



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF PROGRAMMING LANGUAGES AND COMPILERS

Finding code bottlenecks by combining static and dynamic analysis

Supervisor:

Dr. Melinda Tóth, Dr. István Bozó

Senior research fellow, Research fellow

Author:

Róbert Fikó

Computer Science MSc

Budapest, June 5, 2024

Contents

1	Introduction	4
1.1	Static analysis	4
1.2	Dynamic analysis and operational management	5
1.3	Summary of thesis	6
2	Background	7
2.1	Erlang	7
2.1.1	Dynamic typing	8
2.1.2	Concurrency	8
2.2	RefactorErl	10
2.2.1	Semantic Program Graph	10
2.2.2	Data-flow and Control-flow analysis	12
2.2.3	Process and message analysis	13
2.3	WombatOAM	13
2.3.1	Architecture	13
2.3.2	Features	13
2.3.3	Web interface	14
3	Identifying bottlenecks in the source code	15
3.1	Methodology	16
3.2	Atom exhaustion	17
3.2.1	Motivational example	17
3.2.2	Getting the atom table	18
3.2.3	Definitions	19
3.2.4	Analysis	21
3.2.5	Demonstration of the algorithm	22
3.3	Message passing	24
3.3.1	Motivational Example	24

3.3.2	Analysis	25
3.3.3	Definition of the steps of the algorithms	28
3.3.4	Demonstration of the algorithm	31
3.4	Process counts	32
3.4.1	Motivational Example	32
3.4.2	Analysis	34
3.5	Port counts	35
3.5.1	Motivational Example	35
3.5.2	Analysis	37
4	Evaluation	38
4.1	Atom exhaustion in MongooseIM	38
4.1.1	Preparing the environment	38
4.1.2	Scenario: debugging MongooseIM	39
4.1.3	Additional step: WombatOAM, monitoring	40
4.1.4	Runtime	41
4.2	Message queue trailing in Mongoose IM	41
4.2.1	Preparing the environment	41
4.2.2	Scenario: Authentication	42
4.2.3	Runtime	44
5	Related	45
5.1	Tools	45
5.2	Research	47
6	Conclusion and future work	49
6.1	Future work on bottleneck analysis	49
6.2	Future work on dynamic analysis integration	50
	Acknowledgements	51
	Bibliography	52
	List of Figures	56
	List of Algorithms	57

List of Codes	58
---------------	----

Chapter 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. Stopping them is simply not acceptable, except in very necessary scenarios. Because of this, a crash of these systems is just not acceptable.

Most of these systems receive their updated source code through hot code reloading, which means, replacing the existing source code on the machine while it is running and serving users. Over time, these issues can really grow into larger issues. By nature, there are some kinds of bugs and issues that go undetected during unit and system testing and only appear in production systems. This brings in the importance of monitoring. A good monitoring (dynamic analyser) tool can give the developers early signals of issues in the system. Besides the alarm, these systems have 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 only one tool can not point you to the exact issue in the source code. That is the reason why, in my thesis, I combined static and dynamic analysis to be able to find bottlenecks in Erlang [1] source codes more easily.

1.1 Static analysis

Static analysis is the analysis on the source code of some software, without the execution of it. It is widely used in software development, as it is a great support tool for developers and engineers.

Such a tool is often backed by University research, has many publications, and the implementation relies on well-founded scientific research. A static analysis tool can be specific or more general. A few examples of static analysis tools are:

For Erlang

- **Dialyzer** is a discrepancy¹ analyser. It looks for inconsistencies in the source code such as type errors, undefined function call, dead or unreachable code. It is part of the Erlang/OTP distribution. Dialyzer is a good example of a

¹difference or inconsistency

specific static analysis tool, as it is capable of analysing typing information and function relations, it is not able to detect, for example, issues related to memory [2, 3].

- **Gradualizer** is a static type checker tool, which some support for gradual typing [4, 5, 6]. It aims to integrate into existing source codes in a non-intrusive way. It is also a specific static analysis tool [6].
- **infErl** is a general static analysis tool, but it focuses on typing issues and concurrency issues [7].
- **RefactorErl**: is a static general purpose static analyser and refactoring tool, it has many features, such as code comprehension, security checks and many more, which will be discussed in Chapter 2.
- **Concuerror** is stateless model checking analyser. It is checking for concurrency issues, errors and can spot errors that can only occur on a few specific schedulings [8, 9].

For other languages

- **SonarQube** is a general purpose static analyser, developed by SonarSource. It is capable of analysing many languages, such as Java, C#, C++, Python, and many more. It is capable of detecting many issues, such as code smells, bugs, security vulnerabilities, and many more [10].
- **ESLint** is a static analysis tool for JavaScript. It identifies problematic patterns in the source code. It integrates well with many editors, and it is also working for TypeScript [11].
- **ReSharper** is a static analyser for C#. It is a plugin for Visual Studio. ReSharper is capable of detecting many issues, such as code smells, bugs, and many more [12].
- **Code Compass** is an analyser tool, with a plugin architecture for C++ and Java. It has a nice graphical interface, with good insights such as: call path visualisation, inheritance and aggregation diagrams [13, 14].

1.2 Dynamic analysis and operational management

Dynamic analysis is the analysis of the software while running it. Common tools are, for example, unit and other testing tools, debuggers and coverage tools. Less well-known but used tools are different execution tracking and memory usage detecting tools, and operation management tools are part of this category.

An operation and monitoring tool can detect memory errors, flaws, increased CPU usage, and other parts of the system that are not working as expected. These tools are often attached to production systems, to provide the needed visibility. Monitoring tools are also capable of sending alerts to the development or operation team when something is not working as expected.

For Erlang

- **eprof** is a time profiling tool, shipped with the Erlang/OTP package. Its main aim is to find out how the execution time was used. It works by tracing and timing function calls. Expect a system to slow down when used. [15]
- **Observer** is a graphical tool which displays important information about the system, such as application supervision trees, process data, and information about different tables and databases in Erlang. [16]
- **WombatOAM** is a commercial monitoring, operational and management tool, which is capable of gathering metrics from the system, raising alarms if certain thresholds of the system are met, and many more which will be discussed further in detail in a later chapter, in Section 2.3. [17, 18]

For other languages

- **JProfiler** is a Java profiler tool. It is good in finding memory leaks, and performance issues. It works well with Java frameworks, and requires minimal to no configuration. It adds low overhead to the Java Virtual Machine. [19].
- **Py-Spy** is a Python sampler profiler. It adds really low overhead to the observed system and it can give you information about where the Python code is spending time. [20]

1.3 Summary of thesis

In my thesis, I combined static and dynamic analysis for Erlang to find a more efficient way to find source code bottlenecks.

In Chapter 2, I introduce the background of the topic, and the tools that are used in the thesis. In Chapter 3, I describe four different bottleneck families, and related algorithms for static analysis to find them in Erlang.

One of the bottleneck type related to dynamic atom creation. Atoms are constants with a name (other types will be defined later). Atoms are 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. The thesis introduces further issues and analyses.

In Chapter 4, I evaluate the work on open-source projects that are widely used. Chapter 5 is about related work, discussing the state of the art in the field of static and dynamic analysis, and the combination of them. Chapter 6 is about drawing conclusions and discussing future works on the topic.

Building on the knowledge and algorithms, that this thesis introduces, it is possible to extend the analysis to many other types of issues as well.

Chapter 2

Background

Before diving into the details of the thesis, it is important to understand the background of the technologies and tools that are used in this work. This chapter will introduce the reader to the basics of Erlang, RefactorErl and WombatOAM.

2.1 Erlang

Erlang is a general-purpose functional programming language and development platform. An Erlang software is divided into modules.

```
1 -module(demo).  
2 -export([get_id/1]).  
3  
4 get_id({tool, Name}) when is_list(Name) -> list_to_atom("tool_" ++  
    Name);  
5 get_id({user, Name}) when is_list(Name) -> list_to_atom("user_" ++  
    Name);  
6 get_id(Name) when is_list(Name) -> list_to_atom("other_" ++ Name);  
7 get_id(_) -> undefined.
```

Code 2.1: Example Erlang module

In Code 2.1, we can see an example Erlang module. The module is defined with the `module` attribute. The module has one function called `get_id/1`, which takes one argument. As we can see, Erlang functions can take arguments which are, in fact, also patterns. An Erlang function can have multiple clauses, which are selected based on pattern matching. The clauses are checked from top to bottom, and the first matching clause is executed. If no clause matches, an error is thrown.

In Code 2.1, the `get_id/1` function has four clauses. Consider a scenario where we want to generate IDs for tools and users, but we prepare the system for some other entities as well. The first clause of `get_id/1` function matches a tuple, where the first element is the atom `tool`, and the second element is the `Name`. The body of the function is defined after the arrow `->`. In this example, it concatenates the string `"tool_"` with the value of the `Name` argument, then with the help of the `list_to_atom` function, it converts the string to an atom.

The second clause is similar to the first one, but it matches a tuple, where the first element is the atom `user`, and concatenates the string `"user_"` with the value of the `Name` argument, before converting it to an atom.

The third has a single argument without any specific patterns. It concatenates the string "other_" with the value of the `Name` argument before converting it to an atom.

Notice that all three previous clauses have a guard, which is introduced by the `when` keyword. The guard should be a boolean expression, which needs to be true for the clause to be executed. Here the guard checks if the `Name` argument is a list. If it is not, the clause will not be executed.

The fourth and last clause is a catch-all clause, which will be executed if none of the previous clauses matched. It returns the atom `undefined`.

2.1.1 Dynamic typing

Erlang is a dynamically typed language, which means that the type of a variable is determined at runtime. The basic types that will be used in this thesis are:

- `atom()`: a constant with a name
- `binary()`: a sequence of bytes, denoted by `<<>>`
- `char()`: a single character
- `fun()`: a function, denoted by `fun() -> ... end`, also called a lambda

There are compound types, like tuples, lists, and maps. A tuple is a fixed-size collection of elements, with a constant time to access its elements. A list is a linked list of elements, so to access the *n*th element, it takes *n* steps. A map is a key-value store where the key can be any term, and the value can be any term.

Atoms will be especially important in this thesis, as one of the analyses (Section 3.2) will find source code fragments that are responsible for filling the atom table. An atom is a constant with a name. Atoms are stored in the atom table, which has a predefined size and cannot grow over it. Atom tables are not garbage collected, so it is important not to create atoms dynamically, as it can lead to a crash.

2.1.2 Concurrency

Erlang is designed for fault tolerance and concurrency. It is heavily used in the Telecommunication industry, where high availability is crucial. Erlang has a built-in support for concurrency, which is achieved by lightweight processes. Each and every process has its own memory and is isolated from other processes. Processes communicate with each other by sending messages.

```
1 -module(msg_pass).  
2 -export([demo/0]).  
3  
4 demo(Name) ->  
5     SelfPid = self(),  
6     spawn(fun() ->  
7         % send a message to the process  
8         SelfPid ! hello  
9     end).  
10  
11 % parent process receive the message
```

```
12 receive hello -> ok end.
```

Code 2.2: Message passing in Erlang

In Erlang, we use the `spawn` function to create a new process. It creates a new process and returns the process identifier. The `self()` function call returns the process identifier of the current process, and the `!` operator is used to send a message to a process. The `receive` expression is used to receive messages.

In Code 2.2, the `demo/1` function creates a new process with the `spawn/1` function call. The process sends a message to the parent process, which receives the message with the `receive` construct.

The above-mentioned example (Code 2.2) is a simple example of message passing in Erlang. Since data is immutable and processes are isolated, a process cannot modify the data of another process, so the only way they can communicate and exchange data is by sending messages.

When a process sends a message to another process, it will arrive in their message queue. The receive process can read it using the `receive` expression. The messages are processed in the order they arrived. It is possible to process messages using pattern matching in the receive expression selectively. The receive expression, will take the first message out, which matches the pattern, and will process it. If there is no matching message, the process will wait until a matching message arrives, though it is possible to set a timeout.

When not properly designed, messages can queue up in the message queue, which takes up memory.

```
1 -module(msg_pass_2).
2 -export([demo/0]).
3
4 demo(Name) ->
5     SelfPid = self(),
6     register(important_proc, SelfPid),
7     spawn(fun() ->
8         % send a message to the process
9         [important_proc ! {not_ok, X} || X <- lists:seq(1, 1000)],
10        important_proc ! hello
11    end),
12
13    % parent process receive the message
14    receive hello -> ok end.
```

Code 2.3: Message passing in Erlang

For example, in Code 2.3, the process sends 1000 messages to the parent process before sending the `hello` message. The issue is that the receive block needs to iterate through all the messages before it can process the `hello` message that we are interested in. This can be a bottleneck in the system if not handled properly.

