Erlang Code Evolution Control

David Insa, Sergio Pérez, Josep Silva, and Salvador Tamarit[(✉)]

Universitat Politècnica de València, Camí de Vera s/n, 46022 Valencia, Spain
{dinsa,serperu,jsilva,stamarit}@dsic.upv.es

Abstract. In the software lifecycle, a program can evolve several times for different reasons such as the optimisation of a bottle-neck, the refactoring of an obscure function, etc. These code changes often involve several functions or modules, so it can be difficult to know whether the correct behaviour of the previous releases has been preserved in the new release. Most developers rely on a previously defined test suite to check this behaviour preservation. We propose here 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; the choice of an expression of interest whose behaviour must be preserved; and the recording of the sequences of values this expression is evaluated to. All the presented work has been implemented in an open-source tool that is publicly available on GitHub.

Keywords: Code evolution control · Automated regression testing
Tracing

1 Introduction

During its useful lifetime, a program might evolve many times. Each evolution is often composed of several changes that produce a new release of the software. There are multiple ways to control 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* [13,19] to assure that a desired behaviour of the original program is kept in the new release, but there exist other alternatives such as the static inference of the impact of changes [7,9,11,14].

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

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

© Springer International Publishing AG, part of Springer Nature 2018
F. Fioravanti and J. P. Gallagher (Eds.): LOPSTR 2017, LNCS 10855, pp. 128–144, 2018.
https://doi.org/10.1007/978-3-319-94460-9_8

the maintainability or efficiency; or for other reasons such as obfuscation, security improvement, parallelization, distribution, platform changes, and hardware changes, among others. Programmers that want to check whether the semantics of the original program remains unchanged in the new release usually create a test suite. There are several tools that can help in all 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. Therefore, our technique can check the evolution of the code even if no test suite has been defined.

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 asserts) or adding more code (as it happens in unit testing). In this paper, we introduce the ability to specify *points of interest* (POI) 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.

In our technique, (1) the programmer identifies a POI, typically a variable[1], 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 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. When two versions of a program are available at the beginning of the test generation process, we can use both for the generation of ITCs in step (2). In this alternative mode, step (3) is not performed as a separated step but it is integrated in step (2). We named this execution mode the *comparison mode*, and the one that uses only one version the *suite generation mode*. Note that, as it is common in regression testing, both modes only work for deterministic executions. This does not mean that they cannot be used in a program with concurrency or other sources of indeterminism, but POIs not affected by these features should be used in these cases.

[1] While our current implementation limits the POI to variables, nothing prevents the technique from accepting any expression as the POI.

We have implemented our approach in a tool named *SecEr* (*Software Evolution Control for Erlang*), which implements both execution modes and 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 [5], a concolic testing tool, to generate an initial set of test cases; TypEr [10], a type inference system for Erlang, to obtain types for the input functions; and PropEr [12], 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 that requires user intervention in our technique is, at each version, selecting the POI that is supposed to produce the same trace. This task is easy when the performed changes are not too aggressive, but it could be more difficult when the similarities between both codes are hard to find. In those cases where the versions are too different, the expression that define the values returned by the main functions can be a good starting point to start checking the behaviour preservation. In case the users need more refined or intricate POIs, they could move them following the control flow backwards on both versions of the code.

Example 1. Consider the real commit in the `string.erl` module of the standard library of the Erlang/OTP whose commit report is available at:

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

This change optimizes function `string:tokens/2`. SecEr can automatically check, using the comparison mode, whether this change preserves the original behaviour with a single command. We only need to indicate the two files that must be compared, and a POI for each file. Then, the tool automatically generates test cases that evaluate the POIs, trying to cover as many paths as possible. In this particular example, the POI in both versions is placed in the output of function `string:tokens/2`. SecEr generated 7044 test cases (see Listing 1.8 [6]) that reached these POIs, and it reported that both versions produced the same traces for all test cases.

