Safe process state upgrades through static analysis

1st Daniel Ferenczi
Department of Programming Languages and Compilers
ELTE, Eötvös Lóránd University
Budapest, Hungary
https://orcid.org/0009-0002-0611-006X

2nd Melinda Tóth
Department of Programming Languages and Compilers
ELTE, Eötvös Lóránd University
Budapest, Hungary
https://orcid.org/0000-0001-6300-7945

2024 IEEE 18th International Symposium on Applied Computational Intelligence and Informatics (SACI) | 979-8-3503-2952-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/SACI60582.2024.10619854

***Abstract*—Hot code loading allows changing the server code without any noticeable effects for clients. It is an integral part of the Erlang language and the BEAM runtime. Handling code change scenarios has to be developed into the application and requires care from the developer. This paper presents static analysis-based code checkers for detecting code errors which could impede performing hot code loads in Erlang applications. Our work analyses some specific erroneous patterns that impede a hot code load when migrating server states between code versions.**

***Index Terms*—Erlang, RefactorErl, Software Upgrades, Static Analysis, Hot Code Load**

I. Introduction

In today's software delivery environment applying frequent changes to deployed applications has become prevalent. We can see examples of banking services, news sites, gaming and betting websites serving client requests continuously, despite these applications receiving upgrades to their code in the meanwhile. Modern IT operation options, like serverless computing or container orchestration, standardizes many tasks, including the work needed to change code without causing downtime for the most superficial changes. Handling application state however is a more difficult topic, that requires careful planning from the professionals developing and applying the update, as errors could result in service downtime and any consequences of the failing software system.

Handling changes in the state of these applications adds complexity to the task of updating running software. E.g. a schema change in the backing database would introduce additional steps to the deployment to ensure that data is not lost and that the application finds it in a schema it supports. The application is often hosted across several instances, and several different versions could try to access the same persisted state.

In many cases, the application state can be a more fundamental component, like the connections between the server and the clients for long-running requests. In such scenarios losing state, and dealing with the consequent recreation of it, would pose a significant load on the server, further increasing the impact to the client's experience. Service developers want to migrate an application's state without any downtime of the application itself.

In this paper, we present a static analysis checker we developed for analyzing code upgrade and state migration safety in the language Erlang [1]. Erlang is a functional programming language originally stemming from the telecommunications domain but with general applications by now. The language provides built-in features for doing live code upgrades and migrating state. These features have to be coded correctly, as errors in the code responsible for state migrations will lead to application errors and downtime. As the ability to perform state changes is provided by the language, we can use an existing static analysis, refactoring and code comprehension tool, RefactorErl [2] to develop our checker.

II. Related work

The task of adapting existing state to new versions of software depends on the elements of the system. In a setup with state persisted in a relational database the task at hand would be migrating the schema between versions. A migration is necessary, as after a change in the database's table structure, queries in the application will become outdated, and a developer will have to adapt them to the new schema. Determining the query code to be changed is a large endeavor, and methods, tools to support this work have been researched [3], [4]. Even the database administration task of tracking changes across versions can be challenging enough to warrant tools [5] to support rewriting existing queries, assist in data migration or help in comparing schema versions.

RefactorErl has been used [6] to detect other types of upgrade-related errors. Due to the fact that BEAM can only store two versions of a code module simultaneously, the developer has to write function references with care, as when incorrectly used, they will refer to code no longer present in memory after a few upgrades.

RefactorErl has also been used [7] to identify behaviour patterns automatically in source code. Adapting behaviour patterns over existing legacy code is an improvement, as it makes the code easier to maintain, understand and reason about.

