Malaysian Journal of Computer Science, Vol. 17 No. 2, December 2004, pp. 55-67

VIEWING SOFTWARE ARTIFACTS FOR DIFFERENT SOFTWARE MAINTENANCE CATEGORIES USING GRAPH REPRESENTATIONS

Shahida Sulaiman
School of Computer Sciences
Universiti Sains Malaysia
11800 USM, Penang
Malaysia
Tel: 604 - 6533888 ext. 4384
Fax: 604 - 6573335
email: shahida@cs.usm.my

ABSTRACT

Information needed during an iterative process of a software maintenance process is much different from that of a software development process. Without up-dated documents, software maintainers need to gain information required to solve different maintenance categories through source codes hence consuming more time and effort. With the emergence of reverse engineering tools, the process of understanding source codes to solve maintenance tasks in different maintenance categories can be improved. Such tools employ diverse software visualisation methods that generate graph representations of parsed software artifacts. This paper discusses on how the graph representations provided by the proposed DocLike Modularised Graph (DMG) method employed in DocLike Viewer prototype tool can serve different levels of information needed by software maintainers in the case of corrective, adaptive and perfective maintenance category. It is observed that not only do software maintainers require diverse level of information; the necessity of the information is also not exactly of the same degree.

Keywords: ***Software maintenance, Software visualisation, Graph representations, Software understanding***

1.0 INTRODUCTION

A software system is developed once but is maintained infinitely. The iterative maintenance process requires different types of changes for instance to correct bugs or design, to adapt environmental changes or to make enhancement for a better performance. Different types of changes may require different types of information. Without any documents or up-dated documents, software maintainers need to comprehend source codes prior to making any changes. Since software engineers are still confronted with documentation-related problems in their practice [16], this problem still needs further attention by researchers and practitioners.

Reverse engineering (RE) is the process of analysing a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction [3]. Without RE tools, software maintainers need to reverse engineer a software system manually, a process that is tedious and time consuming. An important method integrated with RE tools is known as Software Visualisation (SV) that is the use of interactive computer graphics, typography, graphic design, animation and cinematography to enhance interface between software engineers or computer science students and their programs [10]. Thus in this paper, SV in RE environment refers to the use of graphs to enhance the cognition or understanding of a program that has already been written by representing the information or software artifacts extracted from a RE process. A survey by Koschke [7] shows graphs as the most often used kind of visualisation. According to the study, graph is a simple, yet powerful concept to express in general binary relations and has a 'natural' visualisation as boxes and arrows. Thus graphs represent a generic way to represent information, which is probably the reason why they are so popular. Thus graph representations are also selected in this work.

Some instances of CASE (Computer-Aided Software Engineering) workbenches in the class of maintenance and reverse engineering are CIA [2], Rigi [19], PBS [9], CodeCrawler [8] and SNiFF+ [18]. The tools are normally incorporated with an editor window in which the abstracted software artifacts will be visualised graphically besides their textual information. These tools are able to assist and optimise software engineers' program comprehension or cognitive strategies, particularly when there is an absence of design level documentation that should describe the architecture and dependencies among the components for various levels of software abstraction. Existing methods of

the tools focus on visualising software artifacts. Nevertheless, they do not explicitly relate how much their tools can serve the information needed in different maintenance types or categories. Thus, this paper discusses the use of the proposed DocLike Modularised Graph (DMG) method employed in DocLike Viewer prototype tool that was developed by the author. The tool represents existing software artifacts graphically in a standardised document-like manner, module-by-module to serve different information levels.

The following Section 2 discusses the DMG method employed in DocLike Viewer prototype tool. Section 3 provides a brief explanation of software maintenance categories and discusses the three maintenance cases captured from software engineers. Section 4 summarises the type of information required based on the discussion of the maintenance cases in Section 3. In Section 5 the viewing aspect provided by the proposed DMG method and DocLike Viewer prototype tool will be discussed thoroughly based on the summary of information needs in Section 4. Then Section 6 ponders some related works and finally Section 7 draws the conclusion and possible future work.

2.0 DMG METHOD EMPLOYED IN DOCLIKE VIEWER

The proposed DocLike Modularised Graph or DMG method employs graph to visualise different levels of software abstraction. A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E, such that each edge in E is a connection between a pair of vertices in V [11]. The method uses directed graph described as directed edge $e_n = (v_i, v_j)$. A vertex in G can be of different types. In object-oriented programming the types can be package, class, method or data while the counterparts in structured programming comprises of module, program, procedure (or function) or data. Since DocLike Viewer prototype tool is based on an existing C language parser, only the types as in structured programming are considered, which are symbolised as module (M), program (P), procedure or function (F) and data (D).

