Detecting and Ranking Atom Creation Related Bottlenecks

Róbert Fikó, István Bozó and Melinda Tóth

ELTE, Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary

Abstract

Nowadays, our systems run constantly without the need to shut down long-running applications during source code updates. As a result, developers and operators must handle issues that arise during continuous operation, unpredictable conditions, heavier load, etc. Dynamic analysers can help detect these issues but do not provide information about their source. In this paper, we show how we combined static and dynamic analyses to identify the source of the atom generation bottleneck in Erlang applications. We present our algorithm to rank the candidates and demonstrate the results on an open-source project.

Keywords

static analysis, dynamic analysis, Erlang, atom generation, candidate ranking

1. Introduction

Our world is heavily reliant on software. Think of telecommunication, banking, navigation, nuclear power plant management, and many more. Most of them consist of hundreds of thousands of lines of source code, and they are constantly running, and the crash of these systems is just not acceptable. Even code updates are executed without a shutdown and restart (hot code reloading).

By nature, some kinds of issues go undetected during testing and only appear in production systems. Identifying those issues is time-consuming and not straightforward. The goal of our research is to provide tools for developers to efficiently identify the source of these issues.

A good monitoring (dynamic analyser) tool can notice these issues and can give the developers early alarms. Usually, the monitoring tool also has access to the runtime of the software, so it is possible to obtain additional information and provide this to the static analyser tool.

Both dynamic and static analysers can give good information about issues in the system, but none of them can point you to the exact issue in the source code alone. Therefore, we combined static and dynamic analyses to be able to find bottlenecks in Erlang [1] source codes more easily.

One of the bottleneck types is related to dynamic atom creation. Atoms are constants with a name stored in the **atom table**, which has a limited size. The current number of atoms in the table is the **atom count**. When we have a system that is about to reach one of its system limits, such as atom count, then the dynamic analyser tool provides some additional data, such as the content of the atom table, to the static analyser, which leverages it to make the analysis more accurate.

This paper is structured as follows. In Section 2 we describe the used tools, RefactorErl and Wombat. In Section 3 we briefly introduce the proposed methodology to follow. Section 4 describes a motivational example and the details of our analysis steps. Section 5 demonstrates the steps of the algorithm on a small example, and Section 6 presents the evaluation of the proposed method on an open-source project. Finally, Sections 7 and 8 describe related work and conclude the paper.

SQAMIA 2024: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 9–11, 2024, Novi Sad, Serbia

g55ofz@inf.elte.hu (R. Fikó); bozoistvan@elte.hu (I. Bozó); tothmelinda@elte.hu (M. Tóth)

0000-0001-5145-9688 (I. Bozó); 0000-0001-6300-7945 (M. Tóth)

2. Background

Our work focuses on the Erlang programming language [1]. **Erlang** is a dynamically typed, functional programming language designed for building fault-tolerant, scalable, distributed systems. The BEAM Virtual Machine supports smooth, live code upgrades. Thus, Erlang systems are running continuously.

Erlang offers only a small number of built-in data types and it is not possible to introduce user-defined types. We would like to highlight here the atom type. According to the Erlang documentation, "an atom is a literal, a constant with name" [2]. The text value of every unique atom is stored in the atom table. We use "static" atoms for function names, module names, process names, etc. but it is also possible to generate atoms at runtime.

RefactorErl [3, 4] is a static analysis and transformation tool. The main focus of the project is to provide a tool for Erlang developers, which can help them to understand (code comprehension) and refactor their code. RefactorErl offers security checkers, which can help to identify vulnerabilities in Erlang sources [5].

RefactorErl represents the information in the semantic program graph (SPG), which contains every information that may be needed during refactoring and analysis. The graph is a multi-layered, labelled, directed attribute graph. It has a lexical, syntactic and semantic layer. The directed links represent certain relations in the graph. The SPG also includes the direct data-flow relation among expressions [4]. Our analysis relies on this information to calculate the possible values of certain expressions.

