Gview: Visualising software dependencies in order to support code comprehension[1]

Mátyas Komáromi, István Bozó[0000−0001−5145−9688], **Melinda Tóth**[0000−0001−6300−7945]

ELTE, Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers
makom789@gmail.com, toth_m@inf.elte.hu, bozo_i@inf.elte.hu

Abstract

It is always a great challenge to maintain industrial-scale software. It requires a full understanding and awareness 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 humans are more efficient at understanding visual information than written. 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. We integrated Gview with RefactorErl. RefactorErl is a source code analyser and transformation tool that also supports code comprehension for Erlang. The tool represents the syntactic and semantic information in the Semantic Program Graph, containing a massive amount of nodes and edges as input for Gview.

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 [5].

The visual representation of software may improve the productivity of developers, as it supports code comprehension, helps to find inconsistencies, and to improve 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 [3] is a static source code analyser and transformation tool for Erlang. It aims to support the everyday code comprehension tasks of 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 the huge Semantic Program Graphs [9] as well.

The main contributions of this paper are the introduction of Gview which is a new interactive graph visualisation tool and the extension of RefactorErl that integrates Gview. Gview was designed to utilise the GPU resources, and 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. Section 2 introduces the tool RefactorErl and the first prototype of Gview. Section 3 presents details about the generalised and RefactorErl-independent Gview and its integration with RefactorErl. Section 4 describes use cases about how to use Gview for code comprehension. Finally, Sections 5 and 6 present related work and conclude the paper.

[1] "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development, and Innovation Fund of Hungary, financed under the Thematic Excellence Programme no. 2020-4.1.1.-TKP2020 (National Challenges Subprogramme) funding scheme.

2 Background

Erlang [1] is a functional, concurrent programming language that was designed to build distributed, soft real-time, robust, fault-tolerant applications. Although the language is functional, industrial-scale applications require tools to support software maintenance, code comprehension, refactoring, etc.

RefactorErl [3] 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 [9]. 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 [21] provides more than twenty refactoring steps for the users. Besides the well-known renaming, moving, etc. transformations RefactorErl supports parallelisation by refactorings [31, 4, 14].

RefactorErl aims to support code comprehension in various ways. It defines a query language [30] 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 and elimination component.

RefactorErl is able to handle industrial scale applications [28]. 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. 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 [11] was only able to visualise the module/function views of the *SPG* that was printed to a static *dot* file [13].

3 Gview – Visualising software components

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, the dataflow view of a given variable shows all the possible ways how data could be assigned to it. 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 1 shows an overview of the internal structure of Gview.

3.1 Data Transfer

RefactorErl has its own graph-based internal representation to store the analysed Erlang source code. The graph is accessible through an Erlang querying interface. The data is stored in Erlang terms. Thus our first sub-task is to define a protocol to transfer the data from RefactorErl to Gview.

Our initial approach was based on intermediate data storage such as 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 dependent on the specifics of the Semantic Program Graph. To improve the visualisation, we introduced dynamic data transfer [16]. The transfer is done through our binary

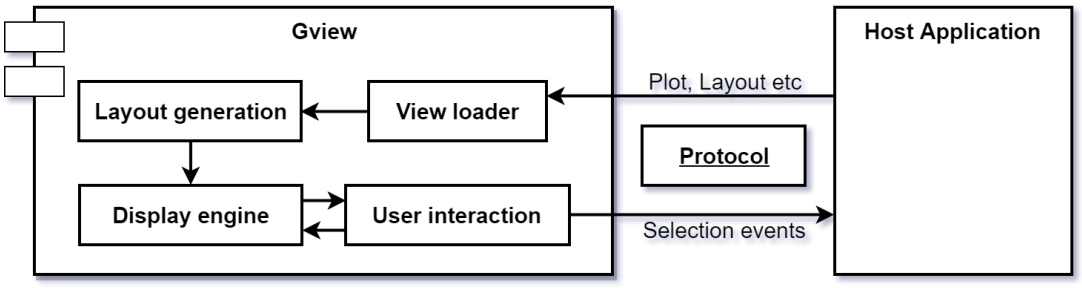

Figure 1: Representation of the inner division of sub-tasks in Gview.

