第6卷第2期 广西科学院学报 Vol.6,No.2
1990年10月 Journal of The Guangxi Academy of Sciences Oct, 1990

以决定为基础的软件开发: 设计和维护

克里斯琴·怀尔德 科特·马里 刘连芳 陈建勋 徐 婷

摘 要

软件维护是由一系列活动组成的: 对所需改变的理解、评价、分析、实现和检验等等。我们为软件维护提出了一种以决定为基础的开发方法。在这个方法中, 根据各种软件实体所涉及的决定把这些实体联系在一起。用决定把问题与结论相连, 给出开发、维护过程中考虑过的各种选择方案, 并对特定的选择做出评价。一个决定, 或者相关的一组决定, 提供了与那个决定有关的软件系统的一个视图, 并且可以让用户获取根据那个决定产生的原程序。这篇文章介绍以决定为基础的超级软件工程工具(D－HyperCase)的设计。D－HyperCase 是以决定为基础的软件开发支撑系统的原型。由于我们正在使用以决定为基础的方法开发这个系统, 所以, 也介绍一下我们使用这种方法的体验。

(刘连芳译)

1990年5月15日收稿

Decision－Based Software Development: Design and Maintenance

Chris Wild Kurt Maly
Lianfang Liu Jann－Shinn Chen Ting Xu

Department of Computer Science
Old Dominion University
Norfolk, VA 23529－0162

ABSTRACT

A software maintenance task comprises the activities of understanding, assessment, analysis, realization and verification of the required changes. We propose a decision－based software development methodology in which the various software objects are related by the decisions in which they are involved. A decision relates a problem to its solution, gives alternate solutions which were considered and justifies the particular solution chosen. A decision, or related set of decisions, provides a view of the software system relevant to that decision and allows the retrieval of source code which resulted from thatdecision. This paper describes the design of D－HyperCase, a prototype decision－based software development support system. Since this system is being developed using the decision based approach, our experiences in decision－based software development are also given.

1 Introduction

The software maintenance activity places a significant cognitive workload on the software engineer. A proposed change could impact any part of the existing system, requiring the software maintainer to be able to understand selected parts of the entire software system in precise detail. The effort curve during software maintenance is significantly different than that during the initial software development [4]. About half of the effort in a maintenance task is spent on understanding the existing system.

The success of a software maintenance task crucially depends on the ability to selectively

This work was sponsored by contract NAS1－18584－41 from NASA Langley Research Center and NAVMASSO, Norfolk, VA

understand those parts of a system which are relevant to the proposed change and to assess the impact of that change on the existing software/hardware/human components of the system. We define the closure[1] of amaintenance task as the minimal set of information which allows the maintainer to perform the task successfully. This information exists in the project documentation base, in the heads of the project personnel or may be inferred from the previous two sources of information. One of the challenges of a software maintenance support system is to capture the required information and to organize it in a fashion which facilitates the process of forming closure. Traditional forms of software documentation suffer both from lack information necessary to understand the relationship between parts of the software system[8]and from the difficulty of accessing the relevant information that is contained there. The content and organization of this documentation is thus crucial to the success of the maintenance task.

Previous studies [11] have suggested the potential value of organizing the documentation of the project around the decision structure of the project. In this paper, we describe a prototype software maintenance support system based onrecording the decision structure, called DhyperCase, which is currently underdevelopment. We believe the decision structure provides a natural decomposition of the system description which can be used to selectively view that description and thus assist in forming the closure of the maintenance task. Since we are using a decision – based approach in the development of the prototype, we describeour experiences with decision – based software development. But first, a short description of decision based software development is given.

2 Decision – Based Software Development

The key to decision based software maintenance is the ability to view the source code through the decisions which led to its creation. It provides alternate means of organizing the source, different from its textual packaging into modules and program files. Potentially, every line of source code must be tied to at least one decision. These decisions will themselves be tied to other decisions through a decision graph providing a path of dependency from the source code to the system requirements. We believe it essential not only to record all decisions made during the design process but to connect these decisions. To assess a maintenance task we would want to know how many lines of code would be impacted if we were to change a decision. We call the lines of source code tied a particular decision a code view. The relationship is bidirectional, to understand a particular line of source code we want to know why it is there, what problem does it solve, and to what requirement can it betraced. In addition, to assist in understanding the system, we wish to depict the behavior of the program to be maintained. Finally, in the modification of the source code, we wish to reuse previously developed code views together withtheir decision history. Given these requirements we postulate the solution shownin Figure 1.
A decision is described by the following four components:

Problem: Describes the problem currently being addressed.

