Acta Cybernetica **26** (2024) 405–429.

Identifying Client-Server Behaviours in Legacy Erlang Systems*

Zsófia Erdei[ab], Melinda Tóth[ac], and István Bozó[ad]

Abstract

In Erlang, behaviours are special forms of design patterns. There are many benefits to using behaviours. For example, behaviours can help abstract away the most common parts when solving similar problems. Design pattern recognition may help understand the source code of the software. It can provide structured information about the purpose of specific parts and the design decisions behind the implementation. For object-oriented languages, several tools exist that use different approaches and methods to identify design patterns. We present a method for identifying source code fragments in legacy Erlang systems amenable to transforming into client-server Erlang design patterns. In our analysis, we identify the base set of server candidates using concurrent process analysis and narrow down the result using further static analysis knowledge using the RefactorErl framework.

Keywords: Erlang, design patterns, client-server behaviour, concurrent behaviours, static analysis

1 Introduction

Design patterns are developed best practices that provide general, reusable solutions to common problems. There are several benefits to using design patterns. Their use promotes transparent and easy-to-maintain code, reduces the possibility of errors, and speeds up the development process. Design patterns are not specific to any programming language, they are general solutions that can be implemented in many programming languages. However, most design patterns are designed for

*Supported by EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies – The Project is supported by the Hungarian Government and co-financed by the European Social Fund. Application Domain Specific Highly Reliable IT Solutions project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

[a]Eötvös Loránd University, Budapest, Hungary
[b]E-mail: zsanart@inf.elte.hu, ORCID: 0000-0002-5089-4984
[c]E-mail: toth_m@inf.elte.hu, ORCID: 0000-0001-6300-7945
[d]E-mail: bozo_i@inf.elte.hu, ORCID: 0000-0001-5145-9688

DOI: 10.14232/actacyb.299529

object-oriented environments. In object-oriented programming, a program design pattern typically depicts the relationships between objects, and documents the inheritance, association, and aggregation relationships in design.

Design patterns can be broadly divided into three categories [5]. Structural patterns are used to define relationships between classes, creation patterns are used to represent the instantiation process, and behavioural patterns are used to describe communication and interaction between objects. Recognising design patterns cannot only help to understand the source code of software but also provide information about the purpose of specific parts of the system and the design decisions behind the implementation. Manually searching for design patterns in larger software is an extremely cumbersome and time-consuming task. For this reason, a large number of methodologies, approaches and tools have been proposed for detecting design patterns and accordingly transforming the source code [21, 9, 22].

In the case of distributed software, supporting the understanding and transformation of source code with software tools is an even more critical task. Erlang [6] is a programming language specifically designed to build fault-tolerant, distributed systems that can contain a large number of concurrent processes. In software written in Erlang, many processes can have similar structures and behaviours. The formalization of these patterns is called behaviour. Using behaviours makes it easier to read, understand and maintain legacy codes. Improvised programming structures, while possibly more efficient, are always more difficult to understand. In the Erlang/OTP libraries, there are several common concurrent design patterns implemented. The programmer only needs to implement a callback module to define the specific behaviour. Using behaviours also enables/makes it possible to use verification tools and techniques developed for Erlang [4]. A simple example of the pre-implemented design patterns included in the Erlang/OTP is the implementation of a client-server behaviour, the so-called *gen_server* behaviour. The client-server model consists of a central server process and an arbitrary number of clients. Its most common application is resource management, where multiple clients share a common resource. The server is then responsible for managing this resource.

Identifying design patterns manually is not efficient. Therefore, various approaches have been developed to automatise this process. Static analysis-based methods are widely used in this domain. The goal of our work is to identify source code fragments in legacy Erlang systems that are amenable to transforming into concurrent Erlang behaviours. This paper presents our first result in order to achieve this, namely to analyse and to identify the client-server behaviour candidates. We base our work on the static source code analysis and transformation tool RefactorErl.

RefactorErl [19] is a static analyser and transformer tool for Erlang. The tool uses static code analysis techniques and provides a wide range of features, like data-flow analysis, dynamic function call detection, side-effect analysis, a user-level query language to gather semantic information or structural complexity metrics about Erlang programs, dependency examination among functions or modules, function call graph with information about dynamic calls, etc.

We propose a two-staged behaviour recognition analysis. At first, we use the communication graph of RefactorErl [20] to find process candidates based on the communication pattern. In the second stage, we filter those elements by a predefined rule set to check the internal structure of the process.

The paper is structured as follows. In Section 2 we discuss related works, where we present multiple methods designed to recognise design patterns in object-oriented programming languages. In Section 3 we first introduce Erlang, its advantages and some of the features of the language that can be used to develop highly scalable soft real-time systems. We describe the characteristics of server processes and present the basic architecture of the client-server behaviour. In Section 4 with the use of an example, we present the structure and operation of an Erlang server process and how it has similar properties to the *gen_server* behaviour included in the Erlang/OTP library. We also show a few examples of Erlang processes that have similar characteristics to server processes but could not be implemented with *gen_server* behaviour. In Section 5 we introduce the RefactorErl tool, and in Section 6 we present a method to identify the client-server behaviour in Erlang programs based on static analysis. Finally, Section 7 presents the first results of the prototype implementation and Section 8 summarises our results.

2 Related work

Different approaches can be used to identify design patterns, both in terms of the method of identification (searching for the components of samples or recognising the full structure of a design pattern) and the type of analysis (static or dynamic analysis). Methods based on static analysis are based on the analysis of the source code, while dynamic analysis collects information while the software is running. Information obtained only from the static or dynamic analysis is seldom sufficient to effectively recognize most design patterns, so many methods work with a hybrid solution. Recognition methods can be broadly classified into several categories: those that rely on database queries, those that use metrics, those that utilize graph or matrix representations, those that are based on the Unified Modeling Language (UML), and those that combine multiple of these techniques [1].

The paper [10] proposes a solution using metrics and a machine learning algorithm for recognising micro-architectures similar to design patterns in the architecture. This can be used to better understand the design problems solved by software developers when designing the program architecture. Fingerprints are sets of metric values characterising classes playing a given role. These fingerprints are based on a set of external attributes that help categorise classes and can be used to reduce the search space of micro-architectures similar to design motifs. The fingerprints were created based on a rule-learner algorithm that inferred rules characterising design motifs' roles with the metric values of the classes playing these roles. The identification process described in the paper consisted of two steps: identifying candidate classes for design patterns by eliminating classes that did not match the expected fingerprint and identifying candidate classes for the remaining roles starting from

key-role candidates and using structural matching.

