A FRAMEWORK FOR SOFTWARE MAINTENANCE MODEL DEVELOPMENT

Aziz Deraman
Department of Computer Science
Universiti Kebangsaan Malaysia
43600 UKM, Bangi
Selangor Darul Ehsan
Malaysia
Tel: 03-8296150
Fax: 03-8254675
email: ad@pknet.cc.ukm.my

ABSTRACT

The software maintenance process is one of the most costly activities within information system practice. The purpose of this paper is to address some of the difficulties in this process, by proposing a framework for the development of maintenance model. Essential to the software maintenance process is an ability to understand not only the software but the required changes as well. This can only be achieved where the relevant knowledge is available. Based upon this primary requirement, the proposed framework has made the knowledge as its basis for modelling other requirements for software maintenance model development. The framework first identifies the three operational elements, i.e. function, static entity and dynamic entity, required for general software maintenance process. With respect to the knowledge (as part of the dynamic entity components), the framework shows how these three operational elements should behave and interact amongst themselves to deliver a successful software maintenance model.

Keywords: ***Software maintenance model, Software maintenance process, Software knowledge, Change request knowledge, Knowledge-base***

1.0 INTRODUCTION

In many cases of human activity, when a particular activity involves more than a reasonable amount of cost, people start looking for the problems, which contribute to that cost. Software maintenance is no exception to this. When year after year, spending on software maintenance has become increasingly dominant in data processing budgets [1, 2], people have realised that problems in software maintenance need to be identified and resolved. The realisation of this fact took place as far back as the 1970s [3], but software maintenance problems still exist.

After more than two decades of research, an effective software maintenance model has yet to be developed. The reason for this is the lack of a model that is proven viable for general use. However, several suggestions have been made as to how the software maintenance model could be approached. The first approach is to apply a Software Development Life Cycle (SLDC) model to construct a model for software maintenance. This kind of model considers software maintenance as another task of software development. For example, [4] suggests the software maintenance model as a model of the 2nd , 3rd , ..., nth round of development. Basili [5], who argues that software maintenance is a continued development, using the same knowledge, methods and tools used for software development, also supports this view. Later, he further develops his view of the software maintenance model as *reuse-oriented software development*. A more complicated idea of using the SDLC to reflect the software maintenance model is proposed by [6] using his *spiral model*. How effective this type of software maintenance model is still open for discussion. Some have argued that SDLC is not compatible with a software maintenance model. According to Chapin, the use of SDLC will generate an inappropriate expectation set of metric requirements such as effort needed, selection of tools, management support and complexity of the relevant task [7]. Therefore, his preference is that software maintenance should have its own *software maintenance life cycle (SMLC)* model.

Another approach to modelling software maintenance is to use the process of software maintenance itself as a starting point for the model concerned. In this case, a number of proposals have been published with some variations between them. However, we have identified amongst them some common features of the overall software maintenance model found in the literature. These features are:

- *understanding the software,*
- *modifying the software,*
- *revalidating the software.*

Amongst those advocating the above model are Boehm, Martin-McClure, Basili and Chen [1, 8, 2, 9, 35]. Martin-McClure has further refined this model to show what has to be done in each of the maintenance model stages (see Fig. 1). Basili's *reuse maintenance model* is primarily concerned with reusing artifacts of software products for new requirements with appropriate modification. Parikh's model of maintenance focuses upon the identification of maintenance objectives before any other maintenance task is performed [10]. This idea can also be found in Patkau's model in which identifying and specifying the maintenance requirements should be accomplished first [11, 12]. Other maintenance models focus on a specific feature of software maintenance. For example, Sharpley proposed a model for corrective maintenance whereby the problem encountered be first verified and diagnosed before reprogramming and revalidation are carried out [13].

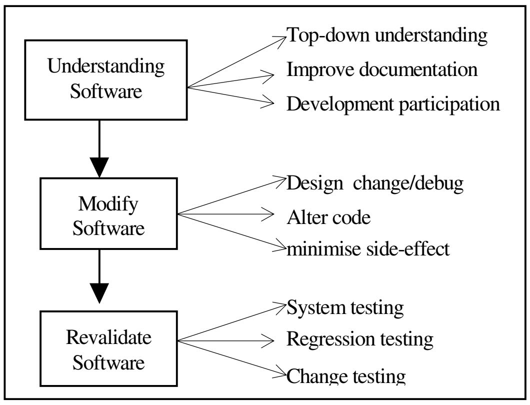


Fig. 1: A software maintenance model [8]

Another important feature of the software maintenance model is the aspect of change impact analysis. Freedman has discussed at a great length the potential side effects of making changes to software [14]. In this case, Martin-McClure, Patkou and Yau [8, 11, 15] have suggested some analyses of the ripple effect of making changes to software. Martin-McClure considers minimising ripple effect as a second objective of changing program code. They urged that the code must be fully examined beginning with the module sharing global variables or common routines with this module. This is specifically important for module that is tightly coupled [16].