Alternatives: A set of proposed solutions. One or more of these alternatives will be chosen as the proposed solution to the problem. The rejected alternates may represent infeasible solutions, less optimal solutions or solutions which require more resources than were available at the time of the decision.

Solution: The decision to select the proposed solution to the problem.

Justification: An explanation of why the particular solution was chosen and why the other alternates were rejected. A justification may refer to otherd ocumentation for support including formal proof, prototypes, simulation models, constraints, resource limitations or the literature.

The **decision graph** is a set of nodes and arcs. Nodes can be decisions (rectangles) or problems (ovals), arcs are directed links between nodes with the restriction that cycles are not permitted. The Alternatives, Solution and Justifications are associated with the decision node in the graph. Justification links can go from problems or decisions to decisions. Solution links go from problem to decision (only one decision per problem) and from decision to multiple problems. Both problems and decisions can be terminal nodes in the acyclic graph. If a decision (solution) is terminal then it represents a solution to a problem which does not involve any more major decisions. A source code view or a document may be such a terminal solution. If a problem is terminal it represents a requirement from which only justification links emanate.

One of the major objectives in the development of the decision based approach was to be able to represent both the functional and non-functional requirements and their interaction. There are numerous differences between functional and nonfunctional requirements. We have found that the non-functional requirements tend to constrain the form of the solution and help to choose among the alternatives. Thus in the solution of a functional requirement, the justifications typically involve the non-functional requirements. Also decisions invelving non-functional requirements tend to cut across module boundaries (such as those involving severe performance constraints or the style of the user interface or security policies). In addition, the satisfaction of non-functional requirements if usually a matter of degree as opposed to being definitely right or wrong. Recently, we have recognized that management decisions regarding schedule and other resources serve as part of justification for many decisions. In this way, the effect of the dynamics of the process on the final form of the solution can be reflected in the decision structure.

The **source code** is considered as an entity by itself or as a collection of code views. Unlike the disjoint partitioning of source code which results from top-down design, code views may overlap. Overlaps result from the multiple viewpoints inherent in the decision structure. One line of code[2] may exist for several reasons. For instance, the line of code which opens a line printer to print two kinds of reports is justified by the decisions to generate each report (**or justification** – since either decision justifies the line of code). Even if you later decide not to print one of the reports, the line is still justified by the other decision. Overlaps can also occur due to **and justifications**, where several decisions are required to justify a line of code. For example, the line of code to print a date is justified both by the decision to generate the report in which the

date appears and by the decision on the representation of dates in general.

In addition to the decision structure and source code, two more pieces of information are provided. The **behavioral grammar** is a set of productions which describe the externally observable behavior of the program to be maintained. Hence the terminal symbols of the grammar are the user inputs and the system responses (see [10] for more details) . Finally, **documentation** is any other information which might be produced during the design process or during maintenance.This documentation is project specific but is tied into the decision structure which provides its own view of that documentation .

As discussed in section 5, the structure of the decision graph is not unique .

3 Designing D－HyperCase: A Case Study in Decision－Based Design

In order to further develop the decision－based approach to software development and to evaluate its effectiveness for performing software maintenance, we are building a prototype called D－HyperCase. Since we will use this prototype in itsown maintenance, we are recording the decisions made during its development, using a process we refer to as deccision－based software design. Not only is the initial design phase the natural place to capture the decision structure, we felt that many of the activities in initial design involve maintenance (maintenance of requirements, specifications, designs, etc) as the system evolves. In addition, software maintenance involves design as well and will require the recording of the decisions made during software modifications . We wanted to understand the extra burden that documenting and maintaining the decision structure would place on the developers .

Naturally, we were faced with the problem of bootsrapping and documenting D－HyperCase. In section 4. 1 , we describe the incremental , extensible structure of D－HyperCase which makes it an ideal test case to show the utility of a decision－based support system for an evolutionary program. At this point, only a samll portion has been implemented and we are forced to illustrate D－HyperCase using hand－drawn figures and graphs rather than those produced by usingD－HyperCase. It is equally difficult to show the dynamic usage of the decision graph with still pictures . In D－HyperCase , the decision graph is drawn on a canvas which can be viewed through a window which can bescrolled in any dircetion or moved at will to related documentation . Here we can only give cutouts and hint at connectios to other parts of the graph .

