Is This Really a Refactoring? Automated Equivalence Checking for Erlang Projects

Bendegúz Seres
Eötvös Loránd University
Budapest, Hungary
k2sy12@inf.elte.hu

Dániel Horpácsi
Eötvös Loránd University
Budapest, Hungary
daniel-h@elte.hu

Simon Thompson
University of Kent
Kent, United Kingdom
s.j.thompson@kent.ac.uk

Abstract

We present an automated approach to checking whether a change to a repository is a refactoring, that is, it makes no change to the behaviour of the system. This is implemented in the EquivcheckEr tool, which detects the places in which the code has changed, and compares the old and new versions of all functions that are affected by the change, applying the functions to randomly generated inputs.

Our tool works for projects written in Erlang, and so needs to deal with effectful as well as pure functions. We aim only to report inequivalence when we have concrete evidence to that effect, avoiding any "false positive" counterexamples.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; **Software testing and debugging**; **Functionality**.

Keywords: Equivalence, Checking, Property-based, Testing, Refactoring, Erlang

ACM Reference Format:
Bendegúz Seres, Dániel Horpácsi, and Simon Thompson. 2024. Is This Really a Refactoring? Automated Equivalence Checking for Erlang Projects. In *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang (Erlang '24), September 2, 2024, Milan, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3677995.3678194

1 Introduction

Refactoring is the process by which the code for a project is changed, but in a way that its observable behaviour should not change; verifying that is typically done by regression testing, which depends on the quality and coverage of the test suite, or by manual code review. In this paper we present an automated alternative approach. We detect the places in which the code has changed, and then compare the old and new versions of all functions that depend on code that has changed by applying them to randomly generated inputs. This is implemented in Erlang using PropEr, the property-based testing tool based on QuickCheck, and delivered in the EquivcheckEr tool.

The tool is designed to work with projects written in Erlang, and so needs to deal with functions with side effects and communication behaviour, as well as pure functions. To make the tool as accessible as possible to developers, we align it with the git workflow, allowing arbitrary commits of projects to be compared, and integrate it with Visual Studio Code.

In order to be able to report results in all cases, we have to over-approximate behaviour; this approach is intended to meet the goal of only reporting an inequivalence when we have concrete evidence to that effect. Because of that, some non-refactorings will not be reported, but we expect that future work will enable us to narrow the gap between the approximated and real behaviour.

Using testing means that we will never be able to provide a guarantee that a change is indeed a refactoring; that can only be done in general by some kind of formal verification. However, our tool provides a robust and efficient automated approach that can easily catch unintentional errors and typos. The tool can also be incorporated into a CI process, for example by checking each commit that is tagged with a 'refactoring' label.

This work is distinctive in taking a pragmatic approach to the problem: rather than aiming to find a comprehensive solution to equivalence checking for a small subset of a programming language, we target a complete language, Erlang, so as to make the tool of value to practising programmers. We aim in future work to extend the precision of the tool, as well as provide a more ergonomic and scalable experience to users.

2 Background

The following sections will thoroughly discuss our method of property-based testing of refactoring correctness; in preparation for that, we first briefly introduce *refactoring* and *property-based testing*; furthermore, we discuss two basic concepts this work relies on, *program slicing* and *parse transformations*.

Erlang '24, September 2, 2024, Milan, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1098-8/24/09
https://doi.org/10.1145/3677995.3678194

2.1 Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written." ([3])

Essentially, refactoring is specified as a program transformation that improves the syntactic structure of the code (making it easier to comprehend) without affecting its (observable) behaviour. In particular, refactoring never introduces or eliminates features and bugs. Refactoring can be done manually or with refactoring tools like Wrangler [21]. Tool-assisted refactoring is usually done by the user first selecting the part of the source they wish to refactor, then specifying the type of refactoring they want.

For example, consider the following code snippet:

Listing 1. Fibonacci sequence in Erlang

```
fib(0) -> 1;
fib(1) -> 1;
fib(X) -> fib(X-1) + fib(X-2).
```

This is an implementation of the well-known Fibonacci sequence. If we want to rename the X variable to, for example, Y, we would select the variable in the source, and tell Wrangler to rename it to Y, which would result in the source being changed to:

Listing 2. Fibonacci with renamed variables

```
fib(0) -> 1;
fib(1) -> 1;
fib(Y) -> fib(Y-1) + fib(Y-2).
```

In this tiny example, it is apparent that the change does not affect the behaviour of the program as it consistently changes the variable names; changing the name of a bound variable is always possible, provided we do it in a way that respects capture-avoidance. However, not all refactorings are local or simple, there are examples of transformations that are way more complex and involved, where the refactoring execution has to check numerous *side-conditions*, such as avoiding capture or preserving effects. These conditions ensure that the refactoring can be done in a way that preserves behavioural equivalence; should the tool (or the developer carrying out the changes by hand) miss a condition or a required modification, the program's behaviour may get altered and spoiled.

2.2 Property-Based Testing

There are two different approaches for showing program properties: formal proof and testing. In general, testing has the advantage of requiring less expertise and being easier to automate. While testing is inherently incomplete and cannot prove the absence of bugs, it can disprove properties and show counterexamples. Thus, it can still provide good evidence that the program is likely to behave as expected (i.e., it is trustworthy), and this evidence can be made arbitrarily strong by making the number of test cases higher.

