Transforming Erlang Server Applications

Zsófia Erdei
ELTE, Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
zsanart@inf.elte.hu
0000-0002-5089-4984

István Bozó
ELTE, Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
bozo_i@inf.elte.hu
0000-0001-5145-9688

Melinda Tóth
ELTE, Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
toth_m@inf.elte.hu
0000-0001-6300-7945

2024 IEEE 3rd Conference on Information Technology and Data Science (CITDS) | 979-8-3503-8788-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CITDS62610.2024.10791346

***Abstract*—In Erlang, behaviours serve as specialized design patterns, offering numerous advantages. For example, they allow us to abstract common components when addressing similar problems. Additionally, recognizing design patterns enhances our understanding of software source code by providing structured insights into specific parts and the underlying design decisions.**

In contrast to object-oriented languages, which have various tools for identifying design patterns, functional programming languages like Erlang have fewer readily available solutions. Our focus is on legacy Erlang systems and their transformation into more readable modern code. Previously we developed a possible method for identifying source code fragments in such systems based on static analysis. In this paper, we propose a method for transforming recognized server candidates into client-server Erlang design patterns. To achieve this, we use the RefactorErl framework to verify the feasibility of the transformation and then transform the code into a generic server process. Our approach is demonstrated through an illustrative example.

***Index Terms*—Erlang, design patterns, client-server behaviour, concurrent behaviours, static analysis, refactoring**

I. Introduction

The development of methods that aid code comprehension and debugging is a current issue, requiring the introduction of new tools and approaches. While maintainability is not a problem for smaller software projects, large software bases present challenges due to the sheer volume and complexity of source code. Without tools to assist code comprehension, it can be extremely difficult or even impossible for a programmer to navigate such extensive codebases. Design patterns are established best practices that offer reusable solutions to common problems [1]. Utilizing design patterns provides several advantages. They enhance code transparency and maintainability, reduce the likelihood of errors, and accelerate the development process. Most design patterns are specifically tailored for object-oriented environments. In object-oriented programming, a program design pattern typically illustrates the relationships between objects and documents the inheritance, association, and aggregation relationships in the design.

Supported by the ÚNKP-23-3 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. 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.

One of the most commonly used OTP behaviours is `gen_server`, which simplifies the implementation of server processes [2]. By leveraging `gen_server`, developers can create concurrent, fault-tolerant systems [3]. Even today there is a huge amount of legacy code used. These legacy codebases often suffer from readability issues due to their historical evolution, maintenance, and lack of consistent design patterns. The ability to refactor legacy code is crucial for long-term maintainability. By identifying server candidates for transformation, and then applying an automatic refactoring developers can gradually migrate the system toward a more behavior-driven architecture.

II. Background

A. Erlang and RefactorErl

Erlang is a versatile, dynamically typed, and concurrent functional programming language [2]. It enables developers to create highly scalable soft real-time systems. Initially designed for telecommunication software, Erlang has found widespread use in domains such as banking, chat services, and database management systems. Its robustness and fault tolerance make it well-suited for large-scale distributed systems.

RefactorErl is a static source code analyzer and transformer tool for Erlang [4], [5]. The primary focus of the tool is to support daily code comprehension and transformation tasks for Erlang developers. 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. RefactorErl provides various refactorings to enhance the design of Erlang programs. These refactorings include structural changes such as renaming functions, variables, and modules, as well as function extraction and generalization.

RefactorErl represents an Erlang program in the form of a graph called the Semantic Program Graph (SPG) [5]. After the source code is analysed, this graph is stored in a database. The tool is capable of transforming the graph back into source code at any time. The process of refactoring essentially consists of graph transformation steps, using the SPG to collect the necessary information for the transformation.

B. Behaviour

Design patterns can be broadly divided into three categories [6]. Structural patterns define relationships between classes, creation patterns handle object instantiation, and behavioural patterns describe communication and interaction between objects. Recognizing design patterns not only helps understand software source code but also provides insights into specific system components and design decisions. Detecting design patterns manually in large software is time-consuming, so various methodologies and tools have been proposed for automated pattern detection and code transformation [7]–[9].