Thus an edge e_n of set E in G can be one of these dependency types: module-to-module (M, M), program-to-program (P, P), procedure/function-to-procedure/function (F, F) or procedure/function-to-data (F, D). The relationships represented by the directed edges are (*calls*, m_1, m_2), (*calls*, p_1, p_2), (*calls*, f_1, f_2) and (*uses*, f, d). The DocLike Modularised Graph method $DMG_i = (V_i, E_i)$ attempts to simplify the visualisation by enquiring modularisation knowledge from users prior to drawing the graph in a document-like manner. For a new re-documentation process, users are provided with a set of components C comprises a number of subsets of procedure or function, for instance the set F_i of set F with its associated program P_i of set P. The rule of modularisation that produces a set of modules M is described as "no procedures or functions of the same program can be in different module" that is if the set F_i associated with P_i and Modularised in M_i then all members of set F_i must be in module M_i. Hence after the modularisation process, a set of modules M is produced in which $M = (F, P)$. After identifying the modules, the sections of the re-documented subject system can be generated automatically. Each section that is associated with the graph representation will be decomposed according to the specified module of set M.

The *DMG* method provides five types of graph representations that are associated with the respective sections as follows:

(i) Section 3.1 Module decomposition: $DMG_1 = (V_i, E_i)$ where the set $V_i \subseteq M$ represents all modules of the set M and E_i represents the relationship (*calls*, m_1, m_2).

(ii) Section 3.1.m_i Module m_i description: $DMG_2 = (V_i, E_i)$ where the set $V_i \subseteq P$ represents all programs of the set P associated with the module m_i and E_i represents the relationship (*calls*, p_1, p_2) in the module m_i only.

(iii) Section 5.1.m_i Module m_i interface: $DMG_3 = (V_i, E_i)$ where the set $V_i \subseteq F$ represents all procedures or functions of the set F associated with the module m_i and E_i represents the relationship (*calls*, f_1, f_2) in the module m_i only.

(iv) Section 4.1.m_i Module m_i dependencies: $DMG_4 = (V_i, E_i)$ where the set $V_i \subseteq F$ represents all procedures or functions of the set F associated with the module m_i and E_i represents relationship (*calls*, f_1, f_2) in the module other than m_i including the compiler standard library.

(v) Section 4.3.m_i Module m_i data dependencies: $DMG_5 = (V_i, E_i)$ where the set $V_i \subseteq F$ and $V_i \subseteq D$ represent all procedures or functions F_i of the set F associated with the program P_i of module m_i and all associated global data of the set D defined in the program P_i or the header file *.h*, while E_i represents the use of data (either read or write or both read and write) by F_i.

DocLike Viewer consists of three main panels: Content Panel, Graph Panel and Description Panel (see Fig. 1). By clicking a section in the Content Panel that is associated with any one of the five graph representations discussed above, users can view different levels of information abstraction. Currently, DocLike Viewer extracts parsed software artifacts produced by C parser of Rigi tool [19] before generating the graph representations in its viewer. Refer the author's previous work [17] for more details of DMG method and its tool.

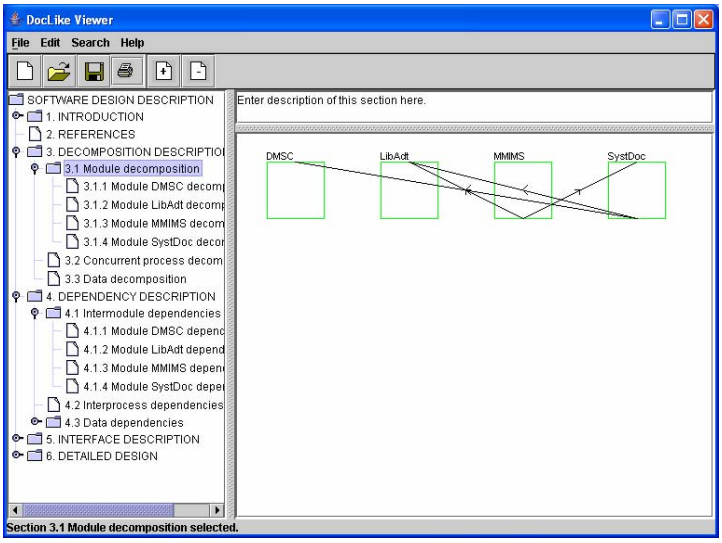

Fig. 1: DocLike Viewer consists of Content Panel (left), Description Panel (top, right) and Graph Panel (bottom, right)

3.0 SOFTWARE MAINTENANCE

Software maintenance is the process of modifying the software system or component after delivery to correct faults, improve performance or other attributes, or to adapt to a change in an environment [5]. In the following sub-sections the four categories of software maintenance will be discussed followed by the three maintenance cases indicated in this study.

3.1 Categories of Software Maintenance

Basically there are four categories of software maintenance. Based on the definition in [12] and [6] the categories are summarised as follows:

(i) Corrective Maintenance: repair design and programming errors.

(ii) Adaptive maintenance: modify system to adapt environmental changes without radical changes of software functionality.

(iii) Perfective maintenance: add desired (not necessarily required) new features to improve performance, maintainability or other attributes of a computer program.

(iv) Preventive maintenance: performed for the purpose of preventing problems before they occur. IEEE only recognised the preventive maintenance in 1998 and this category is still debated by researchers.