WombatOAM[6] (Wombat Operation, Administration and Maintenance) is a dynamic analysis, and monitoring tool developed by Erlang Solutions. The WombatOAM is a standalone server, which is designed to operate with one, more or no observed nodes and communicate via the Erlang distribution. WombatOAM can monitor and measure different metrics of Erlang and Elixir systems. Metrics collect data about the number of processes, number of ports, memory usage, message queue length, and number of atoms. It is possible to configure alarms. WombatOAM can raise alarms when a certain metric reaches a threshold. If an alarm has been dealt with, it can be cleared. Alarms can also trigger notifications, for example, sending an email or a Slack message.

3. Method

The behaviour of production systems can be observed with dynamic analysers. Those tools usually monitor and check some properties of the observed running system without access to the source code. If some of the metrics reach a predefined threshold, the monitoring tool raises an alarm. Unfortunately, an alarm cannot give a location in the source code, but it can give additional data about the system, e.g. information about memory, processes, etc.

On the other hand, static analyser tools can check the properties of the software based on its source code without running it. Therefore when an alarm is raised, the static analyser can run checkers to identify the possible issues in the source code. It starts by collecting all the possible points of errors, and then with the help of data flow and control flow analysis, it will try to eliminate as many false positives as possible. Enhancing further, the data provided by the monitoring tool from the runtime will be loaded into the system, so it can look for specific code points where similar issues can happen. This means that the static analysis is adapted to the currently running system. This extra information helps to rank the statically identified candidates.

Figure 1 shows the proposed methodology and infrastructure that is established to detect the issues. We used WombatOAM for monitoring and RefactorErl for static analysis.

4. Identifying atom related bottlenecks

Atoms are stored in the atom table, whose size is limited, and defined during node startup. By default, it has 1,048,576 [2] available slots but can be overwritten. The problem is that when a system runs for a long time, this table can fill up, as it is not garbage collected. The only way to clear it is to restart the

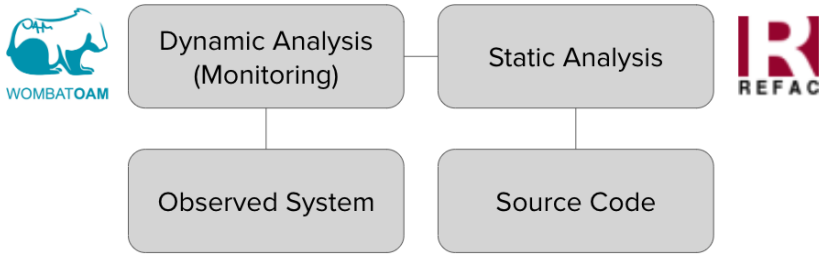

Figure 1: Methodology

system. Consequently, it can crash if no monitoring is applied to the node. We aim to find the source code fragments, where atoms are created and may cause atom table exhaustion.

4.1. Motivational example

The example in Figure 2 demonstrates how the atom table can be filled up. There is a module called `atoms`, with the exported functions `gen_ids/1` and `convert_ids/1`.

The function `gen_ids/1` takes a list of strings and creates atoms from them. When it is called, it will iterate on its argument list, and for each element, it will call the `get_id/1` function. This function has two clauses. The first one is in the case when `Id` is a list: an atom with the `generic_proc_` prefix will be created. The second is when `Id` is a binary: an atom with `example_proc_` prefix will be created.

```
-module(atoms).
-export([gen_ids/1, convert_ids/1]).

convert_ids(List) ->
    [list_to_atom(Id) || Id <- List].

gen_ids(List) ->
    [list_to_atom(get_id(Id)) || Id <- List].

get_id(Id) when is_list(Id) ->
   Format = "test_" ++ Id,
   "generic_proc_" ++ Format;
get_id(Id) when is_binary(Id) ->
   "example_proc_" ++ binary_to_list(Id).
```

Figure 2: Example on possible atom exhaustion

The `gen_ids/1` function is exported, thus it can be called from other modules, and it can create a lot of atoms (in an unsupervised way), and fill up the atom table with atoms like `generic_proc_test_1`, `generic_proc_test_2`, `example_proc_1`, `example_proc_2`, and so on.