From the above discussion, it can be shown that existing software maintenance practices require a more complete model to represent various needs during software maintenance model. These needs not only to respond to the process of software understanding (as has been addressed by [9]) but also to cover other problematic areas such as the intra-maintenance communication [17] and software maintenance resources management [18].

Therefore, it is the aim of this paper to provide a sound framework for the development of a practical software maintenance model within the chosen environment. In section 2, definition that underlies the proposed framework is presented and an explanation of several concepts pertaining to the framework is also given. The proposal of the framework that can be used as a basis of the development of software maintenance model for the chosen environment is outlined in section 3. Finally, this paper is concluded with a summary of the contribution made from the research.

2.0 DEFINITION

Software maintenance model development is not a trivial task where a thorough consideration ought to be made of its requirement as well as its long-term survival. In recognising these needs, a framework is required prior to the development of a relevant software maintenance model. To ensure this framework can sustain in a fragile environment such as software maintenance environment, the following definitions are applied:

Definition 1:

*For any software maintenance practice, the operational elements involved can be explicitly classified into three classes: **Function, Dynamic Entity and Static Entity**.*

Explanation
Let E be a set of operational elements required within a software maintenance process. Therefore,

E = {Function, Dynamic Entity, Static Entity}.

• *Function* is defined as an activity to be accomplished during the course of maintaining a software system.

• *Dynamic Entity* is defined as an entity within software maintenance process that will act as an argument for the function element. The state of this entity will change when the appropriate function is applied to it.

• *Static Entity* is defined as an entity within the software maintenance process that acts as the agent to execute a related function.

The operational elements involved in software maintenance must be considered so that all aspects of software maintenance process can easily be addressed. As has been defined, *Function* should be explicitly identified within the software maintenance process so that each of this function can be effectively performed. With each required function is clearly defined, requirements for implementing specific function can adequately be allocated. Furthermore, any failure that occurs while maintaining a software system can easily be traced based upon the identified function and therefore appropriate action could be taken.

To support various activities of the software maintenance process, several functions have been recognised. These functions cover a wide spectrum of existing software maintenance practice such as acquiring software knowledge, and handling a change request. The following is the definition of those functions as a basis for the proposed framework (to be discussed in the next section):

L: a '*Linguistic Function*' is used to map a change request expression from one state into another;

I: an '*Implementation Function'* is used to modify the existing software system to fulfil a particular need for change;

B: a '*Backtracking Function'* is used to check for similarity of a new change request against the old one;

A: an '*Abstraction Function'* is used to abstract or capture a set of information about software (or software knowledge) from a source code;

H: a '*Human Interaction Function'* is used to communicate with the captured software knowledge as well as to modify the knowledge.

Therefore,

Function = {L, I, B, A, H}

When function is applied in software maintenance, the state of a relevant dynamic entity will change. This entity needs to be defined clearly because software maintenance process may involve various types of activities which may also require different types of maintenance. As a consequence, the affected dynamic entity may also affect other dynamic entities that must be taken into account as well. To fulfil this requirement, we have identified the following as dynamic entities:

Therefore,

Function = {L, I, B, A, H}

When function is applied in software maintenance, the state of a relevant dynamic entity will change. This entity needs to be defined clearly because software maintenance process may involve various types of activities which may also require different types of maintenance. As a consequence, the affected dynamic entity may also affect other dynamic entities that must be taken into account as well. To fulfil this requirement, we have identified the following as dynamic entities:

S_i: denotes an application system at the state of time i where i=0,1,2,...,n.

$C_{j,t}$: denotes the j^{th} change request is submitted at the state of time i (where i always has a value of j-1), and the change request is in the state of l expression. Here, t=0,1,2,...,m where if $t \rightarrow 0$, the change request is in the form close to a user's expression, and if $t \rightarrow m$, the change request is in the form close to the maintainer's expression.

