http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 12 (8): 1522-1530, 2013
ISSN 1812-5638 / DOI: 10.3923/itj.2013.1522.1530

Source Code Visualization in Linux Environment Based on Hierarchica Layout Algorithm

[1]Yi Luo and [2]Yanying Han
[1]Institute of Computer and Communication,
Changsha University of Science and Technology, Changsha 410004, China
[2]Institute of Mathematics and Computing Science,
Changsha University of Science and Technology, Changsha 410004, China

Abstract: Since, the previous source code analysis tools can not reflect the hierarchical system structure of source code in Linux perfectly, we propose an automatic hierarchical layout algorithm for source code in Linux with the emphasis on symmetry. In order to reflect the structure of software according to Linux framework, we construct the call graph from C language source codes and then divide extracted functions into different abstract levels automatically. The call graph similar to a tree is visualized by improving Sugiyama layout constrains and Walker's layout algorithm. In this study a number of problems related to level-crossing and subtrees overlapping are solved to make better visual representation. The experimental results show that this algorithm is appropriate for source code visualization in Linux and can reflect the hierarchical structure and dependencies of functions preferably.

Key words: Software visualization, source code analysis, linux, layout algorithm, hierarchical layout, C language, call graph

INTRODUCTION

Linux, an open-source operating system, is widely used on servers, desktops and embedded systems. With the expansion of system scale, the costs of both forward and reverse software engineering rise dramatically. As a solution, assistive tools such as modeling tools and visualization utilities (Bernardi *et al*., 2012; Wang *et al*., 2003) are adopted to enhance the development efficiency and save costs. In Linux environment, it is very effective to analyze source codes visually for program understanding, embedded system tailoring and reverse engineering. Visual code analysis is usually realized by parsing the source, reflecting the system structure and behavior on a higher abstraction level and representing it by chart or graph (Harman, 2010; De Figueiredo *et al*., 2008).

The source codes in Linux include the kernel source, application programs and library functions. Nowadays the scale of Linux kernel grows increasingly. Meanwhile the applications get varied and the structure of code become complex. But Linux itself is not defined as a hierarchical structure strictly and its module structure is not clearly classified. All of these propose higher requirements on visual analysis tools.

An ideal Linux source code visualization tool should meet the following conditions: Firstly, analysis tools should be able to describe the software structure, dependencies and other important properties. The acquired information can be applied to software maintenance and reverse engineering. Secondly, the visual expression of the source code should be straightforward, readable and understandable. Thirdly, it should be appropriate to analyze all the source codes in each system level extensively.

Currently, popular analysis tools which are suitable for source codes in Linux are characterized into two groups: cross-reference pattern and graphic pattern. LXR (Linux cross-reference) is a typical cross-reference code analyzer for Linux which can reflect functional relationship of the kernel comprehensively (Gleditsch and Gjermshus, 2010). But it has not been visualized. CodeViz is a visualiser which can generate call graph effectively by analyzing functions dynamically. But it is unavailable for uncompiled programs and kernel (Gorman, 2012). There are still many intuitive graphical tools which could be applied to object-oriented programs but not suitable for C language modeling (Davis *et al*., 2003). As a professional layout tool, dot language, which compact the spatial structure effectively, can create exciting visual

Corresponding Author: Yi Luo, Institute of Computer and Communication, Changsha University of Science and Technology, #960, 2nd Section, Wanjiali South Rd, Changsha, Hunan, 410004, China Tel: 86-13723876037

effects. For this advantage, it has been adopted by various code visualiser as layout front-end (Gansner and North, 2000). However, dot is a general-purpose layout instrument without code-analysis back-end, so it cannot reflect the level properties and the calling sequences. If the codes are comprehensive, the readability may be diminished.

According to the above issues, this paper designed a source code parsing tool and visualized source code in Linux by hierarchical layout algorithm. The architectures of source codes are depicted to help users to understand the design concepts of software quickly. Firstly, after analyzing the source code in Linux, we obtained the function dependencies and corresponding properties that can be used during layout processing. Then the functions are divided into different system levels by retrieving the libc-syscall matrix. Finally, considering the structure and characteristics of source codes in Linux, we designed a call graph visualiser by improving Sugiyama's tree-layout constraints and Walker's algorithm to make better visual representation.