In Erlang, behaviours serve as formalisations of design patterns that aid in solving common problems. These patterns help developers understand the source code of software by providing structured information about specific parts and the design decisions behind their implementation. When dealing with object-oriented languages, various tools exist to identify design patterns using different approaches and methods. In the context of Erlang, we introduce a method for identifying source code fragments in legacy systems that can be transformed into concurrent Erlang design patterns [10]. This involves analyzing concurrent processes and refining the results using static analysis knowledge via the RefactorErl framework. Behaviours play a crucial role in simplifying concurrent programming and protecting developers from common pitfalls. Using behaviours also enables/makes it possible to use verification tools and techniques developed for Erlang [11].

C. Identifying server behaviours

The client-server model is commonly used for processes responsible for a resource and services that can be applied to these resources. When clients (typically implemented as Erlang processes) send requests to the server, they gain access to these resources and services. The `gen_server` behaviour module serves as the server in this client-server relationship. A generic server process (`gen_server`) implemented using this module adheres to a standard set of interface functions and incorporates features for tracing and error reporting. The process can be conceptually divided into two parts: a generic part (defined by the behaviour module) that contains reusable functionality across different implementations, and a specific part (the callback module) where users implement custom functionality, exporting a predefined set of functions.

The original server module in our example shown in Fig. 1 is named a. It defines the function `test/0` that spawns a new process (identified with `Pid`) using the `spawn_link/3` function. The first parameter of the `spawn_link` function is the module containing the function to be spawned, the second one is the name of the function and the third is a list of parameters given to the function which in this case gives the starting state of the process. After spawning the new process we send messages to it in the form of a tuple `{add, N}` for each N in the range from 1 to 10. The `loop/1` function receives these messages and calculates the sum of the received value N and the accumulated value R. The current value is printed using `io:format/2`. Sending the message `{remove, self()}` sends the result of the accumulation back to the messaging process and sets the value of resources to zero. The process stops when it receives the `stop` message.

```
-module(a).
-export([test/0,loop/1]).
test() ->
  Pid = spawn_link(a,loop,[0]),
  [Pid ! {add, N} || N<-lists:seq(1,10)],
  Pid ! {remove, self()},
  receive
    {res, N} -> io:format("Collected: ~p~n", [N])
  end,
  Pid ! stop,
  Pid.
loop(R) ->
  receive
    {add, N} -> Res = N+R,
                io:format("Resource sum: ~p~n",[Res]),
                loop(Res);
    {remove, Pid} -> Pid ! {res, R},
                     loop(0);
    stop -> io:format("Server stopping~n")
  end.
```

Fig. 1. Original server example

```
-module(a).
-export([test/0]).
test() ->
  {ok,Pid}= gen_server:start_link(gen_callback,{0},[]),
  [gen_server:cast(Pid,{add,N}) || N<-lists:seq(1,10)],
  case gen_server:call(Pid, {remove, self()}) of
      {res, N} -> io:format("Collected: ~p~n", [N])
  end,
  gen_server:cast(Pid, stop),
  Pid.
```

Fig. 2. Generic server example

```
-module(gen_callback).
-behaviour(gen_server).
-export([init/1,terminate/2,handle_call/3,
    handle_cast/2, handle_info/2]).
init({R}) -> {ok, {R}}.
handle_cast({add, N}, {R}) ->
    Res = N + R,
    io:format("Resource sum: ~p~n", [Res]),
    {noreply, {Res}};
handle_cast(stop, {R}) ->
    io:format("Server stopping~n"),
    {stop, normal, {R}}.
handle_call({remove, Pid}, _Pid, {R}) ->
    {reply, Pid ! {res, R}, {0}}.
handle_info(_Message, State) -> {noreply, State}.
terminate(_Message, _State) -> ok.
```