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 in Figure 2, stating the used version of the protocol, which is currently 1.0, resulting in an error if the two are not compatible.

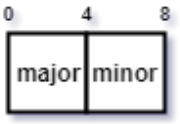

Figure 2: The initial two-way handshake message. Not a large but rather important message.

After that, the plotter application is up and running, and is waiting for new commands. The command string is sent in ASCII encoding, while the label string is 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 in Figure 3.

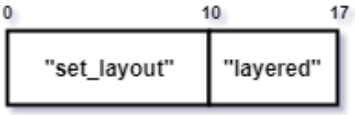

Figure 3: Example of a message sent by the host to set the used layout algorithm to layered.

The host can also issue a plot command via sending "set_view" and then sending the description of the graph. A summary of this message can be seen in Figure 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 can be found in Section 3.4). Each entry of the node palette describes the appearance of a given type of node, 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 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

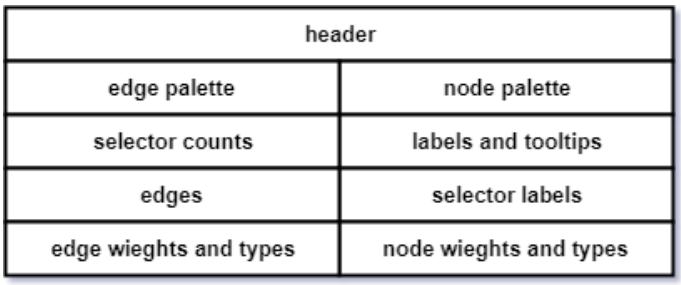

Figure 4: Summary of the complete plot message.

labels of the nodes, the tooltip strings, the selector counts, and the 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 in Figure 5.

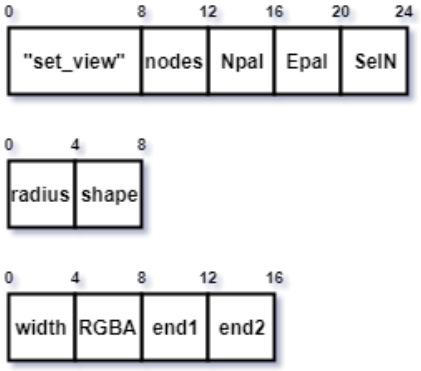

Figure 5: The header message and the palette entry messages of a set_view message.

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.

3.2 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 [17], 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, which 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 parallelizing 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 an 8 by 8 grid is shown in Figure 6.

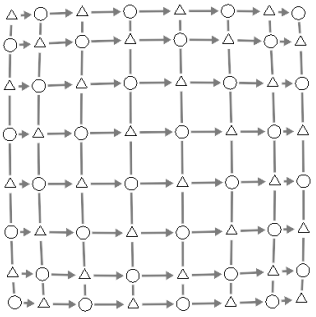

Figure 6: Force-directed layout

The Layered Hierarchical Layout [2, 26, 6] 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 the ordering of nodes on each layer to minimize 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. Our 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 in Figure 7.

While Layered Hierarchical Layout is working well for trees, the Force-Directed Layout is optimal for graphs with no special properties or specific structures 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.

3.3 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 of the massively parallel architecture of modern GPUs. Thus we based Gview on the cross-platform application programming interface

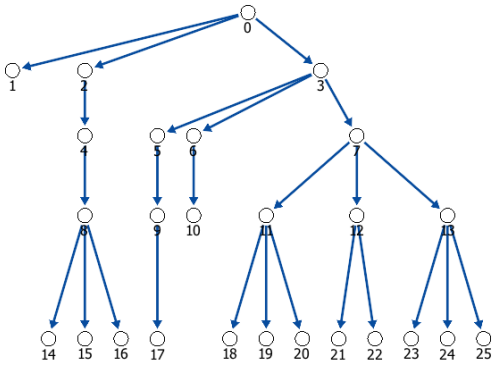

Figure 7: Layered layout

(API) OpenGL (Open Graphics Library) [24]. With OpenGL one is able to utilise the GPU to render the 2D meshes generated from the layout algorithm. It also features Compute Shaders which 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 [12] 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 an 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 the 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 [10] (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 the 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 organised into single batches per triangle and line, 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 of 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 1 ms, 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, does 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 3.4) synchronizing 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 in Figure 8.

