Annales Univ. Sci. Budapest., Sect. Comp. 57 (2024) 245–270

REFACTORING CONCURRENT ERLANG APPLICATIONS FOR DISTRIBUTION

István Bozó, Melinda Tóth and Balázs Varga
(Budapest, Hungary)

Communicated by Zoltán Horváth

(Received January 10, 2024; accepted July 7, 2024)

Abstract. Applications require more and more resources. Distributed software can take advantage of today's easily accessible hardware and allow for easy scaling. Software development might reach the point where originally concurrently written portions of code should be transformed to work in a distributed manner. This process is tedious and error-prone when done manually.

Erlang is a functional programming language built for concurrent and distributed programming. In Erlang's actor model of concurrency, independent processes running on nodes communicate using message passing.

In our paper, we describe ways to transform concurrently written Erlang code to introduce distributed functionality, while preserving the overall semantics of the program. We define refactorings to transfer a selected process to another node, along with its registration if the process was registered. Using static analysis, we can discover and transform the parts of the code that refer to the moved process. This way, we can modify the send expressions, so that the communication between the processes can remain intact.

We define transformation schemes along with their preconditions and necessary compensations. We have implemented the basic cases of the transformations using the open-source RefactorErl static analyser and refactoring tool and used it for the validation of the schemes.

Key words and phrases: Refactoring, Erlang, distributed systems, concurrent systems, program transformations.
2010 Mathematics Subject Classification: 68N99.
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.
This work is a detailed version of a MaCS 2020 presentation.

1. Introduction

Applications are getting more and more complex and require an increasing amount of resources. Distributed software is able to scale easily to take advantage of the available resources. They offer great reliability and fault tolerance. The development of a software project might start without distribution in mind. At some point, as the complexity of the software increases, the developer might realise that the originally concurrently written program should be transformed to function in a distributed manner.

Making concurrent programs distributed is a wide topic, and as such can be approached in many ways. In this paper, we focus on Erlang programs, analysing and transforming them before runtime. Our approach is based on static program analysis and syntax tree transformations.

Erlang [9] is a functional programming language built for distributed systems. It was designed to easily express concurrent and distributed concepts at the language level. It uses the actor model of concurrency, where lightweight processes run on nodes (Erlang Virtual Machines) and communicate with each other using message passing. Concurrent and distributed Erlang programs use similar language constructs, making them ideal for transforming one into the other. The communication between processes on different nodes is handled transparently.

Our goal has been to transform concurrently written, single-node Erlang applications to work in a distributed environment across multiple nodes. Specifically, we focus on the task of moving a set of processes to be spawned on a different node with a known name. This transformation involves many aspects, as processes generally interact with each other using message passing, as well as with various resources on the node. When moving a process to a different node, much care needs to be taken, so that communication and access to resources stay intact. Therefore, this transformation, when done manually, is tedious and error-prone.

Erlang programs can be analysed by static analysis frameworks such as RefactorErl [18, 4]. By examining the code statically, without running it, RefactorErl builds up a Semantic Program Graph that represents the syntactic and semantic relationships within the program. It supports various simple and complex queries, which we can use to understand and reason about concurrent Erlang code. RefactorErl is also a refactoring tool, thus it can perform syntax tree transformations, allowing us to programmatically define modifications of the code.

Such tools have motivated us to explore ways to use static analysis to automate the process of discovering and transforming portions of the code that must be changed to move a process to another node. This is not a trivial task, as processes, as well as connections between sent and received messages, are syntactically implicit in Erlang [14]. We make use of queries and transformations that are available in RefactorErl, but define them in a general, tool- and framework-agnostic way in this paper.

Our main contribution is the definition of transformation schemes [19], focused on changing the spawning of easily movable processes, as well as processes that are bound by local registration. We discuss transformations that cause the registration to happen on the new node so that the moved process can still be referenced by its registered name. We also consider the passing of messages between processes on the original node and the moved process. We define methods to find the relevant send expressions in the code and transform them to work across nodes. For the messages from the moved process to the original node, we discuss a series of transformations to dynamically determine and pass the name of the original node to the appropriate send expressions at runtime.

For all transformation schemes, we explore the preconditions along with the necessary compensations and considerations. We define methods to find and select the relevant portions of the syntax tree using static analysis, and the changes that need to be made in the code to achieve the desired effect. We have implemented the basic cases of the transformations using the open-source RefactorErl framework.

The rest of this paper is structured as follows. In Section 2 we briefly introduce the Erlang programming language. In Section 3 we describe the introduced transformations, the conditions of applicability and the necessary compensations. Finally, in Sections 5 and 6 we present some related work and conclude the paper.

2. Concurrent and distributed programming in Erlang

Erlang was created for telecommunication applications, therefore implementation of concurrent and distributed programs is very natural in Erlang. Its concurrency is based on the actor model, implemented by lightweight processes that communicate with each other using message passing. The processes do not have a shared state, they can send and receive messages among each other to communicate or share data. The processes are lightweight, their creation and deletion take only a small amount of time, and they have little resource overhead.

Erlang provides a runtime system. The Erlang Virtual Machines (BEAM), are often called nodes. The process handling of Erlang is independent of the host operating system. The communication between processes on the same or different nodes is handled transparently by Erlang, and the syntactic aspect of writing concurrent and distributed code is similar. In this section, we describe the most important language features used in the transformations [9].

2.1. Process creation

Processes can be created using versions of the built-in *spawn* function. It returns a unique process identifier (Pid), which can be used to identify and address the function globally, across nodes.