A method based on both static and dynamic analysis was presented in [12] for automatic design pattern recognition in Java. They use static analysis to compute the potential program parts playing a certain role in a design pattern and dynamic analysis to further examine those candidates. The static analysis reads the source code and constructs an attributed AST, then computes the pattern relation on the AST nodes and provides a result as a set of candidates consisting of tuples of AST nodes. The dynamic analysis takes this set as an input and monitors the execution of the nodes. Depending on the node's unique role, dynamic test actions are executed on the object sets of the candidates. In the paper, different approaches for detecting the Observer, Composite, Mediator, Chain of Responsibility and Visitor Patterns are discussed.

Another hybrid method for recognising design patterns is presented in [2]. They developed the software prototype JADEPT (JAva DEsign Pattern deTector) for design pattern recognition based on a predefined set of rules describing properties that may be either structural or behavioural and may define relationships between classes or families of classes. Weights have been associated with rules indicating how much a rule is able to describe a specific property of a given design pattern. JADEPT collects structural and behavioural information through dynamic analysis of Java software by exploiting JPDA (Java Platform Debugger Architecture) and stores the extracted information in a database. A rule is implemented by one or more queries and the existence of a design pattern can be verified through the validation of its associated rules.

Li and Thompson's paper [15] presents a technique for detecting and eliminating similar code in Erlang programs. The technique involves analysing the abstract syntax trees (ASTs) of the source code, computing a similarity measure between the ASTs, and then merging the similar code into a single function.

The authors argue that similar code is a common problem in Erlang programs, and that it can lead to maintenance and readability issues. They propose a solution that involves using a combination of structural and lexical analysis to identify similar code, and then using a technique called "code merging" to eliminate the duplication.

Duplicated code detection and design pattern recognition are two related but distinct techniques used in software development. Duplicate code is code that is identical or very similar to other code in the system mostly caused by copy-pasting and reusing already existing code. While two code snippets that implement the same design pattern can be very different even in the AST, in the case of duplicated code we usually expect them to closely match. While the examination of AST alone is not sufficient to detect design patterns, the recognition of certain similarities could help to filter out results.

The structure of parallel computations in a program can be defined conveniently, and at a high level of abstraction, using parallel design patterns. Algorithmic skeletons [7] implement common patterns of parallelism, allowing the programmer to instantiate parallel skeletons with application-specific code fragments. PaRTE [18] integrates capabilities of the RefactorErl and Wrangler [16] refactoring/program

analysis tools into a parallelisation framework that can be used to identify parallel patterns and determine the best implementations of those patterns.

Some well-known patterns are pipe (parallel pipeline) and task farm (applies a given function to a sequence of independent inputs in parallel), the map-reduce and the divide-and-conquer patterns. Skel [17] is a library of algorithmic skeletons for Erlang, providing a small number of useful, classical skeletons. Since both pipe and farm can be defined to operate on lists of inputs, the analysis described in the paper focuses on identifying certain operations on lists, and also on identifying those data structures that can be transformed to lists.

The tool developed by our research group targets three constructs: list comprehensions, library calls and recursive functions. List comprehensions are categorised based on the output expression as a possible farm or pipe candidate. Calls to functions that exhibit a map-like or pipeline-like behaviour over a sequence of data and certain map-like and pipeline-like recursive functions can also be transformed into farms or pipes. These patterns can be identified based on the syntax of the code but not all code fragments that match a pattern can be safely executed in parallel. In order to guarantee that the transformations preserve the semantics of the program further semantic analysis is required.

The transformation process itself comprises two distinct phases, an initial program shaping phase and the actual transformation into instances of skeletons. The role of program shaping is to prepare the source code for the introduction of parallel skeletons since it is necessary to shape the code into an appropriate canonical form before the transformations can be applied.

The use of PaRTE is demonstrated on a simple worked example, showing how the tool can be used to transform a sequentially implemented image merge to a parallel version and automatically obtain significant and scalable speedups over the original version.

3 Modelling client-server behaviour in Erlang

Our research focuses on legacy Erlang systems. In this section, we start with introducing Erlang. Then we specify the properties of a general server process structure implemented in Erlang, and extend these properties to express the client-server behaviour.

3.1 Erlang

Erlang [6] is a general-purpose, dynamically typed, concurrent functional programming language, which enables developers to write highly scalable, soft real-time systems. Erlang was originally designed for developing telecommunication software, since then, it is also widely used in the world of banking, chat services, and database management systems. Due to its robustness and fault tolerance, it is suitable for the development of large-scale distributed systems.

One of the main advantages of using Erlang instead of other functional languages is Erlang's ability to handle concurrency and distributed programming. In Erlang, the unit of concurrency is the process. A process is a lightweight task that runs concurrently and is independent of the other processes. Processes do not share memory, the only way for processes to interact with each other is through message passing, where the message can be any Erlang term. Message passing is asynchronous, so the process can continue processing once a message is sent. Each process has its own input queue for messages it receives. New messages received are put at the end of the queue. Received messages can be processed selectively, it is not necessary to handle messages in the order of arrival. When a process executes a receive, the first message in the queue is matched against the first pattern in the receive. If it does not match, the next branch of the receive is matched. It is repeated until the first match is found. If none of the patterns matches, the next message from the message queue is examined. Once a match is found, the message is removed from the queue and the actions corresponding to the pattern are executed. If none of the messages match, the process is blocked until a matching message arrives.

Every process in Erlang is identified by a unique process identifier (*PID*). The Erlang *BIF* (Built-In Function) *spawn* is used to create a new process:

spawn(Module, Exported_Function, List_of_Arguments).

This function creates a new process executing the function *Exported_Function* of the module *Module* with the given list of arguments.

Processes can be registered with a given name using the built-in function call *register(Name, Pid).* After registration, the process can be addressed, either with its PID or with its registered name. The process can be unregistered with the built-in function *unregister(Name).* The registered process is automatically unregistered when the process terminates.