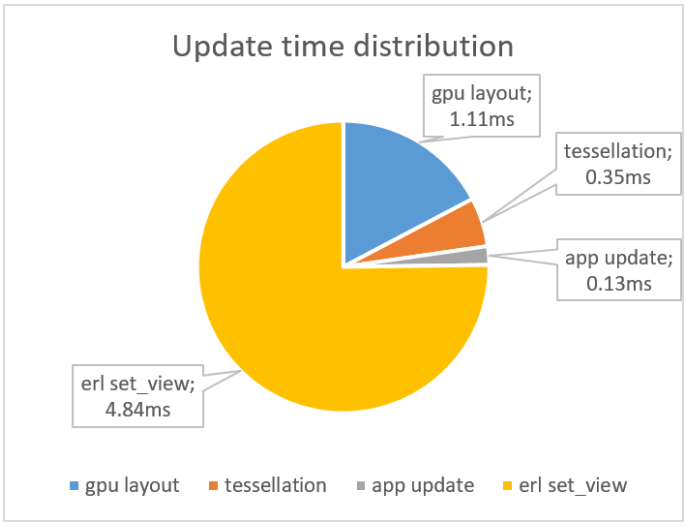

Figure 8: Relation of the time spent on transferring a view, updating one iteration on it, drawing it, and synchronisation. Note that layout updates occur much more frequently than data transfers.

3.4 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 3.1. Different types of nodes may have different sets 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 ignoring them at all without any problem.

Our previous approach [11] 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 in Figure 9: the first is responsible for user interaction handling and drawing the actual graph quickly from the last synchronised layout, and the second is 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.

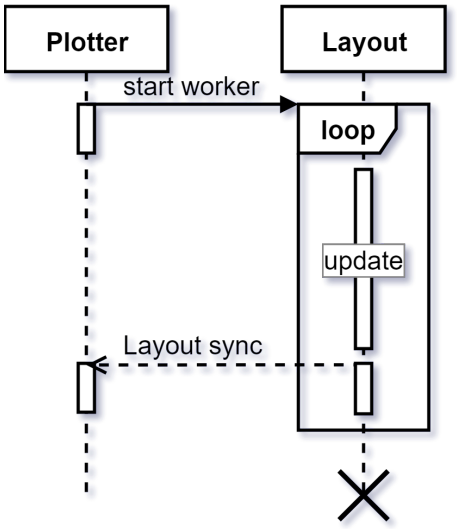

Figure 9: Interaction of the rendering thread which also handles the user interaction and the worker thread that generates the layout.

3.5 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 the 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 requires 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 thousand 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.

According to Erlang's "Let It Crash" philosophy, we wanted to let the smaller building blocks of the tool crash and then restarted without the end user even noticing it, thus 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, and 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, such 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 in Figure 10.

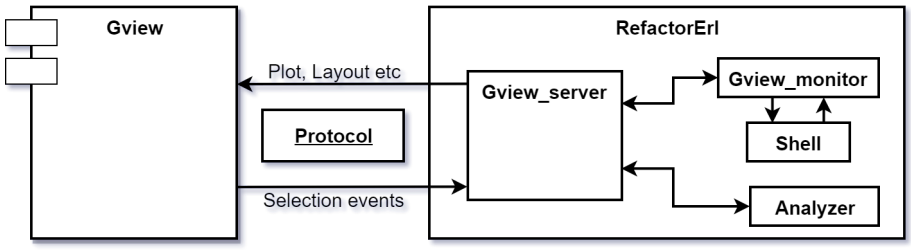

Figure 10: Our graph visualisation architecture, Gview.

4 Using Gview for code comprehension in RefactorErl

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 [15] 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 11 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 2104 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 12 shows the graph created when we clicked on module `mnesia_log`.

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.