$K_{i,l}$: denotes software knowledge K at the state of time i where i=0,1,2,...,n and at the detail level t where t=0,1,2,..,m. Here, when $t \rightarrow 0$, it shows that knowledge is in the form closed to a source code and as the value of l increases towards m, the software knowledge K will gradually change its form into a higher level of interpretation element as perceived by a maintainer (examples of these elements are data and control flow, module definition and I/O structures).

Therefore, a dynamic entity set can be summarized as:

Dynamic Entity = $\{S_i, C_{j,t}, K_{i,t}\}$

Finally, this definition has explicitly identified the need for a static entity to support software maintenance process. Static entity has been defined to act as an agent for executing a particular function. Amongst the agents required in general software maintenance process are user, maintainer, management and implementor. Therefore, a static entity set can be summarised as:

Static Entity = {User, Maintainer, Management, Implementor}

Definition 2:

In order to facilitate a functional interaction amongst the operational elements within software maintenance practice, there must exist a single central point of reference whereby it can be referred to by all elements.

Amongst the persistent software maintenance problems are that of '*information gap'* [19] and that of '*communication breakdown'* [17] which could be generalised as '*intra-maintenance communication'* problems. Therefore, this definition has highlighted the need for a specific medium to facilitate this intra-maintenance communication so that the adopted software maintenance model delivers the expected result. For the framework proposed in this paper, software maintenance knowledge has been chosen as the sole candidate to represent the central point of reference.

Definition 3:

The only feasible approach to enforce software maintenance knowledge as a central point of reference is through knowledge-based representation approach.

Software maintenance knowledge is largely scattered across the software maintenance environment and therefore involves difficulties in using that knowledge [14, 20, 21]. This definition has made a clear requirement for the scattered software knowledge to be gathered and represented in a knowledge-base so that the knowledge is more organised and centralised. Many authors (for example, see [22, 23, 24, 25, 26], have addressed the importance of a knowledge-based approach to support software maintenance. Therefore, it is the most appropriate to use a knowledge-base as a tool for representing a central point of reference to facilitate intra-maintenance communication.

Definition 4:

Computer aided support for software maintenance is inevitable.

The importance of computer aided software maintenance tools for software maintenance is as important as CASE tools that generally aims to make the practice of software development more reliable and productive [27]. For example, it is costly and time consuming to manually generate appropriate knowledge from operational source code [28]. Therefore, this work must be supported by a computer-aided tool [29]. The discussion of various types of software maintenance tools required for software maintenance can be found in [30]. Therefore, this definition has attempted to include the integration of a computer-aided support for effective software maintenance model.

3.0 A FRAMEWORK FOR SOFTWARE MAINTENANCE MODEL

The heart to the framework is the establishment of a central reference point for the various entities involved in the software maintenance model. Software maintenance knowledge has been chosen as the sole candidate to represent this central reference point which comprises of two main knowledge components.

The first component is *software knowledge*. As has been stated frequently in the literature, the available software knowledge is not always reliable except for the operational source code [31, 32, 33, 34]. However, the source code alone is practically difficult to be used as reference point for software knowledge. Therefore, the framework suggests that a higher-level of software knowledge must be derived from the source code. The derived knowledge can be further enhanced to include some higher semantic interpretation.

The other component is *change knowledge*. This knowledge is gradually established as the software system evolves and there are requests for change. In this framework, all the change knowledge must be kept historically for various reasons. One of the main reasons is that the historical knowledge can be used as a reference point for a new change request before it is submitted and hence authorised. The framework also places a greater emphasis on the need for all the changes to be clearly visualised in software knowledge. To achieve this, the new software knowledge, which originated from the implemented changes, once again has to derive from the affected source code. In this way, the reliability and integrity of the software maintenance knowledge is maintained throughout the life of software system.

Considered to be one of the problematic areas in software maintenance process but frequently ignored is the process of handling a change request. As this will involve much of the maintenance personnel effort, the framework proposed here has made this process more visible to every entity involved. In this case, all the changes are submitted according to a prescribed format whereby the terminology used must conform to the defined software knowledge. To further enhance understanding of the required changes, the personnel involved are allowed to iteratively refine the change descriptions until a satisfactory level is achieved. Improvement in communication amongst maintenance personnel is made possible by enforcing a formalism for change request descriptions. This is done gradually as the intended change approaches the implementation stage. In many of these activities, not only previous change requests can be examined, but the available software knowledge can also be interrogated to finally produce a better-formalised change request description.