We can now consider a different scenario and introduce a simple error like, for instance, replacing the expression in line 253 of the new release with the expression `[C | tokens_multiple_2(S, Seps, Toks, [C])]`. In this scenario, SecEr generates 6576 test cases, 5040 of which have a mismatch in their traces (see Listing 1.9 [6]). SecEr stores all the discrepancies found in a file, and it also reports one instance of these failing test cases: ITC `tokens([12,4,5],[2,3,2,5,0,1])`, for which the original program produces the trace `[[[12,4]]]` whereas the new one produces the trace `[[4,[12,4]]]`.

2 A Novel Approach to Automated Regression Testing

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

the white boxes represent intermediate results, and the initial processes of each phase are represented with a thick border box. All these boxes are connected by continuous black arrows (representing the control flow between processes) or by dot-dashed arrows (representing that some data is stored in a database). Finally, there are also parts of these figures contained in light grey boxes with dashed borders. This means that they are only used in the comparison mode. In this case, the process only has two phases instead of three.

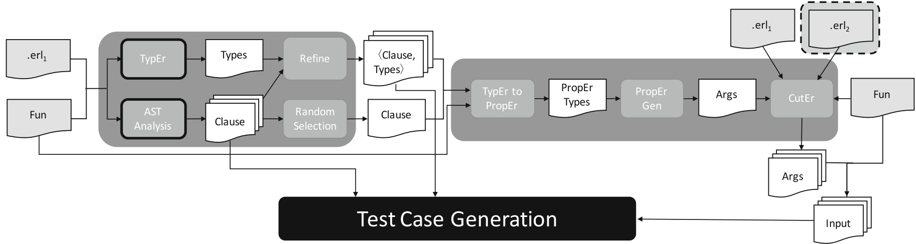

Fig. 1. Type analysis phase

The first phase (Type analysis), depicted in Fig. 1, 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 an input function (`Fun`) specified by the user[2] (for instance, function `exp` in the `math` module). Then, TypEr is used to obtain the types of the parameters (from now on just *types*) 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 types of each clause, but global types 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 types 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., ⟨*Number, Integer*⟩ can be instantiated to ⟨*4.22, 3*⟩ or ⟨*6, 5*⟩). 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., {⟨*1.5, 6*⟩, ⟨*2, 1*⟩, ⟨*1.33, 4*⟩}). Finally, this set is combined with the function to be called to generate 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 Sect. 3.1.

[2] We show the process for only one input function. In case the user defined more than one input function, the process described here would be repeated for each of them.