With regard to performing static analysis on Erlang, recent work published by Meta [8] shows an extension of their static analysis tool, infer, to support Erlang. The extension, InfErl supports as well adding user-defined extensions for code analysis

III. Preliminaries

A. Static Analysis for Erlang

Erlang is a general-purpose programming language, originally developed for use in the telecommunications domain. It is a dynamically typed, functional language designed to build fault-tolerant, concurrent and reliable systems. As by now such requirements are present in more domains, Erlang has been seeing broad use outside of the telecommunications industry for decades. Notably, Erlang has been used to develop WhatsApp, a chat application processing billions of messages every day [9] and is used for example at Amazon, Yahoo!, and finance-related companies like Goldman Sachs and Mastercard [10], [11]. Erlang runs on top of a virtual machine, BEAM, and the Erlang Runtime System (ERTS) which provides the basis for the language's massive concurrency features and hot code reloading. These features are provided inside the language itself for state-preserving code changes, whereas other programming languages, do not provide this feature out of the box, and require additions and changes to the software stack to support such upgrades, and even those might come with limitations. For example, work has been done [12] to support migrating existing network connections between server versions, but it requires changes to the stack operating the software itself. In addition, this approach only offers a solution for migrating connections and does not offer a solution to migrating states in a general manner.

In Erlang, as the capability for hot code loads is provided by the language there is no need to depend on specific details and customizations of the underlying system. The specific details of an upgrade in Erlang are part of the application itself. Consequently, correctness of their implementation depends on the server code itself, and errors will impede performing the upgrade safely. The code responsible for upgrades can however be analyzed with existing static analysis tools.

RefactorErl [2] is an open source tool for refactoring, code comprehension and static analysis of Erlang code. It is available for Windows, Linux and macOS, and provides integration with popular editors like Visual Studio Code, Vim or Emacs.

The tool works by storing source code in a Semantic Program Graph [13] (SPG), which even stores whitespace and comment information, so layout can be preserved during code transformations. SPG is based on an abstract syntax tree, which is enhanced by the semantic information generated by different analysers (function, data-flow, record, variable, etc.).

RefactorErl has an extensive semantic analyzer framework. It is easy to get information from the analysed source code, and implement further static analysis/new checkers on the top of that information. Therefore we choose this tool to implement our analysis.

For code comprehension and static analysis, the tool provides a rich, accessible query language [14] that can be used for analyzing the code in detail. Pre-built queries include for example the option to analyze the code for OWASP vulnerabilities [15]. A more powerful querying tool is also available: graph queries allow for direct traversal and analysis of SPG, and in fact, pre-built queries are implemented in this language. In our work, we have implemented code-checkers as graph queries and exposed these as semantic queries.

B. Erlang/OTP Behaviours

In our work, we analyze errors specific to state changes in upgrades of OTP `gen_server` behaviour implementations [16]. Erlang comes bundled with a library called OTP, which provides a building block called behaviour that helps abstract complex, generic details of applications into behaviours, and allows developers to only have to work on code specific to their business case. Erlang distributions also contain some default behaviours out of the box, which developers can use to implement for example servers or state machines, while only having to care about the code's unique parts. The behaviour will impose a code structure on the developer. They will have to implement callback methods of a given signature, which the library code can call. The behaviours included by default are:

- supervisor: addition of fault tolerance
- gen_server: development of server applications
- gen_event: managing events and triggered actions
- gen_statem: development of state machines

Developers also have the option to develop their own behaviours.

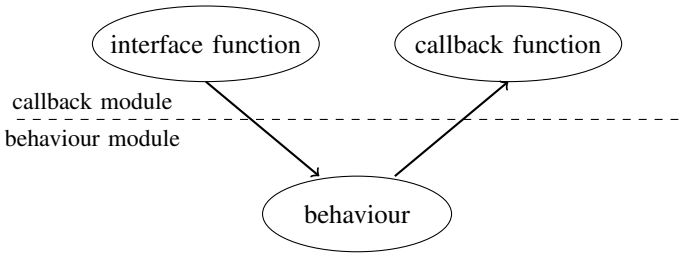

Fig. 1. A schematic drawing of behaviour structure

Figure 1 shows the schematic structure of behaviour implementations. Interface functions are exposed and called by users of the implementation. These functions make a call to the behaviour modules provided by OTP containing the generic implementation. Afterwards the behaviour code calls a callback function containing use-case specific code.

For example, a developer implementing a `gen_server` behaviour has to implement these callback functions. `code_change` is not mandatory, but is of importance for our case, as state updates have to be implemented here.

- `init(Args)` for initializing the server process.
- `handle_call(Request, From, State)/` for handling synchronous calls to the server. The State variable is the internal state of the server process.
- `handle_cast(Request, State)/` for handling asynchronous calls to the server. The State variable is the internal state of the server process.
- `code_change(OldVsn, State, Extra)/` for handling stage transformations during code changes.