In this case, if we examine the atom table, we can see that many atoms match the patterns `generic_proc_*` and `example_proc_*`. With the dynamic analyser tool, it is possible to set an alarm, which can start an on-demand static analysis of the source code and aid it with the content of the atom table.

This analysis aims to find the source code fragments where the atom table can be filled up and associate them with the atoms that can fill up the atom table. In this motivational example, the code fragments, where the atoms are created, is the `gen_ids/1` function.

The function `convert_ids/1` just converts the items in the list into an atom. This function is very general we cannot deduce any information or pattern from it.

4.2. Getting the content of the atom table

We need the current atom table before we can get into the analysis. There is an undocumented feature of the external term format [7], and an Erlang term encoded in it (starts with byte 131 [7]), which contains a type byte 75 which is not described in the official documentation of the external term format [7, 8].

According to EEP-43 [9], which mentions all the possible types, we can find that `ATOM_INTERNAL_REF3` (75) is the needed type byte [7]. With this knowledge, we can create a binary, which returns the 0th atom in the table: `<<131, 75, 0:24>>`. Following this pattern, `<<131, 75, N:24>>` will return the n-th atom. [1]

Now we can keep increasing `N` in the above-mentioned binary, and query the atom table one by one. If we reach the end of the atom table, we will receive a bad argument error.

Note: When extracting the atoms from the observed remote node, it is recommended to convert them to strings before working with them. In this way, you can avoid atom exhaustion on your node.

4.3. Definitions

The content of the atom table during the analysis is represented as strings and we need a way to match the Erlang terms and expressions found by static analysis with these data. For this purpose, we will introduce the **match tree** structure, which is an intermediate representation of the Erlang terms that the static analysis finds.

Match trees can be easily converted to a regular expression to match strings, but it also keeps structural data, so it is possible to calculate a match score from it, which expresses how strict or permissive the pattern is.

Match trees are binary trees containing constructs useful for matching. It is similar to an abstract syntax tree but much simpler. It can contain a **string** node, when the value of certain parts is known, so it needs to be matched. When the value is unknown, a **var** node is introduced. In case of multiple values need to be present next to each other, a **concatenation** node is introduced. It has two children, both can be sub-tree. If a certain part of the text can have multiple values an **option** node is used. This node has two children, and each can be a sub-tree.

Node Type	Example Erlang Expression	Resulting Match Tree Node
String node	`"hello"`	`{str, "hello"}`
Var node	`A` (possible values are "hello" and "word")	`{var, ["hello", "world"]}`
Concatenation	`«"hello", "world"»` or `"hello" ++ "world"`	`{'+++',{str, "hello"}, {str, "world"}}`
Option	`foo()` (with possible evaluations 1 or 2)	`{'|||',{var, [1]}, {var, [2]}}`

Table 1
Examples of Match Tree Nodes

Examples of match tree nodes are summarized in Table 1.

These match trees can be turned into regular expressions, and then match the atoms in the atom table. It is turned into regular expressions with the following rules:

- **Concatenation node**, the regular expression is the concatenation of the regular expressions of the children.
- **Option node**, the regular expression is the regular expression of the children, with a | separator.
- **String node**, the regular expression is the string itself.
- **Var node**, the regular expression is the possible values of the var node, with a | separator.

[1] 131 indicating the needed external term, 75 indicating the ATOM_INTERNAL_REF3 type, 0 means that we are requesting the zeroth atom, and 24 means that it is 24 bit encoded integer

Match score is a number calculated based on the match tree. The higher the score, the more specific the match is. It is important in ordering the final results. The result with a higher match score is more probable to produce exhaustion in a running system. Its calculation:

- **option node**: it will inherit the score of the child with the larger score
- **concatenation**: it will be the sum of the child scores
- **string**: it will be the length of the string
- **var**: in case there is a joker, in the possible results, it will be 0, as no matter the other options, this option will match anyhow, making the score of other possible options invaluable, otherwise 1

4.4. Analysis

The analysis requires the current atom table as an additional input next to the analyzed source code in a semantic program graph representation.

