Formalizing rules of transforming Erlang server applications*

Zsófia Erdei[1,*], István Bozó[1] and Melinda Tóth[1]

[1]Eötvös Loránd University, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary

Abstract

The development of methods that aid code comprehension and debugging is a current issue, requiring the introduction of new tools and approaches. We previously introduced an automated refactoring technique that transforms identified server processes in Erlang programs into OTP `gen_server` implementations. In this paper, we define the transformation rules that map receive clauses to appropriate callback functions while preserving the synchronous/asynchronous communication semantics of the original implementation. The rules ensure that the transformed code maintains functional equivalence with the original server.

Keywords

Erlang, refactoring, static analysis, formalization

1. 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 aid code comprehension [1], 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 [2]. Utilising 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.

Refactoring is a fundamental practice in software engineering that improves code quality without altering external behaviour. It plays a critical role in managing legacy code, which constitutes a substantial portion of enterprise software systems [3, 4]. Legacy code—typically characterised by outdated technologies, ad-hoc design patterns, poor documentation, and accumulated technical debt—presents significant challenges for software maintenance, evolution, and comprehension. Refactoring legacy systems is essential for reducing maintenance costs, improving reliability, and enabling future extensions. However, manual refactoring of large codebases is labour-intensive, error-prone, and time-consuming. Automated refactoring tools [5, 6] have emerged as a promising approach to address these challenges, enabling systematic transformation of legacy code to modern architectures while minimising human effort and reducing the risk of introducing regressions.

Legacy Erlang systems often contain server processes implemented with manual process management, including explicit spawning, message passing, and state handling. While these implementations are functionally correct, they lack the benefits of standardised OTP (Open Telecom Platform) behaviours, such as `gen_server`. OTP behaviours encapsulate common patterns of concurrent computation, providing built-in support for error handling, supervision, hot code reloading, and system monitoring. Transforming legacy server code to use OTP behaviours improves code maintainability, reliability, and integration with the broader Erlang ecosystem [7]. In this paper, we formalise our transformation rules to support the safe introduction of client-server behaviours.

SQAMIA 2026, September 16–18, 2026, Niš, Serbia

*Corresponding author.

zsanart@inf.elte.hu (Z. Erdei); bozo_i@inf.elte.hu (I. Bozó); toth_m@inf.elte.hu (M. Tóth)

0000-0002-5089-4984 (Z. Erdei); 0000-0001-5145-9688 (I. Bozó); 0000-0001-6300-7945 (M. Tóth)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-19

CEUR-WS.org/Vol-4336/paper2.pdf

Section 2 introduces the background of our research, including the server-client behaviour identification algorithm with a motivation example. Section 3 presents the transformed version of the identified server and demonstrates the Erlang/OTP's generic server behaviour. Section 4 presents the rules of the transformation. Section 5 demonstrates the usage of the refactoring on third-party examples. Finally, Sections 6 and 7 present related work and conclude the paper.

2. Background

Erlang [8] is a functional, concurrent programming language designed for building distributed, fault-tolerant systems. Originally developed by Ericsson for telecommunications systems, it is now used across many industries and areas. Erlang has found widespread use in domains where reliability, scalability, and real-time performance are essential. Erlang programs run within a virtual machine (Erlang VM or node), so programs written in Erlang are platform-independent. The standard library of Erlang is called OTP (Open Telecom Platform), the Erlang runtime environment and OTP are collectively called Erlang / OTP.

Its lightweight process model, message-passing concurrency, and robust error-handling mechanisms make it particularly well-suited for applications such as messaging platforms, web servers, and embedded systems. The strengths of Erlang lie in its ability to efficiently manage thousands of concurrent processes, recover from failures, and enable hot code upgrades without downtime.

A significant number of legacy systems, particularly in telecommunications, banking, and large-scale messaging infrastructures, have been built using Erlang. As these systems continue to operate, maintaining and developing them presents unique challenges. By providing automated insights into code structure, potential errors, and dependencies, static analysis helps ensure that updates and improvements do not introduce new faults, supporting the long-term stability and adaptability of these essential systems. By providing an understanding of program behaviour and potential failures, they help developers to build more reliable and maintainable systems. Static analysis tools are especially important for such legacy Erlang systems enabling developers to safely debug, expand, and refactor codebases that may have grown complex over decades. Expanding the set of static analysis and debugging tools [1] for Erlang can significantly enhance developer productivity and code quality.

