Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study

Alexandru Telea* Hessel Hoogendorp† Ozan Ersoy‡ Dennie Reniers§
Institute of Mathematics and Computing Science
University of Groningen, the Netherlands
SolidSource BV
Eindhoven, the Netherlands

Abstract

Investigating program dependencies such as function calls is challenging for very large systems. We present here an integrated pipeline for extraction and visualization of call-and-hierarchy graphs for C/C++ programs. We present several adaptions and enhancements of a recent visualization method for large call graphs and compare its effectiveness with classical node-link diagrams. Examples are given on large real-world code bases such as bison, Mozilla and oink.

1 Introduction

Software systems contain large and complex sets of dependencies between their components, such as call and inheritance graphs, and data flow and type dependency graphs. Analyzing such dependencies is arguably one of the most important tasks of maintenance processes such as reverse engineering and reengineering. A good understanding of such data supports decisions for code refactoring, removing code clones, identification of design patterns, and debugging.

However, understanding large dependency sets is challenging. Visualization is a method of choice, given the inherent difficulty for understanding large, abstract graphs. Although numerous methods are being proposed for visualizing dependency graphs in the information visualization (InfoVis), software visualization, (SoftVis), and graph drawing (GD) communities, it is still unclear how such methods are received by software practitioners in the field, and how they compare when one must accomplish tasks in program comprehension.

In this paper, we focus on a subset of these activities, and look at the problem of understanding call graphs extracted from software systems which have a hierarchical structure. As we aim to understand the effectiveness of such methods in practice, several aspects are relevant besides the visualization method chosen, *e.g.* the availability of a robust method to extract the call graphs; the perfect integration of data extraction and visualization [Koschke 2003]; and the scalability of the entire pipeline to real-world systems of hundreds of KLOC.

We describe here an entire tooling pipeline that covers static code analysis, extraction of calls and hierarchy data and their attributes, and visualization. Our focus here is on C/C++ code bases. For this, we implemented a standalone call graph extractor based on the OINK framework [OINK 2008], one of the most complete, robust, and scalable open-source static analyzers for C/C++. Our call graph extractor enhances the OINK framework with several analyses important for call graph extraction, such as linking declarations to definitions across multiple translation units, and detecting the potential set of called candidates for virtual functions and function pointers. Besides calls, our extractor also delivers hierarchy data (folders, files, classes, methods) and various attributes thereof, such as the call type (static, virtual, by pointer or reference), and details over the function definitions (signature data, access rights, and source code location). The extracted data is saved in several easily importable formats.

For visualization, we needed a scalable, understandable, and easy to use method. As a candidate, we considered the recently published hierarchical edge bundling (HEB) technique, which was very well received in both the InfoVis and SoftVis communities [Holten 2006; Cornelissen et al. 2007]. However, a main question is: How does this technique compare with classical, more accepted, techniques such as node-link diagrams (NLDs)? Such a comparison lacks, and is needed, for large-scale graphs, as the author of the HEB technique also points out. To this end, we performed a study that compares our own implementation of the HEB which adds several enhancements we found useful, and several classical NLD layouts provided in the Tulip graph visualization framework [Auber 2009].

For this comparison and also to test our entire pipeline, we analyzed several large software systems written in C, C++, and a mix of the two, such as *bison*, *Mozilla Firefox*, and the OINK static analysis framework itself. The analyses were done by developers experienced in software engineering in general and C/C++ in particular, but had no knowledge of the analyzed systems. They had to answer several questions solely based on the two visualizations. We compared the results with the aim of drawing conclusions on the two types of visualizations.

Overall, we can describe our work using the 5-dimensional model of Marcus *et al* [Marcus et al. 2003]: our *task* is to analyze how two different visual metaphors support the visual understanding of call relations in large source code bases; the *audience* includes software developers, designers, and architects; the *target* is a graph containing attributed call and hierarchy data; the *medium* consists of two different visualization tools, the Tulip framework and our own enhanced HEB method; finally, the *representation* consists of various types of node-link diagrams and the hierarchical edge bundle metaphor.

This paper is structured as follows. In Section 2, we overview related efforts on static analysis of C/C++ with a focus on call dependencies, as well as visualizing call graphs in general. Section 3 presents our call dependency extractor for C/C++. Sections 5.1,5.3 and [OINK 2008] present the results obtained when visualizing call graphs extracted from the bison, Mozilla Firefox, and OINK open-source code bases. In this part, we also introduce the various enhancements we added to the original HEB technique. Section 5.5 discusses the results found in this study. Section 6 concludes the paper.

2 Related Work

Our goal is to analyze ways to visually explore call dependencies of large-scale C/C++ code bases (Sec. 1). Hence, related work addresses two topics: extraction of the dependencies from source code and visualizing the extracted data.

*a.c.telea@rug.nl
†h.hoogendorp@student.rug.nl
‡o.ersoy@rug.nl
§dennie.reniers@solidsource.nl

For data extraction, several so-called static analyzers exist both in the research and industry arenas. Here, we focus only on tools supporting both C and C++ programming languages, which can scale to real-world systems of millions of LOC. C++ programs are particularly interesting for visualization, as they have a deeper hierarchical structure (folders, files, namespaces, classes, nested classes, methods), whereas C program hierarchical structure is limited to folders, files, and functions. Moreover, object-oriented code is supposed to be more modular than classical procedural code, so a good visualization may be able to emphasize the presence (or absence) of such modularity.