In Figure 2, we begin with the root problem to which we can eventually trace every source code line of D－HyperCase. The problem is to maintain[3] (Italicizes words in the text correspond to decisions or problems in the figures.) existing or to－be－designed systems. Among the many paradigms for maintenance we have selected the one where we understand the program, assess the impact of the change, and make the modification.On the basis of the ideas developed in a previous paper[11] (which forms the justification to the first and second decisionsin Figure 2), we have decided to use a methodology of a decision－based support system. Alternatives at this

stage would be, for instance, data flow methods, Your don's method, ect. This support system has a number of functional and non-functional requirements, which are listed below, starting with source on the left and going right until the assessment node. Next comes the problem of how to relate this method to other tools, how to formally describe the method and how to use the description for actual use. The understanding of the what, how, and why of the source code requirement is decomposed into those of understanding the program behavior, the mapping of source code to decisions (and back) and the mapping of decisions to problems (and back). These last requirements are terminal and are only used as justifications of other decisions in the graph. In this case, they justify the behavioral grammar, the decision to have views consisting of source code lines, and to have an acyclic graph of decisions and problems. The remaining problems will lead to requirements originating from the problems of :

- *reusing* design, decisions, and code
- supporting the *design* process
- *measuring* the success of the task
- providing *ease* of use through a user friendly interface with adequate performance
- maintaining the *consistency* between documentation and source code
- *assessing* the maintenance task

The problem of the relation of this methodology to other methodologies is solved by providing a simple interface to compatible methodologies. This allows us to provide access to other methodologies where appropriate. Other choices would be to have no interface, or to fully intergrate these methodologies into the decision-based one. This decision in turn will justify a decision in the process of implementing a layered architecture for D-HyperCase (described in section 4. 1).

The description problem is decomposed in a theory problem and a content problem. The subgraph developed under this node leads to the specification of a decision-base software maintenance system. The next problem is how to use the formal description. The problem of using the proposed method is solved by building a computer supported environment. Other choices are use paper-and-pencil support (which is what we are doing in the bootstrapping process), using existing tools with only some principles incorporated, e. g., record decisions using emacs or by an existing hypertex tool to link together the related pieces of documentation (Our initial demonstration involved the hypertext system KMS [1] to tie together decisions and problems). Further development under this decision leads to the decisions involved in the design of D-HyperCase.

4 D－HyperCase

4.1 D－HyperCase Abstract Machine

The project document base was defined in section 1 as the sum total of the recorded information about the project. The decision structurer ecords the decisions made, alternatives considered and justifications for the solution chosen. As importantly, the decision structure provides a backbone with which to organize and access the other forms of software documentation. The ability to selectively view related parts of a set of documentation from different viewpoints is closely related to the work on hypertext [5]. In fact some of the early demonstrations of the decision based approach were done using the KMS hypermedia system [1]. The two limitations to current hypertext system which were experienced were the inability to enforce a certain minimal structure on the set of documentation and the difficulty of integrating project related tools. The way we have chosen to introduce structure in the network of information comprising the project document base is to base the design on a set of typed objects and typed links. Thus a problem object always contains a link to some decision object. The set of objects and links is user extensible. Each of the objects is managed by its own set of tools which are bound to the object at the time of object type definition. Each object must haveas a minimum a display tool and an edit tool.

Since the ability to access the various objects in the project document base is a crucial part of our approach, we have chosen to build D－HyperCase on top of an Abstract HyperLink Machine(AHM) as shown in Figure 3. This appproach is similar to that taken in the design of Neptune [2]. The AHM maintains a data base of all the object types and their associated tools and the names of all objects according to type. Using this information, it can invoke the appropriate tool to display or edit any object (thus linking to that object). It also maintains a list of objects which the user has seen this session in order. This allows the user to retrace his steps through the document base.

The AHM supports two types of links: **structured links and name links.** A structured link is defined btween two typed objects, one of which is the "from" object and the other is the "to" object. Whenever an object at the "from" end of the link is defined, it automatically inherits all the structured links for which its type is the "from" object. These links define the minimal structure which must exist for the new object. Thus every object of type "problem" must have a link to a decision. On the other hand, name links provide access to objects using their name. These links are implicit and need not be represented in the AHM object base. Any object can have an arbitrary number of name links.

There is an interface to these facilities which can be accessed by any tool building on this layer. This minimal tool set provided on the Abstract HyperLink Machine layer consists of a modified emacs text editor [9] and a modified version of the Structured Graphical Knowledge Base System (SGKBS) [7] under development at ODU. These tools support generic objects of the su-

per class "text" and "graph" respectively. The minimal tool set defines one instantiation of a HyperMedia system based on AHM. D－HyperCase is then built on this layer. Further Project specified tools can be built on top of D－HyperCase.