The atom creation analyser (Algorithm 1) starts by getting the function calls that potentially create many atoms. We will call these candidates. The following steps are responsible for filtering out the ones that might create atoms, so we will remain with the source code fragments, which are worth our attention. To achieve this, the algorithm will iterate over the candidates and will do the following steps:

- selects the arguments of the calls
- builds the match tree,
- convert the match tree into a regular expression,
- counts the matches on the atom table.

Algorithm 1 Atom exhaustion source code fragments, based on atom table

Funct get_atom_heavy_calls($atom\_table$)

```
1:  calls := get_candidates()
2:  results := []
3:  i := 0
4:  while calls.length > i do
5:      args = get_arguments(calls[i])
6:      match := build_matcher(args)
7:      regexp := create_regexp(match)
8:      score := matcher_score(match)
9:      number := count_matches(atom_table, regexp)
10:     results.push([call, number, regexp, score])
11:     i := i + 1
12: end while
13: return results
```

The above-mentioned and explained algorithm will then return the same list of candidates with meta-information for the number of matches in the atom tree and the match score (Section 4.3). The calls that have zero matches will be dropped from the result. The remaining results will be grouped based on their match score and ordered based on the number of matches.

Note: the implementation of Algorithm 1 uses RefactorErl's security library [10, 5] to identify the vulnerable expressions where atoms are created.

5. Demonstration of the algorithm

Following on the motivation example introduced at the beginning of Section 4.1, we have an Erlang VM with the following atom table (Figure 3), running the source code introduced in the example (Figure 2). It is observable that we have a lot of atoms that can come from a common pattern.

```
false, true,
...
generic_proc_test_1,
generic_proc_test_2,
...
generic_proc_test_350,
example_proc_1,
example_proc_2,
...
example_proc_432,
process_registry,
...
```

Figure 3: Example atom table

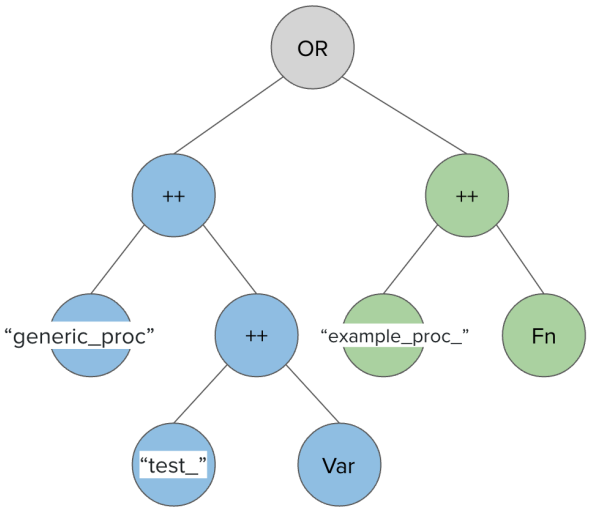

Figure 4: Match tree example

The analysis starts by getting the candidates. In our case, there are two candidates:

1. the `list_to_atom(Id)` function call (Figure 2, line 5)
2. `list_to_atom(get_id(Id))` function call (Figure 2, line 8)

The second step is to build a match tree from the arguments of these function calls. The first candidate does not contain any information on the patterns of the atoms, so the only pattern it is possible to construct is the joker pattern: `.*`, which is not useful. The second candidate is more interesting, as it is an exported function (so more atom creation can be triggered). It uses the `get_id/1` functions. The match tree of `get_id/1` is presented in Figure 4.

The match tree is constructed from the `gen_ids/1` function defined in the Erlang module in Figure 2. From this function call, the match tree on Figure 4 will be built. First, the algorithm detects, that the function has two clauses, so each clause needs to be analysed separately, and the results will have an 'or' relation, so an 'OR' node is introduced. With the help of data-flow reaching, we can see that the first function call ends with a concatenation, so a concatenation node (marked with ++) is introduced. The left child of the concatenation is a constant: `generic_proc_`, and the right child is a variable: `Id`. Another data-flow analysis will tell us, that the value of this variable is a concatenation of the constant `test_` and a variable originated from the function arguments, so a concatenation node will be introduced. The other function clause will be analysed similarly.