A maintenance task can cover a combination of the maintenance categories. For instance a case study in maintenance of an object-oriented software application in which a software engineer replaced a text-based user interface with a GUI based shows that 94.8% of maintenance effort was perfective (development of GUI), 3.2% adaptive and 2.0% corrective [4]. This paper discusses the maintenance cases of the established maintenance categories of corrective, adaptive and perfective.

3.2 Software Maintenance Cases

This section elucidates three maintenance cases derived by interviewing three software engineers in three different companies. The three maintenance categories include corrective, adaptive and perfective category. Each

maintenance case might involve more than one category as discussed in the previous section but in this study only the main category for each case is identified and discussed. The case of preventive category was not studied in this work.

3.2.1 Maintenance Case 1 – Corrective

A user of a DBase system encountered a problem when generating a weekly report. The data value of a field in the report was not produced correctly. It was believed that the current field size could not accommodate the value generated. The problem was quite obvious in which a software maintainer could straight away change the program in which the report layout was defined by checking the source code and identifying the related variables using the search utility. Then the maintainer checked if the field was related to any other files or tables because some reports retained the value of certain fields into certain file or table. The maintainer needed to also check whether the change might affect other reports or forms in the system. The information required in this scenario is at lower level in which components affected might be within a particular module only.

3.2.2 Maintenance Case 2 – Adaptive

A programmer was assigned to reverse engineer a Cobol system manually that should be maintained to migrate from DOS-based to Windows-based platform. No documentation was provided. She took the following steps:

(i) Ran the system and observed the menus, forms, reports and the navigation flow among them.
(ii) Identified the program or name of the source code for each menu, form and report.
(iii) Drew the graphical representation or hierarchy view of the whole modules in the system.
(iv) Identified the database and listed the name of the files or tables.
(v) Used the search utility to list “Open” and “Close” words in each program which indicated the files used and their mode (read, write or both).
(vi) Combined the programs listed in (ii) and files or tables listed in (iv) into a cross-reference table then the open mode in (v) specified accordingly.
(vii) Finalised the list by checking the existing source codes and names of the files in the soft copy of the system. Some programs or files were not used. Thus she had to determine whether to discard the physical files or to retain them. So she used the search utility to double check the dependency of those files with other files or programs.
(viii) Listed the structure of each file or table and identified the one that might be changed based on the new database platform.

The materials could be used to assist the project leader to make the schedule for the maintenance project based on the number of programs to be modified. Besides, the software manager could later use the materials to estimate the maintenance cost to be quoted. In this case, the type of information required is quite global. The overall understanding of the software is vital in order to identify the whole structure of the software system before migrating to the new platform.

3.2.3 Maintenance Case 3 – Perfective

A new programmer was assigned to maintain a Clipper system. The new requirement was to add a new table to the existing system. She derived the whole picture of the system from the out-dated document and started to identify the component in which she could insert the new components.
Briefly the steps taken are as follows:

(i) Identified the menu in which she could insert the new selection to update the new table and generate reports from the tables.
(ii) Designed the new entry form and reports based on existing forms and reports format, then created the new table with all the required fields.
(iii) Tested the new programs written and integrated them with existing components.

However, when she consulted her superior, she was told that some of the lines in the codes could be eliminated if she used the functions provided in common module created by previous programmers or the standard libraries. The programmer did not practice software reuse because she did not study the whole components of existing system thoroughly before enhancing the system. In this case, initial information needed is not that global in order to ensure no redundancy of source codes occur. However, a global understanding of the system is still necessary in order to practice software reuse.

4.0 TYPE OF INFORMATION

This section summarises graphically the information sought in the maintenance cases previously discussed. Fig. 2 shows the information required in the first case (corrective maintenance). In this case modifying the Grand Total Variable in the Report Program may affect the Grand Total Field in Report Table. Consequently the change of the Report Table may also affect existing Report or Form Program that uses the Grand Total Field.

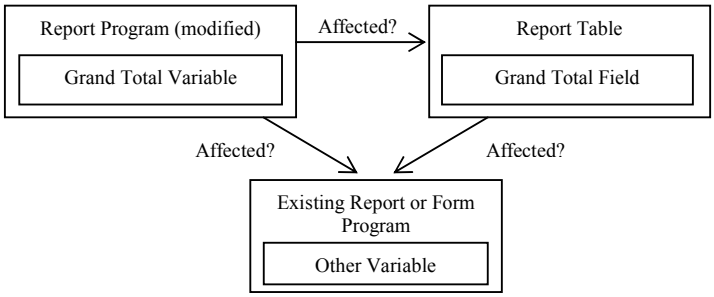

Fig. 2: Information required in the corrective maintenance case study

The information required in the second case (adaptive maintenance) is visualised as in Fig. 3. It reveals that the information needed is of higher level compared to the corrective maintenance case. The Cobol System should be studied thoroughly by studying the system module-by-module. For each module for instance Report Module, the programmer should check all the report programs available in the module. Then the programmer should indicate the dependencies of each program with other report programs including the data or field usage of Report Tables. Hence, a lower level of abstraction is necessary in order to identify which programs and files or tables are actually affected in the maintenance process.

