STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.02

AN EFFICIENT GRAPH VISUALISATION FRAMEWORK FOR REFACTORERL

MÁTYÁS KOMÁROMI, ISTVÁN BOZÓ, AND MELINDA TÓTH

Abstract. Graph visualisation is a well-known and researched field of graphical informatics. Several good algorithms were developed and reviewed by our days. However, most of the graph drawing tools mainly focus on static drawing generation. In this paper we present an approach that is efficient enough to visualise the user-requested parts (views) of a relatively large Semantic Program Graphs of Erlang projects in soft real-time. With the presented approach the visualised graphs can be traversed interactively, by changing between different levels of detailed information, which may support code comprehension in the RefactorErl framework.

1. Introduction

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

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

RefactorErl [11, 22] is a static source code analysis and transformation tool for Erlang [10]. Besides the more than 20 refactorings, the tool provides several functionalities to support program comprehension: semantic queries,

Received by the editors: 31 March 2018.

2010 *Mathematics Subject Classification.* 68N01, 65D18.

1998 *CR Categories and Descriptors.* I.3.1 [**COMPUTER GRAPHICS**]: Hardware Architecture – *Parallel processing*; I.3.5 [**COMPUTER GRAPHICS**]: Computational Geometry and Object Modeling – *Physically based modelling*.

Key words and phrases. graph visualisation, layout generation, physically based modelling, code comprehension.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

static-dynamic call analyses, data-flow analysis, dependence analyses, etc. The results of several analyses can be visualised as graphs.

Also, the tool itself uses a graph based intermediate representation for the source code: the Semantic Program Graph [16]. The SPG contains lexical, syntactic, and semantic information about the source code. These three layers generate huge amount of data (nodes and edges) from the source code. Even for a module with few hundreds of lines of code (LOC) the standard visualisation tools, such as Graphviz [5], are hardly able to generate a proper view.

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

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

The main contribution of this paper is a solution to the above presented problem. We are providing a graph visualisation method and a new component *gview* for RefactorErl that is capable of handling real Erlang projects. We demonstrate different views available through the new component and evaluate the performance on different open-source projects.

2. Related work

Graph visualisation has been subject to research since long time ago, many good visualisation tools are available for use today.

2.1. **Graphviz.** Graphviz [5] is an open source graph visualisation software. It supports many input formats, specifications, and algorithms for presenting graphs. However, Graphviz is unable to render graphs with high node count, in an interactive manner, as experienced during the development of the user interface of RefactorErl. Rendering the main view of Mnesia into an svg file with Graphviz, consisting of around 2200 nodes, took around 3700 seconds (more than an hour). After the layout generation, opening the generated svg file in a browser took 4 minutes. The layout for the very same view can be generated by *gview* in 2 minutes, cached, and then displayed interactively.

2.2. **Wolfram Mathematica.** Wolfram Mathematica [9]: The Wolfram Language provides functions for the aesthetic drawing of graphs. Algorithms

implemented include spring embedding, spring-electrical embedding, high-dimensional embedding, radial drawing, random embedding, circular embedding, and spiral embedding. In addition, algorithms for layered/hierarchical drawing of directed graphs as well as for the drawing of trees are available.

2.3. **MSAGL.** MSAGL [6]: MSAGL is a .NET library and tool for graph layout and viewing. MSAGL was developed in Microsoft by Lev Na-chmanson, Sergey Pupyrev, Tim Dwyer, Ted Hart, and Roman Prutkin.

2.4. **Erlgraph.** Erlgraph [2] is an application which enables a d3js force directed graph to connect to an Erlang VM. The plotted data consists of the active processes of the running application and the messages they send or receive. D3.js is a JavaScript library for manipulating documents based on data. D3 aids creating data driven animations using HTML, SVG, and CSS. D3 is said to emphasis on modern web standards, which gives full capability of modern browsers without tying to a proprietary framework. Erlgraph provides an insight into the underlying mechanism of an Erlang application in runtime. Erlgraph is an extremely useful tool for visualising how processes of the project interact with each other. What different in *gview* and Erlgraph is that while Erlgraph realises a dynamic (runtime) analysis of Erlang code, *gview* targets static analysis of the Erlang project at hand, which means no code needs to be executed.