Listing 3. Functor law properties

```
prop_functor_id() ->
    ?FORALL(Xs, list(int()),
      lists:map(fun(X) -> X end, Xs) =:= Xs).

% Provided that f and g are int() -> int()
prop_functor_composition() ->
    ?FORALL(Xs, list(int()),
      lists:map(fun(X) -> f(g(X)) end, Xs) =:=
      lists:map(fun f/1, lists:map(fun g/1, Xs))).
```

Property-based testing is a technique that combines traditional unit tests with the ability to generate test data automatically. There are different (highly compatible) implementations of property-based testing for Erlang: the most mature tools are Quviq Quickcheck [1] and PropEr [17]—the latter is a free, open-source alternative to the former. PropEr allows its users to state program properties in Erlang source code, which will then be checked by generating random data using *generators*. These generators are provided for the primitive types, and it is also possible to extend PropEr by implementing generators for arbitrary types.

Take for example the functor laws [10], which can be stated with the PropEr properties shown in Listing 3. In this example, we use the `list(int())` generator to randomly generate lists of integers, and we check the property that the two resulting lists are equal, using Erlang's built-in equivalence operator.

PropEr generates 100 test cases by default (if no counterexample is found sooner), but this behaviour can be modified. When an example is found, for which the property does not hold, the tool tries to *shrink* the counterexample, meaning that it attempts to reduce the counterexample it found to the most trivial one that still invalidates the given property.

2.3 Slicing

The technique of taking some subset of a program, based on some property we are interested in, is called *program slicing* [22]. It has multiple applications in the context of static analysis, where it can be used for dependency analysis, dead code elimination or program optimization, among other things.

The subset of the program is called the *slice*, and the property is called the *slicing criterion*. One such criterion, that is relevant for our purposes, is the one that selects a path from the *call graph* of the program. The call graph is a directed graph, where each node represents a function, and the edges between nodes indicate a caller/callee relation, where the edge points from the caller to the callee.

Listing 4. Example caller/callee relations

```
f() -> g().
g() -> h().
h() -> ok.

j() -> i().
i() -> ok.
```

Listing 5. Parse transformation on message sending

```
transform_send(T) -> % `T` is the AST to transform
    L = erl_syntax:get_pos(T), % the affected line
    Out = {call, L,
      {remote,L,{atom,L,io},{atom,L,format}},
      [{string,L,"Sent:~p~n"},{cons,L,T,{nil,L}}]}
    {block, L, [Out, T]}.
```

Take as an example the snippet in Listing 4. If we choose the criterion to be the transitive closure of the relation of calling h explicitly, or calling a function that is already part of the closure, then the slice we get contains f, g and h. On the other hand, the functions i and j are not in this slice, because they do not call h, or any other function that transitively calls h. Normally, refactorings will not have an effect that trickles up all the way to the root of the callgraph, and so it will not be necessary to check the whole of the code.

2.4 Parse Transformation

Parse transformation is a feature of the Erlang compiler, which allows the user to apply arbitrary transformations to the AST (abstract syntax tree) in the compilation phase. By default, parse transformations can be invoked using the `-compile({parse_transform, Module})` compiler option, where `Module` is the module that implements the parse transformation. When the `parse_transform` option is specified, the compiler will first parse the source code, then use the specified module to do the transformation, and it compiles the newly created AST to bytecode.

An example of a simple parse transformation that modifies the original code by transforming every message-sending operation to one that also prints the message to the standard output, can be seen in Listing 5. This transformation will recursively traverse the whole AST, and apply `transform_send/1` on each term in it. If the given term represents a message-sending operation, it gets replaced by the extended one; in every other case, the subtree is returned without any modification.

Consider the expression displayed in Listing 6. This is a message-sending operation that can be reshaped with the transformation defined in Listing 5.

Listing 6. Message sending to be transformed

```
Pid ! {self(), hello}
```

Applying the transformation on it, we obtain the code shown in Listing 7: the message is enclosed in a begin-end block including both the newly added output statement and the original expression.

Listing 7. Transformed message sending

```
Pid ! begin
        io:format("Sent:~p~n",[{self(), hello}]),
        {self(), hello}
      end
```

3 Program Equivalence

Our work investigates the correctness of refactoring instances, i.e., checking semantic equivalence between two program versions, before and after the refactoring.

Semantic equivalence is a relation between two programs: two programs are related if they behave in the same way when executed in the same context (in other words, they are indistinguishable). It follows from Rice's theorem [11] that semantic equivalence is generally undecidable. To get around this problem, we have two options: either we only consider cases where semantic equivalence can be decided and thus formally proved, or we approximate it by running a large number of tests, trying to prove that the two programs are *not* equivalent. Our approach focuses on the latter.

3.1 The Equivalence Relation

We said that "two programs are related if they behave in the same way when executed in the same context". Formally, *behaviour* is defined by the operational semantics of expressions: $\langle e, E\rangle \longrightarrow^* (v, S)$ denotes that the expression e in an environment E evaluates to the value v and emits the series of side effects S.

In [4] the authors formally define equivalence[1], which determines when two programs *behave the same way*. Naive program equivalence (or simple behavioural equivalence) says that two expressions (e_1 and e_2) are equivalent if and only if in every environment E they evaluate to the same result v (value or exception) and emit the same list of side effects S (in our definition, this means text sent to the standard output and process messages sent); see Equation 1.

$$e_1 \equiv e_2 \overset{\text{def}}{=} \forall E, v, S : \langle e_1, E\rangle \longrightarrow^* (v, S) \iff \langle e_2, E\rangle \longrightarrow^* (v, S) \quad (1)$$

The environment[2] in this definition is assumed to contain the values of the free variables, the definitions of the applied functions, and an initial mailbox of the running process. The definition universally quantifies the environment, hence when checking this definition, we evaluate both programs in

[1]It is also shown that the equivalence relation is a congruence, which is essential in proving compound expressions equivalent.
[2]This should not be confused with syntactical contexts used in contextual equivalence.

lots of random environments (i.e., *the same context*), and if we find at least one environment in which the two expressions behave differently, we have disproved their equivalence (in other words, proved the *inequivalence*).

