13TH INTERNATIONAL DEPENDENCY AND STRUCTURE MODELLING CONFERENCE, DSM'11
CAMBRIDGE, MASSACHUSETTS, USA, SEPTEMBER 14 – 15, 2011

DESIGN STRUCTURE OF SCIENTIFIC SOFTWARE – A CASE STUDY

Shahadat Hossain[1], Ahmed Tahsin Zulkarnine[1]

[1]Department of Mathematics and Computer Science, University of Lethbridge, AB T1K 3M4, CANADA

ABSTRACT

In this paper we report results from an exploratory study of design structures in scientific research software. Dependency Structure Matrix (DSM) is used as a modelling tool to capture and analyze dependencies among system elements such as functions. We compute several architectural complexity metrics and present preliminary results from two open-source scientific computing software applications.

Keywords: Research Software, Network, Structural Property, Software Engineering

1 INTRODUCTION

The utility of properties e.g., modularity and information hiding (Parnas, 1972) in complex software design is now widely recognized in software engineering community. An evolving software system needs to have a design architecture that allows easy accommodation of functional changes and asynchronous (re-)development of parts of the system. The dependency structure matrix has been used as a tool to analyze and compare alternative design decisions and quantify structural metrics e.g., modularity in large and complex software systems (MacCormack et al., 2006, Sosa et al., 2007a, Sosa, 2008, Sangal et al., 2005, LaMantia et al., 2008).

In this paper we study software systems architecture specifically designed for problems arising in scientific and engineering applications (Kelly and Sanders, 2008, Marques and Drummond, 2005). While some scientific computing software applications are primarily designed as a proof-of-concept tool, with the advent of more powerful hardware resources e.g., supercomputers, a growing number of scientific applications are being developed to perform large-scale simulation runs that were previously intractable (Trillinos, 2011, SciDAC, 2011). Unlike the one-time throwaway computer code, these simulation software applications are highly complex and large (IPSL-CM5, 2011) (millions of lines of code). The applications involve substantial investment in time and other expensive resources and tend to have lifecycles measured in tens of years. Some of the main concerns in the design of research software are to do with the correctness of the computed output and scalability of the software, especially with regard to high-performance and emerging hardware technology (Marques and Drummond, 2005). A distinguishing feature of the designers of such software applications is that they are highly trained scientists with little or no formal background in modern software engineering practices. The main objective of such software is to produce new scientific knowledge. The finished products typically are of very high quality and efficient (Heroux and Willenbring, 2009, Kelly and Sanders, 2008). On the other hand, being very focused on narrowly defined application domains, important software quality metrics e.g., usability (user interface), extensibility etc., may not be among the list of primary design objectives (Heroux and Willenbring, 2009, Morris, 2008).

The main purpose of this work is to examine and understand the design structure of scientific computing research software by analyzing the interactions between design elements, with particular emphasis on metrics that quantify the modularity of design, the effect of changes in system's architecture due to the need for porting the application to emerging high-performance computing system or the integration with external systems.

In (MacCormack et al., 2006) the DSM technique is applied to study dependencies among system elements of two large-scale software applications. It is noted that there exists a strong correspondence between the design structure of the software and the organization in which it is developed. The geographically distributed nature of the development team is reflected in the more modular architecture of the open-source $Linux$ compared with the proprietary $Mozilla$, in which the developers have direct face-to-face interactions. The identification of dominant subsystems or modules of software systems and their dependency analysis constitute key considerations in managing architectural evolution of complex software products (MacCormack et al., 2006, Sangal et al., 2005, Sosa et al. 2007a, Sosa et al., 2007b). As noted earlier, the design goals of scientific research software systems and the organization in which they are developed are somewhat different from that of commercial or general-purpose software systems (MacCormack et al., 2006) and therefore present itself as an important and interesting case-study. In our work we choose automatic differentiation (AD) software (see Griewank and Walther, 2008) - software applications that are concerned with the automatic computation of derivatives or sensitivities of mathematical functions that are given as computer programs. Our choice for this particular application type is influenced by the observation that computation or estimation of derivatives or sensitivities of outputs of a mathematical model with respect to its input parameters, is a frequently required step in many algorithms for solving scientific and engineering problems. Therefore, software tools implementing automatic differentiation of computer programs constitute appropriate test cases for scientific software applications that are intended to be a part of other major scientific applications. We choose ADOL-C (Griewank et al., 1996) and CppAD (Bell, 2011) as representative AD software that are built utilizing 'operator overloading' technique. We note that the other main implementation technique for AD software, 'source transformation', is not considered in this paper. Both the applications are available from COIN-OR (COIN-OR, 2011) project as open-source software under public license.