Note: the process is registered with the `register` function, so it can be accessed by its name also. In Erlang, there cannot be two processes with the same name, so they need to be unique.

Erlang is often mentioned with OTP, the Open Telecom Platform, as it is a fundamental part of its ecosystem.

Erlang/OTP was developed by Ericsson in the 1980s, and its telecommunication needs drove it. Despite the original intention to develop it for telecommunications,

it is not specialized for this domain. It provides support to overcome technical challenges, such as concurrency, scalability, fault-tolerance and high availability [21].

2.2 RefactorErl

RefactorErl is a static analysis tool (Section 1.1), developed at Eötvös Loránd University in a research-development project. It is written in Erlang, for Erlang. 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.

A notable feature of RefactorErl is the security checkers, which can help to identify vulnerabilities in Erlang sources [22]. This thesis will also rely on its implementation.

2.2.1 Semantic Program Graph

RefactorErl represents the information in the semantic program graph (SPG). The graph 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 lexical layer contains the source code split into tokens. These nodes have properties like text, which represents the original text of the token, and type, which represents the type of the token. It also has a value which is the value of the token.

In the syntactic layer, there are nodes that can be files, forms, clauses and expressions. File represents a physical file from the system. Forms are units of files, for example an Erlang source code is a collection of forms, each form is terminated by a dot. A clause is a list of expressions, they usually represent clauses of functions. Expression nodes, are representing single expressions or patterns.

In the semantic layer, information is represented by semantic links between syntactic and semantic nodes. This information can be about modules, variables, functions and records.

Links or edges have an associated tag based on the type of the relation. For the same type of link tag, the starting node class and the ending node class should be the same.

The Semantic Program Graph is how RefactorErl represents Erlang sources and systems. It encodes useful information, enabling the tool to do many analyses and refactoring [23].

Example

The example module in Code 2.1 is added to RefactorErl, and the visualisation of part of the semantic program graph is shown in Figure 2.1 and 2.2.

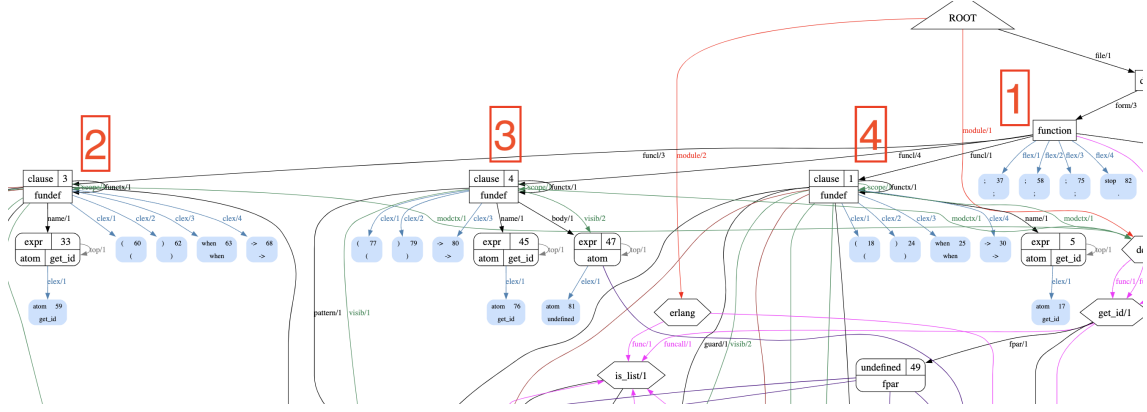


Figure 2.1: Example of the Semantic Program Graph

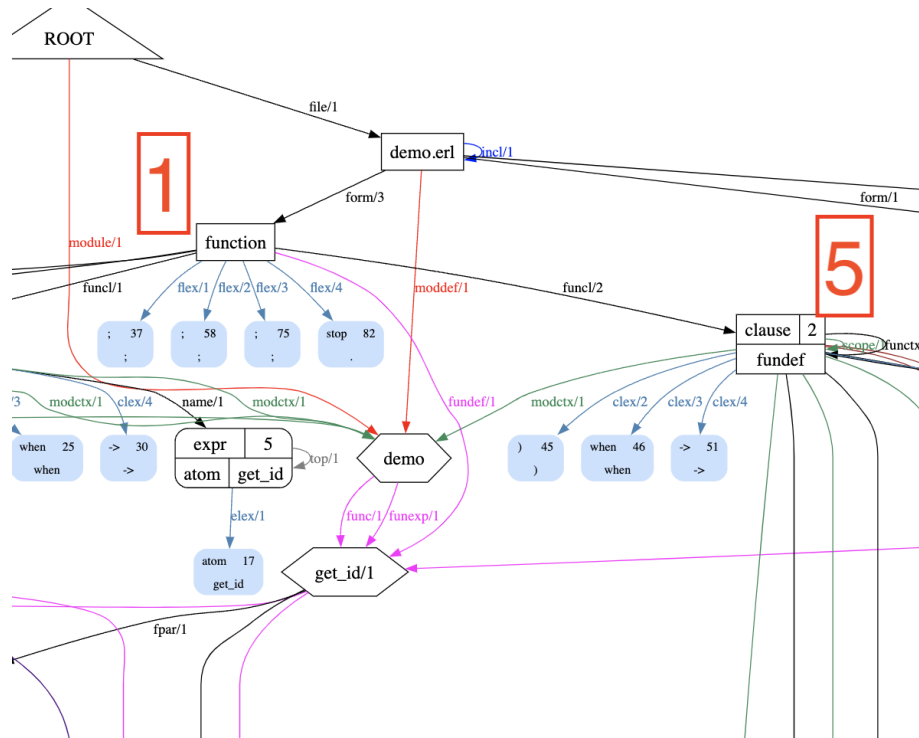


Figure 2.2: Example of the Semantic Program Graph

On Figure 2.1 and Figure 2.2 we can see parts of the SPG. With red 1, I marked the function form node, and we can see that it has links to the clause nodes. Clauses marked with red 2, 3 and 4 are visible in Figure 2.1, while the clause marked with red 5 is visible in Figure 2.2.

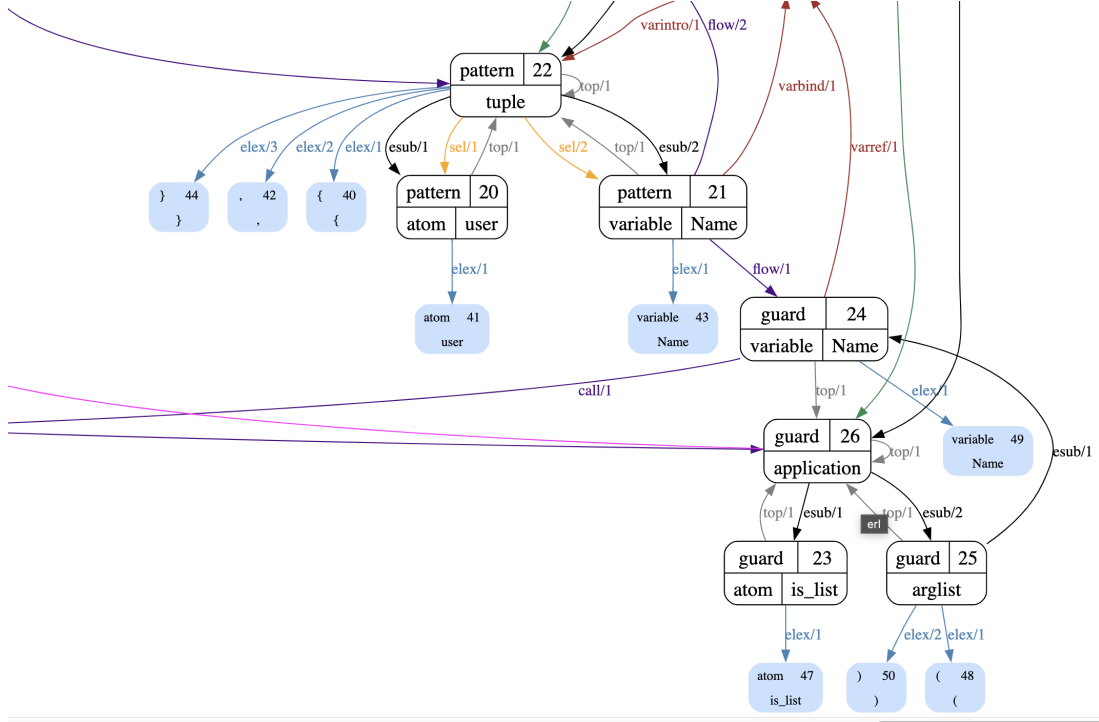


Figure 2.3: Example of the Semantic Program Graph

In Figure 2.2, we can see that a lot of edges are coming out from the function clause node, marked with a red 5 and with a node id of 2. Figure 2.3 shows the patterns and guards in the function clause. The black link on the top, which points to the pattern node with id 22, is a pattern edge in the SPG. This is the pattern of the function clause.

2.2.2 Data-flow and Control-flow analysis

Data-flow and control-flow analysis are two important analysis techniques in static analysis, both are part of the RefactorErl tool, and data-flow analysis will be heavily used in the thesis.

Data-flow analysis is gathering information about how an Erlang program changes and manipulates data and what are the set of possible values of it [24].

In the reaching analysis, we are interested in which values can reach certain points of the program. We can use this information to find out what are the possible values of a variable at a certain point in the code [24].

The analysis builds up the Data-Flow Graph (DFG), which becomes part of the SPG. The analysis enriches it with data-flow links. Because DFG is part of the SPG, it is also a directed, labelled graph. It adds different kinds of edges to the graph: flow, constructor, selector and dependency edges [24].

The Control Flow Graph (CFG) can be built on the top of the SPG. It represents all possible execution paths. It adds conditional branching and pattern-matching edges to the graph. It also extends the SPG with return edges, which return to the previous edges. Function calls, receive, and send expressions are also denoted by labelled edges.

2.2.3 Process and message analysis

As mentioned in Section 2.1.2, processes communicate with each other by sending messages. RefactorErl can analyse the message passing in Erlang sources. It adds new edges to the Semantic Program Graph (SPG) [25]:

- In Erlang processes are created with the `spawn` or `spawn_link` functions. We add `spawn` or `spawn_link` edges to the SPG.
- There is an option to register a process with a name. It is useful when the process has an important role. We can do this using the `register` function. In this case, we add a `register` edge to the SPG.
- When two processes communicate with each other, they send messages. A `send` edge is added to the SPG, pointing from the sender to the receiver process.
- The DFG of RefactorErl is extended with `flow` edges between sent messages and receive patterns.

RefactorErl can detect these relations, which will be useful during the bottleneck analysis [25].

2.3 WombatOAM

Wombat Operation, Administration and Maintenance (short: WombatOAM), is a dynamic analysis, monitoring tool (Section 1.2) developed by Erlang Solutions. It has a closed source, and it is a paid tool, but a free trial can be requested from the website of Erlang Solutions¹.

2.3.1 Architecture

The WombatOAM is a standalone server, which is designed to operate with one, more or no observed nodes and communicate via the Erlang distribution [18]. WombatOAM can even communicate with Erlang nodes running different versions of Erlang releases. These managed nodes can be a simple standalone Erlang nodes or nodes running open-source or commercial Erlang/OTP applications.

2.3.2 Features

WombatOAM can monitor and measure different metrics of Erlang and Elixir systems. Some of its features:

- **Metrics:** Wombat supports a wide range of metrics, for instance, number of processes, number of ports, memory usage, message queue length, and number of atoms.
- **Alarms:** Wombat can send out 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.

¹<https://www.erlang-solutions.com/technologies/wombatoam/>

WombatOAM can do this by starting lightweight agents on the observed node. These agents collect the data and send it back to the WombatOAM node, which processes and stores it. Using this mechanism, WombatOAM can monitor the node without causing a significant overhead.