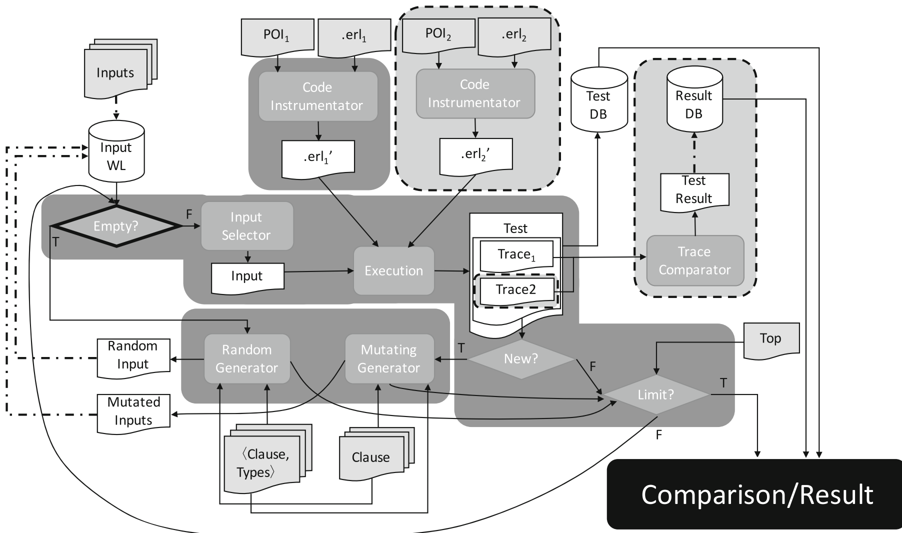

Fig. 2. Test case generation phase

The second phase, shown in Fig. 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 (i.e., the traces). Then, we store all ITCs provided by the previous phase onto 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 executed against the instrumented code, so a trace is produce as a side-effect. The executed ITC and the obtained trace form 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. This phase is slightly modified for the comparison mode where we can directly compare the traces generated by both of them to check discrepancies. Moreover, we only mutate an ITC when the pair of traces generated by both versions has not been obtained before. The entire process is explained in detail in Sects. 3.2 and 3.3.

Finally, the last phase (shown in Fig. 3) checks whether the new version of the code passes the test suite. This suite is only generated in the suite generation mode. In fact, this phase is only applied in this mode. First, the source code of the new release is also instrumented to compute the traces produced at its POI. Then, all the previously generated ITCs are executed and the traces produced are compared with the expected traces.

3 Thorough Description of the Approach

In this section, we describe in more detail the most relevant parts of our approach: the generation of ITCs (Sect. 3.1), the code instrumentation to obtain the traces (Sect. 3.2) and the test case generation (Sect. 3.3).

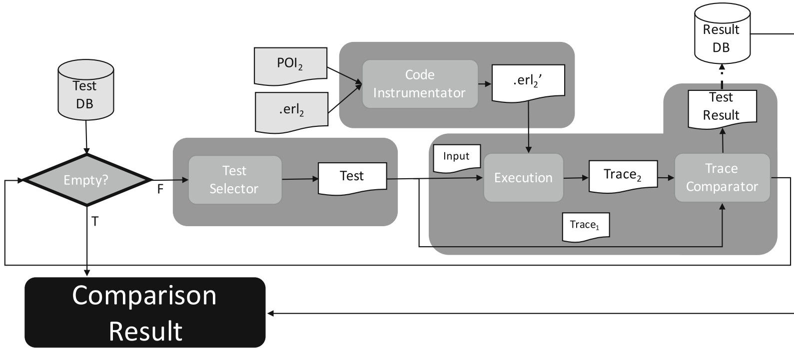

Fig. 3. Comparison phase

3.1 Initial ITC Generation

The process starts from the types 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. TypEr's standard output is an Erlang function type specification returned as a string, which needs to be parsed. For this reason, we have hacked the Erlang module that implements this functionality to obtain the types in a structure, easier to traverse and handle. However, the types returned by TypEr have (in our context) three drawbacks that need to be corrected since they could yield to ITCs that do not match a desired input function.

The first drawback is generated due to the fact that the types provided by TypEr refers to a whole function. We explain the first drawback with an example. Consider a function with two clauses whose headers are `g(0,0)` and `g(1,1)`. For this function, TypEr infers the function type `g( 0 | 1, 0 | 1 )`. Thus, the types obtained by TypEr for both parameters are expected to be formed by integers with the values `0` or `1`, but the value of each parameter may be any of them in every possible combinations. This means that, if we do not identify the types of each clause, we may generate ITCs that do not match any clause of the function, e.g. `g(0,1)`.

The other two drawbacks are due to the type produced for lists and due to the occurrence of repeated variables. We explain both drawbacks with another example. Consider a function with a single clause whose header is `f(A,[A,B])`.

For this function, TypEr infers the function type `f( 1 | 2, [1 | 2 | 5 | 6])`. Thus, the type obtained for the second parameter of the `f/2` function indicates that the feasible values for this 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 do not match the function, e.g. `f(2,[2,1,3,5])`.[3] Apart from that, we have another drawback caused by the fact that the value relation generated by the repeated variable `A` is ignored 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])`.

These three drawbacks show that the types produced by TypEr are too imprecise in our context, because they may produce test cases that are useless (e.g., non-executable). After the TypEr types inference, we can only resolve the type conflict introduced by the first drawback. The other drawbacks will be completely resolved when the ITCs are built.