3. Background

Gview is built upon Flib [4]. Flib (at the time of writing this paper) is a single-author OpenGL development library for C++. It supports creating and handling Windows, OGL contexts and OpenGL objects. It also has a GUI system, graphical and linear mathematical tools. On windows platforms, Flib uses the standard Windows API for window creation and management, on Linux platforms, it uses the XLib windowing system. Consequently, OpenGL context creation is done using WGL and GLX respectively. The Flib API documentation generated by doxygen can be found on the online repository [3].

4. Visualisation framework: *gview*

4.1. **Overview.** Dot [18] is a general purpose graph describing language capable of representing directed and non-directed graphs alike, with extra information options on both edges and nodes. RefactorErl supports exporting all the data from semantic graphs of loaded Erlang files to a single dot file. Therefore, by parsing this exported dot file *gview* is able to create a layout for the views of the graphs and display these views interactively. The architecture of the software can be seen on Figure 1.

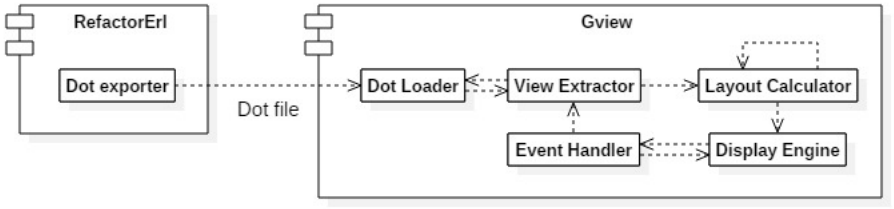

FIGURE 1. The architecture of *gview*.

4.2. **Dot files.** The main reason of using dot files is the existing support for it in RefactorErl and the ease of implementing a custom parser in C++. On the other hand, using dot files as intermediate data representation is quite rigid. Having a file changed one must regenerate the dot file in RefactorErl, and on every startup, *gview* has to parse the whole dot file. The data-flow of the program can be seen on Figure 2.

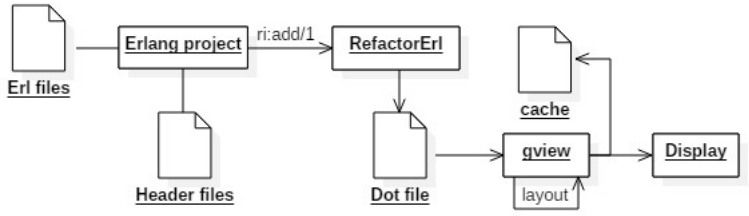

FIGURE 2. Data-flow in the visualisation process.

4.3. **Rendering.** The rendering is done using OpenGL [21] with the supporting classes of Flib. We preferred OpenGL because it is hardware close and very fast and, unlike DirectX, portable across operating systems. The drawing data is generated on the fly after each iteration of the layout algorithm, thick lines are tessellated into triangles, circles into regular polygons by the C++ implementation. Extra information on vertices for anti-aliasing is also added in this process. The drawing data is then uploaded to the graphics card and drawn as a single batch, avoiding the cost of setting up many drawing calls. The anti-aliasing is done by our shader program on the GPU.

4.4. **Layout.** Our layout calculating algorithm is a modified version of two-dimensional N-Body simulation, known as Force-Directed Layout (FDL) [14]. In a classic N-Body simulation we would have N objects, each pair exerting attracting gravitational force, proportional to the mass and inverse square of the distance, to each other. However our FDL algorithm considers these objects (nodes of the graph) to have electric charges, instead of gravitational effects, and thus repel one-another. Furthermore edges between nodes of the graph are represented as springs of logarithmic strength, meaning the force they exert is logarithmically proportional to the distance they stretch across. This way the edges attract nodes they connect. After the initial setup Acting net forces are calculated in every iteration for each node, then we update the position of nodes according to these net forces and the elapsed simulation time. Forces are taken to act instantaneously, which means they are applied directly to the position of nodes not on their speed.