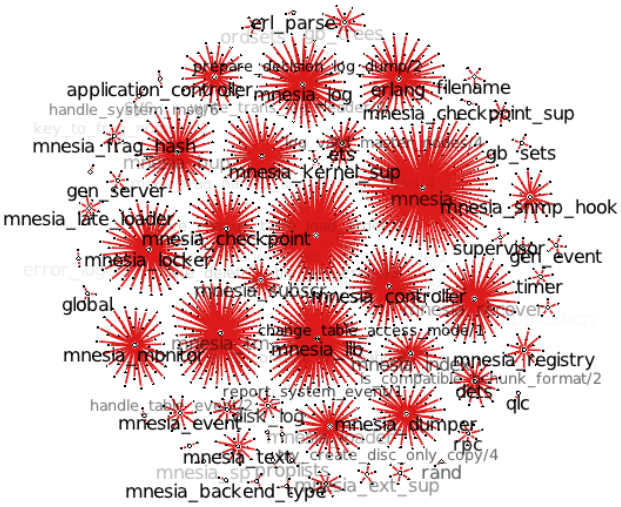

Figure 11: Mnesia view

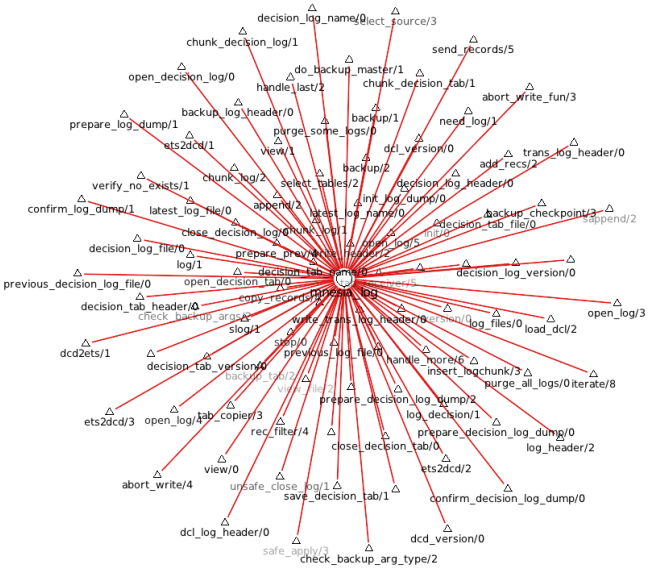

Figure 12: mnesia_log view

4.1 Call graph view

Clicking on the `open_log/3` function in Figure 12 results in the graph shown in Figure 13. This function call graph contains information about the called functions and the caller functions as well. Coloured, directed edges differentiate the functions calling `open_log/3` (blue edges) and those that are called by `open_log/3` (red edges). The software checks the size of the call graph and sets a call depth limit if needed to result in a comprehensible graph. The minimum level of depth is one. The depth of the plot can be adjusted by the user (with buttons located right above the root node).

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 shown in Figure 14.

The user can easily switch between the force-directed and layered layout and choose the most