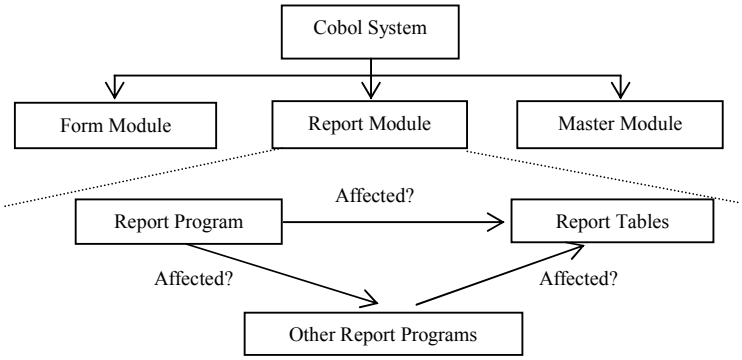

Fig. 3: Information required in the adaptive maintenance case study

The third case (perfective maintenance) needs the information as illustrated in Fig. 4. In this case, an understanding of existing common modules or libraries is necessary prior to creating a new program or a new file or table. The existing menu program definitely needs to be changed in order to add new selection items related to the new requirements. Then the programmer should also determine whether the new form or report programs introduced might affect the existing programs besides the existing menu program.

From the maintenance cases, it is observed that different maintenance categories require different level of information abstraction. As the guideline for this study, the level of information abstraction is described as in Table 1. This study classifies the level of software abstraction as Very high level (L1), High level (L2, L3), Low level (L4, L5) and Very low level (L6). The detail description of each level is shown in the table.

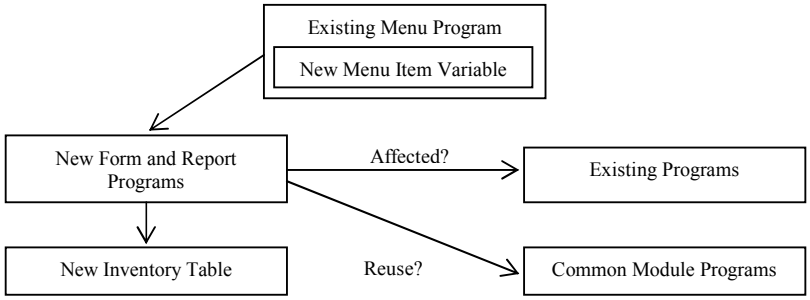

Fig. 4: Information required in the perfective maintenance case

Table 1: The taxonomy of level of Information Abstraction

Level of Abstraction	Level Number	Information Abstracted
Very high	L1	System architecture
High	L2	System-subsystem hierarchy view
	L3	System hierarchy view (module-to-module dependencies or program-to-program dependencies)
Low	L4	Function/procedure/method dependencies within module or inter-module (call graph)
	L5	Function/procedure/method versus variable /data usage dependencies (data flow graph)
Very low	L6	Pseudo code/algorithm

Three important utilities required as observed from the maintenance cases are a browser of physical directory or file corresponds to current running system, search utility to allocate affected components and interface layout generator that specifies variables involved in menu, form or report. The importance of information required or the degree of need is divided into 3 levels: mandatory (M), unnecessary (U) and recommended (R).

Table 2 depicts the connection between each level of information abstraction required and its importance or degree of need for all the three categories of maintenance studied. From the table it deduces that both low level of abstraction (L4 and L5) are mandatory to all the categories prior to changing source codes. In general, adaptive maintenance is recommended to have a higher level of information abstraction compared to the other two categories. Both corrective and perfective categories are mandatory to have program-to-program dependencies provided by the DMG_2 representation. However for the corrective maintenance, the module-to-module dependency is unnecessary to have if the programs to be changed are in the same module. For perfective category it is recommended to have a module-to-module dependency provided by the DMG_1 representation and a system-subsystem hierarchy view (currently is not supported by the tool) if changes involve other sub-systems in order to ensure no redundancy of components.

All the three categories are recommended to have pseudo code or algorithm and interface layout generator. The pseudo code or algorithm abstraction (L6) can enhance program understanding. However, it is not mandatory because most programs can be understood by reading source codes and their comments. As for the interface layout generator, it can be viewed directly by running the system. Information about physical directory or file is only recommended for corrective maintenance but it is mandatory to the other two categories that need to identify the affected or relevant files more precisely. Search result is mandatory for all categories in order to identify affected or relevant components faster and easier.

Table 2: Maintenance categories versus information types

	Abstraction Level						Utility		
	Very High	High		Low		Very Low			
Maintenance Categories	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**Physical Files Browser**	**Search Utility**	**Interface Layout Generator**
Case 1: Corrective Maintenance	U	U	U/M	M	M	R	R	M	R
Case 2: Adaptive Maintenance	R	R	M	M	M	R	M	M	R
Case 3: Perfective Maintenance	U	R	R/M	M	M	R	M	M	R
DMG and its tool support	No	No	DMG_1 and DMG_2	DMG_3 and DMG_4	DMG_5	No	No	Yes	No

Note: M: Mandatory, U: Unnecessary, R: Recommended.

