a unit test is not the same as placing a POI inside it because the goals are different.

In our technique, (1) the programmer identifies a POI and a set of *input functions* whose invocations should evaluate the POI. Then, by using a combination of random test case generation, mutation testing, and concolic testing, (2) the tool automatically generates a test suite that tries to cover all possible paths that reach the POI (trying also to produce execution paths that evaluate the POI several times). Therefore, in our setting, the *input of a test case* (ITC) is defined as a call to an input function with some specific arguments, and the output is the sequence of those values the POI is evaluated to during the execution of the ITC. For the sake of disambiguation, in the rest of the paper, we use the term *traces* to refer to these sequences of values. Next, (3) the test suite is used to automatically check whether the behaviour of the program remains unchanged across new versions. This is done by passing each individual test case (which contains calls to the input functions) against the new version and checking whether the same traces are produced at the POI. Finally, (4) the user is provided with a report about the success or failure of these test cases. Note that as it is common in regression testing, this approach only works for deterministic executions. However, this does not mean that it cannot be used in a program with concurrency or other sources of nondeterminism; it only depends on where the POIs are placed and the input functions used. In Section 7, we clarify how our approach can be used in such contexts.

After presenting the approach for a single POI, we present an extension that allows for the definition of multiple POIs. With this extension, the user can trace several (and maybe unrelated) functionalities in a single run. It is also useful when we want to strengthen the quality of the test suite by checking, for instance, that the behaviour is kept in several intermediate results. Finally, this extension is needed in those cases where a POI in one version is associated with more than one POI in another version (e.g., when a POI in the final source code is associated with two or more POIs in the initial source code due to a refactoring or a removal of duplicated code).

We have implemented our approach in a tool named `SecEr` *(Software Evolution Control for Erlang)*, which is publicly available at `https://github.com/mistupv/secer`. Instead of reinventing the wheel, some of the analyses performed by our tool are done by other existing tools such as `CutEr` [9], a concolic testing tool, to generate an initial set of test cases that maximize the branching coverage; `TypEr` [10], a type inference system for Erlang, to obtain types for the input functions; and `PropEr` [11], a property-based testing tool, to obtain values of a given type. All the analyses performed by `SecEr` are transparent to the user. The only task in our technique that requires user intervention is identifying suitable POIs in both the old and the new versions of the program. In order to evaluate our technique and implementation, we present in Section 8 a comparison of `SecEr` with the most extended alternatives for the detection of discrepancies and their causes in Erlang. All techniques are compared using the same example. Additionally, in Section 9, we complement this study with an empirical evaluation of `SecEr`.

Example 1. In order to show the potential of the approach, we provide a real example to compare two versions of an Erlang program that computes happy numbers. They are taken from the Rosetta Code repository (consulted in this concrete version: http://rosettacode.org/mw/index.php?title=Happy_numbers&oldid=251560#Erlang) and slightly modified (the introduced changes are explained in Section 6):

http://rosettacode.org/wiki/Happy_numbers#Erlang

The initial and final versions of this code as they appear in Rosetta Code are shown in Listings 1 and 2, respectively. In order to check whether the behaviour is the same in both versions, we could select as POI the call in line (9) of Listing 1 and the call in line (18) of Listing 2. We also need to define a timeout because the test case generation phase could be infinite due to the test mutation process (the number of possible execution paths could be infinite and an infinite number of test cases could be generated). In this example, with a timeout of 15 seconds, `SecEr` reports that the executions of both versions with respect to the selected POIs behave identically. In Section 6, we show how `SecEr` can help a user when an error is introduced in this example and how the multiple POIs approach is also helpful to find the source of an error.

2. Overview of Our Approach to Automated Regression Testing

Our technique is divided into three sequential phases that are summarized in Figures 1, 2, and 3. In these figures, the big dark grey areas are used to group several processes with a common objective. Light grey boxes outside these areas represent inputs and light grey boxes inside these areas represent processes, white boxes represent intermediate

```
-spec main(pos_integer(),pos_integer()) ->
  [pos_integer()].
main(N, M) ->
  happy_list(N, M, []).

happy_list(_, N, L) when length(L) =:= N ->
  lists:reverse(L);
happy_list(X, N, L) ->
  Happy = is_happy(X),
   if Happy ->
     happy_list(X + 1, N, [X|L]);
   true ->
     happy_list(X + 1, N, L) end.

 is_happy(1) -> true;
 is_happy(4) -> false;
 is_happy(N) when N > 0 ->
   N_As_Digits =
     [Y - 48 ||
     Y <- integer_to_list(N)],
   is_happy(
     lists:foldl(
       fun(X, Sum) ->
         (X * X) + Sum
       end,
       0,
       N_As_Digits));
is_happy(_) -> false.
```

Listing 1: happy0.erl.

```
is_happy(X, XS) ->
  if
    X == 1 -> true;
    X < 1 -> false;
    true ->
      case member(X, XS) of
        true -> false;
        false ->
          is_happy(sum(map(fun(Z) -> Z*Z end,
             [Y - 48 || Y <- integer_to_list(X)])),
             [X|XS])
      end
   end.
happy(X, Top, XS) ->
   if
    length(XS) == Top -> sort(XS);
    true ->
      case is_happy(X,[]) of
        true -> happy(X + 1, Top, [X|XS]);
        false -> happy(X + 1,Top, XS)
      end
   end.

-spec main(pos_integer(),pos_integer()) ->
   [pos_integer()].
main(N, M) ->
   happy(N, M, []).

```

Listing 2: happy1.erl.

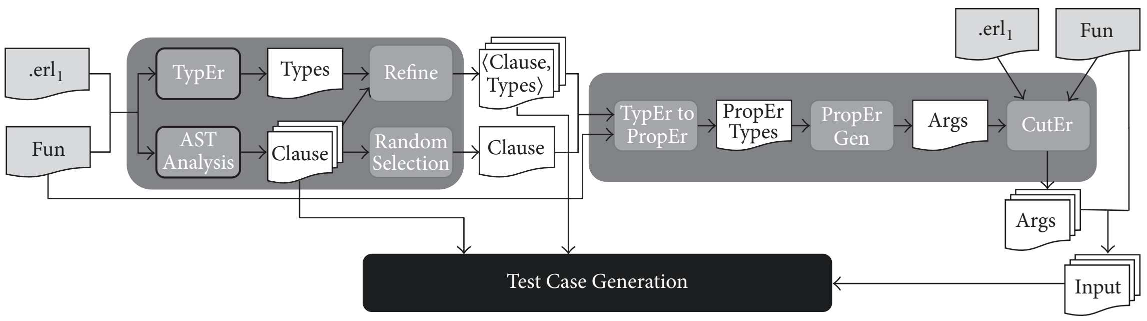


FIGURE 1: Type analysis phase.

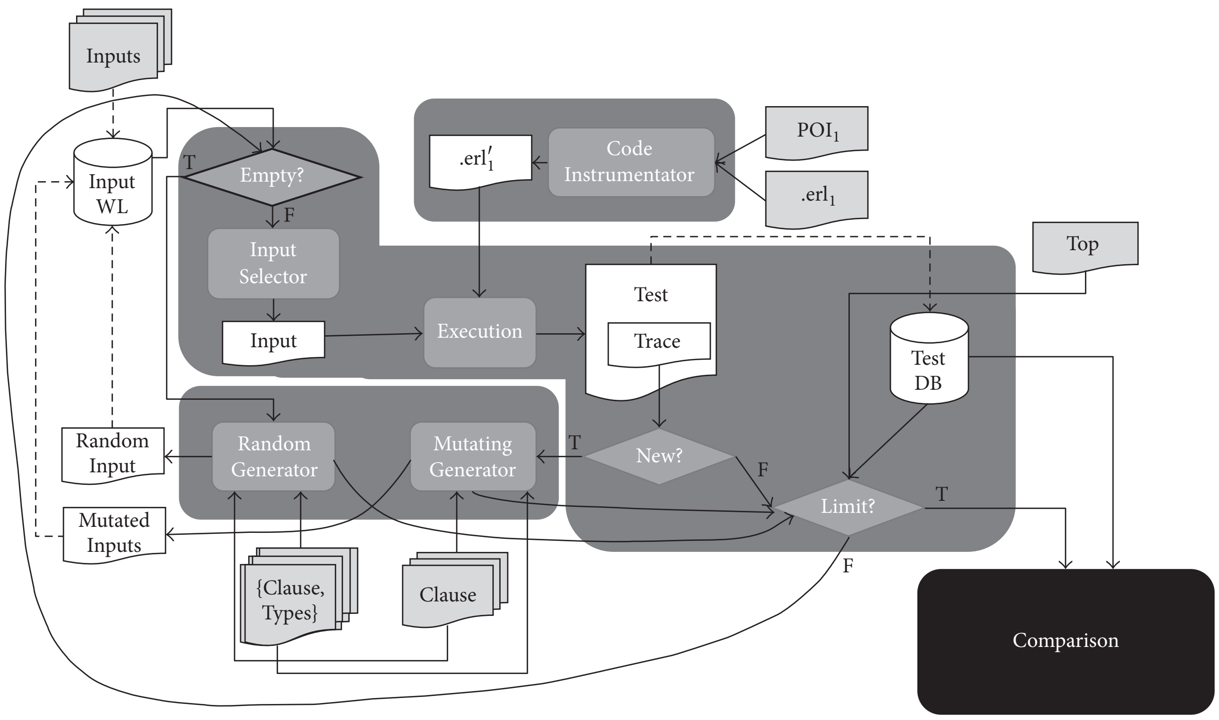


FIGURE 2: Test case generation phase.

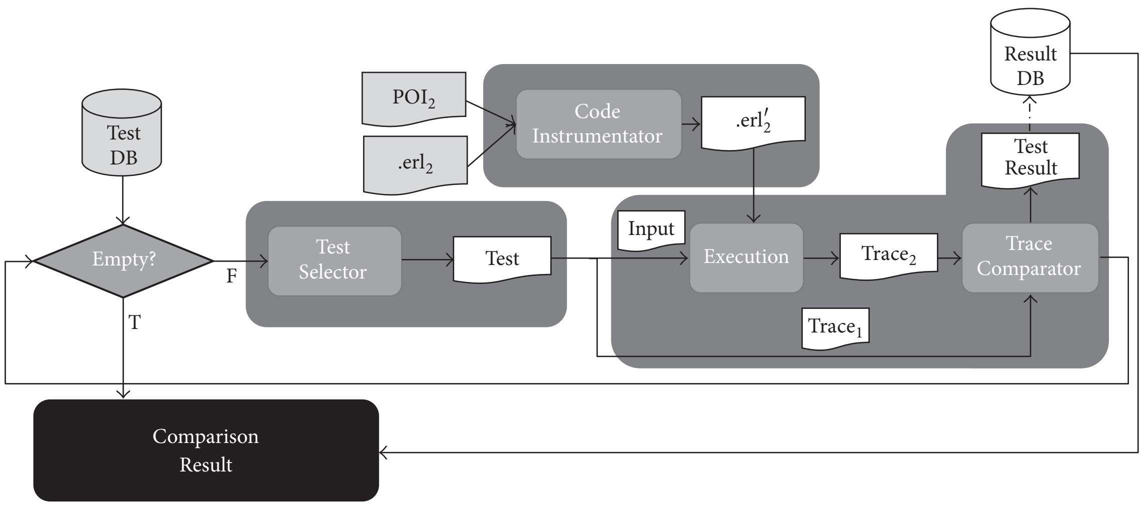


FIGURE 3: Comparison phase.

results, and the initial processes of each phase are represented with a bold border box.

The first phase, depicted in Figure 1, is a type analysis that is in charge of preparing all inputs of the second phase (test case generation). This phase starts by locating in the source code the Erlang module (`.erl`$_1$) and a function (`Fun`) specified in the user input (we show here the process for only one function. In case the user defined more than one input function, the process described here would be repeated for each function), e.g., function `exp` in the `math` module. Then, `TypEr` is used to obtain the type of the parameters of that function. It is important to know that, in Erlang, a function is composed of clauses and when a function is invoked, an internal algorithm traverses all the clauses in order to select the one that will be executed. Unfortunately, `TypEr` does not provide the individual type of each clause, but a global type for the whole function. Therefore, we need to first analyze the AST of the module to identify all the clauses of the input function, and then we refine the types provided by `TypEr` to determine the specific type of each clause. All these clause types are used in the second phase. In this phase, we use `PropEr` to instantiate only one of them (e.g., $\langle Number, Integer\rangle$ can be instantiated to $\langle 4.22, 3\rangle$ or $\langle 6, 5\rangle$). However, `PropEr` is unable to understand `TypEr` types, so we have defined a translation process from `TypEr` types to ProEr types. Finally, `CutEr` is fed with an initial call (e.g., `math:exp(4.22, 3)`) and it provides a set of possible arguments (e.g., $\{\langle 1.5, 6\rangle, \langle 2, 1\rangle, \langle 1.33, 4\rangle, \dots\}$). Finally, this set is combined with the function to be called to generate the ITCs (e.g., {`math:exp(1.5, 6)`, `math:exp(2, 1)`, `math:exp(1.33, 4)`,...}). All this process is explained in detail in Section 3.1.

The second phase, shown in Figure 2, is in charge of generating the test suite. As an initial step, we instrument the program so that its execution records (as a side effect) the sequence of values produced at the POI defined by the user. Then, we store all ITCs provided by the previous phase into a working list. Note that it is also possible that the previous phase is unable to provide any ITC due to the limitations of `CutEr`. In such a case, or when there are no more ITCs left, we randomly generate a new one with `PropEr` and store it on the working list. Then, each ITC on the working list is processed by invoking it with the instrumented code. The execution provides the sequences of values the POI is evaluated to (i.e., the trace). This trace together with the ITC forms a new test case, which is a new output of the phase. Moreover, to increase the quality of the test cases produced, whenever a non-previously generated trace is computed, we mutate the ITC that generated that trace to obtain more ITCs. The reason is that a mutation of this ITC will probably generate more ITCs that also evaluate the POI but to different values. This process is repeated until the specified limit of test cases is reached. All this process is explained in detail in Sections 3.2 and 3.3. In Section 4 there is a discussion of how this approach could be extended to support multiple POIs.

Finally, the last phase (shown in Figure 3) checks whether the new version of the code passes the test suite. First, the source code of the new version is also instrumented to compute the traces produced at its POI. Then, all the generated test cases are executed and the traces produced are compared with the expected traces. Section 5 introduces functions to compare traces that give support to the multiple-POI approach.

3. A Novel Approach to Automated Regression Testing

In this section, we describe in more detail the most relevant parts of our approach. We describe them in separate subsections.

3.1. Initial ITC Generation. The process starts from the type inferred by `TypEr` for the whole input function. This is the first important step to obtain a significant result, because ITCs are generated with the types returned by this process, so the more accurate the types are, the more accurate the ITCs are. The standard output of `TypEr` is an Erlang type specification returned as a string, which would need to be parsed. For this reason, we have hacked the Erlang module that implements this functionality to obtain the types in a data structure, easier to traverse and handle. In order to improve the accuracy, we define a type for each clause of the function ensuring that the later generated ITCs will match it. For this reason, `TypEr` types need to be refined to `TypEr` types per clause.

However, the types returned by `TypEr` have (in our context) two drawbacks that need to be corrected since they could yield to ITCs that do not match a desired input function. These drawbacks are due to the type produced for lists and due to the occurrence of repeated variables. We explain both drawbacks with an example. Consider a function with a single clause whose header is `f(A,[A,B])`. For this function, `TypEr` infers the type `f( 1 | 2, [ 1 | 2 | 5 | 6, ... ] )` (`TypEr` uses a *success typing* system instead of the usual Hindley-Milner type inference system. Therefore, `TypEr`'s types are different from what many programmers would expect, i.e., integer, string, etc. Instead, a `TypEr`'s type is a set of values such as `[ 1 | 2 | 5 | 6 ]`or an Erlang defined type, e.g., number and integer). Thus, the type of the second parameter of the `f/2` function indicates that the feasible values for the second parameter are proper lists with a single constraint: it has to be formed with numbers from the set `[1,2,5,6]`. This means that we could build lists of any length, which is our first drawback. If we use these `TypEr` types, we may generate ITCs that will not match the function, e.g., `f(2,[2,1,3,5])`. On the other hand, our second drawback is caused by the fact that the value relation generated by the repeated variable `A` is lost in the function type. In particular, the actual type of variable `A` is diluted in the type of the second argument. This could yield to mismatching ITCs if we generate, e.g., `f(1,[6,5])`.

Therefore, the types produced by `TypEr` are too imprecise in our context, because they may produce test cases that are useless (e.g., nonexecutable). This problem is resolved in different steps of the process. In this step, we can only partially resolve the type conflict introduced by the repeated variables, such as the `A` variable in the previous example. The other drawback will be completely resolved during the ITC generation. To solve this problem, we traverse the parameters building a correspondence between each variable and the inferred `TypEr` type. Each time a variable appears more than once, we calculate its type as the intersection of both the `TypEr` type and the accumulated type. For instance, in the previous example, we have `A = 1 | 2` for the first occurrence and `A = 1 | 2 | 5 | 6` for the second one, obtaining the new accumulated type `A = 1 | 2`.

