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

Visualising software dependencies to support code comprehension

Gview

TDK thesis

Supervisor:
Melinda Tóth, István Bozó
Assistant professor

Author:
Mátyás Komáromi
Computer Science BSc
3rd year

Budapest, 2019

Abstract

It is always a great challenge to maintain industrial scale software. It requires to fully understand and be aware of the different components and their connections to avoid introducing software errors. Aiding the process of software maintenance by visualisation is a very timely topic, as it is easier for humans to understand visualised information. In our paper, we introduce Gview, a new tool for interactive graph representation. The presented graph is interactive and utilises the GPU to speed up layout generation. As proof of concept, we integrated Gview with RefactorErl. Refactorerl is a source code analyser and refactoring tool that also supports code comprehension. The tool represents the syntactic and semantic information in the Semantic Program Graph, containing a massive amount of nodes and edges.

Contents

1 Introduction **3**

1.1 Background 4

1.2 Previous work 5

2 Gview **6**

2.1 Data Transfer 6

2.2 Layout 9

2.3 Plotting 11

2.4 User interaction 13

2.5 Integration with RefactorErl 14

3 Optimising the Force Directed Layout generation **17**

3.1 The algorithm 17

3.2 Using higher order methods 19

3.3 Methodology 20

3.3.1 Linear parallelisation 20

3.3.2 Refining work per thread 22

3.3.3 Memory usage and synchronization . . . 23
3.4 Evaluation . . . 25

4 Using Gview for code comprehension **28**

4.1 Call graph view . . . 30
4.2 Enhanced syntax view . . . 32

5 Related work **35**

5.1 IslandViz . . . 35
5.2 Graphviz . . . 36
5.3 CodeCity . . . 36
5.4 CityVR . . . 37
5.5 Sourcetrail . . . 37
5.6 Understand . . . 38
5.7 d3js . . . 38
5.8 Gephi . . . 39
5.9 GoJS . . . 39

6 Conclusion and future work **41**

Bibliography **42**

Chapter 1

Introduction

Visualisation of software is mapping a software system and its architecture to a visual representation. The created view can be static, interactive or even animated [1].

The visual representation of software may improve the productivity of developers, as it supports code comprehension, helps to find inconsistencies and to improve the quality. The software visualisation extracts and combines closely related information of the system. The visualised representation is easier to comprehend than gathering the same information manually from the source code.

RefactorErl [2] is a static source code analyser and transformation tool for Erlang. It aims to support the everyday code comprehension tasks of the Erlang developers. Since presenting the semantic information about the source code is quite natural on a graph, we started the Gview project as a new graph visualisation component for RefactorErl. The main goal was to be capable of rendering huge Semantic Program Graphs [3] as well.

The main contributions of this paper is the introduction of Gview that is a new interactive graph visualisation tool. Gview was designed to utilise the GPU resources, to provide different layout generation mechanisms, to support a generic data transfer protocol and an easy to use interface for different tools. We present the integration of Gview with RefactorErl and some use cases. However, the tool was designed for software visualisation, its usage has no restrictions.

The rest of the paper is structured as follows. The rest of this Section introduces the tool RefactorErl and the first prototype of Gview. Chapter 2 presents details about the generalised and RefactorErl independent Gview and its integration with RefactorErl. Chapter 3 introduces the methodology for force-directed layout calculation and

the applied optimisations for GPUs. Chapter 4 describes how to use Gview for code comprehension. Finally, Chapters 5 and 6 present related work and conclude the paper.

1.1 Background

Erlang [4] is a functional, concurrent programming language that was designed to build distributed, soft real-time, robust, fault-tolerant applications. Although the language is functional, the industrial scale applications require tools to support software maintenance, code comprehension, refactoring, etc.

RefactorErl [2] was designed to provide a static source code analyser framework with thorough static semantic analyses for the programming language Erlang. The tool offers a wide range of source code transformations as well. RefactorErl represents the source code in a so-called *SPG*, the Semantic Program Graph [3]. The *SPG* contains the syntax tree of the source code enhanced with lexical information, and different analysers are adding semantic information about the source code relations.

The tool [5] provides more than twenty refactoring steps for the users. Besides the well-known renaming, moving, etc. transformations RefactorErl supports parallelisation by refactorings [6, 7, 8].

RefactorErl aims to support code comprehension in various ways. It defines a query language [9] to allow user-defined semantic queries about the source code and present the gathered information in different formats. For example, the web interface of the tool lets the user navigate between the source code and the results of the queries.

It also implements dependence analyses of software components and is able to utilise the dependence relations for software clustering. RefactorErl defines a duplicated code detection [10] and elimination component.

RefactorErl is able to handle industrial scale applications [11]. For that size, the Semantic Program Graph and also the gathered views are so huge that a static graph visualiser is not able to present it to the developer. However, it is not necessary to show the entire graph to the user at once. In most of the time the user wants to check a filtered subgraph, a predefined view only, and explore the rest of the graph interactively.

Therefore we started to build Gview as part of the RefactorErl project to visualise different views of the Semantic Program Graph. The very first version of Gview [12] was only able to visualise the module/function views of the *SPG* that was printed to a static *dot* file [13].

1.2 Previous work

The techniques and algorithms presented in this work are based on our previous work with which we aimed to serve the very same purpose of code comprehension support. However our initial approach [14], presented a year ago targeted efficient graph visualisation for RefactorErl only and was not suitable for other applications.

This fact was due to our data transfer techniques. The static data transfer method that we employed could only work with tools that already supported exporting the whole graph that is to be plotted into the chosen intermediate representation which was the *DOT* format. To overcome this limitation and to remedy the slow startups that the parsing of the intermediate representation caused, we introduced the dynamic data transfer method. This method depended heavily on the structure of the *SPG*, for example Gview would deduce the shape of the nodes by the type of the actual entity that the node represented, such as triangles for functions.

When generating the layout for the current view, the user only had one option for layout generation, the *FDL* algorithm which was implemented using Euler's method. This numeric algorithm potentially requires a minuscule step size and thus a great amount of steps to avoid instability and can get caught up in loops.

In our current research, we present an extension for the dynamic data transfer method with a generic graph plotting protocol and therefore enable the integration with other applications than that of RefatorErl. We also present a new, parallelised version of the *FDL* with various optimizations, based on higher order methods, and a new option for layout generation in Gview, the Hierarchical Layout Generation. For the user interaction handling and displaying tasks, we kept our choice of using Flib and OpenGL as previously.

Chapter 2

Gview

Our solution for the problem of visualisation can be broken down into four sub-tasks: data transfer, layout generation, displaying the graph with the generated layout and handling user interactions. We define different views of the graph, for example for a given variable, all the different ways data could be assigned to the variable forms a dataflow view. These views have a specific meaning in the context of the host application (RefactorErl) and thus must be generated by it and converted into a visual description containing all the desired graphical properties of the resulting plot such as line thickness and colour. Figure 2.1 shows an overview of internal structure of Gview.

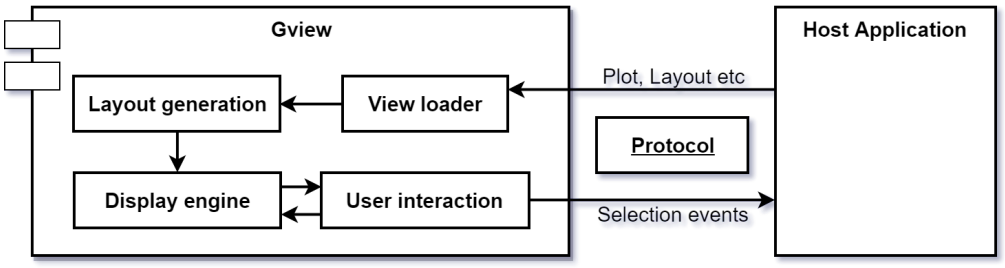


Figure 2.1: Representation of the inner division of sub-tasks in Gview.

2.1 Data Transfer

The first sub-task is to transfer this representation. Our initial approach was based on an intermediate data storage such a file formatted in the DOT language of Graphviz, where RefactorErl would export the whole SPG, often resulting in hundreds of megabytes in size and thus in slow startups. This method also brought the additional cost of our

method being dependant on the specifics of the Semantic Program Graph. To improve the visualisation, we introduced dynamic data transfer [14]. The transfer is done through our binary protocol which was designed to be host independent while being as efficient as possible while not forming a performance bottleneck. The protocol defined in our work also has a control layer that enables host applications to programmatically change properties of the plot such as the used layout algorithm.

Our protocol is a byte protocol, meaning it can be used over any stream that is able to transfer bytes between applications such as TCP/IP. Our implementation with RefactorErl uses the Erlang Ports interface which builds on the standard input and output file handlers of the graph plotter. First, the host application and the plotter exchange a two-way handshake message as seen on Figure 2.2, stating the used version of the protocol, which is currently 1.0, resulting in an error if the two are not compatible.

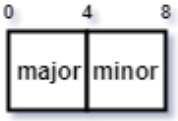


Figure 2.2: The initial two-way handshake message. Not a large but a rather important message.

After that, the plotter application is up and running, and is waiting for new commands. The command string is sent in ASCII encoding, while label string in UTF-8. The first important command is to change the layout generation algorithm. For example, to change the layout from force-directed to hierarchical. To accomplish this, the host must send two messages: "set_layout" and the id of the new algorithm, for example, "layered" for layered hierarchical as seen on Figure 2.3.

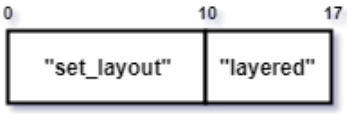


Figure 2.3: Example of a message sent by the host to set the used layout algorithm to layered.

The host can also issue a plot command via sending "set_view" then sending the description of the graph. A summary of this message can be seen on Figure 2.4.