To solve this first drawback, we traverse every clause classifying the possible types of each parameter with the TypEr deduced types. In this traversal every term is given its own value and every 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 `g/2` example we generate two different types: one for the clause `g(0,0)` where the only possible value for both parameters is `0`, and another one for the clause `g(1,1)` where the only possible values for both parameters is `1`. Furthermore, this transformation is useful not only for functions with multiple clauses, but also with the functions with a single clause like the `f/2` function. In this function, 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` after applying this solution.

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 functionality. One of them is the value generators, which, given a PropEr type, are able to randomly generate values belonging to such type. Thus, we can use the PropEr generators in our framework to generate values for a given type. However, TypEr and PropEr use slightly different notations for their types, which is reasonable given that their scopes are completely different. Unfortunately, there is no 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. This translation deals 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 translation, 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.

[3] Note that having ITCs that do not match with any function clause is not a problem. This scenario is common when TypEr returns the *any* type for some of the function's parameters. However, these ITCs are useless since they will produce an empty trace. This explains our effort to avoid them.

The second drawback is solved by recording the value each variable is assigned to and, whenever a variable already treated is found, the value recorded for its first occurrence is used.

Finally, we can feed CutEr with an initial call by using a randomly selected clause and its values generated by PropEr. CutEr is a concolic testing framework that generates a list of arguments that tries to cover all the execution paths. Unfortunately, this list is internally generated by CutEr but not provided as an output, so we have hacked CutEr to extract all these arguments. Additionally, the execution of CutEr can last too long or even not terminate. For this reason, we run CutEr with a timeout. In the comparison mode, the time assigned to CutEr is equally distributed to generate ITCs of both versions. Finally, by using this slightly modified version of CutEr we are able to generate the initial set of ITCs by mixing the arguments that it provides with the input function.

3.2 Recording the Traces of the Point of Interest

There exist several tools available to trace Erlang executions [2–4,16] (we describe some of them in Sect. 5). 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., [3]) has the problem 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 the code is executed.

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 the 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 firstly annotating the AST, then obtaining the path from the root to the POI and finally dividing this path in two. In order to do this, the following three steps are applied:

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.

2. The next step is to find in the code the POI. During the 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. When the POI is found, the traversal finishes. Thus, the output of this step is a path in the AST that yields to the POI.
3. The goal of this step is to divide the AST path into two sub-paths `(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 POI. We call target expression to those expressions that need a special treatment in the instrumentation. In Erlang, these target expressions are: pattern matchings, list comprehensions, and expressions with clauses (i.e., `case`, `if`, `functions`, ...).

After applying the previous steps we can instrument the POI. Most often, 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 the pattern of an expression, this expression needs a special treatment in the instrumentation. Let us show the problem with an example.

Example 2. Consider a POI inside a pattern {`1,POI,3`}. If the execution tries to match it with {`2,2,3`} nothing should be sent to the tracing server because the POI is never evaluated. Contrarily, if it tries to match the pattern with {`1,2,4`}, then value `2` must be sent to the tracing server. Note that the matching fails in both cases, but due to the evaluation order, the POI is actually evaluated (and the partial matching 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.[4]

We explain now how the actual instrumentation is performed. First, the `PathBefore` path is used to reach the deepest target expression that contains the POI. 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 Fig. 4. The first four rules are mutually exclusive, and when none of them can be applied, 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)` and `(CLAUSE_GUARD)`[5] transform an expression with clauses when the POI is in the pattern or in the guard of one of its clauses, respectively.

[4] 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.

[5] Function clauses need an additional transformation that consists in storing all the parameters inside a tuple so that they could be used in the case expressions introduced by these rules.