The client-server model is commonly used for processes responsible for resources and for 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 shared across implementations, and a specific part (the callback module) in which users implement custom functionality and export a predefined set of functions.

2.1. RefactorErl

RefactorErl [1] [9, 4] 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 query semantic information or structural complexity metrics about Erlang programs, dependency examination among functions or modules, a function call graph with information about dynamic calls, etc. The tool offers multiple user interfaces: a web-based interface, an interactive console, and plugins for Emacs [10] or Vim [11] text editors.

RefactorErl builds an abstract syntax tree from the source code during the initial analysis, and complements it with additional semantic information, thus creating the Semantic Program Graph

[1] RefactorErl's website. https://plc.inf.elte.hu/erlang/

(SPG) [12]. The SPG has several types of nodes, each with distinct attributes that describe their properties. After the source code is analysed, this graph is stored in a database.

The SPG is a structurally weakly connected [2], directed graph whose vertices map to the lexical, syntactic and semantic elements of a program. This graph includes all the information that can be statistically extracted from the source code with different analyses. The SPG can be divided into three layers based on the types of stored information. The first is the lexical layer, the second is the syntactic layer, and the third is the semantic layer. The lexical layer contains both the original and the preprocessed tokens of the programs, storing the Erlang source code as is while preserving its formatting, while the syntactic layer stores the abstract syntax tree of the preprocessed source code. The semantic layer contains the results of various semantic analyses, such as side-effect, data-flow, dynamic function reference, and control-flow analyses.

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, dynamic function reference, and concurrent message-flow analyses [9].

2.2. Identifying server processes

Legacy Erlang systems often contain server processes implemented with manual process management, including explicit spawning, message passing, and state handling. While these implementations are functionally correct, they lack the benefits of standardised OTP (Open Telecom Platform) behaviours, such as `gen_server`, `gen_statem` or `gen_event` [13]. OTP behaviours encapsulate common patterns of concurrent computation, providing built-in support for error handling, supervision, hot code reloading, and system monitoring. Transforming legacy server code to use OTP behaviours improves code maintainability, reliability, and integration with the broader Erlang ecosystem.

Beyond easing maintenance, refactoring legacy servers into standardised behaviours materially improves code comprehension. Behaviour callbacks partition responsibilities into well-known entry points (`init/1`, `handle_call/3`, `handle_cast/2`, `handle_info/2`, `terminate/2`), making control flow and message handling explicit. This reduces the likelihood of errors or inconsistencies hidden in ad-hoc receive loops, simplifies the code, and makes it easier to debug or review. Refactoring also reduces technical debt by centralising error handling and supervision, enabling uniform observability and testing. Automated transformations further cut manual effort and the risk of regressions, providing a safer path to modernisation while preserving behaviour.

One of the most commonly used OTP behaviours is `gen_server`, which simplifies the implementation of server processes [14]. By leveraging `gen_server`, developers can create concurrent, fault-tolerant systems. Even today, there is a large amount of legacy code in use. These legacy codebases often suffer from readability issues due to their historical evolution, maintenance practices, and a 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 behaviour-driven architecture.

Overall, the `gen_server` abstraction simplifies the client/server interaction, state management, and error handling compared to the original approach. In our previous paper [15], we described a static-analysis-based method for identifying potential server processes. The identification process consists of the following two steps.

The initial phase involves analysing the communication graph [16]. This graph represents the interactions between processes in the system. Each process is depicted as a node, and the edges symbolise 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 scrutinising this graph, we aim to identify processes that exhibit specific behaviours, such as starting, possibly registering, and receiving messages from other processes.

Once we have identified potential candidates, we proceed to the filtering stage. Our goal is to eliminate

[2] Replacing all of its directed edges with undirected edges would produce a connected (undirected) graph

processes that do not align with the desired server behaviour. To achieve this, we employ a set of rules described below that act as criteria for identifying code fragments that exhibit the desired behaviour.