Sending a graph description begins with sending four integers: the number of nodes, the size of the node palette and edge palette and the number of selectors (details on selectors in Section 2.4). Each entry of the node palette describes the appearance of a given type of nodes, consisting of the radius and the shape of the node. Similarly, one entry of the edge palette holds a preferred width and colour of the edge and the shapes on the end of the given edge. This visual description of the nodes and edges can be extended

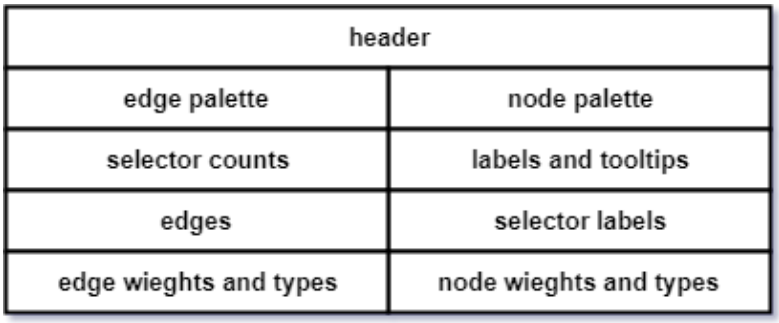


Figure 2.4: Summary of the complete plot message.

in the future. After the integers, the entries of the node palette and the edge palette are sent, each in a separate message. In the next messages, the labels of the nodes, the tooltip strings, the selector counts and selector labels are transferred. After that, a list of integers is sent for each node, denoting the neighbouring nodes of the current node (edge list representation). These integers hold the local id of the nodes which is the number of the given node and thus independent of the global id (used in the host) of the node. The last step is to send node weights, node types, edge weights and edge types. Types are integer lists while weights are lists of floating point numbers. The structure of the header and the palette entries can be seen on Figure 2.5.

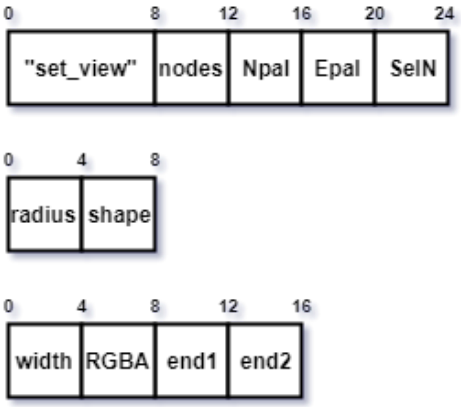


Figure 2.5: The header message and the palette entry messages of a set_view message.

Node and edge types indicate the id of the entry in the palettes (node and edge palette respectively) at which the description of the given node or edge is located, with this palette method, a huge bandwidth reduction can be achieved for a lot of nodes and edges often share these details. Weights, on the other hand, can mean different properties depending on the currently used layout algorithm but in general they can be understood to represent the importance of a node or edge, for example when using the force-directed layout, more important edges are generally shorter and more important nodes repel

other nodes stronger. The key idea to make the transfer fast is that since each message is preceded by a four-byte integer, containing the length of the message, we can create a byte buffer from all the messages and let the used implementation stream the bytes in an efficient manner.

2.2 Layout

The second task we defined is to generate a suitable layout for the view that is currently being plotted. Many layout algorithms have already been developed as this is an important field of visual computing. In our previous paper [15], we presented an efficient GPU parallel extension to the famous Force-Directed Layout algorithm and also a cheap layered layout based on The Sugiyama Method.

The main idea in the Force-Directed Layout (FDL) is to build a physical system corresponding to the graph; each node gets represented by a negatively charged body while edges become springs between these bodies. According to the Coulomb law and Hooke's law, given their position, the acting forces can be expressed on each body, which results in a differential equation system with time as the variable of the unknown function. The goal is to find an approximation for the unknown function. The fixed-point of this function represents a physical equilibrium of the system, that will be the final layout generated by the algorithm. To approximate the unknown function, we use the higher-order Runge-Kutta methods, which are excellent candidates for massive parallelisation. In our previous paper, we worked out the details of parallelising this algorithm in a highly efficient manner and cover various memory and workload optimisations too.

An example of a Gview generated force-directed layout for a 8 by 8 grid is shown on Figure 2.6.

The Layered Hierarchical Layout [16, 17, 18] generation algorithm starts with assigning nodes to layers, which layers will determine the Y coordinate of the final position of the node. In the next step, the algorithm calculates orderings of nodes on each layer as to minimise edge crossings, since this is a very hard (NP-complete) task even for two layers, different approximations are employed here. In the final step X and Y coordinates get calculated for each node. Out version of the algorithm aims to assign more horizontal space for nodes with more descendants on deeper layers, which tends to produce visually pleasing layouts for graphs that possess a tree-like structure. The layer assignment and the crossing minimisation can be done in various ways and can be found in many related research papers.

An example of a Gview generated layered layout for a relatively small random tree is shown on Figure 2.7.

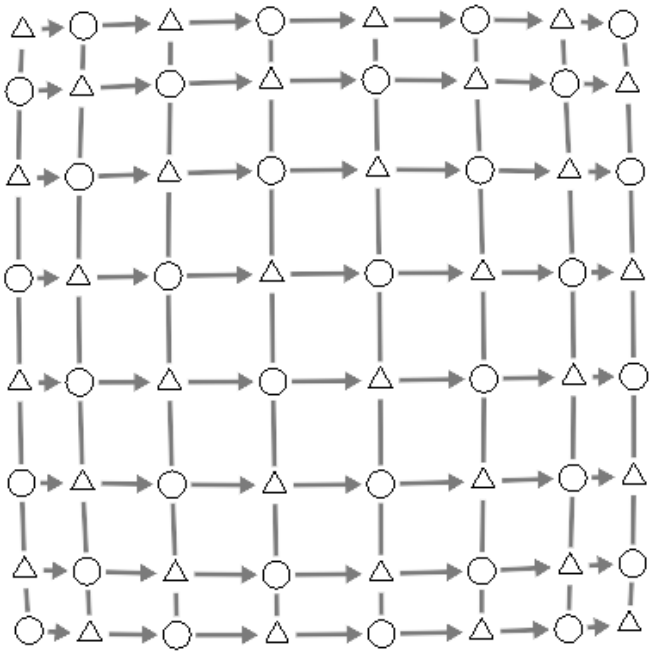


Figure 2.6: Force-directed layout

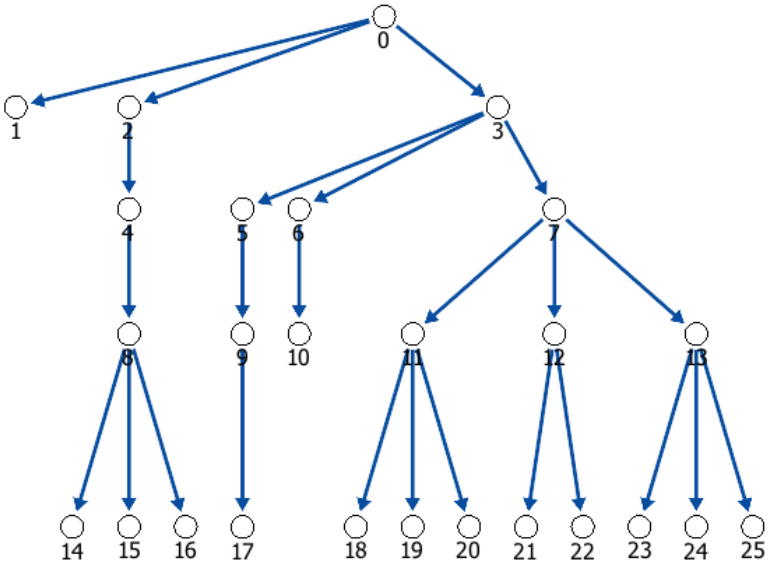


Figure 2.7: Layered layout

While Layered Hierarchical Layout is great for trees, the Force-Directed Layout is a great algorithm to generate layouts for graphs with no special properties or specific structure such as function call graphs. Thus our tool, Gview uses the Force-Directed Layout as the default layout algorithm for graphs for the algorithm produces visually pleasing layouts. The currently employed algorithm can be dynamically changed through our data transfer protocol. Gview is extensible and in the future, we plan to investigate more layout algorithms such as layout generation by Stress-Majoring.

2.3 Plotting

Since our goal is to develop an interactive visualisation, the third task plays a very important role. While the layout is calculated, the tool presents and maintains the latest specified view using the graphical description.

We aimed to preserve platform independence while also not sacrificing low-level access to hardware and thus the ability to gain control on the massively parallel architecture of modern GPUs. Thus we based Gview on the cross-platform application programming interface (API) OpenGL (Open Graphics Library) [19]. With OpenGL one is able to utilise the GPU to render the 2D meshes generated from the layout algorithm. It also features Compute Shaders that are Shader Stages that can be used for computing arbitrary information. In our implementation, we use Compute Shaders to support FDL parallelisation.

While OpenGL enables low-level control of the GPU, to handle user interaction such as a click of the mouse button, or keyboard shortcuts and to actually open a window in which the OGL rendering commands can take effect, we used the library Flib. Flib [20] is an open source GUI (Graphical User Interface) library built on top of OpenGL, written in C++ and hosted on Github. It is also multi-platform and uses the native window handling library on each supported system, for example, WINAPI on machines running Windows. The GUI functionality of Flib is backed by wrapper classes around OGL objects, such as the Array Buffer Objects (ABOs), to take advantage of OOP concepts like RAII (Resource Acquisition Is Initialisation) to ease the task of resource management and in the same time remain as efficient as possible.

Based on the resource handling classes Flib contains a sprite engine featuring automatic image packer called a texture atlas for packing images on a single OGL texture object to then be used by the engine. These sprites only contain small information, such as position, size, rotation and occupied rectangle on the texture atlas and thus by realizing the background of each GUI entity, such as buttons and sliders, using sprites, all of them can be drawn in a single draw command. Text is displayed in a highly similar manner: each font gets a personal texture atlas and characters are plotted via sprites.

The GUI is modelled as a tree in which each node is a GUI element. The events are passed in a top-down manner, thus every element receives and reacts to each event. To automatically convert basic events such as mouse movement and mouse wheel scrolling into more complex events like zooming or rotation, Flib uses listener classes one can inherit from to acquire desired callback functions.

Flib also has utility to tessellate line segments into thick lines built from triangles and generate distance to edge values which we used when employing the anti-aliasing technique DEAA [21] (Distance to Edge Anti Aliasing).