The above definition talks about expressions, but from it, we derive the following ones to check complete programs for equivalence. We say that two *functions* are equivalent if and only if their body expressions are equivalent (with the parameters being captured in the environment), and by further generalization, we define two Erlang *programs* to be equal if and only if they export the same set of function signatures and these exported functions (i.e., entry points) are pairwise equivalent.

3.2 The Equivalence Property

The above program equivalence definition can be turned into a testable property in a natural way: the universal quantifier over the environment becomes a forall-property, while the results can be checked with Erlang's built-in equality. However, we tailor the definition for the purpose of property-based testing. Firstly, we limit the scope of the functions we check to omit the trivially equivalent functions from testing. Secondly, we relax the definition by allowing non-terminating programs and API changes to be checked.

Scope. The equivalence definition requires *all* functions to be equivalent in *all* environments. On the other hand, when checking complex programs refactored locally, the majority of all functions are likely to be left intact and thus they can be omitted from the testing property[3]. To this end, we need a principled way to determine the functions potentially affected by the code changes.

The basis for our equivalence checking is the code *diff*, from which we compute the set of functions to test. We use the source files and the ASTs to identify each function definition altered by the change: this subset of functions is the initial scope, which we may need to expand with their callers. We proceed by computing the call graph, representing the caller/callee relations between the functions in the program.

When computing the closure of the initial set, we separate the functions based on whether their signature (name, number of arguments, types of arguments) has been changed. Functions with unchanged signatures are directly compared, but functions with altered signatures are checked via their callers, so we add them to the test set. Checking all callers in this case not only provides a way to invoke the changed functions but also verifies if all callers were refactored correctly. Should callers also have their signatures changed, we iteratively expand the set of functions to test. Finally, we slice the program according to the caller/callee relation as *slicing criterion*, which yields exactly those program parts that were affected in any way by the changes.

[3]Unchanged functions are syntactically equal, and therefore obviously equivalent.

Example. Consider the functions shown in Listing 8. There is a function named `f` (adding its two arguments), and we have two other functions, `g` and `h` calling `f`. We refactor this code by renaming `f` to `i`, but we forget to accommodate this change in `h`, so it still refers to `f` (see Listing 9). This change does not affect the behaviour of `f` (now `i`), but running the program after this refactoring would probably lead to wrong behaviour, which can easily be detected by checking the callers of `f`. In this simple example, the slice we are interested in will be the whole callgraph; yet, one can easily imagine a more complex program, where there are other functions calling `g` or `h`, but not `f`, in which case we would not have to consider these functions in our evaluation.

Listing 8. Before renaming

```
g(X) -> f(X,X).
h(X) -> f(X,X+2).
f(X,Y) -> X + Y.
```

Listing 9. After wrong renaming

```
g(X) -> i(X,X).
h(X) -> f(X,X+2).
i(X,Y) -> X + Y.
```

Over-approximation of Equality. Besides limiting the scope of the universal quantifier over functions to those that are modified, the property further relaxes the equivalence definition to allow more programs to be considered equivalent:

- In the relation, any two programs with different exported function signatures are non-equivalent. In contrast, the property does not require the compared programs to have the same interfaces (exported functions) but determines a set of comparable functions and only requires these to be equivalent.
- The property uses a strict timeout when evaluating functions, which means time-intensive and divergent functions may be considered to be equivalent; nevertheless, the effects (standard IO and message passing) are compared even in these cases.

When it comes to checking the results and the observed effects, we apply the following strategy. The comparison of the results (values or exceptions) is based on Erlang's equality operator (`=:=`), which checks the strict equivalence of simple or compound values as expected. As for the observed effects, we collect them into a list (in order) and check the lists for equality the same way as we do it with values [4].

Admittedly, the property over-approximates the equivalence relation, that is, it considers more program pairs to be equal than the formal relation: here, programs are considered equal unless shown to be unequal. The reason for this

[4]Process identifiers, function identifiers, and other references are filtered from the output before comparison, in order to eliminate false positive alarms stemming from differences in runtime-specific data.

is to minimize the false positive rate: we only report two programs to be non-equivalent (i.e., a refactoring to be incorrect) if there is simple and tangible evidence of behavioural mismatch. This fact is made clear by the project's GitHub page [14].

4 Evaluating and Observing Functions

With the conceptual basis for the equivalence property laid out, we proceed to discuss its core ingredients: the evaluation of functions and the observation of the effects. We start by discussing challenges that may arise even when only using the sequential subset of the language, such as isolation, random argument generation, divergence and IO. Then we discuss the handling of functions using concurrency-related primitives, particularly message passing.

4.1 Isolation

Equivalence can be shown by taking the original and the refactored functions, evaluating them in the same environment (or a sufficiently similar environment), and comparing their effects and return values. We do this by setting up two Erlang nodes, each responsible for executing parts of the original or the refactored programs in the same context, but in an isolated way (see Figure 1).

Each node can only see the modules of the version of the program it executes (see Section 4.2), and we modify the programs to set up the environment we can observe them in (see Section 4.6). After the nodes have executed the functions, the results are sent to a third node, which compares them, and on success, it provides subsequent test data. The code enabling this is shown in Listing 10.