There are many toggleable elements of this simulation; the strength of the springs, the amount charge a node has, the stepping time between iterations of the simulation, and initial positions of the nodes. Choosing these parameters were done on an empirical basis; we experimented with them until the results looked good. Therefore, by modifying the charge of nodes or the strength of the springs we can change the final spacing among edges or nodes. This way we can emphasis parts of the displayed graph.

4.5. **Related libraries.** Beside the Flib there are many other excellent frameworks for OpenGL development.

SDL (Simple Directmedia Layer) [12] is one of the oldest of these frameworks, it has outstanding wide system and hardware support. However, its interface was designed for C not C++, SDL does not use object-oriented paradigms. SFML (Simple and Fast Multimedia Layer) [8] is another excellent choice for OpenGL development. It is completely object oriented (by the C++ binding) with cross-platform support, but it lacks the GUI module. Qt [7] is a professional and robust framework with good GUI and OpenGL support.

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

4.6. **GUI Framework.** Flib provides GUI classes on top of OpenGL. *Gview* uses Flib to automatically open a window, an OGL context associated with this window, and a GUI context which is responsible for storing GUI related data

such as fonts and shaders. The GUI context also has a main GUI layout where the application can attach GUI elements. The GUI elements are structured in a tree pattern: each GUI element may have a parent layout, each layout may have any number of children elements, layouts are GUI elements too.

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

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

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

Uploading the mesh is done using the designated interface of Flib for simplicity. Tessellation of lines uses the built-in tessellation functionality from Flib which automatically includes distance-field data needed by the anti-aliasing technique. The data is drawn using the drawing API of Flib as well. Results of the tessellation can be seen on Figure 3.

4.8. **Dynamic Level of Detail.** A very useful technique for graphical applications is Dynamic Level Of Detail [23] (DLOD), it involves altering the detailedness of an object (how many triangles it has or what textures it uses) based on how small that object appears on the screen. This technique is effective because our eyes can not make out the difference on small objects, we use DLOD in *gview* to reduce workload on CPU when the mesh is generated. As for an example, circles that represent module or file nodes get drawn as regular n-sided polygons. The value of n is based on the zooming, the more it is zoomed in, the larger the value of n is. Application of DLOD is shown on Figure 4.

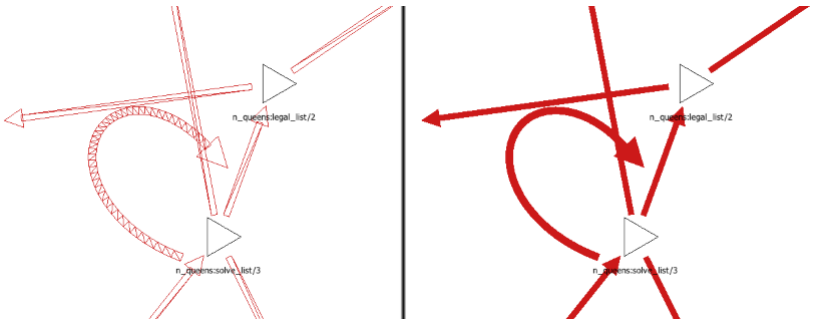

FIGURE 3. Tessellation of the edges (the thickness of the edges was increased for demonstration).

To calculate the number of sides our polygon needs, when approximating a circle, let us say the circle has r radius given in pixels on the screen. The size of the radius in pixels can be calculated from the resolution at which the rendering is done and the zooming level. Considering the formula for the circle perimeter $2*r*\pi$ we can approximate the number of pixels on the perimeter of the circle easily, as $p = 2*r*$M_PI in C++. When drawing a circle of radius r, we ought to approximate a curve of length p. After taking measurements, we found that using $p/4$ line segments produces satisfactory images. It is also worth noting, that letting the number of sides drop below 3 is trivially pointless.