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

(PAT_GEN_LC) [e || gg] ⇒ [e || ngg]
      if       ([e || gg], _ ) = last(PathBefore)
             ∧ ( _ , pos(p_gen)) = hd(tl(PathAfter))
             ∧ ∃ i. 1 ≤ i ≤ length(gg) s.t. gg_i = p_gen <- e_gen
   where       ( _ , npoi, np_gen) = pfv(p_gen, tl(PathAfter))
             ∧ ngg_i = p_gen <- begin tracer!{add, npoi}, [np_gen] end
             ∧ ngg = gg_1 ... gg_{i-1}, np_gen <- e_gen, ngg_i, gg_{i+1} ... gg_{length(gg)}

(CLAUSE_PAT) e ⇒ change_clauses(e, ncls)
      if       (e, _ ) = last(PathBefore)
             ∧ ( _ , pos(p_c)) = hd(tl(PathAfter))
             ∧ ∃ i. 1 ≤ i ≤ length(cls) s.t. cls_i = p_c when g_c -> b_c
   where       cls = clauses(e)
             ∧ ( _ , npoi, np_c) = pfv(p_c, tl(PathAfter))
             ∧ nb_c = begin tracer!{add, npoi}, case np_c of cls end end
             ∧ ncls_i = np_c when true -> nb_c
             ∧ ncls = cls_i, ..., cls_{i-1}, ncls_i, cls_{i+1}, ..., cls_{length(cls)}

(CLAUSE_GUARD) e ⇒ change_clauses(e, ncls)
      if       (e, _ ) = last(PathBefore)
             ∧ ( _ , pos(g_c)) = hd(tl(PathAfter))
             ∧ ∃ i. 1 ≤ i ≤ length(cls) s.t. cls_i = p_c when g_c -> b_c
   where       cls = clauses(e)
             ∧ (poi, _ ) = last(PathAfter)
             ∧ nb_c = begin tracer!{add, poi}, case np_c of cls end end
             ∧ ncl = p_c when true -> nb_c
             ∧ ncls = cls_i, ..., cls_{i-1}, ncl, cls_{i+1}, ..., cls_{length(cls)}

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

Fig. 4. Instrumentation rules for tracing

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 index of an expression e on 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). Function $clauses(e)$ and $change_clauses(e, clauses)$ obtains and modifies the clauses of e, respectively. Function $fv()$ builds a free variable. Finally, there is a key function named pfv, introduced in Fig. 5, 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 $(h : t)$.

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

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

$$cv(list, map) = \begin{cases} ([], map) \quad \text{if } list = [] \\ ((fv : p'_t), map') \quad \text{if } list = (p_h : p_t) \wedge is_var(p_h) \wedge \neg\ is_bound(p_h) \\ \qquad \text{where } fv = fv() \ \wedge\ (p'_t,\ map') = cv(p_t, map \cup \{p_h \mapsto fv\}) \\ ((fv_{map} : p'_t), map') \quad \text{if } list = (p_h : p_t) \wedge is_var(p_h) \wedge p_h \mapsto fv_{map} \in map \\ \qquad \text{where } (p'_t, map') = cv(p_t, map) \\ ((p'_h : p'_t), map'') \quad \text{otherwise} \\ \qquad \text{where } (p_h : p_t) = list \ \wedge\ (children'_{p_h}, map') = cv(children(p_h), map) \\ \qquad\quad \wedge\ p'_h = change_children(p_h, children'_{p_h}) \\ \qquad\quad \wedge\ (p'_t, map'') = cv(p_t, map') \end{cases}$$

Fig. 5. Function pfv

3.3 Generation of New Test Cases Using PropEr and Test Mutation

The test case generation phase uses CutEr whose concolic analyses try to generate 100% branch coverage test cases. Sometimes, these analyses can last too long time. Moreover, even with a 100% branch coverage, frequently the test cases generated by CutEr can be insufficient in our context. 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` cannot detect any difference. Clearly, more values for `Y` are needed to detect the behaviour change in that expression.