For a code snippet to be equivalent to the generic server behaviour it must contain an expression that spawns a function that has to satisfy all of 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.

The server module in our example shown in Figure 1 is named a. It defines the function `test/0`, which spawns a new process (identified by `Pid`) using `spawn_link/3`. 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()}` returns the accumulated result to the messaging process and sets the resource values 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.
```

Figure 1: Original server example

Our methods for identifying and refactoring a server process into an OTP `gen_server` implementation were presented in [15, 17].

3. Transformed server

Building on the pattern-identification work presented earlier [15], we present methods to automatically transform identified server candidates into standardised, OTP-compliant implementations. Our approach combines static analysis with automated refactoring to migrate legacy code to behaviour-driven architectures, enabling developers to gradually modernise large codebases without manual rewriting.

The refactoring process involves several key steps [17]. First, we identify candidate server processes using the communication graph and semantic program graph analysis as described in Section 2.2. Next, we analyse the identified candidates to extract their initialisation logic, message handling patterns, and termination conditions. We then map these extracted components to the corresponding callback

functions defined by the gen_server behaviour. Finally, we apply automated transformations to restructure the code into a gen_server implementation while preserving its original semantics. This process not only modernises the code but also enhances its maintainability and integration with OTP features.

```
-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.
```

Figure 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.
```

Figure 3: Callback module example

In the refactored version of the showed server (Figure 1) using OTP's generic server process (Figure 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 initialised 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` (Figure 3). The `init/1` function in `gen_callback` initialises 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 finalise 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.

4. Formalisation of transformation rules

This section defines the formalization of the refactoring steps that are used to convert the identified code snippets to the appropriate callback funcions. Transformation of a legacy server into a `gen_server` is formalised using rewriting rules that operate on the semantic program graph. We denote a rewriting rule using the following notation:

$$\frac{A \quad \text{when} \quad C}{B} \qquad (T)$$

where A is the *syntactic pattern*, B is the *replacement*, C denotes *semantic conditions* that must be satisfied, and T is the *transformation type*. Semantic conditions are verified using RefactorErl's query language on the SPG and may include control-flow properties (e.g., tail-recursion), data-flow constraints, or side-effect analysis. The transformation process operates in three phases: (i) a preparatory phase that gathers information and verifies preconditions, (ii) a construction phase that builds new graph fragments, and (iii) an integration phase that inserts them into the graph while maintaining consistency.

Since certain properties like tail-recursion and receive reachability are crucial for the transformation and would be a condition for all the transformation rules, they are not shown in the rules but are implicitly assumed to be satisfied for all transformations. The transformation rules are defined as follows:

4.1. Transformation of `init/1`

The initialisation callback is synthesised from the spawn expression and initialisation code. We express this transformation as:

$$\frac{spawn(F, InitState) \quad \text{when} \quad tail_rec(F) \wedge recv_reachable(F)}{init(State)} \tag{T_{init}}$$

Here, $spawn(F, InitState)$ matches the original process creation expression, and $init(InitState)$ denotes the synthesized callback function. The semantic guard requires that F is tail-recursive and that a receive expression is reachable from the function call. The initialisation code from the original function is extracted and placed in the callback body, with the return value wrapped in the tuple $\{ok, InitState\}$.

4.2. Transformation of receive clauses to `handle_call/3`

Receive clauses that contain a reply before the recursive call are transformed into synchronous request handlers:

$$\frac{\langle Pattern \rightarrow Expr1 \ldots Expr_n; RecCall\rangle \quad \text{when} \quad is_reply(Expr_i), i \in 1...n}{\langle Pattern \rightarrow Expr_1 \ldots Expr_n, \{reply, Msg, NewState\}\rangle} \tag{T_{call}}$$

The matched pattern becomes the request, expressions before the reply are executed in the callback body, and the result is returned as $\{reply, Msg, NewState\}$. As the constraint shows, one expression before the recursive call contains a reply. This reply is packaged in a tuple with the new state. This transformation preserves the synchronous semantics of the original receive clause by ensuring a reply is sent before continuing to the next iteration.