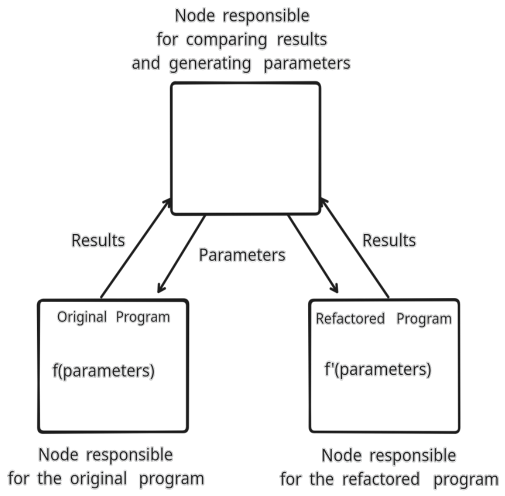
Node responsible for comparing results and generating parameters
Results
Parameters
Results
Original Program
f(parameters)
Refactored Program
f'(parameters)
Node responsible for the original program
Node responsible for the refactored program

Figure 1. Nodes used for testing

4.2 Multiple Module Versions

When a function is invoked, the BEAM looks for the bytecode that corresponds to the function's module, seeking for modules in the so-called *module path*. The module path is an ordered list of directories, where bytecode may reside. The BEAM will always load the first matching module.

By modifying the module path, we can control how a node finds the modules (and thus the functions) to run. This is how we ensure that each node executes the right version of the tested functions (see Listing 10 for more details on how the nodes are used). To achieve this, we explicitly set the right module path via the `code` module.

This method is also beneficial when checking functions of modules that exist in multiple applications, or specifically when checking functions of the standard library. In the latter case, we have to make sure that the module we ought to test is loaded before its standard library counterpart. However, when testing standard library functions, one also has to bypass the feature called *sticky path*. The purpose of sticky paths is to prevent accidental reloading of modules related to more basic parts of the runtime, such as the standard library, the kernel, or the compiler. To reload these modules, we have to explicitly "unstick" them first, bypassing this protection[5].

4.3 Generating Arguments

In practice, we just cannot test functions for each and every possible Erlang term in their parameters, and it is easy to see that generating input data in a completely random fashion would result in most if not all the execution leading to runtime errors, which would provide little information when we are considering the equivalence of the functions. This problem would be trivial in a language with a stricter typing discipline, like Haskell, but Erlang is a dynamically typed language.

Furthermore, Erlang functions tend to be partial, meaning that even well-typed data may not be specific enough; for example, a function that pattern-matches on a single integer input, but only does meaningful computation for the input 42, will yield exceptions on all other integers. The ratio of erroneous and successful function evaluations depends on the size of the function's domain, and the number of elements it can handle from that domain. Clearly, we want to maximize the number of evaluations that result in normal execution, giving us values we can compare for equivalence.

Although Erlang is dynamically typed, it supports type annotations (provided by the user) that can guide argument generation, and we can also rely on static analysis methods like success typing. *Success typing* [9] is a constraint-based type inference algorithm. It starts with the type of every

[5]This can be done on a per-case basis with the `code:unstick_dir/1` standard function, or alternatively, it is also possible to turn off this feature entirely by using the `-nostick` flag when starting the runtime.

Listing 10. Property used for establishing function equivalence

```
% Spawns a process on each node that evaluates the function and sends back the result to this process
-spec prop_same_output(pid(), pid(), atom(), atom(), [term()]) -> boolean().
prop_same_output(OrigNode, RefacNode, M, F, A) ->
    {Val1,IO1} = call_with_node(OrigNode, eval_func, [M, F, A], ?TIMEOUT),
    {Val2,IO2} = call_with_node(RefacNode, eval_func, [M, F, A], ?TIMEOUT),
    % Runtime-specific data is removed before comparison
    Out1 = {Val1,remove_unique(IO1)},
    Out2 = {Val2,remove_unique(IO2)},
    Out1 =:= Out2.

% Evaluates the given function, returning its result and any side-effect made by it
-spec eval_func(atom(), atom(), [term()]) -> {term(), [term()]}.
eval_func(M, F, A) ->
    capture_sideeffects(),
    % If any exception is thrown during evaluation, it is returned as the result
    RetVal = try erlang:apply(M, F, A) of Val -> {normal, Val}
             catch error:Error -> Error
             end,
    IO = stop_capture(),
    {RetVal, IO}.
```

possible term, then successively narrows it down, based on constraints from the term's context.

We use a working implementation of success typing called *Typer* [8], which can work on individual source files, although it requires already existing type information generated by Dialyzer [12]. Dialyzer works by first generating a so called Persistent Lookup Table (PLT), which contains the results of its preliminary analysis. The PLT then can be used later for various kinds of static analysis. TypEr uses the generated PLT to infer the type of functions.

In cases where success typing fails to find a specific type, we fall back to the `any()` type, which is the top type of Erlang, so it is inhabited by every possible term in the language. By using `any()` as the fallback, it is still possible to generate data, even in the lack of type information, although in most cases this will result in runtime errors and thus useless evaluations.

4.4 Divergence

Another challenge is comparing non-terminating programs. Erlang is not a total language, and many Erlang applications are non-terminating programs, like web or telecommunication services. While in the case of terminating functions, it is fairly easy to check their return values and their effects on the environment when deciding if they are equivalent, in the case of functions that do not terminate, our only option is to examine their effects before we eventually stop their evaluation. In this way, it is possible to prove their non-equivalence in a finite amount of time, but establishing their equivalence is theoretically impossible because their behaviour could differ at any point in time, so we can never stop the comparison and conclude that they are equal.

Checking the equivalence of non-terminating programs could be refined based on techniques like bi-simulation, but for now, we went with this simple and straightforward solution, that is, we impose a time limit, after which we stop the evaluation of the functions, and compare the effects, which is sufficient to disprove the equivalence.