From this match tree, the following regular expressions will be constructed: `generic_proc_test_*` and `example_proc_*`. The analysis will associate these regular expressions with the function call `gen_ids/1`.

As a next step, it will match it with the provided atom table. Both of the constructed patterns have a high match score, as they are very specific. The first candidate with the joker (`.*`) pattern has a low match score, as it is permissive.

The result of Algorithm 1 will be candidates with the number of matches and the match score. Then the results are grouped by the match score and ordered by the number of matches. The higher the match score, the more likely the atom table will be filled up. More matches mean more atoms in the current atom table, increasing the chances of filling up the atom table in the future.

The result of the example is shown in Figure 5. The atom generation function call from the `gen_ids/1` function is in Group 14, meaning it got 14 as a match score. The other function call is in Group 1, meaning its match score is only one, and it is less likely to fill up the atom table.

```
(n@host)12> refusr_atom_exhaust:run(AtomTable).
Group: 14
        "list_to_atom(get_id(Id))" (782x)
Group: 1
        "list_to_atom(Id)" (1x)
```

Figure 5: Output of demonstration (atom exhaustion

6. Evaluation

In this section, we will look into a case, where the algorithm which is tasked to find atom creation bottlenecks could drastically reduce the possible atom-extensive calls in the source code of MongooseIM. MongooseIM is an open-source, efficient XMPP server which can be at the core of an Instant Messaging platform [11]. We used Version 6.1.0 in the evaluation[2].

The source was analysed by RefactorErl. Using the security checkers of RefactorErl [5], we were able to identify vulnerabilities which are capable of creating many atoms. The `gen_mod:get_module_proc/2` function can generate numerous atoms, and it is possible to use it in a way to exploit the machine, leading to a node crash, for example, during debugging, or remote node access. We will simulate a debugging session, where we are trying to get multiple host names on a trial-and-error basis.

We applied the following steps:

- Start MongooseIM in a containerised environment
- Check current atom count: `erlang:system_info(atom_count).`
- Run the vulnerable command: `gen_mod:get_module_proc("host1", node())`, and `gen_mod:get_module_proc("host2", node())` and so on. Eventually, this could lead to an atom exhaustion if the first parameter is unique and only access to the shell is needed.
- Find this bottleneck with RefactorErl

Assuming the source code of MongooseIM is added to RefactorErl and it is started. We can now run the following command in the Erlang shell: `refusr_atom_exhaust:run(AtomTable)`, where `AtomTable` is the atom table of the node where MongooseIM runs. This will return the list of functions, where the atoms are probably created.

The first step of the analysis is to get the possible candidates, in this case, MongooseIM, had 55 candidates. The next step is to build the patterns from the function arguments. There are quite a few patterns that are building from MongooseIM, for example: `*_*`, `cyrsasl_*`, `ejabberd_auth_*` and more.

```
true
false
...
base1_myhost
base2_myhost
...
base20_myhost
```

Figure 6: Atom table of MongooseIM

The current state of the atom table can be seen in Figure 6. It is observable that besides the usual and needed atoms such as `true` and `false` (and the names of modules and functions), many atoms are generated dynamically. This is a sign of possible atom exhaustion.

[2]https://github.com/esl/MongooseIM/tree/6.1.0

The next step is to match the patterns with the atom table and calculate the score. We can see that the `base*_myhost` atoms are generated dynamically, which will be matched by pattern: `*_*`. The score of this match is quite low, only 1 because the only criterion is that it should contain an underscore. Even though it is a match with a low score, it matches many atoms, so it is a good result. The analysis will output the function from `gen_mod` module, displayed in Figure 7.

```
get_module_proc(Host, Base) ->
    list_to_atom(atom_to_list(Base) ++ "_" ++ Host).
```

Figure 7: Function that is probably causing the atom exhaustion

Runtime MongooseIM is considered a quite big project, it has 98 568 lines, and there are 55 candidates, where atoms are being generated. The analysis took 3.65 seconds. The complexity of the analysis is based on the number of candidates in the source code and the complexity of the call chain they are used in.