Since we aim at interactivity, we wanted to minimise the time spent on the actual drawing while maintaining good quality graphics. Upon each event such as mouse wheel movement, dragging with the mouse or when a new approximation of the final layout is created, the plotting data is used to generate drawing meshes (tessellated on the CPU) using Flib. For each visible node, we create a regular polygon with n sides, depending on the zooming level and the requested shape of the node. This dependence on the zooming level is called Dynamic Level of Detail. We use this technique to reduce the generated geometry by up to a factor of 100. We also tessellate each visible edge of the plot and send the resulting geometry in one batch to the GPU memory. This streaming process, thanks to the small amount of geometry, takes only a fraction of the update time.

Since the drawing is organized into single batches per triangles and lines, we can issue the drawing in only two draw calls which minimise drawing setup costs. The events mentioned above may seem to be frequent to the user. However, at 60 events per second peak with thousands of nodes, it is still an easily manageable task for an average modern CPU. We conducted measurements hundreds of frequently occurring views (about 20 to 30 nodes per view) and determined the amount of time needed to parse the input graph description from our binary protocol was at most 5ms which is not at all significant.

Updating the layout using parallel FDL with one iteration, however, although only taking around 1ms, needs to be calculated a large number of times. We also evaluated the time taken to tessellate the given view into raw drawing data, which as our predictions showed, do not take up much of the update time (around 0.5ms at most). Since we are running the layout generation on a separate thread (see Section 2.4) synchronising the data on the drawing thread and the worker thread takes up time too, but it is not significant.

The summary of our measurements is shown on Figure 2.8.

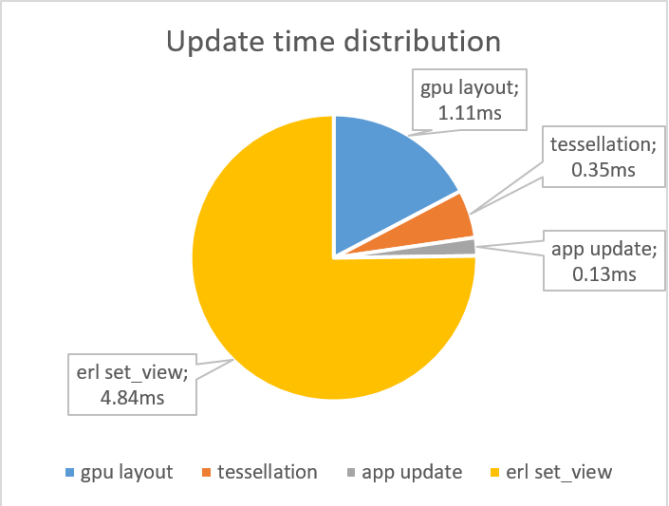


Figure 2.8: Relation of the time spent on transferring a view, updating one iteration on it, drawing it and synchronisation. Note that layout updates occur much more frequently than data transfer.

2.4 User interaction

User interaction happens through the Graphical User Interface of Gview, backed by Flib, built on top of OpenGL, via mouse clicks, keyboard buttons, etc. Hovering over a node or the label of the node highlights it and reveals smaller nodes around it, called selectors, indicating the selected node. Selectors only have a label and can be specified from the host application as described in Section 2.1. Different type of nodes may have different set of selectors, for example, a function node has an expand selector if it is the centre of the view for expanding the call depth shown and a lex selector to switch to the lexical nodes of the SPG spanning from the function node. Clicking a node triggers message sending. It sends the host application the `clicked` and `node` messages and an integer representing the id of the node. When a selector is clicked, beside the messages `clicked`, `selector` and the node id, the number of the selector is also sent. Certain key combinations, like CTRL+Z and CTRL+Y, also produce messages `undo` and `redo` accordingly. With these combinations, the user can go step by step back and forward in the history of previous steps. The host application can react to these events by loading a new view or ignore them at all without any problem.

Our previous approach [12] was a single threaded design. Thus, the layout generation, drawing and user input handling were done on the same thread. In some cases, when the layout generating algorithm took more time than usual, it delayed user interaction handling. To remedy this problem, we split up the process into two threads as shown on Figure 2.9: the first responsible for user interaction handling and drawing the actual

graph quickly from the last synchronised layout, the second responsible for layout generation. The two threads share a mutex used to protect the shared layout data which is updated by the worker thread after every iteration, or after the completion of the generation if the algorithm is non-iterative. This way the zooming and translating can remain interactive even with heavy layout calculations in exchange for a synchronisation overhead.

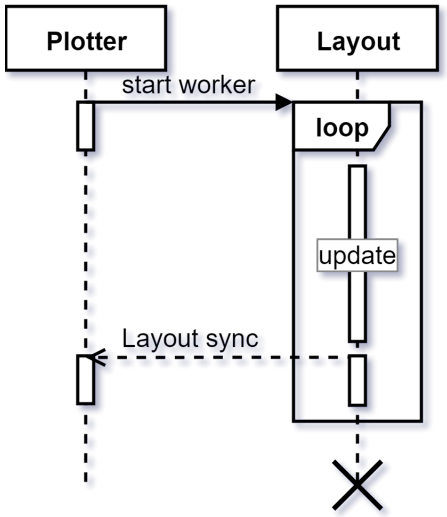


Figure 2.9: Interaction of the rendering thread which also handles the user interaction and the worker thread that generates the layout.

2.5 Integration with RefactorErl

RefactorErl already contains numerous refactorings and code comprehension supporting functionalities. To further enhance the code comprehension capabilities, we wanted to extend it with graph plotting capabilities to allow traversal of the Semantic Program Graph (SPG) interactively.

Our design of the graph displaying component of RefactorErl relies on the strength of the Erlang programming language: robustness. The SPG can grow to an enormous size (hundreds of thousands, or millions of nodes and edges), thus when one wants to display only the closely related entities of a subgraph in focus the dynamic capability of Gview is a good match.

The implementation uses Erlang ports for dynamic data transfer and command protocol between RefactorErl and Gview. The standard input and output of the opened application (Gview in our case) are turned into binary input and output channels. Through

this channel, the processes can communicate by sending and receiving messages. On Windows, this is not that straightforward as the newline characters are changed which require special handling. The exact mechanism of the Erlang Ports is implementation dependent, however, according to our estimations, even larger views of thousands of nodes can be sent quickly.

To confirm our estimation, we measured the data transfer rates on one hundred views of different sizes ranging from the trivially small to even two thousands of nodes and found that the loading times never exceeded one-third of a second, proving that the Erlang Ports are capable of delivering sufficient speed.

The philosophy of Erlang is said to be "Let It Crash", which means that fail safety on the grander level is achieved by creating smaller building blocks that can be let to crash and then restarted without the end user even noticing it.

This philosophy is reflected in the fact that the communication between the static analyser and the plotter is governed by an Erlang server, `gview_server`. The responsibility of `gview_server` is to accept incoming plot requests, layout changes and based on them create the necessary data to be sent to Gview and also receive and propagate events from Gview to the view switch handling code. For large graphs, as dataflow graphs, it may take some time for this server to collect the necessary information for plotting. The `gview_monitor` is a server process that monitors the `gview_server` and acts as a proxy between the user/program and Gview. Thus the caller of the Gview interface of RefactorErl does not have to wait to receive the data until the graph query finishes. Since Gview is an external program RefactorErl stays unaffected by an unexpected failure of Gview.

The module `gview` gives an interface that hides the monitor and the server and makes the usage of the tool much more intuitive. Interface functions such as `gview:start/0`, `gview:layout/2` or `gview:load/2` can be used to pass commands to Gview, for example `P = gview:start()` starts a new instance of Gview and assigns its identifier to variable P which then can be used to plot all the loaded modules and functions via `gview:load(P,modules)` or change the layout view `gview:layout(P,layered)` or query the status of the connection via `gview:status(P)`. An overview of the application structure can be seen on Figure 2.10.

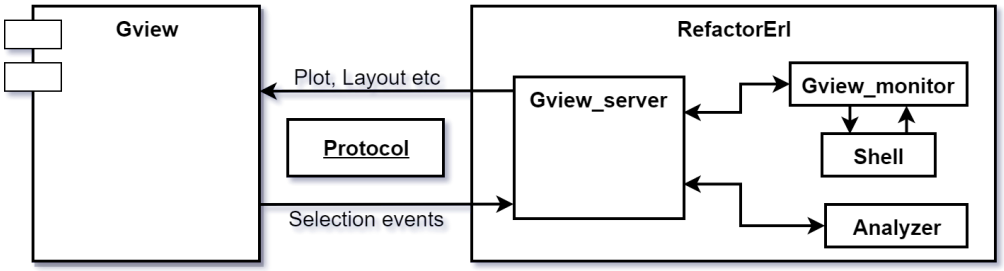


Figure 2.10: Our graph visualisation architecture, Gview.

Chapter 3

Optimising the Force Directed Layout generation

Filtered subgraphs of the *SPG* sometimes exhibit properties that make plotting using specialised algorithms much more desirable, such as the case with trees and the layered layout. Often times, however, the generated view will not bear such nice structure as in the case of general function call graphs which can have loops, multi-edges and even self-edges. The method of force-directed layout [22] generation is a way of generating a two-dimensional layout for general directed and undirected graph alike. In this chapter we introduce this well-known algorithm and present a new version of it that is optimized for modern GPUs.

3.1 The algorithm

The core concept of the algorithm involves fitting a physical system on the graph, in the following way. Take a graph $G = (V, E)$ with weighted vertices V and weighted edges E, totalling n vertices, weight functions m and l! Each node of the n nodes is corresponded with a body in the system with a position denoted by $p_i(t)$, a constant charge, m_i, and a velocity $v_i(t)$. As the position and velocity depend on the time passed since the start of the simulation, p_i and v_i have the type of $\mathbb{R} \to \mathbb{R}^2$. The generation of force-directed layout for graph G starts with calculating an initial, usually random, layout of the nodes, denoted by $p_i(0)$ for $i = 1..n$. After the initial layout has been set up, the algorithm follows by simulating the evolution of the physical system using the

following equations.

$$v_i(t) = \sum_{j=1, i \neq j}^{n} e(i,j,t)) \tag{3.1}$$

$$e(i,j,t) = \frac{d(i,j,t)}{\|d(i,j,t)\|_2} * \left(H * ln(\|d(i,j,t)\|_2^2) * l_{i,j} - \frac{G * m_i * m_j)}{\|d(i,j,t)\|_2^2} \right) \tag{3.2}$$

$$d(i,j,t) = p(t)_i - p(t)_j \tag{3.3}$$

$$v = \frac{\partial p}{\partial t} \tag{3.4}$$

Equation 3.1 means that at any given time point t the velocity of the ith body equals to the sum of the forces exerted by other bodies. Here we use the term force, to describe instantaneous forces, which have a direct effect on the velocity of the bodies rather than the acceleration. Such forces are characterised by Equation 3.2: knowing the position of the ith and jth body at time t, we can easily calculate the force acting on body i at time t.