- `Pid = spawn(Fun).` Creates a process on the node on which it was called. The created process starts to evaluate the implicit or explicit function expression given in the argument.
- `Pid = spawn(Module, Function, Args).` Similarly to *spawn/1*, it creates a process on the node where it was called. The function to be evaluated is specified by its module, function and the length of the Args list. The specified function must be exported.
- `Pid = spawn(Node, Fun).` Similar to *spawn/1*, but the process gets created on the node specified in the first argument. Every function referred in the closure of the function expression must be available on the node.
- `Pid = spawn(Node, Module, Function, Args).` Similar to *spawn/3*, but the process gets spawned on the node given in the first argument. Every function referred in the closure of the function expression must be available on the node.

In this paper, we will use the *spawn* functions to define the transformations, but Erlang also offers the similarly working *spawn_link, spawn_monitor, spawn_opt* functions, whose transformations can be done the same way.

Two additional useful functions for managing processes and nodes are `self()` which returns the Pid of the evaluating process; and `node()`, which returns the name of the node it runs on.

2.2. Locally registering processes

Processes can be registered in Erlang and they can be identified using a fixed name. This registration is local to the node where the process is located.

The registration is done using the `register(regname, Pid)` function. It always returns true (or throws an error when unsuccessful), and its arguments are the name by which to register the process, and the Pid identifying the process. A common pattern is to directly have a spawn expression as the second argument of register.

Registered names can be used in the place of Pids in some places, such as message passing. A process can only be registered by one name, and one name can only identify one process. Upon termination of a process, it is automatically unregistered.

2.3. Sending messages

Processes communicate by sending messages. It is an asynchronous operation. In Erlang, the ! operator allows us to send messages using the `To ! Message` syntax. The recipient of the message (*To*) can be one of the following:

- Pid. As described before, this can be used to refer to the process globally.
- A registered name. Can be used locally on the node where the process is registered.
- A tuple of {`RegName, Node`}. To use a registered name globally, we must specify the node on which the registered process is located.

3. Transforming concurrent Erlang programs

Static analysis can be used to discover certain relationships in the code, ranging from simple syntactic ones to more complex semantic relationships. Making use of the information extracted from these analyses, we define syntactic transformation schemes to make programs function in a distributed manner.

The transformations defined in this paper are to be performed as part of the development workflow, before runtime. The source code of an Erlang program, originally written for running concurrently on a single node is transformed in a way that some of its processes will run on a different node. The name of the other node must be known in advance, it is considered an input to the transformation.

This transformation has many preconditions, which will be defined at each step. Some of these are runtime preconditions, which cannot be derived from the source code using static analysis, but rather depend on the context and state of the environment in which the program will be run.

The schemes defined in the following may be considered refactorings in the sense that they intend to keep the overall functionality of the application intact. However, the inner logic of the application fundamentally changes, as some processes will run on a different node. The transformed program will be subject to the peculiarities of distributed computing, meaning that the identical behaviour of the program may not be guaranteed. This is however not the fault of the schemes, but an unchangeable property of distributed computing. Therefore the schemes still serve as a useful starting point for making applications distributed and more scaleable.

3.1. Moving a simple process

Erlang uses the actor model of concurrency. This means that the lightweight processes do not have a shared state, but rather communicate with each other using message passing. They may, however, make use of various resources that are present on nodes.

The simplest case of transformation is when the process is largely independent of its environment. This process may be messaged and message other processes using Pids (process identifiers), which function as global references to processes across nodes.

In this case, the process may be safely moved to another node, and the application will continue to function, provided that the following two runtime preconditions are fulfilled:

- The node to which the process is to be moved exists and is available at runtime, and the original node can connect to it. Both nodes must use either long or short names and have the same safety cookie.
- All code used by the migrated process must be available on the other node. This includes the closure of the moved functions, so every function that is transitively referenced by the moved process' code.

Each process is created by a `spawn[_link|_monitor]` expression, which can serve as the starting point of the transformation. In this simple case, the transformation is merely a matter of inserting the name of the other node, taken as a parameter, as the first argument of the spawn expression, since the syntax of spawning processes on other nodes is very similar to spawning processes locally. The transformation scheme is illustrated through a minimal example in Figure 1 in the case of *spawn/1* and *spawn/3*, and is identical for the other spawn-like functions.

```
% spawn/1
spawn(fun()-> a() end).

%spawn/3
spawn(mod, fun, Args).
```

```
% spawn/1
spawn('node@host', fun() -> a() end).

%spawn/3
spawn('node@host',mod,fun,Args).
```

Figure 1. The transformation of simple, highly mobile processes.

The moved process itself might contain spawn expressions, creating new processes locally. Since this is local relative to the spawning process, this transformation implicitly moves them along as well.

3.2. Transforming locally registered processes

Many processes are not as mobile as those described in the previous subsection. A typical pattern is that a process is registered and it can be referenced by a constant name. Process registrations are local to the node of the process. If the goal is to move a process that was locally registered, then to preserve the semantics, we need to update the registration to happen on the new node, as well as every reference that was using the registered name.

An additional precondition must be fulfilled for this transformation to be allowed. Each process may only be registered once, and registered names can not be conflicting. The node to which the process is moved must not contain a process that is registered as the same name as the moved process on the original node.

This transformation scheme is focused on the spawning and registration of processes. The goal is to manipulate the *spawn* and *register* expressions that refer to a certain process, using the spawn expression and the name of the other node as inputs to the transformation, so that

- The process created by the selected spawn expression gets spawned on the given other node.
- The moved process no longer gets registered on the original node.
- The moved process is registered on the new node with the same name as it was originally.

The scope of this transformation scheme is restricted to the original, spawning process. In this case, we can use static analysis to find the semantic connection between spawning and the registration of the process. This can be calculated by the combination of control flow and data flow analysis. Control flow analysis, starting from the *spawn* expression, can determine which portions of code are on the same execution path, and which expressions will be evaluated by the spawning process.