Once we have our refined `TypEr` types, we rely on `PropEr` to obtain the input for `CutEr`. `PropEr` is a property-based testing framework with a lot of useful underlying functionalities. One of them is the term generators, which, given a `PropEr` type, are able to randomly generate terms belonging to such type. Thus, we can use the generators in our framework to generate values for a given type.

However, `TypEr` and `PropEr` use slightly different notations for their types, something reasonable given that their scopes are completely different. Unfortunately, there is not any available translator from `TypEr` types to `PropEr` types. In our technique, we need such a translator to link the inferred types to the `PropEr` generators. Therefore, we have built the translator by ourselves. Moreover, during the ITC generation, we need to deal with the previously postponed type drawbacks. For that, we use the parameters of the clause in conjunction with their types. To solve the first drawback, each time a list is found during the generation, we traverse its elements and generate a type for each element on the list. Thereby, we synthesize a new type for the list with exactly the same number of elements. The second drawback is solved by using a map from variables to their generated values. Each time a repeated variable is found, we use the stored value instead of generating a new one.

We can feed `CutEr` with an initial call by using a randomly selected clause and the values generated by `PropEr` for this clause. `CutEr` is a concolic testing framework that generates a list of arguments that tries to cover all the execution paths. Unfortunately, this list is only used internally by `CutEr`, so we have hacked `CutEr` to extract all these arguments. Finally, by using this slightly modified version of `CutEr` we are able to mix the arguments with the input function to generate the initial set of ITCs.

3.2. Recording the Traces of the Point of Interest. There exist several tools available to trace Erlang executions [12–15] (we describe some of them in Section 10). However, none of them allows for defining a POI that points to any part of the code. Being able to trace any possible point of interest requires either a code instrumentation, a debugger, or a way to take control of the execution of Erlang. However, using a debugger (e.g., [13]) has the drawback that it does not provide a value for the POI when it is inside an expression whose evaluation fails. Therefore, we decided to instrument the code in such a way that, without modifying the semantics of the code, traces are collected as a side effect when executing the code.

The instrumentation process creates and collects the traces of the POI. To create the traces in an automatic way, we instrument the expression pointed by the POI. To collect the traces, we have several options. For instance, we can store the traces in a file and process it when the execution finishes, but this approach is inefficient. We follow an alternative approach based on message passing. We send messages to a server (which we call the *tracing server*) that is continuously listening for new traces until a message indicating the end of the evaluation is received. This approach is closer to Erlang's philosophy. Additionally, it is more efficient since the messages are sent asynchronously resulting in an imperceptible overhead in the execution. As a result of the instrumenting process, the transformed code sends to the tracing server the value of the POI each time it is evaluated, and the tracing server stores these values.

In the following, we explain in detail how the communication with the server is placed in the code. This is done by applying the following steps:

(1) We first use the `erl_syntax_lib:annotate_bindings/2` function to annotate the AST of the code. This function annotates each node with two lists of variables: those variables that are being bound and those that were already bound in its subtree. Additionally, we annotate each node with a unique integer that serves as an identifier, so we call it *AST identifier*. This annotation is performed in a postorder traversal, resulting, consequently, in an AST where the root has the greatest number.

(2) The next step is to find the POI selected by the user in the code and obtain the corresponding AST identifier. There are two ways of doing this depending on how the POI is specified: (i) if the POI is defined with the triplet *(line, type of expression, occurrence)*, we locate it with a preorder traversal (we use this order because it is the one that allows us to find the nodes in the same order as they are in the source code) of the tree. However, (ii) when the POI is defined with the initial and final positions, we replace, in the source code, the whole expression with a fresh term. Then, we build the AST of this new code and we search for the fresh term in this AST recording the path followed. This path is replicated in the original AST to obtain the AST identifier of the POI. Thus, the result of this step is a relation between a POI and an AST identifier.

(3) Then, we need to extract the path from the AST root to the AST identifier of the POI using a new search process. This double search process is later justified when we introduce the multiple POIs approach in Section 4. During this search process, we store the path followed in the AST with tuples of the form `(Node, ChildIndex)`, where `Node` is the AST node and `ChildIndex` is the index of the node in its parent's children array. Obtaining this path is essential for the next steps since it allows us to recursively update the tree in an easy and efficient way. When the AST identifier is found, the traversal finishes. Thus, the output of this step is a path that yields directly to the AST identifier searched.

(4) Most of the times, the POI can be easily instrumented by adding a send command to communicate its value to the tracing server. However, when the POI is in

```
(LEFT_PM)    p = e ⇒ p = begin np = e, tracer!{add, npoi}, np end
        if      $(p = e, \_) = last(PathBefore)$
                $\wedge (\_, pos(p)) = hd(PathAfter)$
     where   $(\_, npoi, np) = pfv(p, PathAfter)$

(PAT_GEN_LC)    [e || gg] ⇒ [e || ngg]
           if   $([e \,||\, gg], \_) = last(PathBefore)$
                $\wedge (\_, pos(p\_gen)) = hd(tl(PathAfter))$
                $\wedge \exists\, i.\ 1 \le i \le length(gg)$ s.t. $gg_i$ = p_gen <- e_gen
        where   $(\_, npoi, np\_gen) = pfv(p\_gen, tl(PathAfter))$
                $\wedge$ ngg_i = p_gen <- begin tracer!{add, npoi}, [np_gen] end
                $\wedge$ ngg = $gg_1 \cdots gg_{i-1}$, np_gen <- e_gen, ngg_i, $gg_{i+1} \cdots gg_{length(gg)}$

(CLAUSE_PAT)    e ⇒ $change\_clauses(e, ncls)$
           if    $(e, \_) = last(PathBefore)$
                $\wedge (\_, pos(p\_c)) = hd(tl(PathAfter))$
                $\wedge \exists\, i.\ 1 \le i \le length(cls)$ s.t. $cls_i$ = p_c when g_c -> b_c
        where   cls = $clauses(e)$
                $\wedge (\_, npoi, np\_c) = pfv(p\_c, tl(PathAfter))$
                $\wedge$ nb_c = begin tracer!{add, npoi}, case np_c of cls end end
                $\wedge$ ncls_i = np_c when true -> nb_c
                $\wedge$ ncls = $cls_i, \ldots, cls_{i-1}$, ncls_i, $cls_{i+1}, \ldots, cls_{length(cls)}$

(CLAUSE_GUARD)    e ⇒ $change\_clauses(e, ncls)$
             if    $(e, \_) = last(PathBefore)$
                $\wedge (\_, pos(g\_c)) = hd(tl(PathAfter))$
                $\wedge \exists\, i.\ 1 \le i \le length(cls)$ s.t. $cls_i$ = p_c when g_c -> b_c
          where    cls = $clauses(e)$
                $\wedge (poi, \_) = last(PathAfter)$
                $\wedge$ nb_c = begin tracer!{add, poi}, case np_c of cls end end
                $\wedge$ ncl = p_c when true -> nb_c
                $\wedge$ ncls = $cls_i, \ldots, cls_{i-1}$, ncl, $cls_{i+1}, \ldots, cls_{length(cls)}$

(EXPR)         e ⇒ begin fv = e, tracer!{add, fv}, fv end
otherwise
  where        $(e, \_) = last(PathAfter) \wedge fv = fv()$
```

Algorithm 1: Instrumentation rules for tracing.

the pattern of an expression, this expression needs a special treatment in the instrumentation. Let us show the problem with an example. Consider a POI inside a pattern {1,POI,3}. If the execution tries to match it with {2,2,3} nothing is sent to the tracing server because the POI is never evaluated. Contrarily, if it tries to match it with {1,2,4} we send the value 2 to the tracing server. Note that the matching fails in both cases, but due to the evaluation order, the POI is actually evaluated (and it succeeds) in the second case. There is an interesting third case that happens when the `POI` has a value, e.g., 3, and the matching with {1,4,4} is tried. In this case, although the matching at the `POI` fails, we send the value 4 to the tracing server. We could also send its actual value, i.e., 3. This is just a design decision, but we think that including the value that produced the mismatch could be more useful to find the source of a discrepancy. We call *target expression* to those expressions that need a special treatment in the instrumentation as the previously described one. In Erlang, these target expressions are pattern matchings, list comprehensions, and expressions with clauses (i.e., `case`, `if`, `functions`, ...). The goal of this step is to divide the AST path into two subpaths (`PathBefore`, `PathAfter`). `PathBefore` yields from the root to the deepest target expression (included), and `PathAfter` yields from the first children of the target expression to the AST identifier of the POI.

Finally, the last step is the one in charge of performing the actual instrumentation. The `PathBefore` path is used to traverse the tree until the deepest target expression that contains the AST identifier is reached. At this point, five rules (described below) are used to transform the code by using `PathAfter`. Finally, `PathBefore` is traversed backwards to update the AST of the targeted function. The five rules are depicted in Algorithm 1. The first four rules are mutually exclusive, and when none of them can be applied, the rule `(EXPR)` is applied. Rule `(LEFT_PM)` is fired when the POI is in the pattern of a pattern-matching expression. Rule `(PAT_GEN_LC)` is used to transform a list comprehension when the POI is in the pattern of a generator. Finally, rules `(CLAUSE_PAT)` (function clauses need an additional transformation that consists in storing all the parameters inside a tuple so that they could be used in case expressions) and `(CLAUSE_GUARD)` transform an expression with clauses when the POI is in the pattern or in the guard of one of its clauses, respectively. In the rules, we use the underline symbol (_) to represent a value that is not used. There are several functions used in the rules that need to be introduced. Functions $hd(l)$, $tl(l)$, $length(l)$, and $last(l)$ return the head, the tail, the length, and the last element of the list l, respectively. Function $pos(e)$ returns the child index of an expression e, i.e., its index in the list of children of its parent. Function $is\_bound(e)$ returns `true` if e is bounded according to the AST binding annotations (see step (1)). Functions $clauses(e)$ and $change\_clauses(e, clauses)$ obtain and modify the clauses of e, respectively. Function $fv()$ builds a free variable. Finally, there is a key function named pfv, introduced in (1), that transforms a pattern so that the constraints after the POI do not inhibit the sending call. This is done by replacing all the terms on the right of the POI with free variables that are built using fv function. Unbound variables on the left and also in the POI are replaced by fresh variables to avoid the shadowing of the original variables. In the pfv function, $children(e)$ and $change\_children(e, children)$ are used to obtain and modify the children of expression e, respectively. In this function, lists are represented with the head-tail notation $(h : t)$.

Function pfv

$$pfv(p, path) = \begin{cases} (poi, poi', p'') & \text{if } path = [(poi, pos)] \\ & \text{where } poi' = fv() \land p' = fv\_from(pos, p) \\ & \land p'' = p'_1 \cdots p'_{pos-1}, poi', p'_{pos+1} \cdots p'_{length(p)} \\ (poi, poi', p''') & \text{otherwise} \\ & \text{where } (\_, pos) = hd(path) \land p' = fv\_from(pos, p) \\ & \land (poi, poi', p'') = pfv(p'_{pos}, tl(path)) \\ & \land p''' = p'_1 \cdots p'_{pos-1}, p'', p'_{pos+1} \cdots p'_{length(p)} \end{cases}$$

$$fv\_from(pos, p) = p'_1 \cdots p'_{pos}, fv()_{pos+1} \cdots fv()_{length(p)} \quad \text{where } (p'_1 \cdots p'_{pos}, \_) = cv(p_1 \cdots p_{pos}, [\,]) \tag{1}$$

$$cv(list, map) \begin{cases} ([\,], map) & \text{if } list = [\,] \\ ((fv : p'_t), map') & \text{if } list = (p_h : p_t) \land is\_var(p_h) \land \neg\, is\_bound(p_h) \\ & \text{where } fv = fv() \land (p'_t, map') = cv(p_t, map \cup \{p_h \mapsto fv\}) \\ ((fv_{map} : p'_t), map') & \text{if } list = (p_h : p_t) \land is\_var(p_h) \land p_h \mapsto fv_{map} \in map \\ & \text{where } (p'_t, map') = cv(p_t, map) \\ ((p'_h : p'_t), map'') & \text{otherwise} \\ & \text{where } (p_h : p_t) = list \land (children'_{p_h}, map') = cv(children(p_h), map) \\ & \land p'_h = change\_children(p_h, children'_{p_h}) \\ & \land (p'_t, map'') = cv(p_t, map') \end{cases}$$

3.3. Test Case Generation Using ITC Mutation. The ITC generation phase uses `CutEr` because it implements sophisticated concolic analyses with the goal of achieving 100% branch coverage. However, sometimes these analyses require too much time and we have to abort its execution. This means that, after executing `CutEr`, we might have only the ITC that we provided to `CutEr`. Moreover, even when `CutEr` generates ITC with a 100% branch coverage, they can be insufficient. For instance, if the expression `Z = X - Y` is replaced in a new version of the code with `Z = X + Y`, a single test case that executes both of them with `Y = 0` will not detect any difference. More values for `Y` are needed to detect the behaviour change in this expression.

Therefore, to increase the reliability of the test suite, we complement the ITCs produced by CutEr with a test mutation technique. Using a mutation technique is much better than using, e.g., only the PropEr generator to randomly synthesize new test cases (this statement is clarified by the results obtained in Section 9), because full-random test cases would produce many useless test cases (i.e., test cases that do not execute the POI). In contrast, the use of a test mutation technique increases the probability of generating test cases that execute the POI (because only those test cases that execute the POI are mutated). The function that generates the test cases is depicted in (2). The result of the function is a map from the different obtained traces to the set of ITCs that produce them. The first call to this function is $tgen(top, cuter\_tests, \emptyset)$, where top is a user-defined limit of the desired number of test cases (in SecEr, it is possible to alternatively use a timeout to stop the test case generation) and $cuter\_tests$ are the test cases that CutEr generates (which could be an empty set). Function $tgen$ uses the auxiliary functions $proper\_gen$, $trace$, and mut. The function $proper\_gen()$ simply calls PropEr to generate a new test case, while function $trace(input)$ obtains the corresponding trace when the ITC $input$ is executed. The size of a map, $size(map)$, is the total amount of elements stored in all lists that belong to the map. Finally, function $mut(input)$ obtains a set of mutations for the ITC $input$, where, for each argument in $input$, a new test case is generated by replacing the argument with a randomly generated value, using PropEr (note that we are using PropEr to replace only one argument instead of all arguments. The latter is the full-random test case generation explained above), and leaving the rest of the arguments unchanged.

Test Case Generation Function

$$tgen(top, pending, map) = \begin{cases} map & \text{if } size(map) \geq top \\ tgen(top, pending', map') & \text{if } size(map) < top \\ & \wedge\ \exists\ input \in pending \mid trace(input) \mapsto \_ \notin map \\ & \text{where } pending' = (pending \cup mut(input)) \setminus \{input\} \\ & \wedge\ map' = map \cup \{trace(input) \mapsto \{input\}\} \\ tgen(top, \{proper\_gen()\}, map') & \text{if } size(map) < top \\ & \wedge\ \nexists\ input \in pending \mid trace(input) \mapsto \_ \notin map \\ & \text{where } map' = map \\ & \cup \{ trace(input_p) \mapsto (\{input_p\} \cup inputs_{tp}) \\ & \mid input_p \in pending \wedge trace(input_p) \mapsto inputs_{tp} \in map \} \end{cases} \quad (2)$$

Therefore, our mutation technique is able to generate tests more focused on our goal, i.e., maximizing the number of times a POI is executed. Due to the random generation, a mutant can produce repeated ITCs (which are not reexecuted). It can also produce ITCs whose trace has been previously found (then they are not mutated). Moreover, a mutant can produce unexpected ITCs or execution errors. These test cases are not considered as invalid but, contrarily, they are desirable because they allow us to check that the behaviour is also preserved in those cases. Finally, CutEr is an optional tool that can help to improve the resulting test cases by contributing with an initial test cases suite with high coverage.

4. Extending the Approach to Include Multiple POIs

The previous sections introduced a methodology to automatically obtain traces from a given POI. An extension of this methodology to multiple POIs enables several new features like a fine-grained testing, or checking multiple functionalities at once. However, it introduces new challenges to be overcome.

In order to extend the approach for multiple POIs, we need to perform some modifications in some of the steps of the single-POI approach. The flow is exactly the same as the one depicted in Section 2, but we need to modify some of its internals. There is no need for modifications in all the process described in Section 3.1, since this process depends on the input functions that, in our approach, are shared by all the POIs (we plan to explore in future approaches the idea of defining individual input functions for each POI). On the other hand, we need to introduce changes in the processes described in Sections 3.2 and 3.3.

The tracing method introduced in Section 3.2 needs to be slightly redefined here. This section defined 4 steps that started from a source code and a POI and ended in an instrumented version of the source code that is able to communicate traces. Therefore, the only change needed is that, instead of having only one POI, we have more than one. In order to deal with this change, we follow the same 4 steps but change the way in which they are applied. In the single-POI approach, they are applied sequentially, but here we need to iterate some of them. Concretely, steps (1) and (2) are done only once in the whole process while the rest of the steps are done once for each POI. The result of step (2) is now a set of *POI-AST identifier* relations instead of a single one. Then, we iterate the obtained AST identifiers applying steps (3) and (4) sequentially. Note that although the result of step (4) is a new AST, we are still able to find the AST identifiers of the subsequent POIs since the transformations do not destroy any node of the original AST; instead they only move them inside a new expression. This justifies the double search design performed in steps (2) and (3). If we tried to search for the POI in a modified AST, we could be unable to find it. In contrast, AST identifiers ensure that it can always be found.

