Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Gview: Efficient graph visualisation for RefactorErl

TDK thesis

Supervisor:
Melinda Tóth, István Bozó
Assistant lecturer

Author:
Komáromi Mátyás
Computer Science BSc
2nd year

Budapest, 2018

Abstract

Graph visualisation is a well-known research field of informatics. There are countless algorithms readily available in connection with this topic, however, the aim of most of these algorithms is the static display of graphs. In this thesis, we show an approach that is efficient enough to provide real-time browsing of Semantic Program graphs of higher complexity. With the presented method the user is able to traverse these graphs in different levels of detail referred to as views. The discussed graph visualisation tool (Gview), as part of the RefactorErl tool, can aid users in understanding of Erlang projects and codes. RefactorErl is a static program analysing tool, that stores the source code of the analysed project in Semantic Graphs. A large part of the challenge of this task lays in the amount of data needed to be displayed. Even in small projects, these graphs can become so huge that static plotting tools become unusable for generating a clear layout in real-time. The new component of RefactorErl, Gview, is able to handle large Erlang applications as well.

Contents

1 Introduction **4**

1.1 RefactorErl 4

1.2 Contribution 5

1.3 Gview 5

2 Background **6**

2.1 RefactorErl 6

2.2 The Semantic Program Graph 7

2.3 Flib 7

3 Data transportation **8**

3.1 Static data transfer 8

3.2 Dynamic data transfer 9

3.2.1 Input consistency 9

3.3 Comparison 10

4 Rendering environment **12**

4.1 Layout generation 12

4.2 Related libraries 12

4.3 GUI Framework 13

4.4 Mesh Generation . . . 13
4.5 Dynamic Level of Detail . . . 14
4.6 Anti-aliasing . . . 15

5 Layout generation **16**
5.1 CPU implementation . . . 17
5.2 QuadTrees . . . 17
5.3 GPU implementation . . . 17
5.4 Caching . . . 17

6 Graph views **19**
6.1 Main view . . . 19
6.2 Module view . . . 22
6.3 Function view . . . 22
6.4 Visual elements . . . 25
6.5 Future views . . . 25

7 Evaluation **27**
7.1 Measuring environment . . . 28
7.2 GreenErl . . . 28
7.3 Mnesia . . . 29
7.4 Orber . . . 29
7.5 Stdlib . . . 30
7.6 Sasl . . . 31
7.7 CosTime . . . 31
7.8 Snapshots of Mnesia main view . . . 32

8 Related work **35**

8.1 Graphviz . . . 35

8.2 Wolfram Mathematica . . . 35

8.3 MSAGL . . . 35

8.4 Gephi . . . 36

9 Conclusions **37**

Bibliography **38**

Chapter 1

Introduction

Graph visualisation is a popular research topic, and several algorithms and tools exist that are ready to use. However, the increasing size of the nodes and links among them to visualise on the graph makes the layout calculation more complicated and slow.

Graph visualisation is often used in tools supporting static and dynamic source code comprehension. It is very convenient to denote the relations/dependencies among program entities using a graph view.

1.1 RefactorErl

RefactorErl [1, 2] is a static source code analysis and transformation tool for Erlang [3]. Besides the more than 20 refactorings, the tool provides several functionalities to support program comprehension: semantic queries, static-dynamic call analyses, data-flow analysis, dependence analyses, etc. The results of several analyses can be visualised as graphs.

Also, the tool itself uses a graph based intermediate representation for the source code: the Semantic Program Graph [4]. The SPG contains lexical, syntactic and semantic information about the source code. These three layers generates huge amount of data (nodes and edges) from the source code. Even for a module with few hundreds of lines of code (LOC) the standard visualisation tools, such as Graphviz [5], are hardly able to generate a proper view.

Although, it depends on the complexity of the source code, but in general, there are 50 times more nodes and edges in the graph than lines of code in the source code.

When analysing millions of LOC in industrial scale software, or when analysing single Erlang application having more than twenty thousands of LOC, the graph visualisation is almost impossible. Thus we decided not to visualise the entire graph, but only the relevant parts for the user. We needed a graph that can be traversed fully interactively, switching between the levels of information.

1.2 Contribution

Understanding the structure of a large codebase is a well-studied, important problem, regardless the goal being the introduction of a new developer to an existing project or reviewing, structuring or even debugging code. RefactorErl is great tool for refactoring Erlang programs, it also aids users in code comprehension. Providing information to the user visually has already been considered in RefactorErl, plotting parts of the Semantic Program Graph using various approaches were tried, however the readily available tools focus on plotting graphs in a static manner.

The main contribution of this paper is a solution to the above presented problem and providing an efficient graph visualisation tool for RefactorErl to aid code comprehension. We provide a new component, *Gview* for RefactorErl that is capable of handling real Erlang projects. We demonstrate different views available in the new component through which we can communicate information to the user on the analysed program, and evaluate the performance on different open-source projects.

Part of the results we show in this thesis was submitted as an article in 2018 June to the international MACS conference where we will present Gview [6]. The paper is also waiting to be accepted into the journal of the conference [7].

1.3 Gview

The visualisation process can be broken down into five stages as follows: First the semantic graph is given by RefactorErl, this graph, or part of it, is extracted by the data transfer component of Gview, then layout computations take place and with the resulting layout the graph is plotted. What connects the user to this process is the interaction component of Gview. The component view of Gview is shown on Figure 1.1

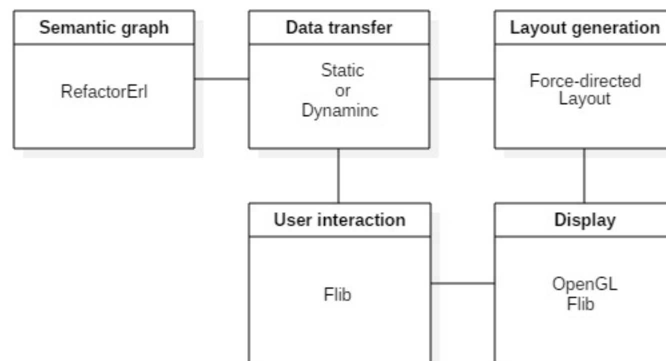


Figure 1.1: Abstract component view of Gview.

Chapter 2

Background

Erlang is a general-purpose functional programming language and runtime environment developed by Ericsson. The language has built-in support for concurrency, distribution and fault tolerance. It is used in several large telecommunication systems from Ericsson to WhatsApp. It has the ability to handle multiple threads (Process in Erlang terms) at the same time without consuming double CPU processing power. Unlike C program thread in which each thread uses separate resources from the CPU.

2.1 RefactorErl

RefactorErl is an open-source static source code analyser and refactoring tool for Erlang, developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. The phrase "refactoring" means a preserving source code transformation, so while you change the program structure you do not alter its behaviour. RefactorErl was built to refactor Erlang programs. Erlang is a dynamically typed functional programming language, thus to gather all of the necessary information for a behaviour preserving transformation is not straightforward. Therefore, a Semantic Program Graph was built to store the Erlang source code and the information could be computed by static analysis [4, 1].

The main focus of RefactorErl is to support daily code comprehension tasks of Erlang developers. Among the features of RefactorErl is included a user-level Semantic Query Language (SQL), that can assist Erlang developers in everyday tasks such as program comprehension, debugging, finding relationships among program parts, etc. RefactorErl represents the Erlang source code in a Semantic Program Graph, thus the mentioned queries are transformed to graph traversals. For industrial scale software the size of this graph can become huge, therefore, the processing of the queries may take several minutes or even hours.

2.2 The Semantic Program Graph

The RefactorErl static analysing tool keeps the information extracted through analysation in a special data structure called the Semantic Program Graph (SPG). This graph represents the lexical, syntactic and semantic structure of the analysed program. Typically the lexical, syntactic and semantic elements of a program maps to one node in the SPG, while the connection between these elements maps to edges between these nodes. The SPG is a rooted graph which refers to there always being a special node called the root which does not represent a program unit. The function of this root node is to be the common ancestor of nodes not having one naturally. Care most be taken when traversing the SPG as it may contain directed loops and thus is not a tree. Although it is not a tree the SPG exposes, in a way, hierarchical properties. As an example taking the subgraph of SPG representing the syntactic data of the program we find the nodes of files right below the root node. Below the files we find function froms, going one level deeper we have clauses that build up the previously mentioned forms. Finally on one level deeper there are the symbols of clause names, parameters of clauses and syntactic trees of expressions in the rows of bodies of clauses.

2.3 Flib

Flib [8] is hobby project of Mátyás Komáromi, the author of this paper, started in 2014. The library is a single-author OpenGL development library written in c++ with around sixty thousand LOC. Gview depends on several features that Flib provides, such as OGL context creation, window management, event handling, GUI and graphics. Flib supports creating and handling windows using OS dependant native libraries, OpenGL contexts and objects. C++ classes that represent OpenGL objects of Flib aim to utilise object oriented features such as encapsulation and RAII, thanks to that, working with OpenGL is much easier and safer. With Flib uploading and downloading data to/from the GPU is also very convient as it provides geometric vector classes that can be tightly packed. Flib also utilises a hierarchical GUI system with text and button elements, graphical tools such as shaders and line tessellators and basic linear mathematical tools. On windows platforms Flib uses the standard Windows API for window creation and management, on Linux platforms the XLib windowing system. Consequently OpenGL context creation is done using WGL and GLX respectively. The Flib API documentation generated by doxygen can be found on the online repository [9].

Chapter 3

Data transportation

The Semantic Program Graph of the project being refactored is stored in RefactorErl in a database depending on the used compilation options and environment. To be able to visualise parts of this graph, the data describing the currently selected view needs to be transferred between RefactorErl and Gview. For this end two methods were developed, as a part of Gview, the first one uses the existing capability of RefactorErl to store the SPG as static data and import it as needed, thus this method is applicable in absence of RefactorErl too. The other approach uses dynamic data transfer between the refactoring tool and Gview with the help of the port utility featured in Erlang, thus this second approach can achieve higher response rates.