2.3.3 Web interface

WombatOAM also has a web interface to check the metrics, alarms and notifications. It is also possible to create custom dashboards to visualise the metrics. In Figure 2.4, we can see an example of the web interface of WombatOAM with the atom count metric. [18, 17]

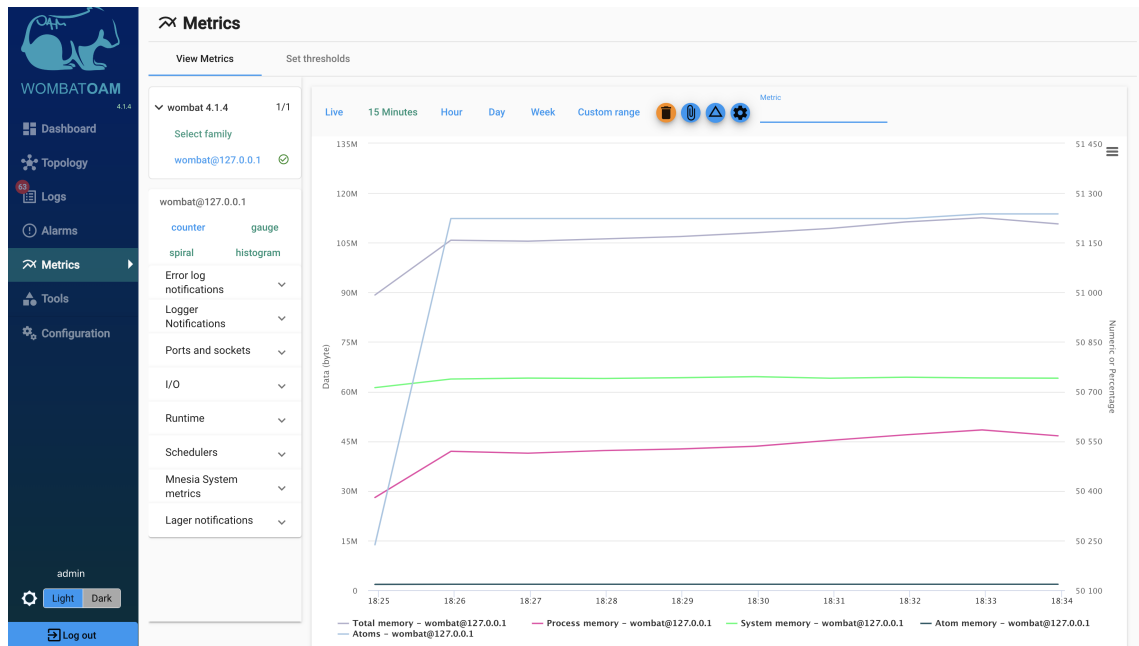


Figure 2.4: WombatOAM showing many metric, including atom count

Chapter 3

Identifying bottlenecks in the source code

Nowadays, our systems are running constantly without the need to shut down these long-running applications during source code updates. That is the reason why quality assurance in the field of software engineering is a key point. In my work, I aim to find certain bottlenecks in the source code. Even though I do not present preemptive solutions, rather solutions that help the developers and engineers when the system starts to show symptoms of an issue.

This chapter aims to define some of the issues of Erlang runtime systems that can be detected by dynamic analysers and monitoring tools. Static analysis can help find the source of the issues.

This way, running on-demand static analysis focused on a certain issue is possible when the system starts showing symptoms. The result of the static analysis, which is run this way, may result in a more narrowed-down result set.

Chapters 1 and 2 already introduced the dynamic and static analysers, so we will build from there. In most cases, data can be collected from the monitoring tool, which helps the static analysis to give more precise results with fewer false positives.

The synergy of static and dynamic analysis is a powerful tool, as they can complement each other. The application and use of the methods described in this thesis can contribute to the quality of the source code.

This chapter will discuss the advantages of combining dynamic and static tools, through examples. Real results will be discussed in Chapter 4.

The following issues can be detected by the combination of the two methods:

- **Atom exhaustion:** It can be a serious issue, as the atom table is not garbage collected, and has a fixed size, so overusing it, can result in a node crash. The size of the atom table can be monitored, and the content of the atom table can be extracted from the system, which can help filter the results during static analysis. This will be discussed in Section 3.2.
- **High message queue:** Although messages of a process are not limited in number (like the atom table), a high message queue can be a bottleneck in the system, as it can slow down the processing of messages and the process itself, as messages are processed in the order they arrive. A receive block will take out the first matching message from the queue. If it needs to check many messages before it, it will take time. The size of the message queues of the

processes can be monitored, and the messages in the queue can be extracted from the system, which can help reduce the results during static analysis. This will be discussed in Section 3.3.

- **Process count:** The total number of processes that are allowed in the system is already defined in the system limits [26]. The number of processes can be monitored, and when it reaches a certain limit, an alarm could be sent out to run a static analysis. The only data we can extract from the system is the function that was used to initiate the process, and the current function of the process, which can help refine results during static analysis. This will be discussed in Section 3.4.
- **Port open:** This issue is similar to the process count one. The number of ports is a predefined system limit [26]. The number of ports can be monitored, and when it reaches a certain limit, an alarm could be sent out to run a static analysis. In this case, we are looking for opened ports that are so-called long-living ports, so they have been open for a long time. This will be discussed in Section 3.5.

3.1 Methodology

To be able to detect the issues that I have mentioned above, I combined static and dynamic analysis. Figure 3.1 shows the methodology and infrastructure that I established to detect the issues.

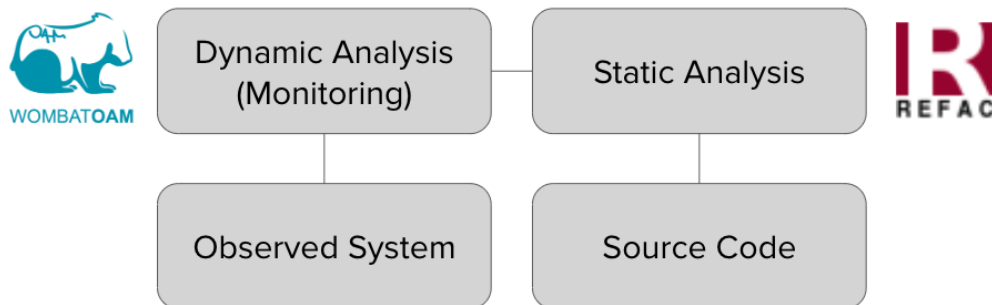


Figure 3.1: Methodology

The monitoring tool is directly connected to the observed system. This monitoring system constantly checks certain system metrics, and if one (or more) reaches a predefined threshold, it raises an alarm. Unfortunately, an alarm cannot give a location in the source code, so I connected the dynamic analysis (monitoring) tool and the static analyser.

The static analyser has information about the source code, so it can find the locations of the issues. The monitoring tool can send additional data to the static analyser, which helps filter the issues so developers can focus on more probable places of issues.

I used WombatOAM (introduced in Section 2.3) for monitoring and RefactorErl for static analysis.

3.2 Atom exhaustion

The Erlang Runtime System has many types of expressions, and one of them is an atom. According to the Erlang documentation, "an atom is a literal, a constant with name" [27]. The text value of every unique atom is stored in the atom table, whose size is limited, and defined during node startup. By default, it has 1,048,576 [27] 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 system. Consequently, it can crash if no monitoring is applied to the node.

In this analysis, I aim to find the source code fragments, where atoms are created leading to the atom table filling up. To achieve this goal, we need the current atom table (in a string representation, so the atoms will not be filling up the atom table of the analysis tool), and the source code analysed by RefactorErl, in the Semantic Program Graph representation.

With the results of this analysis, source code fragments can be identified, where the atom table can be filled up so the developers can refactor and fix the source code to avoid this issue.

3.2.1 Motivational example

The following example in Code 3.1 will demonstrate how the atom table can be filled up.

```
1 -module(atoms).  
2 -export([gen_ids/1, convert_ids/1]).  
3  
4 convert_ids(List) ->  
5     [list_to_atom(Id) || Id <- List].  
6  
7 gen_ids(List) ->  
8     [list_to_atom(get_id(Id)) || Id <- List].  
9  
10 get_id(Id) when is_list(Id) ->  
11     Format = "test_" ++ Id,  
12     "generic_proc_" ++ Format;  
13  
14 get_id(Id) when is_binary(Id) ->  
15     "example_proc_" ++ binary_to_list(Id).
```

Code 3.1: Example on possible atom exhaustion

In Code 3.1, we have a module called `atoms`, with an exported function `gen_ids/1` and `convert_ids/1`. Function `gen_ids/1` takes a list of strings and creates atoms from them.

Now I would like to focus on the function `gen_ids/1`, because when it is called, it will iterate on the list, and for each element, it will call the `get_id/1` function. This function has two clauses, one for lists and one for binaries. If the `Id` is a list, it will create an atom with the `generic_proc_` prefix. If it is a binary, it will create an atom with the `example_proc_` prefix.

If this function is exported, which it is, it can be called from other modules, and it can create a lot of atoms, which can fill up the atom table.

If this source code is executed in a wrong and unsupervised way, one can notice that the atom table can be filled up with atoms like `generic_proc_test_1`, `generic_proc_test_2`, `example_proc_1`, `example_proc_2`, and so on.

If we examine the atom table, we can see that many atoms matching 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.

The aim of this analysis is 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.

Note: In Code 3.1, there is another function, `convert_ids/1`, which is only there to demonstrate, that there is a function that converts a list of strings to atoms. This function is very general function we cannot really deduce any information or pattern from it. The only pattern which we are able to construct is the `*` pattern, which is not really useful.

3.2.2 Getting 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 [28], and an Erlang term encoded in it (starts with byte 131 [28]), which contains a type byte 75 which is not described in the official documentation of the external term format [28, 29].

According to EEP-43 [30], which mentions all the possible types, we can find that `ATOM_INTERNAL_REF3` (75) is the needed type byte [28]. With this knowledge, we can create a binary: `<<131, 75, 0:24>>`.¹ Generally, we can define a function that returns a n-th atom from the atom table.

```
1 -spec get_nth_atom(integer()) -> atom().
2 get_nth_atom(N) ->
3   binary_to_term(<<131,75,N:24>>).
```

Code 3.2: Get n-th atom[28]

There is no simple and convenient way to get the current atom table, but it is possible to keep querying the atom table until there is a bad argument error. From this point, I will switch to an algorithmic notation, as the implementation is only crucial in Code 3.2, to showcase how to retrieve the n-th atom.

I introduce a pseudo-function called `get_nth_atom(n)`, which returns the n-th atom from the atom table. In case the given index parameter (n) is not valid, the Erlang implementation would return a `badarg` (bad argument) exception. To avoid exception management in the pseudo representation, we introduce a function `is_successful(expression)`, which returns true if the parameter is a result of execution without exception.

¹131 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

Algorithm 1 Get the atom table from an index

```

Funct get_atoms_from_index(index)
1: atom := get_nth_atom(index)
2: if is_successful(atom) then
3:   atom_table = get_atoms_from_index(index + 1)
4:   return concat(atom, atom_table)
5: else
6:   return []
7: end if

```

Now we define an Algorithm 1, which can recursively query the atom table until it receives an exception.

- It has one input: an integer, representing the index which from it needs to return the atoms.
- Then it tries to get the *nth* atom, if it is successful the recursive call gets the next atom, if not it means that the whole atom table was queried, so an empty list is returned.
- In the last step, the current atom is concatenated to the already known atoms with the help of the `concat(item, list)` function.

Note: In the above-mentioned representation, it would return the atom table as atoms. 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.

3.2.3 Definitions

In the beginning, I will define some terms that will be useful throughout the thesis. When the dynamic analysis tool creates an alarm about a system metric going over a certain threshold, it is also capable of providing additional data for the static analysis. For example, in the case of atom exhaustion analysis (Section 3.2) the current atom table, or in the case of message passing analysis (Section 3.3) the message queue.

In both cases, these are data represented in string, and we need a way to match the Erlang terms and expressions found by static analysis with these data. For this purpose, I will introduce **match tree** in Section 1, which will be created from Erlang expressions and can be converted further to regular expressions.

The **match tree** provides an intermediate representation of the Erlang terms that the static analysis finds. As I mentioned earlier, it 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.

Definition 1. Match tree: Match trees are used to create regular expressions. It is a binary tree containing constructs useful for matching. It is similar to an abstract syntax tree but much simpler. It has the following nodes:

- **string**: when the value of certain parts is known, it needs to be that.
- **var**: a certain text part needs to be some of the provided values. In case no value is provided, any text can fit in.
- **concatenation**: when texts needed to be evaluated during the regular expression matching as concatenated texts. Has two children, and each can be a sub-tree.
- **option**: when certain parts of the text can have multiple values. This node has two children, and each can be a sub-tree.