In the multiple-POI approach, there is also a justification of why the identifiers are numbers and why the identification process is done with a postorder traversal. First of all, there is one question that should be discussed: is the order in which the POIs are processed important? The answer is yes, because the user could define a POI that includes another POI inside; e.g., POI_1 is the whole tuple `{X,Y}` and POI_2 is `X`. This scenario would be problematic when the POI-inside-POI case occurs inside a pattern due to the way we instrument the code. If we instrumented first POI_1, its trace would be sent before the one of POI_2. Note that this is not correct since POI_2 is evaluated first; therefore, it should be traced first. This justifies the use of a postorder traversal, where the identifier of a node is maximal in its subtree. Thus, as the AST identifiers are numbers and their order is convenient in our context, we can order the AST identifiers obtained from the POIs before starting the transformation loop.

The test case generation phase introduced in Section 3.3 is also affected by the inclusion of multiple POIs. In the original definition, the traces were a sequence of values, and therefore it was easy to check whether a trace had appeared in a previously executed test. However, with multiple POIs, the trace is not such a simple sequence, as the traced values can be obtained from different POIs along the execution. Therefore, we need a more sophisticated way to determine the equality of the traces. The next section explains in detail how we can achieve this goal.

5. Determining the Trace Equality with Multiple POIs

We present in this section several alternatives to compare traces that contain values from multiple POIs. Concretely, we explain the three default comparison functions provided in our approach. In our setting, we also allow a user to define their own comparison functions enabling all needed types of comparison.

A trace of a POI is defined as the sequence of values that the POI is evaluated to during an execution. It has been represented with $trace(input)$, which obtains the corresponding trace when the ITC is executed. In the multiple-POI approach, we need to redefine the notion of $trace$ to also include the POIs that originated the values of the trace. In order to maintain the execution order, the trace is still a sequence, but instead of simple values it contains tuples of the form $(POI, value)$. In this way, $trace(input)$ will contain all the values traced for all the POIs defined by the user, preserving their execution order. This is achieved by slightly modifying the rules in Algorithm 1 to include the POI reference when sending the value to the tracer.

Once $trace(input)$ includes all the sequences of values generated during the execution for each POI, we need a way to compare them. Note that the standard equality function is perfectly valid for comparing traces during the test case generation phase (Section 3.3), because all of them come from the same source code. However, it is no longer valid for comparing program versions since POIs can differ in the original program and the modified one. Therefore, we additionally need to define a relation between POIs. This relation, which we represent with R_{POIs}, is automatically built from the input provided by the user. It is a set that contains tuples of the form (POI_{old}, POI_{new}). Therefore, a simple equality function to compare two traces obtained from different versions of a program can be defined as follows:

$$equal\left(trace_{old}, trace_{new}, R_{POIs}\right) = \begin{cases} true & \text{if } trace_{old} = [\,] \wedge trace_{new} = [\,] \\ equal\left(trace'_{old}, trace'_{new}, R_{POIs}\right) & \\ & \text{if } trace_{old} = \left(\left(POI_{old}, v_{old}\right) : trace'_{old}\right) \\ & \wedge\ trace_{new} = \left(\left(POI_{new}, v_{new}\right) : trace'_{new}\right) \\ & \wedge\ v_{old} = v_{new} \ \wedge \left(POI_{old}, POI_{new}\right) \in R_{POIs} \\ false & \text{otherwise} \end{cases} \quad (3)$$

This equality function is useful when the user is interested in comparing the traces interleaved (i.e., when their interleaved execution is relevant). However, in some scenarios, the user can be interested in relaxing the interleaving constraint and comparing the traces independently. This can be achieved by building a mapping from POIs to sequences of values in the following way:

$$trace\,(input, POI) = [v \mid (POI, v) \in trace\,(input)] \quad (4)$$

The order is assumed to be preserved in the produced sequences. Using these sequences, we can define an alternative equality function as follows:

$$\begin{aligned} &equal\,(trace_{old}, trace_{new}, R_{POIs}) \\ &= \bigwedge_{(POI_{old}, POI_{new}) \in R_{POIs}} trace\,(input, POI_{old}) \\ &= trace\,(input, POI_{new}) \end{aligned} \quad (5)$$

There is a third equality relation that could be useful in certain cases. Suppose that we detect some duplicated code, so we build a new version of the code where all the repeated code has been refactored to a single code. If we want to test whether the behaviour is kept, we need to define a relation where multiple POIs in the old version are associated with a single POI in the new version. This is represented in our approach adding to R_{POIs} several tuples of the form (POI_{old_1}, POI_{new}), (POI_{old_2}, POI_{new}), and so forth. A similar scenario can happen when a functionality of the original code is split in several parts in the new code (an example of this scenario is the use case presented in Section 6.4). In both cases, a special treatment is needed for this type of relations. In order to do this, we define a generalisation of the previous *equal* function where this kind of relations is taken into account. The first step is to extract all the POIs in R_{POIs}.

$$\begin{aligned} pois\,(R_{POIs}) = &\{POI_1 \mid (POI_1, POI_2) \in R_{POIs}\} \\ &\cup \{POI_2 \mid (POI_1, POI_2) \in R_{POIs}\} \end{aligned} \quad (6)$$

Then, we can define the set of POIs related to a given POI in R_{POIs}.

$$\begin{aligned} rel\,(POI, R_{POIs}) = &\left\{POI' \mid \left(POI, POI'\right) \in R_{POIs}\right\} \\ &\cup \left\{POI' \mid \left(POI', POI\right) \in R_{POIs}\right\} \end{aligned} \quad (7)$$

Finally, we need a new trace function that returns a single trace of values that are obtained from all the POIs related to a POI in R_{POIs}.

$$\begin{aligned} trace\_rel\,(input, POI, R_{POIs}) = &\left[v \mid \left(POI', v\right)\right. \\ &\left.\in trace\,(input) \wedge POI' \in rel\,(POI, R_{POIs})\right] \end{aligned} \quad (8)$$

We can now define an equality function that is able to deal with replicated POIs.

$$\begin{aligned} &equal\,(trace_{old}, trace_{new}, R_{POIs}) \\ &= \bigwedge_{POI \in pois(R_{POIs})} trace\,(input, POI) \\ &= trace\_rel\,(input, POI, R_{POIs}) \end{aligned} \quad (9)$$

In case a user needs a more intricate equality function, we provide in our tool a way to define a custom equality function, which should contain the parameters $trace_{old}$ and $trace_{new}$ (the relation R_{POIs} is not a parameter in the user function because it is originally provided by the user). Hence, the user can decide whether the generated traces can be considered as equal or not.

Equality functions constitute a new parameter of the approach that determines how the traces should be compared. For the comparison of versions using multiple POIs, it is mandatory to provide such a function, while for the test case generation phase depicted in Section 3.3 it can be optional. In the second case, the user could be interested in obtaining more sophisticated test cases by providing their own equality function. In order to enable this option, an additional parameter is needed for the *tgen* function. This parameter will contain the equality function that should be used when checking if a trace has been previously computed.

6. The SecEr Tool

In this section, we describe `SecEr` and how to use it to automatically obtain test cases from a source code. Then, we present some use cases that illustrate how `SecEr` can be used to check behavioural changes in the code.

6.1. Tool Description. Given two versions of the same program, `SecEr` is able to automatically generate a test suite that checks the behaviour of a set of POIs and reports the discrepancies. Listing 3 shows the `SecEr` command.

If we want to perform a comparison between two programs, we just need to provide a list of related POIs from both programs. For instance,

```
./secer -pois "[{{‘happy0.erl’,4,‘call’,1},{‘happy1.erl’,27,‘call’,1}},
{{‘happy0.erl’,9,‘call’,1},{‘happy1.erl’,18,‘call’,1}}]" -to 10
```

Because the same POIs are often compared as the program evolves, it is a good idea to record them together with the input functions for future uses. For this reason, the user can save in a file(e.g., `pois.erl`) the POIs of interest and

```
(1) $ ./secer -pois "LIST_OF_POIS" [-funs "INPUT_FUNCTIONS"] -to TIMEOUT [-cfun "COMPARISON_FUN"]
```

Listing 3: `SecEr` command format.

their relations and define a function that returns them (e.g., `pois:rel/0`). Hence, they can simply invoke this function with

```
./secer -pois "pois:rel()" -to 15
```

By default, the traces of the POIs are compared using the standard equality, as it is defined by the first comparison function in Section 5. Alternatively, we can customize our comparison defining a comparison function (`COMPARISON_FUN`). The comparison function defined by the user must be a function with two parameters (the old and the new traces). We also provide a library with some common comparison functions, like, for instance, the one that compares the traces independently (`secer:independent`) as it is described in the second and third functions (depending on the POI relations) in Section 5.

Example 2. Consider two POIs, POI_1 and POI_2, in the original code and their counterparts POI'_1 and POI'_2 in the new code. If an execution executes the POIs in the following order:

original code: $POI_1 = 42 \cdots POI_1 = 43 \cdots POI_1 = 50 \cdots POI_2 = 0$,

new code: $POI'_1 = 42 \cdots POI'_1 = 43 \cdots POI'_2 = 0 \cdots POI'_1 = 50$,

`SecEr` records the traces:

Trace $POI_1 = [42, 43, 50]$

Trace $POI'_1 = [42, 43, 50]$

Trace $POI_2 = [0]$

Trace $POI'_1 = [0]$

If we execute `SecEr` with flag `-cfun "secer:independent()"`, `SecEr` will report that there are no discrepancies between the POIs. In contrast, if no flag is specified, `SecEr` will take into account the execution order of the POIs, and it will alert that this order has changed.

Note that, in the implementation, the limit used to stop generating test cases is a timeout, while the formalization of the technique uses a number to specify the amount of test cases that must be generated (see variable *top* in Section 3.3). This is not a limitation, but a design decision to increase the usability of the tool. The user cannot know *a priori* how much time it could take to generate an arbitrary number of test cases. Hence, to make the tool predictable and give the user control over the computation time, we use a timeout. Thus, `SecEr` generates as many test cases as the specified timeout permits.

6.2. Defining a Configuration File. `SecEr` permits using configuration files that can be reused in different invocations. A configuration file contains functions that can be invoked from the `SecEr` command. For instance, the following command uses functions `rel/0`, `funs/0`, and `cf_length/2` of module `test_happy`:

```
./secer -pois "test_happy:rel()" -fun "test_happy:funs()" -to
5 -cfun "test_happy:cf_length"
```

In Algorithm 2, we can see that POIs can be specified in two different ways: (i) with a tuple with the format `{'FileName', Line, Expression` (expressions with a specific name, e.g., variables, will be denoted by a tuple `{var,'VarName'}`. Note that expressions denoted by reserved Erlang words, e.g., `case` or `if`, must be specified in single quotation marks), `Occurrence}` as shown in Algorithm 2 line (5) and (ii) with a tuple `{'FileName', {InitialLine, InitialColumn}, {FinalLine, FinalColumn}}`(POIs of this type are internally translated to POIs of the first type) representing the initial and final line and column in the specified file; this approach is shown in Algorithm 2 line (7).

The `LIST_OF_POIS` parameter is provided by function `rel/0` (see line (19)). It returns an Erlang list of well defined POIs (or pairs of POIs). The `INPUT_FUNCTIONS` parameter is provided by function `funs/0` (see line (26)). It returns a string containing a list with the desired input functions. The `COMPARISON_FUN` parameter is provided by function `cf_length/2` (see line (29)). It receives two arguments, each of which is a list of tuples that contains a POI and a value. This function must return `true`, `false`, or a tuple with `false` and an error message to customize the error.

6.3. Use Case 1: Happy Numbers. In this section, we further develop Example 1 to show how `SecEr` can check the behaviour preservation in the happy numbers programs (see Listings 1 and 2). First of all, to unify the interfaces of both programs, in the `happy0` module (Listing 1), we have replaced `main/0` with `main/2` making it applicable for a more general case. Moreover, in both modules, we have added a type specification (represented with `spec` in Erlang) in order to obtain more representative test cases. To run `SecEr` we use the configuration file defined in Algorithm 2.

```
-module(test_happy).
-compile(export_all).

poiResultOld() ->
    {‘happy0.erl’,4,call,1}.
poiResultNew() ->
    {‘happy1.erl’,{27,2},{27,16}}.

poiIsHappyOld() ->
    {‘happy0.erl’,9,call,1}.
poiIsHappyNew() ->
    {‘happy1.erl’,18,call,1}.

poiXOld() ->
    {‘happy0.erl’,9,{var,’X’},1}.
poiXNew() ->
    {‘happy1.erl’,18,{var,’X’},1}.

rel() ->
    [{poiResultOld(),poiResultNew()}].
relIsHappy() ->
    [{poiIsHappyOld(),poiIsHappyNew()}].
relX() ->
    [{poiXOld(),poiXNew()}].

funs() ->
    "[main/2]".

cf_length(TO,TN) ->
    ZippedList = lists:zip(TO,TN),
    lists:foldl(
      fun
        (_,{false,Msg,PO,PN}) ->
            {false,Msg,PO,PN};
        ({{_,VO},{_,VN}}, _) when length(VN) < length(VO) ->
            true;
        ({{PO,_},{PN,_}},_) ->
            {false,"Invalid Length",PO,PN}
      end,
      true,
      ZippedList).
```

Algorithm 2: Configuration file to test happy modules.

Listing 4 shows the execution of SecEr when comparing both implementations of the program with a timeout of 15 seconds. The selected POIs are the same POIs mentioned in Section 1. They are the call in line (4) in Listing 1 and the call in line (27) in Listing 2. As we can see, the execution of both implementations behaves identically with respect to the selected POIs in the 1142 generated test cases.

In order to see the output of the tool when the behaviours of the two compared programs differ, we have introduced an error inside the is_happy/2 function of happy1 module (Listing 2). The error is introduced by replacing the whole line (4) with `X < 10 -> false;`. With this change, the behaviour of both programs differs. When the user runs SecEr using the previous POI, it produces the error report shown in Listing 5. From this information, the user may decide to use as new POIs all nonrecursive function calls inside happy_list/3 (for Listing 1) and happy/3 (for Listing 2) functions. With this decision it can discard whether the error comes from the function itself or from the called function. Therefore, the new POIs are the call in line (4) in Listing 1 and the call in line (27) in Listing 2. SecEr's output for these POIs is shown in Listing 6. SecEr reports that the POI was executed several times and in some executions the values of the POI differed. SecEr also reports a counterexample: main(4,2) compute different values. Because the current POIs are the results of calling a function that should be equivalent in both codes, there are two possible sources of the discrepancy: either the common argument in both versions of is_happy (i.e., X) is taking different values during the execution, or something executed by is_happy produces the discrepancies. Listing 7 shows the

```
$ ./secer -pois "test_happy:rel()" -funs "test_happy:funs()" -to 15
Function: main/2
----------------------------
Generated test cases: 1142
Both versions of the program generate identical traces for the defined points of interest
```

Listing 4: SecEr reports that no discrepancies exist.

```
$ ./secer -pois "test_happy:rel()" -funs "test_happy:funs()" -to 15
Function: main/2
----------------------------
Generated test cases: 1143
Mismatching test cases: 45 (3.93%)
    POIs comparison:
        + {{'happy0.erl',4,call,1},
          {'happy1.erl',27,call,1}}
                      Unexpected trace value => 45 Errors
                      Example call: main(5,8)
------ Detected Error ------
Call: main(5,8)
Error Type: Unexpected trace value
POI: ({'happy0.erl',4,call,1}) trace:
             [[7,10,13,19,23,28,31,32]]
POI: ({'happy1.erl',27,call,1}) trace:
             [[10,13,19,23,28,31,32,44]]
----------------------------
```

Listing 5: Result replacing line (4) with `X < 10 -> false`.

```
$ ./secer -pois "test_happy:relIsHappy()" -funs "test_happy:funs()" -to 15
Function: main/2
----------------------------
Generated test cases: 1151
Mismatching test cases: 39 (3.38%)
     POIs comparison:
           + {{'happy0.erl',9,call,1},
             {'happy1.erl',18,call,1}}
                       Unexpected trace value => 39 Errors
                       Example call: main(4,2)
------ Detected Error ------
Call: main(4,2)
Error detected: Unexpected trace value
POI: ({'happy0.erl',9,call,1}) trace:
            [false,false,false,true,false,false,true]
POI: ({'happy1.erl',18,call,1}) trace:
            [false,false,false,false,false,false,true,false,false,true]
----------------------------
```

Listing 6: SecEr reports discrepancies between `is_happy` call as POI.

```
$ ./secer -pois "test_happy:relX()" -funs "test_happy:funs()" -to 15
Function: main/2
----------------------------
Generated test cases: 1624
Mismatching test cases: 64 (3.94%)
     POIs comparison:
           + {{'happy0.erl',9,{var,'X'},1},
              {'happy1.erl',18,{var,'X'},1}}
                       The second trace is longer => 64 Errors
                       Example call: main(6,3)
------ Detected Error ------
Call: main(6,3)
Error Type: The second trace is longer
POI: ({'happy0.erl',9,{var,'X'},1}) trace:
           [6,7,8,9,10,11,12,13]
POI: ({'happy1.erl',18,{var,'X'},1}) trace:
           [6,7,8,9,10,11,12,13,14,15,16,17,18,19]
----------------------------
```

Listing 7: `SecEr` reports discrepancies using variable `X` as the POI.

```
$ ./secer -pois "test_string:rel()" -funs "test_string:funs()" -to 15
Function: tokens/2
----------------------------
Generated test cases: 118878
Both versions of the program generate identical traces for the defined points of interest
```

Listing 8: `SecEr` reports that no discrepancies exist.

report provided by `SecEr` when selecting variable `X` as the POI. The reported discrepancy indicates that both traces are the same until a point in the execution where the version in Listing 2 continues producing values. This behaviour is the expected one, because the result of `is_happy` has an influence on the number of times the call is executed. Therefore, the user can conclude that the arguments do not produce the discrepancy and the source of the discrepancy is inside the `is_happy` function.