3.2 Server processes

When implementing client-server behaviour, clients and servers are represented as Erlang processes. Processes – including server processes – have common characteristics and follow similar patterns. As a result of this, they share a similar code base. Regardless of their function, processes must first be spawned and possibly initialised. A process can be addressed by using its PID, but a server process is also usually registered. After that, we have to initialise the state of the process. The state is specific to the function of our process. This step handles all initialisation of data required for the main loop function to work well. Once the process has been initialised, it is ready to communicate with other processes. The main loop is a tail-recursive function that handles the events, it usually has a receive block and is used to process messages and send replies. A special message can be used to terminate the process when needed. In the case of a normal termination when necessary a cleanup procedure is completed.

Figures 1 and 2 show a general server process skeleton – a general, reusable structure for implementing a process in a concurrent or distributed system. It

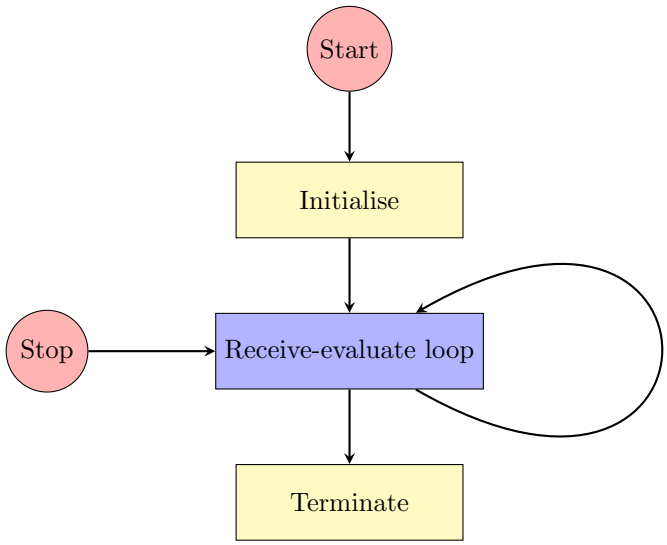

Figure 1: A process skeleton

```
start(Args) ->
    register(server, spawn_link(?MODULE, init, [Args])).

stop() -> server ! stop.

init(Args)->
    InitState = initialise_state(Args),
    loop(InitState).

loop(State)->
    receive
        stop -> terminate(State);
        {handle, Msg} ->
            NewState = handle_req(Msg, State),
            loop(NewState)
    end.
```

Figure 2: A server process skeleton source

typically includes the basic components and structure that are common to many types of processes, such as initialisation, message handling, and termination. A general process skeleton typically includes the following components:

- **Initialisation**: This is the first step in the process, where the process is set up and initialised. This may include allocating resources, setting up data structures, and starting any other necessary sub-processes.
- **Message handling**: This is the core component of the process, where the process waits for and handles incoming messages. This may include performing calculations, updating data structures, and sending messages to other processes.
- **Termination**: This is the final step in the process, where the process is cleaned up and resources are deallocated. This may include stopping sub-processes and closing any open files or connections.

While the general process skeleton fits server processes of the client-server behaviour very well, the communication pattern of this behaviour is unique (see Figure 3). In the next section, we describe the main characteristics of the client-server process architecture and its implementation in Erlang.

3.3 Client-server behaviour

The client-server model describes a way of distributing tasks and services within a network. It is characterised by a central server and an arbitrary number of clients. The client-server model is often used for resource management operations, where several different clients share a resource. Clients implemented as Erlang processes use these resources by sending the server requests.

Figure 3 shows the typical process architecture and communication of a client-server model. Most often there are multiple instances of the client and a single server. The server process receives requests, handles them, and can respond with an acknowledgement and a return value if the request was successful, or with an error if the request did not succeed. Interaction between them takes place through message sending and receiving. If a client using the service or resource handled by the server expects a reply to the request, the call to the server has to be synchronous. If the client does not need a reply, the call to the server can be asynchronous.

The *gen_server* behaviour module provides the server of a client-server relation. A generic server process (*gen_server*) implemented with this behaviour has a standard set of interface functions and includes functionality for tracing and error reporting. The process can be divided into a generic part (a behaviour module) and a specific part (a callback module). The behaviour module contains all the generic functionality reused from one implementation to another. The specific parts of the process implemented by the user are located in the callback module exporting a predefined set of functions.

4 Motivating example

In Figure 4 an example server implementation is shown. In Figure 5 an equivalent server implementation is presented using the generic server library of Erlang/OTP.

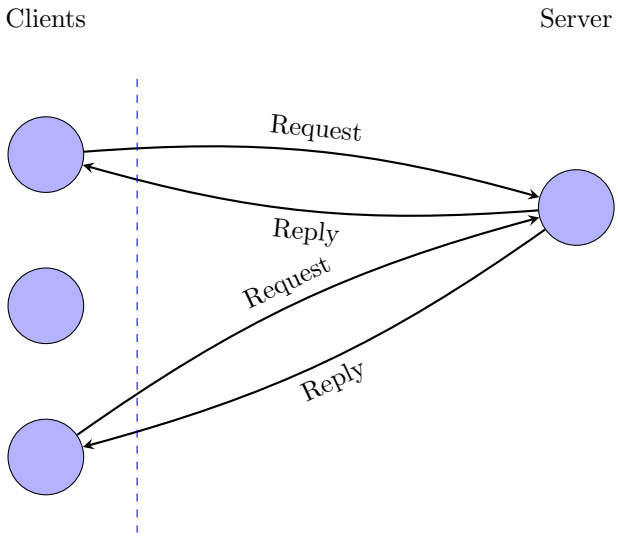

Figure 3: The client-server process architecture

The goal of our work is to identify codes similar to the code fragment shown in Figure 4 as a candidate and later transform them into an application of a client-server design pattern, as presented in the example in Figure 5.

In Figure 4 a simple job server is demonstrated that waits for {*job, ReqId,* {*M, F, A*}} messages where the third element of the tuple represent a function by the name of the implementing module, the name of the function and the list of arguments of the function call to be evaluated. The server spawns a new worker process to evaluate the function. The worker sends the calculated result to the server as a message tagged with the *result* atom. The server adds the result to its state when a *result* message arrives. Once a *reply* message arrives from a client, the server searches the result in its state and sends the value to the client. If the result is not ready yet, the server sends a *pending* answer. The *stop* message terminates the server.