The goal is to find registrations that are guaranteed to refer to the spawned process. The transformation can handle the *register* expressions that are on one of these execution paths. If this is the case, backward data flow analysis is used to determine from the *register* expression, whether the process registered by it is the one that was selected as the transformation's candidate for moving. If the corresponding *register* expression is found, the name by which the process is registered must be saved for use in the later transformation steps.

The transformation scheme depends on the relationship between the spawning and registration of the process, which can be determined using the above technique. The following cases are possible:

1. There is no *register* that belongs to the spawned process. This case was discussed in the previous subsection.
2. The *spawn* expression is embedded into *register* as its second argument.
3. The spawning and registration happen in separate expressions. Typically, as the process is spawned, the Pid returned by *spawn* is stored in a variable, which is then used in the *register* expression later. The applicability of the transformation, in this case, depends on whether the expressions between the *spawn* and *register* fulfil certain conditions. This will be discussed later.
4. Multiple registrations belong to the spawned process on different execution paths. Even though a process may only be registered as one name at the same time, it is allowed to write code where different execution paths contain registrations of the same process.
5. There are a series of registrations and unregistrations of the process on the same execution path. In this case, the transformation is generally not possible. In a transformed distributed program, we could no longer guarantee that the process would still be registered at the same intervals as originally. We do not handle this case.

In the following, we discuss the three cases when the transformation is necessary and possible (cases 2,3 and 4) in more detail.

3.2.1. Embedded spawn and register

If the spawn expression is the second argument of a *register* expression, then the whole expression must be replaced by a *spawn/2* expression. We discuss the case of *spawn/1* and *spawn/3* expressions within the register, but the same approach can be used for transforming processes spawned by *spawn_link* or *spawn_monitor*. The spawned process will no longer be registered on the original node after the replacement. The first argument of the new spawn expression is the name of the other node, which is the parameter of the transformation. The second argument is an anonymous function, which runs on the other node. It is responsible for registering the process by the same name as it was originally, and then evaluating the same functions as before.

In the case of replacing a *spawn/1* expression, the existing anonymous function can be reused with a slight modification. As mentioned, the moved process must perform a self-registration before evaluating its expressions. To achieve this, the transformation inserts a `register(regname, self())` expression at the beginning of the *fun*, where *regname* is the name of the registration extracted and saved in the previous transformation step. The *self/1* function returns the *Pid* of the executing process.

When moving a process spawned using *spawn/3*, the approach is similar. The outer *register* expression is replaced by a *spawn/2*, with the name of the node as the first, and an anonymous function as the second argument. It will perform the self-registration the same way and then apply the original function. This can be done using the *apply(Mod, Fun, Args)* built-in function, which requires the same arguments as the original *spawn/3* to be replaced. The new spawn expression can therefore be constructed using the information from the spawn, and the containing register expressions.

```
% Transforming spawn/1
register(regname,
    spawn(fun() -> a() end)).
```

```
% Transforming spawn/1
spawn('node@host',
  fun() ->
    register(regname, self()),
    a()
  end),
true.
```

```
% Saving the Pid
register(regname,
  Pid = spawn(fun()-> a() end)).
```

```
% Saving the Pid
Pid = spawn('node@host',
  fun() ->
    register(regname, self()),
    a()
  end),
true.
```

```
% Transforming spawn/3
register(regname,
  spawn(mod, fun, [arg1, arg2])).



%
```

```
% Transforming spawn/3
spawn('node@host',
  fun() ->
    register(regname, self()),
    apply(mod, fun, [arg1, arg2])
  end),
true.
```

Figure 2. Transforming spawn/1 in the embedded case; the same transformation when the Pid is assigned to a variable; and the transformation in the case of spawn/3.

There are two small compensations that the transformation needs to perform. If the Pid returned by the inner spawn was assigned to a variable, then the new spawn's returned value must also be assigned to a variable by the same name. Additionally, the *true* atom must be added as the next expression after the newly inserted spawn, since the original register expression's value (which is always *true*) may have been returned as the last expression of a function. Therefore, the last expression of the transformed code must evaluate to *true*. If the register's *true* value was explicitly assigned to a variable, then the variable must be created and a newly inserted *true* atom needs to be assigned to it. The transformation scheme demonstrating all of these cases can be seen in Figure 2.

3.2.2. Separate spawn and register

Another typical pattern is when the Pid of the spawned process is saved to a variable, used in a pattern match, or returned from a function, and is used at a later point in the code to register the process. As described before, the combination of data flow and control flow analyses can be used to find the corresponding *register* expression.

The task is similar to the embedded case: the transformation must replace the *register* expression with the *true* atom, and replace the *spawn* with a *spawn/2* in the same way. This transformation may however change the order in which the registration and other expressions are evaluated. Originally, they were run by the same process, with certain expressions evaluated between them. In the transformed code, the self-registration will be the first action of the moved process, being executed concurrently with the spawning original process, losing all guarantees regarding the order of execution. Due to this, the transformation may not be performed in all cases, so we must introduce additional checks and preconditions:

- If any expression between the spawning and registration of the process has side effects, we do not allow the transformation. A clear example of why side effects must not be allowed is message passing. As a side effect, an expression may try to communicate with the spawned process, and it is possible for that interaction to only make sense when the registration has not happened yet. Since we have no way of reasoning about the actual side effects of a non-pure expression, we set it as a precondition that they may not exist between the spawn and register.

- If the registration has a data flow dependency on one of the expressions between it and the spawn, the transformation is not allowed. If a value required in the register expression is computed only after the spawn, then it can not be used within the transformed *spawn/2* expression. It may be possible to define a transformation that also moves the required computations to the new node since the expressions are pure, but this requires further research.