SOURCE CODE VISUALIZATION METHODS IN LINUX ENVIRONMENT

For object-oriented languages like Java, the entities needing to be visualized are classes and the relationships between them are mainly inheritance. However, for a C program, the visualized entities are functions and the critical relationships are call relations. Call relations are hidden among the codes and can be represented as call tree. But merely the call relationships cannot represent the architecture of program clearly. We tried to combine them with the Linux system structure to form a clear hierarchy. According to the requirements for source code visualization, the visual procedure contains three steps: source code analysis, system structure layering and layout. Hence, our system includes three major modules: (1) a source code analysis tool which is used to analysis the text of C language code and acquire the corresponding attributes, (2) a system structure layering tool which is used to retrieve the libc-syscall and syscall-kernel matrices and locate function's level associated with function attributes and (3) a layout tool which can draw the hierarchical call graph for the specified function based on its level and attributes. The system flowchart is shown in Fig. 1.

Call relation analysis for source codes: The call relation analysis tool analyses the call relationships between functions. There are two kinds of commonly used analytical methods: dynamic and static. The dynamic

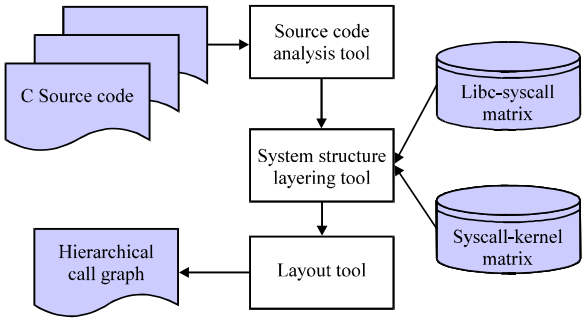


Fig. 1: System flowchart of source code visualization system

methods are usually compiler-dependent and easy to implement, while the code that could be processed must be compiled successfully first (Bohnet *et al*., 2008). So they are not quite suitable for uncompiled applications and the source codes of kernel.

For this reason, static analytical method is adopted in this paper to scan the source code file and find the call relation between functions. At present, there have been various CG(Call Graph) algorithms including NBR(Name-Based Resolution) algorithm, CHA (Class Hierarchy) algorithm, RTA (Rapid Type Analysis) algorithm and so on (Nan, 2006). In Linux, static code scanning tools, such as CallTree, output the CG by NBR algorithm in plain text (Schilling, 2010). This text has not been transferred into hierarchy model, so it can not be used for layout directly.

CG is defined as a directed acyclic graph which represents binary relationship of the selected entities (Gang, 2009). The entities can be programs, functions, modules or files. This paper defined CG H to describe the function relation. Eliminated the recursive calls, H becomes an acyclic graph whose nodes can be ranged according to the sequence that the functions are called.

Definition 1: CG H = (V_G, E, V_D) , V_G is the node set of CG denoting the function set of source codes in Linux. E is the edge set representing the call relations between functions. And V_D is the attribute set of functions representing various kinds of attribute value involved during the analytical procedure.

The core algorithm of call-relation analysis tool is a call parsing algorithm for C language source code. C language program usually starts with a main function, so parsing begins with this function and all the calling points are processed by the order of presence recursively. If there are some nested calls, the nested depth and corresponding file name should be recorded and the subfunction should be located. The functions will be parsed recursively until all the functions are processed.

Algorithm 1: C language parsing method

Step 1: mainFunc $\in V_G$ (normally mainFunc is main function. Users can specify his own mainFuc too.)

Step 2: For each function M in set V_G, if there is a nested function, namely nestFunc, called by M, we let nestFunc$\in V_G$ and M$\neg$nestFunc$\in$E. Meanwhile we set nested depth as nestLevel = nestLevel+1 and deal with the lower level. After all the nested functions are processed, we go back to the upper level. If set V' is the set of functions being called by M, V' also belongs to set V_G, that is $(M \in V_G) \Rightarrow (V' \in V_G)$.

Step 3: Perform step 2 repeatedly, until the end of mainFunc or the nested depth reaches a maximum value.

The principal attributes extracted from source codes include function name, function's file name, subfunction list, nested depth and so on. Taken C code cls.c from software package of Apache as an example, the CG adjacency list is shown in Fig. 2.

System structure layering for functions: The CG we constructed above can represent the call relation of source code in Linux and can be visualized by dot language. But because of the complexity of call relation, the picture draw from this CG directly produces a poor layout without strict system-level hierarchical structure. In order to help users to under stand software structure from the system-level, we divide functions into different abstract level before we start to layout. This improved CG is defined as hierarchical CG H' as follow.

Definition 2: Hierarchical CG H' (V_G, E, V_D, L, C), L is the set of layers representing the layer that the function belongs to. C expressed as C: Source$\neg$T arg et is the association set that corresponds to relations between nodes. Here, Source$\in V_G$, T arg et $\in V_G$ and the relation between them could be ancestor-descendant, father-son or brothers.