Therefore, to increase the reliability of the test suite when using the suite generation mode, we complement the test cases produced by CutEr with a test mutation technique. Using a mutation technique is much better than using, e.g., the PropEr generator to randomly synthesize new test cases, because 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 new test cases using mutation is depicted in Fig. 6. The result of the function is a map from the different obtained traces (i.e., the outputs of the test cases) to the set of ITCs that produced them (i.e., the inputs of the test cases). 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[6] and *cuter_tests* are the test cases that CutEr generates. 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) and leaving the rest of arguments unchanged.

$$
tgen(top, pending, map) = \begin{cases}
map & \text{if } size(map) \geq top \\
tgen(top, pending', map') & \text{if } size(map) < top \\
 & \wedge \ \exists\ input \in pending \\
 & \quad \text{s.t. } trace(input) \mapsto _ \notin map \\
\quad \text{where } pending' = (pending \cup mut(input)) \backslash \{input\} & \\
\qquad \wedge\ map' = map \cup \{trace(input) \mapsto \{input\}\} & \\
tgen(top, \{proper_gen()\}, map') & \text{if } size(map) < top \\
 & \wedge \ \nexists\ input \in pending \\
 & \quad \text{s.t. } trace(input) \mapsto _ \notin map \\
\quad \text{where } map' = map & \\
\qquad \cup \{trace(input_p) \mapsto (\{input_p\} \cup inputs_{tp}) & \\
\qquad\quad \mid input_p \in pending \wedge trace(input_p) \mapsto inputs_{tp} \in \ map\} &
\end{cases}
$$

Fig. 6. Test case generation function

We can define an alternative way to generate tests for the comparison mode. In this scheme, the generated tests are only focused on comparing the differences between two version of the program. This generally yields to better tests for comparison, since some bugs can be explored deeply. For instance, consider that, for a given version of a program, ITCs $f(0,1)$ and $f(0,2)$ produce the same trace, e.g. $[2,3]$. According to the test generation function in Fig. 6, the second generated ITC (e.g., $f(0,2)$) is not further mutated. However, if we used another version of the program and the traces obtained by both ITCs were $[2,3]$ and $[4,5]$, respectively, it would be good to continue mutating the ITC since it is possible that a change in the second parameter reveals new discrepancies between the two versions. Therefore, we do not consider the trace of each individual version to decide whether it should be mutated, but the combination of both of them at the same time. We handle these cases by building a map from a pair of traces to a list of ITCs, instead of from a single trace. For example, the map from the previous example would contain $\langle [2,3],[2,3] \rangle \mapsto [f(0,1)]$ and $\langle [2,3],[4,5] \rangle \mapsto \{f(0,2)\}$ instead of $[2,3] \mapsto \{f(0,1), f(0,2)\}$. Therefore, the key in this new map is different for each ITC so, according to Fig. 6, $f(0,2)$ should be further mutated. In fact, the algorithm to generate traces considering both versions is exactly the same that the one depicted in Fig. 6. The only difference, apart from the change in the map structure, is that the function *trace*(*input*) is returning here a tuple $\langle trace_{old}(input), trace_{new}(input) \rangle$, where $trace_{old}(input)$

[6] In SecEr, a timeout is also used as a way to stop the test case generation.

and $trace_{new}(input)$ are the traces obtained for the ITC *input* using the old and the new version of the code, respectively. Note that in the comparison mode we should obtain a report of the discrepancies during the test generation phase. Obtaining the discrepancies between the traces of both versions is easy using the result of Fig. 6. The following formula calculates them from the map returned by *tgen*.[7]

$$discrepancies(map) = \bigcup\{inputs \mid (\langle t_o, t_n \rangle \mapsto inputs) \in map \text{ s.t. } t_o \neq t_n\}$$

4 The SecEr Tool

SecEr is able to automatically generate a test suite that checks the behaviour of a POI given two versions of the same program. There are two modes of running this tool. The first one allows for generating and storing a test suite of a program that is specific to test a given POI (the suite generation mode). The second one compares the traces of two different versions of a program and reports the discrepancies (the comparison mode).