Fig. 3. Callback module example

In the refactored version using OTP's generic server process (Fig. 2), the code still resides in module a. The `test/0` function now starts a `gen_server` process using `gen_server:start_link/3`. The `gen_server` process is initialized with the initial state {`0`}. Instead of sending messages directly, the `test/0` function uses `gen_server:cast/2` to send {`add, N`} messages to the generic server. The `gen_server` callback module is named `gen_callback` (Fig. 3). The `init/1` function in `gen_callback` initializes the state, in the case of this example with the provided value `R`. The `handle_call/3` and `handle_cast/2` functions are responsible for handling synchronous and asynchronous requests, respectively. The `handle_cast/2` function is used for asynchronous requests. It handles messages without sending a reply. The function signature is:

```
handle_cast(Msg, State) ->
    {noreply, NewState} | {stop, Reason, NewState}
```

- Msg: The message sent to the server.
- State: The current state of the server.
- The function can return one of the following tuples:
  - {`noreply, NewState`}: The server does not reply to the sender, only updates its state.
  - {`stop, Reason, NewState`}: The server stops with a reason (Reason) and an updated state.

The `handle_call/3` function is used for synchronous requests. When a client sends a message using `gen_server:call`, it expects a response. The function signature is:

```
handle_call(Request, From, State) ->
    {reply, Reply, NewState} | {noreply, NewState} |
      {stop, Reason, NewState}
```

- Request: The request sent by the client.
- From: The process ID (PID) of the client process that made the request.
- State: The current state of the server.
- The function can return one of the following tuples:
  - {`reply, Reply, NewState`}: The server replies to the client with `Reply`, and the state is updated to `NewState`.
  - {`noreply, NewState`}: The server does not reply to the client, only updates its state.
  - {`stop, Reason, NewState`}: The server stops with a reason (`Reason`) and an updated state.

The `handle_cast/2` function in the case of our example in `gen_callback` handles the {`add, N`} messages by calculating the sum and returning {`noreply, {Res}`}. When the `stop` message is received, the `handle_cast/2` function returns {`stop, normal, {R}`}, indicating that the process should stop.

The purpose of the `handle_info/2` function is to handle system messages and other asynchronous notifications. It provides a way to react to system events, timeouts, or custom notifications. The `handle_info/2` function in the case of this example `gen_callback` does nothing (returns {`noreply, State`}). Providing the `handle_info/2` callback is optional.

The `terminate/2` function in `gen_callback` is called when the process terminates. It allows the server to perform cleanup tasks, release resources, and finalize any necessary actions before exiting. The `terminate/2` function is called when a `stop` message is received from another `call_back` function. In this case, no cleanup is needed, it just returns `ok` and terminates the server. Providing the `terminate/2` callback is optional.

Overall, the `gen_server` abstraction simplifies the client/server interaction, state management, and error handling compared to the original approach. In our previous paper [10] we described a method based on static analysis that could be used to identify possible server processes. The identification process consists of the following two steps.

1. Analyzing the communication graph. The initial phase involves dissecting the communication graph. This graph represents the interactions between processes in the system. Each process is depicted as a node, and the edges symbolize relationships between these processes. These edges can signify various events, such as process creation, message passing, or interactions with the Erlang Term Storage (ETS). By scrutinizing this graph, we aim to identify processes that exhibit specific behaviours, such as starting, possibly registering, and receiving messages from other processes.

2. Filtering candidates based on rules. Once we have identified potential candidates, we proceed to the filtering stage. Our goal is to eliminate processes that do not align with the desired server behaviour. To achieve this, we employ a set of rules. These rules act as criteria for selecting relevant candidates. For instance, we might look for processes that match a client-server pattern, where a process serves as a resource provider and responds to requests from clients. These rules guide us in identifying code fragments that exhibit the desired behaviour.

One of the conditions for a code snippet to be equivalent to the generic server behaviour is that it must contain an expression that spawns a function that has to 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)

To refine the set of candidates, it might be worth examining if the receive block has branches for certain special messages that a server usually handles, for example, an exit message that leads to the termination of the process.