A match tree is built from an Erlang expression in the following way:

- String node is used, where the Erlang expression is a constant string. For example `"hello"`. This would create a string node, with the value `hello`:
`{str, "hello"}`
- Var node is used, where the Erlang expression is a variable. For example `A`. This would create a var node, with the possible values of `A` as a list. If we know the possible values, for example it can be `hello` or `world`, it would create a var node, with the possible values `hello` and `world`:
`{var, ["hello", "world"]}`.
- Concatenation is used, where the Erlang expression is a concatenation of two expressions. For example: `<<"hello", "world">>` or `"hello" ++ "world"`. This would create a concatenation node, with two string nodes as children:
`{'++', {str, "hello"}, {str, "world"}}`.
- Option is used, where the Erlang expression is a list of possible values. For example in cases of function clauses. If we know that a function has multiple cases, and we can evaluate the possible values of it, for example `foo()` can have an evaluation of 1 or 2. This would create an option node, with two children, one for each possible value: `{'|||', {var, [1]}, {var, [2]}}`.

These match trees can be turned into regular expressions, which can be used to match, for example, atoms in the atom table or messages in the message queue. It is turned into regular expressions with the following rules:

- In the case of a concatenation node, the regular expression is the concatenation of the regular expressions of the children.
- In the case of an option node, the regular expression is the regular expression of the children, with a `|` separator.
- In the case of a string node, the regular expression is the string itself.
- In the case of a var node, the regular expression is the possible values of the var node, with a `|` separator.

Definition 2. Match score: It 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

3.2.4 Analysis

The analysis requires the current atom table (Section 3.2.2) as an additional input next to the analyzed source code in a semantic program graph representation.

The atom creation analyser (Algorithm 2) starts by getting the function calls that are potentially creating many atoms. I will call these candidates. The following steps are responsible for filtering out the ones that might create atoms, but not that much, so we will remain with the source code fragments, which are worth our attention. In order to achieve this, the algorithm will iterate over the candidates and will do the following steps:

- It gets the arguments of the calls, as the information for the pattern and the match building is there
- Then, it builds the so-called match tree (see Definition 1)
- The next step is to convert the match tree into a regular expression
- Finally, it counts the matches on the atom table.

Algorithm 2 Atom exhaustion source code fragments, based on atom table

Func `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 (Definition 2).

As a next step, the calls that have zero matches will be dropped from the result. The remaining results will be grouped based on their match score (see Definition 2). The remaining result is then ordered based on this score.

Note: the implementation of Algorithm 2 uses RefactorErl’s security library [22, 31] to identify the vulnerable expressions where atoms are created.

3.2.5 Demonstration of the algorithm

Let us consider the following example, where the atom table can be filled up. Following on the motivation introduced in the beginning of this section, we have an Erlang VM with the following atom table (Code 3.3), running the source code introduced in the example (Code 3.1). It is observable that we have a lot of atoms that can come from a common pattern.

```
1 false, true,
2 ...
3 generic_proc_test_1,
4 generic_proc_test_2,
5 ...
6 generic_proc_test_350,
7 example_proc_1,
8 example_proc_2,
9 ...
10 example_proc_432,
11 process_registry,
12 ...
```

Code 3.3: Example atom table

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

1. the `list_to_atom(Id)` function call in line 5, in Code 3.1
2. `list_to_atom(get_id(Id))` function call in line 8, in Code 3.1

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 really useful. The second candidate is more interesting, as it is possible to construct the match tree presented in Figure 3.2.

The match tree is constructed from the `gen_ids/1` function defined in the Erlang module in Code 3.1. This function is exported, so it is possible to use it to create many atoms. It uses the `get_id/1` function, which creates an atom either with the `generic_proc_test_` prefix if the `Id` is a string (list), or `example_proc_` prefix if it is binary.

The match tree algorithm will build the tree visible in Figure 3.2. First the algorithm will detect, that the `get_id/1` function has two clauses, so each clause need to be analysed separately. The results will have an ‘or’ relation, so an ‘OR’ node is introduced. Then the first function clause of `get_id/1` is analysed further, and the algorithm will detect with the help of data-flow reaching, that it has a concatenation in it, so it will introduce a so-called concatenation node (marked with ++ in Figure 3.2). The left child of the concatenation is a constant: `generic_proc_`, and the right

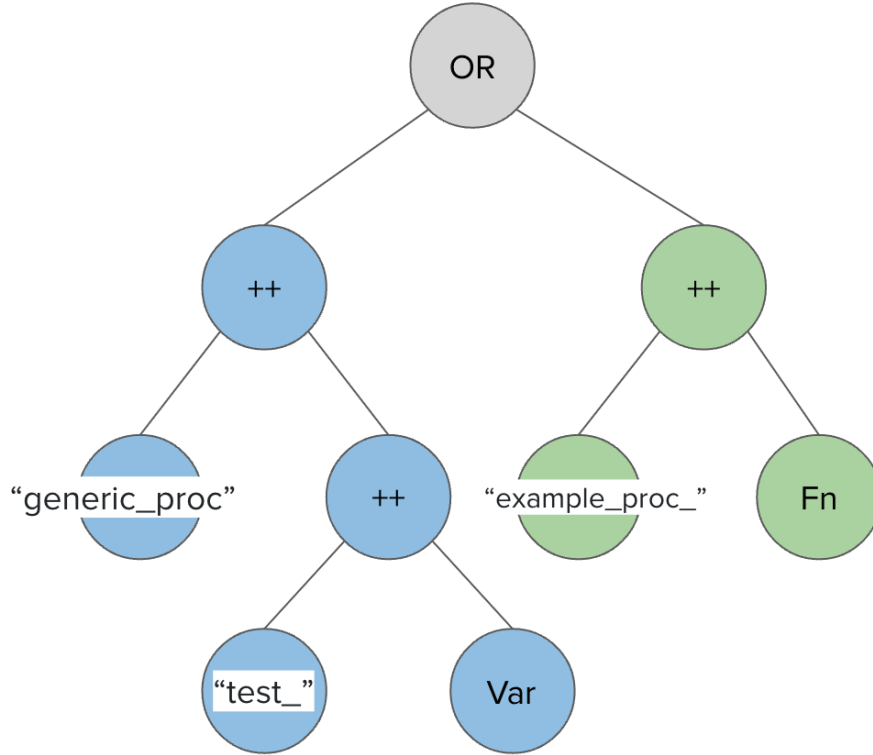


Figure 3.2: Match tree example

child is a variable: `Id`. We apply data-flow reaching to calculate the possible values of `Id`. The algorithm will detect that the `Id` 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 in a similar way.

From this match tree, then, 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 joker (`.*`) pattern has a low match score, as it is not specific at all.

The result will be all of the calls that can dynamically fill up the atom table, the number of matches, and the match score. The result of Algorithm 2 then needs to be grouped by the match score and ordered by it. The higher the match score is, the more probable it is that the atom table will be filled up.

The result of our example is demonstrated in Code 3.4. 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.

```

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

```

Code 3.4: Output of demonstration (atom exhaustion)

The group results are ordered based on the number of matches on the atom table². The more matches, the more atoms it has created in the current atom table, and the more probable chance to fill a future atom table up.

3.3 Message passing

This analysis aims to find bottlenecks in the source code, where messages are sent but not processed. It also requires the source code to be analysed by RefactorErl in the Semantic Program Graph representation.

In addition to the source code, it requires additional input from the monitoring tool the message queue of the process we would like to analyse.

3.3.1 Motivational Example

Consider a scenario in a BEAM system where multiple processes communicate asynchronously through message passing. Effective monitoring can raise an alarm, when a message queue is over a certain limit for a process. This can be a good notice, but it does not give the developers any clue to where or what to look for.

The message queue of a process can be retrieved from the BEAM, and WombatOAM (see Section 2.3) is capable of doing this. The message queue is a list of messages, where each message is an Erlang term.

Below, we illustrate an Erlang module designed to trail some messages.

```
1 -module(messages).
2 -export([run/1, loop/1]).
3
4 run(N) ->
5     Pid = spawn(fun() -> loop(ok) end),
6     register(proc, Pid),
7     foo(N).
8
9
10 foo(N) ->
11     proc ! {self(), process},
12     [proc ! {self(), trail} || _ <- lists:seq(1, N)],
13     proc ! {self(), exit},
14     X = 3,
15     proc ! {self(), X}.
16
17 receive_all() -> receive A -> A end.
18
19 loop(ok) ->
20     %receive_all(),
21     receive
22         {_Pid, process} -> io:format("Process received~n", []),
23         loop(ok);
24         {_Pid, exit} ->
25             io:format("Process exiting~n", [])
```

²The number of matches is based on the artificial example from Code 3.3

```

26
27     end .

```

Code 3.5: Example module on Message Passing

In Code 3.5 a module, called `messages` is introduced with the following functions:

- `run/1`: Spawns a new process that enters a message-processing loop. This process is registered under the name `proc` for easier sending.
- `foo/1`: Sends a series of messages to the `proc` process. These messages include repetitive signals (`trail`), a termination command (`exit`), and some data (`X`).
- `loop/1`: Represents the part of the code that is receiving messages. It processes incoming messages. In this case, it recognizes `process` and `exit` messages. In case of `process` the recursion continues, and in case of `exit` the process exits.

This example shows how message handling is done in Erlang. The issue is that the `proc` process is getting `trail` messages but not processing them. This is an issue because the message queue length can grow indefinitely. Moreover, as the `run/1` function is exported, it can create multiple processes. The parameter of the function is the number of messages to be sent, which can be a huge number, leading to a huge message queue.

Note: the comment on line 20, in Code 3.5 will be explained a bit more in detail at the end of Section 3.3.2.

3.3.2 Analysis

We will look into the algorithm that is working on a unified message queue of the whole system, and then we will look into the algorithm that is working on a single process.

Algorithm 3 Find Unprocessed Messages

Funct `find_unprocessed(MsgQ)`

- 1: Execute `anal_proc()`
 - 2: `Receives = find_nodes_by_type("receive_expr")`
 - 3: `Patterns = concatenate([receive_to_regex(R) for R in Receives])`
 - 4: `MsgQS = [format_msg(Msg) for Msg in MsgQ]`
 - 5: `NotProcMsgs = [Msg for Msg in MsgQS if not is_matching(Msg, Patterns)]`
 - 6: `SendExprs = flatten([get_sends(R) for R in Receives])`
 - 7: `BadSends = send_expressions_for_message(NotProcMsgs, SendExprs)`
 - 8: **return** `BadSends`
-

The Algorithm 3 is the main algorithm of the analysis. It is working on a unified message queue of the whole system.

- In the first line, an important step is needed: run the process analyser. It is needed so the spawn, receive and send expressions are correctly identified, and the needed metadata is added to the program graph.
- The second step is to get all of the receive expressions from the program graph.

- After it, each and every receive expression will be converted into a regular expression, which can be used to match the messages.
- Then, the messages in the message queue are converted into strings, so they can be matched with the regular expressions.
- The next step is to filter the messages, that are not processed by any of the receive expressions. `is_matching/1` is used to check if the message is matching any of the regular expressions. (`NotProcMsgs`)
- The next step is to get the send expressions, which are the origin of the messages (`SendExprs`).
- The final step is to reduce the send expressions obtained in the previous step to only those that are the origin of the unprocessed messages (`NotProcMsgs`).

This solution raises some concerns, as it is working on a unified message queue, it can be hard to find the origin of the messages, as it can be from multiple processes. For example, some processes can have a line like it is shown in Code 3.6. This would have been the case if, in Code 3.5, the comment on line 20 would have been uncommented.

```
1 receive A -> A end
```

Code 3.6: Example of a general receive

This means it processes all of the messages, while some other processes have really specific receive patterns, as we can see in the motivational example. This can completely impair the results of our analysis, so the next step is to improve this. First I will define the improved algorithm for a single process, then we will dive into the details, as we did with unified message queue analysis.

Algorithm 4 describes how the analysis works:

- The first step is the same as we did earlier. Run the process analyser.
- The next step is to get the semantic node of the Erlang module, as we are going to need it in the next steps.
- What we are looking for are the spawn functions, which are used to spawn new processes. We are looking for the 1 and 3 arity versions of the function.
- Then we need the applications of `spawn/1` and `spawn/3` functions.
- After that, we need to get the spawn definitions, which are used in the applications.
- We need to get the spawn definitions and filter out those which are spawned by the initial call. See Algorithm 6.
- Then, we get the spawn definitions that we calculate from the current call. This is needed as a lambda function was used at spawn, initiate the process, we would get `apply` from the `erlang` module as the initial call, which would lead us to nowhere. It finds the function that is used to spawn the process and traverses the references to get to the spawn definition.