2 METRICS FOR ANALYZING DESIGN STRUCTURE

Given below is a description of structural metrics we use in this work.

1. **Characteristic path length.** In an undirected graph, the average distance between nodes i and j is defined by, $= \frac{\sum_{i \neq j} d_{ij}}{N(N-1)}$, where d_{ij} is the shortest path length (minimum number of edges) connecting the nodes.
2. **Clustering co-efficient.** A measure of degree to which nodes in a graph tend to cluster together in an undirected graph is defined by, $C = \frac{1}{N}\sum_{i=1}^{N} C_i$, where $C_i = \frac{(2n_i)}{k_i(k_i-1)}$, denotes the clustering coefficient of node i, with k_i being the number of nodes connected to node i, and n_i being the actual number of edges between those k_i adjacent nodes.
3. **Nodal degree.** The average degree of the nodes in the graph $k = \frac{1}{N}\sum_{i=1}^{N} k_i$ where k_i is the number nodes adjacent to node i (also the degree of node i). For directed graphs the degree of node i is the sum of its in-degree (number of directed edges pointing to node i) and out-degree (number of directed edges pointing away from node i to other nodes).
4. **Strongly connected Components**. A directed graph is called strongly connected if there is a directed path from each vertex in the graph to every other vertex. The strongly connected components of a directed graph are its maximal strongly connected sub-graphs.
5. **Propagation Cost.** This is a measure of the proportion, on average, of design elements that are affected due to a change to a specific design element and is given by $\sum_{i=1}^{N} p_i \ /N^2$, where p_i is the number of nodes reachable from node i using a directed path with minimum number of edges.

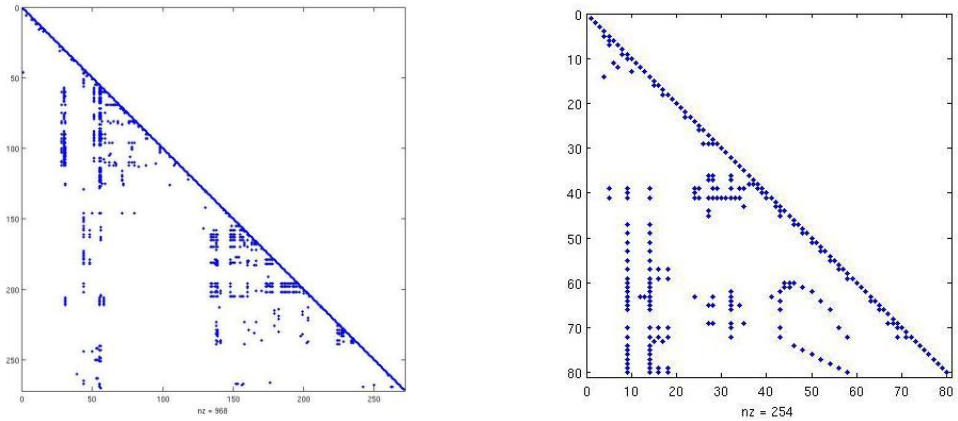
We distinguish between functions that are explicitly implemented in the software under consideration (denoted *user function*) and the functions that are part of the software libraries (e.g., input/output functions). In our work user functions are the basic design elements (nodes in the associated directed graph, henceforth the call graph), and function i is said to depend on function j if it calls j from within its body, which is denoted by a mark in row i and column j of the associated DSM (by a directed edge from node i to node j in the call graph). In order to extract the dependency information and to generate the static call graph we have used the gcc-based call-and-structure extractor developed by a research

team from The University of Groningen, the Netherlands (Telea et al., 2009).

