PROGRAM DOCUMENTATION WITH ADVANCED DATA MANAGEMENT SYSTEMS

Dr. Wayne B. Swift
Computer Sciences Corp.

The problems of program documentation posed by modern data management systems (DMS) are becoming increasingly important as the use of such systems becomes more prevalent. There seem to be two types of program documentation: the kind that should be done and the kind that usually is.

This first kind of documentation occurs when programmers begin a development by preparing requirements documentation as the very first step. They first develop the top level specifications that describe what the program will do. Once it is agreed that this is, in fact, what the program should do, the next level of system design, requirements analysis followed by general design, begins. After these steps have been approved, the design is broken into smaller and smaller pieces. The process of breaking down the design into small pieces is analogous to the engineering practice of detailed design. The process continues until the pieces are small enough for someone to prepare. A document that sets forth the segmented design is thus automatically created before any code is actually written.

What usually happens is that a programmer decides what a program is supposed to do, writes the program, checks it, and satisfies himself that the program does what is required. The documentation that results from this approach usually consists of only scattered notes plus a few general reports that do not fill in all the gaps in the development. At this point, it is usually quite difficult to improve the documentation because the programmer is probably heavily engaged in a new project or has left to join another organization.

The computer business has liked to think that this kind of documentation has fallen into comparative disuse recently. It is still, however, far from dead and probably the most prevalent kind, though everyone in the computer business reports that plans are under way to change over to good practices very soon. Manual documentation is generally so bad that even very poor automatic documentation is often better, which underscores the need to press for almost any kind of automatic supplement to or substitute for manual documentation. If any help at all can be given to the automation of requirements documentation, it may also serve to encourage better documentation practices.

The automated documentation aids that are generally available today, however, are primarily useful at or near the coding level. Automatic flowcharting and the like are, of course, quite useful in their own way. Full use of all such aids should be pushed, but this will not help much in settling requirements and assisting in the development of direct documentation practices of the type that should be followed. This is true even in the traditional

areas of program development where a programmer works on a single job at a time, at least from a logical point of view. This symposium will deal, to a great extent, with available tools in circumstances that include a computer with a normal operating system and the usual associated software. This paper, however, will consider those cases in which the programmer is using a computer along with an advanced DMS.

This situation presents two types of severe special documentation problems. One has to do with the high-level description of the approach that the programmer takes in processing data with the aid of the DMS. The other problem is attempting to figure out what the program has actually been doing. During debugging, many tools are required; this also affects the general documentation problem. Tools that adequately reveal what has been done are, in general, not very widely available with the present generation of DMS's.

There is, perhaps, a third type of problem related to both of these: Given a large and generalized DMS, how does one prove conclusively that this system either does or does not do exactly what it is supposed to do. Certainly, the answer to this problem involves documentation.

A DMS is used to produce a wide separation between procedures, on one hand, and data, on the other. This separation itself creates a considerable number of documentation problems that are not found in normal computer processing. In proposing DMS-oriented, automatic documentation tools to solve some of these problems, five specific problem areas for which some better kind of tool ought to be found come to mind.

The first of these areas is data description. A DMS facilitates data description by separating the problem. Data description lends itself to generalized documentation much better than does process description. It seems quite possible that a generalized data description language may be developed that can itself provide adequate documentation of data without any automated documentation at all beyond the *use* of this language itself. In spite of the fact that processing languages (for characterizing things that are done to data) seem to be evolving toward a babel of different languages tailored for different purposes, there is real promise for the development of a single generalized data description. This will be useful at even the highest level of requirements specification.

The second problem area is the process description language at high macrolevel, really the level of general design or perhaps even requirements specification. Such languages are likely to be intended for application areas only. At this highest level of characterizing what a system does, tools of representation can be automated either by the development of synoptic representations of the logic flow or by the development of languages that are more readily understandable than are typical programming languages. The potential for this is great, but its realization is much further away than is the development of data description languages.

The other three problem areas to be discussed all fall into the general area of execution-time documentation, as opposed to the overall descriptive documentation. Increased use of automatic documentation tools represents practically the only hope of discovering what the program is actually doing. Several things must be discussed to provide an understanding of the connections between overall system description and the actual happenings in the computer at execution time. These connections, or transformations that are made between the

overall program and what is actually being done by the computer, are becoming increasingly complex. There seems to be no reason to expect that they will not continue to become more complex, perhaps even at an accelerated rate. The general software and hardware construction of systems suggests that the use of large machines is going to continue to grow. Their economies of scale are already sufficiently impressive that larger numbers of larger machines will probably continue to be developed for a considerable period of time, in spite of the strong rise in minicomputer use in the last few years.

It seems fair to predict that a program as massive and complex as the average DMS is likely to gravitate toward the larger and more complex computers. This would be true even if the economies of scale of large computers were the only factors that affected this evolution. In fact, there is another factor that tends to encourage even more strongly the use of very large and complex computers. The DMS deals with situations in which a large and complicated collection of facts is created. Whenever a large and complex collection of facts is created, it tends to draw attention from many users in many different places. Therefore, there is a demand to make that collection of facts available to a large number of people either by permitting many users to deal more or less simultaneously with the same collection of facts or replicating the collection.

The factors needed to judge the economy scale of the situation must not be limited to usage alone, i.e., a large number of users simultaneously on one machine versus one user at a time on his own machine, but with many machines active at a time. Another factor is the common data base and the problem of keeping that data base current at many different locations. The problem of updating multiple copies is so severe that it alone will dictate a single-system choice in many cases. The multiple-user situation poses many special problems in the description of whether the right logical things are being done by the DMS. To phrase the problem in its simplest terms: Is the DMS functioning? The three following areas of documentation are those for which the prospects of the creation of automatic execution-time documentation that would be useful in determining how well the DMS is functioning are particularly promising.