Two main classes of C++ fact extractors exist. *Lightweight* extractors, *e.g.* SRCML [Collard et al. 2003], SNIFF+, GCCXML, and MCC, do only partial parsing and type-checking and produce only a fraction of the entire static information. *Heavyweight* extractors, *e.g.* DMS [Baxter et al. 2004], ASF+SDF [van den Brand et al. 1997], CPPX [Lin et al. 2003], ROSE [Panas et al. 2007], OINK [McPeak 2006; McPeak], COLUMBUS [Ferenc et al. 2004], and SOLIDFX [SolidSource BV 2008] perform (nearly) full parsing and type checking. For call data extraction from C and specifically C++, a heavyweight extractor is mandatory, as we need full semantic (type) information, as well as a full implementation of the C++ lookup rules, to be able to correctly link calls to function declarations and those further to function definitions, for all types of functions including constructors, destructors, and operators. Heavyweight extractors can be further classified into strict ones, based on a compiler parser which halts on lexical or syntax errors, *e.g.* CPPX; and tolerant ones, based on fuzzy parsing or Generalized Left-Reduce (GLR) grammars, *e.g.* COLUMBUS, OINK or SOLIDFX. Such extractors are typically run in batch mode. Their output is examined with text-based tools, or, more rarely, visualization tools.

Dependency and call graph visualization is a well-known research area. For very large systems, visualizing only call relations is of limited use, as these have to be correlated with the system structure. As such, many visualization methods combine *call* and *hierarchy* information into so-called compound digraphs. Here, hierarchy describes the function containment in a tree-like structure modeling the system decomposition.

Currently several methods exist to visualize containment and association relations together [Neumann et al. 2005]. SHriMP views and similar methods show containment as nested boxes and associations using the classical node-link model atop of the nesting [Storey and Müller 1995; Bertault and Miller 1999; Raitner 2004]. Variations hereof are well known and used in SoftVis as shown by several toolsets, *e.g.* Rigi [RIGI 2008], CodeCrawler [Lanza 2004], VCG [Lemke and Sander 1994] and SoftVision [Telea 2004]. Although intuitive, such methods have scalability limitations. For large systems, association relations tend to clutter the nested layout, as any two elements in the hierarchy can be connected. ArcTrees draw containment as nested rectangles and associations as curved arcs connecting the elements [Neumann et al. 2005]. However, they have similar association edge cluttering problems as SHriMP views. Curved edges showing associations can also be overlaid on treemaps [Fekete et al. 2003], having however the same cluttering issues. Matrix views remove the clutter by showing associations as an adjacency matrix and hierarchy as tree views or icicle plots along the matrix edges [van Ham 2003]. However, matrix views are less intuitive than node-link diagrams and also are less effective in visually showing modularity, *i.e.* if associations (calls) from a subsystem are mainly directed at a few other subsystems [Ghoniem et al. 2004; van Ham 2003].

For very large graphs, optimizations of both the layout algorithms and graph data management are essential to usability. One system providing these is the graph visualization framework Tulip, which offers a wide range of search, layout, visualization, and interaction features, as well as high scalability for graphs of hundreds of thousands of elements [Auber 2009]. Although less known in the SoftVis community, Tulip is well-known in the InfoVis community, has a development of over 8 years, a large user base, and is arguably one of the most sophisticated graph visualization frameworks available.

Hierarchical edge bundles (HEBs) are a recent advance in displaying large compound digraphs [Holten 2006; Cornelissen et al. 2007]. Containment is compactly shown as a circular icicle plot. Associations are drawn as splines, routed to follow the containment hierarchy. When the analyzed system exhibits modularity in the sense mentioned above, the drawn edges get 'bundled' together making it possible to see this modularity. Visual edge clutter is interpreted as a sign of limited modularity. HEBs have been used in visualizing call graphs in various applications [Cornelissen et al. 2007]. However, as the authors mention themselves, a study on the effectiveness of this method for large-scale software systems, as compared to other dependency visualizations, is still to be done [Holten 2006]. This is one of the aims of the current paper.

3 Call Data Extraction

As mentioned in Sec. 2, we need a call graph extractor able to accurately detect the various types of function calls occurring in C and C++ source code bases, and is also scalable for real-world systems. After analyzing the available options, we choose to build such a tool atop of the OINK static analysis framework. OINK includes a full-fledged C/C++ parser based on GLR technology. Parsing produces possibly ambiguous abstract syntax trees (ASTs), which are next disambiguated and merged by a semantic analysis pass in a single annotated syntax graph (ASG). The semantic analyzer implements the full C/C++ lookup rules, operator overloading disambiguation, type conversions, and all other operations that determine the type of a symbol and associate it with its declaration. The output of OINK is an ASG of the program, *i.e.* an AST annotated with type information. A major advantage of OINK is that it is open source, and also has a fine-grained API to investigate the produced ASG. This allowed us to implement a call graph extractor atop of the basic static analyzer with relative ease, as follows.

3.1 Location of calls and definitions

The first step is to locate the constructs denoting function calls. This is easy, as OINK provides a visitor by which we can find all AST node types denoting function calls. These are 'classical' function calls, constructors, destructors, and operators (including conversions and `new` operators). OINK will also provide implicit function calls that do not appear as explicit syntax in the program, *e.g.* calls of base class constructors in derived class constructors and calls of destructors of local stack objects when a scope is exited. Such information has equal importance to 'classical' function calls in refactoring analyses.

Second, we locate all function definitions, *i.e.* functions having a body. This is equally easy using the AST visitor of OINK. As for calls, all types of function definitions are located, including inline functions and template functions.

Third, for each function call and function definition, we locate its *declaration*. For function definitions, this is trivial, as a definition is its own declaration. For function calls, the type information in the ASG output by the semantic analysis provides us with the unique declaration of the called function within its translation unit.

3.2 Linking

C/C++ programs consist of multiple translation units assembled by a linker which links function declarations from a unit with the corresponding (unique) definitions provided by another unit. We implemented this step atop of OINK as follows. For each translation unit, we save the definitions and declarations of all externally visible functions (*i.e.*, functions that do not have static linkage) in a temporary file. Next, we scan all declarations without definitions in all such files and match them to definitions. This step is massively simplified as OINK provides APIs to check that two function signatures match, according to the full specification of the C/C++ languages. Function declarations for which no definition is found, *e.g.* because they are implemented in binary system libraries, are left unmatched.