3.1 Static data transfer

Static data transfer between RefactorErl and Gview utilises the capability of RefactorErl to export the SPG into DOT files. Dot [10] is a general purpose graph describing language used to represent directed and undirected graphs alike, with extra information options on both edges and nodes. Therefore, by parsing this exported dot file into memory we can extract parts (views) of the SPG in Gview. Thus completing the data transfer task presented in the introduction. The architecture of this solution to the data transfer problem can be seen on Figure 3.1.

The main advantage of using dot files is the existing support for it in RefactorErl and the ease of implementing a custom parser in c++. On the other hand using dot files as intermediate data representation is quite rigid. Having a single file changed one must regenerate the dot file in RefactorErl and on every startup Gview has to parse the whole dot file. To remedy these shortcomings dynamic data transfer was introduced. The data-flow of the static data transfer method can be seen on Figure 3.2.

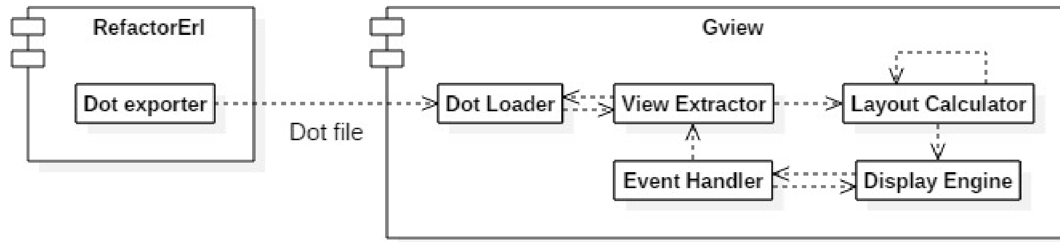


Figure 3.1: Active software components of Gview when static data transfer is in use

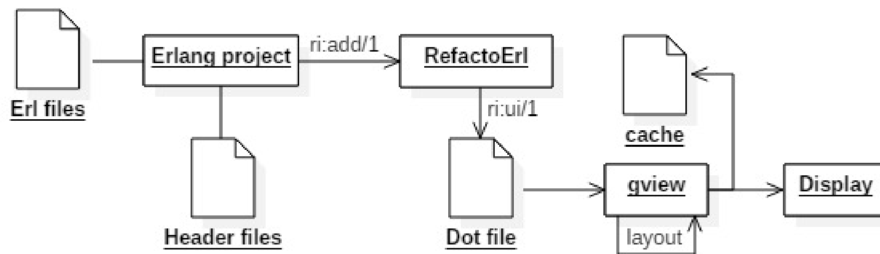


Figure 3.2: Data-flow in the visualisation process when using static data transfer

3.2 Dynamic data transfer

To be able to use dynamic data transfer between the refactoring tool and Gview we created an Erlang extension for RefactorErl. This Erlang part of Gview communicates with the part written in c++ through ports using the `open_port/2` Erlang command. After launching the c++ application, it can receive data and send commands through the standard input and output. This method has the advantage of having the transfer happen in memory without using any disk-space and being faster. The architecture of this solution to the data transfer problem can be seen on Figure 3.3.

3.2.1 Input consistency

It is worth noting that the `open_port/2` command of Erlang opens the standard input of the c++ application in text input mode and thus would be unsuitable for generic data transfer as the interpretation of special byte codes would interfere with the meaning of generic data. This can be solved by reopening the standard streams in binary mode which can only be done in platform specific ways.

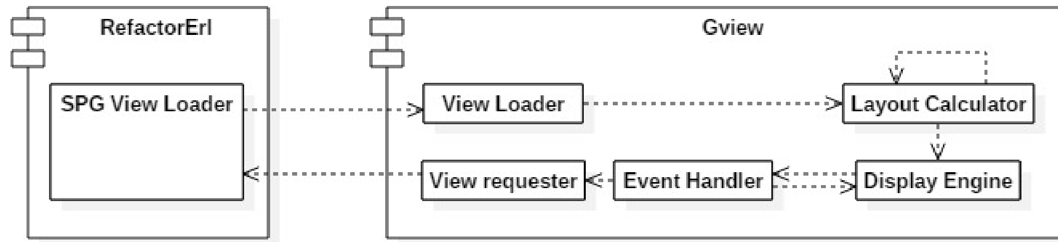


Figure 3.3: Active software components of Gview when dynamic data transfer is in use

The main advantage of using this dynamic data transfer method is that it does not use the disk and that Gview can start without first parsing the whole SPG in. Although it needs RefactorErl to be running in the same time, this method decreases data duplication and load-time. The only disadvantage to it is that it increases the code base and maintaining cost of Gview. The data-flow of the dynamic data transfer method can be seen on Figure 3.4.

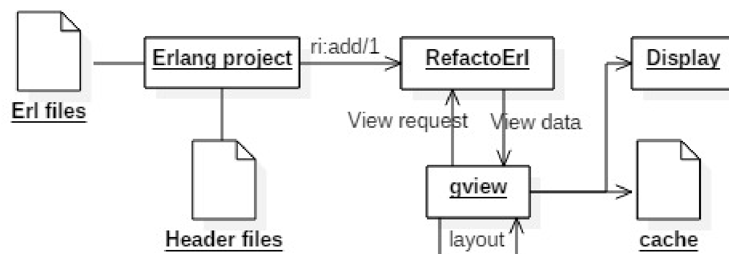


Figure 3.4: Data-flow in the visualisation process when using dynamic data transfer.

3.3 Comparison

Although dynamic data transfer boots up with virtually zero extra time, as one could guess transferring data through memory is much slower than simply reading from memory, as we can see on Figure **??**, we experience a slow-down when using dynamic data transfer, however comparing this extra time of milliseconds to tens of seconds or even minutes of the bootup of the static data load method it is clear that using dynamic data transfer is more advisable. As a future work we plan to combine these two methods as to create one which has both advantages through caching transferred graphs from RefactorErl by the dynamic data transfer method.

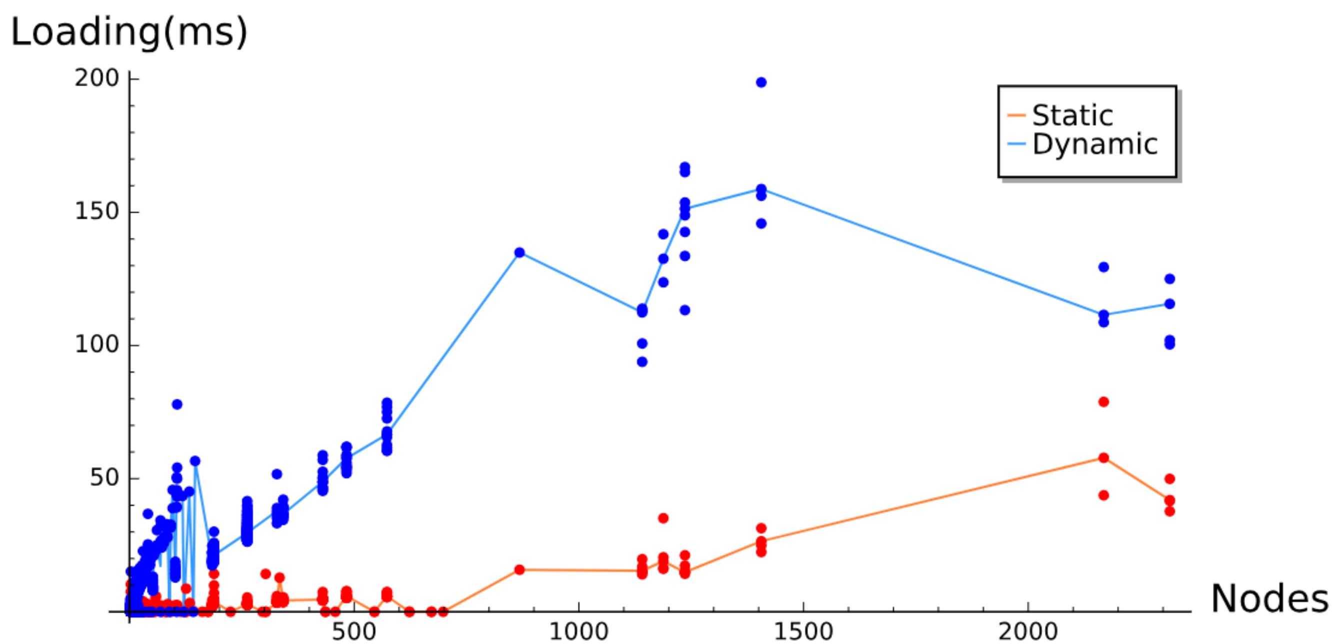


Figure 3.5: Measurements taken on the transfer time of both methods. Points mark measurement results, while the two lines plot the mean of the measurements.

Chapter 4

Rendering environment

The rendering is done using OpenGL [11] with the supporting classes of Flib. We preferred OpenGL because it is hardware close and very fast and, unlike DirectX , portable across operating systems. The drawing data is generated on the fly after each iteration of the layout algorithm, thick lines are tessellated into triangles, circles into regular polygons by the c++ implementation. Extra information on vertices for anti-aliasing is also added in this process. The drawing data is then uploaded to the graphics card and drawn as a single batch, avoiding the cost of setting up many drawing calls. The anti-aliasing is done by our shader program on the GPU.

4.1 Layout generation

Our layout calculating algorithm is a modified version of two-dimensional N-body simulation, known as Force-directed layout [12]. The basic idea is to consider nodes of the graph as charged particles that repulse other nodes and edges as springs which attract adjacent nodes. The simulation is done by performing iterations on calculation of the net force acting on the nodes and changing their position accordingly.