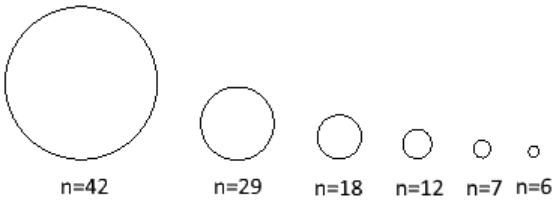

FIGURE 4. Number of sides (denoted as n) of regular polygons representing circles with different zooming level.

4.9. **Anti-aliasing.** Anti-aliasing [13] is done using distance fields, through a technique called distance-to-edge anti-aliasing (DEAA) [17]. The main idea is that, if we know the distance from the edge of the primitive when processing a fragment, then we can set the transparency to drop when getting near the edge of the primitive. Thus, having the transparency stored in the alpha channel of the fragment, the OpenGL blending mechanism will ensure it will be displayed

transparently. The effect of using anti-aliasing can be seen in Figure 5. The distance function is calculated as the minimum of the distances to the edges of the mesh. For this purpose, we store the distances in separate channels (red channel holds the distance from the left edge, green channel from the right edge etc.). Calculating these distances on a per-vertex basis and using the OpenGL built-in linear interpolation functions, we can approximate the distance to the edges of the mesh as the minimum of the interpolated distances.

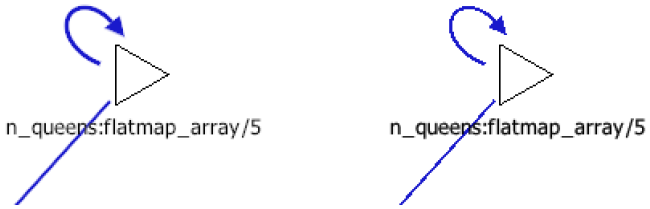

FIGURE 5. Call and recursive edge with and without antialiasing.

5. Generating the layout

The layout of a displayed graph is defined as a list containing points (two-dimensional) for each node of the graph. The algorithm described below takes an initial layout and processes it in iterations. We use the Force-directed layout algorithm, that is described in more detail in Section 4.4. For experimenting, we have defined and implemented three different versions of the algorithm (Sections 5.1, 5.2, and 5.3). The efficiency of different implementations can be seen on Figure 6, where we plotted the number of iterations it took to generate the final layout for tested views, against the number of connections in the views, by connections we mean the number of potential forces acting in the simulation. The number of these connections is in $\mathcal{O}(n^2 + e)$, where n is the number of nodes and e is the number of edges.

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

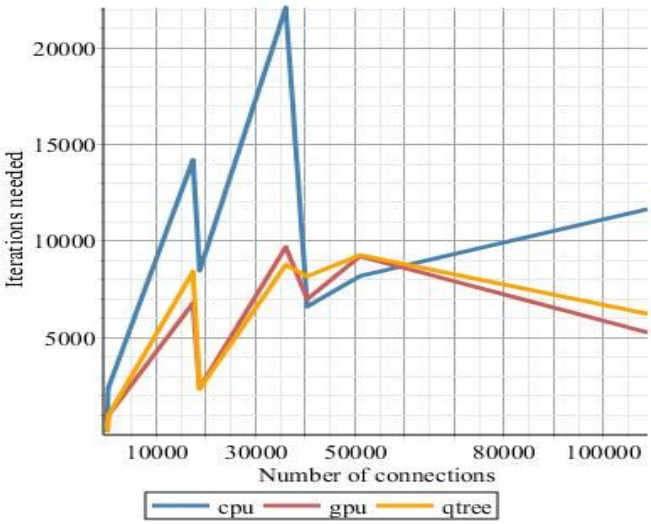

FIGURE 6. Iterations needed to reach final state plotted against number of connections (n^2+e) in view.