3.3 Special cases

The output of the linking step is a program-wide call graph whose nodes are function calls and function definitions (or declarations, for functions having no definitions) and whose edges are the calls. However, some complications exist. In C/C++, functions can be called via pointers, and C++ has virtual functions. In such cases, the previously described method would only find the declarations of the called functions, but not their definitions. We can provide more specific information, as follows. For functions called via pointers, we construct a set of candidates over the entire program which could be the targets of the respective call. This involves all function definitions whose signature matches the call and which do not have static linkage. For virtual C++ functions, we do the same, this time considering all public virtual methods declared in each class hierarchy. This yields a *conservative* set of candidates for each call via pointers or virtual functions. Although such candidate sets may seem to be overly large, they are quite small in practice (5..15 functions), as signature variability and static linkage limit the number of potential candidates. It is possible to further restrict this set by using more sophisticated data flow analyses. The OINK framework provides APIs that could be used to implement this, albeit with more effort.

3.4 Hierarchy

Apart from function calls, we also extract a program hierarchy. This contains nodes that describe the containment of function definitions (or declarations when no matching definitions were found), and has several levels: directories, files, namespaces, classes, and methods, as well as 'free' (file-scope) global functions. Constructing this hierarchy is easy, since OINK provides for each AST node its exact source code location.

Overall, the output of our entire analysis is a program-wide compound digraph containing calls and containment relations. Besides this, we also save data attributes for each node, *e.g.* its name, function details (method, signature, location in the code, access specifiers) and call details (static, virtual, by pointer, and whether the call is exact or determined via our conservative analysis outlined above). Producing such a graph from a given code base is easy: one can simply run an existing makefile, substituting the compiler's name with our extractor, with no further changes. The resulting graph is the input for the visualizations described next.

4 Methodology

To compare the two types of visualizations we target, *i.e.* node-link diagrams (NLD) and hierarchical edge bundles (HEB), we proceeded as follows. First, we extracted several call graphs from increasingly large systems, as described in Sec. 3, among which we mention the *bison* parser generator, the OINK C/C++ static analysis framework, and the *Mozilla Firefox* browser. Next, five developers with no prior knowledge on the analyzed systems were introduced to the NLD and HEB visualization methods to be used, and were given the opportunity to use these systems for a few days, on small datasets, until they were comfortable with their operation. Next, the developers used the NLD and HEB visualizations to answer a number of generic questions on the analyzed systems, *e.g.*: which are the main components in the system; how these components communicate with each other; whether the system is highly modular or not; where is dead code (uncalled functions); and how is the use of polymorphic interfaces (*i.e.* function pointers and virtual functions) spread over the system. Next, several specific questions were asked, *e.g.*: which interfaces (*i.e.* sets of functions declared in the same component) does a specific component call, or provide; and where is a given interface used. The answers, as well as additional comments and remarks on the operations performed to achieve the answers, and the ease-of-use of the respective visualizations, were recorded. A sixth person with detailed knowledge on the analyzed systems performed the study separately and also checked the answers of the other five. Finally, conclusions were drawn using the analyzed answers.

5 Case study 1: The *bison* parser

5.1 Node-link visualizations

The first type of visualization we analyzed is the classical node-link visualization. Nodes are function definitions or containers (directories, files, classes) and edges show calls. For visualization, we used the Tulip framework for several reasons. First, Tulip provides a wide range of functions including many node-link layouts, search and select functions, interactive navigation, and visual customization of colors, shapes, textures, and labels. Second, Tulip is highly memory and speed optimized for very large graphs. Last but not least, all operations are directly accessible via a well-documented user interface (menus, dialogs), making it usable with zero programming effort. This is essential for us, as we assume our users are programmers who want to quickly investigate a large call graph, and have no time or experience to develop their own visualization code.

Figure 1 shows several snapshots produced using the NLD layouts of Tulip on the *bison* call graph (868 functions, 5535 calls). From the recorded procedure, we saw that all users first aimed at obtaining a simple hierarchy view, the reason being to get an idea of the system size, number of layers, and which are the largest subsystems. Images (a) and (b) in Fig. 1 show the two layouts which were found best for this task: the bubble tree layout, which arranges child subtrees in a circle around their parent node [Grivet et al. 2004] and the classical directed tree layout. For this and the other systems analyzed, the bubble tree layout was found easier to comprehend, as it yields layouts with good aspect ratios, and also lets one compare the relative sizes of subsystems quite easily (circle size versus length of a row of nodes in the tree layout).

The next step was to bring the calls in the picture. For this, the first attempt was to add them to the existing hierarchy visualizations. Figure 1 c shows the complete compound graph with the bubble tree layout. Calls are drawn as thin yellow (light) lines, containments are drawn as thick (dark) black lines [1]. Node colors and shapes show their type: directories (blue, squares), files (green, squares), and functions (red, circles). As suspected upfront, the result is quite cluttered. At this scale, the only conclusion that was drawn is that the system is quite tightly connected; its three main

[1] We recommend viewing this paper in full color.