D－HyperCase is defined in terms of its objects and associated tools. These are described in section 4.2. A D－HyperCase tool can access the AHM directly through its tool interface. Such tools are called integrated tools and allow the user to directly traverse HyperLinks from within the tool (by whatever interface the tool writer wishes to provide). In addition, D－HyperCase allows the inclusion of non－integrated tools. Thus the set of tools available in D－HyperCase can be expanded to include commercially available tools. These tools would not have a direct interface into AHM. However, the user could access AHM facilities through the D－HyperCase background menu as described below. In general, the D－HyperCase tools allow display and editing of the software objects defined in the project document base in ways appropriate for that object. For example, the source code editor understands the syntax of the source language (bascause it is a structured language editor), the relationship between source code views and decisions and allows the user to access related information through the HyperLink facility. The editor for source code objects is built upon the modified emacs editor provided in the minimal tool setbecause source code is a subclass of the class of text objects.

4.2 D－HyperCase: User's Perspective

We now describe D－HyperCase from the user's perspective. Figure 4 shows the initial screen shown to the user upon entering D－HyperCase. This screen provides an overview of the D－HyperCase System and is an object of type "document figure." Through the use of the hyperlink facility, this screen provides access to other screens which furtherexplain the various components and usage of D－HyperCase[5]. The top left panel shows the objects which are defined. A more detailed explanation of each of the individual types is provided in the overview figure(not shown here). The top right panel gives a pictorial representation of a prototypical decision graph. Again further explanation is available on demand. The bottom left panel shows the set of tools which are available. The D－HyperCase tools are those provided by the basic D－HyperCase machine. The User tools are those tools which are provided on top of D－HyperCase. The bottom right panel illustrates the HyperLink connection between a decision node in the decision graph and its associated description and the set of source code views affected by this decision. Selecting a decision node in this graph will display a menu which includes the names of all HyperLinks. This allows access tothe description or source code associated with this decision. A tutorial associated with this panel will lead the user through the use of the HyperLink facility in forming the closure of a decision.

---

[5] In fact all the figures in this section are of type document figure and are used to explain D－HyperCase. That explains why the HyperLink "Overview" appears on these figures.

Figure 5 show the layout of the screen during a typical D－HyperCase session. The screen consists of several windows. The large underlying window contains the decision graph (which can be brought to the foreground on demand). The large window on the right is the editor (modified emacs) window for all objects which belong to the superclass of text objects. These two windows are always open. In addition, the user may open several other windows. Figure 5 shows several read only windows for displaying various information and agraphical editor window to be used for creating document figures. Each of these windows contain a menu for accessing both tool specific and D－HyperCase operations.

Surrounding the decision graph window is a background which can be selected to access the D－HyperCase background menu. This menu allows direct access to the hyperlink facilities, such as linking to another object or tracing back to objects previously seen. Using the background menu, the user can access the document base even when using tools which do not have a direct interface to the AHM.

5 Discussion

Although it will still be several months before the first builds of D－HyperCase are finished to the point where we can begin to gather experimental data on the effectiveness of the decision－based approach to software development, there are serveral important observations which can be made. First this approach is not a panacea for all the ills plaguing the software engineering community. While we claim the decision structure can be used to provide multiple "natural" views ofthe software system, the definition of the decision graph is neither easy nor obvious at first try. But no one said that the generation of complex systems would be easy [3,6]. It is not easy to uncover the true dependencies. There is a temptation to represent instead the temporal ordering of decisions instead of the necessary dependencies. Since the decision graph is a directed acyclic graph ordered by the relationship justifies/is－justified－by, it should be possible to trace from the source code back to the set of requirements which justify it or from a requirement to the set of source code views that it justifies. The transitivity of this relationship is what helps us to form the closure of a decision. Anything in the transitive closure of the graph which is not in the closure of the task is noise and canonly serve to confuse the user.

A second difficulty we have experienced in using this approach is that it often easier to identify the solution than the problem it solves. This difficulty is not particular to the decision based approach. It takes great skill to uncover the real problem when given a list of wants and desires. End users find it hard to articulate their real needs but can often recognize when a proposed solution meets or doesn't meet those needs. For example, in the design of D－HyperCase, the need for read－only displays was introduced. Upon examining this need further, it was discovered that we were solving the perceived problem that displaying multiple emacs editor windo wswould be too expensive and that a simpler and less memory intensive display program could be used instead. One of the difficulties in introducing a separate tool for displaying text ob-

jects is that if we used an existing UNIX tool (like "more" or "less") the user interface would be different from that for the modified emacs. The choice of a read-only display tool was only one of several alternatives but until the real problem was identified, no consideration of the possible alternates was made. It is interesting to note with respect to this example, that a second decision justifying read-only displays can be made based on the argument that by default all objects should be read-only. In order to modify an object, it should be explicitly checked out of the version control library.