5.0 VIEWING SOFTWARE ARTIFACTS

This section describes how graph representations of the DMG method employed in DocLike Viewer manage to serve different levels of information required by the three maintenance categories: corrective, adaptive and perfective. This study will just discuss the conceptual solutions based on the scenarios or cases given since the interviewed software engineers only narrated the scenario based on their experience in previous maintenance projects. A sample system written in C language was used in order to capture the five types of graph representations required to conceptualise the possible solutions. One limitation of DocLike Viewer prototype tool is that it does not view any flat files or tables from a database because current Rigi C parser that it depends on does not cater to database RE. Thus when discussing about a field of a file or table, this study refers to members of a data "struct" or "enum" in the C language domain.

5.1 Viewing for Corrective Maintenance

For the first maintenance case of corrective maintenance, the type or level of information required as illustrated in Fig. 2 can be viewed via the DMG_2 representation as shown in Fig. 5. For example if Program P1 named "document.c", from the view it shows that this program is dependent on the other two programs. By clicking the "document.c" node, the source code is displayed. Using the "Find" utility in the Source Code Window, the name of related variable for instance "AtDocName_TDocName" can be searched and highlighted (see Fig. 6). By clicking "Find Next" button, users can search for more occurrences of the variable.

In order to determine whether this variable might affect a field of a file or data "struct" such as in C program, the DMG_5 representation can be viewed (see Fig. 7). Clicking the data node will prompt the Source Code Window and display the associated .c or .h file in which the data is defined. Alternatively, by right-clicking the node the details of the components using the data can be listed textually (Fig. 8). In this case, if the size of the data "AtDocName_TDocName" needs to be changed; the other five procedures or functions using the data might be affected too. Thus further checking of how each procedure or function uses the data should be performed.