5.2. **QuadTrees.** The second implementation uses quad trees for space partitioning to reduce computation time. These quadtrees have regions as nodes, generated by recursively dividing the points in a region into two sets of the equal size, using horizontal splitting lines and vertical splitting lines alternately. This method is also called as Barnes-Hut algorithm [19]. The recursion stops when a certain minimum of nodes in a region is reached. Therefore, when calculating the net force on a node we can traverse the generated tree and approximate the net force exerted by all the nodes in a region of the tree in constant time when it is far enough from the currently processed node. This technique results in $\mathcal{O}(n * log(n))$ time complexity when the distribution of the nodes is even. The problem is that the constant factor of the complexity is quite big as the tree has to be regenerated in each iteration (and not CPU cache friendly). On large views this method outperforms the trivial CPU implementation.

5.3. **GPU implementation.** The third implementation is the parallel equivalent on GPU of the first one using OpenGL Compute Shaders. We chosed OpenGL over Cuda or OpenCL because it has good support for AMD and Intel cards too, integrates nicely with the rest of the drawing code, and requires no extra libraries. The graph is sent to the GPU in adjacency matrix representation in a texture image. A kernel is then dispatched for each node

of the graph which computes the net force acting on that node. The GPU implementation reuses the calculated data and thus data is only streamed to the CPU once every frame (typically 20 iterations). As a consequence, when the number of nodes is smaller than the number of shader cores, the GPU is not used efficiently. This issue could be remedied using a divide and conquer approach: each kernel only calculates the net force on a node exerted by a portion of other nodes. Then another shader is dispatched to sum up the subresults.

5.4. **Alternatives.** Stress majoring [15] is another good algorithm that we looked into using. However, on lack of time it was not implemented. Thus, it is not discussed in this paper.

5.5. **Caching.** Generating the final layout of a graph view is very computational and time costly. Therefore, caching the generated layouts can save important resources and speed up *gview*. Caches need to be stored permanently, thus they are saved as external binary files using the file streams of the C++ standard library.

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

The other way is keeping one cache file per project. This way, the cache management is easier and clearer. The resulting cache file sizes depend on the opened views. The cache files barely reach 50kB even on the largest test project: Mnesia [20].

6. RefactorErl graph views

Plotting the whole graph exported to the dot file would be pointless and computationally extremely expensive. Consequently, we define views of the graph and only plot one of these views at a time reducing workload and letting the user concentrate on one aspect of the project.

6.1. **Main view.** The main view consists of all the modules and functions defined or referenced in the application. Each function is linked to the module they are defined in. One can change the view by clicking on a module node, then the view for that module is shown.

6.2. **Module view.** The module view is similar to the main view, but it displays only one module and the functions defined by the actual module. It has a ROOT node through which one can return to the main view. One can change the view by clicking on a function node, than the view for that function is brought up. The massive main view of Mnesia is shown on Figure 7.

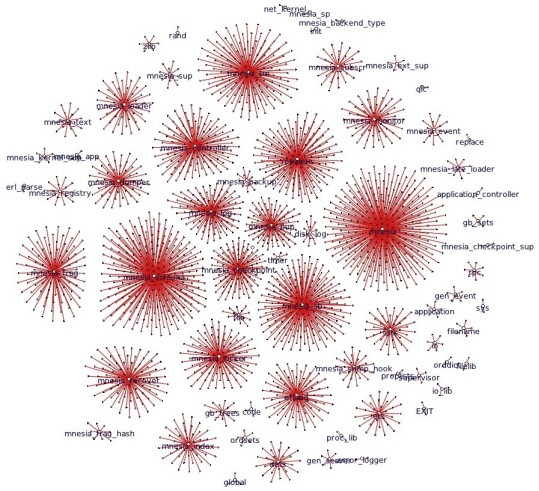

FIGURE 7. The main view of Mnesia containing more than 2500 nodes and edges.