4.3. Transformation of receive clauses to `handle_cast/2`

Receive clauses without a reply before the recursive call are transformed into asynchronous request handlers:

$$\frac{\langle Pattern \rightarrow Expr1 \ldots Expr_n; RecCall\rangle \quad \text{when} \quad \neg is_reply(Expr_i), i \in 1...n}{\langle Pattern \rightarrow Expr_1 \ldots Expr_n, \{noreply, NewState\}\rangle} \tag{T_{cast}}$$

The callback body is constructed from the expressions in the receive clause before the recursive call, and the return value is wrapped as $\{noreply, NewState\}$. This transformation preserves asynchronous semantics by not sending a reply to the caller.

4.4. Transformation of `terminate/2`

Non-recursive receive clauses are transformed into process termination handlers:

$$\frac{\langle Pattern \rightarrow Expr1 \ldots Expr_n\rangle \quad \text{when} \quad \neg tail_rec_call(Expr_n)}{\langle Pattern \rightarrow Expr_1 \ldots Expr_n, \{stop, normal, State\}\rangle} \tag{T_{term}}$$

Non-recursive clauses indicate process termination paths. All logic is handled in a `handle_cast` function created, where the callback body is constructed from expressions in these clauses, and a cleanup routine can be performed. In addition, a `terminate` function is created.

4.5. Transformation of `handle_info/2`

Messages that do not match defined patterns are handled by a catch-all callback:

$$\frac{\langle Pattern \rightarrow Expr1 \ldots Expr_n; RecCall\rangle}{\langle Pattern \rightarrow Expr_1 \ldots Expr_n, \{noreply, State\}\rangle} \tag{T_{info}}$$

The `handle_info/2` callback is generated to handle unexpected messages, system events, or timeouts that are not explicitly handled by other callbacks. This ensures robustness by preventing the server from crashing in response to unexpected messages.

5. Applying the transformation rules

To demonstrate the practical applicability and effectiveness of our automated refactoring approach, we evaluate the transformation methodology using representative examples of Erlang server implementations. The evaluation demonstrates how our algorithm successfully transforms manually implemented server processes into standardised OTP `gen_server` implementations while preserving functional equivalence. Each example illustrates how the transformation rules extract and reorganise code fragments from the original implementation into the appropriate callback functions, automating the tedious and error-prone process of manual refactoring. By examining these examples, we demonstrate that the algorithm correctly handles the key challenges of server transformation: identifying initialisation logic, mapping receive clauses to callback handlers, preserving communication semantics (synchronous vs asynchronous), and maintaining state consistency.

5.1. The kitty server

The example code snippet in Figure 4 shows the cat server module implemented without OTP/`gen_server`, and the transformed version is shown in Figure 5.

```
order_cat(Pid, Name, Color, Description) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, {order, Name, Color, Description}},
    receive
        {Ref, Cat = #cat{}} -> erlang:demonitor(Ref, [flush]), Cat;
        {'DOWN', Ref, process, Pid, Reason} -> erlang:error(Reason)
    after 5000 -> erlang:error(timeout)
    end.
return_cat(Pid, Cat = #cat{}) ->
    Pid ! {return, Cat}, ok.
close_shop(Pid) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, terminate},
    receive
        {Ref, ok} -> erlang:demonitor(Ref, [flush]), ok;
        {'DOWN', Ref, process, Pid, Reason} -> erlang:error(Reason)
    after 5000 -> erlang:error(timeout)
    end.

start_link() ->
    spawn_link(fun init/0).
init() ->
    loop([]).
loop(Cats) ->
    receive
        {Pid, Ref, {order, Name, Color, Description}} ->
            if Cats =:= [] -> Pid ! {Ref, make_cat(Name,Color,Description)}, loop(Cats);
               Cats =/= [] -> Pid ! {Ref, hd(Cats)}, loop(tl(Cats))
            end;
        {return, Cat = #cat{}} -> loop([Cat|Cats]);
        {Pid, Ref, terminate} -> Pid ! {Ref, ok}, terminate(Cats);
        Unknown -> io:format("Unknown message: ~p~n", [Unknown]), loop(Cats)
    end.
make_cat(Name, Col, Desc) ->
    #cat{name=Name, color=Col, description=Desc}.
terminate(Cats) ->
    [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats], ok.
```

Figure 4: Original kitty server code snippet [14]

The `init/0` function initialises the kitty server by calling the `loop/1` function with an empty list state (`[]`). The `loop/1` function implements the cat server behaviour for ordering, returning, terminating, and defaults.