State stores the internal state of the process, while Extra holds information related to the upgrade.

Erlang calls these functions callback functions, as they are invoked during calls, upgrades by the generic behaviour code to handle the particular request. The signature of these callbacks is important for us, as they are the starting point for inspecting references and changes to the process state. Out of these functions `code_change` is of special interest to us, as this function is to be invoked right after a code change, e.g. the upgrade of the application, and is responsible for converting the application's state to a new format.

The State argument is the internal state of a `gen_server` application. It is a variable that is used by handler functions and returned as replies to server calls. Its structure can of course change with new releases of the application. In these events, it is the `code_change` function's role to transform the old application's state to the format expected by the new version.

C. Motivating Example

The logic for a structure change of the state has to be implemented by the programmer creating the new version. It is also possible to handle upgrade paths from different versions. In this case the developer will have to take care to develop `code_change` functions for all different cases. An example for different `code_change` callbacks handling changes from different versions can be seen at Figure 2.

```
1 handle_call(get_num, _From, State) ->
2   {b, N} = State,
3   {reply, N, {b, N+1}}.
4
5 code_change("0", Num, _Extra) ->
6   {ok, {b, Num}};
7 code_change("1", {a, Num}, _Extra) ->
8   {ok, {b, Num}}.
```

Fig. 2. Example of callback function implementations, including code_change

The `code_change(OldVsn, State, Extra)` function has 3 parameters: OldVsn indicates the version code is upgraded from, State is the current internal state of the process, while Extra is additional information used during managed upgrades. In this example `code_change` functions in lines 5 ad 7 will upgrade the application state to the `{b, Num}` tuple. The `handle_call` function in line 1, responsible for handling `get_num` calls, retrieves the variable N through pattern matching, increases it by one and returns it in line 3. An important point to note, is the pattern matching in line 2. Clearly the state must be of the form `{b, N}` for the pattern matching to work correctly. If the state at this point of execution is of another format, for example, if it has a different number of elements, evaluation will fail and result in a runtime error. Finally, note how a state value is updated in line 3. Typically state value updates happen in handle functions.

In general, the use of application state and its structure should be in line with the state returned by code change functions. Changing the state's structure can cause errors, if the structure is not updated consistently across the codebase. A typical error would be the assumption of a given tuple structure in the state, after some element have been removed from it, or the reference of map keys that are already removed in the current version. In our work we inspect how to detect invalid references to maps and tuples in the application's state. We detail such references in the next section.

IV. Analyzed Patterns and Algorithms

Identifying errors related to a change in the state's structure can be done in compilation time, through static analysis. In our work we analyzed two errors of this kind: changes to maps and changes to tuples in the application state. Although we cannot decide whether the state structure present in `code_change` functions or the one referenced to in handle callback functions is the one reflecting developer intent, we can discern inconsistency: in the case of maps, the keys used in code handling calls to the server should be present in the map returned by `code_change`.

Similarly for tuples: in our call handlers we can only retrieve elements that exist in the tuple, thus functions that retrieve the `N`th element from a state tuple only work if the updated state is a tuple of at least `N` elements.

In the following subsections we'll revise functions that might be incompatible with an updated version of either a map or a tuple in the state.

A. Analysis of maps

In Erlang maps have the syntax `#{key1 => "A", key2 => 42}`, where key1 and key2 are the keys of the map, while "A" and 42 are their respective values. A simple example for a `code_change` function altering the map used as a state can be seen in Figure 3.

```
1 code_change("1", State, _Extra) ->
2   NewMap = #{key1 => "A", key2 => 42},
3   {ok, NewMap}.
4
5 handle_call(unsafe, _From, State) ->
6   maps:update(key3, "Danger", State).
```

Fig. 3. Example of map use in behaviour state, including an unsafe use

The `code_change` function in line 1 returns a new state containing keys `key1` and `key2` during an upgrade from version "1". The new version however contains an unsafe handler at line 5. When the server is called with the argument `unsafe`, the associated handler will try to update the key `key3` in the map, which is not present in it. This error will trigger a runtime failure of the application. Such errors could have been caught before deployment, as the absence of `key3` from the state resulting from `code_change` can be determined through static analysis.