Algorithm 4 Find Unprocessed Messages with Initial Call

Funct find_unprocessed({MsgQ, InitialCall})

```

1: Execute anal_proc()
2:
3: ErlangMod = query_exec(find_module("erlang"))
4: Spawn1 = query_exec(ErlangMod, find_function("spawn", 1))
5: Spawn3 = query_exec(ErlangMod, find_function("spawn", 3))
6:
7: Applications = query_exec(concat(Spawn1, Spawn3), applications())
8: SpawnDefs = query_exec(Applications, ["spawn_def"])
9: ValidSpawns = SpawnDefs that are is_spawned_by(InitialCall)
10: SpawnsFromCurrCall = get_spawns_from_current_call(CurrentCall)
11:
12: if (length(ValidSpawns) + length(SpawnsFromCurrCall)) > 0 then
13:   FoundInit = find_initial_call_for(CurrentCall)
14:
15:   ClausesI = fetch_function_clauses(InitialCall)
16:   ClausesC = fetch_function_clauses(FoundInit)
17:   Clauses = concat(ClausesI, ClausesC)
18:
19:   Bodies = flatten([query_exec(C, body()) for C in Clauses])
20:   DeepSubs = flatten([query_exec(B, deep_sub()) for B in Bodies])
21:   Closed = uniq(flatten([close(DS) for DS in DeepSubs]))
22:
23:   Receives = Nodes in Closed, that ere receive_expr(Nodes)
24:   SendExprs = flatten([get_sends(R) for R in Receives])
25:
26:   MsgQS = [ to_string(Msg) for Msg in MsgQ]
27:
28:   Patterns = concatenate([receive_to_regexp(R) for R in Receives])
29:   NotProcMsgs = [Msg for Msg in MsgQS if not is_matching(Msg,
    Patterns)]
30:
31:   BadSends = send_expressions_for_message(NotProcMsgs, SendExprs)
32:   return BadSends
33: else
34:   return []
35: end if

```

- Just to check everything, we should find a spawn definition, which is used to start the process. It is checked in line 11.
- The next step is to make a transitive closure of the function, which initiated that process to get all of the deep sub nodes of the function bodies. We use Algorithm 5 to calculate the transitive closure.
- Now we can move on to the receive expressions, we need to get all of the receive expressions, which are in the transitive closure of the initial call. Algorithm 11 gets the send expressions.
- Then the pattern for the receives are needed, so we convert the receive expressions to regular expressions. We use Algorithm 9 to get the regular expressions.
- The next step is to filter the messages, that are not processed by any of the receive expressions. `is_matching/1` is used for this one, which is just running the regular expressions on the messages with a proper configuration.
- The last step is to get the send expressions, which are the origin of the messages in the `NotProcMsgs` list. Algorithm 7 is used to get the send expressions.

This algorithm is working on a single process, and it is capable of finding the origin of the messages, which are not processed by the process. This is a huge improvement, as it can give a clue to the developers, where to look for the issue.

Note: the current implementation of the analysis in `RefactorErl` requires processes to be registered, but the analysis is not affected by this.

3.3.3 Definition of the steps of the algorithms

In the previous section, we have defined the main algorithms which are used in the analysis. In this section, we are going to define the building blocks of the analysis.

Algorithm 5 is a recursive function that calculates the transitive closure of an SPG node and all of its sub-nodes. It starts off by getting the functions of the application, then the forms of the functions, the clauses of the forms, and the bodies of the clauses. Now we have all of the nodes that are in the bodies of the function that was just applied.

The next step is to filter the list of deep sub-nodes to get only those that are not in the list of nodes we already have. If there are no new nodes, we can return the list of nodes. Otherwise, we can merge the new nodes with the old ones, iterate on the new nodes, and call the function recursively on them.

In Algorithm 6, there is a function that is true in case of the given spawn can be initiated by the given module, function and arity. This is needed to check if the message is sent by the process we are interested in.

Algorithm 7 is a function that finds the send expressions for a message. It starts off by building match trees from the send expressions and then creates regular expressions from these match trees. With these regular expressions, it can find the matching send expressions for the given message. It uses match trees (Definition 1) introduced in the previous chapter.

Algorithm 5 Closure of a Node

Funct close(Node, Nodes)

```

1: if Type(Node) = application then
2:   Funs = get_funs(Node)
3:   Forms = get_forms(Funs)
4:   Clauses = get_clauses(Forms)
5:   Bodies = get_bodies(Clauses)
6:   DeepSubs = get_deep_subs(Bodies)
7:
8:   NotIncluded = elements in DeepSub, that are not in Nodes
9:
10:  if NotIncluded is empty then
11:    return Nodes
12:  else
13:    Filtered = DeepSub without Node
14:    NewNodes = Merge(Nodes, NotIncluded)
15:
16:    Closeds = []
17:    for all N in NewNodes do
18:      Closeds.push(close(N, NewNodes))
19:    end for
20:    return Closeds
21:  end if
22: else
23:   return Node
24: end if

```

Algorithm 6 Check Node Initiation

Funct is_spawned_by(PidNode, {M, F, A})

```

1: Data = get_data_from_graph(PidNode)
2: return (Data.module = M) and (Data.function = F) and (Data.arity = A)

```

Algorithm 7 Find Send Expressions for a Single Message

Funct send_expressions_for_message(Msg, SendExprs)

```

1: Matchers = build_matchers(SendExprs)
2: Regexps = create_regexps(Matchers)
3:
4: Matched = []
5: for each {RP, R} in Regexps do
6:   if regexp_run(Msg, R) returns true then
7:     Matched.push(RP)
8:   end if
9: end for
10: return Matched

```

Algorithm 8 Fetch Function Clauses

Funct fetch_function_clauses(M, F, A)

- 1: Mod = get_module(M)
 - 2: Fun = get_function(Mod, F, A)
 - 3: Forms = get_forms(Fun)
 - 4: Form = get_single(Forms)
 - 5: **return** get_clauses(Form)
-

Algorithm 8 is a function that fetches the clauses of a function, from the given module with the given arity. It works by finding the node which is associated with the module in the semantic program graph, then starts a query from the module node to find the function with the given arity inside that module.

Then it is supposed to get one single form from the forms of the function, and return the clauses of that form, which can be of course more than one.

Algorithm 9 Convert a Receive Pattern to Regular Expressions

Funct receive_to_regexp(Node)

- 1: ReceivePatterns = query_receive_patterns(Node)
 - 2: Matchers = build_matchers(ReceivePatterns)
 - 3: Regexp = create_regexp(Matchers)
 - 4: **return** Regexp
-

Algorithm 9 is a function that converts a receive pattern to a regular expression. It starts off by querying the receive patterns from the given node, then builds match trees from these patterns, and creates regular expressions from these match trees.

Algorithm 10 Find Send Expressions for Multiple Messages

Funct sends_for_messages(Msgs, SendExprs)

- 1: PotentialSends = find_all_potential_sends(Msgs, SendExprs)
 - 2: BadSendPatterns = unique(PotentialSends)
 - 3: BadSends = query_top_expressions(BadSendPatterns)
 - 4: **return** BadSends
-

Algorithm 10 is a function that finds the send expressions for a list of messages. Potentially, it can be a list of not processed messages, which we are looking for the send expressions that can be the source of these messages.

It starts off by getting all of the potential send expressions for the messages, for the given send expression. The `find_all_potential_sends` function is using the `send_expressions_for_message` function, which is defined in Algorithm 7, in an iterative way.

Then it filters out the unique send expressions and queries the top expressions, which can be the source of the messages.

Algorithm 11 is a function that gets the send expressions from a receive node. It starts off by querying the patterns from the receive node, then gets the possible send expressions with the help of data flow reaching. It returns the unique send expressions.

Algorithm 11 Get Send Expressions from Receive Node

Funct get_sends(ReceiveNode)

- 1: ReceivePatterns = query_patterns(ReceiveNode)
 - 2: PossibleSends = dataflow_reach(ReceivePatterns)
 - 3: **return** unique(PossibleSends)
-

3.3.4 Demonstration of the algorithm

To demonstrate the defined analysis, let us consider the motivating example (Code 3.5). The first step is to run the process analysis [25]. Then, we need to get hold of all of the receive expressions. In this case it will be only the two:

- starting from line 21, in Code 3.5.
- starting from line 17, in Code 3.5.

The next step is to convert these receive expressions into regular expressions. The first receive block has two clauses, with the following pattern matches:

- {_Pid, process}
- {_Pid, exit}

The above patterns are converted into regular expressions, which are:

- {.*, process}
- {.*, exit}

The other receive (in line 17, Code 3.5), is a generic one, so the constructed pattern is the joker pattern: .*.

The next step is just a conversion step. The messages in the message queue need to be converted into strings, to be able to match on them.

The next step is to filter out the messages, which are not processed by any of the receive expressions. If we run the `run/1` function, it will spawn a receiving process. Then we can run the `foo/1` function, which will send messages to the receiving process. The messages are:

- {<0.88.0>, process} only once
- {<0.88.0>, trail} N times (where N is the parameter of the function)
- {<0.88.0>, exit} one time
- {<0.88.0>, 3} one time

The filtering will remove {<0.88.0>, process} and {<0.88.0>, exit} as they are processed by the receive expressions³.

Now we have the receive expressions, we need to have the send expressions that can send messages to them. The next step is getting this exactly. In our example, the send expressions are:

³<0.88.0> is the process identifier of the process, which is sending the messages, it can be different in different runs and systems.

- `proc ! {self(), process}` in line 11, Code 3.5
- `proc ! {self(), trail}` in line 12, Code 3.5
- `proc ! {self(), exit}` in line 13, Code 3.5
- `proc ! {self(), 3}` in line 15, Code 3.5

The last step is to get those send expressions which can be used to send the unprocessed messages. In our case, we will get a flag on line 12 and 15, as the messages are not processed by the receive expressions.

3.4 Process counts

In Erlang systems, the number of processes can easily grow into a large number, as Erlang is mostly used for designing and building highly concurrent, even distributed systems. Processes are spawned and terminated frequently, as an Erlang process is really lightweight. Because of this lightweight design, it seems really easy and convenient to spawn new processes for different tasks, but creating new processes for tasks can really raise the number of processes, and it is a limited resource. When an Erlang node is started, it has certain system limits, and the number of processes is one of them. This limit can be changed, but the node must be restarted.

This chapter will introduce how processes in the Erlang system are counted, what information is available about processes, and how to use this information to detect potential bottlenecks in the system.

3.4.1 Motivational Example

Consider a scenario where you have an Erlang system running for a long time, and you notice that you have a high number of processes running, because a good monitoring tool is telling you so. This can be a good notice, but it does not give the developers any clue to where or what to look.

Below, I illustrate an Erlang module (Code 3.7) designed to spawn a few different processes.

```
1 -module(proc_classes).
2 -export([forever_living/0, forever_living2/0, long_living/0,
3         short_living/0]).
4 forever_living() ->
5     F = fun() ->
6         receive
7             Msg ->
8                 tasks:process_msg(Msg),
9                 F()
10        end
11    end
12    end,
13    spawn(F).
14
15 forever_living2() ->
16 F = fun() ->
17     receive
```

```
18         {never_arrive, Msg} ->
19             tasks:process_msg(Msg),
20             F()
21         end
22     end
23 end,
24 spawn(F).
25
26 long_living() ->
27     F = fun() ->
28         receive
29             Msg ->
30                 case tasks:do_task(Msg) of
31                     {continue, Result} ->
32                         tasks:process_result(Result),
33                         F();
34                     {stop, Result} ->
35                         tasks:process_result(Result)
36                 end
37             end
38         end,
39         spawn(F).
40
41 short_living(Pid) ->
42     spawn(fun(Pid) ->
43         Pid ! alive
44     end).
```

Code 3.7: Example module on Process Count

Code 3.7 is a module, called `proc_classes`. It contains the following functions:

- `forever_living/0`: Spawns a process that will enter into a receive block waiting for a message. Then it will call itself recursively, and never exits. We will call this a **forever living process**.
- `forever_living2/0`: Spawns a process that will enter into a receive block waiting for a message, that will never arrive. This is also considered a **forever living process**, even without the recursion.
- `long_living/0`: Spawns a process that will enter into a receive block waiting for a message. It will process the message, and then decide if it should continue or stop, based on the result. This process will be considered a **long living process**, as it has a potential to exit the recursion, unlike the `forever_living/0` process, but it can be alive for a long time.
- `short_living/1`: Spawns a process, which sends a message to another process, and then exits. This is a **short living process**, as its execution time is very short.

We will classify the processes that are spawned into the above described categories: forever living, long living, and short living processes. We are interested in the number of forever living and long living processes, as they can be potential bottlenecks in the system.

3.4.2 Analysis

After the monitoring tool sends out an alarm, we need to run an on-demand static analysis.