Therefore, by modifying the charge of nodes or the strength of the springs we can change the spacing among edges or nodes. The main difference between the used algorithm and real N-body simulation is that the spring-force is taken to be logarithmic with the distance (not linear). It results in computationally more stable behaviour when two nodes are very close to each other, because the calculated forces can be exerted instantly on nodes.

4.2 Related libraries

Beside the Flib there are many other excellent frameworks for OpenGL development.

SDL (Simple Directmedia Layer) [13] is one of the oldest of these frameworks, it has outstanding wide system and hardware support. However, its interface was designed for C not C++, SDL does not use object-oriented paradigms. SFML (Simple Fast Multimedia Layer) [14] is another excellent choice for OpenGL development, it is fully object oriented (by the c++ binding) with cross-platform support, but it lacks the GUI module. Qt [15] is a professional, robust framework with good GUI and OpenGL support.

Flib (developed by the author of this paper) brings the required OpenGL window and context management classes and wrapper classes for the mostly used GL objects and it has a GUI module which we use for simple text output. It also has a robust event handling system and very convenient graphic classes such as vectors and matrices.

4.3 GUI Framework

Flib provides GUI classes on top of OpenGL. Gview uses Flib to automatically open a window, an OGL context associated with this window and a GUI context which is responsible for storing GUI related data such as fonts and shaders. The GUI context also has a main GUI layout where the application can attach GUI elements. The GUI elements are structured in a tree pattern: each GUI element may have a parent layout, each layout may have any number of children elements, layouts are GUI elements too.

The events, draw calls and update calls are forwarded down the hierarchical structure each time. To detect node selection and change the current view we use the event listener functionality of Flib to translate mouse events such as movements, button down, button up and more complex events as click or double click. The graph transformation is also done with the built-in classes of Flib , these are responsible for calculation of scaling, offset and rotation values that can be used for generation of the displayed mesh.

4.4 Mesh Generation

Although a mesh is generated on the CPU after each iteration of the layout calculation by Gview, using the CPU for this task is perfectly sufficient since the displayed part of the graph is expected to have at most 3000 nodes and 10000 edges (not even Mnesia has this much in the main view) which results in at most tens of thousands of triangles. For this, the task is easily handled by an average CPU these days.

Having the layout (the node positions) of the calculated graph we tessellate it into triangles and lines. The data to be transferred is generated in a batch, thus it can be sent in one operation what makes the upload fast. After the mesh calculation the drawing can be performed with a single call. Uploading to the GPU is done by buffer object streaming, dropping the buffer object before each upload and creating a new one. This enables the GL to complete drawing commands referring the previous buffer while uploading the new one.

Uploading the mesh is done using the designated interface of Flib for simplicity. Tessellation of

lines uses the built-in tessellation functionality from Flib which automatically includes distance-field data needed by the anti-aliasing technique. The data is drawn using the drawing API of Flib as well. Results of the tessellation can be seen on Figure 4.1.

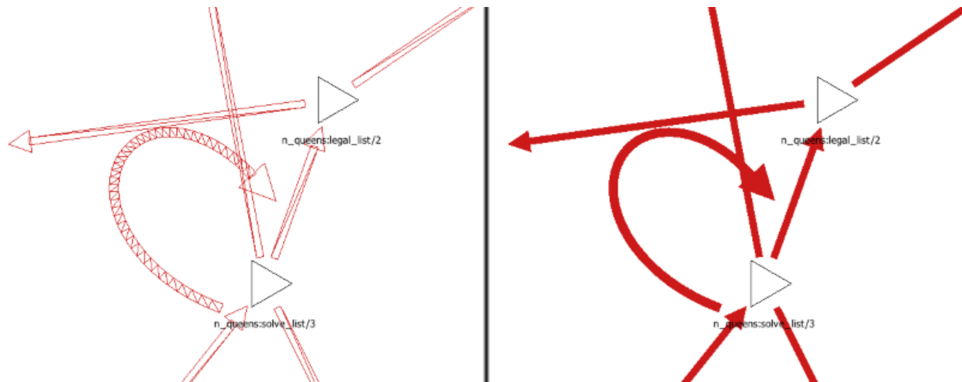


Figure 4.1: Tessellation of the edges (the thickness of the edges was increased for demonstration).

4.5 Dynamic Level of Detail

A very useful technique for graphical applications is Dynamic Level Of Detail [16] (DLOD), it involves altering the detailedness of an object (how many triangles it has or what textures it uses) based on how small that object appears on the screen. This technique is effective because our eyes can not make out the difference on small objects, we use DLOD in Gview to reduce workload on CPU when the mesh is generated. As for an example circles that represent module or file nodes get drawn as regular n-sided polygons. The value of n is based on the zooming, the more it is zoomed in, the larger the value of n is. Application of DLOD is shown on Figure 4.2.

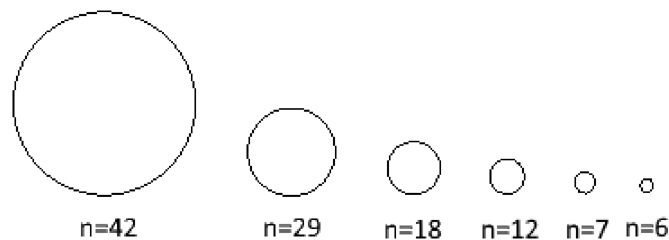


Figure 4.2: Number of sides (denoted as n) of regular polygons representing circles with different zooming level.

4.6 Anti-aliasing

Anti-aliasing [17] is done using distance fields. The main idea is that, if we know the distance from the edge of the primitive when processing a fragment, then we can set the transparency to drop when getting near the edge of the primitive. Thus, having the transparency stored in the alpha channel of the fragment, the OpenGL blending mechanism will ensure it will be displayed transparently. The effect of using anti-aliasing can be seen in Figure 4.3. The distance function is calculated as the minimum of the distances to the edges of the mesh. For this purpose, we store the distances in separate channels (red channel holds the distance from the left edge, green channel from the right edge etc.). Calculating these distances on a per-vertex basis and using the OpenGL built-in linear interpolation functions, we can calculate the distance to the edges of the mesh as the minimum of the interpolated distances.

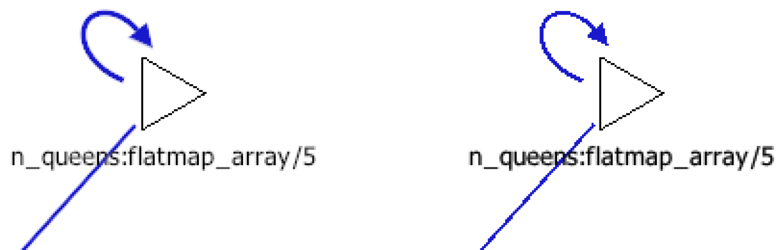


Figure 4.3: Call and recursive edge with and without antialiasing.

Chapter 5

Layout generation

The layout of a displayed graph is defined as a list containing two-dimensional points for each node of the graph. The algorithm described below takes an initial layout and processes it in iterations. We use the Force-directed layout algorithm, that is described in more detail in Section 4.1. For experimenting, we have defined and implemented three different versions of the algorithm (Sections 5.1, 5.2 and 5.3). The efficiency of different implementations can be seen on Figure 5.1.

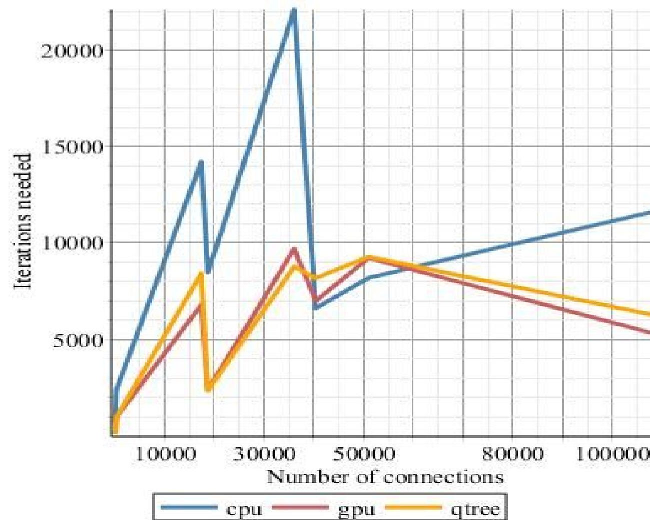


Figure 5.1: Iterations needed to reach final state plotted against number of connections (n^2+e) in view

5.1 CPU implementation

The first implementation uses only the CPU without any optimization, it simply iterates through all node pairs and sums up the forces then applies the forces to the node positions (instantaneous forces). The calculation of forces the nodes exert on other nodes takes $\mathcal{O}(n^2)$ time where n is the number of nodes. Summing the spring forces takes time proportional to the number of edges e: $\mathcal{O}(e)$. Applying the forces on n nodes take $\mathcal{O}(n)$ time.

5.2 QuadTrees

The second implementation uses quad trees for space partitioning to reduce computation time. These quadtrees have regions as nodes, generated by recursively dividing the points in a region into two sets of the equal size, using horizontal splitting lines and vertical splitting lines alternately. This method is also called the Barnes-Hut algorithm [18]. The recursion stops when a certain minimum of nodes in a region is reached. Therefore, when calculating the net force on a node we can traverse the generated tree and approximate the net force exerted by all the nodes in a region of the tree in constant time when it is far enough from the currently processed node. This technique results in $\mathcal{O}(n * log(n))$ time complexity when the distribution of the nodes is even. The problem is that the constant factor of the complexity is quite big as the tree has to be regenerated in each iteration (and not CPU cache friendly). On large views this method outperforms the trivial CPU implementation.