Listings 5, 6, and 7 show that `SecEr` detects the errors and produces a concrete call that reveals these errors showing the effects. With more POIs, the user can obtain more feedback to help in finding the source of a bug. Clearly, with this information we can now ensure that the symptoms of the errors are observable in function `is_happy`.

6.4. Use Case 2: An Improvement of the `string:tokens/2` *Function.* In this case of study, we consider a real commit of the Erlang/OTP distribution that improved the performance of the `string:tokens/2` function. Algorithm 3 shows the code of the original and the improved versions. The differences introduced in this commit can be consulted here:

https://github.com/erlang/otp/commit/53288b441ec721ce3bbdcc4ad65b75e11acc5e1b

The improvement consists in two main changes. The first one is a general improvement obtained by reversing the input string (the one that is going to be tokenized) at the beginning of the process. The second one improves the cases where the separators list has only one element. The algorithm uses two auxiliary functions in both cases, so its structure is kept between versions. However, the optimized version duplicates these functions to cover the single-element list of separators and the rest of the cases separately.

We can use `SecEr` to check whether the behaviour of both versions is the same. In order to do this, we can define as POIs the final expressions of the `tokens/2` function in each version, i.e., the call to `tokens1/3` function in the original version and the whole case in the optimized version. The input function should be `tokens/2` because the changes were introduced to improve it (see Algorithm 4). This is enough to check that both versions preserve the same behaviour (see Listing 8).

We can now consider a hypothetical scenario where an error was introduced in the aforementioned commit.

```
$ ./secer -pois "test_string:rel()" -funs "test_string:funs()" -to 15
Function: tokens/2
-----------------------------
Generated test cases: 105088
Mismatching test cases: 72260 (68.76%)
      POIs comparison:
          + {{'string0.erl',2,call,1},
             {'string1.erl',2,case,1}}
                     Unexpected trace value => 72260 Errors
                     Example call: tokens([9],[5,19,3,2])
------ Detected Error ------
Call: tokens([9],[5,19,3,2])
Error Type: Unexpected trace value
POI: ({'string0.erl',2,call,1}) trace:
           [[[9]]]
POI: ({'string1.erl',2,case,1}) trace:
           [[9,[9]]]
-----------------------------
```

Listing 9: SecEr reports discrepancies after modifying optimized string.erl.

Suppose that line (30) in Algorithm 3 (optimized version) is replaced by the following expression:

```
[C | tokens_multiple_2(S, Seps, Toks, [C])]
```

In this scenario, SecEr reports that some of the traces differ (see Listing 9).

We can add more POIs to try to isolate the error. For instance, the calls in lines (6), (8), (15), and (17) of Algorithm 3 (original version) are a good choice in this case, as it can help in checking that the intermediate results are the expected ones. This selection of POIs is also interesting because each POI in the original version is duplicated in the optimized version. For instance, line (6) in the original version corresponds to lines (14) and (28) in the optimized version. This relation is specified by defining two tuples of POIs: ((original version, line (6)), (optimized version, line (14))) and ((original version, line (6)), (optimized version, line (28))). There is an additional issue that should be considered before calling to SecEr. As one of the improvements was to reverse the input string beforehand, the execution order is different in the optimized version. This means that the traces computed by SecEr for the two versions will surely differ. To solve this inconvenience we can invoke SecEr with the flag -cfun "secer:independent()" activated. Thus, SecEr will ignore the order of the traces computed for different POIs. The result produced by SecEr with this configuration is shown in Listings 10 and 11.

The reported error is effectively pointing to a POI which is a call to the function that produced the error. This scenario demonstrates how useful SecEr can be to find the source of the discrepancies. Another interesting feature of the report is the categorization of errors. In this particular example, there are two kinds of errors: errors related to the length of the trace, where one trace is a prefix of the other, and errors related to the values of the trace, where the values of each trace are completely different. In Listing 10 (and also in Listing 11), the first error detected by SecEr is a length error while the third error is a value error. Moreover, there are errors indicating that some POI was not executed (i.e., it produced an empty trace, represented in the listings by the trace []). This is because some of the POIs are not completely symmetrical in this example. Concretely, when the separators list (the second parameter of the function string:tokens/2) is empty, the algorithms behave differently. As this is not a really interesting test case input, we could use an Erlang's type specifier (spec) to constrain this second parameter to be a nonempty list. An alternative is to use a comparison function that takes into account this particularity. Therefore, by avoiding ITCs of this type, the reported errors will be only related to the actual error.

Now, we can return to the original scenario to explore other interesting uses of SecEr. As we mentioned, this commit improved the performance of function string:tokens/2. We can use SecEr to check that this improvement actually exists. In contrast to previous examples, this would need two small modifications. In concrete, the first one is to replace line (2) of Algorithm 3 (original version) with the following expressions:

```
(1) Start = os:timestamp(),

(2) Res = tokens1(S, Seps, []),
```

Original version

```
tokens(S, Seps) ->
  tokens1(S, Seps, []).
tokens1([C|S], Seps, Toks) ->
 case member(C, Seps) of
  true ->
   tokens1(S, Seps, Toks);
  false ->
   tokens2(S, Seps, Toks, [C])
 end;
tokens1([], _Seps, Toks) ->
 reverse(Toks).
tokens2([C|S], Seps, Toks, Cs) ->
 case member(C, Seps) of
  true ->
   tokens1(S, Seps, [reverse(Cs)|Toks]);
  false ->
   tokens2(S, Seps, Toks, [C|Cs])
 end;
tokens2([], _Seps, Toks, Cs) ->
 reverse([reverse(Cs)|Toks]).
```

Optimized version

```
tokens(S, Seps) ->
  case Seps of
    [] ->
      case S of
        [] -> [];
        [_|_] -> [S]
      end;
    [C] ->
        tokens_single_1(reverse(S), C, []);
    [_|_] ->
tokens_multiple_1(reverse(S), Seps, [])
 end.
tokens_single_1([Sep|S], Sep, Toks) ->
 tokens_single_1(S, Sep, Toks);
tokens_single_1([C|S], Sep, Toks) ->
 tokens_single_2(S, Sep, Toks, [C]);
tokens_single_1([], _, Toks) ->
 Toks.
tokens_single_2([Sep|S], Sep, Toks, Tok) ->
 tokens_single_1(S, Sep, [Tok|Toks]);
tokens_single_2([C|S], Sep, Toks, Tok) ->
 tokens_single_2(S, Sep, Toks, [C|Tok]);
tokens_single_2([], _Sep, Toks, Tok) ->
 [Tok|Toks].
tokens_multiple_1([C|S], Seps, Toks) ->
 case member(C, Seps) of
  true ->
   tokens_multiple_1(S, Seps, Toks);
  false ->
   tokens_multiple_2(S, Seps, Toks, [C])
 end;
tokens_multiple_1([], _Seps, Toks) ->
 Toks.
tokens_multiple_2([C|S], Seps, Toks, Tok) ->
 case member(C, Seps) of
  true ->
   tokens_multiple_1(S, Seps, [Tok|Toks]);
  false ->
   tokens_multiple_2(S, Seps, Toks, [C|Tok])
 end;
tokens_multiple_2([], _Seps, Toks, Tok) ->
 [Tok|Toks].
```

ALGORITHM 3: string.erl (original and optimized versions).

```
-module(test_string).
-compile(export_all).

poiOld() ->
    {'string0.erl', 2, call, 1}.
poiNew() ->
    {'string1.erl', 2, 'case', 1}.

poiOldError() ->
    {'string0.erl', 6, call, 1}.
poiNewError1() ->
    {'string1.erl', 14, call, 1}.
poiNewError2() ->
    {'string1.erl', 28, call, 1}.


rel() ->
    [{poiOld(), poiNew()}].
relError() ->
    [{poiOldError(), poiNewError1()},
    {poiOldError(), poiNewError2()}].

funs() ->
    "[tokens/2]".
```

Algorithm 4: Configuration file to test string modules.

(3) `timer:now_diff(os:timestamp(), Start),`

(4) `Res.`

The second change is similar and consists in assigning to variable `Res` the result of the case expression in line (2) of Algorithm 3 (optimized version). To invoke `SecEr`, the first step is to choose a POI. We can select in both codes the expression `timer:now_diff(os:timestamp(), Start)` which computes the total time. Then, we need to use the comparison function `secer:comp_perf/1` that returns true when the execution time of the optimized version is smaller than or equal to the execution time of the original version. Note that we used the parameter of this function which defines a threshold (of 30 μs in this case) to filter those evaluations whose execution times are almost equal. We discard, in this way, downgrade alerts that are not significative. The report of `SecEr` (see Listing 12) shows that effectively there is an efficiency improvement in the optimized version; i.e., the time used by the optimized version is less than the one of the original version in all 127573 generated test cases.

We can create a different scenario where the performance has not been improved. We introduce a simple change to simulate this case by replacing line (28) of Algorithm 3 (optimized version) with the following line:

```
timer:sleep(5), tokens_multiple_1(S, Seps, Toks);
```

This will introduce a delay of 5 milliseconds before calling function `tokens_multiple_1/3` affecting consequently the overall performance. We can run `SecEr` again with this version of the code and the report (Listing 13) reveals two relevant problems: (i) many test cases show a worse performance in the new code than in the original code (those cases affected by the downgrade) and (ii) fewer test cases are being generated by `SecEr` due to the sleep time introduced in the execution.

The user could now easily introduce more time measures in the code and rerun `SecEr` to find the source of the downgrade in the performance.

6.5. Use Case 3: Regression Bug Fixed in a Real Commit in `etorrent`. In this use case, we study a real commit in the `etorrent` GitHub repository, a repository with an implementation of a bittorrent client in Erlang. This commit can be consulted here:

https://github.com/edwardw/etorrent/commit/d9d8cc13bab2eaa1ce282971901b7a29bf9bc942

The commit corrects an error introduced in a previous commit (https://github.com/edwardw/etorrent/commit/a9340eb5b4e2da3cf08094d1f942bb31173f4011): the output of a decoding function is modified from a single variable

```
$ ./secer -pois "test_string:relError()" -funs "test_string:funs()" -to 15
Function: tokens/2
----------------------------
Generated test cases: 64458
Mismatching test cases: 31187 (48.38%)
      POIs comparison:
         + {{'string0.erl',6,call,1},
              {'string1.erl',14,call,1}}
                   The second trace is longer => 40 Errors
                   Example call: tokens([11,6,4,4],[4])
         + {{'string0.erl',6,call,1},
              {'string1.erl',14,call,1}}
                   The first trace is empty => 364 Errors
                   Example call: tokens([47,3,19,7,1,10,1,25,4,16],[16])
         + {{'string0.erl',6,call,1},
              {'string1.erl',28,call,1}}
                   Unexpected trace value => 18078 Errors
                   Example call: tokens([4,24,0,4,13,10,1,0],[2,8,12,1,0])
         + {{'string0.erl',6,call,1},
              {'string1.erl',28,call,1}}
                   The first trace is empty => 7991 Errors
                   Example call: tokens([13,7],[1,1,2,3,6,4,11,8,7])
         + {{'string0.erl',6,call,1},
              [{'string1.erl',28,call,1},
              {'string1.erl',14,call,1}]}
                   The first trace is longer => 3058 Errors
                   Example call: tokens([6,3,1,7,4,9,5,7,28],[1,10,46,3,4,8,34,6])
         + {{'string0.erl',6,call,1},
              [{'string1.erl',28,call,1},
              {'string1.erl',14,call,1}]}
                   The second trace is empty => 8231 Errors
                   Example call: tokens([12,1],[2,10,0,4,12,4,6,2,22])
```

Listing 10: SecEr reports discrepancies in the multiple-POI execution.

to a tuple containing the atom ok together with this value. This bug was found by the commit's authors using unit testing (EUnit in Erlang). Therefore, in this case, we do not use the test generation feature of SecEr, but instead we start from the test case that revealed the error. Therefore, the input function is the failing unit test case, and we take advantage of the multiple-POI approach, placing several POIs in the function called by the unit test case. There are two modules implied in the process. The fragments of the involved modules defining the affected functions are shown in Algorithm 5.

Therefore, the input function of SecEr is query_ping_0_test/0. This function calls to the decode_msg/1 function, so we place some POIs inside it to check its behaviour. In particular, we place one POI in each function call (lines (3) and (5)) and one POI in the return expression of the function (case expression in line (6)). All the parameters defined in this use case can be found in the configuration file in Algorithm 6. After placing these three POIs in both versions, we execute SecEr obtaining the result shown in Listing 14.

The results provided by SecEr show that the bug is located inside function etorrent_dht_net_old:decode/1. With a quick inspection of both versions of the decode function we can easily discover that the format of the return expression is different. Although we should be confident that the error is located in this expression, we can be completely sure by placing some POIs in this function. This can be easily done by adding the POIs to the configuration file.

After fixing an error, it is a good practice to rerun SecEr in order to verify that there are no more mismatches between the defined POIs. Remember that SecEr only reports the first mismatch found in the execution. In the first execution, if the function call to get_value/2 (line (5)) had also an error, it would have been omitted by the previous mismatch found in call to decode/1 (line (3)).

7. Using SecEr in a Concurrent Environment

Nondeterministic computations are one of the main obstacles for regression testing. In fact, they prevent us from comparing the results of a test case executed in different versions because the discrepancies found can be well produced by sources of nondeterminism such as concurrency. In some specific situations, however, we can still use SecEr to report whether the behaviour of a concurrent program is preserved.

```
------ Detected Error ------
Call: tokens([11,6,4,4],[4])
Error Type: The second trace is longer
POI: ({'string0.erl',6,call,1}) trace:
         [[[11,6]]]
POI: ({'string1.erl',14,call,1}) trace:
         [[[11,6]],[[11,6]]]
----------------------------
------ Detected Error ------
Call: tokens([47,3,19,7,1,10,1,25,4,16],[16])
Error Type: The first trace is empty
POI: ({'string0.erl',6,call,1}) trace:
         []
POI: ({'string1.erl',14,call,1}) trace:
         [[[47,3,19,7,1,10,1,25,4]]]
----------------------------
------ Detected Error ------
Call: tokens([4,24,0,4,13,10,1,0],[2,8,12,1,0])
Error Type: Unexpected trace value
POI: ({'string0.erl',6,call,1}) trace:
         [[[4,24],[4,13,10]]]
POI: ({'string1.erl',28,call,1}) trace:
         [[10,24,[4,24],[4,13,10]],[10,24,[4,24],[4,13,10]]]
----------------------------
------ Detected Error ------
Call: tokens([13,7],[1,1,2,3,6,4,11,8,7])
Error Type: The first trace is empty
POI: ({'string0.erl',6,call,1}) trace:
         []
POI: ({'string1.erl',28,call,1}) trace:
         [[13,[13]]]
----------------------------
------ Detected Error ------
Call: tokens([6,3,1,7,4,9,5,7,28],[1,10,46,3,4,8,34,6])
Error Type: The first trace is longer
POI: ({'string0.erl',6,call,1}) trace:
         [[[7],[9,5,7,28]],[[7],[9,5,7,28]],[[7],[9,5,7,28]]]
POI: ([{'string1.erl',28,call,1},
      {'string1.erl',14,call,1}]) trace:
         [[[7],[9,5,7,28]],[[7],[9,5,7,28]]]
----------------------------
------ Detected Error ------
Call: tokens([12,1],[2,10,0,4,12,4,6,2,22])
Error Type: The second trace is empty
POI: ({'string0.erl',6,call,1}) trace:
       [[[1]]]
POI: ([{'string1.erl',28,call,1},
      {'string1.erl',14,call,1}]) trace:
       []
----------------------------
```

Listing 11: SecEr reports discrepancies in the multiple-POI execution (cont.).

For instance, consider the client-server model depicted in Figure 4. In this simple example, a POI should not be placed in *Server*, because we cannot know *a priori* whether req_1 is going to be served before or after req_2, and this could have an impact on the traces obtained from that POI. However, we could place a POI in any of the clients, as long as the request is not affected by the state of the server. This is acceptable for many kinds of servers, but it is still a quite annoying limitation for many others.