Algorithm 12 Get long-living processes

Func `get_long_living_processes()`

```

1: Execute anal_proc()
2:
3: ErlangMod = query_exec(find_module("erlang"))
4: Spawn1 = query_exec(ErlangMod, find_function("spawn", 1))
5: Spawn3 = query_exec(ErlangMod, find_function("spawn", 3))
6: Spawns = concatenate(Spawn1, Spawn3)
7:
8: Applications = query_exec(Spawns, applications())
9: SpawnDefs = query_exec(Applications, ["spawn_def"])
10:
11: LongLiving = []
12: for all Spawn in Spawns do
13:   if is_long_living(Spawn) or is_forever_living(Spawn) then
14:     LongLiving.push(Spawn)
15:   end if
16: end for
17: return LongLiving
```

Algorithm 12 is the main algorithm of the analysis.

- Running the process analysis is an important first step, to get more data about processes.
- Then we need to get the graph node, representing the `erlang` module. From that, we are able to query the functions `spawn/1` and `spawn/3`.
- Then, with the concatenation of the results, another query needs to be run to get all of the applications and then the definition of the spawn calls.
- Finally, we iterate over all of the spawn calls and check if they are long-living or forever-living processes. If yes, add it to the list of long living processes.

It is important to give a good definition for forever living and long living processes. Functions are started with the help of the `spawn/1` and `spawn/3` functions. The `spawn/1` requires lambda function as an argument, while `spawn/3` needs the module and function name we want to spawn, with its arguments in a list. In both cases, this function is the **initial function of that process**. Another important terminology is the **current function** of the process. This is the function that the given process is currently executing.

A process is considered **forever living** if it has a receive block and it calls itself recursively without a clause that could break the recursion. `is_forever_living/1` function will return true if the spawn call that is given as an argument will spawn a forever living process.

A process is also considered **forever living** if it has a receive block, either in a recursive function or not, that has a pattern(s) that will never match. This way the receive expression will wait for forever a message that will never arrive, so it will block that process for indefinitely and never terminate.

A process is considered **long living** if it has a receive block in a recursive function. If the function has a clause that could break the recursion, it is considered a long-living process. `is_long_living/1` function will return true if the spawn call that is given as argument will spawn a long living process.

3.5 Port counts

In Erlang, ports are used to communicate with external programs. All external communication between Erlang and C code must be done through ports. An Erlang process can create a port, using the `open_port/2` function. This function will be the connected process of the port, and all communication needs to go through this process [32].

Just as the number of processes (Section 3.4), the number of ports is also a limited resource in an Erlang system. The number of ports can be changed, but it requires the node to be restarted.

3.5.1 Motivational Example

```
1 -module(complex).
2 -export([start/1, init/1]).
3
4 start(ExtPrg) ->
5   spawn(?MODULE, init, [ExtPrg]).
6
7 init(ExtPrg) ->
8   register(complex, self()),
9   process_flag(trap_exit, true),
10  Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
11  loop(Port).
12
13 loop(Port) ->
14   receive
15     {call, Caller, Msg} ->
16       Port ! {self(), {command, encode(Msg)}},
17       receive
18         {Port, {data, Data}} ->
19           Caller ! {complex, decode(Data)}
20       end,
21     loop(Port)
22   end.
23
24 encode({foo, X}) -> [1, X];
25 encode({bar, Y}) -> [2, Y].
26
27 decode([Int]) -> Int.
```

Code 3.8: Example module on Port usage

Code 3.8 is an example Erlang module from the official tutorials in the Erlang documentation [32]. The module is an example of how to communicate with an external port. The `start/1` is supposed to start a process, which will communicate with the external program. The external program should be passed as an argument. After that the process enters into a loop, where it waits for messages. When a message arrives to it from a certain caller, it sends it to the port, and waits for the result from the port. When the result arrives, it sends it back to the caller.

```
1 -module(complex).
2 -export([start/1, init/1]).
3
4 ...
5
6 loop(Port) ->
7   receive
8     {never, Caller, Msg} ->
9       Port ! {self(), {command, encode(Msg)}},
10      receive
11        {Port, {data, Data}} ->
12          Caller ! {complex, decode(Data)}
13      end,
14      loop(Port)
15  end.
16
17 ...
```

Code 3.9: Example module on Port usage with message that never arrives

Code 3.9 is a modified version of Code 3.8. The only difference is that the `loop/1` function has a new message pattern, which will never arrive. The aim is similar to what is discussed in Section 3.4. The process will wait for a message that will never arrive, and it will be a **forever living port**, but in this case, it will also halt a port, so that port is considered as a **forever living port**.

```
1 -module(complex).
2 -export([start/1, init/1]).
3
4 ...
5
6 loop(Port) ->
7   receive
8     {call, Caller, Msg} ->
9       Port ! {self(), {command, encode(Msg)}},
10      receive
11        {Port, {data, Data}} ->
12          Caller ! {complex, decode(Data)}
13      end,
14      loop(Port);
15   stop ->
16     Port ! {self(), close},
17     receive
18       {Port, closed} -> exit(normal)
19     end;
20   {'EXIT', Port, Reason} -> exit(port_terminated)
21 end.
22
23 ...
```

Code 3.10: Example module on Port usage, with option to stop

Code 3.10 is the second modification of Code 3.8. The `loop/1` function introduces a new message pattern, and when that clause is executed, it will stop the process. When the process receives this message, it will send a message to the port to close, then it will wait for the response from the port. If everything is right, the port will close and the process will terminate.

Since there are receive blocks in the initial call (see Section 3.4.2), this process will be a **long living port**, and the port will be a **long living port**.

3.5.2 Analysis

After the monitoring tool sent out an alarm about reaching the threshold for port counts, we can run the static analysis to find forever or long living ports.

Algorithm 13 Get long living ports

Func `get_long_living_ports()`

```

1: ErlangMod = query_exec(find_module("erlang"))
2: Ports = query_exec(ErlangMod, find_function("open_port", 2))
3:
4: PortApp = query_exec(Ports, applications())
5: LongLiving = []
6: for all PA in PortApp do
7:   if is_long_living(PA) or is_forever_living(PA) then
8:     LongLiving.push(PA)
9:   end if
10: end for
11: return LongLiving

```

Algorithm 13 is used to find long living ports.

- The first step is to get the graph node, which represents the `erlang` module.
- From the `erlang` module, we can find all the functions that are calling the `open_port/2` function, by running a query to find the function `open_port/2`.
- After it, we can iterate over all the applications and check if they are long-living or forever-living; if yes, add it to the list of long-living ports.

To decide if a port is long-living or forever-living is similar to the one used for processes in Section 3.4:

- If the port is opened in a process that is forever living, then the port is also forever living.
- If the port is in a long-living process, then the port is also long-living.
- If the port is never closed because there is no function call to `port_close/1` from the `erlang` module, then the port is considered as a forever living port.

Chapter 4

Evaluation

Evaluating the results of the thesis is a crucial part of the work. It verifies that the concepts of bottleneck analysis is working.

The analysis on atom exhaustion and message passing is ready to be used, and implemented in RefactorErl. The process and port count analysis is still under implementation, and it is not yet ready to be used.

The algorithms are already tested on artificial examples, and they are working as expected. In this chapter, I will evaluate the algorithms on open-source projects, and I will show how the algorithms can be used in a real-world scenario.

4.1 Atom exhaustion in MongooseIM

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.

4.1.1 Preparing the environment

RefactorErl

We need to prepare the RefactorErl. It can be downloaded from the official page of RefactorErl: <https://plc.inf.elte.hu/erlang/>. The modifications are currently not available in the official release, but it will be in the next release.

MongooseIM

MongooseIM is an open-source, efficient XMPP server which can be at the core of an Instant Messaging platform [33]. Version 6.1.0 will be used, which can be cloned from the GitHub page of the project¹. This source code needs to be added and analysed by RefactorErl.

For easier usage, I used the same 6.1.0 version, but from a Docker Image, which can be downloaded from the website of Dockerhub ².

¹<https://github.com/esl/MongooseIM/tree/6.1.0>

²<https://hub.docker.com/layers/mongooseim/mongooseim/6.1.0/images/sha256-16da98b09ed9f38136f5791c1435da49be8524dbf019ec745de3dc0b7715a02f0?context=explore>

4.1.2 Scenario: debugging MongooseIM

Using the security checkers of RefactorErl [22], I could identify a vulnerability, which is 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. In this section of evaluation, we will simulate a debugging session, where we are trying to get multiple host names on a trial-and-error basis.

Step 1: preparing MongooseIM

We are going to use Docker, so that is needed for this evaluation. Now start MongooseIM in a containerized environment. Using the shell of the host system, just run this command:

```
docker run -dtP -h mongooseim-1 --name mongooseim-1
mongooseim/mongooseim:6.1.0
```

This will start MongooseIM in a docker container, and we will get the identifier of the container as a response, something like this:

```
214c0196556a43db1fd233f75977a38af0cb3fe78e179fae8976f09e91362f46
```

The next step is to connect to this already running container, by running the following command in the shell of the host machine: `docker exec -it <container_id> bash`, replacing the `<container_id>` with the one you got in the previous step.

Now we have access to the shell of the container, in that shell navigate to the following directory: `/lib/mongooseim` and execute the following: `bin/mongooseim debug` this will run a script, that will attach to the already running MongooseIM node.

Step 2: check current atom count

Before we start our experiment request the current size of the atom table, using the built-in system information. Run the following command in the attached Erlang shell: `erlang:system_info(atom_count) ..` It should return the current number of atoms in the atom table.

Step 3: run the vulnerable command

Then I run the following commands on the node to get some identifiers: `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.

Step 4: 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. Here we should find the function call mentioned in Section 4.1.2.

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_*`

```

1 true
2 false
3 ...
4 base1_myhost
5 base2_myhost
6 ...
7 base20_myhost
8 ...

```

Code 4.1: Atom table of MongooseIM

The current state of the atom table can be seen in Code 4.1. It is observable that besides the usual and needed atoms such as `true` and `false` (and the names of modules and functions), there are many atoms that are generated dynamically. This is a sign of possible atom exhaustion.

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 Code 4.2.

```

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

```

Code 4.2: Function that is probably causing the atom exhaustion

4.1.3 Additional step: WombatOAM, monitoring

There can be an additional step where we can monitor the atom table size, as it can be seen in Figure 4.1. WombatOAM (Section 2.3) can do this. It is possible to create metrics and alarms based on the number of atoms in the system.

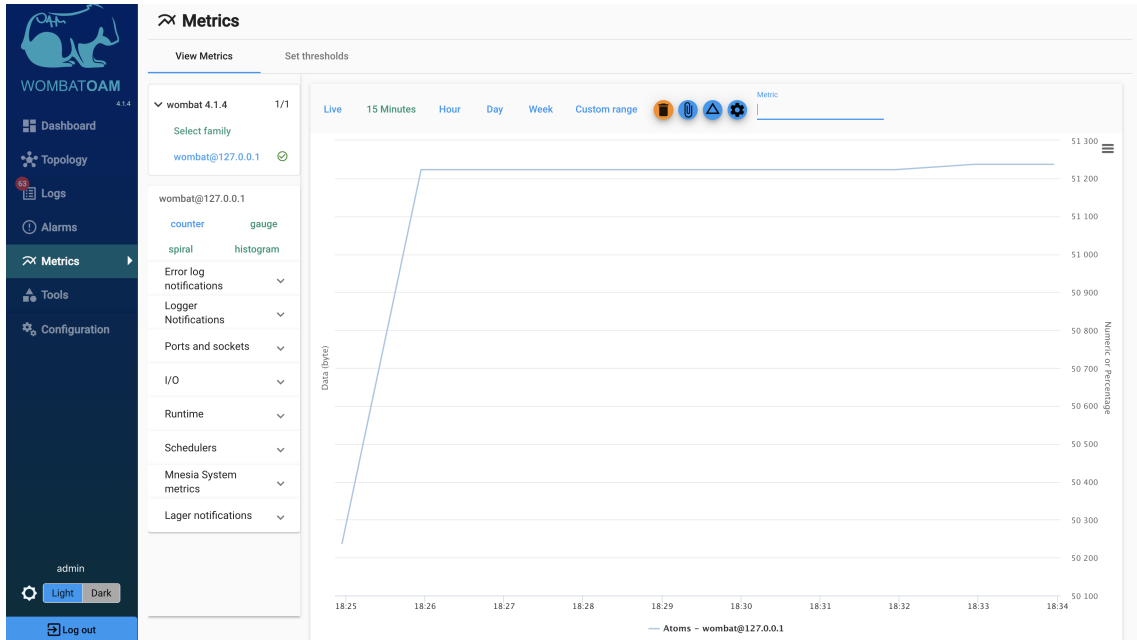


Figure 4.1: WombatOAM monitoring the atom count

Currently, WombatOAM is not capable of firing an on-demand static analysis, in case of an alarm, but it is a possible future work. Right now we can set up a notification, an alarm. It can be sent to an email address, or to a Slack channel and many more [17].