Table 1 – Structural Properties of ADOL-C and CppAD

Software	Nodes		Directed Edges	
	Files	User Functions	Files	User Functions
ADOL-C	60	271	66	703
CppAD	66	80	67	175

Table 1 displays the number of system elements (nodes) and the number of links (directed edges) in the two software tools under consideration. The column labelled 'Files' represent the number of files the functions are contained in. It is readily apparent that both the call graphs are very sparse i.e., only a small fraction of the possible edges are present. After constructing the DSM we use a strongly-connected component partitioning tool (Hossain, 2010) to rearrange the DSM into block triangular form.



ADOL-C CppAD

Figure 1 Partitioned DSM of ADOL-C and CppAD

In Figures 1, the DSM for ADOL-C and CppAD are displayed after the partitioning algorithm is applied to the user function call graph. An important observation that can be made from the figures is the absence of any feedback mark in the respective DSMs. This is also indicated by the number of strongly connected components being the same as the number of design elements in Table 2. From a graph-theoretic viewpoint, a triangular DSM is manifested in the acyclic (directed) nature of the associated function call graph. We note that ADOL-C project involves multiple (about ten) developers while CppAD is a one-person project. In scientific software development where computational efficiency is one of the main goals, running-time profiling is a necessary step. Profiling tools e.g. 'gprof' (Susan et al., 2004) usually provide information on whether a function is part of a cycle in the static call graph of the program. In the context of static function call DSM, the presence of feed-back marks complicates the accurate profiling (computing time values). For example, functions a and b are mutually dependent in the call graph if function a calls function b which in turn calls function a. The execution time incurred in function a will include the time incurred in the called function b, whose running time, in turn, must include the time for executing function a - thereby invalidating the profiling procedure. Generally speaking, circular dependencies (direct or indirect recursions) are avoided to enable certain code optimization features in the compiler. We conjecture that for the software tools studied, circular dependencies have most likely been discovered early and reworked at the initial design phase.

Table 2- Design Structure Metrics

Software	Characteristic Path length, l	Clustering co-efficient, C	Nodal Degree	Number of Components	Propagation Cost (%)
ADOL-C	2.05005	0.080382	5.18819	271	3.41635
CppAD	2.37373	0.0342364	4.375	80	6.64062

Table 2 displays a suite of structural metrics and their values from the two DSMs. The propagation costs of 3.4 and 6.6 indicate that, on average, a change in the implementation of any function in the software has the potential of affecting only 3.4% and 6.6%, respectively, of functions. A similar observation with regard to propagation cost has been made in (MacCormack et al., 2006). Following Braha and Bar-Yam (Braha and Bar-Yam, 2007) we ignore the direction of the edges in the respective call graphs concerning the metrics 'characteristic path length' and 'clustering coefficient'. This is a reasonable assumption since, in general, the caller and the called functions may exchange information via in- and out-parameters. Viewing the call-graph as an information flow network (Braha and Bar-Yam, 2007), structural metrics such as characteristic path length, clustering co-efficient, and nodal degree and its distribution provide useful information about the architecture of the underlying product. From Table 2, we observe small average nodal degree and shorter average distance between any two nodes in the networks. On the other hand, the tendency of the related functions being highly interacting (measured by the clustering coefficient), is almost an order of magnitude smaller than that of the operating system software reported in (Braha and Bar-Yam, 2007).

