N73-19221

# AUTOMATED ENGINEERING DESIGN (AED): AN APPROACH TO AUTOMATED DOCUMENTATION

Charles W. McClure
*SofTech*

The automated engineering design (AED) system is essentially a system of computer programs designed for use in building complex software systems, especially those requiring the development of new problem and user-oriented languages. The AED system itself may be considered to be composed of a high-level systems programming language, a series of modular, precoded subroutines, and a set of powerful software machine tools that effectively automate the production and design of new languages. The AED language itself is a modification of ALGOL 60 to make it suitable for use as a systems programming vehicle.

The AED system was developed by Douglas T. Ross and associates over a 10-year period while they formed the nucleus of the Computer Applications Group at the Massachusetts Institute of Technology's Electronic Systems Laboratory. Before their work on AED, this group had developed the automatically programmed tools system for numerical control of machine tools. So, it is natural that people with such extensive experience in the data processing field would be concerned with the documentation of their own activities and with providing facilities for the users of their work to effectively document their own new developments in the software field.

It is necessary to define what is meant by documentation before it is possible to discuss automated documentation. In a general sense, the term documentation refers to everything that can be used by a human being to help understand a computer program or system. This specifically includes flowcharts, listings, cross-references, and, of course, user manuals. This discussion will be limited to the more mechanical aspects of the total documentation spectrum. User manuals, which are more appropriately considered in terms of the psychology of human learning and communication, will be excluded from consideration.

In examining the process of software production, it may conveniently be divided into the design, programming, testing, and production phases. This is illustrated in figure 1. Current manual documentation methods ignore the process of documentation during many of these phases. For an automated documentation facility to work properly, data that are generated during all these phases must be captured and reasonably ordered. This is especially relevant in the programming and test phases, in which numerous individuals are involved and in which errors, oversights, and shortcomings in the original software design are discovered. In addition, the need for new and revised features unfolds as the understanding of the problem matures through usage. These new and potentially valuable insights are frequently lost during the current process, which tends to ignore documentation at these points in the process.

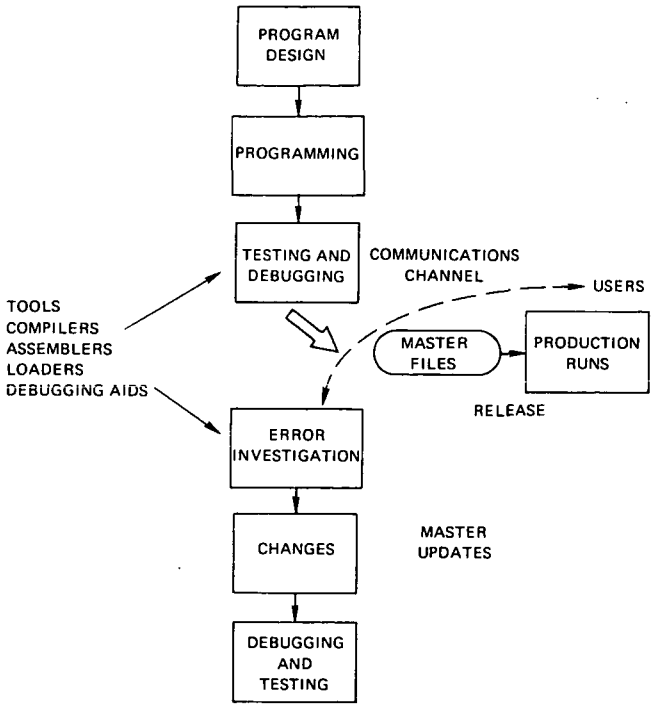


Figure 1.–Software production phases.

SofTech strongly believes that it is erroneous to view the problem of system documentation as an annoying, mundane, and unfortunate concomitant of the software development process. On the contrary, documentation is an important, challenging, and integral part of the total software production picture. Major system software efforts are among the most complex and difficult projects undertaken by man, especially when real-time physical phenomena as well as the vagaries of human behavior play an important part in the overall successful operation of the system. In software, the design-to-production cycle can never be considered completely finished.

Even if present efforts toward the development of highly reliable programming techniques are fully successful in the future, for major systems, the continuing evolution of new user experiences that must be reflected in modified system behavior still will cause the documentation function to be of primary and lasting importance. In SofTech's view, documentation is a natural part of the life cycle of a software system that comes to the fore not at a certain stage of maturity of that system but must be essentially functional as a useful problem-solving tool at each stage.