Here H and G are arbitrary positive constants, used to regulate the strength of the two types of acting force. In our research having $H = 2$ and $G = 6100$ turned up the best looking results. The most important equation is 3.4, which describes the analytic connection between position and velocity in the physical system. It can be used to rewrite the former equation system as a differential equation with the common vector function of positions $p = t \rightarrow (p(t)_1, p(t)_2, ..., p(t)_n)$ as the unknown, as follows.

$$\frac{\partial p(t_{cur})}{\partial t} = f(t_{cur}, p(t_{cur})) \tag{3.5}$$

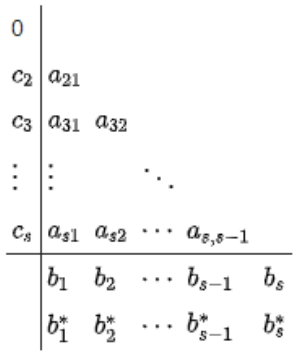
$$p(t_0) = p(0) = p_0 \tag{3.6}$$

$$f(t,p) = \sum_{j=1, i \neq j}^{n} e(i,j,t) \tag{3.7}$$

The canonical form of ordinary differential equation is given in Equation 3.5 with the definition of f in Equation 3.7, while the initial position requirement is defined in Equation 3.6.

Therefore, our goal is to approximate the vector function p for increasing values of t until we get close enough to its fixed point. For this purpose, we want to use higher order methods, to better utilise computational power and stable convergence as t approaches ∞.

Figure 3.1: Example of an extended explicit Butcher tableau of degree s



3.2 Using higher order methods

The Runge–Kutta methods [23] are a family of explicit iterative methods, used in temporal discretization for the approximate solutions of ordinary differential equations. The methods include the well-known routine called the Euler Method and can be considered as the generalisation of the routine. The method has an adaptive version, which can adjust the step-size of the simulation and thus keep the error below a given value, ϵ.

These methods are called a family for the reason that they depend on numerous parameters: a, b, and c. The latter two are vectors and the former one is a matrix [24]. These parameters can be arranged in a so-called Butcher tableau as can be seen in Figure 3.1.

With the initial positions given in p_0, the method proceeds to create further approximations, $p(t_{i+1})$, from the previous one, $p(t_i)$ for $i = 1..\infty$, according to Equations 3.8 and 3.9. The difference in the result of these two equations is used to approximate the error introduced by taking a step of length h. Using this calculated error term, we can choose a new step-size in order to decrease the error below ϵ, or allow larger steps in exchange for larger error term.

$$p_{i+1} = p_i + \sum_{j=1}^{s} b_j k_j \tag{3.8}$$

$$p^*_{i+1} = p_i + \sum_{j=1}^{s} b^*_j k_j \tag{3.9}$$

where

$$k_j = f(t_i + c_j * h, p_i + h * \sum_{l=1}^{j-1} a_{j,l} k_l) \tag{3.10}$$

The exact steps of choosing the next h and the very mathematics behind this algorithm is a well-known topic and has been discussed in many papers. Our goal is to apply this method to the problem at hand and to optimise it for the parallel architecture of modern GPUs.

As we can see in Equation 3.7, our f does not depend on the p parameter directly. The indirect dependence through d is actually replaced by the previous best approximation p_i for k_1 and $p_i + h * c_{j+1} * k_j$ for k_{j+1}, thus eliminating the need for the matrix a of the RK method. This property of the simulation resulted from the fact that we do not employ friction, which would depend on the velocity of the bodies but rather use instantaneous forces.

3.3 Methodology

Our goal is to find ways to improve the performance of the force-directed layout generation through utilising the massively parallel architecture of modern GPUs.

The final form of the equation that we are using, taking into consideration the relevant properties of the physical system, described at the end of Section 3.2, is Equation 3.11. In each step of the simulation, we have to calculate the k coefficients for each of the n bodies. Since the dimension of k is s and evaluating f requires $\mathcal{O}(n)$ operations, the total cost of advancing one step comes out to be $\mathcal{O}(sn^2)$.

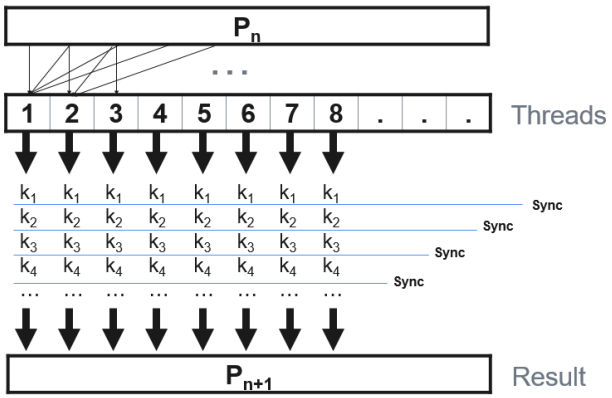
$$k_j = f(t_i + c_j * h) = \sum_{b=1, a \neq b}^{n} e(a, b, t) \tag{3.11}$$

3.3.1 Linear parallelisation

The first idea for parallelisation that one should consider is simply assigning the task of evaluating the exerted forces of all other bodies to a single body. Thread i allocates a single two-dimensional vector v, loops through all the bodies in the system and calculates the force exerted on body i through body j at the current time point t_n[1]. The calculated

[1] with exception of the i^{th} body

Figure 3.2: Architecture of basic parallelisation technique



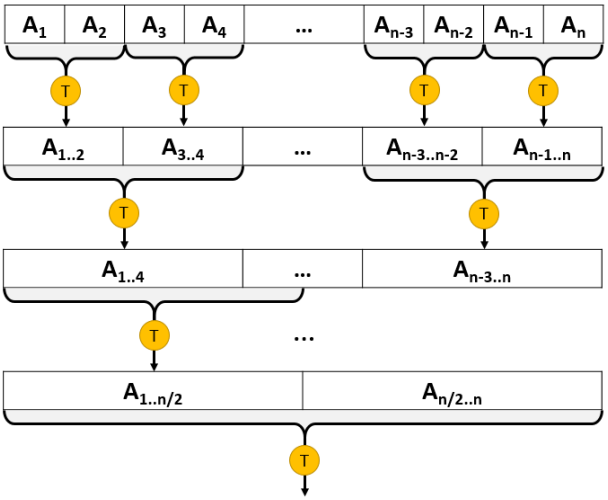
forces are accumulated on the fly into the local variable v of the thread and the result is stored in the k_1 array. The k_{j+1} approximations are generated in a similar manner, however p_i is replaced by $p_i + h * c_{j+1} * k_j$. A schematic representation of the linear parallelisation method can be seen in Figure 3.2.

The above mention dependence of k_{j+1} on k_j results in the need of a global synchronisation of all the employed threads in order to make the calculated k_j values visible to the other threads. This explicit synchronisation can only be realised on the GPU if the number of invocations is below certain driver defined limits, which can be queried using OpenGL command and is usually at least 1024.

In case of having a larger amount of bodies in the system than the hardware exposed limit, we need CPU synchronisation, which basically means splitting the calculation of each k_j vector into different dispatches.

The performance bottleneck, however, comes from the fact that the amount of work each thread is doing is proportional to sn which can grow too large. The OpenGL standard guarantees [19] the ability to dispatch at least a maximum of 2^{16} workgroups, all of which may consist of a maximum of at least 1024 work items (threads). Which means that by reducing the number of threads dispatched from n can potentially result in a great increase in performance. This idea is further supported by the fact that GPU cores are much less powerful than CPU cores and thus employing more of them can lead to better resource utilisation, which gives reason to the optimisation in Section 3.3.2.

Figure 3.3: Basic idea of divide and conquer strategy used in the parallel reduction. The circles with T represent threads of execution.



3.3.2 Refining work per thread

To better utilise the parallel architecture of modern GPUs for the problem at hand, we want to dispatch more than $\mathcal{O}(n)$ threads. Thus we take Equation 3.11, and for each (a, b) pair, we create an invocation, totalling in $n(n-1)$ threads. After calculating each $e(a, b, t)$ value in the summation on Equation 3.11, however, we need to evaluate the actual sum of these two-dimensional vectors and this is where parallel reduction comes in. Reduction (or folding) is the generalisation of summation: for a given A array of size n and a binary combining function f, the result of the folding expression is $b = f(A_1, f(A_2, f(A_3, ...)))$ where the ... goes until n. Parallelising a reduction is not a trivial task and is a well-known candidate for optimisation [25]. In our example, we have the benefit that our combination function is associative and commutative. Thus enabling us to employ a divide and conquer technique as shown in Figure 3.3.

In the presented approach, we divide the reduction into levels of reduction, in each level, the size of the array that is to be combined is decreased by half. In each level, one thread only has to combine two elements of the array of the current level, which would imply that the work per thread has changed to be $\mathcal{O}(1)$. However by noting that the levels must come in increasing order, each one depending on the previous one, after reusing the allocated threads through levels, the total work per thread totals $\mathcal{O}(log_2(n))$.

Figure 3.3 shows an optimal scenario, with n being a power of two, if n is not a power of the amount of work one thread is responsible for, then the last thread my index out of the array. To avoid this, we need to employ bound checking. An example for the size of 5 can be seen on Figure 3.4. This bound checking may induce a maximum of $\mathcal{O}(n)$

Figure 3.4: Example of maximum amount of extra work introduced by not a power of two n, for $n = 5$.