In Walker's algorithm, the layers are merely computed by the Inherit sequence of nodes, therefore can not reflect the system-level structure of software perfectly (Walker, 1990; Buchheim *et al*., 2002). To solve this problem, according to their abstract system-level, we divide the functions into four layers: user application level, libc level, syscall level and kernel level. The application program belongs to user level. Libc level contains the functions in the standard function library of C language. And in Linux it particularly refers to glibc which is widely used in Unix-like systems. Glibc contains many APIs (Application Program Interface) which normally correspond to a homonymic system call. System calls associated itself with API via wrapper routine and provide the service that is needed by the application from the kernel. The kernel level belongs to operating system and can not be called by applications directly. Single system call can call multiple kernel functions and sometimes API is not associated with any system calls, so

Fig. 2: CG adjacency list of cls.c

the call relations between functions in four levels are not one-to-one correspondence. The application programs call the kernel function by system call, i.e., the system call service routine.

So, we define the node set of CG as $V_G = V_{app} \cup V_{libc} \cup V_{sys}$, V_{app}, V_{libc} and V_{sys} refers to the function set of applications, libc and system calls, respectively. To set the various function levels for source codes, each function should be located into one of these sets. In a specified Linux system, the CG of libc, system call and kernel, which is a disordered hierarchy graph, is determinate. In Linux, the name and identifier of system calls is saved in a file named unistd.h and the corresponding service-routine address is saved in the table called syscall_table.S. By parsing these files, the relationship between system call and kernel function is obtained. Also by using the C language analysis algorithm we mentioned above to process libc source code, we construct the relationship between system call and libc. After that the function layering tools can query these two relationship tables to locate a specified function to its corresponding set.

In order to embody the system-level of the function, nodes are arranged top to bottom according to the user-libc-SystemCall sequence.

Definition 3: We defined function's level as L = (Depth, LinuxSysLevel). Depth refers to the calling depth of the function node in H. LinuxSysLevel refers to system-level of the function in H'.

With our method the height of function node is set by LinuxSysLevel instead of Depth. The value of LinuxSysLevel is growing as the function node getting closer to the bottom. For arbitrary function node that belongs to set V_G, the LinuxSysLevel can be set by Eq. 1.

$$\text{node.LinuxSysLevel} = \begin{cases} \text{node.Depth} & (\text{node} \in V_{app}) \\ \text{MaxDepth} + 1 & (\text{node} \in V_{libc}) \\ \text{MaxDepth} + 2 & (\text{node} \in \cup V_{sys}) \end{cases} \quad (1)$$

MaxDepth is the maximum depth of nodes. There will be a cross-layer when the LinuxSysLevel of a node is not

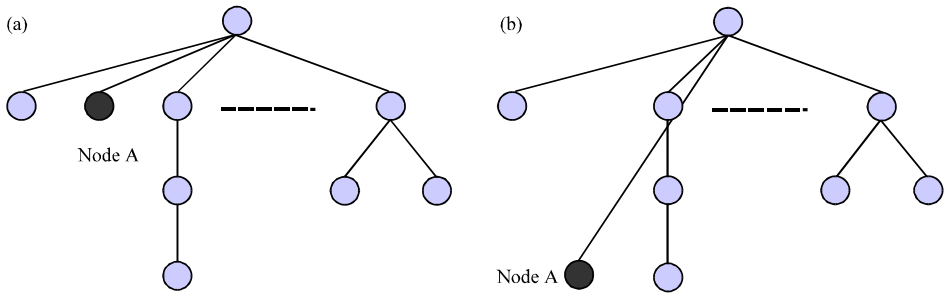


Fig. 3(a-b): Layer crossing issue (a) Call tree set by Depth and (b) Call tree set by LinuxSysLevel

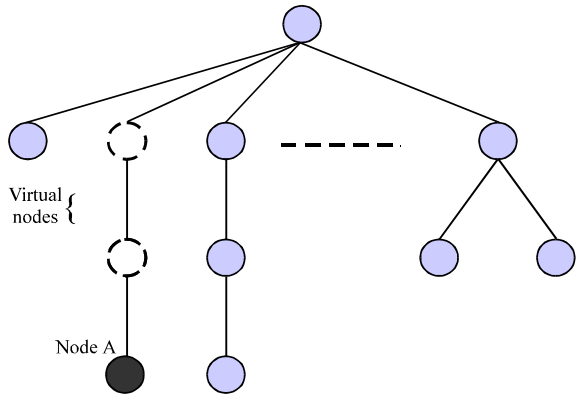


Fig. 4: Call graph after adding virtual nodes

equivalent to its Depth, such as node A shown in Fig. 3. In such a situation, two problems arise. One is that the sequence of function calls shown in the call graph may change. For example the node will move to the right side of its right brother so that the calling sequence will be destroyed. On the other hand, the number of crossing-edge will increase too (Fig. 3).