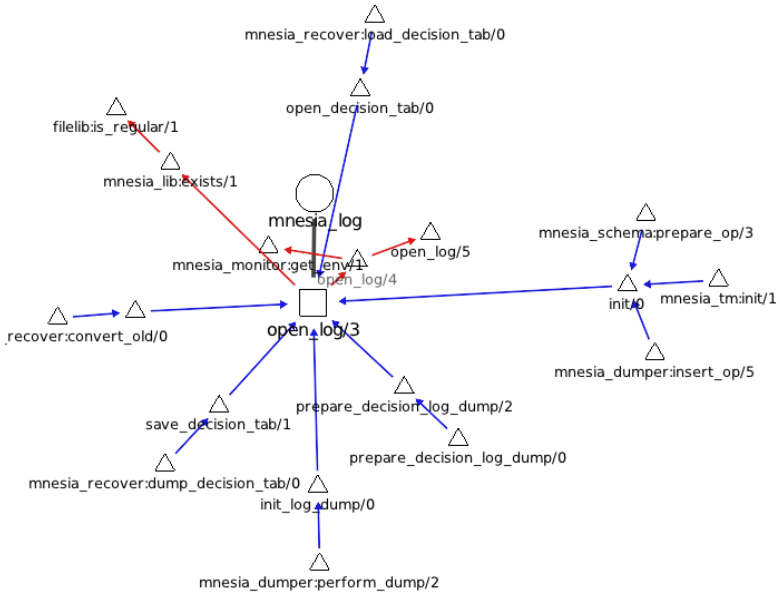

Figure 13: `open_log/3` function call graph

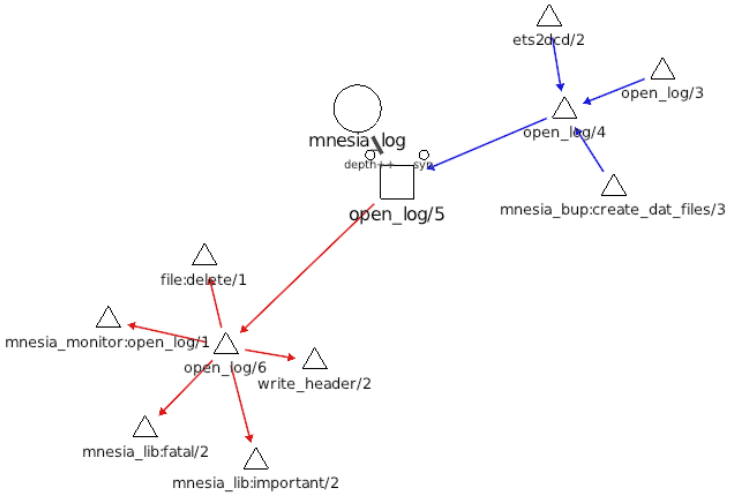

Figure 14: `open_log/5` function call graph

appropriate for the task. For example, the following command results in the view presented in Figure 15:

```
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 tools to help the developer to draw and explore the call graph. The semantic information available in RefactorErl extends the classical static call graph views with dynamic call

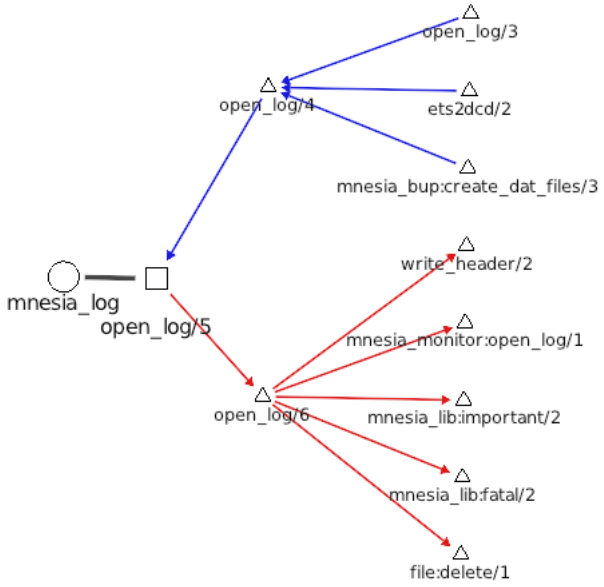

Figure 15: `open_log/5` function call graph (layered view)

information as well [8].

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 in Figure 16.

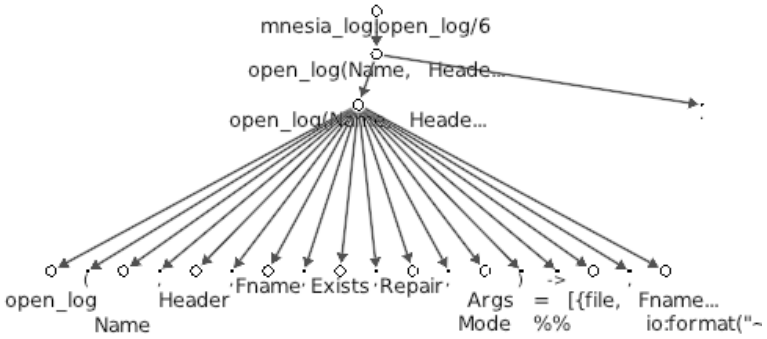

Figure 16: `open_log/6` high level syntax view

The syntax view can be further expanded with new levels of details from the syntax tree until the developer finds the appropriate information (Figure 17).

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, 27]. The dataflow reaching analysis of RefactorErl is able to calculate the possible

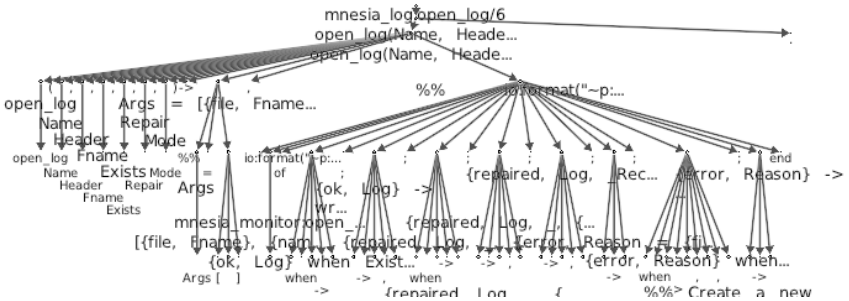

Figure 17: Extending the syntax view of `open_log/6`

values of a particular expression. During the analysis, all possible execution paths are considered and the possible values of the expressions are collected. This information can be useful in the case of debugging, code comprehension, testing, etc. For example, pointing the possible values of an expression can help in fault localisation when we are analysing runtime errors containing some information about the values causing the error. Using dataflow reaching we can also calculate the possible arguments of a function call, thus, we can narrow the scope of our investigation by selecting and focusing on the matching function clauses only.

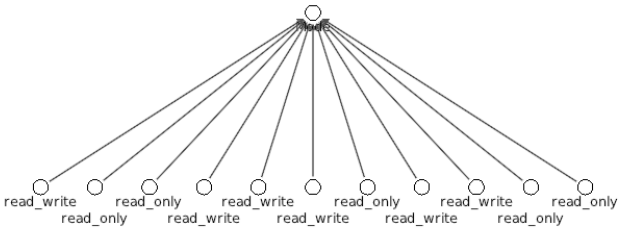

Figure 18: Values of the variable `Mode`

Using Gview, the developer who is interested in the possible values of a variable only needs to click on the targeted variable node. For example, by clicking on the variable `Mode` on the syntax view (Figure 16) we can deduce from the generated graph (Figure 18) that the possible values are `read_write` and `read_only`.

However, the desired information is not always available within a single step. Clicking on `Fname` in Figure 16 does not give us the possible values as a constant expression (Figure 19) since it is calculated by the function application `filename:join(...)`.

Figure 19: Values of the variable `Fname`