Finally, a CASM (Computer-Aided Software Maintenance) toolset is required to complete the proposed framework. For example, the software knowledge derivation is unlikely to be possible without some degree of automation. Similarly, software maintenance knowledge must be strongly represented and supported by an effective retrieval tool so that knowledge interrogation is fruitful.

3.1 Software Knowledge

In this framework, it is assumed that initially the only reliable source of software knowledge is a set of source code programs. Using the given definition, this software knowledge is represented as $K_{0,t=P}$, which is in the form of a programming language L_P (that is why a special value for t is given). Therefore we can apply an *Abstraction* function A to transform this low-level knowledge into a higher one and can be shown as:

$$A (K_{0,P}) \Rightarrow K_{0,t} \qquad ... \quad (1)$$

The *Abstraction* function A is considered to have a limited power for generating a precise and comprehensive software knowledge since it is meant to be fully automatic. Therefore, the resultant knowledge still lacks semantic interpretation. To enrich the knowledge $K_{0,t}$ with higher level interpretation, we use a *Human Interaction* function H to complete the transformation. In this case, H will provide some high level interaction whereby suggestions of the required knowledge are made to the maintainer as well as to modify relevant knowledge as instructed again by a maintainer. Thus,

$$H (K_{0,t}) \Rightarrow K_{0,t+1} \qquad ... \quad (2)$$

This process is an iterative one and can be repeated until satisfactory level of knowledge is acquired. At this stage we can see that $t \rightarrow m$, therefore,

$$H(K_{0,t+p-1}) \Rightarrow K_{0,t+p}, \quad \text{where } t+p \rightarrow m \quad \ldots \quad (3)$$

Furthermore, when a software system is in the state i (S_i), we will have the relevant software knowledge $K_{i,t}$. However, when the (i+1)th change request is implemented, the knowledge $K_{i,1}$ no longer represents the true software knowledge at the state (i+1), i.e. S_{i+1}. Therefore, this knowledge at the state (i+1) is presented as $K_{i+1,\delta t}$, i.e:

$$K_{i+1,\delta t} = K_{i,t} \quad \ldots \quad (4)$$

$K_{i+1,\delta t}$ has to be transformed into the true software knowledge of the existing system version (i+1). Therefore, the same *Abstraction* function is used to automatically abstract some of the knowledge resided in the new version of the source code.

$$A(K_{i+1,\delta t}) \Rightarrow K_{i+1,t} \quad \ldots \quad (5)$$

Similarly, equation (2) can be applied repeatedly to the knowledge produced by equation (5) so that higher level interpretation of the change can be reflected in the software knowledge. Thus,

$$\begin{gathered} H(K_{i+1,t}) \Rightarrow K_{i+1,t+1} \\ \vdots \quad \ldots \quad (6) \\ H(K_{i+1,t+p-1}) \Rightarrow K_{i+1,t+p}, \\ \text{where } t+p \rightarrow m. \end{gathered}$$

3.2 Change Request Knowledge

Within the proposed framework, a change request knowledge is assumed to be an important part of overall software maintenance knowledge. Therefore, the change request knowledge has to be derived to the extent that it is usable for implementation as well as a reference point for future maintenance purposes. It is our intention here to show the modelling process of a change request evolution.

In a typical software maintenance practice, an implemented change request can cause a new version of software system to be produced. However, in an optimised situation, one can generally gather several change requests into one implementation unit and hence will also produce one new version of software system. Therefore, two possible relationships can be established as shown in Fig. 2.

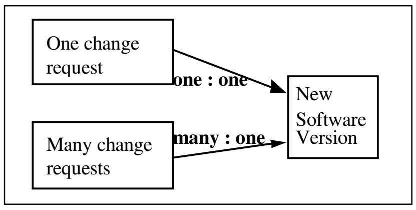


Fig. 2: Relationship between Change Request and Software Version

To accommodate this new requirement (relationship of *Many Change Requests → One New Software Version*), we have to extend the definition of a change request representation by considering the jth change request is as a set of different change requests, i.e.:

$$\begin{aligned} C_{j,t} &= \sum_{k=1}^{r} C_{j(k),t} \\ &\cong \{C_{j(1),t} + C_{j(2),t} + \ldots + C_{j(r),t}\} \quad \ldots \quad (7) \end{aligned}$$

(r will have a minimum value of 1)