4.5 Standard IO

Now let us consider the handling of side effects. Although Erlang is considered a functional language, nothing stops the user from reading from the standard input or writing to the standard output, to the disk, to some socket, or having any other kind of uncontrolled side effect. Apart from some special cases (like guards), IO is allowed anywhere in Erlang programs, and there are generally no indications, either conventional (like function names ending in '!' for lisps) or forced by the compiler (like monadic IO in languages like Haskell), that a given function has side effects. To extend the notion of equivalence to include side effects, we can keep track of any effects that a specific function had on its execution environment, while still taking note of the value it returned. To check if the two functions are equivalent, we compare the effects, together with the return value.

We do not perform a static analysis of whether a function will do IO when evaluated, but treat every function as one that can potentially have side effects and observe these effects for the purposes of checking the equivalence. We solve this problem by implementing our own *group leader* process, which is responsible for capturing any output. The group leader is the process that manages anything IO-related, by

Listing 11. Custom group leader sending output requests to a specified process and ignoring input

```
capture_group_leader(Pid) ->
    receive
      {io_request, From, ReplyAs, Request}
        when element(1,Request) =:= 'put_chars' ->
          From ! {io_reply, ReplyAs, ok},
          Pid ! {io, Request},
          group_leader(self(), self()),
          capture_group_leader(Pid);
      {io_request, From, ReplyAs, _} ->
          From ! {io_reply, ReplyAs,
                  {error, 'input'}},
          capture_group_leader(Pid)
    end.
```

receiving and sending messages to other processes. It is possible to replace this process with our own, overriding the way IO works.

Each time a process wants to write to an IO device, an `io_request` message is sent to the group leader, which will process this message, and execute the requested operation. When our modified group leader gets this request, it simply finds the exact string the process wanted to be written to IO device, and sends it to the process that evaluates the function, as shown in Listing 11.

For functions requesting input from standard input, or any other source, the group leader simply replies with an error message, stopping the evaluation of the function. We decided against generating random data for input because strings have very little structure, their space is infinite, and—without more thorough static analysis—the chance of generating relevant data is negligible.

4.6 Concurrency and Communication

Once concurrency and message passing are taken into consideration, examining the behaviour of some isolated part of a broader system becomes insufficient. Take for example the case of a function that receives a message and based on its content, does some computation. Without the context it depends on, this function will never terminate, because the process will block the first time it tries to read from its empty mailbox. Another notable example is an infinitely recursive function sending messages. Sending a message always succeeds (even if it will never be received), so we can observe such a function a finite amount of time and take the sent messages into consideration when deciding on equivalence.

This subsection describes how we used parse transformations, introduced in 2.4, to create the necessary context by populating the running process's mailbox. Then we explain how we observe and compare the concurrent behaviour of processes.

Populating the Ether. Parse transformations allow us to make arbitrary modifications to the program in the compilation phase, ultimately modifying the behaviour of the program to set up the necessary context by altering the semantics of communication primitives. To mitigate the problem related to "expected messages", we implement a parse transformation that modifies the program to populate its own mailbox.

Using parse transformations gives us nearly limitless flexibility and freedom to make arbitrary modifications to programs, it is exactly this expressiveness that makes it easy to misuse. By itself, the Erlang standard library only provides very basic tools for implementing parse transformations, so we decided to use a third-party library called `parse_trans` [23] to make the implementation of parse transformations easier.

For receiving messages, we need the mailbox to already contain messages, so the process will not block while trying to read from it. As we did before for function arguments, we can use PropEr to generate random messages for the process. Another important thing to realise is that nothing stops a process from sending messages to itself; we can use this fact to our advantage, and modify the function to generate random data with PropEr, send these messages to itself, and then go on with its original implementation: the `receive` expression gets rewritten to `self() ! RandomData, receive`.

Note that the messages that the process sends to itself cannot be fully random: when checking two versions of a function for equivalence, we need to make sure that the two instances receive the same messages in the same order. Since PropEr allows for specifying the seed for the generators explicitly, we can mitigate this issue by passing the same seed in the generators of the two running instances.

In our current implementation, the messages we populate the ether with are entirely random, independent of the receive block that reads them. When the `receive` block does pattern matching and guard checking on the incoming message, the randomly generated messages will likely cause stuck execution and thus false negative equivalence. A better solution would be to analyse the `receive` block and only generate data that conforms to the expected form, an idea we plan to incorporate in future versions.

Capturing Messages. To handle the checking of outbound communication, we can reuse our solution used in capturing standard output. In this way, the only thing needed is to modify the program, so that every time it sends a message, it will also print it to the standard output: `Pid ! Msg` is replaced by `Pid ! (print Msg, Msg)`. The group leader that captures the output will take care of this message, so it will be included when the equivalence is checked. Messages with a non-existent target are simply ignored by the runtime.

Observed messages may contain data that can vary based on the runtime context, which we would ideally exclude when checking equivalence. An example of this is process identifiers. Processes often send their PIDs, so the receiving process knows where to send a reply if needed. However, when we evaluate the two versions of a function, each of them is given a PID that is unique for the node it runs on. Although it is possible that the two processes get the same PID assigned to them by their runtime, we cannot expect this to happen every time.

If the function is implemented in a way that it sends its PID to some other processes, then the messages we compare will likely be different, even though their observable behaviour would be indistinguishable. We solve this problem by traversing each message sent and replacing any occurrences of PIDs with the same atom (`pid`). Similar considerations are to be made in the case of function references, unique references and other kinds of context-dependent data to be considered irrelevant for the purposes of equivalence checking (e.g., pseudo-random numbers or timestamps).

5 The EquivcheckEr Tool

The main result of this work is an open-source tool, called EquivcheckEr [14], which implements the equivalence property and allows the user to apply it in various scenarios, including checking the correctness of code refactoring.