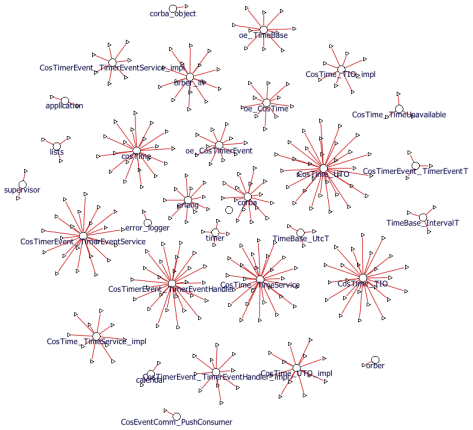

FIGURE 8. The main view of the CosTime application.

6.3. **Function view.** The function view plots a function and all functions that directly or indirectly call/get called by it (Figure 9). The plot depth of the call graph can be adjusted, this depth is an argument of the view. Currently the depth is not limited, but in the future we plan to limit it to a reasonable size.

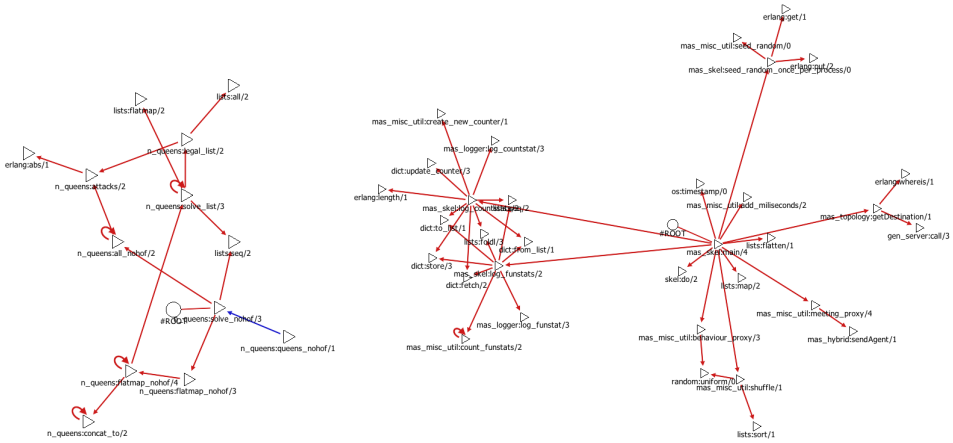

FIGURE 9. Deep function views of GreenErl and MAS projects.

6.4. **Visual elements.** The displaying engine in *gview* supports various graphical elements to provide information to the user. The shape of the nodes can be customised and set for example to triangle, square, or circle. The width of edges can be adjusted, the colour of the edges, the displayed text below nodes, and the shape of the arrows of edges can be customised as well. Through these options the view generator is able to highlight the differences between the semantic entities (function, module, etc.) of the displayed graph. Some of the available elements are shown on Figure 10.

7. EVALUATION

To measure the performance of *gview* and profile it we tested it on several open-source projects.

The largest project was the Mnesia [20], a robust, distributed database management system, written in Erlang. With more than 2500 functions (and around 25 thousands LOC) this is by far the largest project *gview* was tested on. Plotting all the texts of the main view interactively could be considered a challenge alone. Part of the main view is shown on Figure 11.

CosTime [1] is an Erlang implementation of the OMG CORBA Time and TimerEvent Services. It has many modules with few functions, thus it was a good candidate to test *gview* on. Main view of CosTime can be seen on Figure 8.

The measurements were made on loading of dot files and generating the views. We can conclude that the loading of dot files was the major slowdown on startup. The loading may take more than 90% of the start time. Other

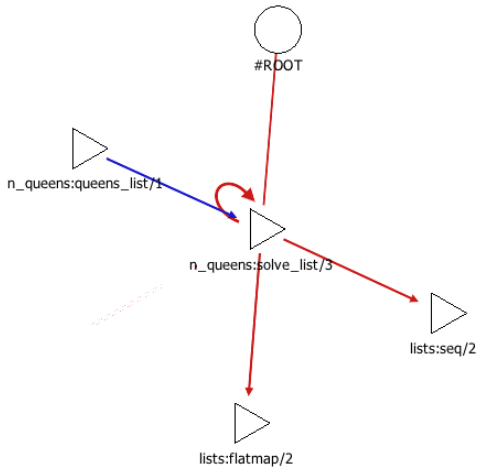

FIGURE 10. Some of the customisable visual elements: triangles, circles, edges, recursive edges, etc.