Thus, we have to further investigate the structure of the application that is concatenating a directory name with a filename. Once we click on the function application node (the right-hand side node in Figure 16), Gview plots the syntax tree of the application (Figure 20). Here we can find the name of the file represented by the `Fname` node. Selecting this node will list all the possible file names used during the concatenation (Figure 21).

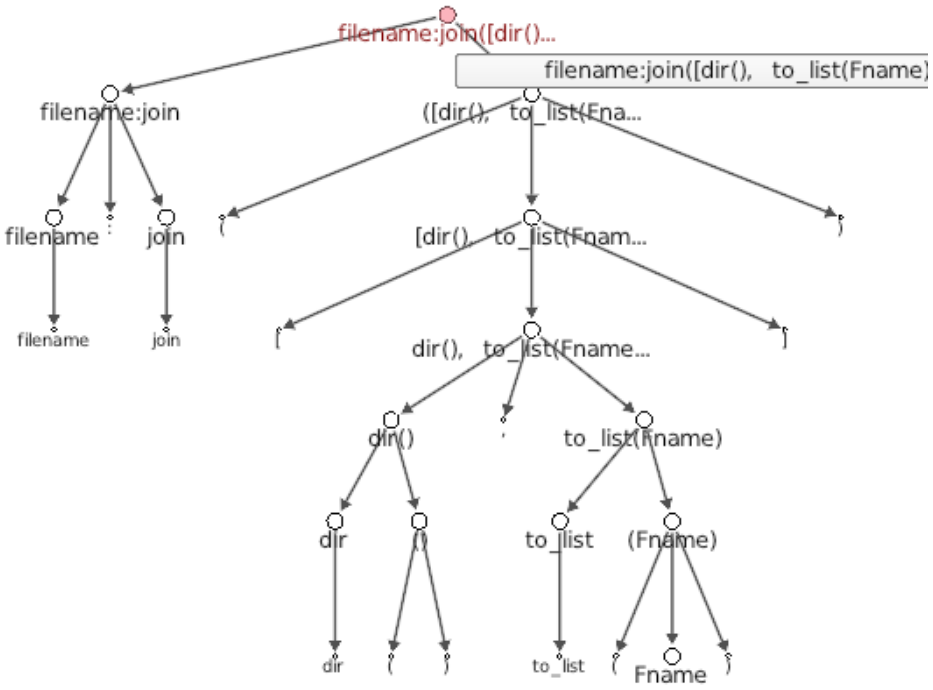

Figure 20: Exploring the syntax tree values of the application `filename:join(...)`

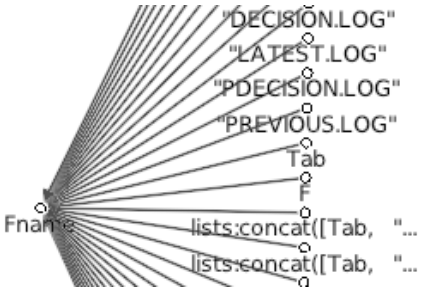

Figure 21: Exploring the values of the variable `Fname`

Exploring the syntax tree and using dataflow relations are extremely useful in bug-fixing tasks. 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.

4.3 Using the interactive graphs

RefactorErl can export the Semantic Program Graph to a dot file, thus for a few lines of code `dot`[2] is able to visualise the SPG. However, even for a small application (Crypto, 3KLOC), the graph contains thousands of edges and nodes. For a medium-scale application (Mnesia, 26KLOC) the graph contains more than a million relations (See Figure 7.1 in [16] and a partial view in Table 1). `dot` is not able to generate a layout for such applications, therefore without Gview, we were not able to visualise the SPG. It is obvious, that the user does not want to see thousands of nodes on a static graph, but rather wants to start with a high-level view and dynamically expand it to get the desired details. Using the external `dot` command was not the best tool to develop this functionality, thus we started the implementation of Gview.

Table 1: Exporting the Semantic Program Ggraph [16]

Project	DOT size	DOT generation	Nodes	Edge count	Function count
Crypto	9,84 MB	22,3s	24 551	86 222	174
Sasl	45,84 MB	151,9s	107 936	438 508	928
SSH	94,71 MB	241,2s	230 398	913 924	1117
Mnesia	141,79 MB	477,1s	332 360	1 374 466	2104
StdLib	283,74 MB	1028,4s	652 300	1 438 325	3206

RefactorErl has a dependency graph calculation and visualisation component. This can be considered as predefined views on the module and function level. The function level dependency graph is a call graph. We compared the layout generation for the dependency graphs and the function call graph view generation of Gview. For the Mnesia application generating the function dependency graph took more than an hour, and it took for 4 minutes to open the static graph. Gview was able to generate the layout in 2 minutes and open it in 14 seconds to interactively traverse and further expand it.

5 Related work

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