A code base with, for example, 100,000 lines can be analysed faster if it has just a few candidates; on the other hand, a code base with only 10,000 lines might contain many more possible atom creation points (candidates).

Another aspect that could influence the runtime is the complexity of the code that needs to be analysed during the data-flow reaching calculation The more complex it is, the more time it will take. This is an important factor, as this analysis relies heavily on reaching calculation.

7. Related work

Several tools exist that can help to identify bottlenecks in the source code.

Code Compass [12, 13] is an open-source static analyser tool; its main aim is to help understand legacy code bases. It can give exact information on class relations and inheritance information. While its main focus is code comprehension and aiding developers in understanding the code base, it can also help to find bugs, bad designs, and other issues, which can help to improve the code base and avoid possible bottlenecks or fix existing ones.

JProfiler [14] is a dynamic analyser, profiler tool for Java. It can find memory leaks, and it can help developers find and resolve performance bottlenecks. It can also monitor and profile CPU threads, locks and many more concurrency-related issues. It also aims to help developers to understand the performance of their Java applications, and find bottlenecks.

Sobelow [15] is a static analysis tool mainly for the Phoenix web framework [16], which is built on the Elixir [17] programming language, which is a functional language running on the BEAM, just like Erlang. Sobelow is looking for security-related vulnerabilities, such as cross-site scripting, injection, and many more. It doesn't offer performance bottleneck detection, and it is just a static analyser it does not combine static and dynamic analysis.

A research group at National Cheng Kung University and Iowa State University examined bottlenecks in Java systems [18]. They checked hardware data and memory usage data to check for performance bottlenecks in Java systems. They concluded that minor garbage collection can cause performance bottlenecks in the systems.

Aston Parlindungan Tampubolon and others published a paper [19] on the identification of features in source code using regular expressions. In the case of industrial software development, it is not uncommon that source codes are reused from previous projects. This research aims to identify features in the source code, that are reusable. They use predefined regular expressions to identify certain features, such as classes, methods, and namespaces (a more detailed list can be found in their paper in Table 1). We are using an opposite approach: the regular expressions are not defined in advance but are built dynamically and matched on runtime values.

We have not seen other algorithms finding bottlenecks in Erlang source code, based on runtime data in an automated, algorithmic way. Usually, manual analysis is applied which is very time-consuming.

8. Conclusion

Creating complex systems has its challenges, especially concurrent and distributed systems. Software systems can be all over the world, and our lives depend on them more than we could imagine. Due to this, it is essential to write and maintain quality software. Often, developers write code that can introduce bottlenecks that are not easy to catch and become apparent only in production. It is recommended to use monitoring on a production system to catch these issues as soon as possible, and if this system is good enough, it not only catches the issue but provides additional data to the developer to fix the issue. In this paper, We introduced the concept of bottleneck analysis in Erlang. It leverages data available during runtime and enriches static analysis with it.

In the paper, We showed first how atom creation can be a bottleneck because it fills the atom table, and it can lead to a crash. We introduced an algorithm to find these issues and also implemented it in RefactorErl to test it out on real-world examples.

The algorithms that are introduced are static analysis algorithms enriched with data from a dynamic analyser tool. An improvement could be made to connect the static analysis tool with the dynamic analyser tool so an analysis could be run on-demand whenever the monitoring tool raises an alert.

After the analysis has been run, the results could be sent back to the monitoring tool for another iteration of refinement. For example, if the static analysis tool finds a set of functions to be bottlenecks, the dynamic analyser tool could run profiling or tracing on those functions to sort issues by importance.

In the future, we would like to automatise the presented process: set alarms in WombatOAM and automatically call the RefactorErl static analysis engine to filter out the candidates and rank the results. After this, we will extend the framework with other identifiable bottlenecks: mailbox overloading (such as unhandled messages, wrong message tagging), resource overusing (process or port count limit issues), etc.