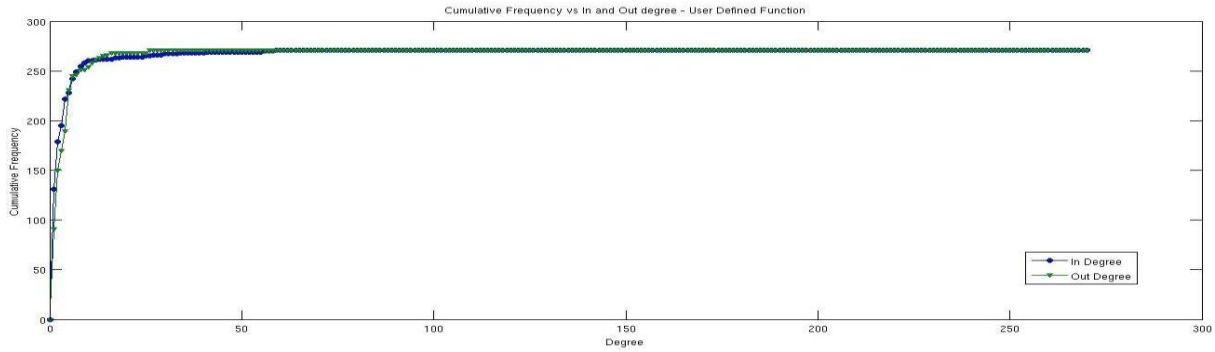


Figure 2 - Cumulative Frequency Vs. Degree (In-degree/Out-degree) of ADOL-C

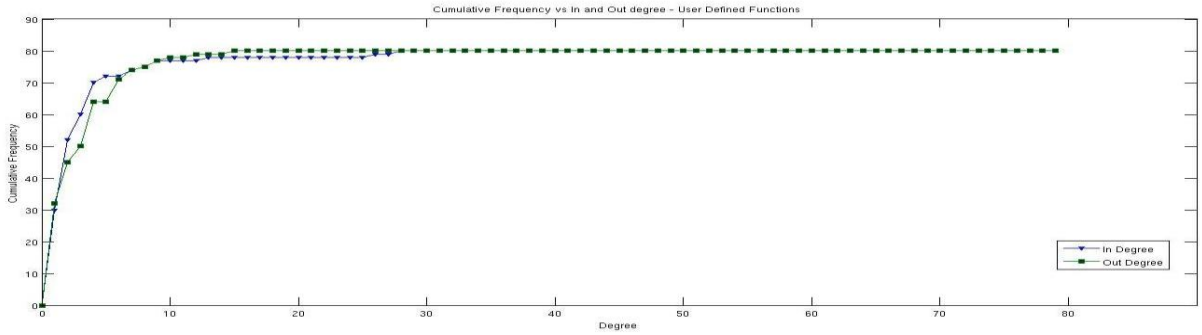


Figure 3 - Cumulative Frequency Vs. In-degree and Out-degree of CppAD

Figures 2 and 3 present cumulative (in-, -out) degree distributions of call graph nodes. We note that the total nodal degree varies from 1 to 62 for ADOL-C and from 1 to 29 for CppAD with approximately 80% of the nodes having degree less than or equal to 8 for ADOL-C and 6 for CppAD. In other words, only a small fraction of the functions in both software tools are most relevant with regard to the functioning of the software. The degree distribution analysis provides local information only. To obtain global information on how function elements exert their influence on other functions we use an index that measures the centrality of a node by the number of shortest paths in the call graph containing that node. For ADOL-C, function **fail** has been found to be included in the maximum number (16398) of shortest paths (directed) between any pair of nodes while for CppAD the corresponding function is **constructor-special** which is included in 2238 shortest paths. This observation is not surprising since 'correctness' of computed values is one of the main design goals in scientific computing software (Kelly and Sanders, 2008, Heroux and Willenbring, 2009). In case of CppAD, we note it heavily uses object-oriented features compared with ADOL-C and a constructor function is one of the most frequently called member function.

3 CONCLUDING REMARKS

In this paper we perform dependency analysis of function call graphs for two scientific research software tools. Unlike projects (e.g., computer operating systems) where formal software engineering practices are perceived important for their success, the main goal in scientific research software is the creation and validation of new scientific knowledge. The call graphs for the studied software tools display shorter characteristic path lengths, small nodal degrees, and small propagation costs, similar to general-purpose software such as operating systems (Braha and Bar-Yam, 2007, MacCormack et al., 2006). On the other hand, a relatively small clustering coefficient in ADOL-C and CppAD points to a less modular design structure. Furthermore, absence of circular dependencies in the studied software can be attributed to the strong emphasis placed on the computational performance of the code (noting that recursive function calls, in general, are considered a hindrance to the performance enhancing code optimization e.g. 'in-lining' of functions, regularly performed by modern optimizing compilers).

In addition to performing more detailed analyses of the structural metrics, there are a number of extensions to this work that we envision in future. First, it will be interesting to perform design structure analysis to compare and contrast scientific software from multiple application domains. Secondly, to obtain a better understanding of the architecture of software products from multiple application domains, it is helpful to develop domain-specific centrality metrics. For example, natural inquiries in this regard could be 'how well does this software integrate into a larger and complex super system?', 'how sensitive is the software to new or emerging hardware technologies?'.