As the simple server implementation example shows, the server module provides interface functions for starting (*start/0*) and stopping (*stop/0*) the server process. The *start* function spawns and registers the process with the name *jobsrv*. The server implements the function *init/0* to initialise the server process with a state and the iterating function (*loop/1*) that receives messages and performs the requested tasks. Creating a process that calls the *init/1* function can be generic. The arguments passed to the call and the implementation of the function that initialises the main loop are specific to the task. The function *loop(State)* stores the connected clients in the server state variable (*State*) and handles incoming messages

```
-module(server).
-export([start/0, stop/0]).

start() -> register(jobsrv, spawn(fun init/0)).

stop() ->
  jobsrv ! stop.

init() ->
  loop(#{}).

loop(State) ->
  receive
    stop ->
      io:format("Server stopped in state: ~p~n", State);
    {job, ReqId, {M, F, A}} ->
      spawn(fun() ->
              jobsrv ! {result, ReqId, apply(M, F, A)}
            end),
      loop(State);
    {reply, ReqId, To} ->
      case State of
        #{ReqId:=Data} -> To ! {final, ReqId, Data};
                     _ -> To ! {pending, ReqId}
      end,
      loop(State);
    {result, ReqId, Result} ->
      loop(State#{ReqId => Result})
  end.
```

Figure 4: Non gen-server implementation

in the receive block. All incoming messages are matched against the patterns in the receive and the corresponding branch is executed. Storing the loop data in between calls is the same from one process to another, but the loop data itself will be different depending on the function of the process. Sending requests to the server process will also be generic, but the types and contents of the messages and how they are handled will differ depending on the task. While each response is specific, the method of sending it back to the client process is handled in a generic way. When a *stop* message is received, the process calls the terminate function, which is responsible for a clean termination. While sending a stop messageis generic, the steps to clean up the state prior to termination will be specific.

The previously described generic parts of the server could be separated and reused for many different applications, and only the specific parts of the code would

have to be reimplemented. The Erlang/OTP module solves this with the *gen_server* library that formalizes the general server behaviour of a client-server model. The *gen_server* module contains the generic part of server implementation and the user only has to implement a callback module to define the specific behaviour.

```
-module(gserver).
-export([start/0, stop/0, init/1,
         handle_call/3, handle_cast/2, handle_info/2]).
-behaviour(gen_server).

start() ->
  gen_server:start({local, jobsrv}, gserver, [], []).

stop() ->
  gen_server:cast({local, jobsrv}, stop).

init(_) ->
  {ok, #{}}.

handle_cast({job, ReqId, {M, F, A}}, State) ->
  spawn(fun() ->
          jobsrv ! {result, ReqId, apply(M, F, A)}
        end),
  {noreply, State};
handle_cast(stop, State) ->
  io:format("Server stopped in state: ~p~n", State),
  {noreply, stop, State}.

handle_info({result, ReqId, Result}, State) ->
  {noreply, State#{ReqId => Result}}.

handle_call({reply, ReqId}, _, State) ->
  case State of
    #{ReqId:=Data} -> {reply, {final, ReqId, Data}, State};
    _ -> {reply, {pending, ReqId}, State}
  end.
```

Figure 5: Gen-server implementation

With the *gen_server* behaviour (Figure 5), instead of using the *spawn* and *spawn_link* BIFs, the *gen_server:start/4* and *gen_server:start_link/4* functions are used. In the example shown here, the server can be started by calling the *gserver:start()* function call, which then calls the *gen_server:start_link/4* function. This function spawns and links to the created server process. The first argument of the function specifies the name in the form of a tuple, in this case, {*local,jobsrv*}.

The server is then locally registered as jobsrv[1]. The second argument is the name of the callback module, which is the module where the callback functions are located. The third argument is a list of arguments that are passed to the *init/1* callback function when the process is started. Usually, lists are used to pass multiple arguments. These arguments can be used to initialise the process's state or to configure its behavior. Here, *init* does not need any data and ignores the argument. The fourth argument is a list of options, for example, it enables the user to set memory management flags as well as tracing and debugging flags. Most behaviour implementations, like in the example, just pass the empty list as an argument.

The *gen_server* start functions will spawn a new process that calls the *init/1* callback function from the callback module, with the arguments supplied. The task of the *init* function in the *gserver* module (Figure 5) is same as it was in the *server* module (Figure 4): to initialise the state of the server. Synchronous communication can be initialised by calling the *gen_server:call/2* function that sends a message to the server, while *gen_server:cast/2* calls are responsible for asynchronous communication. When a client sends a message to the server process by calling these functions, the *handle_call/3* or *handle_cast/2* callback function is called. The *handle_call/3* function is used to handle synchronous requests, where the client expects a reply. The *handle_cast/2* function is used to handle asynchronous requests, where the client does not expect a reply. Stopping the server can be handled synchronously or asynchronously by calling *gen_server:call/2* or *gen_server:cast/2*. Messages that are not sent to the server through *gen_server:call/2* or *gen_server:cast/2* function calls can be handled in the *handle_info/2* function definitions.

In our example, *job* messages and the stopping are asynchronous request, *reply* messages are synchronous, and the *result* messages sent from the worker processes are handled by the *handle_info* definition.

4.1 Other server-like processes

In Erlang, a process can be identified with its evaluating function. The process is created when we spawn the function and the process is alive until the evaluation of its function is finished. Long-living processes usually evaluate recursive functions. Thus when we are identifying server processes we need to identify recursive function definitions. However, we do not want to consider all recursive definitions which were spawned in the program as server processes. In this subsection, we will introduce a few counterexamples.

4.1.1 Taskfarm

Let us consider the code skeleton on Figure 6 that implements a parallel taskfarm in Erlang: where we want to evaluate a function on the elements of a list. We start a dispatcher, a collector process and some worker processes to evaluate the function depending on the number of available resources. The dispatcher and the

[1]The first argument can be omitted, so the server might not be registered. In this case, the process id of the newly created process could be used to refer to the server.

collector are registered processes. The dispatcher waits for *free* messages from the workers and sends an element of the list back. The collector waits for results from the workers and stores those in its state. It also notifies about the collected data on request. Workers are notifying the dispatcher if they are free to work and send the result of the computation to the collector process.

```
run(F, L) ->
    register(disp, spawn(fun() -> dispatcher(L) end)),
    register(coll, spawn(fun() -> collector([]) end)),
    N = erlang:system_info(logical_processors_available),
    [spawn(fun() -> worker(F) end) || _ <- lists:seq(1, N)].

dispatcher([H|T]) ->
    receive
        {free, Worker} -> Worker ! {data, H}, dispatcher(T)
    end;
dispatcher([]) ->
    receive
        stop -> terminate
    end.

collector(Acc) ->
    receive
        {result, Result} -> collector([Result | Acc]);
        {give_me, From} -> From ! Acc, collector(Acc)
    end.

worker(F) ->
    disp ! {free, self()},
    receive
        {data, Data} -> coll ! {result, F(Data)}, worker(F)
    end.
```

Figure 6: Parallel taskfarm implementation

The most important characteristic of a server process is a containing receive expression to handle messages from client processes and answering to them. If we consider only this condition, we might say that all the processes in this example could be server processes. However, we do not want to consider worker processes as server processes in client-server behaviour. We might have the expectation that a server process is unique in the system, it has some special role. Thus when we create multiple instances of an actor we do not want to consider those as servers. We can delete some processes from our server candidate list based on the context of the initialisation. The worker processes are spawned in a list comprehension, therefore we can assume that multiple occurrences exist, and thus we will not consider them.