Therefore virtual node is added in the original position of node to avoid calling sequence changing and more virtual nodes are added in each level between original and updated position to avoid edge-crossing. The call graph after adding virtual nodes is shown in Fig. 4.

Hierarchical call graph is accomplished after all the nodes are layered. The main properties of each function node include function name, storage location, father, children list, calling depth and system-level. Still taken cls.c as an example, the main properties list is shown in Fig. 5.

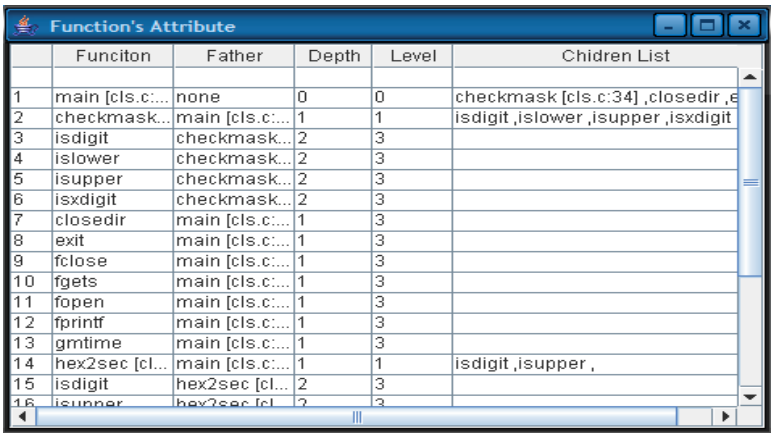
Hierarchical layout for call relations: The hierarchical CG describes the inner logical relationship between functions, but still hasn't been visualized. To visualize the source code, the layout must be generated based on the call graph.

Layout constraints of call graph: The layout of CG should be understandable, readable and capable of reflecting the structure of the function and characteristics. Considering the aesthetics, it should be clear and balance.

Considering the features of source code in Linux and based on Sugiyama's drawing rules (Sugiyama *et al*., 1981), the layout constraints include the following:

- Layout is a descending graph. The function being analyzed is placed at the top. The level of the node can reflect the actual level of function in Linux system
- Nodes should be sorted according to the calling order, which makes it easy for users to understand
- To improve the comprehensibility of figure, subtrees' outlines should not overlap each other
- With a clear hierarchical structure, the figure can represent the calling relations between functions
- Edge crossing are avoided (minimization of edge crossing)
- Parents are placed at the barycenter of their children

There are a variety of layout algorithms that can satisfy the drawing rules 4 to 6 at present. But the



Function's Attribute

	Funciton	Father	Depth	Level	Chidren List
1	main [cls.c:...	none	0	0	checkmask [cls.c:34] ,closedir ,e
2	checkmask...	main [cls.c:...	1	1	isdigit ,islower ,isupper ,isxdigit
3	isdigit	checkmask...	2	3	
4	islower	checkmask...	2	3	
5	isupper	checkmask...	2	3	
6	isxdigit	checkmask...	2	3	
7	closedir	main [cls.c:...	1	3	
8	exit	main [cls.c:...	1	3	
9	fclose	main [cls.c:...	1	3	
10	fgets	main [cls.c:...	1	3	
11	fopen	main [cls.c:...	1	3	
12	fprintf	main [cls.c:...	1	3	
13	gmtime	main [cls.c:...	1	3	
14	hex2sec [cl...	main [cls.c:...	1	1	isdigit ,isupper ,
15	isdigit	hex2sec [cl...	2	3	
16	isupper	hex2sec [cl...	2	3	

Fig. 5: Function attributes table of cls.c

traditional algorithms compute the layer merely by inheritance relationship and cannot reflect the system structure of software in Linux. To compress layout area, subtrees are overlapped along X axis direction. Furthermore he traditional algorithms support (k_2)-partite tree which cannot solve the cross-layer problem. Hence the existing algorithms need to be improved.

Layout algorithm of call graph: The layout algorithm calculates the coordinates of each node based on the calling relation we obtained above. And the edges are drawn after that by a preorder traversal. The layout algorithm is described as follows:

Algorithm 2: Hierarchical layout algorithm

```
Function DrawCallTree();
 Position(); //Calculates the coordinates
 DrawGraph(); //Routing
End
```

The position algorithm, which is improved based on the Walker algorithm, computes x coordinates, while y coordinates are computed by node level (Buchheim *et al*., 2002). To improve the readability of figure rather than try to the compress the layout area as much as possible, the distance between two trees are set in this algorithm.

In order to generate the node's position, PreX is set as the initial value of X coordinate which refers to the relative coordinate in the subtree it belongs to. And ModX is the modification value of X coordinate which refers to the distance that the subtree needs to move from left to right to make the call tree be balance.