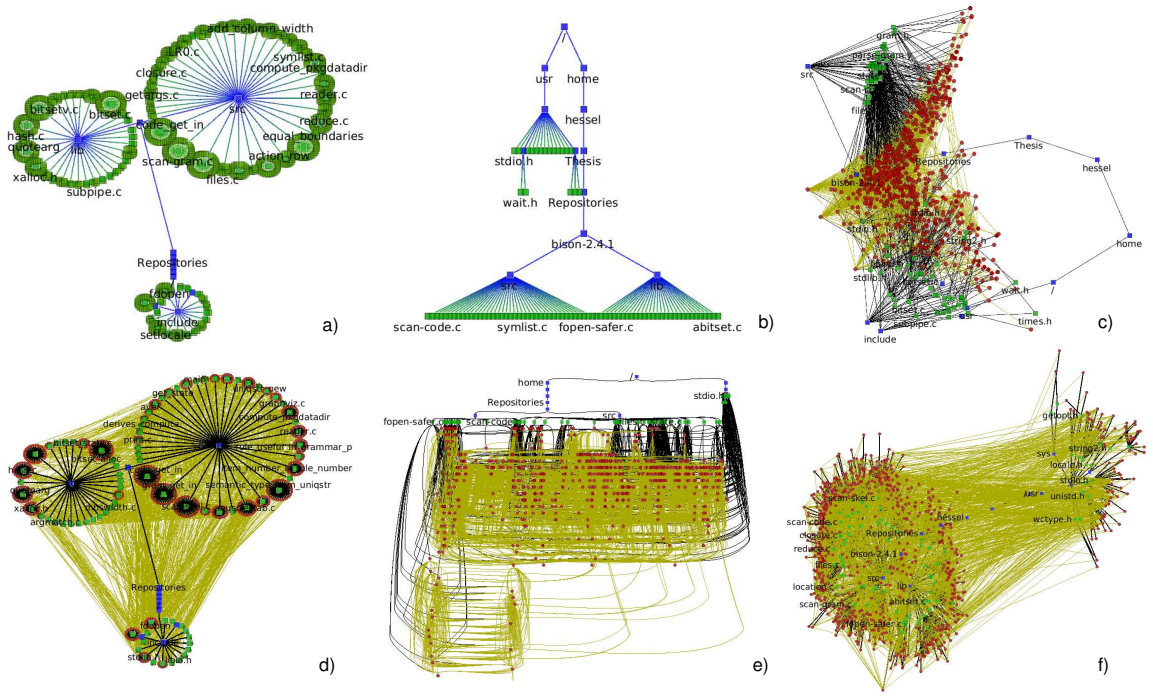

Figure 1: Visualizations of the *bison* call graph using Tulip: hierarchy only using bubble trees (a) and directed trees (b); hierarchy and calls using bubble trees (c) and dendrograms (d); force-directed layouts of hierarchy and calls using HDE embedder (e) and GEM (f)

subsystems `lib`, `src` and `include`, *i.e.* the top-left, top-right, and bottom large circles respectively, are all strongly interacting. Another observation achieved with this view is that functions are not uniformly spread over files: some green squares are surrounded by many red circles. These are files containing many functions, whereas others have only one or a few such circles. These are files containing few used functions, *e.g.* the `include` subsystem.

Alternative types of tree layouts provided by Tulip were explored to show both hierarchy and calls, as well as various layout parameter settings. Most of them did not produce useful results, due to the high clutter caused by the call edges. For example, Fig. 1 d shows a dendrogram layout overlaid with call edges drawn as splines. It might be argued that this layout is useful in comparing call depths between different subsystems, by looking at the height of the red dot sequences (functions) in the lower part of the image. However, showing the actual call edges is not useful, as they produce just clutter.

Further, several force-directed layouts were tried out. Figures 1 e and f show the compound graph drawn using the HDE embedder [Harel and Horen 2002] and GEM [Frick et al. 1994] layouts of Tulip. These are optimized versions of the original publications, which add several heuristics and speed enhancements to deliver higher quality in less computational time (see [Auber 2009] for details). The HEB layout is able to pull the hierarchy nodes (directories and files, shown in blue, respectively green) apart from the function nodes (shown in red, in the middle). For example, we see the files in the `src` directory being isolated in the upper-left part of the image. However, the function nodes, strongly connected by many calls, form a cluster in the middle which is not understandable. Figure 1 f shows a layout using an enhanced version of the well-known GEM spring embedder. This layout is able to pull apart the `include` subsystem, which contains system functions used by the *bison* core, but cannot separate well the *lib* and *src* subsystems, as these are tightly coupled.

Overall, the bubble layout was considered to be the best for the generic comprehension tasks, as it exhibits a stable, regular node placement pattern. For the specific comprehension tasks (see Sec. 4), the built-in search-by-attribute-value and path highlighting functions of Tulip were used. Although these functions are easily accessible via Tulip's GUI, the high visual clutter caused by the dense call pattern in *bison* severely limits the effectiveness of the node-link visualizations. Here again, the bubble tree performed best. The reason seems to be the fact that this layout strongly emphasizes the hierarchy, which is used as a visual guide when analyzing specific call relations.

5.2 Hierarchical edge bundling visualizations

For the second type of visualization, we used SOLIDSX, our own implementation of the HEB method with several enhancements, described next[2]. The design of SOLIDSX is minimalist: all operations are available within a single interface, the main visualization. There are no extra buttons or menus except pop-ups. All operations are accessible with the smallest number of mouse clicks possible. The main HEB idea is simple: hierarchy is drawn as a set of concentric rings divided in sectors, each sector being the container of inner ring sectors corresponding to it; calls are drawn as splines between their corresponding ring sectors; splines are further bundled according to the containment hierarchy, as described in [Holten 2006].

Figure 2 a is an overview of the same *bison* call graph. Several points were made when comparing this image with the NLD layouts in Fig. 1. Showing containment as concentring rings was very easy to understand. The fact that node labels are, at least on the larger rings, readable was seen as a great advantage compared to the NLD label display. Although great effort was done in Tulip to eliminate label overlaps and provide an automatic level-of-detail control of the label size, this was not seen as highly effective. Labels still overlap call edges, and the level-of-detail feature makes labels pop

[2] SOLIDSX is available for academic or commercial users from `www.solidsourceit.com/products`

in and out the view depending on the zoom level in a disturbing way.

We enhanced the concentric ring design to display attributes. Each node in SOLIDSX's input graph can have any number of data attributes, stored as (name,value) pairs, the values being string, numerical, or boolean. We map these values to node colors. A pop-up widget displays all different attribute names present in the input (Fig. 2 a top-right). Attributes can be sorted by name or number of different values they take in the input data. Simply moving the mouse over the listed attributes (brushing) changes the colormapped attribute. For numerical and boolean attributes, we use a simple blue-to-red colormapping based on range. Strings are mapped based on alphabetical ordering. Overall, one can see which attributes are available, and quickly change the one shown to compare different attributes over the same dataset, with one single mouse click and mouse stroke. An identical mechanism is provided for edge attributes, which are mapped to edge colors.