4.1.4 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.

4.2 Message queue trailing in Mongoose IM

This section will look into a case where a certain part of Mongoose IM could produce a trailing message queue.

4.2.1 Preparing the environment

RefactorErl

The same environment is needed as in Section 4.1.1.

MongooseIM

MongooseIM is introduced in Section 4.1.1. The same version will be used, but the source code will be modified, to create a trailing message queue.

4.2.2 Scenario: Authentication

MongooseIM uses `extauth` for authentication. This module also spawns processes, and we can communicate with them using message passing. This module is really well written, handling a lot of edge cases, so some minor modifications were necessary to make it a good example for this evaluation.

Modifications

The original source code is accessible on the GitHub page of MongooseIM³.

In Code 4.3 there is some part of the modified version of `extauth` module. On line 67 the register call is modified. In the original source, the process name was generated dynamically, and it was passed along to the process as a parameter. This was changed to a constant because RefactorErl is not capable of speculative evaluation with this analysis.

This highlighted a possible improvement, where the name of the process could be calculated. This is a future work, and it is mentioned in Chapter 6.

```

66 init(ProcessName, ExtPrg) ->
67     register(auth_proc, self()),
68     process_flag(trap_exit, true),
69     Port = open_port({spawn, ExtPrg}, [{packet, 2}]),
70     loop(Port, ?INIT_TIMEOUT, ProcessName, ExtPrg).
```

Code 4.3: Starting the process in `extauth` (`src/auth/extauth.erl`)

The process that is spawned in Code 4.3, gets messages from `call_port/2`, shown in Code 4.4. This function was also modified, to send to the `auth_proc` process. Another modification was addressed on this function as well, the message it sends is modified to `{cal, self(), Msg}`. This simulates a scenario where the developers made a typo.

```

118 call_port(HostType, Msg) ->
119     %ProcessName = get_process_name(HostType, random_instance(
120         get_instances(HostType))),
121     auth_proc ! {cal, self(), Msg},
122     receive
123         {eauth, Result} ->
124             Result
```

Code 4.4: Sending messages in `extauth`

The last modification that was necessary is in the body of `loop/4` function, which is visible on Code 4.5. That function has multiple receives, and two clauses of it call a function called `flush_buffer_and_forward_messages/1`. I showed the original line number information in comments.

³<https://github.com/esl/MongooseIM/blob/6.1.0/src/auth/extauth.erl>

```

1  -module(extauth).
2  ...
3  -spec loop(port(), integer(), atom(), any()) -> no_return().
4  loop(Port, Timeout, ProcessName, ExtPrg) ->
5      receive
6          {call, Caller, Msg} ->
7              port_command(Port, encode(Msg)),
8              receive
9                  ...
10             after
11                 Timeout ->
12                     ...
13                     %flush_buffer_and_forward_messages(Pid), %
14                     ln. 161
15                     exit(port_terminated)
16             end;
17             stop ->
18                 ...
19                 end;
20                 {'EXIT', Port, Reason} ->
21                     ...
22                     %flush_buffer_and_forward_messages(Pid), % ln. 175
23                     exit(port_terminated)
24             end.

```

Code 4.5: Flushing the buffer in extauth

In Code 4.5, I removed the functions calls to `flush_buffer_and_forward_messages/1`, which is shown in Code 4.6 because it has a receive which is catching all of the values in the message queue. Since this receive is nested, this can only run if the first receive is received an `{'EXIT', Port, Reason}` message or timed out. This analysis can be improved by implementing control flow analysis, which is a future work, and it is mentioned in Chapter 6.

```

1
2  -spec flush_buffer_and_forward_messages(pid()) -> 'true'.
3  flush_buffer_and_forward_messages(Pid) ->
4      receive
5          Message ->
6              Pid ! Message,
7              flush_buffer_and_forward_messages(Pid)
8      after 0 ->
9          true
10     end.

```

Code 4.6: Flushing the buffer in extauth

Running the analysis, by calling the following command in RefactorErl:

```
refusr_msg_trail:find_unprocessed([cal, <0.344.0>, "hello"],
                                   {extauth, loop, 4})
```

will point of this source code fragment: `auth_proc ! {cal, self(), Msg}`, because that is the part where we send a message that is not processed.

4.2.3 Runtime

I have run the analysis on a specific module of MongooseIM, which is the `extauth` module, not on the whole source code. That module is 207 lines long, and the analysis took 0.17 seconds. The complexity of the analysis is based on the number of receive and send expressions. The more of them are in the source code, the more complex the analysis will be.

In this module, there were 4 receive expressions and 6 send expressions, of which 3 were associated with the receive expressions.

There can be a source code of 100 000 lines, with just a few receives and sends, and the analysis will be faster than a 10 000 line source code with many receives and sends.

This analysis also uses data-flow reaching calculation, therefore an increasing data-flow complexity of the code might increase the runtime.

Chapter 5

Related

The field of static analysis is very diverse. In Chapter 1, I have already mentioned these kinds of tools. Most of them are used for finding bugs, security vulnerabilities, or performance bottlenecks. In this chapter, I will present some works that are related to my thesis.

5.1 Tools

When we talk about static analysis, we would usually like to see a ready-to-use tool that can be tried out in a project. In the following, I would like to highlight a few of them.

Code Compass

Code Compass 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 [34, 13].

JProfiler

JProfiler 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 [19]. It also aims to help developers to understand the performance of their Java applications, and find bottlenecks, just like my thesis aims to do for Erlang. JProfiler, is a dynamic profiler tool, so it has a relevance on the first part of my thesis, when we are looking for issues in the runtime.

The manual way

Besides the findings of this thesis, there is no way to finding bottlenecks in Erlang source code, based on runtime data in an automated, algorithmic way, that I am aware of.

Currently, the manual way is to open or connect to the shell of the node that is experiencing issues and obtain atom information or process information manually. This can be very time-consuming, and the only tool is the character-based shell.

Then, with this data in hand, it is possible to use `grep`¹ or equivalent tools on the source code, such as search features of IDEs to find the bottlenecks.

After a bottleneck is found and fixed, to test it, this process needs to be repeated from the beginning.

This is a very time-consuming process, and my thesis aims to automate this process and make it more efficient by providing algorithms for static analysis. An improvement could be made to integrate it with the dynamic analysis tool to make the process even more efficient.

Erlang tracing

Erlang tracing is also a tool used by developers, and it can be another refinement to the previously mentioned manual process. When we have a function which is a possible bottleneck, we can apply tracing to it.

With tracing, it is possible to see how many times a function was called. This is a good indicator and a way to verify if the found bottleneck is valid [35].

An important note is that tracing adds serious overhead to the node. Tracing each and every function call in a production system is not recommended because it would slow down the system significantly. It is recommended to only trace the suspected functions, and only for a short period of time.

Erlang tracing is dynamic monitoring tool, since it is not a combined tool nor a tool set, the only relevance to my work is that, tracing could be used to verify the results of the static analysis tool.

Go Race Detector

Go is an open-source programming language which has built-in concurrency and a robust standard library [36]. The Race detector uses the C/C++ thread sanitizers [37], which proved to be useful in finding concurrency issues in C/C++ programs.

As its name suggest the tool is looking for race conditions. It has a part included inn the compiler which checks how memory is accessed and used, and it has runtime library what is watches for unsynchronized memory access.

We can notice similiarities between Go Race Detector and my work here, it also has a static and dynamic part, where the static part is the compiler and the dynamic part is the runtime library.

Sobelow

Sobelow is a static analysis tool mainly for the Phoenix web framework [38], which is built on the Elixir [39] 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 [40].

¹Grep is a command-line utility for searching texts that match a regular expression.

Akka.Analyser

Akka is an open-source toolkit for building concurrent and distributed applications on the JVM. Akka has an actor model inspired by Erlang [41]. It has another version called Akka.NET, which is a version for the .NET framework. Akka.Analyser is a tool that looks for commonly made mistakes in the Akka.NET framework. It has no combination of static and dynamic analysis [42].

5.2 Research

Since the field is quite active, we can find several good research papers and articles on the topic.

Java performance bottlenecks

There is also a lot of research on this topic. For example, research at National Cheng Kung University and Iowa State University examined bottlenecks in Java systems [43]. 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.

Relating hardware performance monitors to source code

Another research about Java was carried out at Ghent University, where they linked microprocessor level hardware characteristics to source code. They concluded that it is helping identify performance bottlenecks and their causes [44]. It is positioned as a gap filler between profilers and hardware performance monitors. I would still consider this as a dynamic analysis tool, as it needs the system to be running, even though it manipulates the source code, it does not analyse it.

Concurrency Errors in Erlang

There is a research paper about concurrency errors, where they present a testing tool called Concuerror (introduced in Section 1.1) and methods used in it. The paper is mainly about how they took advantage of the actor model of Erlang to instrument the parts of programs that are under testing selectively. Since Concuerror is a stateless model checker, it can find concurrency errors and errors that can only occur on a few specific schedulings [8, 9].

Dialyzer

A proceeding paper by Konstantinos Sagonas discusses Dialyzer, a static analysis tool for Erlang [3]. Dialyzer started as a defect detection, which used mostly ad hoc data flow analysis, but the development shifted towards analysing typing specifications of dynamically typed languages. Now, Dialyzer uses a constraint-based analysis to infer success typing of functions based on the contracts (type specifications) that the user provides.

In this paper, he also discusses the challenges of the tool being part of the Erlang/OTP distribution and some of its features, like the concurrency defect analyser.

Percept2

Percept2 is an improvement of the Percept tool. It is a profiler tool for Erlang. It shows the users how the Erlang application could benefit from parallelism on a multi-core Erlang system. Percept2 also enables the programmers to improve their system's performance and scalability [45]. In a paper [45], they discuss the topic of profiling multi-core Erlang programs.

Feature identification using regular expressions

There is a research paper by Aston Parlindungan Tampubolon and others about 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). I am using an opposite approach: the regular expressions are not defined in advance but built dynamically and matched on runtime values [46].

Chapter 6

Conclusion and future work

Creating complex systems has its own 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 really 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 thesis I introduced the concept of bottleneck analysis in Erlang. It leverages data available during runtime and enriches static analysis with it.

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

Second, the thesis was about message passing and how a long mailbox can slow down a process, and even if our whole system is quick and efficient, a slow process can slow down the whole system. I also introduced an analysis to find these spots.

The third topic was about process creation and process count. An Erlang system is highly concurrent. Even though its processes are quite lightweight, if we have too many of them, it can still be a bottleneck. The thesis introduced a new algorithm to find these issues.

The last bottleneck family the thesis was about was the port count. Ports can also be a limited resource, and if we have too many, they can also be an unnecessarily managed resource. The thesis introduced an algorithm to find these issues as well.

After defining the algorithms, the final stage was to start to implement them in RefactorErl. Currently there is implementation for atom exhaustion and message passing, and the other two are under development.

6.1 Future work on bottleneck analysis

Even though the algorithms introduced in this thesis are for the bottleneck families, their implementation needs to be finished and improved. First of all the implementation of process and port count analyses needs to be implemented. After that, the process analyser should be improved, as it is used heavily by the message queue analysis. Currently, it is only stable on the registered processes spawned by `spawn/3`,

and it is not working in all cases. If the process analyser is improved, the message queue analysis can benefit from it. Another place for improvement is to extend the analysis to the so-called, `gen_servers`. These `gen_servers` are often used in production systems, as they provide a high-level interface, but they still use message passing under the hood [47, 48]. The message queue analysis could be also improved by implementing execution path analysis with the help of control flow, so nested receive clauses and unreachable code could be found. Another improvement would be when the names of processes are dynamically generated. In this case we could use speculative evaluation to find the name of the processes. Similar approaches are needed what are used in the atom exhaustion analysis, when we match on the atom table, but in this case we need to match on the table of registered processes, which can be also requested from the Erlang VM.

Currently, the bottleneck analysis supports four families of issues: atom exhaustion, message queue length, process count and port count. The analysis could be extended to other bottleneck families, such as orphan processes and the handling of different kinds of Erlang databases and tables. Also, research could be done on different kinds of bottlenecks to try to find new ones.

The ideas and methodologies used in this thesis will most likely be applied to new bottleneck families.

6.2 Future work on dynamic analysis integration

The algorithms that are introduced in this thesis 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 to issues by importance.