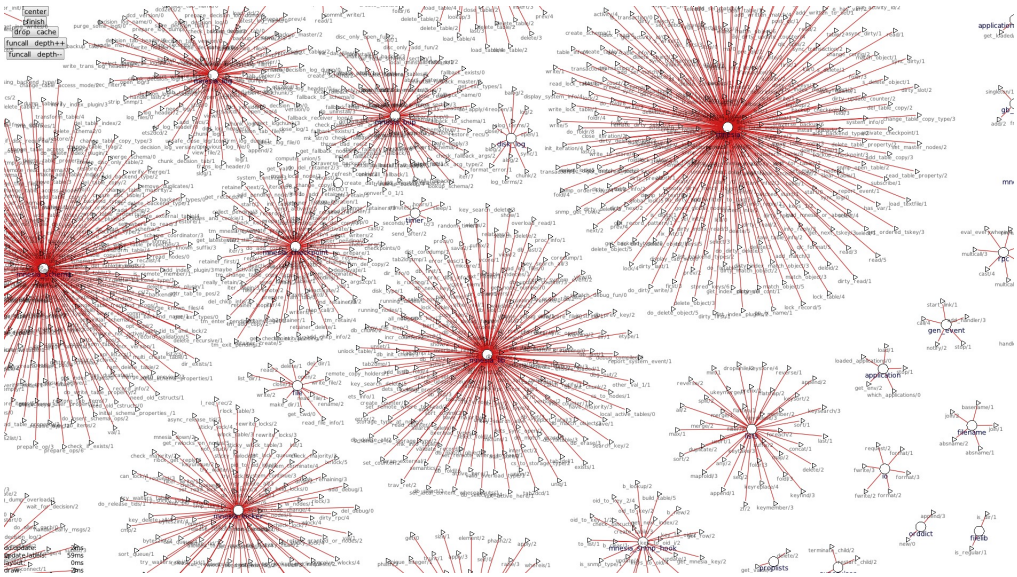

FIGURE 11. The massive amount of functions in Mnesia.

events performed on startup, such as window or OGL context creation take negligible time compared to loading and interpreting dot files.

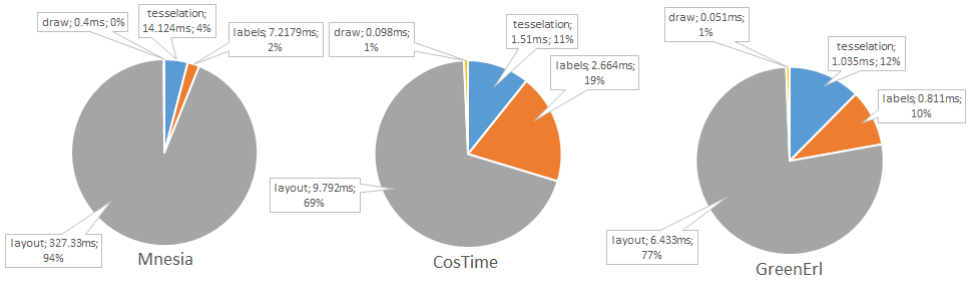

FIGURE 12. The average time taken for layout generation, tessellation, label setting, and drawing the view.

When displaying a view, the most time is spent on generating the layout (70% and up). Label placement and mesh generation could run in real-time (without layout calculation). Figure 12 demonstrates distribution of time spent on different stages. Analysing the charts, it is obvious that described caching mechanism largely improves the efficiency and ensures real-time response on large views.