5.3 GPU implementation

The third implementation is the parallel equivalent on GPU of the first one using OpenGL Compute Shaders. We chose OpenGL over Cuda or OpenCL because it has good support for AMD and Intel cards too, integrates nicely with the rest of the drawing code and requires no extra libraries. The graph is sent to the GPU in adjacency matrix representation in a texture image. A kernel is then dispatched for each node of the graph which computes the net force acting on that node. The GPU implementation reuses the calculated data and thus data is only streamed to the CPU once every frame (typically 20 iterations). As a consequence when the number of nodes is smaller than the number shader cores, the GPU is not used efficiently. This issue could be remedied using a divide and conquer approach: each kernel only calculates the net force on a node exerted by a portion of other nodes. Then another shader is dispatched to sum up the subresults.

5.4 Caching

Generating the final layout of a graph view is very computational and time costly. Therefore, caching the generated layouts can save important resources and speed up Gview. Caches need

to be stored permanently, thus they are saved as external binary files using the file streams of the c++ standard library.

One way to implement caching is having a cache file for each view of the graph. This potentially results in thousand of files per graph, although they can be loaded separately resulting in less memory usage.

The other way is keeping one cache file per project. This way, the cache management is easier and clearer. The resulting cache file sizes depend on the opened views. The cache files barely reach 50kB even on the largest test project: Mnesia.

Chapter 6

Graph views

The size of the Semantic Program Graph grows linearly proportionally to the code base with a possibly large constant multiplier. For this fact plotting the whole semantic graph would be pointless and computationally extremely expensive. Consequently we define views of the graph only consisting of nodes and edges that represent data connected with that view. And only plot the currently selected one of these views at a time reducing workload and letting the user concentrate on one aspect of the project.

To test Gview we used several open-source projects such as the Mnesia DBMS and the CosTime application. The images of views described in the following sections were captured from Gview when analysing these projects.

6.1 Main view

Plotting all the modules defined or referenced in a project and their functions can aid the user in getting an overview on the analysed project. To this end we define a view, called the main view, consisting of the module nodes and function nodes of SPG. In this view each function node is linked to the module node they are defined in. The SPG contains modules not just from directly the analysed program but modules referenced by it, consequently this view may also contain standard modules of Erlang. The user is able to browse the currently plotted view using the mouse and the wheel of the mouse. Selecting a new view is by pointing a node representing the new view and then clicking on that node, then the view for that node is loaded. The main views of Mnesia and CosTime is shown on Figure 6.1 and Figure 6.2.

Figure 6.1: The main view of Mnesia containing more than 2500 nodes and edges.

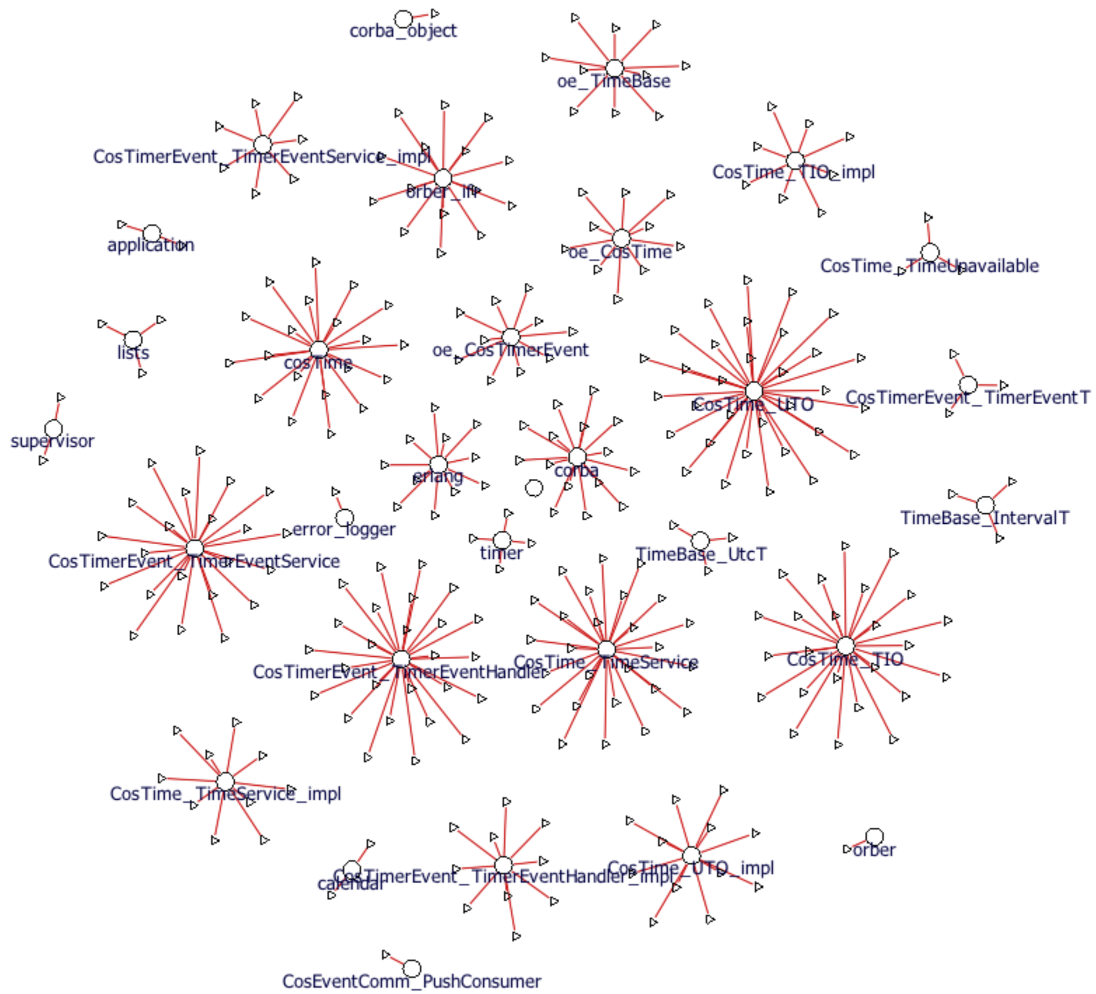


Figure 6.2: The main view of the CosTime application.

6.2 Module view

To be able to focus on a single module of the analysed program the user can use the module view. The module view is similar to the main view, only it displays just one module and the functions defined in that module. It has a ROOT node on which by clicking the user can return to the main view. Selecting a function node and clicking it changes the view yet again and brings up a function view. The view of `eunit_data` is shown on Figure 6.3.

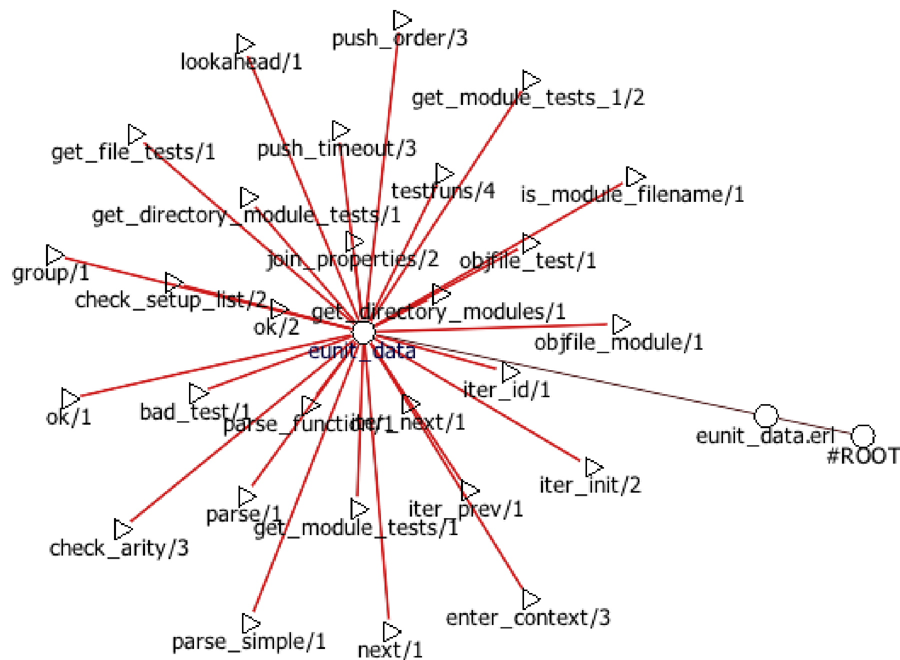


Figure 6.3: The module view of `eunit_data` in the EUNIT application.

6.3 Function view

The most useful of the currently implemented views is the function view. This view focuses on one function node of the SPG, by showing this node and nodes of function called directly or indirectly by the function in focus. The view also contains nodes of functions that directly or indirectly call the function in focus. Deep function call views can be seen on (Figure 6.5). The plot depth of the call graph can be adjusted, this depth is an argument of the view. Currently the depth is not limited, but in future we plan to limit it to a reasonable size.