Acknowledgements

This thesis would not have been possible without the guidance and support of many people. First and foremost, I would like to thank the endless patience, help, support and countless consultations, that I have got from my supervisors.

I am grateful to my colleagues at Erlang Solutions, especially Zoltán Literáti, for providing support a flexible schedule and the chance to write parts of it at work. I am also thankful to Mohamed Ali Khechine, who added his invaluable professional insights to this thesis.

Last but not least, I would like to thank my parents, sister, and significant other for their support throughout writing the thesis.

Bibliography

- [1] Simon Thompson Francesco Cesarini. *Erlang Programming*. 1st. O'Reilly Media, 2009. ISBN: 978-0-596-51818-9.
- [2] Konstantinos Sagonas. “Fifteen years of successfully dialyzing Erlang and Elixir code (invited talk)”. In: *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450386128. DOI: 10.1145/3471871.3480952. URL: <https://doi.org/10.1145/3471871.3480952>.
- [3] Konstantinos Sagonas. “Using Static Analysis to Detect Type Errors and Concurrency Defects in Erlang Programs”. In: *Functional and Logic Programming*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 13–18. ISBN: 978-3-642-12251-4.
- [4] Jeremy Siek. “Challenges and progress toward efficient gradual typing (invited talk)”. In: *SIGPLAN Not.* 52.11 (Oct. 2017), p. 2. ISSN: 0362-1340. DOI: 10.1145/3170472.3148570. URL: <https://doi.org/10.1145/3170472.3148570>.
- [5] Matteo Cimini and Jeremy G. Siek. “The gradualizer: a methodology and algorithm for generating gradual type systems”. In: *SIGPLAN Not.* 51.1 (Jan. 2016), 443–455. ISSN: 0362-1340. DOI: 10.1145/2914770.2837632. URL: <https://doi.org/10.1145/2914770.2837632>.
- [6] Josef Svenningsson. *Gradualizer: A Gradual Type System for Erlang*. <https://github.com/josefs/Gradualizer>. Accessed: 2024-05-15. 2024.
- [7] Ákos Hajdu et al. “InfERL: scalable and extensible Erlang static analysis”. In: *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*. Erlang 2022. New York, NY, USA: Association for Computing Machinery, 2022, 33–39. ISBN: 9781450394352. DOI: 10.1145/3546186.3549929. URL: <https://doi.org/10.1145/3546186.3549929>.
- [8] Stavros Aronis and Kostis Sagonas. *Concuerror: Stateless Model Checking Tool for Erlang Programs*. Visited: 2024-05-16. 2024. URL: <https://github.com/paraplui/Concuerror>.
- [9] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. “Systematic Testing for Detecting Concurrency Errors in Erlang Programs”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 154–163. DOI: 10.1109/ICST.2013.50.
- [10] SonarSource. *SonarQube Documentation*. Visited: 2024-05-15. 2024.

- [11] ESLint Team. *ESLint: Find and Fix Problems in Your JavaScript Code*. <https://eslint.org/>. Visited: 2024-05-15. 2024.
- [12] JetBrains. *ReSharper Features*. <https://www.jetbrains.com/resharper/features/>. Visited: 2024-05-15. 2024.
- [13] Anett Fekete, Máté Cserép, and Zoltán 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] CodeCompass Team. *CodeCompass: An Open Software Comprehension Framework*. Visited: 2024-05-16. 2024. URL: <https://codecompass.net/>.
- [15] Erlang/OTP Team. *Erlang – eprof*. Visited: 2024-05-16. 2024. URL: <https://www.erlang.org/doc/man/eprof.html>.
- [16] Erlang/OTP Team. *Erlang – Observer*. Visited: 2024-05-16. 2024. URL: <https://www.erlang.org/doc/man/observer.html>.
- [17] *Metrics - WombatOAM*. Visited: 2024-05-14. 2024. URL: <https://s3.eu-central-1.amazonaws.com/www.wombat-docs.com/metrics.html>.
- [18] Phil Trinder et al. “Scaling Reliably: Improving the Scalability of the Erlang Distributed Actor Platform”. In: *ACM Trans. Program. Lang. Syst.* 39.4 (Aug. 2017). ISSN: 0164-0925. DOI: 10.1145/3107937. URL: <https://doi.org/10.1145/3107937>.
- [19] ej-technologies GmbH. *Java Profiler - JProfiler*. Visited: 2024-05-16. 2024. URL: <https://www.ej-technologies.com/products/jprofiler/overview.html>.
- [20] Ben Frederickson. *py-spy: Sampling profiler for Python programs*. Visited: 2024-05-16. 2024. URL: <https://github.com/benfred/py-spy>.
- [21] Saša Jurić. *Elixir in Action, Third edition*. Manning Publications Co. LLC, 2024.
- [22] Brigitta Baranyai. *Funkcionális nyelvek és a statikus kódelemzéssel támogatott biztonságos szoftverfejlesztés*. TDK dolgozat. Budapest, 2020.
- [23] Zoltán Horváth et al. “Modeling Semantic Knowledge in Erlang for Refactoring”. In: *Studia Universitatis Informatica* 54 (Jan. 2009), pp. 7–16.
- [24] Melinda Tóth and István Bozó. “Static Analysis of Complex Software Systems Implemented in Erlang”. In: *Central European Functional Programming School: 4th Summer School, CEFPS 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*. Ed. by Viktória Zsóka, Zoltán Horváth, and Rinus Plasmeijer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 440–498. ISBN: 978-3-642-32096-5. DOI: 10.1007/978-3-642-32096-5_9. URL: https://doi.org/10.1007/978-3-642-32096-5_9.
- [25] Melinda Tóth and István Bozó. “Detecting and Visualising Process Relationships in Erlang”. In: *Procedia Computer Science* 29 (2014). 2014 International Conference on Computational Science, pp. 1525–1534. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2014.05.138>. URL: <https://www.sciencedirect.com/science/article/pii/S1877050914003159>.

- [26] *Erlang - Efficiency Guide: Advanced*. Visited: 2024-05-14. 2024. URL: https://erlang.org/documentation/doc-5.8.4/doc/efficiency_guide/advanced.html.
- [27] *Erlang - Data Types: Atom*. Visited: 2024-05-14. 2024. URL: https://www.erlang.org/doc/reference_manual/data_types.html#atom.
- [28] Magnus Henoch. *Can I get a list of all currently-registered atoms?* — *stackoverflow.com*. <https://stackoverflow.com/questions/13480462/can-i-get-a-list-of-all-currently-registered-atoms/34883331#34883331>. Visited 2024-03-24.
- [29] *Erlang - External Term Format* — *erlang.org*. https://www.erlang.org/doc/apps/erts/erl_ext_dist.html. Visited 2024-03-24.
- [30] Björn-Egil Dahlberg <egil(at)Erlang.org>. *Eep 0043 - Erlang/OTP* — *erlang.org*. <https://www.erlang.org/eeps/eep-0043.html>. Visited 2024-03-24.
- [31] Melinda Tóth and István Bozó. “Supporting Secure Coding for Erlang”. In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. SAC ’24. Avil, Spain: Association for Computing Machinery, 2024, 1307–1311. ISBN: 9798400702433. DOI: 10.1145/3605098.3636185. URL: <https://doi.org/10.1145/3605098.3636185>.
- [32] *Erlang - Tutorial: Ports*. Visited: 2024-05-14. 2024. URL: https://www.erlang.org/doc/tutorial/c_port.
- [33] Erlang Solutions. *MongooseIM* — *esl.github.io*. <https://esl.github.io/MongooseDocs/latest/>. Visited 2024-03-19.
- [34] Zoltán Porkoláb and Tibor Brunner. “The codecompass comprehension framework”. In: *Proceedings of the 26th Conference on Program Comprehension*. ICPC ’18. Association for Computing Machinery, 2018, 393–396. ISBN: 9781450357142. DOI: 10.1145/3196321.3196352. URL: <https://doi.org/10.1145/3196321.3196352>.
- [35] Erlang/OTP Team. *Erlang - Tracing*. Version 14.2.5, Visited 2024-05-15. Erlang/OTP.
- [36] The Go Programming Language. *The Go Programming Language*. Visited: 2024-05-22. 2024. URL: <https://go.dev/>.
- [37] Google. *Sanitizers: AddressSanitizer, ThreadSanitizer, MemorySanitizer*. Visited: 2024-05-22. 2024. URL: <https://github.com/google/sanitizers>.
- [38] The Phoenix Team. *Phoenix Framework*. <https://www.phoenixframework.org/>. Visited: 2024-05-23. 2024.
- [39] The Elixir Team. *Elixir Documentation*. <https://elixir-lang.org/docs.html>. Visited: 2024-05-23. 2024.
- [40] The Sobelow Team. *Sobelow Documentation*. <https://hexdocs.pm/sobelow/readme.html>. Visited: 2024-05-23. 2024.

- [41] Satish Narayana Srirama, Freddy Marcelo Surriabre Dick, and Mainak Adhikari. “Akka framework based on the Actor model for executing distributed Fog Computing applications”. In: *Future Generation Computer Systems* 117 (2021), pp. 439–452. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2020.12.011>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X20330739>.
- [42] The Akka.NET Team. *Akka.Analyzers*. <https://github.com/akkadotnet/akka.analyzers?tab=readme-ov-file>. Visited: 2024-05-23. 2024.
- [43] Kuo-Yi Chen, J. Morris Chang, and Ting-Wei Hou. “Multithreading in Java: Performance and Scalability on Multicore Systems”. In: *IEEE Transactions on Computers* 60.11 (2011), pp. 1521–1534. DOI: 10.1109/TC.2010.232.
- [44] Dries Buytaert et al. “Bottleneck analysis in java applications using hardware performance monitors”. In: *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA ’04. Vancouver, BC, CANADA: Association for Computing Machinery, 2004, 172–173. ISBN: 1581138334. DOI: 10.1145/1028664.1028735. URL: <https://doi.org/10.1145/1028664.1028735>.
- [45] Huiqing Li and Simon Thompson. “Multicore profiling for Erlang programs using percept2”. In: *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*. Erlang ’13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, 33–42. ISBN: 9781450323857. DOI: 10.1145/2505305.2505311. URL: <https://doi.org/10.1145/2505305.2505311>.
- [46] Aston Parlindungan Tampubolon, Bayu Hendradjaya, and Wikan Danar 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.
- [47] Erlang/OTP Team. *Erlang – gen_server*. Version 5.2.2, Visited: 2024-05-15. Erlang/OTP.
- [48] István Bozó and Melinda Tóth. “Analysing and visualising Erlang behaviours”. In: *AIP Conference Proceedings* 1738.1 (June 2016), p. 240004. ISSN: 0094-243X. DOI: 10.1063/1.4952023. eprint: https://pubs.aip.org/aip/acp/article-pdf/doi/10.1063/1.4952023/13138228/240004\>_1_online.pdf. URL: <https://doi.org/10.1063/1.4952023>.

List of Figures

2.1	Example of the Semantic Program Graph	11
2.2	Example of the Semantic Program Graph	11
2.3	Example of the Semantic Program Graph	12
2.4	WombatOAM showing many metric, including atom count	14
3.1	Methodology	16
3.2	Match tree example	23
4.1	WombatOAM monitoring the atom count	41

List of Algorithms

1	Get the atom table from an index	19
2	Atom exhaustion source code fragments, based on atom table	21
3	Find Unprocessed Messages	25
4	Find Unprocessed Messages with Initial Call	27
5	Closure of a Node	29
6	Check Node Initiation	29
7	Find Send Expressions for a Single Message	29
8	Fetch Function Clauses	30
9	Convert a Receive Pattern to Regular Expressions	30
10	Find Send Expressions for Multiple Messages	30
11	Get Send Expressions from Receive Node	31
12	Get long-living processes	34
13	Get long living ports	37

List of Codes

2.1	Example Erlang module	7
2.2	Message passing in Erlang	8
2.3	Message passing in Erlang	9
3.1	Example on possible atom exhaustion	17
3.2	Get n-th atom[28]	18
3.3	Example atom table	22
3.4	Output of demonstration (atom exhaustion)	23
3.5	Example module on Message Passing	24
3.6	Example of a general receive	26
3.7	Example module on Process Count	32
3.8	Example module on Port usage	35
3.9	Example module on Port usage with message that never arrives	36
3.10	Example module on Port usage, with option to stop	36
4.1	Atom table of MongooseIM	40
4.2	Function that is probably causing the atom exhaustion	40
4.3	Starting the process in extauth (src/auth/extauth.erl)	42
4.4	Sending messages in extauth	42
4.5	Flushing the buffer in extauth	43
4.6	Flushing the buffer in extauth	43