One is the fixing of the binding point, the point in processing when the indexes of a body of data in the system are actually connected to that data. When DMS performs a search, it usually does so by following a series of steps. First, search criteria are formulated in some fashion. Then, a process occurs whereby some type of preliminary search is made. The term "preliminary search" denotes an activity that stops short of actually examining the data elements themselves and checking whether they are hits or misses on the particular search involved. This is usually done by using the indexes already in the system, as described in the following example.

An activity occurs that attempts to narrow down the search so that a considerable portion of the total file is somehow excluded from consideration. Thus, when an examination is finally made of the particular elements of data that eventually emerge from the search, the indexes, perhaps on several levels, that point toward the body of data in the system are manipulated continually. Tests are made to find out which data elements may be valid and may satisfy the conditions of the search. Often, the system is designed to postpone until a later time the actual retrieval of the surviving elements of data for the final tests of whether each satisfies the criteria of the search. Some systems delay these final tests until the items

are recovered from storage; in other instances, the issue is settled completely by examination of the index. Whether this can be done depends, of course, on how the indexes are generated. Whatever is done, the programmer, at debugging time, needs to be able to know the binding point.

From a total processing point of view, it is generally advantageous to delay the binding point as long as possible because to find an item often involves several levels of index lookup before a particular physical location on a disk is found and read. This tends to be a relatively time-consuming operation. Moreover, the item itself is generally much bulkier than its corresponding index entries in a well-conceived system. Therefore, a procedure with a delayed binding point tends to reduce input/output (I/O) load on the computer. In any case, it will have an important effect on what the machine is doing and therefore, will be found in working memory whenever a programmer requests a dump. He must understand this to interpret the dump, to see whether his search is proceeding in the specified way. In other words, he needs to see whether the DMS is functioning correctly. This is one area in which some kind of automatic documentation is needed for some indication of how binding points are being established and when the binding points occur in DMS processing.

The fourth area has to do with things that are concealed from the programmer while he is writing the program. The programmer need not worry about these points as long as the system is functioning properly. Many systems try to optimize processes at execution time. Whatever optimization the system applies to a specified search or to any other process that is specified by the programmer, it will affect the dumps that the programmer sees in case of system malfunction. Optimization results from an effort to make things convenient for the computer at execution time, but it causes the computer to execute steps that are somewhat different from the ones that were actually stated by the programmer. This process of optimization is admirable as long as everything in the optimization logic has been checked out and is valid, but when one is trying to certify that what the program is doing agrees with what it is supposed to do, sometimes difficulties result.

Typically, a DMS causes processes to be data directed. That is, drastically different things happen when different values are actually manipulated by the system at execution time. Systems generally are put into use before every conceivable combination of paths programmed into the system has been exercised and tested. Therefore, it is important for the user to have some way of discovering what the optimizers actually did to his original code. He must know this before he can honestly and sensibly judge whether an apparent malfunction is his mistake or a system error that needs to be corrected. Documentation suitable for this needs to be improved in DMS's.

The final problem area to be discussed is the need for system-supported traces that are reflective of the steps that are followed. These are needed both from the system point of view, so that, for example, the man who is working for the system operator and trying to understand what the computer is doing can proceed in a step-by-step fashion, and from the user point of view, so that the activities relevant to a given user's program can be isolated for review even though his work may be interspersed with that of many other users. All these things need to be documented for the user in a form that is sufficiently detailed to be useful to him in understanding what the system did with his program and, at the same time,

general enough that he is not buried by a mass of paper. One of the things that makes this a muddy area is that, clearly, the internal design of the DMS affects the values of a trace. The system features may affect what is worth showing.

For example, one particular system being checked by Computer Sciences Corp. (CSC) has been built with a series of processing modules that are programmed in such a way that there are queues going into and out from each processing routine. All I/O activity is controlled by the overall supervisor, and each routine is self-contained. In a case of this sort, a trace that simply indicates the order in which transactions traverse these individual modules of procedure, all of which have to run all the way to conclusion because of the rules governing the system, can be readily automated, but no more is really needed to permit a user to understand the validity or the lack of validity of what the computer did to his program. It also will facilitate his comparison of what the system actually did with what he programmed. This will greatly assist him in deciding whether he made a mistake or there is something basically wrong with the way the system is performing.

In summary, the modern form of DMS, with the discipline that it enforces by separating the data and the process, often creates a situation in which the process is specified more in a manner that is similar to that employed with decision tables. That is, the details of the steps are sometimes of interest to the programmer, but many times they are not. In cases for which the steps are not of interest to the programmer, it is probably safe to say that, from program to execution, present-day systems provide very little aid in the discovery of the connection, or mapping, and the automatic documentation usually produced by such systems to tie together the things that the programmer does when writing his program and the things that the machine does when executing the program is inadequate to support a proper check of system operation. It seems that this is an area in which automatic documentation can probably make an important contribution, but so little has been done that immediate and important progress can be made whenever the profession mounts a serious effort.

DISCUSSION

MEMBER OF THE AUDIENCE: Would you say a little more about binding points?

SWIFT: Let me use an analogy. Consider the two ways in which algebraic equations are solved. With one approach, values are chosen and substituted into the equation at the start, and the equation is solved numerically. Therefore, the binding point, the point at which the connection is made between the definition of a variable and its value, occurs at the beginning of the solution procedure. If the other approach, the algebraic solution of the equation, is used instead, the binding point occurs at the end of the procedure.

Obviously, what a given system does in developing a binding point is important to our understanding of what ought to be in a given space and core at a given time. Therefore, in a certain sense, it is really a debugging tool.