Listing 1.1. `SecEr` command format

```
$ ./secer -f FILE -li LINE -var VARIABLE [-oc OCCURRENCE]
        [-f FILE -li LINE -var VARIABLE [-oc OCCURRENCE]]
        [-funs INPUT_FUNCTIONS] -to TIMEOUT
```

Listing 1.1 shows the usage of the SecEr command. If we want to run the command in the suite generation mode, we need to provide the path of the target file (`FILE`), the POI (`LINE`, `VARIABLE`, and `OCCURRENCE`), a list of initial functions (`INPUT_FUNCTIONS`),[8] and a timeout (`TIMEOUT`). If we execute the tool in the comparison mode, the required inputs are slightly different. For each version, we need to specify the path of its file and the details of its POI. Note that both versions use the same input functions. Therefore, these functions need to be exported in both versions of the program for the proper execution of SecEr.

The implementation uses a timeout as a limit to stop generating test cases, while the formalization of the technique uses a number to specify the amount of test cases that must be generated (see variable *top* in Sect. 3.3). This is not a limitation, but a design decision to increase the usability of the tool. The user cannot know a priory 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.

We have collected some interesting use cases and described how the source of a discrepancy can be spotted using SecEr in them. They are available at [6].

[7] According to the definition of function *tgen*, the selection of the ITC that is going to be mutated is completely random. However, in our tool we give more priority to the inputs belonging to *discrepancies*(*map*) since they are more susceptible of revealing new discrepancies between versions.

[8] The format for this list is `[FUN1/ARITY1, FUN2/ARITY2 ...]`. If the user does not provide it, all functions exported by the module are used as input functions.

5 Related Work

Automated behavioral testing techniques like Soares et al. [14] and Mongiovi [11] are very similar to our approach, but their techniques are restricted in the kind of changes that can be analyzed (they only focus on refactoring). Contrarily, 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.

Automated regression test case generation techniques like Korel and Al-Yami [8] are also very similar to our approach, but they can only generate test cases if they have available both the old and the new releases. Contrarily, in our approach, we can generate test cases with a single release, and reuse the test cases to analyze any new releases by only specifying the points of interest.

Yu et al. [20] 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. [21] use mutation injection and classification to identify commits that introduce faults.

Most of the efforts in regression testing research have been put in the regression testing minimization, selection, and prioritization [19]. In fact, in the particular case of the Erlang language, most of the works in the area are focused on this specific task [1,15,17,18]. 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 [7,9].

With respect to tracing, there are multiple approximations similar to ours. In Erlang's standard libraries, there are implemented two 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 [3], 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 [4] and the output trace of all the nodes can be formatted in many different ways. Cronqvist [2] presented a tool named redbug where a call stack trace is added to the function call tracing, making possible to trace both the result and the call stack. Till [16] implemented erlyberly, a debugging tool with a Java GUI able to trace the previously defined features (calls, messages, etc.) but also giving the possibility to add breakpoints and trace other features such as exceptions thrown or incomplete calls. All these tools are accurate to trace specific features of the program, but none of them is able to trace the value of an arbitrary point of the program. In our approach, we can trace both the already defined features and also a point of the program regardless of its position.

6 Conclusions

During the lifecycle of any piece of software, different releases 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 release preserves the correct behaviour of previous releases. 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 release.

In this work, we propose a new approach to automatically check whether the behaviour of 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 more useful test cases. All the ideas presented have been implemented and made publicly available in a tool called SecEr.

There are several interesting evolutions of this work. We would like to extend our technique to other paradigms. Therefore, we would have to define a generalization of this approach so it could be implemented in other languages. Then, some important features present in most languages (e.g., concurrency) could also be properly traced.

Some extensions to improve the results of the approach can be carried out. Adding some relevant data into the traces, e.g., computation steps, would allow for checking the preservation (or even the improvement) of non-functional properties such as efficiency. In a similar way, allowing for the specification of a list of POIs instead of a single one would enable the tracing of several functionalities in a single run, or the reinforcement of the quality of the test suite. Another way to increase the usefulness of our approach is a special tracing for function calls that would permit to distinguish easier whether an error is due to a discrepancy between either function implementations or calls' arguments. With respect to the mutation technique, we want to find more efficient ways of deciding what should be mutated, e.g., mutate certain elements of a list instead of changing the whole list.