To produce the final x coordinate of a node, two tree traversals are used. The first traversals, PostTraversal, is a postorder traversal which is used to calculate the PreX and ModX for each node. The second traversals, PreTraversal, is a preorder traversal which determines the final x coordinate for each node. To separate subtrees, the distance between trees should be given when we calculate the value of PreX. To keep the subtrees from crossing and minimize the distance between their roots, TreeSpace is set as the minimum distance between subtrees.

In the postorder traversal, the nodes' default coordinates in subtree are given. Assume that the distance between the most right son of left subtree and left subtree's root is d_1 and the distance between the most left son of right subtree and right subtree's root is d_2. Normally the space between two subtrees is the sum of d_1 and d_2. But if there is only one son in each layer of these subtrees, the space we've calculated may be smaller than the width of the node itself, namely NodeSize.

Therefore TreeSpcace = Max((d_1+d_2), NodeSize).

The postorder traversal algorithm is described as follows:

Algorithm 3: PostTraversal(node,level)

Step 1: After initialized and then taken the boundary of current layer, we add the current node to form the new boundary. ModX value of the node is set to 0.

Step 2: If the node is a leaf and has left brothers, then we set PreX(node) = PreX(leftSib)+spaceX+TreeSpace. If the node is a leaf and has not any left brothers, then we set PreX(node) = 0. Here, spaceX is the minimum horizontal space between nodes.

Step 3: If the node is not a leaf and not all of the sons have been processed, we take a son and perform step 4 recursively.

Step 4: We take most-left unprocessed son as the current node to perform PostTraversal recursively. After that we calculate the middle point, namely midPoint, of the subtree. If node has left brother, then set Pre(node) = PreX(leftSib)+spaceX+TreeSpace and ModX = Pre (node)-midPoint. If node has not left brother, then PreX(node) = midPoint.

Step 5: End

During a second preorder walk, each node is given an x coordinate. PreTraversal processes the root first and then subtrees from left to right. Finally leafs are processed. All of the subtrees are moved to right according to the modifiers which makes sons arranged

under father balanced. The x coordinate value of node v is given by summing its preliminary x coordinate and the modifiers of all its ancestors. The y coordinate depends on the system level Eq. 2.

$$\begin{aligned} x &= xAdjust + PreX(v) + \sum_{v_i \in ancestor(v)} ModX(v_i) \\ y &= yAdjust + v.level \times spaceY \end{aligned} \quad (2)$$

Here, xAdjust and yAdjust is the adjustment value of coordinates. The ancestor(v) is the ancestors set of node v and spaceY is the longitudinal separation.

Routing: The routing algorithm reads the function-node coordinates from the hashtable and connects the nodes which have been called with their farther. This algorithm judges the validity of the coordinates firstly. And then it sets the size of the layout based on the width and height of call tree. Finally it draws the call graph.

Actual function nodes are connected according to the call relationship directly with node name. When the connection contains virtual nodes, nodes are connected in accordance with the inheritance relationships without virtual nodes' name.

EXPERIMENTAL RESULTS AND ANALYSIS

To verify the validity of our method, we designed software named CallTreeLin by java language which can parse the source codes in Linux and acquire the layout of call graph. CallTreeLin can run on both Windows and Linux platform. This study analyzed a set of programs written in C language in Linux and drew their call graphs as shown in Table 1. The C language parsing tool analyzed 5 kinds of software. Each kind includes 3 specified source codes. The numbers of functions that are parsed see the third column. We can see that number of functions varied with scale of code and the biggest call tree is produced by cum which includes 315 function nodes. And in total we processed 2472 functions nodes.

Graphviz, which is developed by Bell laboratories, adopts general composite layout-algorithm and become the most commonly used tool for call graph layout currently (Gansner and North, 2000). We use dot, the hierarchical layout tool of Graphviz, to compare the automatic layout algorithm we proposed. Because the dot language itself cannot analyze the source codes, we designed a transformation program that could convert the calling relationship between functions into the dot language. Then dot language can produce its own layout.

Readability improvement: Firstly, this paper draws the call graph which can reflect the system level of all functions. Fig. 6 is drawn by dot. It represents the call graph of parse-dir-colors which is function of software named tree. Because dot usually changes the order of function nodes, it cannot reflect the order that each function being called. Dot arranges the nodes in layers in inherit order, so the system level is not very clear. In Fig. 6 we can see that xmalloc calls the same libc function repeatedly. The same call tree drawn by CallTreeLin as Fig. 7. Scopy, sprintf and strlen are subfunctions of parse-dir-colors. Scopy, which is written by user, belongs to application level. Sprintf and strlen belongs to libc level which is lower than scopy. Sprintf called system call write which belongs to the bottom level, that is, system call level. From this picture we can see that with our method the functions are arranged in order that they are called. In application level of the tree structure, there is not any crossing. The applications, libc functions and system calls are well arranged with a clear system structure which makes the call tree readable and understandable.