If the above conditions are fulfilled, the transformation can be performed according to the scheme in Figure 3. The reason behind this is that the register can be swapped with a pure expression directly before it if there is no data flow dependency between them. Repeating this process, the register can be brought directly next to the spawn, where the execution order is the same as in the embedded case (the registration would take place directly after the spawn), and thus the transformation can be performed in the same manner.

```
Pid = spawn(fun() -> a() end),




A = pure_function(12),
B = hello,
register(regname, Pid).
```

```
Pid = spawn('node@host'
  fun() ->
    register(regname, self(),
    a()
  end),
A = pure_function(12),
B = hello,
true.
```

Figure 3. Replacing spawn and register expressions that have pure expressions between them.

3.2.3. Registers on multiple execution paths

The result of the control flow analysis may show that the spawned process is registered on multiple execution paths. This time, the goal is to prove that the *spawn* expression could be moved to just before each registration of the process.

Let us try to propagate the spawn expression's node through the control flow graph using the following three rules.

1. The spawn can be swapped with the expression following it in the execution path if it is pure, and its value does not depend on the spawn's value (the Pid of the spawned process).
2. If the control flow graph diverges, propagate the spawn expression to each branch.
3. At a merge node (where multiple execution paths converge), all of the incoming branches must contain the spawn expression.

If by following these three rules, all registrations of the process can be reached by the spawn expression then we can perform the transformation. Figure 4 shows the transformation of a code sample with multiple control flows. This time, each of the *register* expressions will be replaced with the constructed *spawn/2* (followed by the true atom).

3.3. Restoring communication to the moved process

Processes typically do not work in isolation. They interact by sending and receiving messages. Erlang supports message passing across nodes at the language level using *send expressions*, which were introduced in Section 2.

```
X = something(),
Pid = spawn(fun() -> b() end),
case X of
    1  -> Y = one;
    11 -> Y = eleven;
    _  -> Y = notone
end,
function_without_side_effects(X),
case Y of
  one ->
    register(some, Pid);





  eleven ->
    register(other, Pid);





  notone  -> noreg
end.
```

```
X = something(),

case X of
    1  -> Y = one;
    11 -> Y = eleven;
    _  -> Y = notone
end,
function_without_side_effects(X),
case Y of
  one ->
    Pid = spawn('node@host',
      fun() ->
        register(some, self()),
        b()
      end),
      true;
  eleven ->
    Pid = spawn('node@host',
      fun() ->
        register(other,self()),
        b()
      end),
      true;
  notone -> noreg
end.
```

Figure 4. The transformation is allowed here, because using the three rules, the spawn could be brought to be before each register. The replacement happens at the registrations, keeping the registered names, and not losing the assignment of Pid or the *true* returned by the case branches.

When moving a process to another node, the send expressions addressing it with a Pid do not need to be modified, as it works transparently across nodes. In our research, we have focused on cases where the program has originally been written for a single node.

The main focus of this transformation step is when a locally registered process, addressed in send expressions using its name (`regname ! Message`), is moved to another node of a known name. The goal of the transformation scheme is to find the send expressions in the code where the recipient is the moved process and change the atom in the left argument to a tuple that also contains the new node's name.

The transformation scheme is a simple syntactic replacement, shown in Figure 5, since the name of the other node is considered a parameter of the transformation.

```
regname ! Message
```

```
{regname, 'node@host'} ! Message
```

Figure 5. Transformation of a send expression.

The challenge is to find the send expressions to transform. The name by which the moved process is registered is known from the previous section. We must find those send expressions whose recipient is the moved process. The potential candidates are those who send expressions whose left side is an atom (or an expression that is evaluated to an atom) and it matches the registered name of the moved process. This is a necessary, but not sufficient condition. The difficulty is caused by multiple execution paths of the program with multiple entry points and diverging control flows. We must assume that every exported function may be called in any context. It is possible that in one of these contexts, the same name is used to refer to a different process that is executed at a different time from the moved process. The transformation must not break the application in any of these cases.

3.3.1. Control flow-based approach

By performing control flow analysis, we can limit the transformation to those send expressions that are guaranteed to refer to the moved process. This is a conservative approach, it will not include any false positives that refer to a different process. There might be sending expressions in the code base that do not get transformed, even though they should have been.

The analysis starts from the *register* expression where the moved process was registered and finds all possible execution paths of the program after the registration. This search can be extended to also include the possible control flows of all processes spawned (locally) after the registration. When these parts of the program are running, the process in question is already registered on the node. Therefore, all send expressions on the execution paths with the recipient being the registered name are sure to refer to the process in question. The unregistration of the process is not handled by this approach.

If we sacrifice certainty, we might also start searching backwards from the *register*. Specifically, we look for processes that were spawned locally before the registration took place, and that are on the same execution path. Since the timing of the execution of different processes is not orderable, the send expressions in these processes may run after the registration takes place.

If there was such a message passing in the original code, we can assume that the intention was to address the process in question. The author of the original code must not have assumed that the send expression on this other process would run before the *register*. And since a registered name can only identify one process at a time, the send expression's recipient is the process to be moved.

Adding this extension invites the possibility of false positives, so it is to be used with care. Figure 6 shows the reasoning behind this approach. Send expressions to *b*'s registered name in the green portions are transformed, in the red portions they are not. Yellow portions are transformed if the extended version is used.

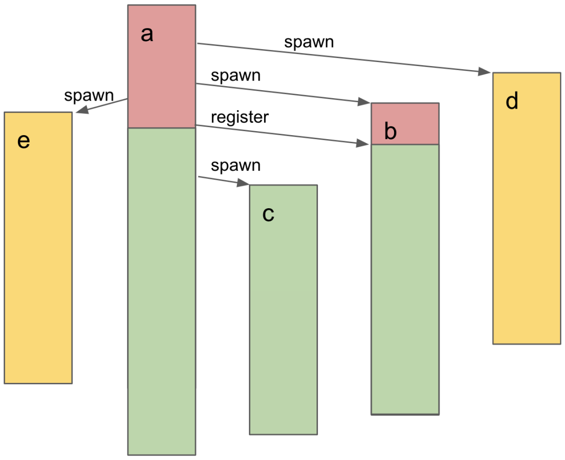

Figure 6. The timing of function spawns and registrations. The question is that if a send expression has the same left side as **b**'s registered name, can we be sure it refers to **b**? The green portions are guaranteed to be executed when this is the case, the yellow portions can not be decided, and in the red parts, the name does not refer to **b**.