To initiate the refactoring, we select the spawn expression of the server process, in this case, the `spawn_link(fun init/0)` expression in `start_link/0`. If not a valid range is selected, we abort.

The first step is to validate the preconditions for continuing. Rules from Section 4 are applied, but we also ensure the transformation can be triggered manually. If the candidate is not tail-recursive, has no receive block, or has side effects before receive, we report an error. In the kitty server case, `init/0` is a thin wrapper that calls `loop/1`, and `loop/1` is tail-recursive and contains the receive expression. There are no side effects before receiving, so we proceed.

If every check is successful, we collect transformation metadata in the `init_info` record. This record includes the spawn call arguments, initial state, and any expressions evaluated before entering the recursive loop. The first `gen_server` callback we synthesise is `init/1`. For the kitty server, `init/1` returns `{ok, []}` because the original `init/0–loop/1` sequence initializes with an empty list of cats. For the transformation, we use the rule T_{init} (Equation T_{init}), which extracts the initialisation logic and wraps it in the standard `init/1` callback format.

```
init([]) ->
    {ok, []}.
handle_call({Pid, Ref, {order, Name, Color, Description}}, Cats) ->
    if
        Cats =:= [] -> {reply, make_cat(Name, Color, Description), Cats};
        Cats =/= [] -> {reply, hd(Cats), tl(Cats)}
    end;
handle_call({Pid, Ref, terminate}, Cats) ->
    {stop, normal, ok, Cats}.
handle_cast({return, Cat = #cat{}}, Cats) ->
    {noreply, [Cat|Cats]}.
handle_info(Unknown, Cats) ->
    io:format("Unknown message: ~p~n", [Unknown]), {noreply, Cats}.
terminate(_Reason, _Cats) ->
    ok.
```

Figure 5: Transformed kitty server code snippet

The next step is to construct the `handle_cast/2` and `handle_call/3` functions by mapping each receive clause in `loop/1` to OTP callbacks. Based on the pattern of message handling in the receive block, we determine which clauses correspond to synchronous requests (with replies) and which correspond to asynchronous updates (without replies). The pattern `{Pid, Ref, {order, Name, Color, Description}}` contains a reply to the sender, so it maps to `handle_call/3` using rule T_{call} (Equation T_{call}). The pattern `return, Cat` does not contain a reply, so it maps to `handle_cast/2` using rule T_{cast} (Equation T_{cast}).

The termination pattern `{Pid, Ref, terminate}` is non-recursive and leads to process termination, so it maps to a stop branch in `handle_call/3` using rule T_{term} (Equation T_{term}). In the case of this example, there is no cleanup needed, so the `terminate/2` function returns the atom `ok`.

Finally, any unmatched patterns fall into the catch-all case and map to `handle_info/2` using rule T_{info} (Equation T_{info}). In our example, the `Unknown` pattern is handled in `handle_info/2` to log unexpected messages without crashing the server.

The transformed handlers are shown in Figure 5.

5.2. The echo server

Consider a simple server process from legacy code, as shown in Figure 6. The server maintains state and processes messages from clients. To transform this identified candidate, we systematically apply our transformation rules.

This loop receives messages in two forms: either a tuple `{From, Msg}` containing a sender and a message, or the atom `stop` to terminate the process.

Since the original `loop/0` function has no initialization parameters, the `init/1` callback is minimal:

```
init([]) -> {ok, #{}}.
```

The empty initialiser returns an empty map as the server state. This follows rule T_{init} (Equation T_{init}), extracting the initialisation logic and wrapping it in the standard `{ok, State}` return tuple.