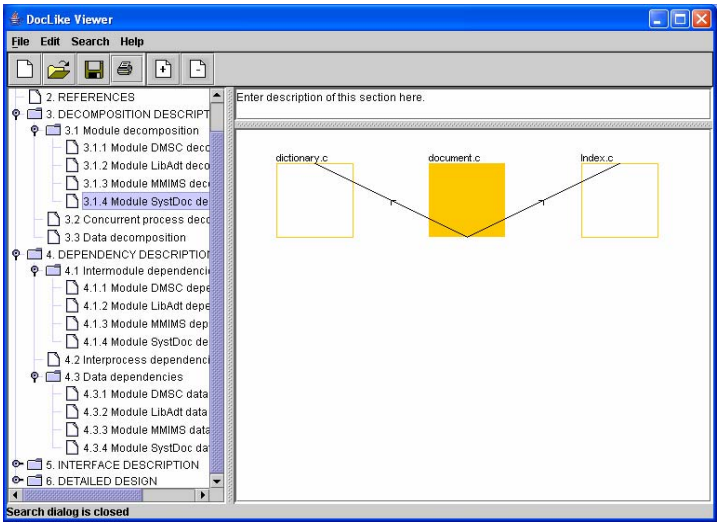

Fig. 5: DMG_2 representation shows the dependencies of the program

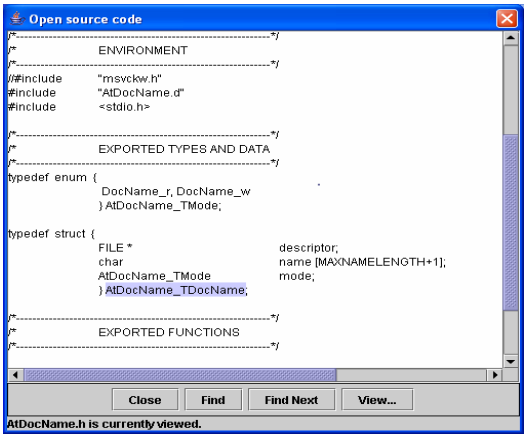

Fig. 6: The searched variable is highlighted in the Source Code Window

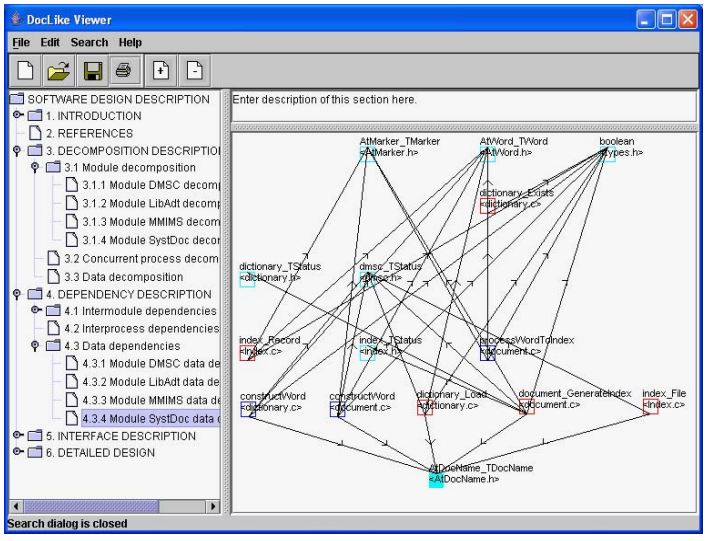

Fig. 7: DMG_5 representation illustrates the usage of the concerned data

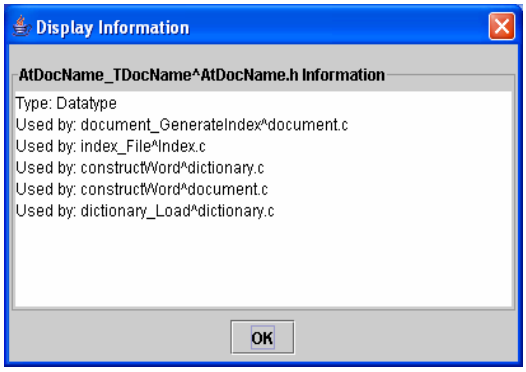

Fig. 8: An alternative way of providing information in a textual-based

5.2 Viewing for Adaptive Maintenance

In the scenario of adaptive maintenance, the level of information required is quite high at the initial stage. But when actual changes have been started software maintainers might perform corrective maintenance in order to correct existing design of the system due to migration of platform made. However, as mentioned earlier, this study just indicates the main category of maintenance for each scenario without discussing the overlapping of the maintenance categories that most probably can occur.

The DMG_1 representation (see Fig. 9) shows a module-to-module system hierarchy view in which the dependencies among the modules are shown. Since the adaptive maintenance case given involves a change in platform, the whole system needs to be studied including if there are any other systems linked to the existing system. Starting from the DMG_1 representation, the following step indicates the programs included in each module via the DMG_2 representation (see the example in previous Fig. 5), followed by determining the procedures and functions that reside in each program by viewing the DMG_3 representation as in Fig. 10. The DMG_4 representation (refer Fig. 11) indicates the inter-module dependencies while the DMG_5 representation (see Fig. 7) indicates the data usage.

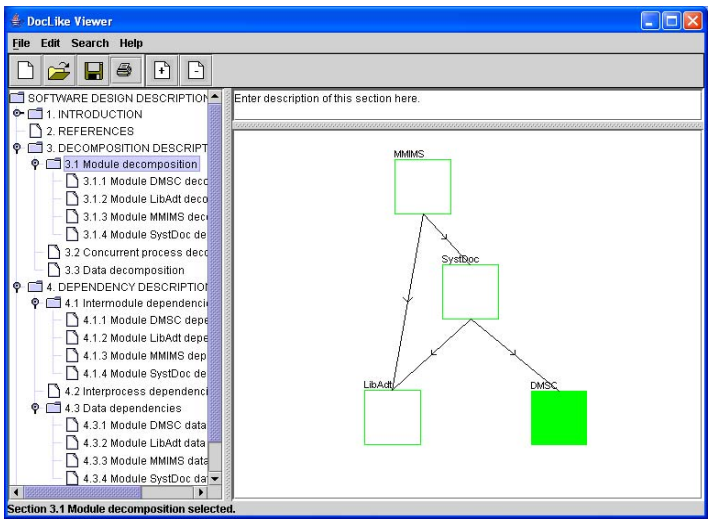

Fig. 9: DMG_1 representation showing module-to-module relationships

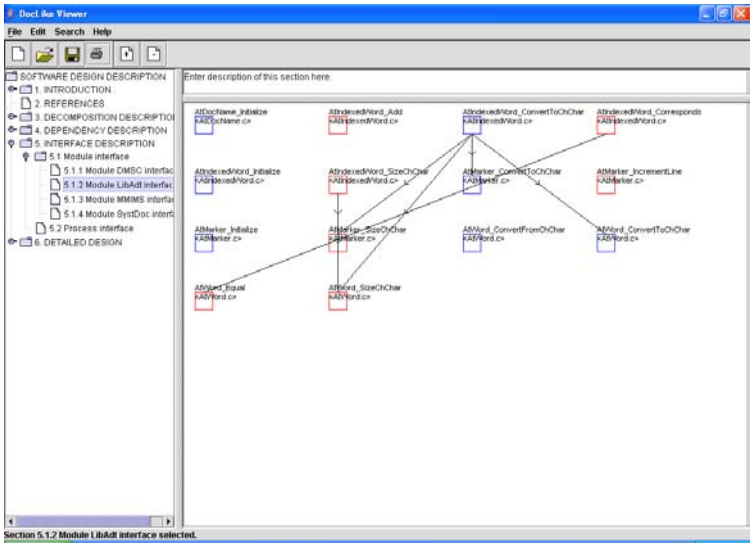

Fig. 10: DMG_3 representation showing dependencies in a particular module

Therefore all the five types of DMG representations are required in order to determine the number of programs in each module and the number of procedures or functions in each program, in order to determine the cost of maintenance changes to be quoted to customers and schedule the time required for the adaptive maintenance project. If the migration of the system involves direct transition without major changes in the actual design, the dependencies shown in DMG_3, DMG_4 and DMG_5 representation will not be so crucial. However, if the DOS-based system needs to undergo massive changes in design, the views that display the dependencies within a module, inter-module and data usage will be critical to ensure that the same business flow and functionalities will be carried out in the new migrated system.