Looking at the overview of *bison* in Fig. 2 a, we quickly locate the main interactions between the three subsystems: `src-lib` (1), `src-include` (2), and `lib-include` (3). Due to the bundling effect, visual clutter is much smaller than in the NLD visualizations in Fig. 1. It was easy to find the many functions which do not get called: these are the innermost circle segments which have no edges connected to them. Doing this task in the NLD visualizations was only possible using Tulip's search functions, but not the images.

Assessing the usage of 'polymorphic' interfaces[3] was easy, by coloring edges based on call type. In Fig. 2 a, static function calls are red, and pointer calls are blue. Clearly, the pointer calls are a minority. Many such calls exist in the `bitset.h` file (below in the image). We added to SOLIDSX the ability to show only calls related to a given node, by clicking on that node. When clicking on `bitset.h`, the file and all its contained functions are displayed in black outline (Figure 2 b). We see that this file has many blue edges going to itself, two blue edges going to `bitset.c` to its left, and a few red edges going to various other parts of the system. This is interpreted as follows: `bitset.h` provides many function declarations, which have equivalent signatures (the loop-like edges atop `bitset.h`); these are only called via pointers; there are only few clients who call such function pointers (the red edges going outwards from `bitset.h`); and there are only two function definitions, in `bitset.c`, which can implement these interfaces. This is precisely the type of information stored in the candidate sets of the pointer-call analysis (Sec. 3).

Adding color to the function definitions in SOLIDSX brings additional insight. In Fig. 2, we show the static linkage attribute of a function. Green indicates `static` functions, while blue shows functions visible by a linker. Interestingly, all function declarations in `bitset.h` are static. Hence, access to these 'polymorphic' features of *bison* can only be done via pointers to them.

5.3 Case Study 2: Mozilla Firefox

In this second, example, we analyzed the Mozilla Firefox code base. Given space limitations, we will only discuss two plugins of the entire system. Figure 3 a,b show the entire call graphs of the *libgklayout* plugin (11817 functions, 21167 edges), visualized using SOLIDSX and Tulip's GEM layout. Directory nodes are drawn blue, files are yellow, classes are green, and functions are blue. Static calls (edges) are red, virtual calls are cyan. At this scale, the GEM layout is clearly not able to disentangle the calls. However, the HEB layout is reasonably easy to read, due to the edge bundling and edge aggregation. For example, we see that almost all virtual calls are directed at a few functions in a single file, *nsCOMPtr.h*, outlined in black in the upper-left of Fig. 3 a. The virtual calls are only visible as a blue spot in the GEM layout (Fig. 3 b).

This figure illustrates also a further enhancement we added to the basic HEB idea. In SOLIDSX, we allow users to show or hide entire hierarchy layers by simple mouse operations. Hidden layers, usually the top-level ones in the system, are drawn as very thin outer rings, as opposed to the regular visible layers, which are thick. Hiding layers saves screen space for the inner layers in deep hierarchies. Still, one can see the color of the hidden (thinly drawn) layers, and count them, thereby getting a cue of how deep one is in the hierarchy. The width of the hidden rings, regular rings, and leaf-node (functions) ring can be also controlled explicitly by the user, if desired. A zoom-in on a small sector of Fig. 3 a is shown in Fig. 4. We see here 10 hidden hierarchy layers which take up only the space needed by a single layer in the big picture.

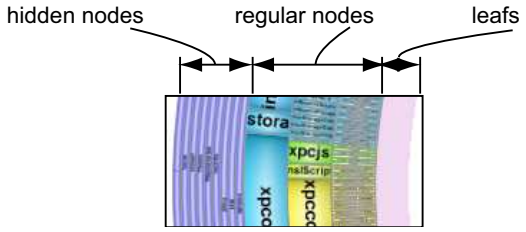

Figure 4: Zoom-in on Fig. 3 a illustrating the hierarchy hiding

Figure 3 c,d shows a much smaller plugin, *libembed* (677 nodes, 936 edges). At this scale, both the NLD and HEB layouts perform comparatively. The users detected here quite easily, in both images, that this plugin contains only a single virtual function (marked by a circle and arrow in the images), called 7 times. This figure shows yet another enhancement of the original circular layout: Instead of rendering all nodes on the same level as contiguous segments on the same circle, we leave gaps between neighbor segments which correspond to nodes which do not have the same parent. In other words, contiguous circle segments indicate siblings, and gaps separate subtrees. This view helps emphasizing the software's hierarchical tree structure, at the expense of a small space trade-off.

5.4 Case Study 3: The OINK Framework

In this third and last example, we analyzed the OINK C/C++ static analysis framework itself. OINK contains around 350 KLOC written mainly in C++, with small parts in C, developed over 6 years by a team of 10 people. The architecture of OINK is quite elaborate. It consists of a lexer (implemented using *flex*), a GLR parser (implemented using the *elkhound* parser-generator library—[McPeak]), an *ast* class library for the over 180 C/C++ grammar nodes, and a semantic analyzer (*elsa*). Our expert programmer, who worked for over 2 years on OINK development, stated that the lexer, parser generator, and AST class library are rather modular and reusable subsystems, in line with the intentions of the OINK developers to make these reusable for a family of languages; however, the semantic analyzer is a much more complex subsystem, with tight couplings throughout the entire system. The question was if this kind of insight could be obtained by the other users using only dependency visualizations.