SofTech's many years of experience in major system development, maintenance, and distribution have given rise to a clear definition of the ultimate goal toward which we strive. This goal is the automated production of high-quality reliable and functional software in a "software factory" environment with the automated production of documentation as an integral process. It is not surprising that the production tools of the documenter and the application of those tools in an orderly technology are closely related to the similar functions required at the generative stages of software design and prototype construction. In

fact, it is the purposeful extraction of pertinent information from those production tools as an easily acquired byproduct of the production process that best provides an orderly approach to the solution of documentation problems. This is apparent because the documenters must mentally disassemble a working major software system into its component parts and examine the behavior of those parts in concert as well as separately. At the present time this analysis is supported only by an informal collection of listings, flow diagrams, and sundry debugging aids, most of which have been humanly generated in a sometimes well-directed, but all too frequently ineffective, effort to provide the needed information.

Quite a different state of affairs would prevail if the vast amounts of relevant information which appeared briefly and then evaporated in the process of initial construction and assembly had been saved as a part of the development process. For example, the documentation job would be greatly facilitated if the compiler used to compile a program left in a data bank not only the basic symbol table information but also the cross-reference information concerning external functions defined in or needed by that compilation, the linking loader had left behind its records of program module placement, and the system generation program had retained all relevant information including commentary by the programmer who made some last-minute adjustments, for example, to overlay linkage characteristics. Furthermore, if all of this information were ordered and collected in an automatically generated data bank as a byproduct of the software machinery, it would be essentially free of human error. This information could be combined with other features of the software production methodology, such as the requirement that appropriate commentary be incorporated in each module of a hierarchically assembled system. Before such an assembly operation would be considered complete and acceptable to the system, the body of information available to the system would attain some real substance and would provide a firm basis for a well-structured approach to the documentation task.

The value of this approach can be appreciated by noting that the construction of overlays (to permit a program to execute within a core memory constraint) has been a problem for a great many programmers using various computer systems. Yet an acceptable overlay structure can be produced quite mechanically if the program and data interrelations are clearly stated. This acceptable structure may be improved if frequency of usage information is available.

Programmers attempting to document their activities can benefit by having a high-level language uniquely tailored to order their thoughts and supply direction to the computer. The nouns of such a problem-oriented language are such things as program modules and system assemblies. The adjectives of the language are the various states of those modules, such as unedited source deck texts with commentary. The verbs correspond to the control of the various assembly operations cited above as well as the disassembly operations needed to unfold a structured program to gain access to internal points. With this accomplished, software probes or statistical measuring submodules nested within this language and interlocked with its features can be employed. Once a change has been made, the same set of tools controlled by the high-level language will record the updated information about what was changed, why, and how, as the single change must be reflected throughout the entire

system. This language could easily be used through a graphics-oriented terminal, thus providing the combined capabilities of high-level metalanguage with the most effective computer hardware man-machine communications capability available.

This description of the documentation function as an integral and natural part of the software factory of the future may seem utopian at first, but such a view does not do it justice. Instead, it provides the overall vision whereby order can be brought out of our present chaos. It provides a yardstick whereby current and proposed methods can be evaluated and also provides a concrete series of subgoals, many of which are immediately achievable as natural adjuncts to present documentation methods.

Another major aspect of the documentation retrieval problem that also merits particular attention is the language in which the system itself communicates its wealth of information back to the programmer. It is in this area that the various forms of computer graphics can play a particularly important role. The thought processes that accompany the detective work associated with debugging and design can be rapidly overloaded by unending detail. The information must be presented in a way that is most readily assimilated by the human mind. In debugging, even more so than in many other activities, it is very important that the programmer be able to maintain intellectual momentum in pursuit of an idea. This can be greatly aided by use of on-line graphic displays. Here again, the key to successful efforts must lie in the conscious treatment of the graphical displays as a proper language, not merely as a picture-making capability, coupled with appropriate information and data structuring for the data base.