Initially, a change request is in a user-oriented form and can be represented as $C_{j(k),t=0}$. However, users can use a given *Linguistic* function L which facilitates a mapping process of a change request into one step closer to the software knowledge domain or maintainer's language. This process can be repeatedly applied as follows:

$$\begin{gathered} L(C_{j(k),t}) \Rightarrow C_{j(k),t+1} \\ \vdots \quad \ldots \quad (8) \\ L(C_{j(k),t+p-1}) \Rightarrow C_{j(k),t+p} \end{gathered}$$

Satisfied with his change request expression, a user can now submit the change request. The maintainer then applies a similar process of invoking a *Linguistic* function L to map the change request description more closely to his language (in terms of software knowledge domain). However, within this process, a user or a maintainer can also alternatively use the L function as a medium for them to communicate if there are queries which arise that need to be clarified. Therefore,

$$\begin{gathered} L(C_{j(k),t+p}) \Rightarrow C_{j(k),t+p+1} \\ \vdots \quad \ldots \quad (9) \\ L(C_{j(k),t+q-1}) \Rightarrow C_{j(k),t+q}, \\ \text{where } t+q \rightarrow m. \end{gathered}$$

The above transformations are also true when (in practice) a maintainer issues a change request for example. Perhaps, the maintainer will express the intended change request straight into his term, i.e. $C_{j(k),t}$ where $t \rightarrow m$. Therefore, in this case the function L will be less useful.

Sometimes, a submitted change request can be found not to be unique. Therefore, this framework provides two stages of checking. First it checks the change request within a user's domain. Having an intention to submit a change request, users can always check the intended change request against previously submitted requests. This process will reveal whether the change request is redundant, partly redundant or a new one. By definition, this process is accomplished by means of a *Backtracking* function B. If we have the i(k)th change request, then it must be checked against all the other change requests, i.e.:

$C_{i(k),t}$ is checked against
$\{C_{i(h),t}, C_{i-1(h),t}, \ldots, C_{2(h),t}, C_{1(h),t}\}$

for all h,
where $i(h) \neq ik$).

Using a function B, this process can be represented as:

$$B(C_{i(k),t}, C_{j(h),t}) = \begin{cases} -1 & \text{if } C_{i(k),t} = C_{j(h),t}\,; \\ 0 & \text{if } C_{i(k),t} \approx C_{j(h),t}\,; \\ +1 & \text{if } C_{i(k),t} \neq C_{j(h),t}\,, \end{cases} \quad \ldots (10)$$

for all h,
where $h \neq k$, for $i = j$
and $j = i, i-1, \ldots, 1$.

The above equation shows that the function B will have a value of -1 if the ith change request is similar to the previous change request. B is 0 when some of the change requests in the set of ith change request are found similar to the previous change requests and B will have a value of +1 if the change request is absolutely a new one. Equation (11) is used to show the checking against one of the previous change request. To represent the whole checking process, i.e. for j, (j-1),...,2,1, then, the equation (10) can be generalised using a product notation as:

$$\prod_{j=i}^{1} B\,(C_{i(k),t}\ C_{j(h),t}), \qquad \text{for all } h, \text{ where } h \neq k, \text{ for } i = j \qquad \ldots \quad (11)$$

Equation (11) will have a negative value if the ith change request is actually similar to one of the previous change request. A change request has to be resubmitted if equation (11) has a zero value because the change request is partly similar with previous change requests. Only a positive value of equation (11) will indicate that the i(k)th change request is unique.

The second check is made within the maintainer's domain. Here, the maintainer will ensure the uniqueness of a change request by comparing it with the previous change requests. As a result, the intended change request may be found totally or partially redundant or probably a unique one. The checking process shown in (10) and (11) is also applicable within a maintainer's domain where $t \rightarrow m$.

Finally, only the change request that passed the two checks will be used to modify code. The various details of the change request produced by the function L will actually be used during actual code modification. Within our framework, this process is accomplished by a function I and can be shown as:

$$I\,(S_{i-1}, \sum_{h=1}^{r} \sum_{t=0}^{m} C_{i(h),t}) \Rightarrow S_i \qquad \ldots \quad (12)$$

The first summation is used to collect all ith change request elements and the second summation is used to show the various details of the change request specification for system S.