III. Transformation

In the preparatory section, the first step is for the transformation to verify whether it can be executed. It uses a series of checks needed based on the specific transformation using queries. If one of the checks fails the transformation cannot be executed and an error is thrown. Otherwise using the querying language of the tool, all information necessary for creating new

graph fragments is collected. Additionally, this is where new vertices or entire subgraphs to be inserted later are constructed. Subgraphs are typically built following a bottom-up approach.

The graph manipulation itself is performed in a list of lambda functions. At the beginning of each fun, we can still perform queries until the fun is called and the graph is modified. The actual graph modification always happens after the execution of each lambda function. The information flow between the two lambdas can be achieved by the parameters.

A. Steps

To refactor a possible server into an OTP `gen_server`, first, we have to do a series of checks based on the SPG and the data-flow analysis to see if the transformation is possible. First, we verify that a valid range for a function call is selected, and then we determine whether the spawned function behaves as a server by identifying a single spawned function. If there are multiple or none, it cannot be a server so the transformation is aborted. Next, we have to check if the spawned function has a receive clause. If not, recursively examine called functions. If none contain a valid receive clause, the transformation is not possible. Additionally, we check for any expressions with side effects before the receive clause in the server function. If such expressions exist, the transformation may not be possible, so we abort the transformation. If every check succeeds, we can construct the necessary callback functions for the new `gen_server`.

The `init/1` function is responsible for initializing the server state and performing any necessary setup. To construct the `init/1` function, we gather the arguments, body and return value needed from the original function call. The return value will be a tuple in the form of {`ok, {InitState}`}. If the server process was started with a `spawn/3` function, we can construct the `init/1` function by copying the expressions in the original function call. Otherwise, the result is constructed as follows: we copy the expressions in the spawn before the lambda function, we collect the arguments based on the arguments of the `spawn/3` function and finally, we construct the body of the new `init/1` function by copying the expressions of the original initialization.

The next important part is to construct the `handle_call/3` and `handle_cast/2` functions. To construct these we must process the receive expression inside of the loop function. Every tail recursive clause of the receive expression can correspond to a `handle_call/3` or `handle_cast/2` function. If a reply is the last expression before the recursive call it is a synchronous request and we can wrap the list of expressions in a `handle_call/3`, otherwise in a `handle_cast/2`.

We can also construct `handle_info/2` and `terminate/2` functions while processing the same receive expression. A terminate function will be built from a clause that is non-recursive and contains no reply.

IV. Use cases

In this section, we will look at our motivating example for recognizing server processes shown in our last paper and demonstrate how our refactoring algorithm transforms it into an OTP/`gen_server` implementation [10]. The example code snippet in Fig. 4 shows a simple Erlang server module implemented without OTP/`gen_server`.

The `init/0` function initializes the server by calling the `loop/1` function with an empty state (`#{}`). The `loop/1` function provides the main functionality of the server. It receives messages and responds accordingly.

To initiate the refactoring, first, we need to select the spawn expression of the server process, in this case, the `spawn(fun init/0)` expression. If not a valid range is selected, we abort. The first step is to check certain requirements that are needed to proceed with the refactoring. Some of the rules of the automated recognition process presented in II-C overlap with these checks, but before the transformation, another check is needed because the transformation can be started manually. If the server-candidate is not tail-recursive, does not contain a receive block or there is an expression with a side effect before the receive block, the transformation does not continue and we return with an appropriate error. In the case of the example, we have to examine the `init/0` function which is the process spawned in the `start/0` function. The `init/0` function itself directly does not have a receive block, but it calls the `loop/1` function which is a tail-recursive function containing a receive expression. Since there are no expressions with a side effect before the receive expression, we can proceed with the next step of the refactoring.

If every check is successful, we can collect all the information that would be needed for the transformation itself. We store the queried results in a record named `init_info` which contains everything needed to initialize the server process, the arguments of the spawn call, the expressions of the spawned function evaluated before the looping recursive call, etc. The first `gen_server` callback function we build is the `init/1` function. In the case of our example, the `init/0` function has no arguments, but since it starts the `loop/1` function we need to add the arguments of the function call (in this case `#{}`) in the return value of the constructed `init/1` function. The resulting `init/1` function is shown in Fig. 4.