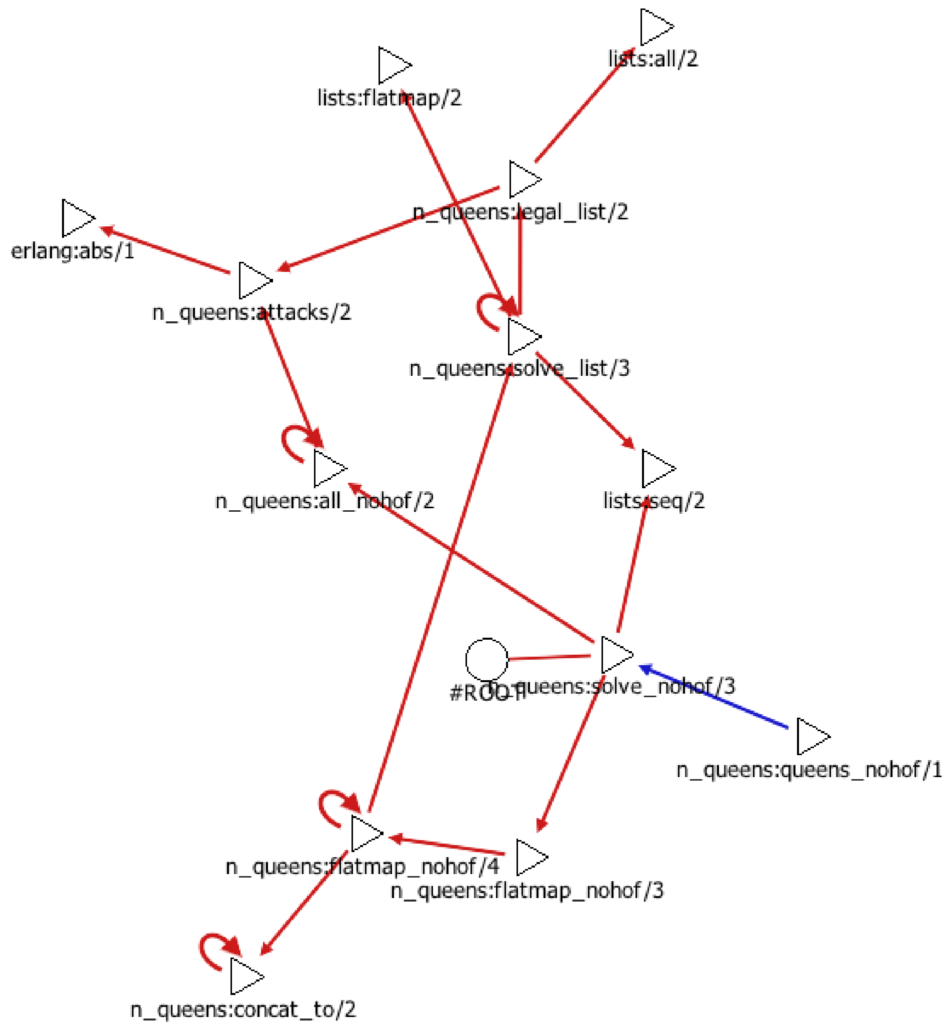


Figure 6.4: Deep function views of the GreenErl project.

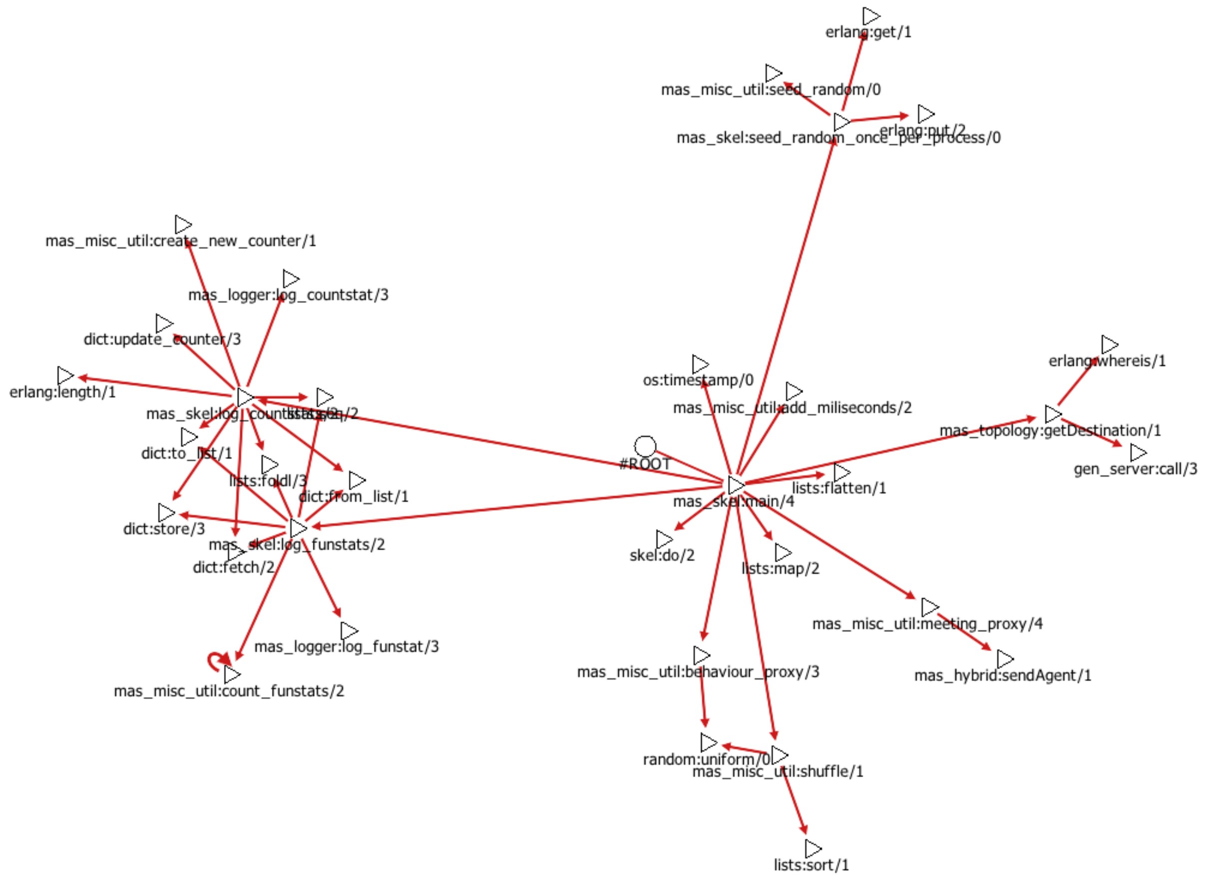


Figure 6.5: Deep function views of the MAS projects.

6.4 Visual elements

Graphical data representation can aid code understanding greatly as we are able to comprehend great amount of information visually. Building on that aspect different shapes and colors can be used to communicate properties of the plotted view to the user aided by Gview. The displaying engine in Gview supports rendering various graphical elements for this very purpose. The shape of the nodes can be specified in the erlang code that extracts the current view and set for example to triangle, square or circle. The width of edges can be adjusted, the colour of the edges, the displayed text below nodes and the shape of the arrows of edges can be customised as well. Through these options the view generator is able to highlight the differences between the semantic entities (function, module etc.) of the displayed graph. Some of the available elements are shown on Figure 6.6.

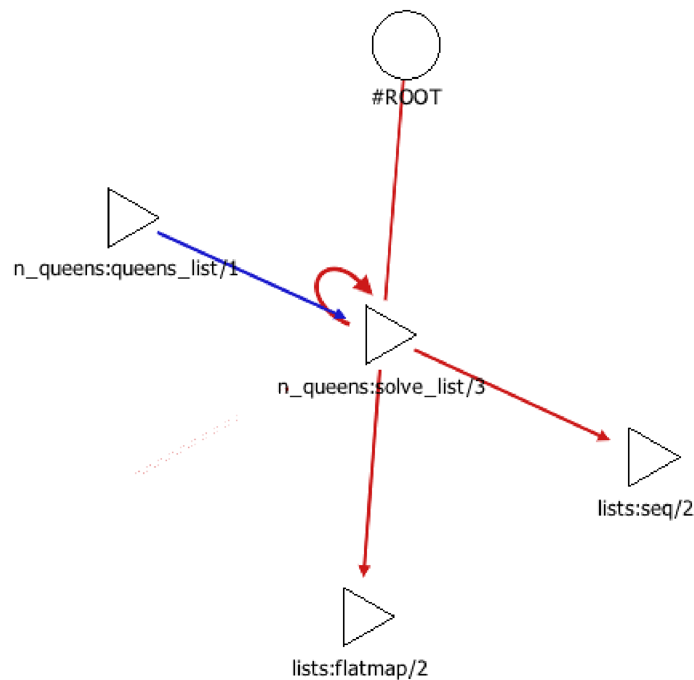


Figure 6.6: Some of the customisable visual elements: triangles, circles, edges, recursive edges, etc.

6.5 Future views

New views can be introduced into gview by implementing the required code to extract these view into the data transfer components of gview. Views have many ways to communicate information, as discussed in Section 6.4 For this reason plotting important refactoring information like the expected frequency of the execution visiting a given function call edge can be integrated into existing and future views alike. Creating new views such as a view of the connection between

modules like dependency is subject to future work.

Chapter 7

Evaluation

To measure the performance of Gview and profile it we tested it on several open-source projects. Following is a description of these projects, measurements were taken on the time of generation, size and loading time of dot files, opening of the dynamic data transfer channel, loading views statically and dynamically and generating the views. We can conclude that the loading of dot files was the major slowdown on startup, wich was eased by implementing dynamic data load. The loading may take more than 90% of the start time using static data transfer. Other events performed on startup, such as window or OGL context creation take negligible time compared to loading and interpreting dot files. Evaluating measurements shown on Figure 3.5 it may be an appropriate further step to cache views transferred from RefactorErl.

When displaying a view the most time is spent on generating the layout (70% and up). Label placement and mesh generation could run in real-time (without layout calculation). Figure 7.5 demonstrates distribution of time spent on different stages. Analysing the charts, it is obvious that described caching mechanism largely improves the efficiency and ensures real-time response on large views.

As an efficiency test, we have compared the execution times of *Gview* and the old Graphviz based dependence graph drawing component of RefactorErl. We have exported a graph of the Mnesia application to a dot file. The graph generation with *Gview* needed cc. 90 seconds, which could be then interactively used. To generate the same graph in SVG with Grapviz took more than an hour, and because of the amount of nodes displayed and slow browsing the content was hard to manage.