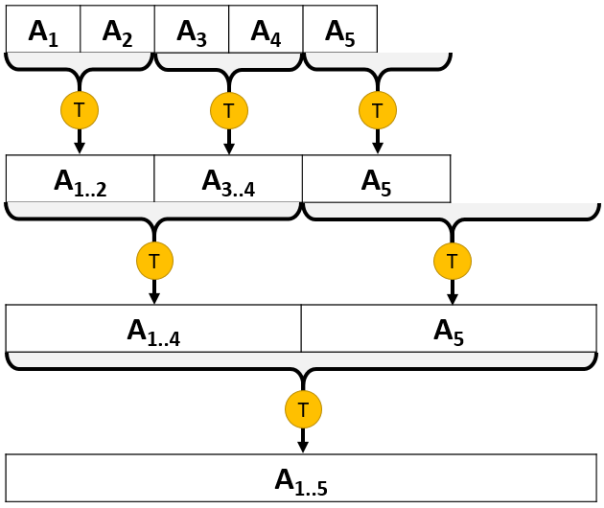
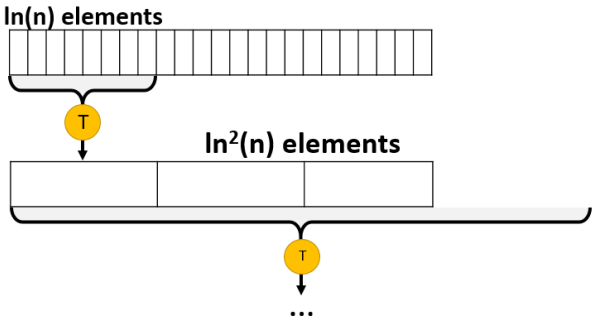


Figure 3.5: Optimal $ln(n)$ work per thread.



extra work.

The theoretical optimum of giving one thread $\mathcal{O}(ln(n))$ work can also be achieved. However, it requires extra mathematics behind the indexing and bound checking, and also the potential extra work growth to $\mathcal{O}(nln(n))$. A visual representation is show on Figure 3.5.

3.3.3 Memory usage and synchronization

When programming for a modern GPU, it is possible to arrange threads of execution (say invocations) into so-called workgroups. Threads (also referred to as work items in this

Figure 3.6: In-place memory usage of the parallel reduction.

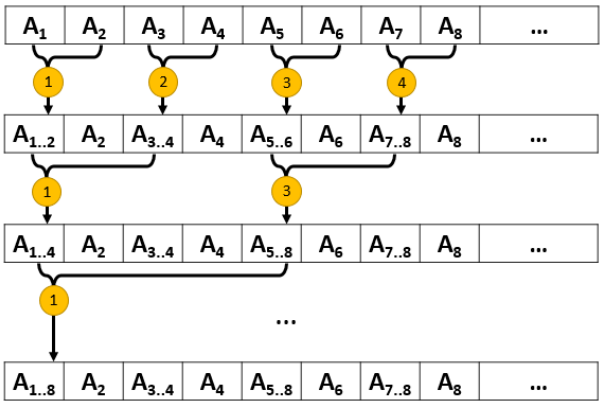
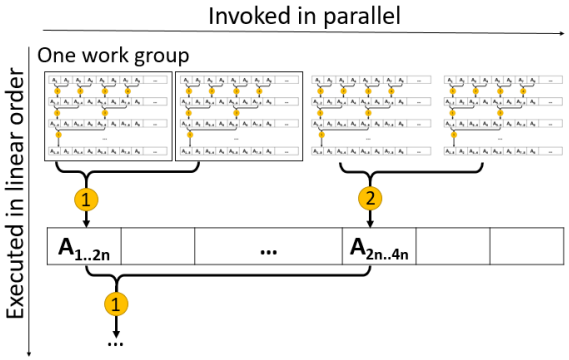


Figure 3.7: Dividing the parallel reduction into smaller tasks that can be handled by one work group.

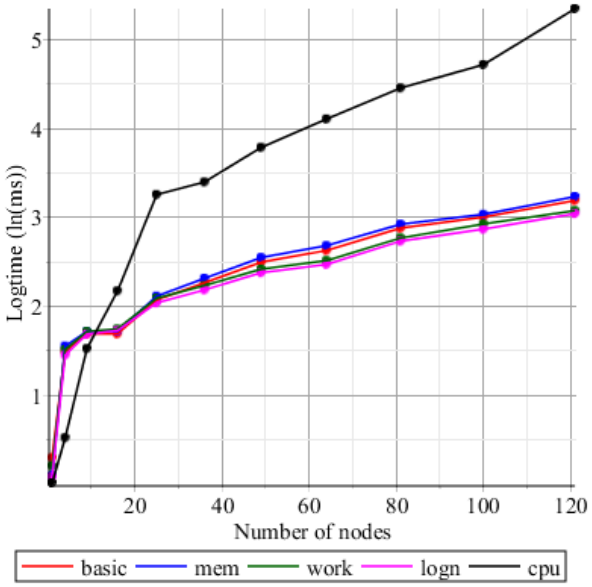


context) in a workgroup can share group local memory and can synchronise group-wise without the need for CPU intervention.

Because the levels of reduction are calculated linearly, we can store the partial results in-place, which thanks workgroups, can be synchronised on the GPU. This in-place storing of partial results can be seen in Figure 3.6. Thanks to this technique, we only need $\mathcal{O}(n)$ memory and only need to transfer one element to the CPU each frame.

However, to be able to utilise the local synchronisation of the GPU workgroups, a work-group size equal to the number of bodies is needed, which brings us back to our previous problem. To solve this, we investigated how we could organize the parallel reduction into batches of maximal size, and recursively apply the already presented reduction method. As Figure 3.7 shows, by creating partial results of maximal size, using multiple work groups and then applying a global synchronisation, we can take advantage of the parallel architecture.

Figure 3.8: The huge difference of CPU and GPU algorithm, note the logarithmic scale! Tested on grid graphs.



3.4 Evaluation

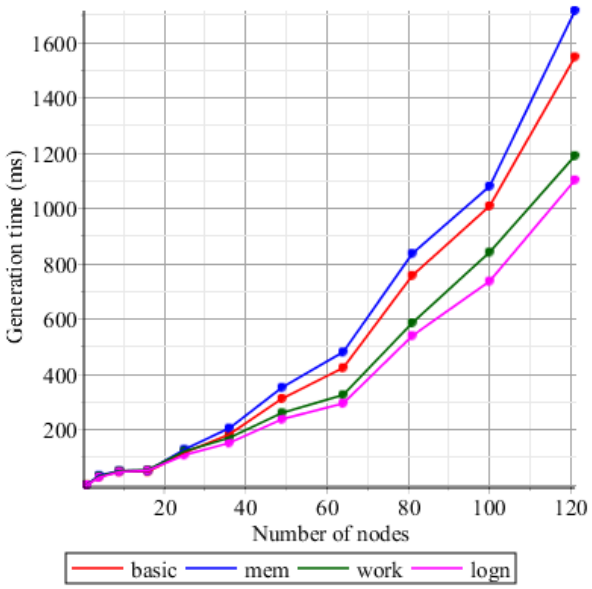
The researched techniques were tested on a laptop with 5^{th} generation Intel Core i5 processor, 8GB of memory, using Intel HD 6000, running OpenGL 4.5. Parallel GPU programs were realised with GLSL version 430 [26].

In our tests, we incrementally generated square grids of increasing sizes from 1x1 to 11x11, with random starting positions and run first the CPU, then the GPU and refined GPU algorithms on the same starting points, around 100 times each. The final layout of an 8x8 grid is shown on Figure 2.6. As these measurements may vary according to environment properties such as the OS scheduler or extra load from updates and scheduled cleaning etc, we ignore the highest and lowest 5% of data and perform a normal distribution fitting on the rest. The resulting expected value of generation time is plotted against the number of nodes in Figure 3.8 and Figure 3.9.

Figure 3.8 clearly shows the enormous improvement of the GPU parallel algorithm, even on the integrated card used in testing. Our expectation is that with a higher tier dedicated card this gap is to increase further.

The comparison presented in Figure 3.9 shows how different techniques described in Section 3.3 improves generation time for increasing sizes of grids. The interesting thing

Figure 3.9: Comparison of the refined GPU algorithms, tested on different sized grid graphs.



to note is that the memory (but not workload) optimised method performs poorer than the trivial approach. This can be due to the hardware memory locality of Intel integrated GPUs, which implies that the extra copy operations introduced by memory optimisation by hand have a higher toll on performance than of the improvements in the locality it creates. Thus on a dedicated card, the memory optimised version might improve performance considerably.

We also investigated the performance of different realisations of the Runge-Kutta family: the Heun-Euler method, the lesser famous Bogacki-Shampine method and a high order Fehlberg method. The tests were performed on a tree graph with nodes ranging from 3 to 133 and the same refining techniques were applied as mentioned previously. The results of this comparison can be seen in Figure 3.10. One can clearly see how the advantage of being able to take larger steps turns into a disadvantage of higher required work per step. Based on these measurements, we can conclude that the Bogacki-Shampine method is most efficient for our problem.

Figure 3.10: Comparison different RK methods, tested on varying sized tree graphs.

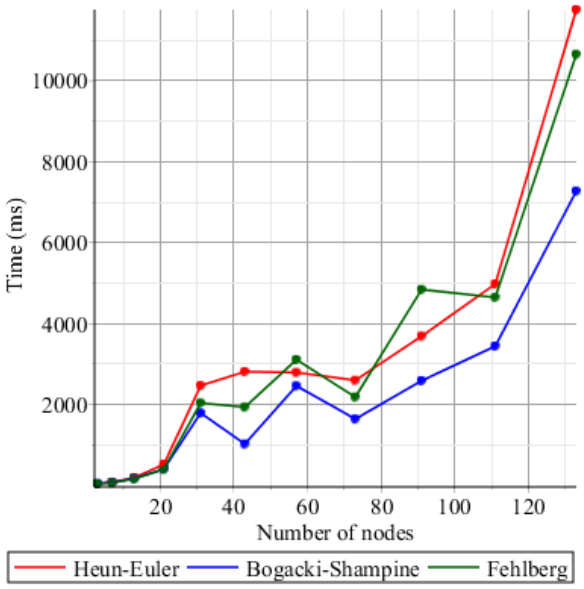
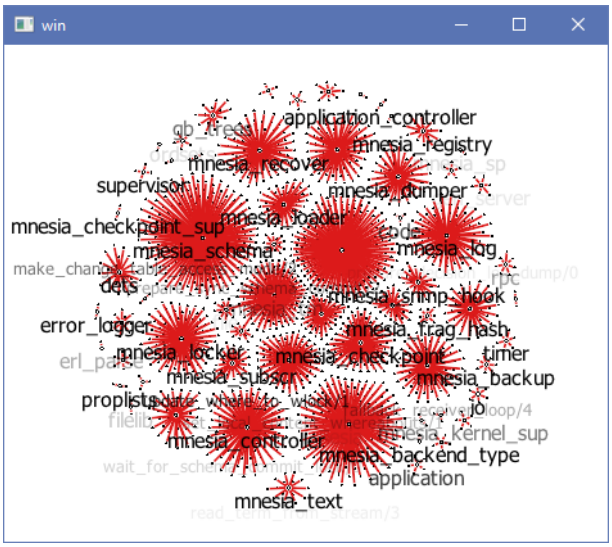


Figure 3.11: View of all the modules in the Mnesia DBM.