The dispatcher process could be considered as a server where the clients are the worker processes. Later we might prioritise our candidate list and put functions like dispatcher at the end if we want our servers to do more work.

4.1.2 Timeout looping

Server processes have to be tail-recursive. However, we do not want to consider all of them. The code snippet on Figure 7 shows a process definition which waits for cancelling messages and terminates. Otherwise, it waits for T seconds and recursively calls itself if there is no more event to handle. We do not want to consider the recursion in the after branch of the receive expression as a proper server behaviour. It would not fit the client-server behaviour, thus we cannot transform it.

```
loop(S = #state{server=Server, to_go=[T|Next]}) ->
    receive
        {Server, Ref, cancel} -> Server ! {Ref, ok}
    after T*1000 ->
        if Next =:= [] -> Server ! {done, S#state.name};
           Next =/= [] -> loop(S#state{to_go=Next})
        end
    end.
```

Figure 7: An event handler [14]

4.1.3 Multiple recursive calls

Figure 8 contains a parallel implementation of the Fibonacci number calculation based on a caching optimisation to store the already calculated values. The cache process has no termination branch, it is an infinite recursive definition. However, we might consider it a server process. On the other hand, the process evaluating the fib function could not be considered a server process. It has multiple recursive calls in its body, thus it would not be possible to transform it into a *gen_server* behaviour-based process.

5 RefactorErl

The RefactorErl tool uses a directed, rooted graph, with typed nodes and edges as an internal representation to store the source code. The graph is called Semantic Program Graph (SPG) [13]. The SPG stores lexical, syntactic and semantic information about the source code, calculated by various static semantic analysers. Every module, function, and expression is a node with a unique identifier and a set of properties. Figure 9 shows a small part of a generated SPG. Besides the

```
fib(0) -> 1;
fib(1) -> 1;
fib(N) when is_integer(N), N > 1 ->
    cache ! {fib, N, self()},
    receive
        {value, N, FibN} -> FibN;
        no_value ->
            Fib1 = fib(N-1),
            Fib2 = fib(N-2),
            Fib = Fib1+Fib2,
            cache ! {store, N, Fib},
            Fib
    end.

start_cache() ->
    register(cache, spawn(fun() -> cache(#{}) end)).

cache(State) ->
    receive
        {fib, N, From} ->
            case State of
                #{N := Fib} -> From ! {value, N, Fib}, cache(State);
                _ -> From ! no_value, cache(State)
            end;
        {store, N, Fib} -> cache(State#{N=>Fib})
    end.
```

Figure 8: Fibonacci calculation

syntactic (black) nodes and edges, various semantic (coloured) and lexical (blue) information are presented there.

Based on the initial static analyser framework provided by RefactorErl, various complex static analysers have been implemented. For example, the tool provides control flow, control dependence, data-flow, data-dependence analysis, dynamic function reference analysis, concurrent message flow analysis, etc [19].

RefactorErl supports the analysis of concurrent programs as well. It is able to identify the spawned processes and the communication between them based on data-flow analysis and expression value calculation. RefactorErl builds a communication graph as a result of the analysis [20]. The nodes of the graph represent the processes in the system. The edges represent various forms of communication between the processes, for example, process creation, process name registration, message passing, ETS table creation and reading from or writing into an ETS table.

ETS (Erlang Term Storage) [8] tables are a built-in feature of the Erlang/OTP

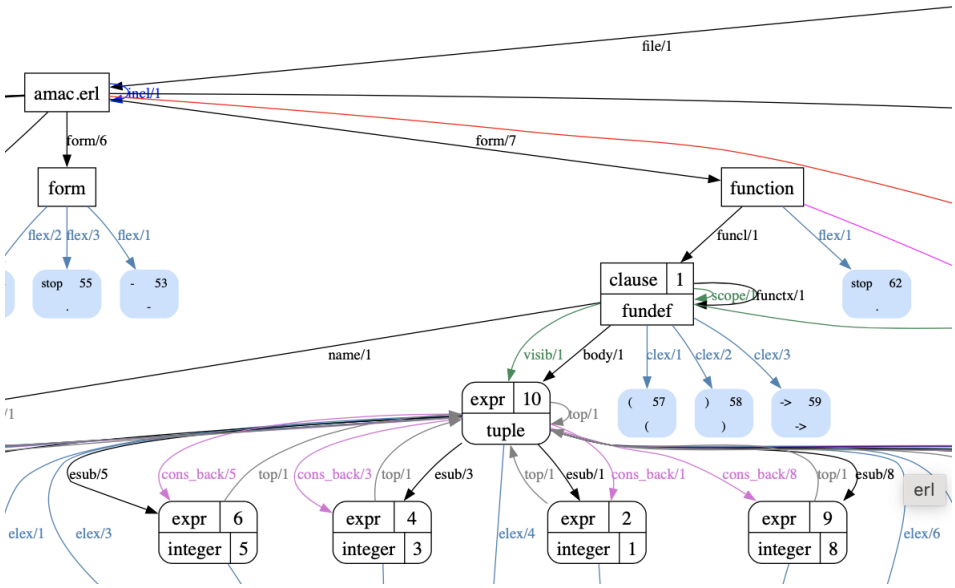

Figure 9: Part of a Semantic Program Graph

system that allows for fast and efficient storage and retrieval of data among multiple processes. They are similar to hash tables or key-value stores, but they are implemented in the Erlang virtual machine and are designed to work well with the Erlang concurrency model. Since one process can put some data into the table that others can read, thus it can be considered as a special form of process communication.