However, in Erlang, as it is common in other languages, there is a high-level way to define a server. In particular, real Erlang programmers tend to use the Erlang-OTP's behaviour

```
$ ./secer -pois "[{'string0.erl', LINE_timer:now_diff, call, 1}, {'string1.erl', LINE_timer:now_diff,
call,  1}]"
          -funs "test_string:funs()" -to 15 -cfun "secer:comp_perf(30)"
Function: tokens/2
----------------------------
Generated test cases: 127573
Both versions of the program generate identical traces for the defined points of interest
```

Listing 12: SecEr reports the result of comparing the performance POI.

```
$ ./secer -pois "[{'string0.erl', LINE_timer:now_diff, call, 1}, {'string1.erl', LINE_timer:now_diff,
call,1}]"
          -funs "test_string:funs()" -to 15 -cfun "secer:comp_perf(30)"
Function: tokens/2
----------------------------
Generated test cases: 4587
Mismatching test cases: 1286 (28.03%)
     POIs comparison:
          + {{'string0.erl', LINE_timer:now_diff,call,1},
            {'string1.erl', LINE_timer:now_diff,call,1}}
                  Unexpected trace value => 1286 Errors
                  Example call: tokens([7,5,4,2,16,3,11,3],[4,2,9,2])
------ Detected Error ------
Call: tokens([7,5,4,2,16,3,11,3],[4,2,9,2])
Error Type: Slower Calculation
POI: ({'string0.erl', LINE_timer:now_diff,call,1}) trace:
          [2]
POI: ({'string1.erl', LINE_timer:now_diff,call,1}) trace:
          [5395]
----------------------------
```

Listing 13: SecEr reports discrepancies after entering the sleep expression.

named gen_server. By implementing this behaviour, the programmer is only defining the concrete behaviours of a server, leaving all the low-level aspects to the internals of the gen_server implementation. These concrete behaviours include how the server's state is initialized, how a concrete request should be served, or what to do when the server is stopped. When using gen_server, programmers could use the functions implementing these concrete behaviours as input functions for SecEr. In this way, they can check, for instance, that a server is going to reply to the user and leave the server's state in the same way across different versions of the program.

We can explain it with a real example. Consider the code in Algorithm 7, which shows a fragment (although those parts of the module that are not shown here are also interesting, we have removed them because they are not used in the use case) of the gen_server defined in

https://github.com/hcvst/erlang-otp-tutorial#otp-gen_server

The server's state is simply a counter that tracks the number of requests served so far. The server defines three types of requests through the functions handle_call and handle_cast:

(1) The synchronous request (i.e., a request where the client waits for a reply) get_count, which returns the current server's state.

(2) The asynchronous request (i.e., a request where the client does not wait for a reply) stop, which stops the server.

(3) The asynchronous request say_hello, which makes the server print hello in the standard output.

The first and the third requests modify the server's state by adding one to the total number of requests served so far. The second one does not modify the state but rather it returns a special term that makes the gen_server stop itself.

To illustrate how SecEr can detect an unexpected behaviour change between two versions of the code, consider that the current (buggy) version is the one depicted in Algorithm 7, while the (correct) original version of the code contains line (38) instead of line (39).

Then, we can define a configuration file like the one in Algorithm 8 and run SecEr to see whether the behaviour

etorrent_dht_net_old.erl
```
decode_msg(InMsg) ->
    io:format("0: ~p\n", [InMsg]),
    Msg = etorrent_bcoding_old:decode(InMsg),
    io:format("0: ~p\n", [Msg]),
    MsgID = get_value(<<"t">>, Msg),
    case get_value(<<"y">>, Msg) of
        <<"q">> ->
            MString = get_value(<<"q">>, Msg),
            Method = string_to_method(MString),
            Params = get_value(<<"a">>, Msg),
            {Method, MsgID, Params};
        <<"r">> ->
            Values = get_value(<<"r">>, Msg),
            {response, MsgID, Values};
        <<"e">> ->
            [ECode, EMsg] = get_value(<<"e">>, Msg),
            {error, MsgID, ECode, EMsg}
    end.
query_ping_0_test() ->
    Enc = "d1:ad2:id20:abcdefghij0123456789e1:
            q4:ping1:t2:aa1:y1:qe",
    {ping, ID, Params} = decode_msg(Enc),
    ?assertEqual(<<"aa">>, ID),
    ?assertEqual(
    <<"abcdefghij0123456789">>,
    fetch_id(Params)).
```
etorrent_dht_net_new.erl
```
decode_msg(InMsg) ->
    io:format("0: ~p\n", [InMsg]),
    Msg = etorrent_bcoding_new:decode(InMsg),
    io:format("0: ~p\n", [Msg]),
    MsgID = get_value(<<"t">>, Msg),
    case get_value(<<"y">>, Msg) of
        <<"q">> ->
            MString = get_value(<<"q">>, Msg),
            Method = string_to_method(MString),
            Params = get_value(<<"a">>, Msg),
            {Method, MsgID, Params};
        <<"r">> ->
            Values = get_value(<<"r">>, Msg),
            {response, MsgID, Values};
        <<"e">> ->
            [ECode, EMsg] = get_value(<<"e">>, Msg),
            {error, MsgID, ECode, EMsg}
    end.
query_ping_0_test() ->
    Enc = "d1:ad2:id20:abcdefghij0123456789e1:
            q4:ping1:t2:aa1:y1:qe",
    {ping, ID, Params} = decode_msg(Enc),
    ?assertEqual(<<"aa">>, ID),
    ?assertEqual(
    <<"abcdefghij0123456789">>,
    fetch_id(Params)).
```
etorrent_bcoding_old.erl
```
-spec decode(string() | binary()) -> bcode().
decode(Bin) when is_binary(Bin) ->
  decode(binary_to_list(Bin));
decode(String) when is_list(String) ->
     {Res, _Extra} = decode_b(String),
     Res.

```

Algorithm 5: Continued.

```
etorrent_bcoding_new.erl
(8)   -spec decode(string()| binary()) ->
(9)           {ok, bcode()}|{error, _Reason}.
(10)  decode(Bin) when is_binary(Bin) ->
(11)        decode(binary_to_list(Bin));
(12)  decode(String) when is_list(String) ->
(13)        try
(14)          {Res, _Extra} = decode_b(String),
(15)          {ok, Res}
(16)    catch
(17)      error:Reason -> {error, Reason}
(18)    end.
```

Algorithm 5: etorrent source files (original and buggy versions).

```
$ ./secer -pois "test_etorrent:rel()" -funs "test_etorrent:funs()" -to 15
Function: query_ping_0_test/0
----------------------------
Generated test cases: 1
Mismatching test cases: 1 (100.0%)
      POIs comparison:
          + {{'etorrent_dht_net_old', 3, call, 1},
              {'etorrent_dht_net_new', 29, call, 1}}
                    Unexpected trace value => 1 Errors
                    Example call: query_ping_0_test()
------ Detected Error ------
Call: query_ping_0_test()
Error Type: Unexpected trace value
POI: ({'etorrent_dht_net_old', 3, call, 1}) trace:
          [[{<<97>>,[{<<105,100>>,<<97,98,99,100,101,102,103,104,105,106,48,49,50,51,52,53,54,
                   55,56,57>>}]},
            {<<113>>,<<112,105,110,103>>},{<<116>>,<<97,97>>},{<<121>>,<<113>>}]]
POI: ({'etorrent_dht_net_new', 29, call, 1}) trace:
          [{ok,[{<<97>>,[{<<105,100>>,<<97,98,99,100,101,102,103,104,105,106,48,49,50,51,52,53,
                   54,55,56,57>>}]},
                {<<113>>,<<112,105,110,103>>},{<<116>>,<<97,97>>},{<<121>>,<<113>>}]}]
----------------------------
```

Listing 14: SecEr reports discrepancies after defining multiple POIs.

is preserved or not. This configuration file uses two input functions (`handle_call` and `handle_cast`) and a POI relation that defines three POIs, one for each request output. If we run SecEr using this configuration we obtain the output shown at Listing 15. In the output we can see that no errors are reported for function `handle_call`, which means that the request `get_count` is served in the same way in both versions. In contrast, an error is reported in function `handle_cast`, pointing to the POI defined in line (19) of Listing 14. This means that for the request `say_hello` the behaviour has not been preserved, while for the request `stop` it has been preserved. In particular, the error found reveals that there is a discrepancy between the new server's states returned by each version of the program.

This simple example shows how SecEr can be used to check behaviour preservation even in concurrent context. The key is that there is no need to run an execution with real concurrency; instead we can study directly the relevant functions that are used during the concurrent execution, like `handle_call` or `handle_cast` in the example above.

8. Alternative Approaches to SecEr

There exist several techniques that are currently being applied in professional Erlang projects to avoid regression faults. SecEr has been designed as both an alternative and a complement to these techniques.

In this section, we compare SecEr with the already available debugging and testing techniques that could be used when behaviour preservation is checked in an Erlang project. To illustrate these techniques, we use a real improvement

```
-module(test_etorrent).
-compile(export_all).
poio1() ->
    {'etorrent_dht_net_old', 3, call, 1}.
poio2() ->
    {'etorrent_dht_net_old', 5, call, 1}.
poio3() ->
    {'etorrent_dht_net_old', 6, 'case', 1}.

poin1() ->
    {'etorrent_dht_net_new', 29, call, 1}.
poin2() ->
    {'etorrent_dht_net_new', 31, call, 1}.
poin3() ->
    {'etorrent_dht_net_new', 32, 'case', 1}.

rel() ->
    [{poio1(), poin1()},
     {poio2(), poin2()},
     {poio3(), poin3()}].

funs() ->
    "[query_ping_0_test/0]".
```

Algorithm 6: Configuration file to test etorrent modules.

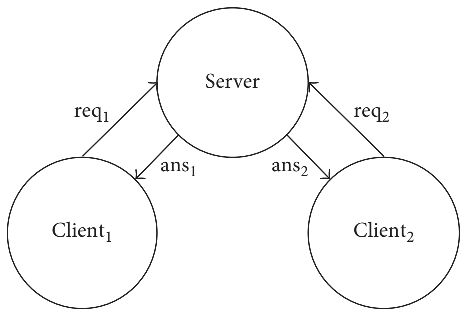


Figure 4: A simple client-server model.

of performance done in the `orddict:from_list/1` function from the standard library of Erlang-OTP. The commit description can be found at

> https://github.com/erlang/otp/commit/5a7b2115ca5b9c23aacf79b634133fea172a61fd

This commit did not introduce any regression faults so, in order to make this study more interesting, we have also included a fault (see lines (8-9) in Listing 17). Listing 16 shows the code changed in the commit (function `from_list/1`) and the code involved in the change (function `store/3`). Listing 17 shows the new version of function `from_list/1` and function `reverse_pairs/2`, used to reverse a list. The new version also uses function `lists:ukeysort/2` (this function is described within a comment in Listing 17) to sort the given list. The original version uses function `store/3` that, whenever an already stored key is stored again, replaces its current value by the new one. On the other hand, function `lists:ukeysort/2` does exactly the contrary (see lines (13-14) in Listing 17). This is the reason why the list needs to be previously reversed. Therefore, the fault introduced assumes that programmers forgot to reverse the list. This is what line (9) in Listing 17 stages. The correct version is commented on above in line (8). In the following, all the techniques are applied to the described scenario.

8.1. Unit Testing. This is the most common way of checking behaviour preservation. In Erlang, it is common to define unit test cases and execute them with `EUnit` [16] (Erlang's unit testing tool). The test file of Algorithm 9 includes unit test cases specific for our scenario. Those tests using function `from_list_test_common/1` check whether the intended behaviour has been implemented by using three simple cases. On the other hand, the tests using function `from_list_vs/2` check whether the behaviour is preserved across different versions for the same three cases. The output of `EUnit` with these test cases is shown in Listing 18.

```
-module(hello_server).

-behavior(gen_server).

-record(state, {count}).

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% gen_server Function Exports
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-export([          % The behavior callbacks
   init/1,         % - initializes our process
   handle_call/3, % - handles synchronous calls
   handle_cast/2, % - handles asynchronous calls
   terminate/2]). % - is called on shut-down

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% gen_server Function Definitions
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

init([]) ->
     {ok, #state{count=0}}.

 -spec handle_call(get_count, any(),{state, integer()}) ->
        {reply, integer(),{state, integer()}}.
handle_call(get_count, _From, #state{count=Count}) ->
     {reply, Count, #state{count=Count+1}}.

 -spec handle_cast(stop | say_hello, {state, integer()}) ->
        {stop, any(),{state, integer()}}
        |{noreply, {state, integer()}}.
handle_cast(stop, State) ->
     {stop, normal, State};

handle_cast(say_hello, State) ->
     io:format("Hello~n"),
     {noreply,
      % #state{count = State#state.count+1}    % RIGHT
      #state{count = State#state.count-1}      % WRONG
      }.

 terminate(_Reason, _State) ->
     error_logger:info_msg("terminating~n"),
     ok.
```

Algorithm 7: `hello_server.erl`.

`EUnit` reports 3 failing tests: two were expected, since they are pointing to the tests that include the wrong version of the code (functions `from_list_new_wrong_test/0` and `from_list_old_test_vs_new_wrong_test/0`). However, there is a third one that is a false positive. False positives happen because EUnit cannot find discrepancies when comparing erroneous computations. Therefore, an input that produces an error in the first version is reported as a failing test without checking whether it also fails in the second version.

All in all, unit testing allows us to identify a failing test case, which is a starting point to find the source of the discrepancy. The main problem is that unit testing requires writing a robust set of tests. Note that, without the second test case, i.e., `[{0, 1}, {0, 2}, {2, 3}]`, no test would fail (except for the false positive).

8.2. Property Testing. This approach is similar to unit testing, but it allows us to define more test cases in an easy way. It was first defined for Haskell and named QuickCheck [17]. Erlang has two implementations of this approach: QuviQ's `Erlang QuickCheck` [18] and `PropEr` [11]. Both are almost equivalent, with the exception of some small particularities (https://github.com/manopapad/proper#incompatibilities-with-quviqs-quickcheck). The big difference is that the `Erlang QuickCheck` is a commercial tool developed by QuviQ AB,

```
$ ./secer -pois "test_hello_server:rel()" -funs "test_hello_server:funs()" -to 15
Function: handle_call/3
----------------------------
Generated test cases: 19083
Both versions of the program generate identical traces for the defined points of interest
----------------------------
Function: handle_cast/2
----------------------------
Generated test cases: 42
Mismatching test cases: 21 (50.0%)
    POIs comparison:
        + {{'examples/gen_server/hello_server.erl',37,tuple,1},
          {'examples/gen_server/hello_server_wrong.erl',37,tuple,1}}
                Unexpected trace value => 21 Errors
                Example call: handle_cast(say_hello,{state,4})
------ Detected Error ------
Call: handle_cast(say_hello,{state,4})
Error Type: Unexpected trace value
POI: ({'examples/gen_server/hello_server.erl',37,tuple,1}) trace:
          [{noreply,{state,5}}]
POI: ({'examples/gen_server/hello_server_wrong.erl',37,tuple,1}) trace:
          [{noreply,{state,3}}]
----------------------------
```

Listing 15: SecEr reports discrepancies in the functions implementing the requests.

```
-module(test_hello_server).
-export([rel/0, funs/0]).

file(0) ->
  'examples/gen_server/hello_server.erl';
file(1) ->
  'examples/gen_server/hello_server_wrong.erl'.

poi_rel(POI) ->
  {POI(0), POI(1)}.

poi1(Version) ->
  {file(Version), 27, tuple, 1}.

poi2(Version) ->
  {file(Version), 33, tuple, 1}.

poi3(Version) ->
  {file(Version), 37, tuple, 1}.

rel() ->
  [poi_rel(fun poi1/1), poi_rel(fun poi2/1),
    poi_rel(fun poi3/1)].

funs() ->
  "[handle_call/3, handle_cast/2]".
```

Algorithm 8: test_hello_server.erl.

```
-spec from_list(List) -> Orddict when
      List:: [{Key:: term(), Value:: term()}],
      Orddict:: orddict().

from_list(Pairs) ->
     lists:foldl(
       fun ({K,V}, D) -> store(K, V, D) end, [], Pairs).

-spec store(Key, Value, Orddict1) -> Orddict2 when
       Key:: term(),
       Value:: term(),
       Orddict1:: orddict(),
       Orddict2:: orddict().

store(Key, New, [{K,_}=E|Dict]) when Key < K ->
     [{Key,New},E|Dict];
store(Key, New, [{K,_}=E|Dict]) when Key > K ->
     [E|store(Key, New, Dict)];
store(Key, New, [{_K,_Old}|Dict]) -> % Key == K
     [{Key,New}|Dict];
store(Key, New, []) -> [{Key,New}].
```

Listing 16: orddict_old.erl.

```
-spec from_list(List) -> Orddict when
      List:: [{Key:: term(), Value:: term()}],
      Orddict:: orddict().

from_list([]) -> [];
from_list([{_,_}]=Pair) -> Pair;
from_list(Pairs) ->
      lists:ukeysort(1, reverse_pairs(Pairs, [])) % RIGHT
      lists:ukeysort(1, Pairs).                   % WRONG

% ukeysort(N, TupleList1) -> TupleList2
%      Returns a list containing the sorted elements of
%      list TupleList1 where all except the first tuple of
%      the tuples comparing equal have been deleted.
%      Sorting is performed on the Nth element of the tuple

reverse_pairs([{_,_}=H|T], Acc) ->
      reverse_pairs(T, [H|Acc]);
reverse_pairs([], Acc) -> Acc.
```

Listing 17: orddict_new.erl.

while `PropEr` is an open-source project available at GitHub. The authors of the commit defined their property tests for `Erlang QuickCheck`. Therefore, we have adapted them to `PropEr` (1st property: https://github.com/mistupv/secer/blob/master/examples/orddict/orddict_t1.erl; 2nd property: https://github.com/mistupv/secer/blob/master/examples/orddict/orddict_t2.erl) (with really few modifications) to make it available for any interested researcher that wants to reproduce the outputs shown below.