3.3 Dependency of Knowledge Components

During the software maintenance process, support from both software and change request knowledge is required, with each knowledge component being dependent on one another. Within the given framework, the existing software knowledge $K_{i,\delta t}$ can be used to facilitate a process of expressing a change request $C_{i(k),t}$ by a user as well as a maintainer. In this case both of them will use the software knowledge $K_{i,\delta t}$ as a repository for supplying them with the appropriate knowledge that relevant for the change request description. Using the given notation, this process can be shown as:

$$\begin{tikzcd} S_{i-1} \Leftrightarrow C_{i(h),t \rightarrow 0} \arrow[r] & \boxed{L} \arrow[r] & C_{i(h),t \rightarrow m} \\ & K_{i,\delta t} \arrow[u, Rightarrow] & \end{tikzcd} \quad \ldots\ldots(13)$$

$\boxed{L} \cong$ Iterative function

From the opposite perspective, the execution of an *Implementation* function I for a change request $C_{i,1}$, in turn will trigger the execution of other functions to reflect the implemented change in the software knowledge $K_{i,\delta t}$. This repository is then will change into a new state of $K_{i,t}$

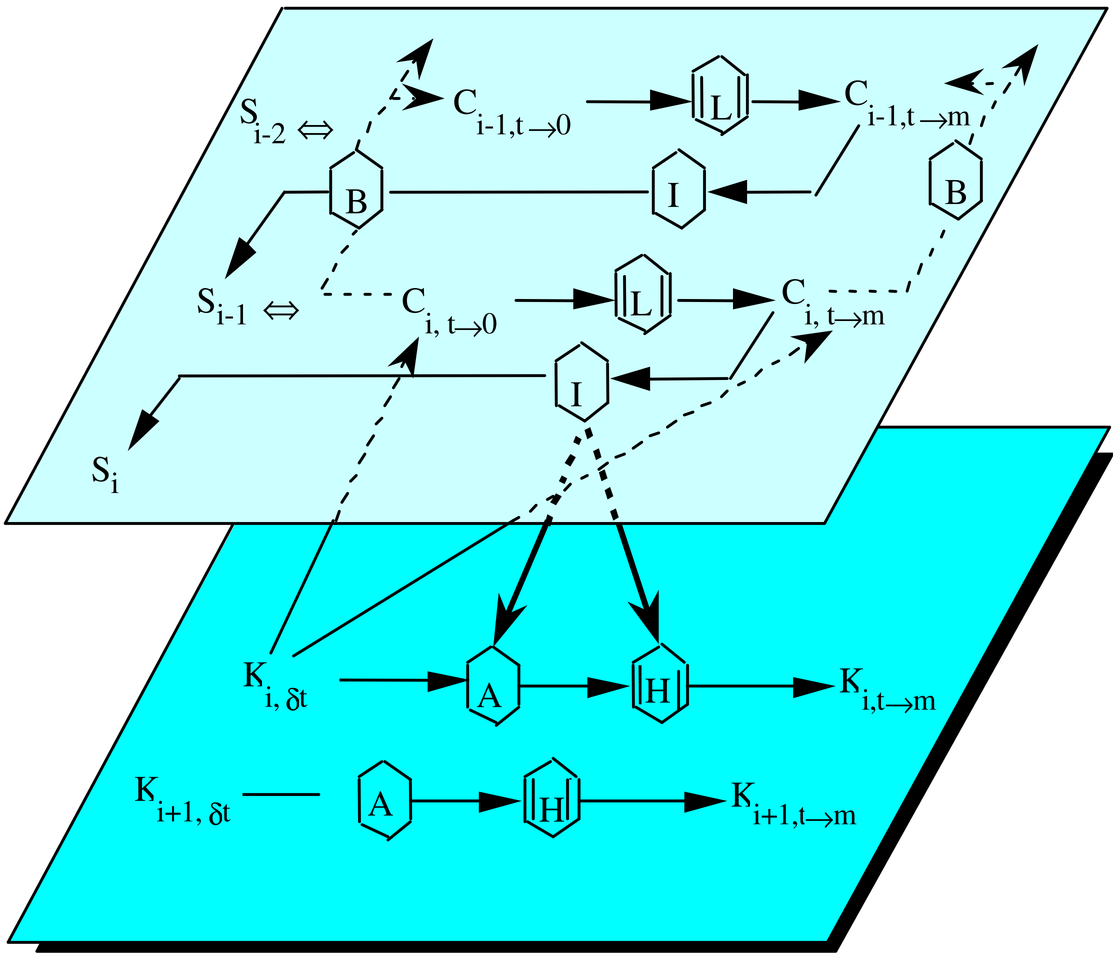
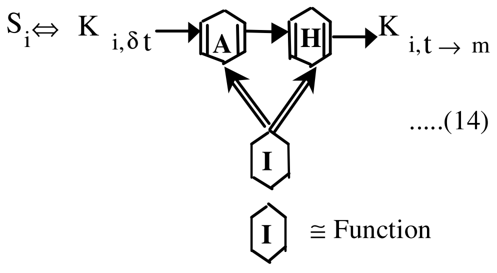