In general we are looking for the use of functions in handlers that refer to a given key. Our goal is to verify the presence of these keys in the state returned by `code_change` functions. We have identified functions in figure 4 as those using map keys as arguments:

- `maps:get(Key, Map)`: returns the value associated with Key
- `maps:update(Key, Value, Map)`: updates Key's value to Value
- `maps:update_with(Key, Fun, Map)`: applies Fun over Key's value

Fig. 4. List of functions with keys as arguments

When using these functions with a key not present in the map argument a {`badkey,Key`} exception will be generated. These functions have variants that an additional argument, a default value. These functions are safe for the purposes of our analysis, as when the key is not found, the default value is returned.

B. Analysis of tuples

```
1 code_change("1", State, _Extra) ->
2   NewTuple = {A, B, C},
3   {ok, NewTuple}.
4
5 handle_call(unsafe, _From, State) ->
6   erlang:delete_element(4, State).
```

Fig. 5. Example of map use in behaviour state, including an unsafe use

Erlang tuples have the syntax of {`"1st", 2, third`}. Similarly to maps, a version upgrade may change the state's value, and set it to a tuple containing less items than in the version before. An example for this can be seen in Figure 5. In

- `erlang:element(Index, Tuple)`: returns variable at given index
- `erlang:setelement(Index, Tuple, Value)`: sets element at given index to Value
- `erlang:delete_element(Index, Tuple)`: removes element at given index
- `erlang:insert_element(Index, Tuple, Term)`: insert element at given index, pushes elements with larger index

Fig. 6. List of functions with keys as arguments

this example, `code_change` sets the state to a 3-tuple in line 3. A handler function attempts to delete the fourth element of the state in line 6, but will result in a runtime exception, as the delete argument is out of range. Again, errors like the above can be determined before execution, as we can compare the size of the tuple returned by `code_change` with the index argument of functions that work on tuples. These functions are presented in Figure 6.

Using these functions with an `Index` larger than the size of the tuple set in the state will result in an exception. Safety can be checked statically, as we can analyze the tuple set by `code_change` and check if there are any references to invalid indices in functions working with the state tuple.

C. Examples

```
1  -module(server_maps).
2  -version("3").
3  -behavior(gen_server).
4  -export([get_num/0]).
5  -export([init/1, handle_call/3,
6    code_change/3]).
7
8  get_num() ->
9    gen_server:call(?MODULE, num).
10 init([]) ->
11   {ok, #{num => 0, requests => 0}}.
12 handle_call(num, _From, State) ->
13   StateCopy = State,
14   SafeRetrieval = maps:get(num, StateCopy),
15   UnsafeUpdate =
16     maps:update(missing_key, 42, StateCopy),
17   {reply, UnsafeUpdate}.
18
19 code_change("1", State, _Extra) ->
20   {ok, #{num => 0, requests => 0}};
21 code_change("2", State, _Extra) ->
22   {ok, #{num => 0, requests => 0}}.
```

Fig. 7. A more complex example using a map in the state

For some more elaborated examples, in Figure 7 we can find a more complex module using a map in the application's internal state. Lines 2, 19 and 21 suggest, that the version being loaded is version "3", and it supports upgrades from versions "1" and "2". Both upgrade paths set a state with the keys `num` and `requests`. The application's only handler in line 12 first copies the `State`, and then retrieves the value of key `num` in line 14. In line 16 it attempts to update key `missing_key`. This expression would fail during execution as both upgrades set a state that lacks this key. Such errors can be identified by static analysis, so that the developer can fix them before delivering the application.

Similarly, lines 20 and 22 of the example in Figure 8, show the upgrade paths to version "3". The handler first modifies a copy of the state in line 15, after which it attempts the removal of the variable's third element in line 17. As the tuple only has two elements, the function call fails and results in an error. Similarly as with maps, errors like the one in the example can be detected by static analysis.