The root of the communication graph is a 'super process' (SP) node which represents the runtime environment. It represents the fact that the communicating functions can be called from the currently running process, for example from the Erlang shell.

Figure 10 shows an example communication graph [20]. These graphs can be useful when we want to find client/server communication in the code as the first step in our analysis. This helps us to find potential candidates and narrow the scope of the analysis. Figure 11 contains an even more detailed communication graph with hidden communication through ETS tables [20].

6 Identifying server processes

In this section, we present our approach to detecting one of the Erlang design patterns, the generic server process. The proposed method is built on the capabilities of the RefactorErl tool introduced in the last section. The method can be divided

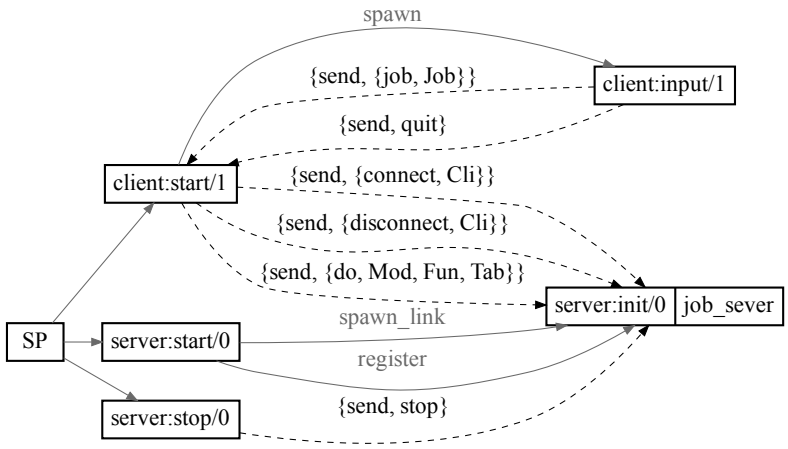

Figure 10: Communication graph [20]

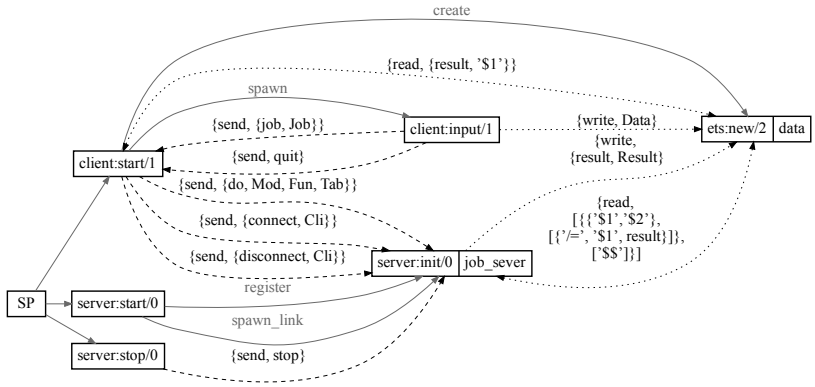

Figure 11: Communication graph with hidden data sharing [20]

into two major steps. We use initial filtering based on the communication graph to reduce the search space and eliminate processes not matching the client-server behaviour. To identify possible candidates, we use data-flow analysis to examine relations between processes. After the initial filtering, we use the Semantic Program Graph to identify functions that match the structure of the generic server design pattern. To do this we have developed a set of rules that the previously determined candidates must comply with.

6.1 Detecting candidates

For the first part of the method, the communication graph is used to find the possible candidate processes. First, we are looking for processes that start and possibly register a process. This can be achieved by examining the communication graph and filtering nodes that are connected with an edge that signifies process creation. The process nodes of the communication graph and the data associated with them are stored in the "processes" ETS table.

We can use the *match_object(Table, Pattern)* function from the *ets* module to find edges that represent process creation, from this we can determine our set of candidate functions.

After the initial filtering of the candidates, we use a set of rules to identify the ones that match the patterns we are looking for. Since the communication graph does not provide enough information for this we use the SPG built by the RefactorErl tool. The information we need to check if a candidate satisfies the rules can be gathered efficiently from the SPG using the query language RefactorErl provides.

6.2 Filtering the initial candidates

To effectively recognize server processes, we need to examine their structure. The Figure 1 shows an example skeleton for a general server process [6]. After the process has been spawned and possibly registered, it initialises the process loop data. The loop data is often the result of arguments passed to the spawn function. The receive-evaluate function receives messages and handles them, updates the state, and passes it back as an argument to a tail-recursive call. If one of the messages it handles is a stop message, the receiving process will clean up after itself and terminate.

Based on this general behaviour we determined a set of rules that candidates must comply with to be considered to be equivalent to the *gen_server* behaviour. These criteria mostly apply to the structure of the processes and can be checked based on the syntactic and semantic information contained in the SPG.

We consider to server processes those functions that satisfy the following criteria:

- the spawned function must be tail-recursive;
- it must contain a receive block;
- one of the branches of the function must be non-recursive to ensure the process can terminate;
- the receive block might contain a reply (synchronous), or no reply (asynchronous) branch;
- the recursive call cannot be in an after block;
- the spawned process is registered; and

- the process is unique.

The first rule comes from a common behaviour of processes, which is typical not only for server processes but also for many other types. Since processes can handle thousands of messages per second over sustained periods of time, using tail-recursion, where the very last thing the function does is to call itself, we can ensure that it executes in constant memory space without increasing the recursive call stack every time a message is handled. To determine whether the candidate complies with this rule, we have to examine the spawned function. It either has to be a tail-recursive function, or the last expression has to be a function call such that the called function itself complies with this rule. Using the SPG and following the function calls we must ultimately check if the last function on the chain is a tail-recursive function. If it is not, we can rule it out from our set of candidates.

If we have established that our function is tail-recursive, the second rule can also be checked in the same step. For this, we only need to examine whether the last expression of the path we followed is a receive block or not. This rule differs from the first one in that it is not a strict rule. We can implement a process complying with the generic server pattern such that it does not have receive block, but it would be incredibly uncommon in practice.

Often one of the messages handled by the server process is a ‘stop‘ message that when received the process will clean up after itself and terminate. For this behaviour, the spawned function must have a non-recursive branch in the receive block. This is also not a strict rule, but using this, we can rank the results according to how well they match the general behaviour. Usually, the function of a server is to receive requests, handle them, and respond with some appropriate message. In order to do this, the receive block must contain replies. Similar to the second rule a process complying with the *gen_server* behaviour can be implemented without this rule being met, but it would not be general use of a server. This rule can be checked by examining the receive block found during the checking of the first rule.