The OINK call graph, extracted as described in Sec. 3, has 23497 function definitions, 242132 calls, and 2060371 attribute values. This is *two* orders of magnitude larger than all systems visualized so far with the HEB method [Cornelissen et al. 2007]. At this scale, all NLD layouts in Tulip break down - some only produce fully cluttered images, some abort with no result. Since showing all these

[3] By this, we mean C functions called via pointers in *bison*

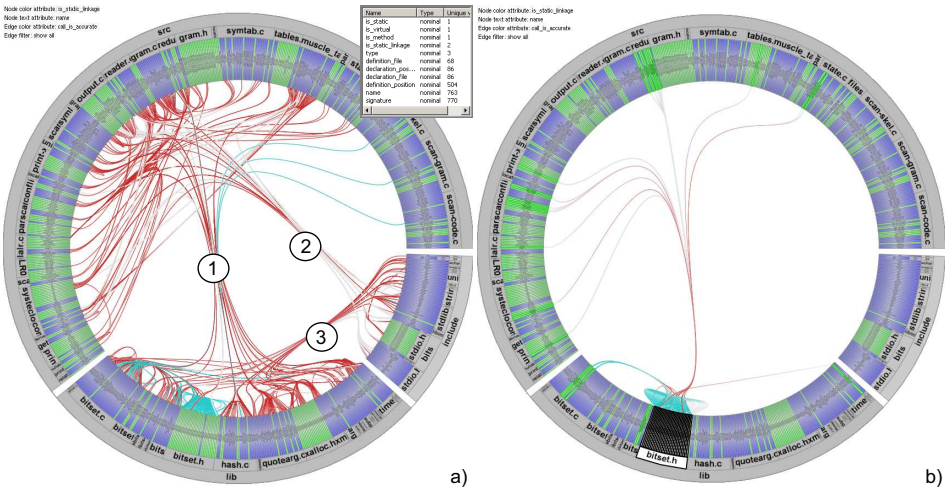

Figure 2: Visualizations of the *bison* call graph using SOLIDSX: entire system (a); selected subsystem with most function-pointer calls (b)

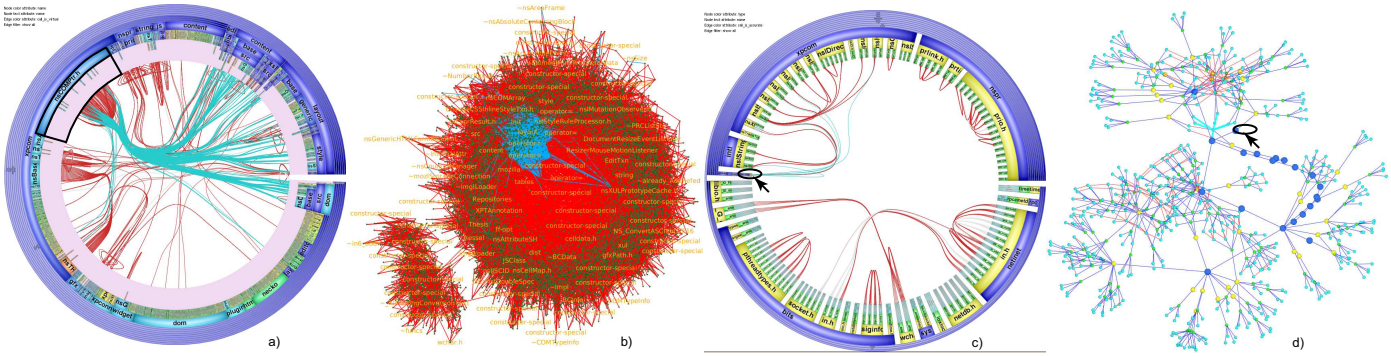

Figure 3: Call graphs of Mozilla plugins: *libgklayout* (a,b) and *libembed* (c,d). Color emphasizes virtual calls.

calls at once may be sometimes too much information even for the HEB layout, we added support in SOLIDSX to allow to navigate the input graph by hierarchy layers. Clicking on nodes allows expanding or collapsing a node. Collapsed nodes aggregate all their calls from/to outside nodes n_i and display a single thick edge per node n_i. If the attributes of all aggregate edges have the same value, then this value is used to color the edge, else the edge is colored gray. Figures 5 a-c show the call dependencies of the entire OINK system at file level (a), class level (b), and method level (c). Only three clicks are needed to produce these three views - each click further expands a deeper hierarchy level. Directory nodes are drawn blue, files are yellow, classes are green, and functions are blue.

In Fig. 5 a, the *ast* and *elhound* systems, visible by their white background and thick black borders, were selected by clicking as explained in Sec. 5.2. We see that these systems have few calls from the analyzer's core, *elsa*. This suggests a potentially good separation of these three subsystems. The next image, Fig. 5 b, shows the entire system one level deeper, *i.e.* at class level. The innermost ring is predominantly blue, which means function definitions (blue) are directly contained in implementation files. The few green spots denote private implementation classes, which are thus only sparsely used in this system. The next image, Fig. 5 c zooms one level deeper, showing all functions in the system. As the number of functions in this case is far larger than the amount of available pixels, we chose a simple solution: we render only the nodes involved in the relations with the selected nodes, and render the remainder of the nodes in gray (see Fig. 4 for a detail zoom-in corresponding to the image in Fig. 3 a). These are shown in green on the inner circle in Fig. 5 c. Here, we see that, although the two selected subsystems *ast* and *elkhound* have very strong internal cohesion (many self edges), their communication with the system's core (*elsa*) is indeed quite limited. This is a good sign for modularity.

Figure 5 c also shows the relative sizes of OINK's components. Files containing many functions occupy a larger part of the circular layout. We see that these are the files of the semantic analyzer: the scoping environment (`cc_scope.cc`), the template analysis code (`template.cc`), the type system classes (`variable`, `cc_type.cc`). To analyze how modular the semantic analyzer is, we select its components by clicking (see Fig. 5 d-f). We now see not only that these are large, but also have much more connections with large parts of the entire system than the *ast* and *elkhound* subsystems - compare the amount of green segments on the innermost ring and number of edges in Figs. 5 d-f with those in Fig. 5 c. This finding correlates with the expert programmer's experience: OINK is modular with respect to the *ast* and *elkhound* parser generator, but its semantic analyzer is over half of its code, and a tightly coupled one for that matter.

Finally, to assess the polymorphism of the OINK code base, we use again edge coloring. In Fig. 5, red denotes static function calls, and blue denotes virtual calls. We see relatively few virtual calls - this is in line with the OINK design documentation, which stresses a minimal use of virtual methods for optimal performance.