In the commit, the authors explain what properties they check and how they check them (the source code of the properties can be found in the commit):

The first QuickCheck test first generates a list of pairs of terms, then uses the list to create both an original and revised `orddict` *using* `from_list/1`, *then verifies that the results of the operation are the same for both instances. The*

```
> eunit:test(orddict_tests).
orddict_tests: from_list_new_wrong_test...*failed*
in function orddict_tests:'-from_list_test_common/1-fun-1-'/1 (orddict_tests.erl, line (21))
in call from orddict_tests:from_list_test_common/1 (orddict_tests.erl, line (19))
**error:{assertEqual,[{module,orddict_tests},
                 {line,(21)},
                 {expression,"Mod: from_list ( [{ 0 , 1 }, { 0 , 2 }, { 2 , 3 } ] )"},
                 {expected,[{0,2},{2,3}]},
                 {value,[{0,1},{2,3}]}]}
    output:<<"">>
orddict_tests: from_list_old_test_vs_new_wrong_test...*failed*
in function orddict_tests:'-from_list_vs/2-fun-1-'/2 (orddict_tests.erl, line (44))
in call from orddict_tests:from_list_vs/2 (orddict_tests.erl, line (42))
**error:{assertEqual,[{module,orddict_tests},
                 {line,(44)},
                 {expression,"Mod2: from_list ( Case2 )"},
                 {expected,[{0,2},{2,3}]},
                 {value,[{0,1},{2,3}]}]}
    output:<<"">>
orddict_tests: from_list_old_test_vs_new_ok_test...*failed*
in function orddict_old:'-from_list/1-fun-0-'/2 (orddict_old.erl, line 60)
    called as '-from_list/1-fun-0-'(1,[])
in call from lists:foldl/3 (lists.erl, line 1263)
in call from orddict_tests:'-from_list_vs/2-fun-2-'/2 (orddict_tests.erl, line (46))
**error:function_clause
    output:<<"">>
=======================================================
    Failed: 3. Skipped: 0. Passed: 2.
error
```

Listing 18: EUnit's output.

second QuickCheck test is similar except that it first creates an instance of the original and revised orddicts and then folds over a randomly-generated list of orddict *functions, applying each function to each* orddict *instance and verifying that the results match.*

The output of PropEr is depicted in Listings 19 and 20 (because the inputs are randomly generated, the results may vary across different runs). The first property fails with input [{1,false},{1,true}]. This is one of the cases where the buggy version behaves differently, so the error reported actually identifies a discrepancy. Nevertheless, the error found by PropEr with the second property is a mismatch in the comparison of the resulting dictionaries, without even executing the list of orddict functions; thus this error is synonymous of the first one.

As it happens with unit testing, this approach is handy to find a failing test case to begin the debugging process that finds the source of the discrepancy. However, the definition of properties is difficult and can miss some corner cases. In general, property testing is more powerful than unit testing because each property can be used to generate an arbitrary number of tests, but the definition of properties often involves more time.

8.3. CutEr. Even though we use CutEr [9] in the internals of our tool to generate inputs, it was conceived as a standalone tool. The main difference with the previous approaches is that CutEr is a white-box approach. It does not randomly generate the inputs, but it analyzes the source code to generate inputs that explore different execution branches.

In our scenario, one can use CutEr to generate test cases for the current version and/or for the previous version. Unfortunately, by doing this, the relationship between the versions is not considered during the generation; i.e., the introduced changes are not considered in the test generation. In Listing 21, we can see that CutEr was able to generate for the previous version some list where some of the elements have a common key, i.e., tests that could reveal the error. The time used to compute all the tests was 2 minutes 43 seconds. For the current version, Listing 22 shows that CutEr only generated 5 input tests. The time used to generate them was 7.4 seconds. Only one generated test has tuples with a repeated key: from_list([0.0,2.0,0,1.0]). Note that this case is useful in our scenario, but it could be useless in other situations. For instance, if pattern matching was used to compare values, 0 and 0.0 would not be considered as matching values.

This example shows that CutEr can be very helpful to generate a lot of test cases that cover most paths in the code.

```
-module(orddict_tests).
-compile(export_all).

-include_lib("eunit/include/eunit.hrl").

from_list_old_test() ->
  from_list_test_common(orddict_old).

from_list_new_ok_test() ->
  from_list_test_common(orddict_new_ok).

from_list_new_wrong_test() ->
  from_list_test_common(orddict_new_wrong).

from_list_test_common(Mod) ->
  ?assertEqual(
     [{0,1}, {1, 2}, {2, 3}],
      Mod:from_list([{0,1}, {1, 2}, {2, 3}])),
  ?assertEqual(
     [{0,2}, {2, 3}],
      Mod:from_list([{0,1}, {0, 2}, {2, 3}])),
  ?assertError(
     function_clause,
     Mod:from_list([1, {1, 2}, {2, 3}])).

from_list_old_test_vs_new_wrong_test() ->
  from_list_vs(orddict_old, orddict_new_wrong).

from_list_old_test_vs_new_ok_test() ->
  from_list_vs(orddict_old, orddict_new_ok).

from_list_vs(Mod1, Mod2) ->
  Case1 =
      [{0,1}, {1, 2}, {2, 3}],
  Case2 =
      [{0,1}, {0, 2}, {2, 3}],
  Case3 =
     [1,{1, 2}, {2, 3}],
  ?assertEqual(
     Mod1:from_list(Case1),
     Mod2:from_list(Case1)),
  ?assertEqual(
     Mod1:from_list(Case2),
     Mod2:from_list(Case2)),
  ?assertEqual(
     Mod1:from_list(Case3),
     Mod2:from_list(Case3)).
```

Algorithm 9: EUnit tests.

However, what test cases can reveal a different behaviour remains unknown. Therefore, we are forced to run the test cases on the other version to check that the results are the same. When a discrepancy is found, a debugging process should be started. Moreover, as we can see in the CutEr's execution with the old version, the time needed to compute all the test cases is significantly bigger due to the white-box analysis.

8.4. Print Debugging

8.4. Print Debugging. Print debugging is still a very extended practice because it allows us to quickly check the values of any variable at some specific point. Essentially, we must modify the code to catch the value of some selected expressions. Two drawbacks are that these changes can introduce new errors, and they should be undone when debugging has finished.

In our scenario, we can use one of the test cases reported by PropEr to start the debugging process, e.g., `orddict:from_list([{1,false},{1,true}])`. Before executing this test case in the two versions, we must add some prints to show some intermediate values. In particular, we replaced lines (6) and (7) of Listing 16 with the code shown in Listing 23. With this addition we are able to observe

```
> orddict_t1:test().
............................!
Failed: After 29 test(s).
An exception was raised: error:{badmatch,false}.
Stacktrace: [{orddict_t1,'-prop_equivalent_dict_modules/0-fun-0-',1,
                           [{file,"orddict_t1.erl"},{line,25}]}].
[{1,1},{1,-2}]
Shrinking...(3 time(s))
[{1,false},{1,true}]
false
```

Listing 19: PropEr's output for the first property.

```
> orddict_t2:test().
..........................................................................
.................!
Failed: After 100 test(s).
An exception was raised: error:{badmatch,false}.
Stacktrace: [{orddict_t2,'-prop_equivalent_dict_modules/0-fun-0-',2,
                           [{file,"orddict_t2.erl"},{line,39}]},
               {lists,foldl,3,[{file,"lists.erl"},{line,1263}]},
               {orddict_t2,'-prop_equivalent_dict_modules/0-fun-1-',1,
                           [{file,"orddict_t2.erl"},{line,37}]}].
{[{2.50411088062667,-7.851645940978741},{1,<<239,224,172,126>>},
  {-0.6360219784877551,4},{1,<<177,118,23,95,55>>}],[store,fetch,fetch,find]}
Shrinking........(8 time(s))
{[{1,false},{1,0}],[is_key]}
false
```

Listing 20: PropEr's output for the second property.

the input accumulator and the output accumulator in each iteration. For the code in Listing 17, the changes should be made inside function `lists:ukeysort/2` because it is the function in charge of calculating the final output of `orddict:from_list/1`. The idea of the change in this case is similar, so the code is modified as it is shown in Listing 24 (all complete files are available at https://github.com/mistupv/secer/tree/master/examples/orddict).

The output produced by the print statements for both the old and new versions is depicted in Listings 25 and 26, respectively. It can help to understand that the old version keeps the new value of a key that is already at the dictionary, while function `lists:ukeysort/2` does the contrary. If this is not evident for programmers, then they need to add more print statements in the auxiliary functions of `lists:ukeysort/2` and in function `orddict:store/3`.

This example shows that, even with a small change like the one considered here, the number of functions involved can be quite big and the use of prints to the standard output can become an impracticable approach. Moreover, this approach can introduce new errors due to the additions/deletions in the (already buggy) code.

8.5. The Erlang Debugger. Known as `Debugger` [19], it is a GUI for the Erlang interpreter that can be used for the debugging of Erlang programs. It includes common debugging features such as breakpoints, single-stepped execution, and the ability to show/modify variable values.

In our scenario, we can use `Debugger` to place some breakpoints in the code and observe, step by step, how the dictionaries are created in each version. The first intuition is to place a breakpoint in line (7) of Listing 16 to observe the evolution of the accumulator (as we did in the previous technique). However, because breakpoints refer to a (whole) line, we are already facing one of the problems of this approach: we cannot place the breakpoint in the desired spots (the input accumulator and the output accumulator). Fortunately, one can easily change the code so each expression of interest is placed in one different line allowing the use of breakpoints as desired, i.e., in the header of the anonymous function and in the line that contains the call to the `orddict:store/3` function. When we run the test we realize that the breakpoints are ignored and that the test is run without stopping. Therefore, we need to add breakpoints for each line of the `orddict:store/3` function, which has various clauses. Note that all the breakpoint definitions are done through the graphical interface so this process is quite slow. After adding these new breakpoints, the execution does stop at some of these new breakpoints and we are able to inspect the intermediate results. We should do something

```
$ ./cuter orddict_oldfrom_list '[[{0,1}]]' -r -v
Compiling orddict_old.erl... OK
Testing orddict_old:from_list/1...
orddict_old:from_list([{0,1}])... ok
xxx
orddict_old:from_list([])... ok
...
orddict_old:from_list([{0,0.0},{0,1.0}])... ok
...
orddict_old:from_list([{[],0.0},{[],1.0}])... ok
...
orddict_old:from_list([{0,0.0},{0,1.0},{4,2.0}])... ok
...
Solver Statistics...
    - Solved models: 84
    - Unsolved models: 432
```

Listing 21: CutEr's output for the old version (trimmed).

```
$ ./cuter orddict_new_wrong from_list '[[{0,1}]]' -r -v
Compiling orddict_new_wrong.erl... OK
Testing orddict_new_wrong:from_list/1...
orddict_new_wrong:from_list([{0,1}])... ok
orddict_new_wrong:from_list([])... ok
xx
orddict_new_wrong:from_list([{0.0,1.0},{3.0,2.0}]) ... ok
xxxxxxxxxxx
orddict_new_wrong:from_list([{0,0.0},{1.0,2.0}])... ok
orddict_new_wrong:from_list([{0.0,2.0},{0,1.0}])... ok
orddict_new_wrong:from_list([{0,0.0},{1,1.0}])... ok
xx
Solver Statistics...
    - Solved models: 5
    - Unsolved models: 15
```

Listing 22: CutEr's output for the current version.

```
(1) lists:foldl(
(2)   fun ({K,V}, D) ->
(3)     io:format("Input: ~p\n", [D]),
(4)     Res = store(K, V, D),
(5)     io:format("Output: ~p\n", [Res]),
(6)     Res
(7)   end,
(8)    [],
(9)   Pairs).
```

Listing 23: Fragment of orddict_old.erl with io:format/2.

```
(1) ukeysort(I, L) when is_integer(I), I > 0 ->
(2)   io:format("Input: ~p\n", [L]),
(3)   Res =
(4)     case L of
(5)     ...
(6)     end,
(7)   io:format("Output: ~p\n", [Res]),
(8)   Res.
```

Listing 24: Fragment of lists:ukeysort/2 with io:format/2.

similar with the code of Listing 17. In this case, it makes more sense to add breakpoints inside the `lists:ukeysort/2` function. Figure 5 shows an instant of the Debugger's session where a user can realize that the second element with repeated key is ignored, instead of the first one.

Although `Debugger` is very helpful in general, the insertion of breakpoints through the GUI can become tedious. Moreover, as shown in the example, a single line in Erlang can include several interesting spots where one would place a breakpoint. Unfortunately, `Debugger` does not include a

```
> orddict_old:from_list([{1,false},{1,true}]).
Input: []
Output: [{1,false}]
Input: [{1,false}]
Output: [{1,true}]
[{1,true}]
```

Listing 25: Output of `orddict_old.erl` with `io:format/2`.

```
> orddict_new_wrong:from_list([{1,false},{1,true}]).
Input: [{1,false},{1,true}]
Output: [{1,false}]
[{1,false}]
```

Listing 26: Output of `orddict_new.erl` with `io:format/2`.

way to define breakpoints in a fine-grained way, thus forcing users to modify their code so they can define the breakpoints as desired. The good point of this approach is that when all configurations are set up, it becomes a very illustrative way to see what the code is doing in each place in order to find the source of an unexpected behaviour.

8.6. Erlang's Declarative Debugger (EDD). Algorithmic debugging is a technique that allows debugging a program through an interview with the programmer where questions refer to the intended behaviour of the computations. In functional languages, the questions are about the validity of a function call and its result. Erlang has an implementation of this approach named EDD [20].

Listing 27 shows an EDD session to debug our buggy program. In this session, we use the same input test case as in the previous debugging approaches, i.e., `from_list([{1, false},{1,true}])`. EDD starts asking about the call `lists:ukeysort(1, [{1, false}, {1, true}])`. The computed value is correct, so the answer given by a user should be yes (y). With only this question EDD finishes (because there are no more calls inside this function) and blames the third clause of `orddict:from_list/1`.

Although, in general, declarative debugging is very helpful to debug buggy programs, for this example, it does not help too much. EDD points to the source of the discrepancy, so the user can find that there is something wrong in one of the arguments of `lists:ukeysort/2`. Nevertheless, with the information provided, it is still not easy to interpret what the error is or how to solve it.

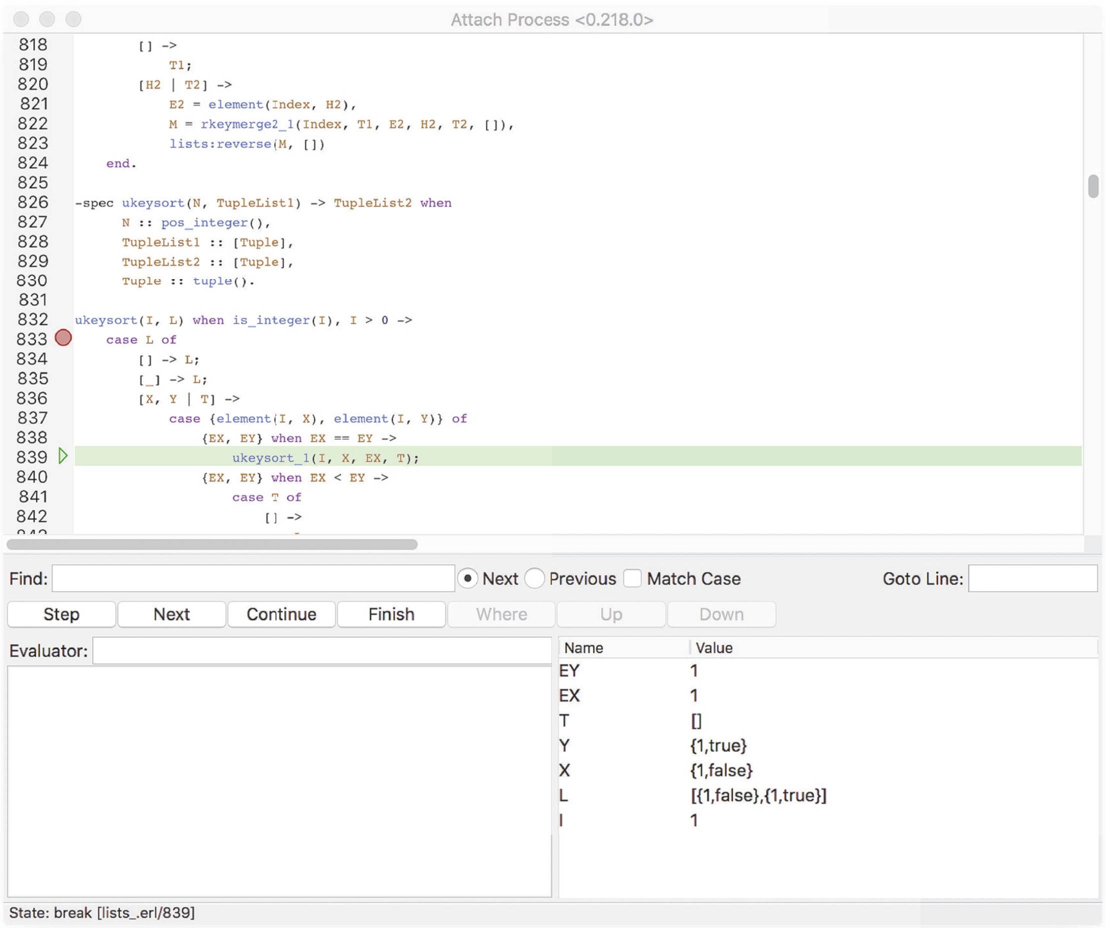
8.7. Dialyzer. Erlang is a dynamically typed language. Therefore, type checking is not performed at compilation time and this can result in several undetected errors that eventually arise at execution time. While many programmers like this feature, others miss a type system. Erlang has a tool named `Dialyzer` [21] that partially solves this problem through the use of static analysis. `Dialyzer`'s input is an Erlang module, and it reports any type discrepancies found in its function definitions. The use of type contracts (`spec` in Erlang) helps `Dialyzer` to improve their results.