The above remarks illustrate one of the advantages of this approach for initial development. By requiring explicit recording of everydecision, the alternates considered and its justification, and by requiring the explicit dependency path from every decision to the initial set of requierments, visibility of the design process isincreased. This visibility helps in identifying generalizations of the proposed design. If you wish to solve problem A, first try solving amore general problem B such that the original problem A is just a particular solution to problem B. When we first started on D-HyperCase, we had only two tools, emacs and SGKBS and no separate HyperLink machine. We first tried to build hyperlinks between emacs and SGKBS before we realized that we could solve the more general problem, resulting in the HyperLink Machine, and use it to solve our particular problem. Once the more general problem is solved, extending D-HyperCase to other tools is much simpler. Aiso, by requiring anexplicit path back to the requirements, unneeded "bells" and "whistles" are identified. Additionally, missing requirements may be uncovered.

We have not found the design process to be top down and we suspect that most design is not. Top down is a good method for presenting a final design decomposition, but is not an accurate reflection of the process of design[6] (Much like the distinction between the activity of doing a mathematical proof from its presentation). During the creative ferment of design, people work at many levels, exploring issues in depth until a solution path becomes clear (this avoids too much backtracking). Many decisions are made, many problems are identified, alternatives are explored and rejected and justifications are sometimes hastily made at many levels of the design somewhat concureently. We have found that the decision based approach helps to organize and structure this process. As problems are identified, alternative rejected and decisions made, they are recording without worrying about tying everything together in the decision graph. Later on after the major insights have been made, the variouspieces are tied together using the principles of decision baseddesign (e.g. only include necessary dependencies on a dep endency path).

We have found recording alternatives to be valuable both todocument dead ends and to provide starting points for later improved designs. This later is especially useful if one is using anincremental build philosophy.

6 Conclusions and Future Work

Although we are gaining much valuable experience in decision based software development in the

design and implementation of D－HyperCase, we are still undoubtably on the learning curve. we have found that documenting the decision structure provides good discipline. Thus, this approach may be less of a burden and more of an aid for initial design than originally thought. The ability of the decision based approach to incorporate management decision and to relate them to the technical decisions and to work across the lift cycleattest to the generality ofthe approach.

Several of the design choices made for D－HyperCase with respect tothe underlying Abstract HyperLink Machine (based on typed objects andlinks) and user/project extensibility (including the binding of commercially available tools)needs further investigation but may lead to new approaches to building software development environments. Although we now feel that a decision－based can help in the initial design, it effectiveness during software maintenance is untested. The major objective still to be met is the evaluation of a decision based methodology in software maintenance. We are planning controlle dexperiments using the D－HyperCase prototype as it evolves.

(All figures are omitted.)

References

[1] Robert Akscyn, Donald McCracken, and Alise Yoder. Kms: adistributed hypermedia system for managing knowledge in organizations. CACM, 31(7): 820－835,July 1988.

[2] James Bigelow. Hypertext and case. IEEE *Software*, 23－27,March 1988. wild tektronix hypertext Abstract Machine HAM neptune dynamic design.

[3] Fred Brooks. No silver bullet: essence and accidents of software engineering *Computer*, 20. 10－20, April 1987 wild ifip incremental development object oriented design.

[4] Ned Chapin. Software maintenance life cycle. *Proceedings of theSoftware Maintenance Conference*, 6－13, October 1988.

[5] Jeff Conklin. A survey of hypertext. *MCC Technical Report*, STP－356－86(Rev. 2), Dec. 1987. wild r880803a.

[6] David Lorge Parnas. Software aspects of strategic defense systems. *American Scientist*, 73: 432－440,Sept. －Oct 1985.

[7] S. N. T. Shen, L. Liu, and J. Hsu. A hypergraphics tool formultidisciplinary applications. *Proc. of the Int. Symp. Expert Systems Theory and Their Applications*, Dec. 1988.

[8] Eliot Soloway, Jeannine Pinto, and Stan Letovsky. Designingdocumentation to compensate for delocalized plans *CACM*, 31(11): 1259－1267,Nov. 1988.

[9] Richard Stallman. Gnu emacs manual. October, 1986.

[10] Chris Wild and Kurt Maly. Final report contractnasl－17993－56. March 1988.

[11] Christian Wild and Kurt Maly. Towards a software maintenancesupport environment. *Proceedings of the Software Maintenance Conference*, 80－85,october 1988.