The next step is to construct the `handle_cast/2` and `handle_call/3` functions by processing the main server loop. The receive block of the example code snippet contains multiple clauses depending on the message received. It handles the following message patterns:

- `stop`: Prints the server state and stops.
- {`job, ReqId, {M, F, A}`}: Spawns a new process to execute the function {`M, F, A`} and sends the result back to the `jobsrv`.
- {`reply, ReqId, To`}: Depending on the state, sends either a final result or a pending status to the recipient.
- {`result, ReqId, Result`}: Updates the state with the result.

To decide which of these clauses could be transformed to `handle_call/3` or `handle_cast/2` functions, we have to check the following: if a clause is tail-recursive and contains a reply exactly before the recursive call, we can transform

```
-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: ~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.
```

```
-module(gserver).
-behaviour(gen_server).
-export([init/1, handle_cast/2]).

start() ->
  {ok,X} = gen_server:start_link(jobsrv,gserver,{},[]),
  X.

stop() -> gen_server:cast(jobsrv, stop).

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



handle_cast(stop, {State}) ->
  io:format("Server stopped: ~p~n", [State]),
  {stop, normal, {State}};

handle_cast({job,ReqId,{M,F,A}},{State}) ->
  spawn(fun() ->
          jobsrv ! {result, ReqId,
          apply(M, F, A)}
        end),
  {noreply, {State}};

handle_cast({reply, ReqId, To}, {State}) ->
  case State of
    #{ReqId:=Data} -> To ! {final, ReqId, Data};
    _ -> To ! {pending, ReqId}
  end,
  {noreply, {State}};

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

handle_info(_Message, _State) -> ok.
terminate(_Reason, _State) -> ok.
```

Fig. 4. Original and transformed code snippet

the clause to a `handle_call/3` function by copying the expressions inside the clause and from this building the function itself. The return value of the function will be constructed from the reply expression and the argument of the recursive call wrapped in a tuple. This gives us the needed functionality of the `handle_call/3` function and provides the next state of the server. Similarly, in the case of the `handle_cast/2` functions, we build the body of the function from the expressions in the receive blocks appropriate clause before the recursive call and wrap the next server state in the return value.

The `terminate/2` function is called by a `gen_server` process when it is about to terminate. The main responsibility of this function is to do any necessary cleaning up. The return value of the function provides the reason why the `gen_server` process terminates. To ensure that the process can terminate, at least one of the branches of the server must be non-recursive. From these branches, we generate a `handle_call/3` function whose return value will be a tuple in the form of $\{\texttt{stop}, \ldots\}$. Using these we can ensure the server terminates as needed. In the case of this example there is no cleanup needed, so the `terminate/2` function returns the atom `ok`.

The last type of callback function needed is the `handle_info/2`, which is called by a `gen_server` process in the case of a time-out or when it receives any other message than a synchronous or asynchronous request or a system message (for example when using the ! operator to send a message).

V. Related work

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 implement common patterns of parallelism, allowing the programmer to instantiate parallel skeletons with application-specific code fragments [12]. PaRTE integrates capabilities of the RefactorErl and Wrangler refactoring/program analysis tools into a parallelisation framework that can be used to identify parallel patterns and determine the best implementations of those patterns [4], [13], [14].

Some well-known patterns are pipe (parallel pipeline) and task farm (applies a given function to a sequence of indepen-

dent inputs in parallel), the map-reduce and the divide-and-conquer patterns. Skel 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 [15]. 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.