A summary of the results of the measurements taken can be seen on Figure 7.1; DOT size refers to the disk-space occupied by the DOT files exported, DOT gen to the amount of time RefactorErl had taken generating the DOT files, DOT load to the boot-up time penalty Gview taken when starting in static data transfer mode, thus loading the whole DOT file, DOpen refers to the amount of time Gview needed to start when using dynamic data transfer, Nodes to the number of nodes in the whole SPG, Edge count to the number of edges in the whole SPG and Func count to the number of functions in total in the SPG.

Project	DOT size	DOT gen	DOT load	DOpen	Nodes	Edge count	Func count
CosEvent	16.82 MB	38.6s	1.6s	<1ms	41 832	145 856	529
CosPropErt	21.98 MB	46.5s	2.1s	<1ms	54 430	187 362	437
CosTime	12.62 MB	26.6s	1.3s	<1ms	31 424	114 336	297
Crypto	9.84 MB	22.3s	1.6s	<1ms	24 551	86 222	174
Edoc	68.45 MB	169.9s	6.1s	<1ms	163 156	648 988	1181
Eldap	7.56 MB	16.5s	0.8s	<1ms	18 196	74 114	174
Erts	61.26 MB	152.8s	5.5s	<1ms	151 937	544 556	1110
Eunit	16.97 MB	41.5s	1.7s	<1ms	41 623	154 544	393
GreenErl	9.02 MB	44.0s	0.8s	<1ms	21 920	88 172	240
Mnesia	141.79 MB	477.1s	24.8s	2ms	332 360	1 374 466	2104
Odbc	4.73 MB	9.9s	0.4s	<1ms	11 659	43 612	86
Orber	131.23 MB	475.7s	16.4s	<1ms	311 514	1 236 636	2213
OsMon	13.32 MB	31.9s	1.3s	<1ms	32 331	125 546	314
OTP_MIBS	0.93 MB	2.0s	0.1s	<1ms	2 366	8 848	42
Sasl	45.84 MB	151.9s	6.1s	<1ms	107 936	438 508	928
SSH	94.71 MB	241.2s	8.8s	<1ms	230 398	913 924	1117
SSL	133.29 MB	343.7s	13.2s	<1ms	325 029	1 298 234	1317
StdLib	283.74 MB	1028.4s	0.0s	<1ms	652 300	1 438 325	3206

Figure 7.1: Summary of measurements taken on opensource projects.

7.1 Measuring environment

Measurements were done on a Windows 10 machine with Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz and 8GB of memory with Intel(R) HD Graphics 6000 integrated GPU, using a TOSHIBA MQ02ABD100H 1TB HDD 5400Hz which could be considered a low-end setup today. Using a 7th generation Intel processor and a much faster SSD could potentially greatly improve the results of these benchmarks.

7.2 GreenErl

GreenErl [19, 20] is a TDK thesis by Gergely Nagy and Áron Mészáros. The ultimate goal of the GreenErl research is to extend the static source code analysis and transformation framework RefactorErl [1, 4]. Their work aim to define static analyses to find those source code fragments that are presumably more energy-intensive than other equivalent solutions and to define a set of refactorings that can be applied, either automatically or semi-automatically, to reduce the energy used by the Erlang programs. The tested and the tester codes were analysed using Gview.

7.3 Mnesia

Mnesia [21] is a distributed Database Management System (DBMS), appropriate for telecommunication applications and other Erlang applications, which require continuous operation and exhibit soft real-time properties. The Mnesia DBMS is written entirely in the Erlang programming language, it was not thought as a general office-based data processing database management system, for it does not replace SQL-based systems. Instead Mnesia exists to support Erlang, where DBMS-like persistence is required. It has more in common with embeddable DBMS such as Berkeley DB than with a SQL database server. "Rows" in tables are represented as records that contain a key value and a data field. This data field may in turn be a tuple containing an Erlang data structure of any complexity [21].

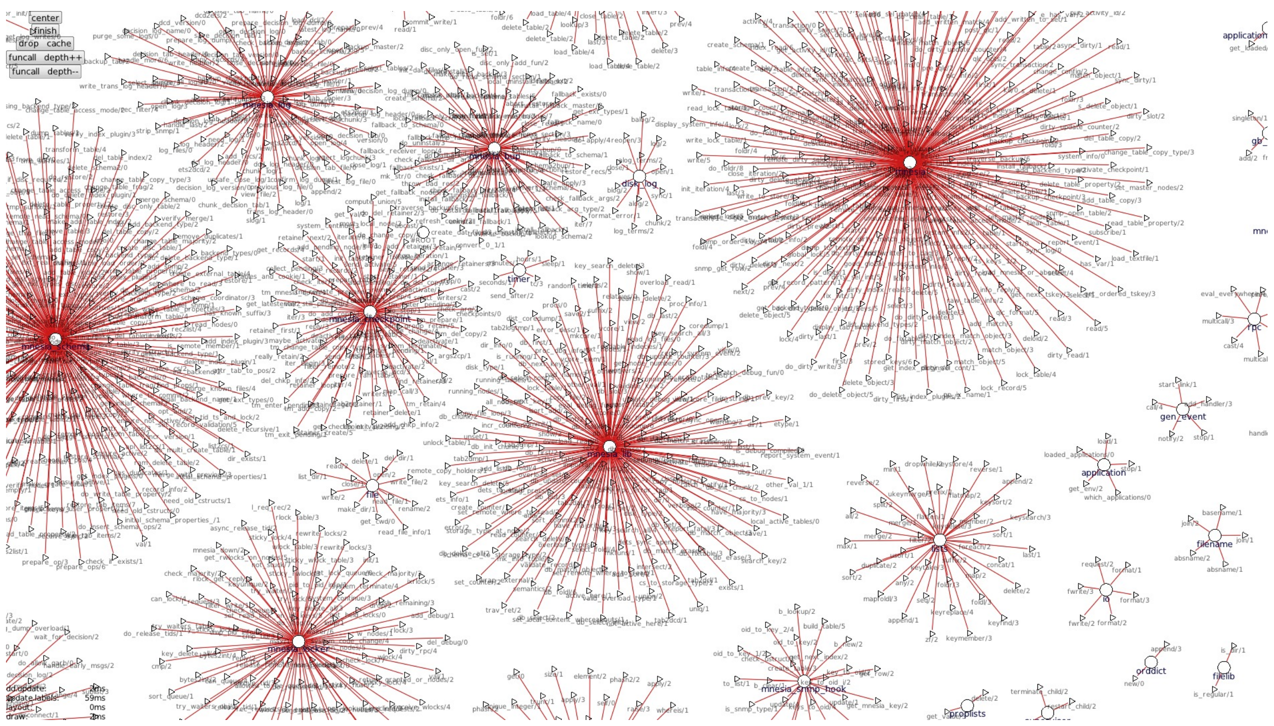


Figure 7.2: The massive amount of functions in Mnesia.

7.4 Orber

Orber [22] is a CORBA [23] compliant Object Request Broker (ORB), which provides CORBA functionality in an Erlang environment. Essentially, the ORB channels communication or transactions between nodes in a heterogeneous environment. The Orber application contains ORB kernel and IIOP support, Interface Repository, Interface Definition Language Mapping for Erlang and the CosNaming Service.

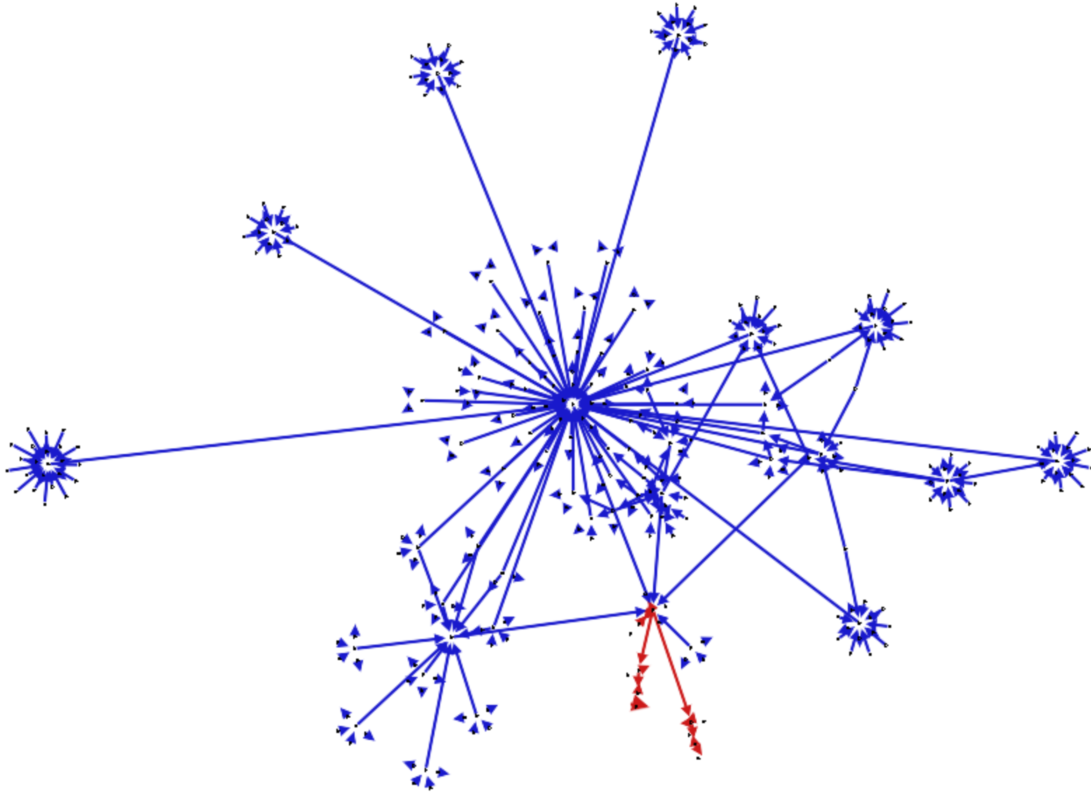


Figure 7.3: Deep function call view in the Orber application.