In our scenario, `Dialyzer` is not really helpful. It does not report any type discrepancy (because they do not exist) in the buggy code. However, for other scenarios, it could discover regression faults when incorrect values (with an unexpected type) are involved.

8.8. Checking the Performance Improvement. The authors of the commit provide a link to a system implemented by themselves that they used to check whether the performance of the `orddict:from_list/1` function has been actually improved. When this program is run the results obtained are similar to the ones included in the commit message. The fact that they implemented ad hoc that program to check the performance improvement demonstrates (recall that this is a commit on the OTP-Erlang package, i.e., the official Erlang release) that, unfortunately, there is not any tool available to check nonfunctional features like the execution time.

8.9. SecEr. Finally, we show how `SecEr` makes a step forward, being an alternative (or a complement) to the previous approaches to identify discrepancies.

In order to check the performance of both versions, the first step is to build a configuration file for `SecEr` (it is shown in Algorithm 10). This configuration file specifies that the results computed in function `orddict:from_list/1` are the same in both versions. Lines (10), (11), and (12) define POIs to compare the unique clause in the original version with each of the three new clauses in the current version (in future versions, we plan to add special POIs that refer to all outputs of a function, without the need for defining the output of each clause. The POI relation `rel1/0` could be redefined to something like `[{{file(o), 60, function, 1}, {file(n), 59, function, 1}}]`). The output of `SecEr` with this configuration is shown in Listing 28. `SecEr` does automatically all the work that we had



```
        [] ->
            T1;
        [H2 | T2] ->
            E2 = element(Index, H2),
            M = rkeymerge2_1(Index, T1, E2, H2, T2, []),
            lists:reverse(M, [])
    end.

-spec ukeysort(N, TupleList1) -> TupleList2 when
      N :: pos_integer(),
      TupleList1 :: [Tuple],
      TupleList2 :: [Tuple],
      Tuple :: tuple().

ukeysort(I, L) when is_integer(I), I > 0 ->
    case L of
        [] -> L;
        [_] -> L;
        [X, Y | T] ->
            case {element(I, X), element(I, Y)} of
                {EX, EY} when EX == EY ->
                    ukeysort_1(I, X, EX, T);
                {EX, EY} when EX < EY ->
                    case T of
                        [] ->
```

Name	Value
EY	1
EX	1
T	[]
Y	{1,true}
X	{1,false}
L	[{1,false},{1,true}]
I	1

State: break [lists_.erl/839]

Figure 5: Debugger's session.

```
> edd:dd( "orddict_new_wrong:from_list([{1,false},{1,true}])").
lists:ukeysort(1, [{1, false}, {1, true}]) = [{1, false}]? [y/n/t/v/d/i/s/u/a]: y
Call to a function that contains an error:
orddict_new_wrong:from_list([{1, false}, {1, true}]) = [{1, false}]
Please, revise the third clause:
from_list(Pairs) -> lists:ukeysort(1, Pairs).
```

Listing 27: EDD's session for the current version.

to do manually in the previous approaches: (i) test cases generation, (ii) checking whether test cases evaluate the POIs, (iii) comparison of traces, taking into account the fact that the same error in both versions is not a discrepancy, and (iv) producing a report with all discrepancies. As a result, SecEr has identified the discrepancies and it has shown a sample call that produces the discrepancy.

At this point, we can use SecEr again to inspect the intermediate results produced during the computation reported as responsible for the discrepancy. Since both versions compute the dictionary in a very different way, instead of using SecEr to compare the traces, it is better to use it to store the values generated at certain points and print all of them afterwards so that we can manually check them. For this, we can (i) define POIs to observe the key value introduced in each iteration and the resulting dictionary, (ii) define an input function that simply calls one of the failing cases that were reported in Listing 28, and (iii) use the predefined comparison function `secer:show/0` that simply prints the values of the POIs traces. Algorithm 11 shows a configuration file that implements these ideas. The output of SecEr with this configuration prints the values generated in the selected POIs for each version of the program (it is shown in Listing 29).

SecEr can also be used to check whether a performance improvement has been actually achieved. In the current

```
$ ./secer -pois "test_orddict:rel1()" -funs "test_orddict:funs()" -to 15
Function: from_list/1
----------------------------
Generated test cases: 16428
Mismatching test cases: 555 (3.37%)
    POIs comparison:
         + {{'orddict/orddict_old.erl',60,call,1},
           {'orddict/orddict_new_wrong.erl',62,call,1}}
                Unexpected trace value => 555 Errors
                Example call: from_list([{{},-9},{{},3.1981469696010247}])
------ Detected Error ------
Call: from_list([{{},-9},{{},3.1981469696010247}])
Error Type: Unexpected trace value
POI: ({'orddict/orddict_old.erl',60,call,1}) trace:
           [[{{},3.1981469696010247}]]
POI: ({'orddict/orddict_new_wrong.erl',62,call,1}) trace:
           [[{{},-9}]]
----------------------------
```

Listing 28: SecEr's output when comparing the two versions.

```
-module(test_orddict).
-compile(export_all).

file(o) ->
  'orddict/orddict_old.erl';
file(n) ->
  'orddict/orddict_new_wrong.erl';

rel1() ->
  [{{file(o), 60, call, 1}, {file(n), 59, list, 2}},
    {{file(o), 60, call, 1}, {file(n), 60, {var, 'Pair'}, 2}},
    {{file(o), 60, call, 1}, {file(n), 62, call, 1}}].

funs() ->
  "[from_list/1]".
```

Algorithm 10: SecEr's configuration file to identify discrepancies.

version of SecEr, this involves small changes in the code, but we are working to automate these changes. The changes needed are very simple and applied in the last expression of each clause involved. For instance,

```
Start = os:timestamp(),
Res = lists:ukeysort(1, reverse_pairs(Pairs, [])),
{Pairs, timer:now_diff(os:timestamp(), Start)},
Res.
```

With these modifications we can now trace the input and the time used to compute each result. In order to identify a significative difference in the calculated runtimes, we need to produce big lists for the input. However, the lists that SecEr generates are in most cases too small for this purpose. Therefore, in the configuration file we can include the following function to produce bigger lists:

```
from_list_replicate(Pairs) ->
  from_list(replicate(5000, Pairs, [])).

replicate(1, List, Acc) ->
  Acc ++ List;
replicate(N, List, Acc) ->
  NList = lists:map(fun inc/1, List),
  replicate(N - 1, NList, Acc ++ List).
```

```
$ ./secer -pois "test_orddict:rel1()" -funs "test_orddict:funs()" -to 3 -cfun "secer:show()"
Trace old version:
POI: {orddict/orddict_old.erl',60,tuple,1}
Value: {{},-9}
POI: {'orddict/orddict_old.erl',60,application,2}
Value: [{{},-9}]
POI: {orddict/orddict_old.erl',60,tuple,1}
Value: {{},3.1981469696010247}
POI: {'orddict/orddict_old.erl',60,application,2}
Value: [{{},3.1981469696010247}]
Trace new version:
POI: {'orddict/lists.erl',836,{var,'X'},1}
Value: {{},-9}
POI: {'orddict/lists.erl',836,{var,'Y'},1}
Value: {{},3.1981469696010247}
POI: {'orddict/lists.erl',839,application,1}
Value: [{{},-9}]
Function: secer_failing_test/0
----------------------------
Generated test cases: 1
Both versions of the program generate identical traces for the defined points of interest
----------------------------
```

Listing 29: SecEr's output when tracing and comparing values in both versions.

```
-module(test_orddict).
-compile(export_all).

file(o) ->
  'orddict/orddict_old.erl';
file(l) ->
  'orddict/lists.erl'.

rel1() ->
  [{{file(o), 60, tuple, 1}, {file(l), 836, {var, 'X'}, 1}},
   {{file(o), 60, tuple, 1}, {file(l), 836, {var, 'Y'}, 1}},
   {{file(o), 60, call, 2}, {file(l), 839, call, 1}}].

funs() ->
  "[secer_failing_test/0]".
```

Algorithm 11: SecEr's configuration file to trace values.

```

inc({X, V}) ->
  {{1, X},V}.
```

This function creates 5000 copies of the input list changing their keys in each iteration. The new configuration file (shown in Algorithm 12) uses this function as input function, and it keeps the same POI relation as in Algorithm 10. Additionally, we use the predefined comparison function `secer:list_comp_perf/1` to report an error when the new version takes more time than the original version. We should discard those test cases where the difference is not significative, so we fix a threshold (defined by the `secer:list_comp_perf/1` function parameter) of, e.g., 2000 microseconds. The output generated by `SecEr` using this configuration file is shown in Listing 30. After having generated 503 lists of different sizes, in only one case the new version performs worse than the original version. This is probably an outlier, so maybe if we use this test input again, it will also run faster in the new version. However, reporting extra information, we can increase the confidence in the performance study. We can modify the comparison function used by `secer:io_list_comp_perf/1`, which works exactly like the previous one but also prints information about the computation that runs faster in the new version. With this change, the output indicates the exact improvement achieved by the new version. A sample of some lines produced for this example is as follows:

```
(1) ···
(2) Faster Calculation: Length: 400 -> 13365 vs 1524 ms.
(3) Faster Calculation: Length: 800 -> 49912 vs 2566 ms.
(4) Faster Calculation: Length: 1000 -> 26944 vs 3128 ms.
(5) ···
```

In all the printed cases, 398 in this example (the rest were discarded because of the defined threshold), there is a significative improvement in the computation time. These data increase the reliability of the study.

We can conclude that SecEr can be especially helpful in those contexts where no test suite is available; and even if we already have test cases, it can be used to generate new test cases that are specific to test POIs. It can also be helpful to print values for a concrete failing test or even for various (through the introduction of more input functions). Moreover, it can be also helpful in checking nonfunctional features like performance improvement. Finally, we want to highlight that SecEr has been used in the previous examples for three different purposes (discovering discrepancies, finding the source of a discrepancy, and checking the performance preservation), and in the three cases the methodology was exactly the same.

9. Experimental Evaluation

In this section we study the performance and the scalability of SecEr. In particular, we compare different configurations and study their impact on the performance. First, we collected examples from commits where some regression is fixed. Most of the considered programs were extracted from EDD [20] (https://github.com/tamarit/edd/tree/master/examples) because this repository contains programs with two code versions: one version of the code with a bug and a second version of the same code with the expected behaviour, i.e., where the bug is fixed. In order to obtain representative measures, the experiments were designed in such a way that each program was executed 21 times with a timeout of 15 seconds each. The first execution was discarded in all cases (because it loads libraries, caches data, etc.). The average computed for the other 20 executions produced one single data. We have repeated this process enabling and disabling the two most relevant features of SecEr: (i) the use of CutEr and (ii) the use of mutation during the ITC generation. The goal of this study is to evaluate how these features affect the accuracy and performance of the tool. To compare the configurations we computed three statistics for each experiment: the average amount of generated tests, the average amount of mismatching tests, and the average percentage of mismatching tests with respect to the generated ones.

Table 1 summarizes the experiments (all data and programs used in this experiment are available online at https://github.com/mistupv/secer/tree/master/benchmarks), where the best result for each program has been highlighted in bold. These results show that our mutation technique is able to produce better test cases than random test generation. Clearly, the configuration that does not use CutEr (the one in the middle) is almost always the best: it generates in all cases the highest amount of test cases, and it also generates more mismatching test cases (except for the *erlson2* program). The interpretation of these data is the following: CutEr invests much time to obtain the initial set of inputs, but the concolic test cases it produces do not improve enough the quality of the suite. This means that in general it is better to invest that time in generating random test cases, which on average produce more mismatching test cases. There are two exceptions: *erlson2* and *vigenere*. In *erlson2* the error is related to a very particular type of input (less than 0.02% of the generated tests report this error), and CutEr directs the test generation to mismatching tests in a more effective way. With respect to the second program *(vigenere)*, although the configuration that does not run CutEr generates more mismatching tests than the rest, the tests generated by CutEr allow the tool to reach a mismatching result faster. This is the reason for the slight improvement in the mismatching ratio. The common factor in both programs is that the mismatching ratio is rather low. This is a clear indication that CutEr can be useful when some corner cases are involved in the regression.

We can conclude that the results obtained by the tool are strongly related to the location of the error and the type of error. If it is located in an infrequently executed code or it is a corner case, the most suitable configuration is the one running CutEr. In contrast, if the error is located in a usually executed code, we can increase the mismatching tests generation by disabling CutEr. Because we do not know beforehand what the error is and where it is, the most effective way of using the tool is the following: First, run SecEr without CutEr, trying to maximize the mismatching test cases. If no discrepancy is reported, then enable CutEr to increase the reliability of the generated test cases.

We have also evaluated the growth rate of the generated test cases and of the percentage of mismatching ITCs. For this experiment we selected the program *turing* because it produces a considerable amount of tests in the three configurations, and also because the mismatching ratio is similar in all of them and not too close to 100%. We ran the three configurations of SecEr with this program with a timeout ranging between 4 and 20 seconds with increments of 2 seconds.

```
$ ./secer -pois "test_orddict_perf:rel1()" -funs "test_orddict_perf:funs()" -to 60 -cfun "secer:lists_
comp_perf(2000)"
Function: from_list_replicate/1
----------------------------
Generated test cases: 503
Mismatching test cases: 1 (0.19%)
    POIs comparison:
        + {"User Defined","User Defined"}
                 Unexpected trace value => 1 Errors
                 Example call: from_list_replicate([{{[],4.686994537220225,{},-1.5219780046371083},
               ...])
------ Detected Error ------
Call: from_list_replicate([{{[],4.686994537220225,{},-1.5219780046371083},...])
Error Type: Slower Calculation: Length: 700 -> 1031 vs 22564μs.
----------------------------
```

Listing 30: SecEr's output when comparing performance.

```
-module(test_orddict_perf).
-compile(export_all).

file(o) ->
  'orddict/orddict_old_perf.erl';
file(n) ->
  'orddict/orddict_new_ok_perf.erl'.

rel1() ->
   [{{file(o), 62, tuple, 1}, {file(n), 62, tuple, 1}},
    {{file(o), 62, tuple, 1}, {file(n), 67, tuple, 1}},
    {{file(o), 62, tuple, 1}, {file(n), 72, tuple, 1}}].
funs() ->
  "[from_list_replicate/1]".
```

Algorithm 12: SecEr's configuration file to check the performance improvement.

Table 1: Experimental evaluation of three SecEr configurations with a timeout of 15 seconds.

	CUTER + MUTATION			NO CUTER			NO MUTATION		
	Generated	Mismatching	%	Generated	Mismatching	%	Generated	Mismatching	%
ackermann	13.9	12.9	93.274%	**21.8**	**21.8**	**100.0**%	12.85	11.65	91.27%
caesar	37765.94	1615.1	4.2714%	**103072.0**	**4534.95**	**4.3997**%	38830.55	1702.7	4.3865%
complex_number	69420.2	67236.55	96.8549%	**89670.2**	**86891.75**	**96.9015**%	67451.75	65349.95	96.8825%
erlson1	14780.05	1.55	0.0105%	**14966.2**	**2.65**	**0.0177**%	14872.5	1.9	0.0127%
erlson2	15494.5	**0.95**	0.0059%	**16758.59**	0.8	0.0047%	15553.8	**0.95**	**0.0061**%
mergesort	29718.35	25634.45	86.2585%	**34315.1**	**29622.9**	**86.3259**%	29994.3	25884.2	86.299%
rfib	28.05	28.05	**100.0**%	**29.0**	**29.0**	**100.0**%	28.4	28.4	**100.0**%
roman	513.79	101.95	19.8415%	**535.35**	**108.05**	**20.1801**%	512.2	101.7	19.8461%
sum_digits	426.3	422.3	99.0615%	**534.0**	**534.0**	**100.0**%	434.0	430.0	99.078%
ternary	85.9	28.05	29.4187%	**1005.4**	**323.25**	**32.2485**%	130.0	39.7	27.8311%
turing	41828.65	28268.95	67.5825%	**77247.45**	**52651.5**	**68.1595**%	41573.1	28150.95	67.7135%
vigenere	115.55	2.1	1.269%	**308.7**	**4.59**	1.1849%	114.9	1.95	**1.4849**%

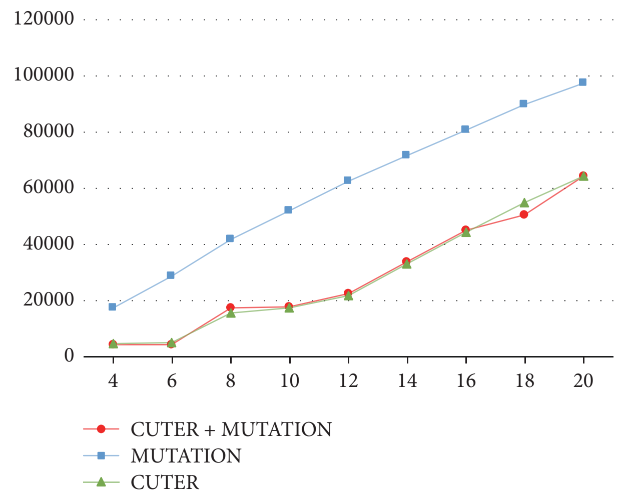


Figure 6: Number of tests generated for `Turing`.