Fig. 3: Functional interaction within a Framework of Software Maintenance Model

Here, both *Abstraction* and *Human Interaction* (A and H) functions will be invoked and is shown as:



.....(14)

Therefore, by applying representation (13) and (14) into a consolidated model, we can have an overall graphical view of the proposed framework for software maintenance model. This is shown in Fig. 3.

4.0 CONCLUSION

In this paper we have argued that one of the recurring software maintenance problems is that of inadequate support of proper software maintenance model. To solve this problem we have presented a framework for software maintenance model development. Central to this framework is the creation of a software maintenance knowledge-base as a common point of reference for software maintenance activities. Two main components of this knowledge are identified, i.e. software knowledge and change request knowledge. Therefore, any committed software change must be confirmed and validated within the knowledge-base whereby a better change management control can be realised. Implemented change is also controlled over the knowledge-base so that integrity and reliability of the knowledge is guaranteed. Any reference to the knowledge-base will reflect the current state of the software.

For a practical application of the model, its strength very much relies on how far we can automate the process. For example, implementation of the knowledge-base requires a fast and reliable algorithm and this will increase the cost of the development. Therefore, this model is economically applicable only for a big MIS application. Nevertheless, with the proposed framework, the process of deriving an appropriate software maintenance model within the chosen environment could be made easier.

REFERENCES

[1] B. W. Boehm, “Software Engineering”, *IEEE Transaction on Computers*, Vol. 25, December 1976, pp. 1226-1241.

[2] G. Parikh, “The World of Software Maintenance”, in *Techniques of Program and System Maintenance*, G. Parikh (Ed.), Little, Brown and Company, Boston, MA, 1982, pp. 9-13.

[3] E. B. Swanson, “The Dimension of Maintenance”, *Proceeding of 2nd International Conference on Software Engineering*, San Francisco, 1976, pp. 492-497.

[4] J. R. McKee, “Maintenance as a Function of Design”, *AFIPS National Conference Proceeding*, Vol. 53, 1984, pp. 187-193.

[5] V. R. Basili, “Viewing Maintenance As Reuse-Oriented Software Development”, *IEEE Software*, January 1990, pp. 19-25.

[6] B. W. Boehm, “A Spiral Model of Software Development and Maintenance”, *IEEE Computer*, Vol. 21, No. 5, May 1988, pp. 61-72.

[7] N. Chapin, “Software Maintenance Life Cycle”, *Proc. of Conference on Software Maintenance*, Computer Soc. Press, IEEE, New York, 1988, pp. 6-13.

[8] J. Martin and C. L. McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, 1983.

[9] S. Chen, K. G. Heisler, W. T. Tsai, X. Chen and E. Leung, “A Model for Assembly Program Maintenance”, *Software Maintenance: Research and Practice*, 1990, pp. 3-32.

[10] G. Parikh, “Structured Maintenance: The Warnierr/Orr Way”, *Computerworld: IN DEPTH Section,* 1981.

[11] B. H. Patkau, *A Foundation for Software Maintenance*, Ph.D. Thesis, Department of Computer Science, University of Toronto, December 1983.

[12] S. L. Pfleeger and S. A. Bohner, “A Framework for Software Maintenance Metrics”, *Proceeding of Conference on Software Maintenance* - Computer Soc. Press, IEEE, New York, 1990, pp. 320-327.

[13] Sharpley, “Software Maintenance Planning for Embedded Computer Systems”, *Proceeding of the IEEE COMPSAC*, November 1977, pp. 520-526.

[14] D. P. Freedman and G. M. Weinberg, “A Checklist for Potential Side Effect of a Maintenance Change”, *Techniques of Program and System Maintenance*, G. Parikh (ed.), Ethotech Inc., 1980, pp. 61-68.

[15] S. S. Yau and J. S. Collofello, “Some Stability Measures for Software Maintenance”, *IEEE Transaction on Software Engineering*, Vol. 6, No. 6, November 1980.

[16] W. Stevens, G. Meyer and L. Constantine, “Structured Design”, *IBM System Journal*, Vol. 13, No. 3, 1974, pp. 115-139.