7.5 Stdlib

STDLIB [24] is mandatory in the sense that the minimal system based on Erlang/OTP consists of Kernel and STDLIB. The STDLIB application contains no services, however it provides the basic functionalities of handling lists, queues, dictionaries and many more.

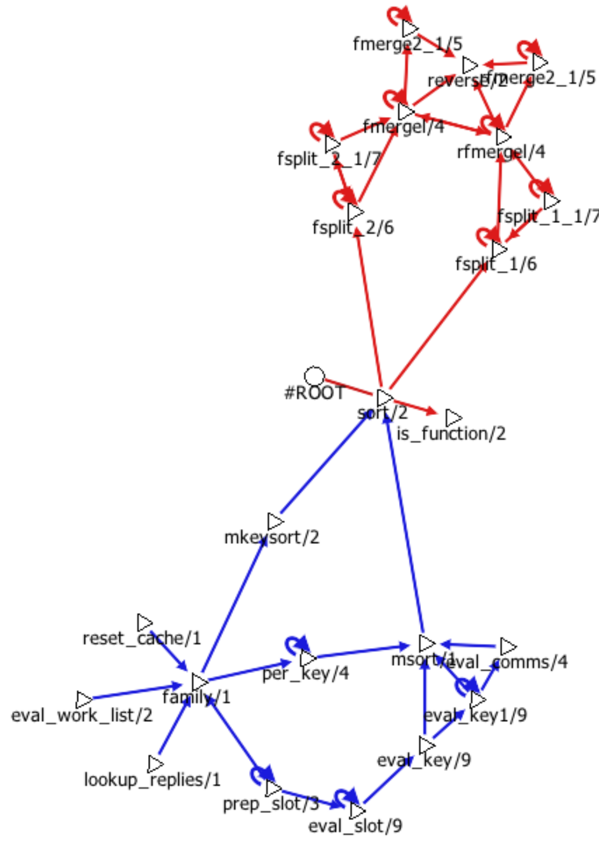


Figure 7.4: Function call view showing the dependency and usage of the sort/2 function in STDLIB.

7.6 Sasl

The System Architecture Support Libraries (SASL) [25] application provides support for alarm handling, release handling, and related functions.

7.7 CosTime

CosTime [26] is an Erlang implementation of the OMG CORBA Time and TimerEvent Services. It has many modules with few functions, thus it was good candidate to test Gview on. Main view of CosTime can be seen on Figure ??.

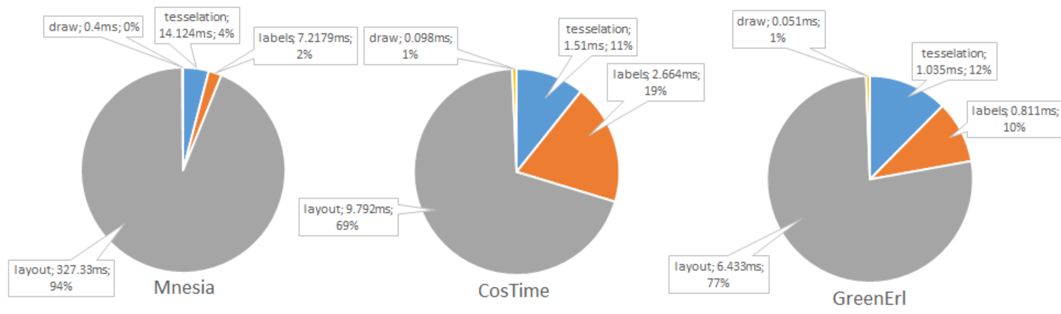
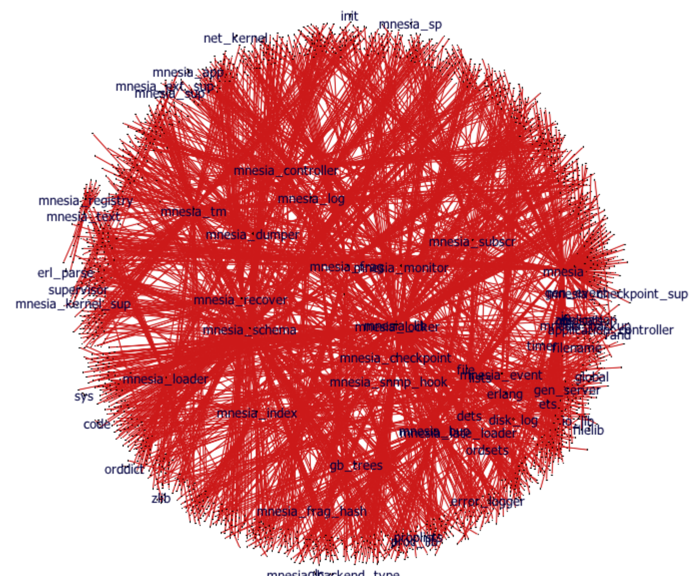


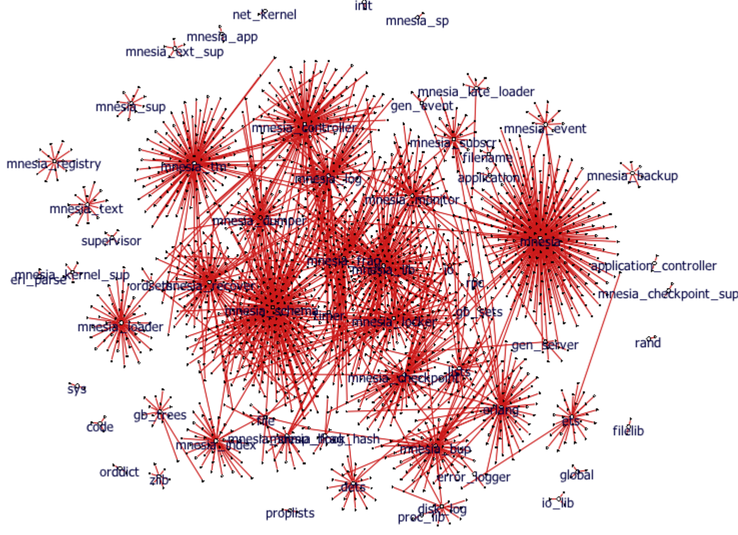
Figure 7.5: The average time taken for layout generation, tessellation, label setting and drawing the view.

7.8 Snapshots of Mnesia main view

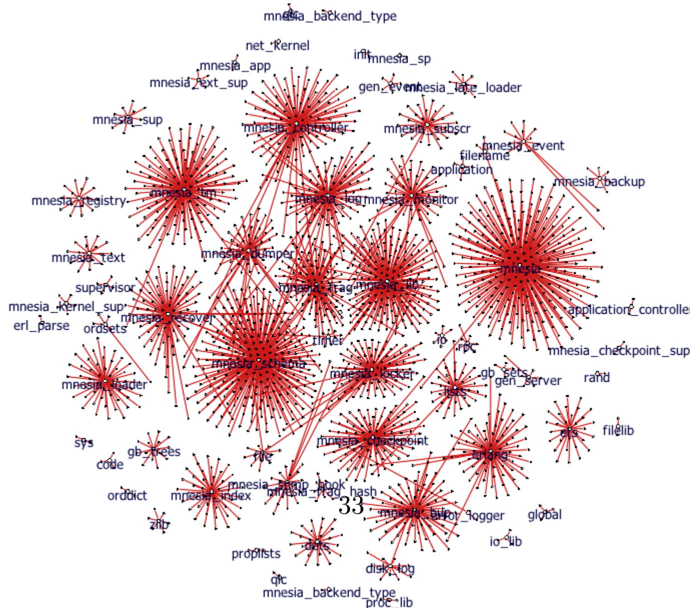
The followings are snapshots of the calculation of the Mnesia main view.



init
mnesia_sp
net_kernel
mnesia_app
mnesia_ext_sup
mnesia_sup
mnesia_controller
mnesia_registry
mnesia_text
mnesia_tm
mnesia_log
mnesia_dumper
mnesia_subscr
erl_parse
supervisor
mnesia_kernel_sup
mnesia_recover
mnesia_schema
mnesia
mnesia_checkpoint
mnesia_loader
mnesia_snmp_hook
mnesia_event
global
gen_server
erlang
ets
sys
code
mnesia_index
dets
disk_log
filelib
ordsets
orddict
gb_trees
zlib
error_logger
mnesia_frag_hash



init
net_kernel
mnesia_sp
mnesia_app
mnesia_ext_sup
mnesia_late_loader
gen_event
mnesia_sup
mnesia_controller
mnesia_event
mnesia_subscr
filename
mnesia_registry
mnesia_tm
application
mnesia_backup
mnesia_log
mnesia_monitor
mnesia_text
mnesia_dumper
supervisor
mnesia
application_controller
mnesia_kernel_sup
mnesia_checkpoint_sup
mnesia_loader
gen_server
rand
sys
mnesia_checkpoint
erlang
gb_trees
ets
code
filelib
mnesia_index
error_logger
orddict
global
zlib
dets
io_lib
proplists
disk_log
proc_lib



mnesia_backend_type
net_kernel
mnesia_sp
mnesia_app
mnesia_ext_sup
mnesia_late_loader
mnesia_sup
mnesia_controller
mnesia_subscr
mnesia_event
filename
application
mnesia_backup
mnesia_registry
mnesia_tm
mnesia_log
mnesia_monitor
mnesia_text
mnesia_dumper
mnesia_frag
mnesia_lib
mnesia
supervisor
mnesia_kernel_sup
application_controller
erl_parse
ordsets
mnesia_recover
timer
mnesia_schema
mnesia_locker
mnesia_checkpoint_sup
gen_server
rand
mnesia_loader
lists
sys
mnesia_checkpoint
ets
filelib
gb_trees
erlang
code
mnesia_index
mnesia_snmp_hook
orddict
error_logger
global
zlib
io_lib
dets
proplists
disk_log
qlc
mnesia_backend_type
proc_lib