Mel Ó Cinnéide proposes an automated approach to introduce design patterns into existing object-oriented programs [16]. The scenario assumed is as follows: an existing legacy program is being extended with a new requirement. The goal is to automate the introduction of design patterns to improve code quality and maintainability. The paper presents a pragmatic methodology for developing automated design pattern transformations. It breaks down each design pattern into smaller building blocks, which he refers to as "minipatterns". For example, if you consider the Factory Method pattern, it consists of components like the creator class, concrete product classes, and the factory method itself. Each of these components becomes a minipattern. Once the minipatterns are identified, corresponding transformations are developed. These transformations modify the existing code to introduce the desired pattern. Each minitransformation focuses on a specific aspect of the pattern. For instance, a minitransformation might create a new class for the factory method or adjust method signatures to match the pattern requirements.

Ensuring that behaviour is preserved during transformations is crucial. The automated process should not alter the program's functionality. The algorithm that describes the minitransformations is expressed as a composition of refactorings, using sequencing and iteration constructs. The author proves that the minitransformation itself is behaviour-preserving by computing the pre- and postconditions of the transformations using a novel algorithm presented in the paper.

VI. Conclusion

Using design patterns has some key advantages, for example, it simplifies code maintenance, promotes best practices and ensures consistent behaviour across different server processes. Previously we developed a process to identify possible server processes using the communication graph and the SPG of the RefactorErl tool. In this paper, we present a methodology to refactor custom server processes to the Erlang OTP/`gen_server` implementation. The main benefit of our approach is that it makes it easier to refactor legacy code bases by providing functionality to both identify the possible candidates for refactoring and apply the transformations to the code. By using the automatic refactoring developers do not need to spend as much time and effort on understanding the code-base and the result of the transformation will be a more easily understandable and maintainable code.

In future work, we aim to expand the prototype implementation to be able to handle more complex candidates as well as automate the full behaviour recognition and transformation process: so we can apply the transformations on the identified server candidates without human guidance.

References

[1] M. Ali and M. O. Elish, "A comparative literature survey of design patterns impact on software quality," in *2013 International Conference on Information Science and Applications (ICISA)*, pp. 1–7, 2013.

[2] F. Hebert, *Learn You Some Erlang for Great Good! A Beginner's Guide*. USA: No Starch Press, 2013.

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

[4] M. Tóth 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), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743, 2012.

[5] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, and R. Király, "Modeling semantic knowledge in Erlang for refactoring," in *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, vol. 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, (Cluj-Napoca, Romania), pp. 7–16, Jul 2009.

[6] K. Brown, "Design reverse-engineering and automated design-pattern detection in smalltalk," tech. rep., North Carolina State University at Raleigh, USA, 1996. https://dl.acm.org/doi/10.5555/890497.

[7] D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, and J. Chen, "Efficiently detecting structural design pattern instances based on ordered sequences," *Journal of Systems and Software*, vol. 142, pp. 35–56, 2018.

[8] M. Gaitani, V. E. Zafeiris, N. A. Diamantidis, and E. Giakoumakis, "Automated refactoring to the Null Object design pattern," *Information and Software Technology*, vol. 59, pp. 33–52, 2015.

[9] V. E. Zafeiris, S. H. Poulias, N. Diamantidis, and E. Giakoumakis, "Automated refactoring of super-class method invocations to the Template Method design pattern," *Information and Software Technology*, vol. 82, pp. 19–35, 2017.

[10] Z. Erdei, M. Tóth, and I. Bozó, "Identifying client-server behaviours in legacy erlang systems," *ACTA CYBERNETICA*, vol. x, pp. 1–25, 2024.

[11] T. Arts, C. Benac Earle, and J. Derrick, "Development of a verified erlang program for resource locking," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 2, pp. 205–220, 2004.

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

[13] M. Tóth, I. Bozó, and T. Kozsik, "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), Association for Computing Machinery, 2017.

[14] H. Li, S. Thompson, G. 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*, ERLANG '08, (New York, NY, USA), p. 61–72, Association for Computing Machinery, 2008.

[15] "A Streaming Process-based Skeleton Library for Erlang," 2012. https://github.com/ParaPhrase/skel.

[16] A. Christopoulou, E. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated refactoring to the Strategy design pattern," *Information and Software Technology*, vol. 54, no. 11, pp. 1202–1214, 2012.