Chapter 4

Using Gview for code comprehension

In this section, we show two use cases to demonstrate how Gview supports code comprehension. In our use cases, we analysed the Mnesia DBMS system [27] and built the Semantic program Graph from it. The first use case generates a function view of the SPG, while the second uses the syntactic view to explore the graph searching for possible values of different language constructs.

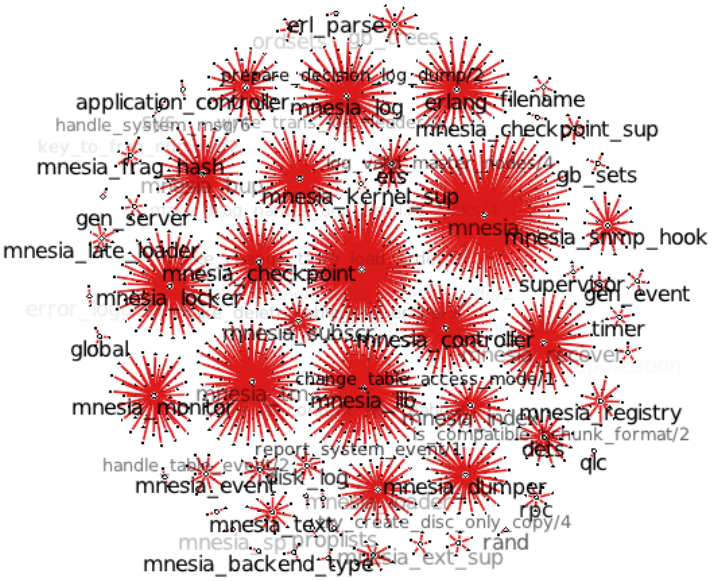


Figure 4.1: Mnesia view

Figure 4.1 illustrates the force-directed layout based representation of the Mnesia modules and functions. Mnesia has 26 KLOC, which is a medium scale Erlang application. It contains 1804 function definitions in 30 different modules. However, the presented view contains the referred functions and modules as well. Therefore in total 63 modules and 2103 functions are visualised.

After starting RefactorErl's interactive Erlang shell interface, we can easily start Gview and load the module view with the following commands:

```
P = gview:start().
gview:load(P, modules).
```

Our interactive tool makes it possible to select a node in the graph and generate a new view. Figure 4.2 shows the graph created when we clicked on module `mnesia_log`.

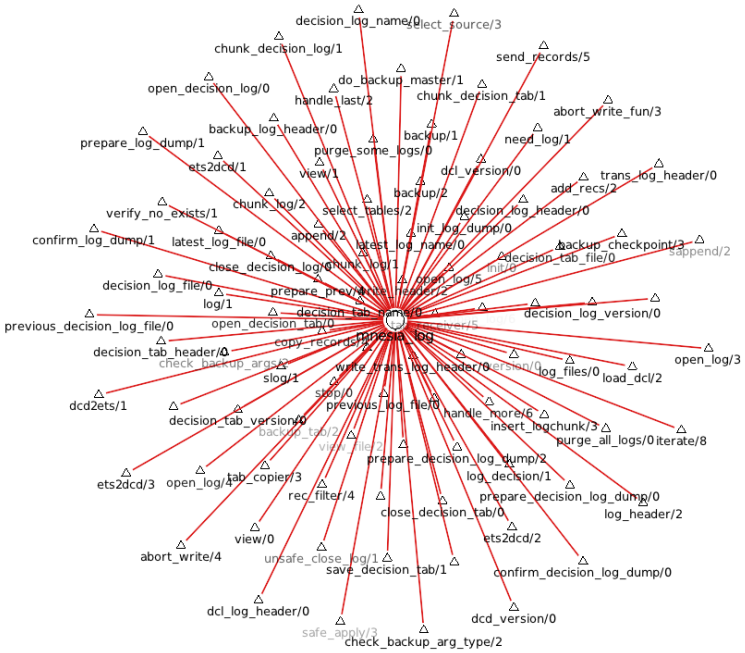


Figure 4.2: mnesia_log view

The same view is available through a direct call in the Erlang shell as well:

```
gview:load(P, [{modules, [mnesia_log]}]).
```

The size of these graphs for even this medium scale application is too big but makes a good starting point for further analysis. The interactive nature of the tool makes it

possible to easily dig deeper into the modules/functions/structures to find the required information.

4.1 Call graph view

Clicking on the `open_log/3` function on Figure 4.2 results in the graph showed on Figure 4.3. Coloured edges differentiate the functions calling `open_log/3` and those that are called by `open_log/3`.

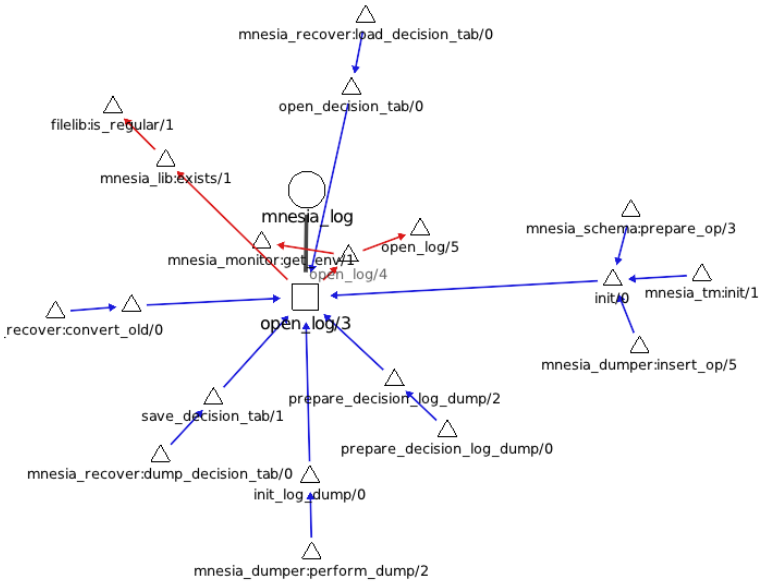


Figure 4.3: `open_log/3` function call graph

Any element of the graph can be clicked to expand the next level of the call graph. For example, selecting the `open_log/5` label results in the graph showed on Figure 4.4.

The user can easily switch between the force-directed and layered layout and choose the most appropriate for the task. For example, the following command results in the view presented in Figure 4.5:

```
gview:layout(layered).
```

Using call graphs in software maintenance is useful, it can contribute to bug detection and fixing and code comprehension tasks as well. An interactive tool, such as the integrated Gview to RefactorErl, provides a handful tool to help the developer to draw and explore

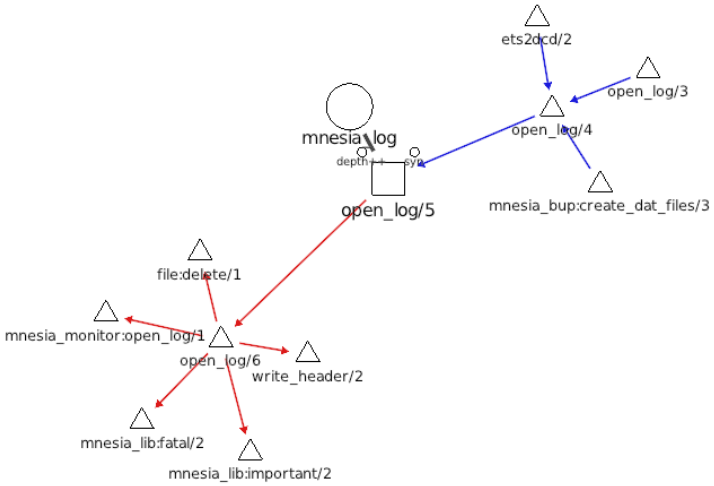


Figure 4.4: `open_log/5` function call graph

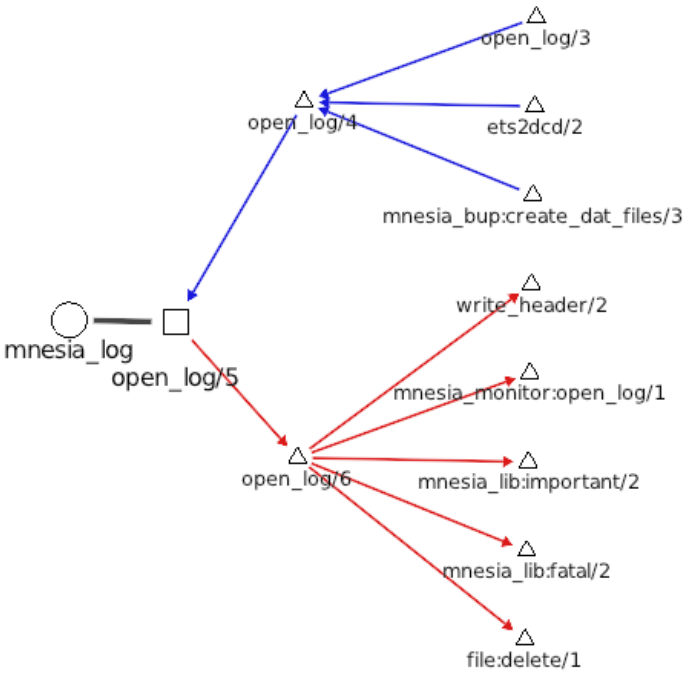


Figure 4.5: `open_log/5` function call graph (layered view)

the call graph. The semantic information available in RefactorErl extends the classical static call graph views with dynamic call information as well [28].

4.2 Enhanced syntax view

Besides the high-level function view, a developer might be interested in the details of the implementation as well. At this point, Gview provides a syntactic view of the functions and expressions as well. The syntax view of `open_log/6` is illustrated on Figure 4.6.