In order to make the tool easy to adopt, we made sure its usage was sufficiently straightforward from a user's point of view. In accordance with this, we made sure that the tool has a low barrier of entry, and can be used by non-experts.

In particular, the tool is shipped as a script ready-to-use out of the box, it provides sensible default configurations (while also allowing the user to change these if needed), and it can be invoked both from IDEs (integrated with Visual Studio Code [20]) and from the command line. It is possible to use EquivcheckEr during the development phase by running it inside the editor, but it can also be part of automated software pipelines, increasing software quality standards.

It is also important to mention that EquivcheckEr can work on any two versions of a program, even if the refactoring was made by hand or by a tool. Only the source code is used to determine if the two versions are equivalent, no additional information about the scope of the change is needed.

In the rest of this section, we overview the main use cases, the command-line and editor interfaces and the existing configuration options.

5.1 Installation

We provide self-contained executables for EquivcheckEr on the project's GitHub page, under releases. For manual compilation, the project contains configuration for Rebar3 [18], a build tool for Erlang, so libraries that EquivcheckEr depends on will be automatically downloaded. Further details can be found on EquivcheckEr's GitHub page. Currently, EquivcheckEr only supports UNIX-like systems.

Listing 12. Help message

```
Usage:
  Equivalence checker [-jcs] [--json] [--commit]
      [--statistics] [<target>]
                      [<source>]

Arguments:
  target           target
  source           source

Optional arguments:
  -j, --json       json , default: false
  -c, --commit     commit , default: false
  -s, --statistics stats , default: false
```

5.2 Configuration

EquivcheckEr has its own configuration handling: it tries to locate the configuration file inside the `XDG_CONFIG_HOME` folder, as specified by freedesktop.org [15]—this currently limits the usage to UNIX-like systems, where the *XDG Base Directory Specification* is used. However, currently the only configuration parameter available is the path of the persistent lookup table (PLT) used when performing type inference. In fact, in most scenarios, the tool can rely on the default location of the PLT, but if needed, the user can override this and specify a custom location where it is loaded from.

5.3 Command-Line Interface

EquivcheckEr is easiest used through its command-line interface. The currently supported options are displayed in the helper text in Listing 12. In this subsection, we explain the invocation options in detail.

As one of our main goals was to make the tool user-friendly, we prioritized integration with other tools used by most software developers, so new users would not be expected to change their familiar workflows. Per this, EquivcheckEr is aware of existing Git [16] source repositories, but can handle local folders as well. In particular, the checker can be invoked in the following ways:

- By default (with the target and source parameters unspecified), the tool compares the current *working folder to* the *latest commit* (assuming the current folder is version-controlled).
- When the target is specified, the behaviour depends on the `--commit` option: if set, the parameter is interpreted as a commit identifier and the tool compares the current *working folder to* the *specified commit*; otherwise, the parameter is interpreted as a path and we compare the current *working folder to* the *specified folder.*

Listing 13. Statistics

```
> equivchecker -s
Results: [{"test.erl",{test,k,1},[[5]]},
          {"test.erl",{test,j,1},[[5]]},
          {"test.erl",{test,l,1},[5]}]
Number of functions that failed: 3
Average no. tries before counterexample found: 3.5
```

Listing 14. JSON output

```
> equivchecker -j
{
  "results": [
    {
      "filename": "test.erl",
      "mfa": "{test,k,1}",
      "counterexample": [0]
    },
    {
      "filename": "test.erl",
      "mfa": "{test,j,1}",
      "counterexample": [0]
    }
  ]
}
```

- When both the source and target parameters are given, we compare the *two specified commits* or the *two specified folders*, depending on whether the commit flag is set.

After running the tool with valid parameters, the output will contain all the functions that were found to be semantically different, and it also provides the counterexamples for each reported case.

Furthermore, there are two options to modify the default output: *json* and *statistics*. When the `--statistics` flag is used, the output will also contain information about the number of failed checks and the average number of tests needed before finding a counterexample (see Listing 13). This provides a quantitative summary of the testing process.

When the `--json` flag is used, EquivcheckEr will format its output as JSON (see Listing 14). This can be useful in the case of automation when other programs consume the output, and it is also used for providing the Visual Studio Code interface.

This simple CLI provides all necessary features for the programmer to apply the tool before committing changes, or can be employed in CI pipelines.

5.4 Visual Studio Code Interface

While we tried to make the command-line interface as simple and user-friendly as possible, we also created a prototype integration with the Visual Studio Code IDE. This graphical interface makes it even more convenient to use EquivcheckEr during the development phase.

VSCode is built with Electron [13], a cross-platform GUI framework based on web technologies. It can be extended by developing extensions for it, written in TypeScript [19]. It has a fairly extensive extension API that gives control over most aspects of the editor. VSCode also has the *Visual Studio Marketplace*, where extensions can be published for other users. The EquivcheckEr VSCode integration is currently in an early phase, and not available on the marketplace, but we intend to make it available once it is ready for use.

After installing the extension, a button for EquivcheckEr will appear in the status bar, allowing the user to invoke EquivcheckEr as an external process. For now, invoking the tool will apply the default behaviour of the CLI tool, which compares the current state of the working directory to the latest commit in version control. A notification is displayed to the user, indicating that the equivalence checking is in progress. When the checking is finished, the output of the process, formatted as JSON, is parsed and presented to the user in a new window as a simple Markdown text buffer. We believe that this simple usage fits the typical use cases, but we will add more customizability in future releases.

6 Evaluation

During the project, we tested the tool on a number of refactorings of different sizes and complexity, ranging from function renaming to local expression rewriting. Our primary approach was to create fairly small and simple examples, introduce errors into their refactoring, and see if the tool could find the error. This has resulted in a rather tight feedback loop, where we could readily see the effects of implementing features or fixing bugs.