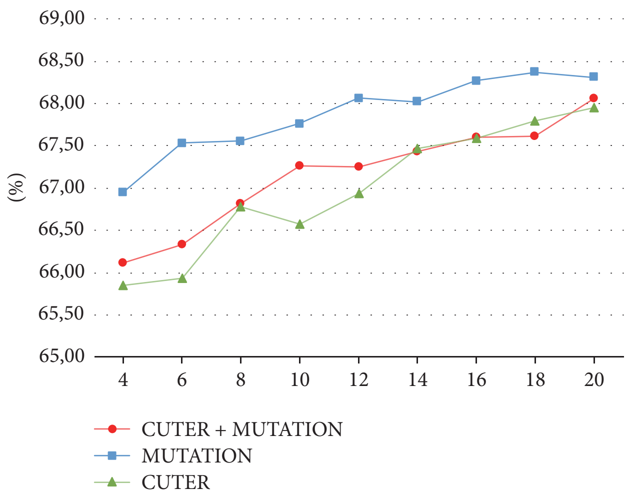


Figure 7: Mismatching ratio produced for `Turing`.

The results of the experiment are shown in Figures 6 and 7. In Figure 6, the X axis represents `SecEr` timeouts (in seconds), and the Y axis represents the number of test cases generated. In all cases, the configuration that does not run `CutEr` generates the highest number of tests. This configuration has a linear growth. On the other hand, the configurations using `CutEr` show a slow onset. They need at least twelve seconds to reach a considerable increase in the number of generated tests. There is not a significative difference between the configurations using `CutEr`. This means that the mutation technique does not slow down the test generation. In Figure 7, the X axis represents `SecEr` timeouts (in seconds), and the Y axis represents the percentage of mismatching tests over the total amount of tests generated. Clearly, in the three configurations, the quality of the generated test cases increases over time (i.e., the mismatching tests ratio increases over time). The configuration that does not run `CutEr` presents the highest percentage. In this case, the two approaches using `CutEr` produce different results. With smaller timeouts, it is preferable to enable mutation.

10. Related Work

The orchestrated survey of methodologies for automated software test case generation [22] identifies five techniques to automatically generate test cases. Our approach could be included in the class of *adaptive random technique as a variant of random testing*. Inside this class, the authors identify five approaches. Our mutation approach of the test input shares some similarities with various of these approaches like selection of best candidate as next test case or exclusion. According to a survey on test amplification [23], 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 under the "other approaches" subcategory.

Automated behavioural testing techniques like Soares et al. [6] and Mongiovi [5] are similar to our approach, but they are restricted in the kind of changes that can be analyzed (they only focus on refactoring). In contrast, our approach is independent of the kind (or the cause) of the changes, being able to analyze the effects of any change in the code regardless of its structure.

There are several works focused on the regression test case generation. DiffGen [24] instruments programs to add branches in order to find the differences in the behaviour between two versions of the program, and then it explores the branches in both versions, and finally it synthesizes test cases that show the detected differences. An improvement of this approach is implemented in the tool eXpress [25] where the irrelevant branches are pruned in order to improve the efficiency. Our technique, in contrast, is not directed by the computation paths, but it is directed by the POIs (i.e., what the user wants to observe). Another related approach is model-based testing for regression testing. This approach studies the changes made in a model, e.g., UML [26] or EFSM models based on dependence analysis [27], and test cases are generated from them. We do not require any input model neither infer it, so although some ideas are similar, the way to implement them is completely different. Finally, there are works that use symbolic execution focused on the regression test generation, like [28, 29]. All these works are directed to maximize the coverage of the generated test suites. Moreover, they need an existing regression test case suite to start with. There exist alternatives, but with the same foundations, like [30] where a tree-based approach is proposed to achieve high coverage. Our approach is not directed by coverage, but instead by the POIs, and we do not require any regression test as input.

Automated regression test case generation techniques like Korel and Al-Yami [31] are also very similar to our approach, but the user can only select output parameters of a function to check the behaviour preservation. Then, their approach simply runs that specific function and checks that the produced values are equal for both versions of the program. Therefore, their approach helps to discover errors in a concrete function, but they cannot generate inputs for one function and compare the outputs of another function. This limits, e.g., the observation of recursion. In contrast, we allow selecting any input function and place the POIs in any other function of the program. Additionally, their test input generation relies on white-box techniques that are not directed by the changes. Our approach, however, uses a black-box test input generation which is directed by changes.

Yu et al. [32] presented an approach that combines coverage analysis and delta debugging to locate the sources of the regression faults introduced during some software evolution. Their approach is based on the extraction and analysis of traces. Our approach is also based on traces although not only the goals but also the inputs of this process are slightly different. In particular, we do not require the existence of a test suite (it is automatically generated), while they look for the error sources using a previously defined test suite. Similarly, Zhang et al. [33] use mutation injection and classification to identify commits that introduce faults.

The Darwin approach [34] starts from the older version of the program, a program that is known to be buggy, and an input test case that reveals the bug. With all this information, it generates new inputs that fail on the buggy program and then runs them using dynamic symbolic execution and stores the produced trace (in their context, a trace contains the visited statements). Finally, the traces from the buggy version and from the old version are compared to locate the source of the discrepancy. Although the approach could seem similar to ours, the goals are different. We try to find discrepancies, while they start from an already-found discrepancy.

Our technique for mutation of inputs shares some similarities with RANDOOP [35]. In their approach, they start from test cases that do not reveal any failure, and randomly construct more complex test cases. The particularity of their approach is that the random test generation is feedback-directed, in the sense that each generated test case is analyzed to take the next decision in the generation. We do something similar, although our feedback is directed by the POIs selected by the user.

DSpot [36] is a test augmentation technique for Java projects that creates new tests by introducing modifications in the existing ones. The number of variants that will be generated is known beforehand and determined by parameters like the number of operations or the number of statements. In order to define the output of a test case, they introduce a concept called *observation point*, which is similar to our POIs. The difference is that they define and select their observation points (in particular, attribute getters, the `toString()` method, and the methods inside an assertion) while in our approach it is the user who defines them. Additionally, our approach does not need an already existent test suite.

Sieve [37] is a tool that automatically detects variations across different program versions. They run a particular test and store a trace that in their context is a list of memory operations over variables. The generated traces are later studied in order to determine what changed in the behaviour and why it changed. Although their goal is not the same as ours, their approach shares various similarities with ours, in particular code instrumentation and trace comparison.

Mirzaaghaei [38] presented a work called *Automatic Test Suite Evolution* where the idea is to repair an existing test suite according to common patterns followed by the practitioners when they repair a test suite. A repair pattern can be something like the introduction of an overloaded method. A modification of our technique could be used to achieve a similar goal, by not only producing test case input, but also repairing patterns in order to check whether they are effectively repairing an outdated test suite.

Most of the efforts in regression testing research have been put in the regression testing minimization, selection, and prioritization [1], although among practitioners it does not seem to be the most important issue [7]. In fact, in the particular case of the Erlang language, most of the works in the area are focused on this specific task [39–42]. We can find other works in Erlang that share similar goals but more focused on checking whether applying a refactoring rule will yield to a semantics-preserving new code [3, 4].

With respect to tracing, there are multiple approximations similar to ours. In Erlang's standard libraries, there are two implemented tracing modules. Both are able to trace the function calls and the process related events (spawn, send, receive, etc.). One of these modules is oriented to trace the processes of a single Erlang node [13], 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 [14] and the output trace of all the nodes can be formatted in many different ways. Cronqvist [12] presented a tool named redbug where a call stack trace is added to the function call tracing, making it possible to trace both the result and the call stack. Till [15] implemented erlyberly, a debugging tool with a Java GUI able to trace the previously defined features (calls, messages, etc.) but also giving the possibility of adding breakpoints and tracing 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 the program. In our approach, we can trace both the already defined features and also a point of the program regardless of its position.

11. Conclusions

During the lifecycle of any piece of software, different versions may appear, e.g., to correct bugs, to extend the functionality, or to improve the performance. It is of extreme importance to ensure that every new version preserves the correct behaviour of previous versions. Unfortunately, this task is often expensive and time-consuming, because it implies the definition of test cases that must account for the changes introduced in the new version.

In this work, we propose a new approach to automatically check whether the behaviour of a certain functionality is preserved among different versions of a program. The approach allows the user to specify a POI that indicates the specific parts of the code that are suspicious or susceptible of presenting discrepancies. Because the POI can be executed several times with a test case, we store the values that the POI takes during the execution. Thus, we can compare all actual evaluations of the POI for each test case.

The technique introduces a new tracing process that allows us to place the POI in patterns, guards, or expressions. For the test case generation, instead of reinventing the wheel, we orchestrate a sophisticated combination of existing tools like `CutEr`, `TypEr`, and `PropEr`. But we also improve the result produced by the combination of these tools introducing mutation techniques that allow us to find the most representative test cases. All the ideas presented have been implemented and made publicly available in a tool called `SecEr`.

There are some limitations in the current technique. First of all, it is not always easy to infer good types for the input functions. If an inferred type is too generic (like *any*()), our approach could start generating ITCs that are not useful. However, this is a problem specific of dynamically typed languages (like Erlang). In statically typed languages, this limitation does not exist. Value generation can be also a limitation if we need to generate complex values. This limitation is also common in other techniques such as property testing, where it is overcome by allowing users to define their own value generators. We plan to make our technique compatible with these user-defined value generators. Finally, we think that the ITC mutation could be more sophisticated, e.g., by incorporating some kind of importance ranking per parameter. The final goal of the improvements will be to generate better ITCs and to produce them faster.

There are several interesting evolutions of this work. One of them is to adapt the current approach to make it able to compare modules when some source of nondeterminism is present (e.g., concurrency). We could also increase the information stored in traces with, e.g., computation steps or any other relevant information, so that we could also check the preservation (or even the improvement) of nonfunctional properties such as efficiency. An alternative way of doing this is by defining a special POI that indicates that we are interested in a certain performance measure (time, memory usage, etc.) instead of the value. The tool could also provide comparison functions for such *performance* POIs. Another interesting extension is the implementation of a GUI, which would allow the user to select a POI by just clicking on the source code. We could also define quality attributes linked to each test case in order to ease the prioritization and selection for test cases. This feature would be very appreciated by the programmers as [7] reports. Finally, the integration of our tool with control version systems like Git or Subversion would be very beneficial to easily compare code among several versions.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

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

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[2] J. S. Rajal and S. Sharma, "A Review on Various Techniques for Regression Testing and Test Case Prioritization," *International Journal of Computer Applications*, vol. 116, no. 16, pp. 8–13, 2015.

[3] E. Jumpertz, *Using QuickCheck and semantic analysis to verify correctness of Erlang refactoring transformations*, Radboud University Nijmegen, 2010.

[4] H. Li and S. Thompson, "Testing erlang refactorings with QuickCheck," in *Symposium on Implementation and Application of Functional Languages*, pp. 19–36, Springer, 2007.

[5] M. Mongiovi, "Safira: A tool for evaluating behavior preservation," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 213-214, 2011.

[6] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, 2013.

[7] E. Engström and P. Runeson, "A Qualitative Survey of Regression Testing Practices," in *Product-Focused Software Process Improvement, 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010*, M. A. Babar, M. Vierimaa, and M. Oivo, Eds., vol. 6156 of *Lecture Notes in Business Information Processing*, pp. 3–16, Springer, 2010.

[8] co. uk, wired.co.uk. Ubisoft is using AI to catch bugs in games before devs make them. https://www.wired.co.uk/article/ubisoft-commit-assist-ai, 2018.

[9] A. Giantsios, N. Papaspyrou, and K. Sagonas, "Concolic testing for functional languages," *Science of Computer Programming*, vol. 147, pp. 109–134, 2017.

[10] T. Lindahl and K. Sagonas, "TypEr: A type annotator of erlang code," in *Proceedings of the Erlang'05 - ACM SIGPLAN 2005 Erlang Workshop*, pp. 17–25, September 2005.

[11] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, K. Rikitake and E. Stenman, Eds., pp. 39–50, Tokyo, Japan, September 2011.

[12] M. Cronqvist, https://github.com/massemanet/redbug, 2017.

[13] A. B. Ericsson, dbg. http://erlang.org/doc/man/dbg.html, 2017.

[14] A. B. Ericsson, Trace tool builder. http://erlang.org/doc/apps/observer/ttb_ug.html, 2017.

[15] A. Till, erlyberly. https://github.com/andytill/erlyberly, 2017.

[16] R. Carlsson and M. Rémond, "EUnit - A lightweight unit testing framework for erlang," in *Proceedings of the Erlang'06 - Proceedings of the ACM SIGPLAN 2006 Erlang Workshop*, p. 1, usa, September 2006.

[17] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pp. 268–279, September 2000.

[18] T. Arts and J. Hughes, "Erlang/quickcheck," in *Proceedings of the In Ninth International Erlang/OTP User Conference*, 2003.

[19] A. B. Ericsson, Debugger User's Guide. http://erlang.org/doc/apps/debugger/users_guide.html, 2018.

[20] R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit, "EDD: A declarative debugger for sequential Erlang programs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 8413, pp. 581–586, 2014.

[21] T. Lindahl and K. Sagonas, "Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story," in *Programming Languages and Systems*, vol. 3302 of *Lecture Notes in Computer Science*, pp. 91–106, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[22] S. Anand, E. K. Burke, T. Y. Chen et al., "An orchestrated survey of methodologies for automated software test case generation," *The Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[23] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry, *The Emerging Field of Test Amplification: A Survey*, abs/1705.10692, CoRR, 2017.

[24] K. Taneja and T. Xie, "DiffGen: Automated Regression Unit-Test Generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 407–410, L'Aquila, Italy, September 2008.

[25] K. Taneja, T. Xie, N. Tillmann, and J. De Halleux, "eXpress: Guided path exploration for efficient regression test generation," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011*, pp. 1–11, Canada, July 2011.

[26] L. Naslavsky, H. Ziv, and D. J. Richardson, "MbSRT2: Model-based selective regression testing with traceability," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, ICST 2010*, pp. 89–98, fra, April 2010.

[27] Y. Chen, R. L. Probert, and H. Ural, "Model-based regression test suite generation using dependence analysis," in *Proceedings of the 3rd Workshop on Advances in Model Based Testing, A-MOST 2007, co-located with the ISSTA 2007 International Symposium on Software Testing and Analysis*, pp. 54–62, London, United Kingdom, July 2007.

[28] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pp. 397–406, Antwerp, Belgium, September 2010.

[29] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008*, R. Draves and R. van Renesse, Eds., pp. 209–224, USENIX Association, San Diego, California, USA.

[30] Z. Zhang, J. Huang, B. Zhang, J. Lin, and X. Chen, "Regression Test Generation Approach Based on Tree-Structured Analysis," in *Proceedings of the 2010 International Conference on Computational Science and Its Applications*, pp. 244–249, Fukuoka, Japan, March 2010.

[31] B. Korel and A. M. Al-Yami, "Automated regression test generation," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 1998*, pp. 143–152, usa, March 1998.

[32] K. Yu, M. Lin, J. Chen, and X. Zhang, "Practical isolation of failure-inducing changes for debugging regression faults," in *Proceedings of the 2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012*, pp. 20–29, deu, September 2012.

[33] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 765–784, 2013.

[34] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "DARWIN," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 3, pp. 1–29, 2012.

[35] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *Proceedings of the 29th International Conference on Software Engineering*, pp. 75–84, Minneapolis, MN, USA, May 2007.

[36] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, "DSpot: Test Amplification for Automatic Assessment of Computational Diversity," *CoRR*, 2015, abs/1503.05807.

[37] M. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A Tool for Automatically Detecting Variations Across Program Versions," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pp. 241–252, Tokyo, Japan, September 2006.

[38] M. Mirzaaghaei, "Automatic test suite evolution," in *Proceedings of the SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*, T.

Gyimóthy and A. Zeller, Eds., pp. 396–399, Szeged, Hungary, September, 2011.

[39] I. Bozó, M. Tóth, T. E. Simos, G. Psihoyios, C. Tsitouras, and Z. Anastassi, “Selecting Erlang Test Cases Using Impact Analysis,” in *Proceedings of the AIP Conference*, vol. 1389, pp. 802–805.

[40] R. Taylor, M. Hall, K. Bogdanov, and J. Derrick, “Using behaviour inference to optimise regression test sets,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 7641, pp. 184–199, 2012.

[41] M. Tóth, I. Bozó, Z. Horváth, L. Lövei, M. Tejfel, and T. Kozsik, “Impact analysis of Erlang programs using behaviour dependency graphs,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 6299, pp. 372–390, 2010.

[42] I. B. M. Tóth and Z. Horvóth, *Reduction of Regression Tests for Erlang Based on Impact Analysis*, 2013, Reduction of Regression Tests for Erlang Based on Impact Analysis.