Secondly, when dot is drawing the call graph, one line represents one call of function. Hence it only indicates the frequency that the function is called rather than the order and position of functions. In the case of functions being called frequently, the graph will lost its readability. This paper draws a new node for each function call to represent the order and position of functions. Meanwhile it solve the problem that there may be too many crossings and raise readability of call graph. The effect comparison of both methods is shown as Fig. 8.

Testing all the functions in Table 1, the experimental results show that the number of node and edge is very close in our method (Fig. 9). That is because we arranged the nodes in a specified order and tried to reduce the

Table 1: Software system studied

Software type	Software name	Number of functions
Developing tool	Busybox, gcc, tree	10+22+24=56
Multimedia	Paino, Mp3fs -0.32, Mpd-0.17	458+13+135=606
Image processing	Ristretto-0.1.1, cum-20110726, gif2png	39+307+306=652
Video display	Gxine_0.5905, notmuch-0.13, radeontool-1.5	146+315+96=557
Text processing	Abcm2ps-7.0.16, difffilter-0.3.3, gummi-0.6.5	135+44+47=226
Network	Apache, vsftpd-3.0.2,openssl-1.0.1c	214+51+110=375

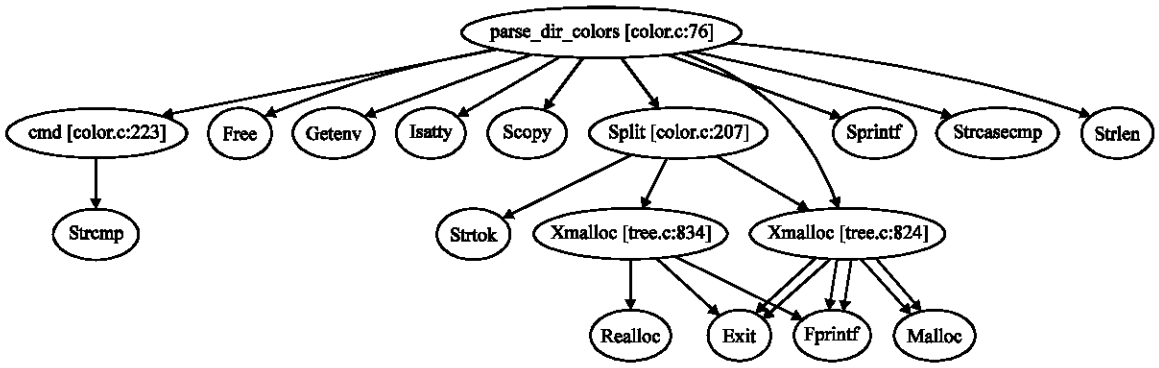


Fig. 6: Call graph generated by dot

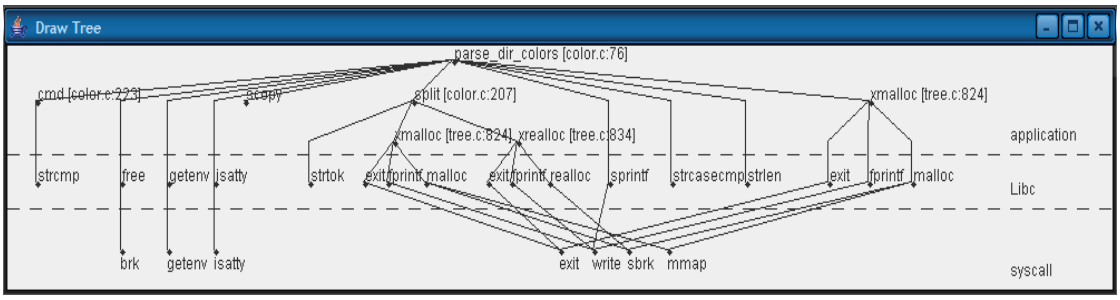
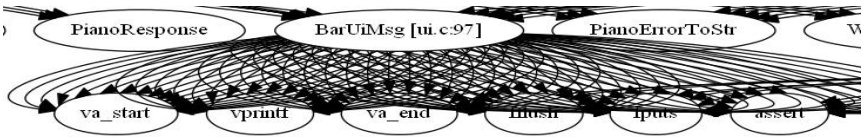
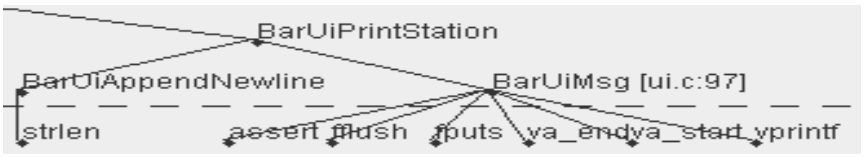


Fig. 7: Call graph generated by our method



Repeated function calls generated by dot



Repeated function calls generated by CallTreeLin

Fig. 8: Comparison of repeated functions calls

crossing. The experimental results also show that there is little difference between the numbers of layer of out methods and dot. In order to reduce the area of layout, crossings generated by dot increase remarkably with the increasing of function number. While there is not any crossing generated by our method Fig. 10.