Once the proof-of-concept was ready, we started experimenting with larger, public codebases. On the one hand, we sought existing refactoring commits, and on the other hand, we also did some refactorings ourselves to create testing data for the checker.

One of the earliest but most useful feedback was the time it took for the checking to finish. Initially, our implementation was sequential, in the sense that the comparisons of functions on random inputs were done one after the other. Although this was not a problem in smaller examples, it did not scale. Since the checker was written in Erlang, it was apparent that we needed to find a way to exploit the BEAM's concurrency capabilities; we went with the obvious solution and allowed the (isolated) function behaviour testing to run in parallel.

6.1 Case Study: `regexp` to `re`

As part of the evaluation, we wanted to test the tool on larger codebases, using well-known refactorings. We chose the reworking of the `regexp` module in the standard library,

Listing 15. xref_utils, before refactoring

```
match_list(L, RExpr) ->
    {ok, Expr} = regexp:parse(RExpr),
    filter(fun(E) -> match(E, Expr) end, L).

match_one(VarL, Con, Col) ->
    select_each(VarL, fun(E) -> Con =:= element(Col, E) end).

match_many(VarL, RExpr, Col) ->
    {ok, Expr} = regexp:parse(RExpr),
    select_each(VarL, fun(E) -> match(element(Col, E), Expr) end).

match(I, Expr) when is_integer(I) ->
    S = integer_to_list(I),
    {match, 1, length(S)} =:= regexp:first_match(S, Expr);
match(A, Expr) when is_atom(A) ->
    S = atom_to_list(A),
    {match, 1, length(S)} =:= regexp:first_match(S, Expr).
```

Listing 16. xref_utils, after refactoring

```
match_list(L, RExpr) ->
    {ok, Expr} = re:compile(RExpr),
    filter(fun(E) -> match(E, Expr) end, L).

match_one(VarL, Con, Col) ->
    select_each(VarL, fun(E) -> Con =:= element(Col, E) end).

match_many(VarL, RExpr, Col) ->
    {ok, Expr} = re:compile(RExpr),
    select_each(VarL, fun(E) -> match(element(Col, E), Expr) end).

match(I, Expr) when is_integer(I) ->
    S = integer_to_list(I),
    {match, [0,length(S)}] =:= re:run(S, Expr, [{capture, first}]);
match(A, Expr) when is_atom(A) ->
    S = atom_to_list(A),
    {match, [0,length(S)}] =:= re:run(S, Expr, [{capture, first}]).
```

containing regular expression-related functions, as our target. The `regexp` module was replaced by the `re` in OTP R13, while `regexp` itself was deprecated, and later removed from the standard library.

This necessitated the replacement of the usage of the module `regexp` by `re` everywhere it was used in the standard library. The API of `re` remained similar, although it had some easy-to-miss changes, like changing the way indexing works (starting from 0 instead of 1). Even so, upgrading to the newer module usually meant a simple refactoring, replacing a function used from the `regexp` module with a function having the same or similar name, but coming from `re` instead, with some slight modifications.

An example of such a refactoring, on the `xref_utils` module, can be seen in Listings 15 and 16. Note the change from `regexp:parse` to `re:compile`, the change of 1s to 0s (related to where indexing starts from) and the way the usage of the `first_match` function was made unnecessary by requiring the user to specify which match is needed by an argument for `re:run`.

By running the tool on commits related to these changes, we were able to identify functions that do in fact behave differently as a result of this refactoring. However, all of these functions were internal, not exported from the module, and all of them were called in a way that prevented these differences from manifesting and causing unwanted behaviour. Most of these problems resulted from discrepancies between the way `regexp` and `re` handle erroneous inputs. While `regexp` usually gave back some value even for meaningless input, `re` threw an error in these cases, making their behaviour differ.

We also found that running the tool on repositories like OTP, which consists of many applications, can often cause problems like *include* files not being found due to relative path resolutions. However, it is important to mention that OTP is somewhat of an outlier in this regard, not resembling the average Erlang project in size or complexity.

We also found that running the tool on projects as large as OTP necessitates the use of on-demand compilation, without which it is necessary to compile the whole codebase twice before the checking could start, once for each version. We will incorporate this feature in future releases. Yet, despite the above-mentioned limitations, we think the results show that our approach is feasible and has practical value.

6.2 Performance

The main factors that determine the runtime of the tool in a particular example are (1) the number of functions that were affected by the change, (2) the time that it takes to evaluate them, and (3) the number of runs needed before a counterexample is found. Smaller examples usually take a few seconds, making the tool convenient to use as part of a regular development workflow, in our opinion.

It is part of the design of the tool that there are limits on how long we allow a function to run, and how many test cases are generated: function evaluation is stopped after 1 second, and a maximum of 100 test cases are explored. These limits are not configurable for now, but it is our intention to make them so in the future, so users can make their own cost/benefit analysis, deciding how much time they are willing to sacrifice for more reliable results.

It is also important to mention that providing type information in the source files can speed up the process, due to the more precise data generated, meaning that fewer function evaluations are needed before a counterexample is found.

While evaluating the tool on the regexp case study in Section 6.1, we found that the running time stayed consistently around 10 seconds. However, we had to modify the tool to only compile modules affected by the change, due to the lack of the aforementioned on-demand compilation capability. Without this modification, the whole repository would have had to be compiled, increasing the runtime significantly.