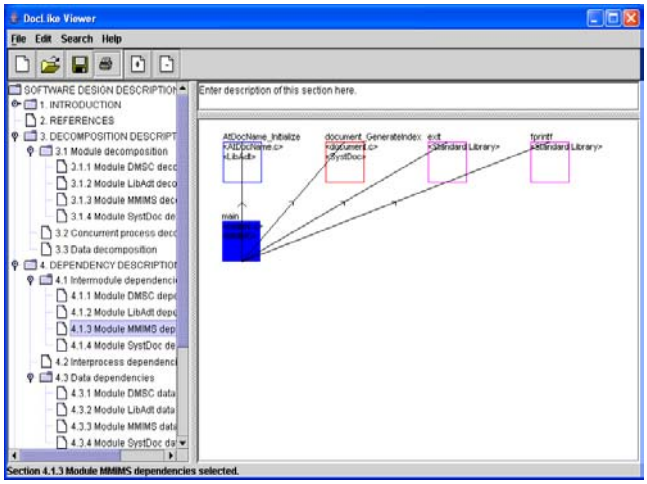

Fig. 11: DMG_4 representation shows inter-module dependencies of a program

5.3 Viewing for Perfective Maintenance

For the third maintenance case that is perfective maintenance, it is conducted to enhance the existing system, different from the corrective and adaptive maintenance. So software maintainers need to study the existing system before they can add a new procedure or function, or program or even module. It is recommended to study the system hierarchy view provided by the DMG_1 representation in order to understand existing modules and then decide whether the new functionalities can be added into any of the existing modules or a new module should be introduced.

The DMG_4 representation of inter-module dependencies (refer Fig. 11) should be viewed to indicate which module has common functionalities of the system that can be re-used by the newly added module. In case a new module should be introduced, the new program or tables added within the new module most probably will have no effect on other modules. However, in the case that the new functionalities require software maintainers to add a new program into the existing module or a new procedure or function into the existing program, DMG_3 and DMG_5 representations are mandatory to be viewed.

6.0 RELATED WORK

Research and studies of SV methods particularly static analysis in RE environments are initially intended to improve program comprehension or cognitive strategies of software maintainers in solving maintenance tasks prior to changing the actual running source codes. Some mostly cited SV tools are CIA [2], Rigi [19], SHriMP [13] in Rigi environment, PBS [9] and SNiFF+ [18]. These tools view software artifacts for different types of languages based on what the parsers of the tools can support. They employ different methods or approaches of SV. CIA employs E-R diagram in its view, Rigi utilises multiple individual windows and SHriMP promotes nested graph approach. SNiFF+ is a commercial tool without much publication on how it works. The view in SNiFF+ employs column-by-column approach. A language-independent SV tool called CodeCrawler [8] within Moose re-engineering environment employs a lightweight approach that integrates metrics in its view.

Some comparative studies include the work of Bellay and Gall [1] that compares four tools (Refine/C, Imagix 4D, Rigi and SNiFF+) by applying them on a real world embedded software system that consists of two languages (C and Assembler) as a case study. The assessment criterias are defined by four categories: analysis, representation, editing or browsing and general capabilities. The study also evaluates the views generated, extensibility and usability of the four tools. It was concluded as "no single tool could be declared as the best". Another study [14] reports a user study, which involved thirty students who were assigned a set of maintenance tasks using three tools (Rigi, SHriMP and SNiFF+). The study shows that the tools did improve users' preferred program understanding strategies. However in some occasions users needed to change their strategies to solve maintenance tasks due to ineffective searching utility provided by the tools. The author's previous work [15] studied four RE tools (Rigi, PBS, SNiFF+ and Logiscope) in terms of the level of information abstraction they could serve to software maintainers.

As a complementary to existing studies, this paper discusses how the proposed DMG method that integrates standardised; module-by-module re-documentation environment can serve different information needs in different maintenance categories, whereby the issue has yet to be highlighted explicitly by any existing work. The method was implemented in DocLike Viewer prototype tool developed by the author of this paper.

7.0 CONCLUSION AND FUTURE WORK

This paper has described three cases of software maintenance that include corrective, adaptive and perfective category. Based on the cases, the levels of information required by software maintainers have been outlined. It is observed that not only do software maintainers require diverse levels of information; the degree of necessity or need towards the different information level is also not exactly the same. This study has also conceptualised how the DMG method utilised in DocLike Viewer prototype tool can provide different levels of information required by software maintainers for different categories of software maintenance. The five types of DMG representations managed to serve the diversity of information needs in the three maintenance cases.