When a process is spawned, it can be registered with a name. After registration, the process can be addressed with its registered name or if registration was omitted with its PID. Not every process needs to be registered, as we have shown previously in the task farm example 4.1.1 where only the dispatcher and collector were registered processes but the workers were not. In contrast to this server processes are almost always registered, so checking if a given process is registered can help us filter the candidates.

Server processes have a special role in the system, they communicate with multiple clients receiving, processing and handling multiple requests. Some server-like processes exist that comply with most of our structural requirements but still cannot (or should not) be considered server processes. A good example of such processes is the worker processes of the task farm. When we create multiple instances of an actor we do not want to consider those as servers. For this reason, it is worth examining if there are multiple instances spawned from a given candidate process.

A process being tail-recursive in itself is not enough for it to be a valid server process. There can be a special case where the tail-recursive call of the spawned

function is in an after block. We have to filter out such cases because they would not fit the client-server behaviour. An example of such a process is shown in Section 4.1.2.

It can be the case that a process has multiple recursive calls in its body. Since types of processes would be not transformable to a *gen_server* behaviour-based process, so they cannot be considered server processes despite being structurally similar. Such an example can be found in Section 4.1.3 where a parallel implementation of the Fibonacci number calculation is shown. These types of processes have to be also filtered out from our results.

To refine the set of candidates, it might also be worth examining if the receive block has branches for certain special messages that a server usually handles, for example an *'EXIT'* or *'DOWN'* messages sent to the server process when linked/monitored processes exit[2].

7 Evaluation

Our method for identifying source code fragments in legacy Erlang systems that can be transformed into client-server Erlang behaviors is based on the static source code analysis and transformation tool RefactorErl. We present examples of server-like processes we found in some analysed projects that could have been implemented using the *gen_server* behavior.

We analysed example source codes and solutions[3] to the exercises to the books *Erlang Programming* [6], *Programming Erlang: Software for a Concurrent World* [3] and *Learn You Some Erlang for Great Good* [11]. We found snippets that match the pattern of the client-server behaviour.

The first server-like process we identified is the basic server implementation example from the book Learn You Some Erlang for Great Good. The *kitty_server* is a simple Erlang application that demonstrates the use of the client-server pattern and message-passing between processes. The example is a simulation of a server that manages a collection of 'kitty' objects, which are represented as Erlang processes. Clients can interact with the *kitty_server* process by sending messages to it, such as requesting to create a new kitty, or asking for the status of a particular kitty. The *kitty_server* process then communicates with the appropriate process to fulfill the request, and sends a response back to the client. The prototype algorithm identifies the loop function shown in Figure 12 as a server-like process as it satisfies all the established criteria.

From the example codes provided to *Erlang Programming* the prototype algorithm found multiple server-like processes. In Chapter 4 of [6] there are two small examples (Figure 13) demonstrating message passing between processes (the loop function is the same in both). While it might be unnecessary because of the simplicity of the example, it would be possible to convert both to a *gen_server*

[2] https://learnyousomeerlang.com/errors-and-processes

[3] https://github.com/francescoc/erlangprogramming, https://github.com/Stratus3D/programming_erlang_exercises, https://learnyousomeerlang.com/

```
loop(Cats) ->
    receive
        {Pid, Ref, {order, Name, Color, Description}} ->
            if Cats =:= [] ->
                Pid ! {Ref, make_cat(Name, Color, Description)},
                loop(Cats);
               Cats =/= [] -> % got to empty the stock
                Pid ! {Ref, hd(Cats)},
                loop(tl(Cats))
            end;
        {return, Cat = #cat{}} ->
            loop([Cat|Cats]);
        {Pid, Ref, terminate} ->
            Pid ! {Ref, ok},
            terminate(Cats);
        Unknown ->
            %% do some logging here too
            io:format("Unknown message: ~p~n", [Unknown]),
            loop(Cats)
    end.
```

Figure 12: Main loop function of the kitty server [11]

implementation[4]. In the modules provided to Chapter 5 the algorithm identified several examples where a process could be implemented with the *gen_server* behaviour. For example in the module *frequency* a server process is responsible for managing radio frequencies on behalf of its clients, the mobile phones connected to the network. The phone requests a frequency whenever a call needs to be connected, and releases it once the call has terminated. This is an example that demonstrates the client-server design pattern and the *loop/1* function fits the criteria perfectly.

Although in the analysed simpler examples we successfully found the possible server-like processes with the help of the implemented prototype algorithm, not all rule-checks have yet been fully implemented, so we also received a few false positives. Such an example was the *loop/1* function shown in Figure 14 (Exercise 3 from Chapter 12 of [3]). The example code implements a process ring, where a number of Erlang processes are connected in a ring-like structure, with each process communicating with one neighbor in the ring. The process with ID 1 then sends a message to the process with ID 2, which in turn sends the message to the process with ID 3, and so on, until the message has been passed around the entire ring. At first this process seems like a server based on its structure and communication but it does not comply with the rule of uniqueness. Such candidates have to be eliminated in the future.

[4]In the future, we might implement a prioritisation to present the results in the order of relevance. Simple candidates might be listed at the end of the candidate list.

```
loop() ->
   receive
     {From, Msg} ->
        From ! {self(), Msg},
        loop();
     stop ->
       true
   end.
```

Figure 13: Loop function from the simple example in Chapter 4 of [6]

```
loop(NextPid) ->
    receive
        stop ->
            NextPid ! stop,
            ok;
        Value ->
            NextPid ! Value,
            loop(NextPid)
    end.
```

Figure 14: Loop function from a process ring in Chapter 12 of [3]

In Exercises 5 and 6 from Chapter 13 of [6] (Figure 15) the prototype algorithm identified the spawned *handle_crashes/1* function as a server-like process. The example code implements a supervisor process which is responsible for starting, stopping, and monitoring the other processes. The worker processes are responsible for performing specific tasks, and the supervisor process is responsible for monitoring and managing the worker processes. When a worker process crashes or exits, the supervisor process is notified and restarts the worker process. This could be potentially implemented with the *gen_server* behaviour, but there exists a separate behaviour for exactly this type of process, the *supervisor* behaviour. In this case, it would be preferable to use the latter behaviour. However, we would like to note here that the *supervisor* behaviour is implemented as a server using a *gen_server* behaviour. Thus our result is correct.