D. Algorithms

In order to identify unsafe use of state maps and tuples as in Figures 3 and 5 we have considered one algorithm for each data type. Both algorithms should take a module's name as

```
1  -module(server_tuples).
2  -version("3").
3  -behavior(gen_server).
4  -export([get_num/0]).
5  -export([init/1, handle_call/3,
6    code_change/3]).
7
8  get_num() ->
9    gen_server:call(?MODULE, num).
10 init([]) ->
11   {ok, #{num => 0, requests => 0}}.
12 handle_call(num, _From, State) ->
13   StateCopy = State,
14   ModifiedState =
15     erlang:setelement(2, StateCopy, b),
16   UnsafeDelete =
17     erlang:delete_element(3, ModifiedState),
18   {reply, UnsafeDelete}.
19
20 code_change("1", State, _Extra) ->
21   {ok, {ok, {a, 1}}};
22 code_change("2", State, _Extra) ->
23   {ok, {ok, {a, 1}}}.
```

Fig. 8. A more complex example using a tuple in the state

1) Infer valid state structure from `code_change` return values
2) Find the handler functions for the behaviour
3) Find expressions that can be reached by the state argument of the callback function
4) Filter to those expressions that can use either maps or tuples in an unsafe way
5) Determine safety of expressions
6) Return unsafe expressions

Fig. 9. Overview of the unsafe state use detection algorithm

input and return those expressions that contain an unsafe use of the state returned by `code_change` implementations. In general, our algorithms must

When we filter down to expressions that can use maps and tuples in an unsafe way we are looking for the expressions we already presented in Figures 4 and 6.

To analyze the reach of state arguments in callback functions, we use inter-procedural data-flow analysis [17]. With the first order data-flow relation we can find the expressions that are reached by the state. This step is necessary as the state might be passed on to other functions, pattern matched against or copied into other variables. To determine safety, we are interested in these copies of the state as well, as they retain structure, and can be used in expressions. RefactorErl's Semantic Program Graph stores edges representing data-flow in a Dataflow Graph layer. RefactorErl already provides functions that provide first order data-flow reaching analysis, and can be used when developing custom queries. Our checker relies on using these queries to find the expressions where the application's internal state is used. For an example for code where data-flow reaching is needed, consider line 13 on

Algorithm 1 Finding unsafe uses of *gen_server* states

```
1:  function FIND(Module, FunctionsAndConditions)
2:      set_state ← find_state_in_code_change(Module)
3:      callbacks ← find_callback_functions(Module)
4:      for all callback ∈ callbacks do
5:          callback_state ← get_callback_state(callback)
6:          callback_expressions ←
                get_reached_expressions(callback_state)
7:          for all callback_expr ∈
                callback_expressions do
8:              expr_args ← get_expr_args(callback_expr)
9:              expr_verifier ← get_verifier(callback_expr,
                    FunctionsAndConditions)
10:             insecure_callback_expressions ←
                    expression_verifier(expr_args, set_state)
11:         end for
12:     end for
13: end function
```

both examples from Figures 7 and 8. The original state is copied into another variable which is then used in subsequent expressions. Using data-flow reaching is needed to detect those expressions where for example a copy of the original state is used.

When looking at the overview of our algorithm (Figure 9) we can note that the parts specific to either the analysis of maps or that of tuples are points 4 and 5. Based on this observation we developed a single algorithm that apart from the module to scan, also receives a list of tuples with functions and associated safety conditions. With such a list, our algorithm can determine unsafe uses dynamically, and becomes more open for extension. Of course the end user shouldn't have to care about specifying under which conditions a given function is unsafe. To this end, we have created a wrapper together with our generic algorithm (Algorithm 1) that applies it on a list of function and condition pairs required to validate map and tuple safety in states.

Algorithm 1 starts with gathering the state set after the upgrade and the callback function that are present in the module. Afterwards it iterates through each callback function and identifies expressions reachable by the argument state. It then proceeds to find a safety condition for these expressions. If none are found, we can consider the expression to be safe. In case of a match it checks if the expression arguments are compatible with the originally set state, and returns those expressions that are found incompatible.

V. Results and Further Work

A. Results

In order to test our checker on real-life code-bases we have studied the upgrade logic of large open-source Erlang projects like cowboy [18], VerneMQ [19] and yaws [20]. As unfortunately they did not contain state transformations where we could apply our checker, we only tested it on synthetic examples. Complex upgrade logic is more common in closed

code-bases and we are currently looking for an industrial partner to test and refine our checkers with.