[17] P. J. Layzell and L. Macaulay, “An Investigation into Software Maintenance - Perceptions and Practices”, *Proc. of Conference on Software Maintenance* - Computer Soc. Press, IEEE, New York, 1990, pp. 130-140.

[18] R. B. Grady, “Measuring and Managing Software Maintenance”, *IEEE Software*, Vol. 4, No. 9, September 1987.

[19] W. M. Osborne, “Building and Sustaining Software Maintainability”, *Proceeding of Conference on Software Maintenance -1987*, Computer Soc. Press, IEEE, New York, 1988, pp. 13-27.

[20] N. F. Schneidewind, “Quality Metrics Standard Applied to Software Maintenance”, *Proceeding on Computer Standards Conference*, Addendum, IEEE Computer Soc., May 1986.

[21] N. F. Schneidewind, “The State of Software Maintenance”, *Transaction on Software Engineering*, Vol. 13, No. 3, March 1987, pp. 303-310.

[22] F. J. Lukey, “Understanding and Debugging Programs”, *International Journal of Man-Machine Studies*, Vol. 12, 1980, pp. 189-202.

[23] E. Soloway and W. L. Johnson, “PROUST: Knowledge-Based Program Understanding”, *IEEE Transaction on Software Engineering*, Vol. 11, No. 3, March 1985, pp. 267-275.

[24] N. W. Wilde and S. M. Thebaut, “The Maintenance Assistant: Work in Progress”, *SERC-TR-10-F*, CIS Department, University of Florida, September 1987.

[25] M. T. Harandi and J. Q. Ning, “Knowledge-Based Program Analysis”, *IEEE Software*, January 1990, pp. 74-81.

[26] P. Benedusi, V. Benvenuto and M. G. Caporaso, "Maintenance and Prototyping at the Entity-Relationship Level: a Knowledge-Based Support", *Proceeding of Conference on Software Maintenance* - Computer Soc. Press, IEEE, New York, 1990, pp. 161-169.

[27] C. L. McClure, *CASE is Office Automation*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[28] G. Richardson and E. D. Hodil, "Redocumentation: Addressing the Maintenance Legacy", *AFIPS 1984 National Computer Proceedings*, AFIPS Press, Arlington, Virginia, May 1984, pp. 203-208.

[29] P. Antonini, P. Benedusi, G. Cantone and A. Cimitile, "Maintenance and Reverse Engineering: Low-Level Design Document Production and Improvement", *Proc. of Conference on Software Maintenance* - Computer Soc. Press, IEEE, New York, 1987, pp. 91-100.

[30] A. B. Deraman, and P. J. Layzell, "Computer-Aided Software Maintenance: A Classification and Analysis", *Malaysian Journal of Computer Science*, Vol. 6, December 1993, pp. 21-42.

[31] H. M. Sneed, "SoftDoc - A System for Automated Software Static Analysis and Documentation", *Proc. of ACM Workshop on Measurement and Evaluation of Software Quality, (ACM Sigmetrics)*, Vol. 10, No. 1, 1981, pp. 173-178.

[32] C. Wilson and L. J. Osterweil "OMEGA - A Data Flow Analysis Tool for the C Programming Language", *IEEE Transaction on Software Engineering*, Vol. 11, No. 9, September 1985.

[33] D. R. Kuhn, "A Source Code Analyser for Maintenance", *Proc. of Conference on Software Maintenance* - Computer Soc. Press, IEEE, New York, 1987, pp. 176-180.

[34] L. D. Landis, P. M. Hyland, A. L. Gilbert and A. J. Fine, "Documentation in a Software Maintenance Environment", *Proceeding of Conference on Software Maintenance* - Computer Soc. Press, IEEE, New York, 1988, pp. 66-73.

[35] M. K. Khan, M. A. Rashid and W. N. Lo, "A Task-Oriented Software Maintenance Model", *Malaysian Journal of Computer Science*, Vol. 9, No. 2, December 1996, pp. 36-42.

BIOGRAPHY

Aziz Deraman obtained his Master degree in Computer Science from Glasgow University in 1984 and Ph.D in Software Engineering from UMIST in 1992. Currently he is an Associate Professor at the Department of Computer Science and Deputy Director for the Computer Centre, Universiti Kebangsaan Malaysia. His research interests include software maintenance management, IT strategic planning, software testing and temporal DB and multimedia engine development for education.