Finally, the user experience can be enriched in several ways. Currently, the approach receives a sequence of functions as input. However, a sequence of concrete calls could be an alternative that could better lead to obscure errors. This alternative also makes it easier to reuse unit test cases building, in this way, a link between unit testing and our approach. 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 are also interested in investigating whether it

is possible to automatically infer POI candidates from the differences between two versions. We could use tools like *diff* to obtain the differences, and either, suggest the inferred candidate POIs to the user, or use them directly without user interaction. 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.

References

1. Bozó, I., Tóth, M., Simos, T.E., Psihoyios, G., Tsitouras, C., Anastassi, Z.: Selecting Erlang test cases using impact analysis. In: AIP Conference Proceedings, vol. 1389, pp. 802–805. AIP (2011)
2. Cronqvist, M.: redbug (2017). https://github.com/massemanet/redbug
3. Ericsson AB: dbg (2017). http://erlang.org/doc/man/dbg.html
4. Ericsson AB: Trace tool builder (2017). http://erlang.org/doc/apps/observer/ttb_ug.html
5. Giantsios, A., Papaspyrou, N., Sagonas, K.: Concolic testing for functional languages. Sci. Comput. Program. **147**, 109–134 (2017). https://doi.org/10.1016/j.scico.2017.04.008
6. Insa, D., Pérez, S., Silva, J., Tamarit, S.: Erlang code evolution control (use cases). CoRR, abs/1802.03998 (2018)
7. Jumpertz, E.: Using QuickCheck and semantic analysis to verify correctness of Erlang refactoring transformations. Master's thesis, Radboud University Nijmegen (2010)
8. Korel, B., Al-Yami, A.M.: Automated regression test generation. ACM SIGSOFT Softw. Eng. Notes **23**(2), 143–152 (1998)
9. Li, H., Thompson, S.: Testing Erlang refactorings with QuickCheck. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 19–36. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85373-2_2
10. Lindahl, T., Sagonas, K.: TypEr: a type annotator of Erlang code. In: Sagonas, K., Armstrong, J. (eds.) Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, Tallinn, Estonia, 26–28 September 2005, pp. 17–25. ACM (2005). http://doi.acm.org/10.1145/1088361.1088366
11. Mongiovi, M.: 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. ACM (2011)
12. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Rikitake, K., Stenman, E. (eds.) Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Tokyo, Japan, 23 September 2011, pp. 39–50. ACM (2011). http://doi.acm.org/10.1145/2034654.2034663
13. Rajal, J.S., Sharma, S.: Article: a review on various techniques for regression testing and test case prioritization. Int. J. Comput. Appl. **116**(16), 8–13 (2015)
14. Soares, G., Gheyi, R., Massoni, T.: Automated behavioral testing of refactoring engines. IEEE Trans. Softw. Eng. **39**(2), 147–162 (2013)
15. Taylor, R., Hall, M., Bogdanov, K., Derrick, J.: Using behaviour inference to optimise regression test sets. In: Nielsen, B., Weise, C. (eds.) ICTSS 2012. LNCS, vol. 7641, pp. 184–199. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34691-0_14
16. Till, A.: erlyberly (2017). https://github.com/andytill/erlyberly

17. Bozó, I., Tóth, M., Horváth, Z.: Reduction of regression tests for Erlang based on impact analysis (2013)
18. Tóth, M., et al.: Impact analysis of Erlang programs using behaviour dependency graphs. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) CEFP 2009. LNCS, vol. 6299, pp. 372–390. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17685-2_11
19. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012)
20. Yu, K., Lin, M., Chen, J., Zhang, X.: Practical isolation of failure-inducing changes for debugging regression faults. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 20–29. ACM (2012)
21. Zhang, L., Zhang, L., Khurshid, S.: Injecting mechanical faults to localize developer faults for evolving software. In: ACM SIGPLAN Notices, vol. 48, pp. 765–784. ACM (2013)