7 Related Work

There is considerable literature based on program equivalence checking. Generally, it differs from the work reported here in two ways. First, it tends to address programs in tractable (sub-)languages of existing programming languages, rather than complete languages; secondly, the tests aim to check arbitrary programs for equivalence, whereas here we focus particularly on systems before and after refactoring, rather than arbitrary pairs of systems. The case of refactorings is special in that the two systems share much of their structure, and we exploit this to allow checking to be restricted to the parts of a program that are affected by the changes.

One way of checking function equivalence is to use testing, and a recent example of that is given by the PEQtest system [6], which checks a refactoring by deriving a test program from the original program by replacing each code segment being refactored with program code that encodes the equivalence of the original and its refactored code segment; the programs checked by PEQtest belong to a (relatively) small subset of C, in contrast to our work that aims to check programs written in a complete programming language. The PEQtest work builds on earlier work on context-aware, localised equivalence verification in PEQCheck [5] that checks C programs extended with the OpenMP library.

Property-based testing has been used to test refactorings for Erlang [7] by generating random refactoring commands for existing Erlang open-source code, and then checking whether they conform to a set of properties required of the refactoring. This work was extended to checking refactoring tools working on synthesised programs and comparing results of refactorings in two Erlang refactoring tools [2].

8 Conclusions

We have demonstrated the potential of using automated, property-based testing to compare arbitrary revisions of projects written in Erlang, reporting discrepancies only when there is concrete evidence. This requires handling of side effects and communication as well as pure functional behaviour, and we have taken a relatively simplistic approach to this. Future work will allow us to model behaviour more faithfully, and so to report more results. We encourage the engagement of the Erlang community in giving feedback and suggesting improvements to the tool.

In the near term, we aim to make ergonomic improvements to the system, packaging it for the VSCode marketplace, and for other distribution media such as nix, improving the user interface, and delivering an on-demand compilation of projects, which should result in a much better performance on larger projects. To improve the approximation of the systems under test, we aim to support type-correct mailbox generation, by examining the structure of `receive` statements, and to deal with standard input in a more sophisticated way, guided by the program context.

Acknowledgements

We are very grateful to the Erlang Ecosystem Foundation for their support of this work. Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

References

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang* (Portland, Oregon, USA) *(ERLANG '06)*. Association for Computing Machinery, New York, NY, USA, 2–10. https://doi.org/10.1145/1159789.1159792

[2] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. 2010. Quickchecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang* (Baltimore, Maryland, USA) *(Erlang '10)*. Association for Computing Machinery, New York, NY, USA, 75–80. https://doi.org/10.1145/1863509.1863521

[3] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA.

[4] Dániel Horpácsi, Péter Bereczky, and Simon Thompson. 2023. Program equivalence in an untyped, call-by-value functional language with uncurried functions. *Journal of Logical and Algebraic Methods in Programming* 132 (2023), 100857. https://doi.org/10.1016/j.jlamp.2023.100857

[5] Marie-Christine Jakobs. 2021. PEQCHECK: Localized and Context-aware Checking of Functional Equivalence. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE, Madrid, Spain, 130–140. https://doi.org/10.1109/FormaliSE52586.2021.00019

[6] Marie-Christine Jakobs and Maik Wiesner. 2022. PEQtest: Testing Functional Equivalence. In *Fundamental Approaches to Software Engineering*, Einar Broch Johnsen and Manuel Wimmer (Eds.). Springer International Publishing, Cham, 184–204.

[7] Huiqing Li and Simon Thompson. 2008. Testing Erlang Refactorings with QuickCheck. In *Implementation and Application of Functional Languages*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–36.

[8] Tobias Lindahl and Konstantinos Sagonas. 2005. TYPER: A Type Annotator of Erlang Code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*. ACM, New York. https://dl.acm.org/doi/abs/10.1145/1088361.1088366

[9] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical type inference based on success typings. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, New York. https://dl.acm.org/doi/abs/10.1145/1140335.1140356

[10] Encyclopedia Of Mathematics. 2020. *Functor*. https://encyclopediaofmath.org/index.php?title=Functor

[11] H. Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366. https://api.semanticscholar.org/CorpusID:120980829

[12] The Dialyzer team. 2006-2024. Dialyzer Reference. https://www.erlang.org/doc/man/dialyzer.html. Last accessed: 05-05-2024.

[13] The Electron team. 2013-2024. Electron Home Page. https://www.electronjs.org/. Last accessed: 05-05-2024.

[14] The EquivcheckEr team. 2024. EquivcheckEr Source Repository. https://github.com/harp-project/EquivcheckEr. Last accessed: 05-05-2024.

[15] The Freedesktop team. 2006-2024. freedesktop.org Home Page. https://www.freedesktop.org/wiki/. Last accessed: 05-05-2024.

[16] The Git team. 2005-2024. Git Home Page. https://git-scm.com/. Last accessed: 05-05-2024.

[17] The PropEr team. 2011-2024. PropEr Home Page. https://proper-testing.github.io/. Last accessed: 05-05-2024.

[18] The Rebar team. 2016-2024. Rebar3 Home Page. https://www.rebar3.org/. Last accessed: 05-05-2024.

[19] The Typescript team. 2012-2024. TypeScript Home Page. https://www.typescriptlang.org/. Last accessed: 05-05-2024.

[20] The VSCode team. 2015-2024. Visual Studio Code Editor Home Page. https://code.visualstudio.com/. Last accessed: 05-05-2024.

[21] The Wrangler team. 2005-2024. Wrangler Home Page. https://refactoringtools.github.io/wrangler/. Last accessed: 05-05-2024.

[22] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[23] Ulf Wiger. 2018-2024. Parse transform utilities. https://github.com/uwiger/parse_trans. Last accessed: 05-05-2024.

Received 2024-05-30; accepted 2024-06-27