3.4. Sending messages from the moved process

The moved process might have been sending messages to other processes on the original node, addressing them by their registered names, in the form of `name ! Message`.

When the process is moved, these recipient specifications are no longer valid, because their registrations are only present on the original node. Similarly to the previous point, the left side of the send expression must be transformed into a tuple, to include the name of the original node.

The original node's name is not an outside parameter of the transformation. It is runtime information that can not be decided through static analysis. It can be generated at runtime by evaluating the *node()* function on the node. This information then needs to be propagated dynamically to all functions of the moved process that send messages to a process on the original node.

The passing of the node name will happen through function arguments. We must make sure that the transformation does not break existing functionality, so changing the interface of the functions (adding an extra argument) is not desirable. Therefore, the transformation will duplicate some function definitions, transforming only the duplicates. With this compensation, the original code will remain usable as well.

3.4.1. Finding functions to transform

The task is to find a subset of the functions used by the moved process that will need to be transformed in some way (let us call these *marked* functions).

There are two rules for a function used by the process to be marked:

1. The function must be marked if it contains a send expression in the form of `atom ! Message`. This is a message passing to a process on the same (i.e. the original) node.

2. The function must be marked if it calls – or locally spawns a process that evaluates – an already marked function. The reason for this is that it will need to propagate the variable containing the original node's name to the called marked function.

The search starts from the *spawn* expression of the process to be migrated. If the process was created by *spawn/3*, the starting point is the referred function. In the case of *spawn/1*, the search will begin in the anonymous function expression's body.

The search can be done by a recursive algorithm that constructs a search tree based on the control flow graph and implements a depth-first search extended with some loop-handling functionality. In the tree, each node represents a function. A directed edge connects two function nodes **x** and **y** if **y** comes directly after **x** on an execution path in one of the control flows; or if **x** spawns a process locally that evaluates **y** (specified in *spawn/3* or the fun expression of a *spawn/1*). The algorithm's goal is to check rule 1 for each encountered function node, mark it accordingly, and propagate the marking back up through the tree according to rule 2. An example search tree built by the algorithm can be seen in Figure 7.

This search algorithm can be abstracted to work with any predicate. Rule 2 from above, describing the transitive nature of the marking, is part of the logic of the abstract algorithm. The predicate checked by the algorithm can be generic and set as a parameter. In this specific case of finding functions to transform, rule 1 from above is used as the predicate.

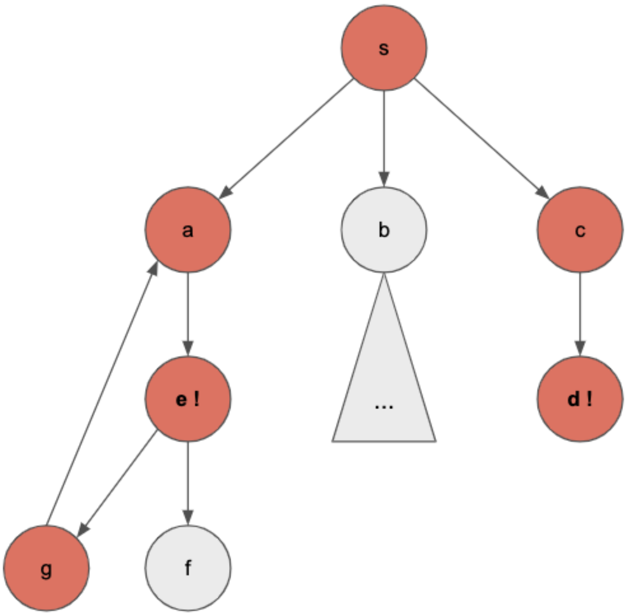

Figure 7. The search tree built by the algorithm. The search starts from node **s**, the nodes with ! fulfil the predicate (containing local message passing), and the red nodes are marked by the algorithm. The graph is based on the control flow graph but is limited to function calls and local spawns. Upon encountering a loop by traversing the edge between **g** and **a**, the depth-first search steps back and runs the loop handling logic.

Algorithm 1 shows the abstract version of the search algorithm. Each instance of the algorithm has a *Start* node and knows about the already *Visited* nodes. The children of the *Start* are discovered in lines 3-4. Line 5 recursively calls the algorithm for the children, adding *Start* to the list of visited nodes. The result of the recursive calls is a list of the *Start* node's marked children.

The conditional in lines 6-14 interprets the result. A node gets marked if any of its children are marked (line 13), or if the node itself fulfils the predicate (line 8). Otherwise, it does not get marked (line 10).

The other *if* expression encompassing almost the whole algorithm, whose conditional is on line 2, is necessary due to loops. Let us consider an example based on the tree in Figure 7 being built. The algorithm is on node **g**, and *Visited = [e,a,s]* (new nodes are always added to the beginning, so the list is in reverse order). Upon the recursive algorithm being called from **g** to **a**, a loop is detected (lines 1-2).