In a nutshell, there is an extremely important relationship between the established categories of software maintenance and the level of information sought by software engineers or maintainers. Although in some cases the degree of necessity towards certain information abstraction is not discrete, in general the difference can still be observed. Therefore this study suggests that SV researchers consider how much their proposed SV methods can serve the information required for different cases of maintenance categories.

The possible future work may include the enhancement of the DMG method to serve more levels of information abstraction (L1, L2 and L6), providing the utility to browse physical directories or files and generating the interface layout. Other future work could also be the incorporation of database RE in order to serve database management systems (DBMS), which are commonly used in industries and also to test the DMG method in a real maintenance

case to determine the relevance of this conceptual solution. It is also expected that DocLike Viewer will be able to have its own parser rather than depending on the C language parser of Rigi tool [19].

REFERENCES

[1] B. Bellay and H. Gall, "A Comparison of Four Reverse Engineering Tools", in *Proceedings of the 4th Working Conference on Reverse Engineering*, USA, IEEE CS Press, 1997, pp. 2-11.

[2] Y. –F. Chen, M. Y. Nishimoto and C. V. Ramamoorthy, "The C Information Abstraction System". *IEEE Transactions on Software Engineering*, Vol. 16 No. 3, 1990, pp. 325-334.

[3] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy". *IEEE Software*, January 1990, pp. 13-17.

[4] M. L. Domsch and S. R. Schach, "A Case Study in Object-oriented Maintenance", in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*. USA: IEEE CS Press: 1999, pp. 346-352.

[5] IEEE Inc., *IEEE Standard Glossary of Software Engineering Terminology*, USA, IEEE Std 610.12-1990, 1991.

[6] M. Kajko-Mattsson, "Preventive Maintenance! Do We Know What It Is?", in *International Conference on Software Maintenance*, USA, IEEE CS Press, 2000, pp. 12-14.

[7] R. Koschke, "Software Visualization in Software Maintenance, Reverse Engineering and Re-engineering: a Research Survey". *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 15, USA, John Wiley & Sons, 2003, pp. 87-109.

[8] M. Lanza, "Lessons Learned in Building a Software Visualization Tool", in *Proceedings of the 7th European Conference On Software Maintenance and Reengineering (CSMR'03)*, USA, IEEE CS Press, 2003, pp. 1-10.

[9] T. III. Parry, H. S. Lee and J. B. Tran, "PBS Tool Demonstration Report on Xfig", in *Proceedings of Seventh Working Conference on Reverse Engineering*, USA, IEEE CS Press, 2000, pp. 200-202.

[10] B. A. Price, R. M. Baecker and I. S. Small, "A Principled Taxonomy of Software Visualization". *Journal of Visual Languages and Computing*, Vol. 4, 1993, pp. 211-266.

[11] C. A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*. New Jersey, Prentice-Hall, 1997.

[12] I. Sommerville, *Software Engineering*. 5th. ed., England, Addison Wesley, 1997.

[13] M. -A. D. Storey, *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD Dissertation, Canada, Simon Fraser University, 1998.

[14] M. –A. D. Storey, K. Wong and H. A. Muller, "How Do Program Understanding Tools Affect How Programmers Understand Programs?", in *Proceedings of the 4th Working Conference on Reverse Engineering*, USA, IEEE CS Press, 1997, pp. 12-21.

[15] S. Sulaiman, N. B. Idris and S. Sahibuddin, "A Comparative Study of Reverse Engineering Tools for Software Maintenance", in *Proceedings of 2nd World Engineering Congress (ICT)*, Malaysia, UPM Press, 2002, pp. 478-483.

[16] S. Sulaiman, N. B. Idris and S. Sahibuddin, "Production and Maintenance of System Documentation: What, Why, When and How Tools Should Support the Practice", in *Proceedings of 9th Asia Pacific Software Engineering Conference (APSEC 2002)*, USA, IEEE CS Press, 2002, pp. 558-567.

[17] S. Sulaiman, N. B. Idris, S Sahibuddin and S. Sulaiman, "Re-documenting, Visualizing and Understanding Software Systems Using DocLike Viewer", in *Proceedings of 10th Asia Pacific Software Engineering Conference (APSEC 2003)*, USA, IEEE CS Press, 2003, pp. 154-163.

[18] Wind River, SNiFF+, *http://www.windriver.com/products/html/sniff.html*, 2004.

[19] K. Wong, S. R. Tilley, H. A. Muller and M. -A. D. Storey, "Structural Redocumentation: A Case Study". *IEEE Software*, Vol. 12, Issue 1, 1995, pp. 46-54.

BIOGRAPHY

Shahida Sulaiman is a lecturer at the School of Computer Sciences, Universiti Sains Malaysia. She received her BSc (Hons.) in Computer Science from Universiti Sains Malaysia in 1997. After graduation she worked as a programmer in a software house for two years where she was mostly involved in software maintenance projects. She obtained both MSc in Real-Time Software Engineering and PhD in Computer Science from Universiti Teknologi Malaysia in 2001 and 2004 respectively. Her field of interest include software visualisation for software maintenance, software documentation, reverse engineering and re-documentation of existing software systems to improve software understanding or program comprehension.