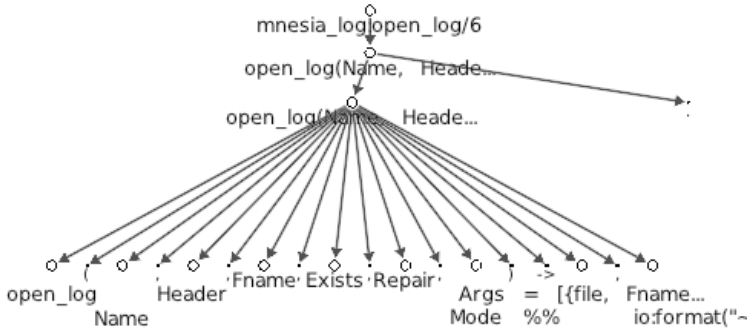


Figure 4.6: `open_log/6` high level syntax view

The syntax view can be further expanded with new levels of details from the syntax tree until the developer finds the appropriate information (Figure 4.7).

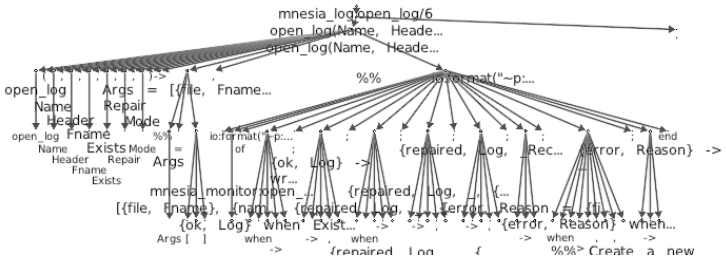


Figure 4.7: `open_log/6` high level syntax view

Besides the syntax tree, the interface makes it possible to reach semantic information from the Semantic Program Graph of RefactorErl. For example, a view can be generated from the dataflow information [29, 30]. Once a developer is interested in the possible origins of a variable, the values that may flow to the variables are also shown in the graph. For example, clicking on the variable `Mode` on the syntax view (Figure 4.6) we can deduce from the generated view (Figure 4.8) that the possible values are `read_write` and `read_only`.

However, the desired information is not always available with a single step. Clicking on `Fname` on Figure 4.6 does not give us the possible values since it is calculated by a

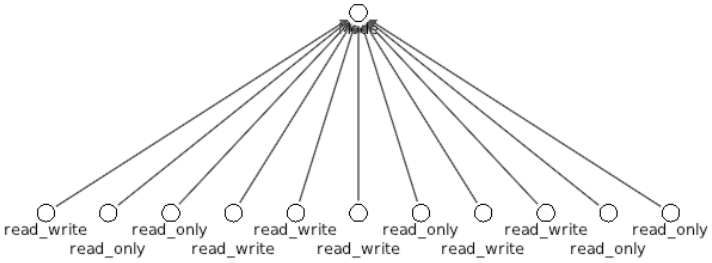


Figure 4.8: Values of the variable `Mode`

function application (Figure 4.9).

Figure 4.9: Values of the variable `Fname`

Thus, we have to further investigate the structure of the application (Figure 4.10) and select the `Fname` node in the tree of the application to find all the possible file names used (Figure 4.11).

Exploring the syntax tree and using dataflow relations are extremely useful in bug fixing task. Once the developer detects a wrong value at a certain point of execution, the enhanced syntax tree explorer can help to find out where the wrong value comes from.

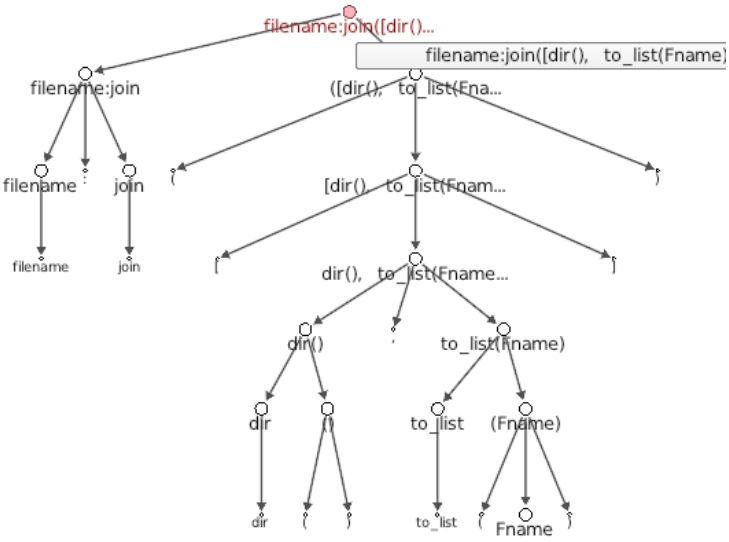


Figure 4.10: Exploring the values of the variable `Fname`

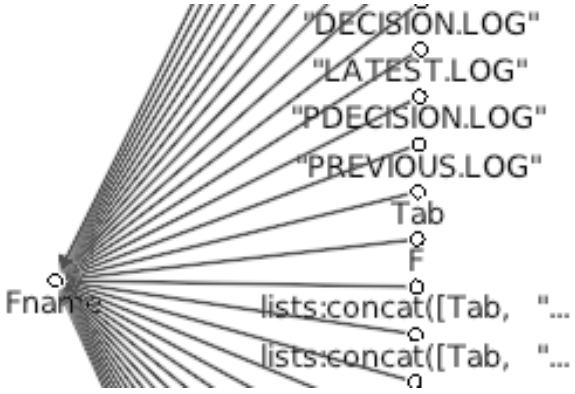


Figure 4.11: Exploring the values of the variable `Fname`

Chapter 5

Related work

Following are some tools suitable for software visualisation that we would like to compare to our solution.

5.1 IslandViz

With the tool IslandViz [31], one can explore modular software systems in virtual reality. The metaphor "island" is used for modules, each module representing a distinct island. The system is displayed as a virtual table, where users can explore the software by performing navigational tasks on multiple levels of granularity. The tool enables the users to get an overview of the complexity of the software and to traverse the system interactively and explore the modules and the dependencies between them.

The project is built on Unity, a cross-platform real-time game engine developed by Unity Technologies. It supports many target platforms from Android and iOS phones to web applications and even graphics-heavy 3D shooter games. The engine offers a primary scripting API in C# which is used by IslandViz to implement the layout algorithms in the program. IslandViz offers two layout generation algorithms, one pseudo-random based and a variant of the Force-Directed Layout. The random based algorithm incrementally puts nodes (islands) on random positions, retrying up to fixed times if the placed island collides with another. The problem with this approach is that it disregards the actual structure of the graph. The other available option in the software is an FDL which differs from our variant. The IslandViz uses regular physical springs as opposed to logarithmic springs and employs friction instead of instantaneous forces. However, the real difference lies in the terminating condition and the underlying mathematical tool. The computation in IslandViz stops after a certain number of steps. Our solution is

able to keep a moving average of the maximum relative error thanks to the embedded higher order methods and terminates only if this error is below a certain threshold. Our redesign of the Force-Directed Layout algorithm [15] also targets the massively parallel architecture of modern GPUs which results in a massive performance increase over the CPU implementation.

5.2 Graphviz

Graphviz [32] is a well-known graph visualisation software package developed by AT&T Labs Research since 1991 and is open-source. Among the supported layout generation algorithms, there are energy minimizing, stress-majoring methods and hierarchical ones, each of which has an own program in the Graphviz package. These programs transform the input graph, described in a text-based language, and apply the given technique to produce an image file that can be then embedded in other applications and web pages. Many details can be adjusted using these tools, such as the colours, fonts, tooltips, line styles and the shape of nodes. Among the several applications that use Graphviz, there are many UML drawing tools, computer-aided design systems such as FreeCAD.

Numerous software are able to produce graph descriptions in the native language of Graphviz, the DOT graph description language. Since the RefactorErl is able to export the entire SPG to DOT, we tried the approach of parsing the exported DOT file in the Gview plotter. This version resulted in slower startups.

There were former attempts at using Graphviz as a graph plotter for RefactorErl. The resulted graphs were static, and it was too complicated to switch between different views.

In general, Graphviz targets and excels at the static plotting of graphs that did not meet our requirement.

5.3 CodeCity

CodeCity [33] is a software visualisation tool that brings software systems to the screen with a city metaphor. In the interactively discoverable 3D city classes show up as buildings that stretch higher depending on the complexity of the given class. The buildings that represent classes are grouped into neighborhoods based on their packages. The program uses OpenGL to render the scene and was built using VisualWorks Smalltalk. The goal of the project is to aid users in discovering software artefacts such as god classes that appear as large skyscrapers on the landscape.

CodeCity targets the visualisation of different code metrics and thus is not suited for plotting other relations such as dataflow or function dependency graphs.

5.4 CityVR

CityVR [34] is an interactive software visualisation tool that uses virtual reality to achieve the gamification of software engineering. Similarly how headphones help to filter noise, the used I3D medium enables users to isolate from the outer world in a visual way and thus encourages immersion.

CityVR was built using Unity3D 5.5 and uses CodeCity to create the displayed model which is generated from source code files. Since monitors are the most frequently used tools for software visualisation, a more exciting and thought provoking way of exploring software dependencies and metrics is using a virtual reality headset. This way users of such visualisation tools can achieve better recollection [35].

5.5 Sourcetrail

Sourcetrail [36] is a lightweight source explorer that supports many mainstream IDEs and code editors. It features interactive code and dependence exploration and fast searching functionality. The aim of the program is to help programmers in quickly finding relevant pieces of code in large projects without digging through the code base but rather through interactive diagrams of variables, functions and classes. They argue that by exploring the code through this visual representation, finding important information is much easier and convenient. By accompanying diagrams with relevant code snippets, Sourcetrail shortens the investigation time.

Sourcetrail is distributed in a commercial license for developers and companies and a non-commercial license, but only for personal usage. It is a standalone program and integrates with IDEs and editors such as VSCode through free downloadable extensions.

One drawback of Sourcetrail is that it only supports projects written in C/C++ or Java and thus is not applicable in case of Erlang projects.