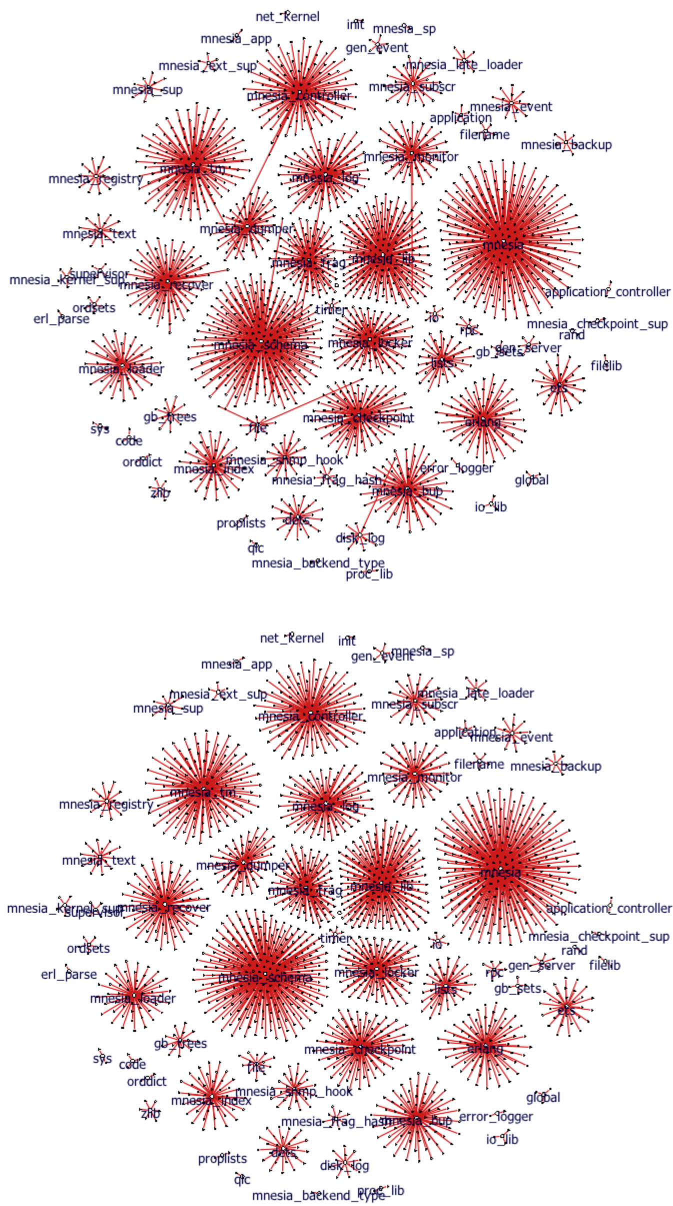


Figure 7.6: Five stages of the generation of the main view Mnesia, taken 4, 25, 35, 46 and 63 seconds after layout generation has started, from a hairy ball of inconsistency to pretty little sea urchins.

Chapter 8

Related work

Graph visualisation is a well-researched field of computer graphics and thus many readily available tools exist supporting it, described below are some of the best known such tools.

8.1 Graphviz

Graphviz [5] (short for Graph Visualization Software) is a package of open-source tools initiated by AT&T Labs Research for drawing graphs specified in DOT language scripts. It also provides libraries for software applications to use the tools. Graphviz is free software licensed under the Eclipse Public License. It supports many input formats, specifications and algorithms for presenting graphs. However, Graphviz is unable to render graphs with high node count as experienced during the development of the user interface of RefactorErl.

8.2 Wolfram Mathematica

Wolfram Mathematica [27]: The Wolfram Language provides functions for the aesthetic drawing of graphs. Algorithms implemented include spring embedding, spring-electrical embedding, high-dimensional embedding, radial drawing, random embedding, circular embedding, and spiral embedding. In addition, algorithms for layered/hierarchical drawing of directed graphs as well as for the drawing of trees are available.

8.3 MSAGL

Microsoft Automatic Graph Layout (MSAGL) [28] is a .NET library for automatic graph layout. MSAGL was developed in Microsoft by Lev Nachmanson, Sergey Pupyrev, Tim Dwyer, Ted

Hart, and Roman Prutkin. Earlier versions carried the name GLEE (Graph Layout Execution Engine).

8.4 Gephi

Gephi [29] is an open-source network analysis and visualisation software package written in Java on the NetBeans platform. The Gephi Consortium, created in 2010, is a French non-profit corporation which supports development of future releases of Gephi. Members include SciencesPo, Linkfluence, WebAtlas, and Quid. Gephi is also supported by a large community of users, structured on a discussion group, a forum and producing numerous blogposts, papers and tutorials.

Chapter 9

Conclusions

RefactorErl framework has several graphical and command-line interfaces, that support refactorings, static code analysis and code comprehension as well. The tool uses a Semantic Program Graph as an intermediate representation of the source code. The SPG includes static semantic information beside the syntactic and lexical information. The conversion of the SPG to an SVG file with Graphviz was possible only on relatively small graphs. There was high demand for an efficient and interactive graph visualisation tool that led us to create the Gview component presented in this paper.

Code comprehension using visual data representation is an efficient way of getting to understanding large programs better. Defining views of the Semantic Program Graph helps in communicating logically connected and useful information, in amount and meaning, to the user.

RefactorErl has support for exporting the SPG to a dot file, thus we employed this functionality and used the dot graph describing file format for representing the graph. These files are then processed by our custom dot parser implemented in c++. The views of the exported graph are generated and visualised using OpenGL and Flib. We have used Flib also for GUI and OGL object management. Although this static data transfer was a great starting point, it turned out that processing/parsing the dot file is the bottleneck in the generation of visual representation. As an aid we created an alternative, dynamic method for extracting the needed view of the SPG, in which we acquire data directly from RefactorErl using the `open_port` functionality of Erlang.

To calculate the layout of the graph we used Force-directed layout generation. To improve the performance of the interactive viewer, the resulted layout is saved to cache files to avoid the continuous need of recalculating. To enhance the visual quality of rendered scene we used antialiasing. Dynamic Level Of Detail is applied to reduce geometry, which is then tessellated using Flib and drawn in a single batch. We made the appearance fully customisable, thus the users are able to use different shapes and arrows for different semantic entities and relations among them accordingly. The user is able to switch between views using the cursor; pointing on a node and clicking brings up a more detailed view associated with that node.

For the described views prooved to be useful we plan to extend them in the future with more

information and define other views of the SPG. Also, using the GPU for parallelisation of mesh tessellation is an appropriate subject for future research.

Part of the results we show in this thesis was submitted as an article in 2018 June to the international MACS conference where we will present Gview [6]. The paper is also waiting to be accepted into the journal of the conference [7].

Bibliography

[1] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[2] Melinda Tóth and István Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer, 2012.

[3] Joe Armstrong. *Programming Erlang.* The Pragmatic Bookshelf, 2nd edition, October 2013.

[4] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling Semantic Knowledge in Erlang for Refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, July 2009.

[5] Graphviz homepage. https://www.graphviz.org/.

[6] Mátyás Komáromi, Melinda Tóth, István Bozó. An Efficient Graph Visualisation Framework For RefactorErl, Abstract accepted to the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-17, 2018.

[7] Mátyás Komáromi, Melinda Tóth, István Bozó. An Efficient Graph Visualisation Framework For RefactorErl, Paper submitted to the Special Issue of Studia Universitatis Babes-Bolyai, series Mathematica, Informaticaand Physica, MACS'18, 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-17, 2018.

[8] Flib project github page. https://github.com/Frontier789/Flib/.

[9] Flib documentation. http://makom789.web.elte.hu/docs/index.html.

[10] Eleftherios E. Koutsofios and Stephen C. North. Drawing graphs with dot. 1991.

[11] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. Opengl(r) shading language. 2004.

[12] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[13] Fabian Fagerholm. Simple directmedia layer (sdl). 2006.

[14] Simple fast multimedia library. https://www.sfml-dev.org/.

[15] Qt | cross-platform software development for embedded and desktop. https://www.qt.io/.

[16] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of the 7th conference on Visualization '96*, volume 25, pages 327–334, 1996.

[17] A. R. Forrest. Antialiasing in practice. In Rae A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, pages 113–134, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[18] N-body algorithms. http://www.cs.hut.fi/ ctl/NBody.pdf.

[19] Gergely Nagy, Áron Mészáros, Melinda Tóth, István Bozó. Towards green computing in Erlang, Paper submitted to the Special Issue of Studia Universitatis Babes-Bolyai, series Mathematica, Informaticaand Physica, MACS'18, 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-17, 2018.

[20] Gergely Nagy, Áron Mészáros, Melinda Tóth, István Bozó. Towards green computing in Erlang, Abstract accepted to the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-17, 2018.

[21] Håkan Mattsson, Hans Nilsson, and Claes Wikström. Mnesia - a distributed robust dbms for telecommunications applications. *practical aspects of declarative languages*, pages 152–163, 1999.

[22] The orber application - an erlang implementation of the corba object request broker. http://erlang.org/doc/apps/orber/index.html.

[23] Steve Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[24] The standard erlang libraries application, stdlib - a mandatory part of systems based on erlang/otp. http://erlang.org/doc/apps/stdlib/index.html.

[25] System architecture support libraries for erlang. http://erlang.org/doc/apps/sasl/index.html.

[26] The costime application. http://Erlang.org/doc/apps/cosTime/cosTime.pdf.

[27] Wolfram mathematica homepage. https://www.wolfram.com/mathematica/.

[28] Microsoft automatic graph layout homepage. https://www.microsoft.com/en-us/research/project/microsoft-automatic-graph-layout/.

[29] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Third International AAAI Conference on Weblogs and Social Media*, 2009.