Algorithm 1 predicateSearch(Start, Visited, Predicate)

```
 1: OnLoop ← dropWhile(notEq(Start), reverse(Visited))
 2: if OnLoop == [] then
 3:     Calls ← getNextExecutionSteps(Start)
 4:     LocalSpawns ← getLocalSpawnedFunctions(Start)
 5:     Res ← flatten( [ predicateSearch(E, [Start | Visited], Predicate) || E ← Calls ++
    LocalSpawns ] )
 6:     if Res == [] then
 7:         if Predicate(Start) then
 8:             return [Start]
 9:         else
10:             return []
11:         end if
12:     else
13:         return [Start | Res]
14:     end if
15: else
16:     if any (node ∈ OnLoop fulfills Predicate) then
17:         return [Start]
18:     else
19:         return []
20:     end if
21: end if
```

Due to the nature of the depth-first search, at this point, **a** is not marked yet, since its marking depends on its subtree, which **g** is a part of. If the algorithm had no additional logic to handle this, and simply stepped back, **g** would be left unmarked, since after the marking of **a** it would not be revisited. Therefore, lines 16-20 run whenever a loop is discovered (when *Start* is **a**, *Visited=[g,e,a,s]* in our example). The *OnLoop* list contains the *[a,e,g]* nodes on the loop, and each node is checked for the predicate. If any of them fulfils it, the node **a** is considered to be marked, and the marking will correctly be propagated back in the loop, resulting in **g** not being left out.

The algorithm finds all of the functions that need to be modified. Let us make a copy of the definitions of these functions to preserve the originals, and continue with their transformation.

3.4.2. Transforming function calls and definitions

The copied function definitions need to take an extra argument that contains the name of the original node. The transformation modifies the parameter lists of each clause by adding a first argument for taking the node information. The name of this argument must be unbound in the scope of the function.

Since adding this argument increases the arity of the function copies, a precondition of the transformation is that there is no function in the same module with the same name and one higher arity as any of the copied functions. If there is such a function, the copies have to be renamed, for example by adding a postfix to the name to make it unique. If the original function was exported, the transformation needs to add an export to the copy as well.

The next step is to transform the function applications in the bodies of the function copies that call one of the original functions. The node name argument must be passed along. It is added as the first argument of the application. To find these, we filter the expressions in the copied function definitions, looking for function applications whose referred function is in the set of originals. The moved process may use functions that locally spawn other processes (this was one of the edges in the search tree). If this is done using *spawn/1* and a fun expression, the body of the *fun* is searched the same way.

The node name argument must also be passed along through *spawn/3* or *apply/3* calls, found by the same filtering technique. The difference is that the argument must be added to the beginning of the list in the third parameter of the *spawn* or *apply* calls.

Note that by performing this transformation, if a function was copied, all of the function copies will use the newly transformed version of it, while the originals will keep using the original version, allowing them to be used with the same interface, and not breaking any functionality. Figure 8 shows the transformation logic of this step. The copied function definition on the right is created.

```
% f, a, b are marked
    functions
f() ->
  a(),
  spawn(mod,b,[some,value]),
  not_marked_fun(123),
  f().
```

```
% f, a, b are marked functions
f(OrigNode1) ->
  a(OrigNode1),
  spawn(mod, b,
    [OrigNode1,some,value]),
  not_marked_fun(123),
  f(OrigNode1).
```

Figure 8. The definition of the function on the left is copied. The copied function takes a new argument (*OrigNode1*) and passes it along the calls and local spawns of marked functions. Functions that were not marked remain unchanged.

With this compensation, the functions of the moved process will correctly pass the node information around. If there were such functions (because somewhere a backward communication was performed), the node name must be given to the first function that the spawned process evaluates. The node name can be discovered at runtime by running the *node()* function on the original node, by the process that performs the spawning of the moved process.

Let us insert the `ThisNode = node()` expression before the spawn to be able to pass it to the first called (modified) function of the spawned process.

If the migrated process was spawned using *spawn/1*, the technique is the same as with modifying function calls – looking for them in the body of the *fun* expression – and adding the *ThisNode* variable as the first argument. With processes spawned using *spawn/3*, the modification is also the same as within the function copies: the *ThisNode* variable is added to the beginning of the list in the third argument of the *spawn*. This transformation step can be seen in Figure 9.

```
% spawn/1 case

spawn('node@host',
  fun() ->
    register(regname, self()),
    b(some, value)
  end).

% spawn/3 case

spawn('node@host',
  fun() ->
    register(regname, self()),
    apply(mod, a, [])
  end).
```

```
% spawn/1 case
ThisNode = node(),
spawn('node@host',
  fun() ->
    register(regname, self()),
    b(ThisNode, some, value)
  end).

% spawn/3 case
ThisNode = node(),
spawn('node@host',
  fun() ->
    register(regname, self()),
    apply(mod, a, [ThisNode])
  end).
```

Figure 9. Passing the name of the original node in a variable to the spawned process

3.4.3. Transforming the send expressions

The goal of these compensations is that the send expressions located in the copied function definitions, whose left side is an atom, can be transformed. We must be careful here, as there might be messages being sent to one of the processes that were moved as well. These processes were registered within the body of the function copies. In Section 3.2, we have computed the list of these registered names. So if the atom of the send expression is one of these, it must not be transformed, as both the sender and the receiver are on the new node.

With the function copy now taking the name of the original node as its first argument (let us call it *OriginalNodeName*, but the actual name will depend on the bound variables in the scope), the necessary information is readily available. The `name ! Message` send expression is transformed to `{name, OriginalNodeName} ! Message`.

This concludes the transformation step, the moved process (and those locally spawned by it) can send messages to processes that have remained on the original node. The original versions of the transformed functions were kept, so that they may be used in a different context, possibly by different nodes or processes without having to use a different interface.

There may be some functions that were only used by the moved process, and therefore the original version does not need to be kept. A series of semantic queries can be used to find unused function definitions after this transformation and remove them from the code. A detailed description of this compensation can be found in [19].

4. Validating the implemented transformations

The basic cases of the transformations are implemented in the RefactorErl static source code analyser and transformation framework.