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

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

8. Conclusion and future work

RefactorErl framework has several graphical and command-line interfaces, that support refactorings, static code analysis, and code comprehension as well. The tool uses a Semantic Program Graph as an intermediate representation of the source code. The SPG includes static semantic information beside the syntactic and lexical information. The conversion of the SPG to an SVG file with Graphviz was possible only on relatively small graphs. There was

high demand for an efficient and interactive graph visualisation tool that led us to create the *gview* component presented in this paper.

RefactorErl has support for exporting the SPG to a dot file, thus we used this functionality and the dot format for representing the graph. These files are then processed by our custom dot parser implemented in C++. The views of the exported graph are generated and visualised using OpenGL. We have used Flib for GUI and OGL object management. To calculate the layout of the graph we used Force-directed layout generation. To improve the performance of the interactive viewer, the resulted layout is saved to cache files to avoid the continuous need of recalculating. To enhance the visual quality of the rendered scene we used anti-aliasing. Dynamic Level Of Detail is applied to reduce geometry, which is then tessellated using Flib and drawn in a single batch. We made the appearance fully customisable, thus the users are able to use different shapes and arrows for different semantic entities and relations among them accordingly. The user is able to switch between views using the cursor; pointing on a node and clicking brings up a more detailed view associated with that node.

Although static data access through dot files was a good starting point, it turned out that processing/parsing the dot file is the bottleneck in graph generation. In the future we plan to replace it with dynamic graph information acquired directly from RefactorErl.

Also, using the GPU for parallelisation of mesh tessellation is also an appropriate subject for future research.

References

[1] The CosTime application. http://erlang.org/doc/apps/cosTime/cosTime.pdf. [access date: Jun. 2, 2018].
[2] Erlgraph on GitHub. https://github.com/aol/erlgraph/. [access date: Jun. 4, 2018].
[3] Flib documentation. http://makom789.web.elte.hu/docs/index.html. [access date: Jun. 2, 2018].
[4] Flib project github page. https://github.com/Frontier789/Flib/. [access date: Jun. 2, 2018].
[5] Graphviz homepage. https://www.graphviz.org/. [access date: Jun. 2, 2018].
[6] Microsoft automatic graph layout homepage. https://www.microsoft.com/en-us/research/project/microsoft-automatic-graph-layout/. [access date: Jun. 2, 2018].
[7] Qt — cross-platform software development for embedded and desktop. https://www.qt.io/. [access date: Jun. 2, 2018].
[8] Simple and Fast Multimedia Library. https://www.sfml-dev.org/. [access date: Jun. 2, 2018].
[9] Wolfram Mathematica homepage. https://www.wolfram.com/mathematica/. [access date: Jun. 2, 2018].
[10] Joe Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, 2nd edition, October 2013.

[11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.
[12] Fabian Fagerholm. Simple Directmedia Layer (SDL). 2006.
[13] A. R. Forrest. Antialiasing in practice. In Rae A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, pages 113–134, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
[14] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.
[15] Emden R. Gansner, Yehuda Koren, and Stephen North. Graph drawing by stress majorization. In János Pach, editor, *Graph Drawing*, pages 239–250, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
[16] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling Semantic Knowledge in Erlang for Refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, July 2009.
[17] 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 2011 Courses on*, page 6, 2011.
[18] Eleftherios E. Koutsofios and Stephen C. North. Drawing graphs with dot. *Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ*, 1991.
[19] Tancred Lindholm. N-body algorithms. http://www.cs.hut.fi/ ctl/NBody.pdf. [access date: Jun. 2, 2018].
[20] Håkan Mattsson, Hans Nilsson, and Claes Wikström. Mnesia - a distributed robust dbms for telecommunications applications. *practical aspects of declarative languages*, pages 152–163, 1999.
[21] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. OpenGL® Shading Language. 2004.
[22] 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.
[23] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of the 7th conference on Visualization '96*, volume 25, pages 327–334, 1996.

ELTE, Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest 1117, Hungary

Email address: {makom789, bozoistvan, tothmelinda}@caesar.elte.hu