With the tool IslandViz [20], 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 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. 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

[2] A command from the GraphViz toolchain

the Force-Directed Layout algorithm [17] also targets the massively parallel architecture of modern GPUs which results in a massive performance increase over the CPU implementation.

Graphviz [7] is a well-known graph visualisation software package developed by AT&T Labs Research in 1991 and is open-source. Among the supported layout generation algorithms, there are energy-minimising, stress-majoring methods, and hierarchical ones, each of which has its 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 and computer-aided design systems such as FreeCAD.

Numerous software packages 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 resulting 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 requirements.

CodeCity [22] 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 neighbourhoods 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.

CityVR [19] 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 themselves 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 [18].

Sourcetrail [25] 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 find 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 more convenient. By accompanying diagrams with relevant code snippets, Sourcetrail shortens the investigation time. 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 the case of Erlang projects.

Understand [23] 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.

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, which 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 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 in 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.

References

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

[2] 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.

[3] 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.

[4] István Bozó, Viktoria Fördös, Dániel Horpácsi, Zoltán Horváth, Tamás Kozsik, Judit Köszegi, 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] Mátyás Komáromi. Flib project github page. https://github.com/Frontier789/Flib/, 2016-2019.

[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] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] Melinda Tóth. Using the open-source RefactorErl in Ericsson. Talk at Functional Programming Meetup, Craft Edition, 2015.

[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] 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.

[31] 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.