We examined the edge cases presented in Section 4.1. The prototype implementation had a false positive hit, the *worker* function (Figure 6). This is a known limitation, since the uniqueness check is not yet implemented.

8 Conclusions

Design patterns provide solutions to recurring issues in software development. For object-oriented languages, various tools exist that use different approaches and

```
handle_crashes(WorkerData) ->
  receive
      {get_workers, Pid} ->
          Pid ! {self(), workers, WorkerData},

          handle_crashes(WorkerData);
      {'DOWN', Ref, process, Pid, Why} ->

          ...

          % Recursively call this function to handle later crashes
          handle_crashes(NewWorkerData)
  end.
```

Figure 15: Function `handle_crashes` from the example in Chapter 13 of [6]

methods to identify these patterns. In Erlang, behaviours are the formalised versions of these design patterns. In this paper, we proposed a method for identifying a specific design pattern, the client-server behavior, in legacy Erlang systems.

In this paper, we proposed a method based on static analysis of Erlang programs to identify processes complying with the client-server behaviour. The method is based on the analyses provided by the RefactorErl tool and can be divided into two major steps. Initial filtering based on the communication graph is used to reduce the search space and eliminate processes not matching the server behaviour. After the initial filtering, the Semantic Program Graph, an intermediate representation of the source code built by the RefactorErl tool is used to identify functions that match the structure of the generic client-server design pattern. To achieve this, we have developed a set of rules that the previously determined candidates must comply with.

We implemented the prototype algorithm and tested it on small open-source examples. Using this prototype implementation, we identified basic server processes that could be turned to equivalent *gen_server* process. We also examined a few edge cases where the described rules might fail. In the future, we would like to analyse further open-source projects and refine the rules based on the findings.

References

[1] Al-Obeidallah, M., Petridis, M., and Kapetanakis, S. A survey on design pattern detection approaches. *International Journal of Software Engineering*, 7:41–59, 2016. URL: https://www.cscjournals.org/manuscript/Journals/IJSE/Volume7/Issue3/IJSE-163.pdf.

[2] Arcelli Fontana, F., Perin, F., Raibulet, C., and Ravani, S. Design pattern detection in Java systems: A dynamic analysis based approach. *Communications in Computer and Information Science*, 69:163–179, 2010. DOI: 10.1007/978-3-642-14819-4_12.

[3] Armstrong, J. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. DOI: 10.1017/S0956796809007163.

[4] Arts, T., Benac Earle, C., and Derrick, J. Development of a verified Erlang program for resource locking. *International Journal on Software Tools for Technology Transfer*, 5(2):205–220, 2004. DOI: 10.1007/s10009-003-0114-9.

[5] Brown, K. Design reverse-engineering and automated design-pattern detection in smalltalk. Technical report, North Carolina State University at Raleigh, USA, 1996. URL: https://repository.lib.ncsu.edu/items/ec9a80d5-c9c6-47c5-afd5-03f21a36bb63.

[6] Cesarini, F. and Thompson, S. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, Inc., 2009. URL: https://www.oreilly.com/library/view/erlang-programming/9780596803940/.

[7] Cole, M. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991. URL: https://dl.acm.org/doi/10.5555/128874.

[8] Ericsson AB. Erlang Reference Manual: `ets` module. URL: https://www.erlang.org/doc/man/ets.html.

[9] Gaitani, M., Zafeiris, V., Diamantidis, N., and Giakoumakis, E. Automated refactoring to the Null Object design pattern. *Information and Software Technology*, 59:33–52, 2015. DOI: 10.1016/j.infsof.2014.10.010.

[10] Guéhéneuc, Y.-G., Sahraoui, H., and Zaidi, F. Fingerprinting design patterns. In *11th Working Conference on Reverse Engineering*, pages 172–181. IEEE, 2004. DOI: 10.1109/WCRE.2004.21.

[11] Hebert, F. *Learn You Some Erlang for Great Good! A Beginner's Guide*. No Starch Press, USA, 2013. URL: https://learnyousomeerlang.com/.

[12] Heuzeroth, D., Holl, T., Högström, G., and Löwe, W. Automatic design pattern detection. In *11th IEEE International Workshop on Program Comprehension*, pages 94– 103, 2003. DOI: 10.1109/WPC.2003.1199193.

[13] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Nagyné Víg, A., Nagy, T., Tóth, M., and Király, R. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques*, Volume 54 Special Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, 2009.

[14] Learn you some Erlang. An event module. URL: https://learnyousomeerlang.com/designing-a-concurrent-application#an-event-module.

[15] Li, H. and Thompson, S. Similar code detection and elimination for Erlang programs. In *International Symposium on Practical Aspects of Declarative Languages: Practical Aspects of Declarative Languages*, Volume 5937 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-11503-5_10.

[16] Li, H., Thompson, S., Orosz, G., and Tóth, M. Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, page 61–72, New York, NY, USA, 2008. Association for Computing Machinery. DOI: 10.1145/1411273.1411283.

[17] skel: A streaming process-based skeleton library for Erlang, 2012. URL: https://github.com/ParaPhrase/skel.

[18] Tóth, M., Bozó, I., and Kozsik, T. Pattern candidate discovery and parallelization techniques. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, IFL 2017, New York, NY, USA, 2017. Association for Computing Machinery. DOI: 10.1145/3205368.3205369.

[19] Tóth, M. and Bozó, I. Static analysis of complex software systems implemented in Erlang. In *Central European Functional Programming School*, Volume 7241 of *Lecture Notes in Computer Science*, pages 451–514. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-32096-5_9.

[20] Tóth, M. and Bozó, I. Detecting and visualising process relationships in Erlang. *Procedia Computer Science*, 29:1525–1534, 2014. DOI: 10.1016/j.procs.2014.05.138.

[21] Yu, D., Zhang, P., Yang, J., Chen, Z., Liu, C., and Chen, J. Efficiently detecting structural design pattern instances based on ordered sequences. *Journal of Systems and Software*, 142:35–56, 2018. DOI: https://doi.org/10.1016/j.jss.2018.04.015.

[22] Zafeiris, V., Poulias, S., Diamantidis, N., and Giakoumakis, E. Automated refactoring of super-class method invocations to the Template Method design pattern. *Information and Software Technology*, 82:19–35, 2017. DOI: 10.1016/j.infsof.2016.09.008.