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

Figure 6: Loop function from the simple echo server example in Chapter 4 of [18]

The receive clause that handles `{From, Msg}` is transformed using rule T_{call} (Equation T_{call}):

```
{From, Msg} -> From ! {self(), Msg}, loop();
```

This clause produces a reply (`From ! {self(), Msg}`) before the recursive call. We identify this as a synchronous message pattern and transform it to a `handle_call/3` callback:

```
handle_call({From, Msg}, _From, State) -> {reply, self(), Msg, State};
```

Here, the message pattern `{From, Msg}` is abstracted as a `handle_call/3` request pattern. The reply expression (`From ! {self(), Msg}`) indicates that the caller expects a synchronous response. The `_From` parameter captures the caller's PID automatically provided by `gen_server`, and the response `Msg` is wrapped in the standard `{reply, self(), Msg, State}` tuple. This transformation preserves the semantics where the client waits for a response.

The non-recursive receive clause handling the `stop` message is transformed using rule T_{term} (Equation T_{term}):

```
stop -> true
```

Since this clause contains no recursive call and simply returns `true`, it signals process termination. It is transformed to:

```
handle_cast(stop, State) -> {stop, normal, State};
```

The return value `{stop, normal, State}` instructs `gen_server` to terminate the process with a normal shutdown reason.

Following rules T_{info} (Equation T_{info}), we add callbacks to handle edge cases robustly:

```
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
```

The `handle_info/2` callback ensures that unexpected messages do not crash the server. The `terminate/2` callback is optional in this simple example since no cleanup is required.

The transformed handlers are shown in Figure 7.

```
init([]) ->
    {ok, #{}}.
handle_call({From, Msg}, _From, State) ->
    {reply, self(), Msg, State}.
handle_cast(stop, State) ->
    {stop, normal, State}.
handle_info(_Info, State) ->
    {noreply, State}.
terminate(_Reason, _State) ->
    ok.
```

Figure 7: Transformed loop function from the simple echo server example in Chapter 4 of [18]

6. Related works

Refactoring has long been an important technique for improving internal structure without changing external behaviour, and it supports many approaches to software evolution and maintenance [3, 4]. In the Erlang ecosystem, refactoring is especially relevant because legacy systems frequently predate

OTP conventions and rely on ad-hoc concurrency patterns. Prior work has therefore focused on identifying common patterns of process interaction and enabling automated transformations that preserve semantics while aligning code with established behavioural abstractions. Our earlier work on recognising server processes provides a concrete starting point for such transformations, using static analysis to locate candidate processes and reason about their communication structure [15]. Building on the tools identification function, we implemented a prototype algorithm to transform the detected processes into OTP `gen_server` versions [17].

Automated refactoring tools further support this direction. RefactorErl and Wrangler provide the analysis and transformation infrastructure needed for syntax- and semantics-aware refactoring in Erlang, demonstrating that large-scale, behaviour-preserving transformations are practical in real development settings [9, 19] and can be integrated into developer workflows.

Related research has also explored the automation of parallelisation through pattern recognition. The structure of parallel computation in a program can be conveniently defined 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 [20]. PaRTE, which builds on RefactorErl and Wrangler, identifies code fragments that can be expressed as parallel skeletons such as pipes, task farms, map-reduce, and divide-and-conquer [21, 19, 9]. The approach targets list comprehensions, library calls, and recursive functions, and uses an initial shaping phase to normalise code before applying the transformation. Although such patterns can often be recognised syntactically, not every matching fragment is automatically safe to execute in parallel.

Some well-known patterns are the pipe (parallel pipeline) and task farm (applies a given function to a sequence of independent inputs in parallel), as well as the map-reduce and divide-and-conquer patterns. Skel [22] is a library of algorithmic skeletons for Erlang that provides a small set of useful, classical skeletons. 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.

The use of PaRTE is demonstrated with a simple worked example, showing how the tool can transform a sequentially implemented image merge into a parallel version and automatically achieve significant, scalable speedups over the original.