It should thus be obvious that the data base must be content addressable, or be provided with "backward" threaded lists. Perhaps a simple example will demonstrate this need. Consider that the description of a tape file has been stored in the data base so that all programs that are to manipulate that tape obtain the details of its format from the data base. It is clear that reassembling the program will not only incorporate the latest changes to the program but also all modifications to the file description. But how are we to know which programs should be reassembled unless the data base retains the information about which programs do in fact rely on a generic file description?

## THE IDEAL ENVIRONMENT

The ideal environment and long-term future goal is a comprehensive, totally computer-automated system that provides a rich, system-independent metalanguage for assembling software systems, tearing them apart, examining them under specific test conditions, modifying master source program files, and other tasks. The nouns in this control metalanguage are master file names, loading and overlay patterns, and test procedures, while the verbs include compiling, loading, updating, debugging, and data base query actions.

The system design provides that any changes made to a master file will be rejected by the system unless all required updates are completed, including documentation updates and updates to related programs affected by the original change. Management control reports are also automatically provided, such as reports of schedule slippage and accuracy of maintenance estimates versus actual performance.

The primary user control device for this system is a graphics terminal, which is particularly adapted to rapid rifling through source files, program editing, display of program flow, and documentation updating. The research in machine-aided cognition by Dr. Engelbart at Stanford University has demonstrated the power and utility of such devices.

The behavioral scientist will play a central role in the development of the language by providing the needed human engineering. He will insure that the language is properly matched to the skill level, human limitations, and working conditions of the maintenance programmers. He will also criticize the management control reports and the user/maintenance project communication facilities to tune these important interfaces for maximum compatibility.

The data base required for effective operation of the maintenance facility is initially constructed by the computer by stripping off program information at each stage of software production, somewhat along the lines of the COMPOOL facility of JOVIAL and the AED insert file control mechanism. Compilers, assemblers, loaders, and other software production tools already produce listings, load maps, subroutine cross-reference printouts, and other aids as natural byproducts of their function. These valuable aids form a major portion of the needed data base, although at present these data are not retained and cross-indexed automatically by the computer for later use. Original program documentation and facts discovered at each stage of the maturing of the software form another important component of the data base.

## FACTORS WHICH INHIBIT EFFECTIVE DOCUMENTATION

The engineered documentation environment described above is at present not a reality. Without engineering tools and disciplines, documentation efforts suffer from several inhibiting factors including–

(1) Lack of program understanding by programmers, compounded by poorly engineered and programmed software
(2) Lack of automated software tools for maintaining, updating, and distributing documentation of systems to users
(3) Communications gap among the programmers about status of known errors and correction schedule
(4) Lack of automated testing techniques to insure that modifications do not in themselves create additional errors
(5) Interference and cross-coupling between several programmers simultaneously changing and correcting related programs
(6) Inability of programmers to reproduce error conditions, at least in a sufficiently simple environment to isolate a single problem effectively

## THE PATHWAY TO THE GOAL

Given the gap between the ideal environment and the present-day facilities, an evolutionary path to the goal is needed. A graceful degradation of the ideal to fit the present-day hardware and operating system limitations provides a usable facility whose

expansion and elaboration insure the needed incremental progress toward the ideal. Many of the programming aids needed to carry out the actions of the metalanguage already exist in present computer software libraries:

(1) Automated flowchart aids such as AUTOFLOW to produce quick, accurate flow diagrams directly from source program statements
(2) Machine-language listings showing the primitive steps compiled to carry out statements in a high-level language
(3) Symbol prints to show memory locations and other values associated with user variables
(4) Load maps showing program layout when executing the program
(5) Cross-reference maps showing interactions among program modules
(6) Library statistics showing the names and sizes of programs called in from software libraries
(7) Traces of program flow showing argument values and machine conditions at selected points during program execution
(8) Linkage tables showing names, locations, and lengths of selected programs
(9) Timers and program counters which illustrate dynamic characteristics of program operation

It is clear that many of the needed software tools are now available. What is missing is the overall system which will permit each of the components of the system to retrieve the data that it needs to perform its function without asking for additional information from the programmers. It is clear that if processors are going to obtain information from the data base, then it is desirable that each processor store information in the data base for the use of other processors. This is the essence of the software factory of the coming decade.

The software factory provides the framework within which the programmer functions and, in particular, should provide both the present processors and the data base discussed above.

SofTech feels that the AED approach permits us to work toward this final goal in the most direct manner now possible.