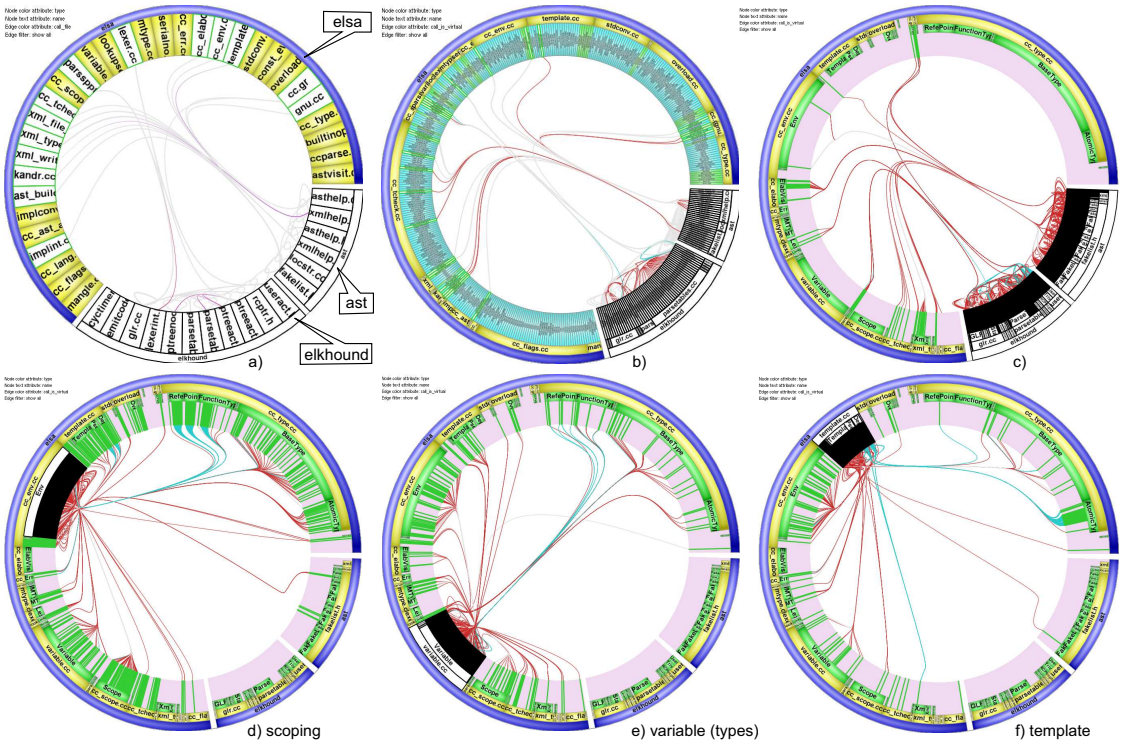

Figure 5: OINK framework: multilevel visualization of calls on the level of files (a), classes (b), and functions (c); Main semantic analysis subsystems: the scoping environment (d), variables (e), and template analysis code (f). Selected subsystems are shown in black

5.5 Discussion

5.5.1 Usability comparison

We distilled several points from the reports provided by the five users in this study. All users strongly agreed that the HEB layout is overall vastly superior to node-link diagrams (NLDs) for navigating call-and-hierarchy graphs larger than a few hundred nodes, for virtually all considered tasks, because:

1. HEBs show more data on the same amount of screen space
2. edges in HEBs are much less cluttered
3. hiding/showing nodes changes HEB layouts less than NLDs
4. the circular layout draws parent nodes naturally larger
5. HEBs show more node labels with less clutter than NLDs
6. interaction in HEB is always near-real-time, while some NLDs take long to compute

However, some advantages of NLDs were mentioned too:

1. NLDs allow more freedom in manual layout editing
2. NLDs make it easier to follow a path than the HEB
3. the HEB circular layout places sometimes unrelated nodes close to each other

For our tasks of interest, the advantages of HEB compensated the advantages of NLDs. Although not rigorously timed, we noticed users of HEBs being 3..5 times faster in accomplishing the same task than when using NLDs, the average task in HEB being 1..3 minutes. The search and select functions of both tools used are comparable in effectiveness and simplicity, so the difference can be attributed to the visualization part. For instance, obtaining a view as Figs. 2 or 5 takes around 1.5 minutes and around 10-15 mouse clicks, including loading the data. Obtaining a similar image in Tulip takes around 5 minutes and a few tens of clicks and selections. In both cases, we used no custom application presets.

5.5.2 Performance comparison

Both Tulip and SOLIDSX are highly engineered for performance, which is important for graphs of hundreds of thousands of elements and attributes. For example, the OINK dataset (Sec. 5.4) takes 178 MB to store in Tulip and 395 MB in SOLIDSX. The difference is explained by Tulip's special memory management which uses custom bit-level allocation to limit memory waste [Auber 2003]. All Tulip tree-like layouts are of comparable, near-real-time, performance as the HEB layout. The HDE embedder and GEM are considerably slower, taking *e.g.* about 2 minutes on the relatively small *bison* dataset (Sec. 5.1) on a 2.8 GHz PC.

5.5.3 Threats to validity

For our comparison of visualization methods for call-and-hierarchy data, the following points are important. First, we only compared a limited number of NLD layouts with the HEB layout. Other layouts, *e.g.* SHriMP-like ones, could perform better than those studied here. There are, however, reasons to believe the opposite. SHriMP-like layouts are effective in showing containment, but they do not scale well in number of associations. These tend to occlude the containment drawing, and also are hard to distinguish among themselves (see *e.g.* [Lanza and Marinescu 2006; Telea 2004]). Such layouts are effective for top-level architecture-like views, but not for massive call graphs. However, we could not test all possible NLD layouts in existence. The choice for Tulip was explicitly done from an end-user perspective: choose a scalable, documented, user-friendly, highly optimized NLD visualization tool, compare it with a HEB implementation sharing the same features, and see which one is better accepted by users.