5.6 Understand

Understand [37] is a multi-platform Integrated Development Environment (IDE) developed by SciTools for maintaining, measuring and visualising code bases. It features many code metrics from the basics like class or file count to custom metrics such as knots, path count and weighted methods per class. Understand employs an incrementally built indexing that enables users to search quickly in even millions of lines of code. It supports control flow graphs, hierarchy graphs, dependency graphs and many more but no dataflow graphs. Various coding standards can be checked by the tool such as naming guidelines and general best practices. The tool can generate overviews, like quality or metrics reports.

Using Understand naturally comes with the limitation that it has to be the IDE of choice for a project unlike in the case of Sourcetrail. Despite being able to handle more than a dozen languages and projects that use more than one language, Understand does not support Erlang, which is an important aspect of our scope.

5.7 d3js

D3.js [38] is a JavaScript library for manipulating documents based on data. D3 helps bring data to life using HTML, SVG, and CSS. It emphasis on web standards giving the full capabilities of modern browsers without the need of tying to a proprietary framework, combining powerful visualisation components and a data-driven approach to DOM manipulation. This library is a modern, browser-based solution to visualisation problems with countless useful features such as pie charts, hierarchical graph drawing and force-directed layout generation.

D3js supports force-directed layout generation using velocity Verlet integration which may require a much smaller step size than the RK methods in order to minimize oscillations in the solution, but the method is symplectic. Thus the two methods were meant to solve different kinds of problems, as our version of the force-directed layout generation uses logarithmic springs and instantaneous forces, our simulation need not be energy conserving or symplectic for short. The key difference between our research and d3js is that we aim to exploit the parallel architecture of modern GPUs, while the simulations of d3js get calculated on the CPU[1].

[1] unless some JavaScript optimisation happens

5.8 Gephi

Gephi [39] is an open source software for graph and network analysis. It uses a 3D render engine to display large networks in real-time and to speed up the exploration. Gephi advertises itself as having a flexible multitasking architecture that brings new possibilities to work with complex data sets and produce informative graphics. It has been used in a number of research projects in academia, journalism and elsewhere. For instance, it was used in visualising the global connectivity of New York Times content and examining Twitter network traffic during social unrest along with more traditional network analysis topics.

Development of Gephi was started in the summer of 2008, while the last stable update was in 2017, nearly one and a half year ago. It was created in the Java programming language and although it features an OpenGL renderer, it uses immediate mode rendering, that became obsolete with OpenGL 3.1 in 2009 which means Gephi does not use GPU for layout generation. Today, with OpenGL 4.6, much faster rendering tools are available, such as instanced rendering, VBOs and compute shaders. Also, it is built on top of the NetBeans IDE, which means it cannot be integrated into another project, only added as an external tool.

5.9 GoJS

GoJS [40] is a JavaScript and TypeScript library for building interactive diagrams and graphs. GoJS claims to let the user build all kinds of diagrams and graphs, ranging from simple flowcharts and org charts to highly-specific industrial diagrams, SCADA and BPMN diagrams, medical diagrams like genograms, and more. The library is meant for implementation of interactive diagrams and visualization on modern web browsers and platforms. It allows easy construction of custom and complex diagrams of nodes, links, and groups with customizable templates and layouts. As a matter of fact, it does not depend on any JavaScript libraries or frameworks, so it should work with any web framework or with no framework at all. The library focuses on interactivity and flexibility. There are many demos available online on the webpage of the tool. It also offers rich features like drag-and-drop, copy-and-paste, in-place text editing, tool-tips, templates, data binding and models, transactional state and undo management, palettes, event handlers, commands, and an extensible tool system for real-time custom operations on the diagram. It also features many automatic layout generation algorithms, which can be extended by the user of the library.

One of such automatic layouts is the force-directed layout generation algorithm. They describe the method as a layout generation method that treats the graph as if it were a

system of physical bodies with repulsive electrical, attracting gravitational, and spring forces acting on them and between them. The engine uses the CPU for layout generation, thus it is not optimized for modern parallel GPUs.

Chapter 6

Conclusion and future work

Providing tools to support code comprehension and visualise software are highly desirable in industrial-scale development. In this paper, we demonstrated our tool, Gview, that was designed to provide an interactive, dynamic, tool independent visualisation framework built on top of Flib. Gview also utilises the GPU to provide efficient layout calculation.

We presented the internal design and structure of the tool. We demonstrated the data transfer protocol defined as a communication channel to Gview and also the interaction handlers. The paper describes the graph layout calculation algorithms and plotting. The current version of the tool supports force-directed and hierarchical layout generation. In the future, we plan to extend the options.

We also presented a better approach to simulating the evolution of the physical system that is founding idea of the Force Directed Layout generation. We presented multiple approaches for parallelising the *FDL* on the GPU and analyse performance of the technique. We further investigated and measured optimisation opportunities for the GPU algorithm. These optimisations included different ways of refining the memory usage, tweaking the distribution of work among threads to reach a better configuration and the importance of different synchronisation functionalities.

We presented the integration of Gview with RefactorErl to demonstrate the applicability of Gview in large-scale software visualisation. The usefulness of the tool was shown on code comprehension use cases through call chain detection and expression value investigation.

Gview is open source and available on GitHub:

https://github.com/Frontier789/Gview

The integration with RefactorErl will be released soon with the upcoming release of the tool.

The next step in the evaluation of Gview is to implement the data transfer protocol for the Testing at Scale project [41, 42].

Bibliography

[1] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

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

[3] 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, Jul 2009.

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

[5] RefactorErl. Static source code analyser and refactoring tool for Erlang. https://plc.inf.elte.hu/erlang, 2019.

[6] Melinda Tóth, István Bozó, and Tamás Kozsik. Pattern candidate discovery and parallelization techniques. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, IFL 2017, pages 1:1–1:26, New York, NY, USA, 2017. ACM.

[7] István Bozó, Viktoria Fördős, Dániel Horpácsi, Zoltán Horváth, Tamás Kozsik, Judit Kőszegi, and Melinda Tóth. Refactorings to enable parallelization. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming*, Lecture Notes in Computer Science, pages 104–121. Springer International Publishing, Berlin, Heidelberg, 2015.

[8] Tamás Kozsik, Melinda Tóth, and István Bozó. Free the conqueror! refactoring divide-and-conquer functions. *Future Generation Computer Systems*, 79(Part 2):687 – 699, 2018.

[9] Melinda Tóth, István Bozó, Judit Kőszegi, and Zoltán Horváth. Static Analysis Based Support for Program Comprehension in Erlang. *Acta Electrotechnica et Informatica*, 11(3):3–10, 2011.

[10] SciTools. Understand. https://scitools.com/, 2019.

[11] Melinda Tóth. Using the open-source RefactorErl in Ericsson. Talk at Functional Programming Meetup, Craft Edition, https://prezi.com/ae0dthd87slb/?utm_campaign=share&utm_medium=copy, 2015.

[12] M. Komaromi, I. Bozo, and M. Toth. An Efficient Graph Visualisation Framework for RefactorErl. *Studia Universitatis Babeș-Bolyai Informatica*, 63(2):21–36, 2018.

[13] Eleftherios Koutsofios, Stephen North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.

[14] Mátyás Komáromi. Gview: Efficient graph visualisation for RefactorEr. Scientific Students' Associations Conference, ELTE, Budapest, Hungary, Received 1st prize, 2018.

[15] Mátyás Komáromi, Melinda Tóth, István Bozó. Optimising the Force-Directed Layout Generation, Paper submitted to the Acta Cybernetica scientific journal published by the Institute of Informatics, University of Szeged, Szeged, Hungary, April 8, 2019.

[16] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1988.

[17] Matthew Suderman. *Layered Graph Drawing*. PhD thesis, McGill University, Montreal, Que., Canada, Canada, 2005. AAINR21700.

[18] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. In János Pach, editor, *Graph Drawing*, pages 155–166, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[19] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[20] Mátyás Komáromi. Flib project github page. https://github.com/Frontier789/Flib/.

[21] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, and Tiago Sousa. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*, 2011.

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

[23] C. Harper. *Introduction to mathematical physics.* Prentice-Hall physics series. Prentice-Hall, 1976.

[24] J.R. Dormand and P.J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[25] Mark Harris et al. Optimizing parallel reduction in cuda.

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

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

[28] Dániel Horpácsi and Judit Kőszegi. Static analysis of function calls in Erlang. Refining the static function call graph with dynamic call information by using data-flow analysis. *e-Informatica Software Engineering Journal*, 7(1), 2013.

[29] 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.

[30] M. Tóth, I. Bozó, Z. Horváth, and M. Tejfel. 1st order flow analysis for Erlang. In *Proceedings of 8th Joint Conference on Mathematics and Computer Science, ISBN 978-963-9056-38-1*, pages 403–416, Komárno, Slovakia, July 2010.

[31] M. Misiak, A. Schreiber, A. Fuhrmann, S. Zur, D. Seider, and L. Nafeie. Islandviz: A tool for visualizing modular software systems in virtual reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 112–116, Sep. 2018.

[32] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[33] Wettel Richard and Lanza Michele. Visually localizing design problems with disharmony maps. In *Softvis 2008 (4th International ACM Symposium on Software Visualization)*, pages 155–164. ACM Press, 2008.

[34] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. Cityvr: Gameful software visualization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 633–637. IEEE, 2017.

[35] Leonel Merino, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel A Keim. On the impact of the medium in the effectiveness of 3d software visualizations. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 11–21. IEEE, 2017.

[36] SourceTrail. A cross-platform source explorer for C/C++ and Java*. https://www.sourcetrail.com/, 2019.

[37] SourceTrail. A cross-platform source explorer for C/C++ and Java*. https://www.sourcetrail.com/, 2019.

[38] Nick Qi Zhu. *Data visualization with D3. js cookbook*. Packt Publishing Ltd, 2013.

[39] 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.

[40] Farrukh Shahzad, Tarek R Sheltami, Elhadi M Shakshuki, and Omar Shaikh. A review of latest web tools and libraries for state-of-the-art visualization. *Procedia Computer Science*, 98:100–106, 2016.

[41] ELTE-Ericsson Software Technology Lab. Development and testing at scale. http://compalg.inf.elte.hu/~attila/Testing_at_scale.html, 2019.

[42] K. Szabados, A. Kovács, G. Jenei, and D. Góbor. Titanium: Visualization of ttcn-3 system architecture. In *2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–5, May 2016.