While using formal methods to prove the transformations' correctness would be the most desirable, it is infeasible due to their complexity. There are ongoing attempts [11, 8] to formalize Erlang's semantics, and to describe refactorings by defining rewrite rules or behavioural equivalences. However, in their current stage, these methods are not applicable yet to the transformations presented in this paper. Thus we have tested the refactoring for a large number of different use cases to ensure its correctness. We have defined several unit tests based on RefactorErl's testing framework, and we have defined a manual testing process and applied it to different open-source projects.

The manual testing method involved the following steps:

1. Experiment with the original, concurrently written program. Run its exported functions, log some messages and try to understand the behaviour of the application.

2. Use RefactorErl's process analysis tools to aid the understanding of the process and messaging relationships.

3. Perform the desired transformation using RefactorErl.

4. Observe the transformed code and examine the changed parts. Check if there are any unexpected changes.

5. Behaviour testing: Experiment with the transformed program, and run its functions. Focus on the modified or newly copied functions. See whether at any time new errors are produced. See if running the program with the same parameter in the same context produces the results as before (this only applies in deterministic or pure programs). Run any available unit- and integration tests.

The transformation involves moving processes, removing and creating registrations and modifying communication between processes. Therefore, we have defined a checklist for behaviour-testing the transformed code to find any issues that might have been caused by the refactoring. The following non-exhaustive list provides a starting point for validating the success of the transformation, and ensuring that the application's behaviour is unchanged:

- The moved processes can access all of their required resources (these are available on the new node, accessible the same way as before, and compatible with the process). *Test: Run the functions that access resources on the new node. Look for errors.*
- The moved processes are no longer registered on the original node. *Test: Run registered() on the original node.*
- The moved processes that were registered before being registered on the new node. *Test: Run registered() on the new node.*
- Messages sent between processes on the original node and the moved processes (by Pid, registered name, or global) still function. *Test: Depends on the program. Try to produce behaviours that involve message passing. Log the received messages.*
- Messaging among the moved processes is intact. *Test: Same as above*
- Messaging among the processes that stayed on the original node still works. *Test: same as above.*

Further information about the validation and testing with examples can be found in [19] and [20].

5. Related work

Transforming concurrent applications to distributed ones using static transformations is not a very widely researched topic. However, there are various approaches by researchers, whose main goal – increasing performance, enabling easier scaling and fault tolerance – aligns with the theme of our research. Most of these papers are focused on the earlier step of this effort, transforming sequential programs to parallel; while some provide the theoretical background behind the transformations.

5.1. Process migration

Process migration means moving the execution of a process to another machine while it is running. This is advantageous for dynamic load balancing and configuration, allowing a high level of flexibility since parts of the execution can be moved around the distributed system.

Tanenbaum and van Steen's book on distributed systems [17] offers an additional explanation of the framework devised by Fuggetta et al. [10] for the categorisation of relationships of resources. It considers two factors, the relationship between a process and the resource; as well as between the resource and the underlying machine. The process can require the resource by identifier, value, or type; and the resource can be unattached, fastened, or fixed to the machine. The combination of these two factors gives 9 categories with different migration characteristics.

In [7], this categorisation is applied to Erlang processes and resources. The author notes that local registration is a fixed identifier resource, which means that the moving of locally registered processes is a difficult task. The proposed solution is to use a global reference, which in Erlang terms is either a {`registered name, node name`} tuple or a Pid-based messaging. The author also suggests the use of the *global* module as an alternative.

The runtime migration of Erlang processes has also been studied [15]. The authors have created a mechanism that allows the migration between two Erlang nodes without interruptions while keeping the state of the process.

5.2. Refactoring tools

The transformations described in this paper are designed to be implemented using the RefactorErl framework. However, there exist other Erlang refactoring tools as well. We used this tool because it supports thorough semantic analysis besides its transformation backend.

Erlang Syntax and Metaprogramming Tools [2] are a collection of modules for handling and transforming abstract syntax trees. One of its modules is *erl_tidy*, which provides basic automatic refactorings of Erlang code. Tidier [16] is another fully automatic refactoring tool for Erlang programs. It offers relatively simple refactorings involving a smaller scope and provides strong reliability guarantees for the transformed code.

Wrangler [13, 3] is a semi-automatic refactoring tool for Erlang programs, providing several refactorings ranging from simple to large structural changes. It provides interaction with Emacs and Eclipse interfaces. It is extensible as it provides an API for performing refactorings.

5.3. Process related refactorings

Wrangler includes some process-related refactorings, including registering a process, renaming a registered process or creating processes from function definitions. In the accompanying paper [14], the authors provide a thorough evaluation of the challenges of analysing processes. They note that the key difficulty is caused by the implicit nature of many process-related structures. The authors point out that Erlang processes do not have a clear syntactically defined scope, but may consist of various functions that may also be shared by other processes. Parts of code, even those involving process spawning or message sending may be used from various places. We have made sure to address this point when designing the transformation, by making copies of certain function definitions, as described in Section 3.4.

Wrangler's process-related refactorings use an Abstract Syntax Tree annotated with process information, but there are other approaches as well. Program slicing [21] is a method used to reduce a program to parts that produce a subset of the program's behaviour. Considering data and control flow dependencies, slicing techniques separate parts of the code that influence a given statement. A recent publication [12] uses slicing techniques to implement three refactorings to introduce concurrency to sequential Erlang code.

The PaRTE framework [5] combines the capabilities of RefactorErl and Wrangler, with the main goal of parallelising sequential Erlang code. The tool discovers parallelisation pattern candidates using algorithmic skeletons [6]. The framework is part of the Paraphrase-Enlarged project [1], whose goal is to statically detect parallel patterns in (Erlang) programs.