Another major objective of our study is to reduce layout area on the basis of better readability and understandability. So we compared the layout area of our method with dot under the circumstance that the character size of function name is same in both methods (Fig. 11).

When the software is relatively simple, because the node size is smaller, with our method the layout area is much smaller than that of dot. With the increasing of software complexity, more nodes are added to the layout by our method and the layout area increase dramatically. Therefore with increase of function number, the layout area of out method become closer to dot but still has a little advantage in number of pixels Fig. 11.

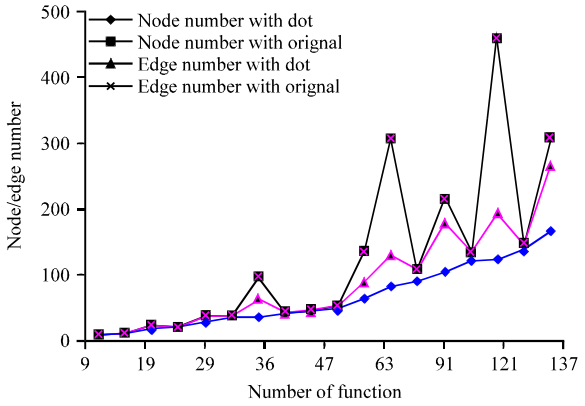


Fig. 9: Comparison of node and edge number

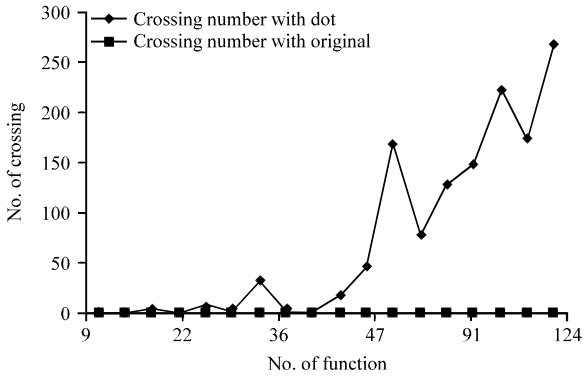


Fig. 10: Comparison of edge-crossing number

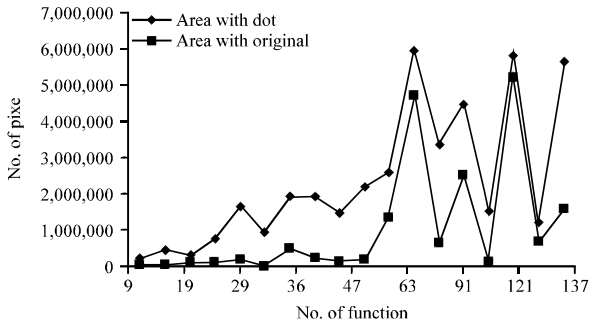


Fig. 11: Comparison of layout area

The Significance of Layout: When designing the source code visualizer, we wish that the layout could reflect more information about functions.

First, the layout should embody the call-relation and dependent-relation between functions directly. We can see that the functions within a subtree are relevant to each other and the nodes depend on their descendants. Hence, the layout may assist users to understand the software structure, to tailor Linux into a smaller special-purpose system and work as an auxiliary tool for software maintenance.

Second, layout should describe corresponding libc functions and system calls that are called by each function particularly. By analyzing the features of libc functions and system calls, the core functions of software can be derived. According to functions, libc functions and system calls can be classified into file processing, data processing, time control, user management and so on. As shown in Fig. 7, libc functions, such as strcmp, strlen and strcasecmp, are operations that manipulate the string. So, we can infer from the libc functions that the main goal of these source codes is to operate on string.

Finally, the complexity of layout reflects the complexity and scale of the program directly. As we can see in Fig. 9, program complexity is proportional to the node number and the layout breadth. Large-scale software usually has bigger width. If the width of single function is too big, it may imply that the module is relatively complicated and should be divided into submodules. The depth of call graph is proportional to the height of layout. If the height of layout is too high, it may imply that the coupling degree of function is relatively high and it needs to be improved. By testing the existing software, the layout of most mature software product keeps a depth within 6. In conclusion, the layout may assist users to measure the complexity of software and to evaluate the software.