7. Conclusion

In this paper, we introduced a formal framework for transforming legacy Erlang server loops into OTP-compliant `gen_server` implementations. The proposed approach defines transformation rules for the main structural elements of server processes, including initialisation, synchronous and asynchronous message handling, termination, and fallback behaviour. By expressing these transformations precisely and systematically, we provide a clear foundation for automating the migration of ad hoc server code to standardised OTP abstractions. We formalised transformation rules that map common patterns in receive-based server loops to the appropriate `gen_server` callbacks, ensuring that the refactored code preserves the original semantics while benefiting from OTP's robustness and maintainability features. Through concrete examples, we demonstrated how our algorithm identifies initialisation logic, distinguishes between synchronous and asynchronous message handling, and ensures proper termination paths. This automated refactoring process significantly reduces the manual effort required to migrate legacy codebases to OTP standards, enabling developers to leverage OTP's powerful features to build reliable concurrent applications. The presented formalisation also provides a solid basis for further development of automated refactoring tools.

Declaration on Generative AI

Either:
The author(s) have not employed any Generative AI tools.

Or (by using the activity taxonomy in ceur-ws.org/genai-tax.html):
During the preparation of this work, the author(s) used X-GPT-4 and Gramby in order to: Grammar and spelling check. Further, the author(s) used X-AI-IMG for figures 3 and 4 in order to: Generate images. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, 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, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, Cluj-Napoca, Romania, 2009, pp. 7–16.

[2] M. Ali, M. O. Elish, A comparative literature survey of design patterns impact on software quality, in: 2013 International Conference on Information Science and Applications (ICISA), 2013, pp. 1–7. doi:10.1109/ICISA.2013.6579460.

[3] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2004) 126–139. doi:10.1109/TSE.2004.1265817.

[4] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, T. M., M. Tóth, RefactorErl - source code analysis and refactoring in Erlang, in: Proceedings of the 12th Symposium on Programming Languages and Software Tools, Tallin, Estonia, 2011, pp. 138–148. ISBN: 978-9949-23-178-2.

[5] M. Alharbi, M. Alshayeb, A comparative study of automated refactoring tools, IEEE Access 12 (2024) 18764–18781. doi:10.1109/ACCESS.2024.3361314.

[6] A. A. B. Baqais, M. Alshayeb, Automatic software refactoring: a systematic literature review, Software Quality Journal 28 (2020) 459–502. URL: https://doi.org/10.1007/s11219-019-09477-y. doi:10.1007/s11219-019-09477-y.

[7] F. Cesarini, S. Vinoski, Designing for scalability with Erlang/OTP: implement robust, fault-tolerant systems, " O'Reilly Media, Inc.", 2016.

[8] F. Cesarini, S. Thompson, Erlang programming, O'Reilly, 2009. URL: http://shop.oreilly.com/product/9780596518189.do, iSBN: 978-0-596-51818-9.

[9] M. Tóth, 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. doi:https://doi.org/10.1007/978-3-642-32096-5_9.

[10] Gnu emacs, 2026. URL: https://www.gnu.org/savannah-checkouts/gnu/emacs/emacs.html, [Accessed: August, 2026].

[11] Vim, 2026. URL: https://www.vim.org/, [Accessed: August, 2026].

[12] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, 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, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, Cluj-Napoca, Romania, 2009, pp. 7–16.

[13] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007. doi:https://doi.org/10.1007/978-3-540-73589-2_1.

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

[15] Z. Erdei, M. Tóth, I. Bozó, Identifying client-server behaviours in legacy erlang systems, ACTA CYBERNETICA x (2024) 1–25. URL: https://m2.mtmt.hu/api/publication/34749283.

[16] M. Tóth, I. Bozó, Detecting and Visualising Process Relationships in Erlang, Procedia Computer Science 29 (2014) 1525–1534. doi:10.1016/j.procs.2014.05.138.

[17] Z. Erdei, I. Bozó, M. Tóth, Transforming erlang server applications, in: 2024 IEEE 3rd Conference on Information Technology and Data Science (CITDS), 2024, pp. 1–6. doi:10.1109/CITDS62610.2024.10791346.

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

[19] H. Li, S. Thompson, G. Orosz, 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, Association for Computing Machinery, New York, NY, USA, 2008, p. 61–72. URL: https://doi.org/10.1145/1411273.1411283. doi:10.1145/1411273.1411283.

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

[21] M. Tóth, I. Bozó, 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, Association for Computing Machinery, New York, NY, USA, 2017. URL: https://doi.org/10.1145/3205368.3205369. doi:10.1145/3205368.3205369.

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