6. Conclusion and future work

In this paper, we have examined the task of transforming concurrently written Erlang programs to function in a distributed manner. We have defined transformation schemes that make use of static analysis to transform Erlang code so that some of its processes will run on a different node.

We have examined the preconditions of moving a registered process to preserve the overall semantics of the application. We have defined transformations that consider the relationship and relative location of the spawning and registration of the process.

To restore the communication between the moved process and the processes that remain on the original node, we have defined transformations to modify the send expressions. We have described the challenges of finding the relevant expressions. To allow messaging from the moved process, we have devised a search algorithm and a series of transformation steps to dynamically generate the original node's name at runtime, and pass it to the send expressions where it is necessary.

We have implemented the basic cases of the transformations described in the paper using RefactorErl. This implementation has been used to manually validate the refactorings. We have transformed multiple open-source example projects and compared the functionality of the original and the transformed versions.

Future work might be done towards improving analysis regarding the communication among processes. Currently, we can gain a limited understanding of the process relationships using static analysis. By better describing these connections, more specific and useful cases of the transformation could be defined. We could define metrics regarding the strength of the connection between certain processes by analysing messages passing between them, and then apply clustering algorithms to automatise the refactoring to achieve the highest performance or robustness.

There are still some cases for which we have not developed transformations. The goal is to cover these in the future and extend the already implemented refactorings. It is worthwhile to also study the applicability of the global module in the transformations.

In distributed systems, efficiently storing and accessing large amounts of data is not a trivial task. Erlang offers storage options such as ETS, DETS and Mnesia databases, with different distribution characteristics. For example, ETS tables are owned by a process, thus the movement of the process must involve compensations so that the table can still be accessed. There are many new transformations to be defined involving this topic.

Our contribution is the definition of schemes for common patterns of concurrent to distributed transformations. We have examined preconditions, querying and transformation steps, as well as additional considerations that need to be taken. The topic of our paper is relatively unexplored, and there is much future research to be done in this area.

References

[1] ParaPhrase Enlarged project, `http://paraphrase-enlarged.elte.hu/`, [Acc. 02.07.2024].

[2] Erlang Syntax and Metaprogramming tools, `https://www.erlang.org/doc/apps/syntax_tools/chapter.html`, [Acc. 02.07.2024].

[3] Wrangler, `https://refactoringtools.github.io/docs/wrangler/`, [Acc. 02.07.2024].

[4] **Bozó, I., D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, M. Tejfel and M. Tóth,** RefactorErl - Source Code Analysis and Refactoring in Erlang, in *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, 2011.

[5] **Bozó, I. V. Fördős, Z. Horváth, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown and K. Hammond,** Discovering Parallel Pattern Candidates in Erlang, in *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, page 13–23, Association for Computing Machinery, 2014.

[6] **Cole, M.,** *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, 1991.

[7] **Fauszt, Zs.,** *Konkurens Erlang programok vizsgálata és transzformálása* – Analysis and transformation of concurrent Erlang programs, Master's thesis, Eötvös Loránd University, 2018.

[8] **Francalanza, A. and E. Tanti,** Towards Sound Refactoring in Erlang, *Xjenza Online - Journal of The Malta Chamber of Scientists*, **3** (2015), 31–35.

[9] **Francesco, C. and S. Thompson,** *ERLANG Programming*, O'Reilly Media, Inc., 1st edition, 2009.

[10] **Fuggetta, A., G. Picco and G. Vigna,** Understanding Code Mobility, *Software Engineering, IEEE Transactions*, **24** (1998), 342–361.

[11] **Horpácsi, D., J. Kőszegi and S. Thompson,** Towards Trustworthy Refactoring in Erlang, *Electronic Proceedings in Theoretical Computer Science*, **216** (2016), 83–103.

[12] **Li, H. and S. Thompson,** Safe Concurrency Introduction through Slicing, in *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, page 103–113, Association for Computing Machinery, 2015.

[13] **Li, H. and S. Thompson,** Tool Support for Refactoring Functional Programs, in *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, page 199–203, Association for Computing Machinery, 2008.

[14] **Li, H., S. Thompson, Gy. Orosz, and M. Tóth,** Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse, In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, page 61–72, Association for Computing Machinery, 2008.

[15] **Piotrowski M. and W. Turek,** Software Agents Mobility Using Process Migration Mechanism in Distributed Erlang, in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, page 43—50, Association for Computing Machinery, 2013.

[16] **Sagonas, K. and T. Avgerinos,** Automatic Refactoring of Erlang Programs, in *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, page 13–24, Association for Computing Machinery, 2009.

[17] **Tanenbaum, A.S. and M. Steen,** *Distributed Systems: Principles and Paradigms*, Prentice-Hall, Inc., USA, (2nd Edition), 2006.

[18] **Tóth, M. and I. Bozó,** Static Analysis of Complex Software Systems Implemented in Erlang, *Central European Functional Programming Summer School – Fourth Summer School, CEFP 2011, Revisited Selected Lectures*, Lecture Notes in Computer Science (LNCS), **7241** (2012), 451–514.

[19] **Varga, B.,** *Refactoring concurrent Erlang applications for distribution*, Thesis at the Student Association Conference, Faculty of Informatics. Received 1st prize., 2020

[20] **Varga, B.,** *Elosztott programozást segítő programtranszformációk megvalósítása* – Implementing program transformations to help distributed programming, Bachelor's Thesis, Eötvös Loránd University, 2020.

[21] **Weiser, M.,** Program Slicing, in *Proceedings of the 5th International Conference on Software Engineering*, page 439–449. IEEE Press, 1981.

I. Bozó, M. Tóth and B. Varga
ELTE, Eötvös Loránd University
Budapest
Hungary
bozo_i@inf.elte.hu
toth_mi@inf.elte.hu
balazsvarga@student.elte.hu