At present, there are two kinds of source-code analysis tool for Linux: cross-reference pattern and graphic pattern. LXR analyze the code in Linux automatically but reflect functional relationship in a textual form (Gleditsch and Gjermshus, 2010). Most of visual tools, such as CodeViz (Gorman, 2012), Doxygen (Heesch, 2008.) and so on can generate call graph effectively using Graphviz (Gansner and North, 2000) as front end for call graph layout. We designed a layering tool that can change the hierarchical attribute of a node according to its system level. And the layout algorithm is optimized to reflect the system hierarchical structure which makes our method unique to other source-code analysis tools. Meanwhile the area of layout is relatively smaller than Graphviz.

CONCLUSION

This study developed a tool that can parse the C source codes in Linux into call graph. A set of layout constrains based on Sugiymama hierarchical layout algorithm are selected. With these constrains, the issues caused by call graph generating and hierarchical layout

are analyzed. And we improved the tree layout algorithm based on Walker's algorithm to make it suitable for C code in Linux. We analyzed the problem arose when combining Linux system structure to hierarchical layout algorithm and gave possible solutions. Virtual nodes are set avoiding level-crossing problem and Tree Space is set avoiding subtrees overlapping along X-axis direction to improve visual presentation.

According to the layout effects of applications, the experimental results show that our method is more suitable for source code visualization in Linux than common source code visualizer. Functional dependency and software structures are represented clearly. The above information play an important role in Linux code analysis, Linux software reverse engineering, system optimization, system tailoring and customization of embedded system. In the future we will consider how to represent other attributes of functions during the visualization procedure, such as functional property, data dependence, coupling degree and cohesion.

ACKNOWLEDGMENT

This research study is financially supported by Scientific Research Fund of Hunan Provincial Education Department (No. 10C0368).

REFERENCES

Bernardi, S., J. Merseguer and D. C. Petriu, 2012. Dependability modeling and analysis of software systems specified with UML. ACM Comput. Surv., 45: 14-22.

Bohnet, J. and J. Doellner, 2008. Analyzing dynamic call graphs enhanced with program state information for feature location and understanding. Proceedings of the 30th International Conference on Software Engineering, May 10-18, 2008, Leipzig, Germany, pp: 915-916.

Buchheim, C., M. Junger and S. Leipert, 2002. Improving walker's algorithm to run in linear time. Proceedings of the 10th International Symposium on Graph Drawing, August 26-28, 2002, Irvine, CA, USA., pp: 347-364.

Davis, T.A., K. Pestka and A. Kaplan, 2003. Kscope: A modularized tool for 3d visualization of object-oriented programs. Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis, May 26-28, 2003, Grenoble, France, pp: 128-134.

De Figueiredo Carneiro, G., R. Magnavita and M. Mendonca, 2008. Combining software visualization paradigms to support software comprehension activities. Proceedings of the 4th ACM symposium on Software visualization, September 16-17, 2008, Munich, Germany, pp: 201-202.

Gang, X., 2009. Design and Implementation of C program call graph construction algorithms. J. Guizhou Univ., 27: 77-81.

Gansner, E.R. and S.C. North, 2000. An open graph visualization system and its applications to software engineering. Software-Pract. Experience, 11: 1203-1233.

Gleditsch and P.K. Gjermshus, 2010. LXR, Linux cross-reference. LXR community. http://lxr.linux.no/.

Gorman, M., 2012. CodeViz: A callgraph visualiser. http://www.csn.ul.ie/~mel/projects/codeviz/.

Harman, M., 2010. Why source code analysis and manipulation will always be important. Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation, September 12-13, 2010, Timisoara, Romania, pp: 7-19.

Heesch, D., 2008. Doxygen: Generate documentation from source code. http://www.stack.nl/~dimitri/doxygen/.

Nan, L., 2006. Design and implementation of call graph analysis tool for aspect-oriented program. Shanghai Communication Univ., pp: 12-15

Schilling, J., 2010. Calltree: Valgrind skin for cache simulation and call tracing. http://www.usinglinux.org/devel/calltree.html.

Sugiyama, K., S. Tagawa and M. Toda, 1981. Methods for visual understanding of hierarchical system structures. IEEE Trans. Syst. Man Cybernet., 11: 109-125.

Walker, J.Q., 1990. A node-positioning algorithm for general trees. Software-Pract. Experience, 20: 685-705.

Wang, Q., W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren and C. Verbrugge, 2003. EVolve: An open extensible software visualization framework. Proceedings of the 2003 ACM symposium on Software visualization, June 11-13, 2003, California, USA, pp: 37-43.