Acknowledgments

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.

References

[1] S. T. Francesco Cesarini, Erlang Programming, 1st ed., O'Reilly Media, 2009.

[2] Erlang - data types: Atom, 2024. URL: https://www.erlang.org/doc/reference_manual/data_types.html#atom, visited: 2024-05-14.

[3] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. Víg, T. Nagy, M. Tóth, R. Király, Modeling semantic knowledge in erlang for refactoring, Studia Universitatis Informatica 54 (2009) 7–16.

[4] M. Tóth, I. Bozó, Static Analysis of Complex Software Systems Implemented in Erlang, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 440–498. doi:10.1007/978-3-642-32096-5_9.

[5] M. Tóth, I. Bozó, Supporting secure coding for erlang, in: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC '24, Association for Computing Machinery, 2024, p. 1307–1311. doi:10.1145/3605098.3636185.

[6] P. Trinder, N. Chechina, N. Papaspyrou, K. Sagonas, S. Thompson, S. Adams, S. Aronis, R. Baker, E. Bihari, O. Boudeville, F. Cesarini, M. D. Stefano, S. Eriksson, V. fördős, A. Ghaffari, A. Giantsios, R. Green, C. Hoch, D. Klaftenegger, H. Li, K. Lundin, K. Mackenzie, K. Roukounaki, Y. Tsiouris, K. Winblad, Scaling reliably: Improving the scalability of the erlang distributed actor platform, ACM Trans. Program. Lang. Syst. 39 (2017). doi:10.1145/3107937.

[7] M. Henoch, Can I get a list of all currently-registered atoms?, https://stackoverflow.com/questions/13480462/can-i-get-a-list-of-all-currently-registered-atoms/34883331#34883331, 2016. Visited 2024-03-24.
[8] Erlang – External Term Format — erlang.org, https://www.erlang.org/doc/apps/erts/erl_ext_dist.html, 2023. Visited 2024-03-24.
[9] B.-E. D. <egil(at)Erlang.org>, Eep 0043 - Erlang/OTP — erlang.org, https://www.erlang.org/eeps/eep-0043.html, 2013. Visited 2024-03-24.
[10] B. Baranyai, Funkcionális nyelvek és a statikus kódelemzéssel támogatott biztonságos szoftverfejlesztés, TDK thesis, 2020.
[11] E. Solutions, MongooseIM — esl.github.io, https://esl.github.io/MongooseDocs/latest/, 2024. Visited 2024-03-19.
[12] Z. Porkoláb, T. Brunner, The codecompass comprehension framework, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Association for Computing Machinery, 2018, p. 393–396. doi:10.1145/3196321.3196352.
[13] A. Fekete, M. Cserép, Z. Porkoláb, Measuring developers' expertise based on version control data, in: 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), 2021, pp. 1607–1612. doi:10.23919/MIPRO52101.2021.9597103.
[14] ej-technologies GmbH, Java Profiler - JProfiler, 2024. URL: https://www.ej-technologies.com/products/jprofiler/overview.html, visited: 2024-05-16.
[15] The Sobelow Team, Sobelow documentation, https://hexdocs.pm/sobelow/readme.html, 2024. Visited: 2024-05-23.
[16] The Phoenix Team, Phoenix framework, https://www.phoenixframework.org/, 2024. Visited: 2024-05-23.
[17] The Elixir Team, Elixir documentation, https://elixir-lang.org/docs.html, 2024. Visited: 2024-05-23.
[18] K.-Y. Chen, J. M. Chang, T.-W. Hou, Multithreading in java: Performance and scalability on multicore systems, IEEE Transactions on Computers 60 (2011) 1521–1534. doi:10.1109/TC.2010.232.
[19] A. P. Tampubolon, B. Hendradjaya, W. D. Sunindyo, Feature identification of program source code using regular expression, in: 2016 International Conference on Data and Software Engineering (ICoDSE), 2016, pp. 1–6. doi:10.1109/ICODSE.2016.7936133.