5.6 Availability

The entire toolset, including the C/C+ call-and-hierarchy extractor, the SOLIDSX visualization tool, and the extracted call graphs in Tulip and SQL formats, are available from the authors upon request. Additional components to our toolset, not discussed here, include plug-ins to automatically extract dependencies, syntactic information, and metrics from Visual C++ projects and .NET assemblies.

6 Conclusions

We have presented a study that compares the usage of node-link diagram (NLDs) and hierarchical edge bundle (HEB) layouts for the visualization of large call-and-hierarchy graphs of software systems. To perform this, we have constructed a fully automatic pipeline for extracting call graphs from C/C++ programs, including a call static analyzer, and an enhanced implementation of the HEB method for navigating very large graphs. The study points out an important advantages of the enhanced HEB method for typical comprehension tasks involving call-and-hierarchy data, and demonstrates the applicability of such methods for the understanding of large, real-world, programs.

We are currently working on extending our call-and-hierarchy visualization with additional views to support investigation of additional graphs, *e.g.* class hierarchies, usage of types, and data flow, as well as visualizing multiple attributes in a single view, *e.g.* static type information, type matching, and source code metrics. It is also interesting to study how some of the perceived advantages of NLD layouts could be merged with the HEB views to obtain a visualization that combines the benefits of both methods.

References

AUBER, D., 2003. Visualization of large graphs in the Tulip system. PhD thesis, U. of Bordeaux, France.

AUBER, D. 2009. The Tulip graph visualization framework. `www.tulip.org`.

BAXTER, I., PIDGEON, C., AND MEHLICH, M. 2004. DMS: Program transformations for practical scalable software evolution. In *Proc. ICSE*, 625–634.

BERTAULT, F., AND MILLER, M. 1999. An algorithm for drawing compound graphs. In *Proc. Graph Drawing*, 197204.

COLLARD, M. L., KAGDI, H. H., AND MALETIC, J. I. 2003. An XML-based lightweight C++ fact extractor. In *Proc. IWPC*, IEEE Press, 134–143.

CORNELISSEN, B., HOLTEN, D., ZAIDMAN, A., MOONEN, L., VAN WIJK, J., AND VAN DEURSEN, A. 2007. Understanding execution traces using massive sequence and circular bundle views. In *Proc. ICPC*, IEEE, 49–58.

FEKETE, J. D., WANG, D., DANG, N., ARIS, A., AND PLAISANT, C. 2003. Overlaying graph links on treemaps. In *Proc. InfoVis (poster)*, 8283.

FERENC, R., SIKET, I., , AND GYIMÓTHY, T. 2004. Extracting facts from open source software. In *Proc. ICSM*.

FRICK, A., LUDWIG, A., AND MEHLDAU, H. 1994. A fast adaptive layout algorithm for undirected graphs. In *Proc. DIMACS'94*, Springer LNCS, 388–403.

GHONIEM, M., FEKETE, J. D., AND CASTAGLIOLA, P. 2004. A comparison of the readability of graphs using node-link and matrix-based representations. In *Proc. InfoVis*, IEEE, 1724.

GRIVET, S., AUBER, D., AND MELANCON, G. 2004. Proc. intl. conf. on comp. vision and graphics. 633–641.

HAREL, D., AND HOREN, Y. 2002. Graph drawing by multidimensional embedding. In *Proc. Graph Drawing*, 388–393.

HOLTEN, D. 2006. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. In *Proc. InfoVis*, 741–748.

KOSCHKE, R. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance: Research and Practice 15*, 2, 87–109.

LANZA, M., AND MARINESCU, R. 2006. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

LANZA, M. 2004. CodeCrawler - polymetric views in action. In *Proc. ASE*, 394–395.

LEMKE, I., AND SANDER, G., 1994. VCG: visualization of compiler graphs. Tech. Report, Univ. des Saarlandes, Saarbrücken, Germany.

LIN, Y., HOLT, R. C., AND MALTON, A. J. 2003. Completeness of a fact extractor. In *Proc. WCRE*, 196–204.

MARCUS, A., FENG, L., AND MALETIC, J. 2003. 3D representations for software visualization. In *Proc. ACM SoftVis*, 2736.

MCPEAK, S. Elkhound: A fast, practical glr parser generator. Computer Science Division, Univ. of California, Berkeley. Tech. report UCB/CSD-2-1214, Dec. 2002.

MCPEAK, S., 2006. The Elsa C++ parser. `www.cs.berkeley.edu/\~smcpeak/elkhound/sources/elsa`.

NEUMANN, P., SCHLECHTWEG, S., AND CARPENDALE, M. S. 2005. ArcTrees: Visualizing relations in hierarchical data. In *Proc. EuroVis*, IEEE, 5360.

OINK, 2008. The oink C++ static analyzer. `www.cubewano.org`.

PANAS, T., QUINLAN, D., AND VUDUC, R. 2007. Tool support for inspecting the code quality of HPC applications. In *Proc. SE-HPC*, 2–12.

RAITNER, M. 2004. Visual navigation of compound graphs. In *Proc. Graph Drawing*, 403413.

RIGI, 2008. Rigi: A visual tool for understanding legacy systems. U. of Victoria, `www.rigi.csc.uvic.ca`.

SOLIDSOURCE BV. 2008. SOLIDFX product information. `www.solidsource.nl/products`.

STOREY, M., AND MÜLLER, H. 1995. Manipulating and documenting software structures using SHriMP views. In *Proc. ICSM*, 275284.

TELEA, A. 2004. An open architecture for visual reverse engineering. In *Managing Corporate Information Systems Evolution and Maintenance (ch. 9)*, Idea Group Inc., 211–227.

VAN DEN BRAND, M., KLINT, P., AND VERHOEF, C. 1997. Reengineering needs generic programming language technology. *ACM SIGPLAN Notices 32*, 2, 54–61.

VAN HAM, F. 2003. Using multilevel call matrices in large software projects. In *Proc. InfoVis*, 227–232.