B. Further Work

Our work can be extended by analyzing further patterns that impede upgrades. We could improve our analysis by considering nested structures in the state. Another closely related check could be the analysis of states returned by `code_change` and handler instances as they should be consistent with each other. In general terms, the types of the returned states should be compatible.

Exploring the use of InfErl, and comparing its features for analysis and extensionability with those of RefactorErl would also be a topic of interest.

Exploring how machine learning, particularly large language models can be combined with static analysis to aid error checking would also be of interest. Models like GitHub's copilot and OpenAI's ChatGPT are being used for code generation, but increasing the model results' quality [21] and ensuring correctness is still a task to be solved.

VI. Conclusion

There is an increased demand for upgrading application without an impact on availability. This requirement adds complexity to state changes of applications. Although Erlang applications provide the means to perform hot code loads and changes to the application's internal state, it is the developer's responsibility that the code performs the state changes correctly.

In our research we looked at possible changes to Erlang `gen_server` application's internal state. We identified two problem sources: changes in the size of tuples or absence of keys in a map. We developed our checker using RefactorErl. We built a generalized algorithm which can be expanded to support checking future expressions and safety conditions.

References

[1] F. Cesarini and S. Thompson, *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, 2009.

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

[3] L. Meurice, C. Nagy, and A. Cleve, "Detecting and preventing program inconsistencies under database schema evolution," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 262–273.

[4] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 451–460. [Online]. Available: https://doi.org/10.1145/1368088.1368150

[5] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: The prism workbench," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 761–772, aug 2008. [Online]. Available: https://doi.org/10.14778/1453856.1453939

[6] D. Ferenczi and M. Tóth, "Static analysis for safe software upgrade," in *Annales Mathematicae et Informaticae*, 2023.

[7] I. B. Zsófia Erdei, Melinda Tóth, "Identifying concurrent behaviours in erlang legacy systems," in *The 13th Conference of PhD Students in Computer Science : Volume of Short Papers*, no. 5. Szegedi Tudományegyetem, Informatikai Intézet (2022), 2022, pp. 207–210.

[8] Á. Hajdu, M. Marescotti, T. Suzanne, K. Mao, R. Grigore, P. Gustafsson, and D. Distefano, "Inferl: scalable and extensible erlang static analysis," in *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*, 2022, pp. 33–39.

[9] E. Solutions. (2018) 20 years of open source erlang: Openerlang interview with anton lavrik from whatsapp. [Online]. Available: https://www.erlang-solutions.com/blog/20-years-of-open-source-erlang-openerlang-interview-with-anton-lavrik-from-whatsapp/

[10] F. Cesarini and S. Thompson, *Erlang programming: a concurrent approach to software development*. " O'Reilly Media, Inc.", 2009.

[11] F. Cesarini. (2019) Which companies are using erlang, and why? Erlang Solutions. [Online]. Available: https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus/

[12] U. Naseer, L. Niccolini, U. Pant, A. Frindell, R. Dasineni, and T. A. Benson, "Zero downtime release: Disruption-free load balancing of a multi-billion user website," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 529–541. [Online]. Available: https://doi.org/10.1145/3387514.3405885

[13] 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*, ser. Studia Universitatis Babeş-Bolyai, Series Informatica, vol. 54(2009) Sp. Issue, Cluj-Napoca, Romania, Jul 2009, pp. 7–16.

[14] M. Tóth, I. Bozó, J. Kőszegi, and Z. Horváth, "Static analysis based support for program comprehension in erlang,," In Acta Electrotechnica et Informatica, Volume 11, Number 03, October 2011. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), pages 3-10.

[15] B. Baranyai, I. Bozó, and M. Tóth, "Supporting Secure Coding with RefactorErl," *Submitted to the ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica*, 2020.

[16] M. Logan, E. Merritt, and R. Carlsson, *Erlang and OTP in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[17] 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.

[18] L. Hoguin. cowboy. [Online]. Available: https://ninenines.eu

[19] verneMQ authors. Vernemq. [Online]. Available: https://vernemq.com

[20] yaws authors. yaws - yet another webserver. [Online]. Available: https://erlyaws.github.io

[21] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," *arXiv preprint arXiv:2304.10778*, 2023.