ACKNOWLEDGEMENTS

This research is supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada Discovery Grant (Individual). The authors wish to thank the anonymous referees for many helpful suggestions that improved the presentation of the manuscript.

REFERENCES

Braha, D., & Bar-Yam, Y. (2007). The Statistical Mechanics of Complex Product Development: Empirical and Analytical Results. *Management Science, 53*(July 2007), 1127-1145.

Bell, B. M. (2011). CppAD: A Package for Differentiation of C++ Algorithms. *http://www.coin-or.org/CppAD/* (accessed May 2011).

Computational Infrastructure for Operations Research (COIN-OR). *http://www.coin-or.org/*. (accessed May 2011)

Griewank, A., Juedes, D. & Utke, J. (1996). Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS).*22(2). 131-167.

Griewank, A, & Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (Second ed.), SIAM, Philadelphia, PA, USA, ISBN: 978-0-898716-59-7.

Heroux, M. A., & Willenbring, J. M. (2009). Barely Sufficient Software Engineering: 10 Practices to Improve your CSE Software. *SECSE'09* (May 2009 , Vancouver, Canada), 15-21.

Hossain, S. (2010). Efficiently Computing with Design Structure Matrices. *In the Proceedings of 12th International DSM Conference* (July 2010), 345-358.

Kelly, D., & Sanders, R. (2008). Assessing the Quality of Scientific Softwares. *First International Workshop on Software Engineering for Computational Science and Engineering* (May 2008, Leipzeig, Germany).

LaMantia, M. J., Chai, Y., & MacCorMack, A. (2008). Analyzing the Evolution of Large-Scale Software Systems using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases. *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)* (WICSA '08). IEEE Computer Society, Washington, DC, USA, 83-92.

MacCorMack, A., John Rusnak, & Baldwin, C. Y. (2006). Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science, 52* (July 2006), 1015-1030.

Marques, O., & Drummond, T. (2005). Building a Software Infrastructure for Computational Science Application: Lessons and Solutions. *SE-HPC'05* (May 2005, St. Louis Missouri, USA), 40-44.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM 15* (December 1972), 1053-1058.

Sangal, N., Jordan, E., Sinha, V., & Jackson, D. (2005). Using Dependency Models fo Manage Complex Software Architecture. *OOPSLA'05* (October 2005), 167-176.

Scientific Discovery through Advanved Computing (SciDAC). *http://www.scidac.gov/*. (accessed May 2011)

Sosa, M., Browing, T. R., & Mihm, J. (2007a). Dynamic, DSM-Based Analysis of Software Product Architecture. *9th International DSM Conference* (October 2007), 349-362.

Sosa, M. E. (2008). A structured approach to predicting and managing technical interactions in software development. *Research in Engineering Design*, 19(1), 47-70.

Sosa, M. E., Browning, T., & Mihm, J. (2007b). Studying the dynamics of the architecture of software products. *ASME 2007 International Design Engineering Technical Conferences & Computer and Information in Engineering Conference.* (September 2007), 329-342.

Sullivan, K. J., Griswold, W. G., Chai, Y., & Hallen, B. (2001). The Structure and Value of Modularity in Software Design. *ESEC/FSE 2001* (2001 Vienna, Austria), 99-108.

Susan L. Graham, P. B. K., Marshall K. McKusick. (2004). gprof: a call graph execution profiler. *ACM SIGPLAN Notices - Best of PLDI 1979-1999*, *39*(4), 49-57.

Telea, A., H. H., Ozan Ersoy and Dennie Reniers. (2009). Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study. *5th IEEE International Workshop on Visualizing Software* (Edmonton, Canada), 81-88.

The Trillinos Project, *http://trilinos.sandia.gov/*. (accessed May 2011).

Contact: Shahadat Hossain, University of Lethbridge, Department of Mathematics and Computer Science, Lethbridge, Alberta T1K 3M4, Canada, Phone (1) 403 329 2475, Fax (1) 403 317 2882, e-mail shahadat.hossain@uleth.ca