A FUNCTIONAL APPROACH TO STRUCTURED PROGRAMMING

DR. M. LEONARD BIRNS
Senior Computer Scientist, Computer Sciences Corporation

1. INTRODUCTION

As the functional requirements of training devices have become more complex, as digital techniques have become more sophisticated, and as the hardware required to implement these techniques has become more available, the digital computer has become increasingly essential in the field of training simulators. Simultaneously, because of the increasing amount of software required, as well as the changing ratio of hardware to software costs, it has become necessary to emphasize software development and software reliability.[1] The growing concern for software reliability has led to a proliferation of techniques for software implementation. Of these, structured programming is presently most popular.

Structured programming has attained such a level of importance as a technique that it has become something of a "buzz word" in the industry, being used to describe many techniques which imply some predetermined structure, such as modularity, even if they do not necessarily adhere to structured programming principles. In addition, certain developmental techniques, such as top-down development, have become associated with structured programs – although there is no necessary correspondence between the structure of the program and the techniques by which it was developed.

The objective of this paper is to present a definition for structured programming (in no way considered to be original), and a description of some structured programming techniques. This description will lead to a discussion of a limitation of the totally top-down, structured approach to real-time software development, this limitation being the frequent lack of a functional orientation. A technique will be described which allows this limitation to be overcome by superimposing a functional critique on the control structure provided by structured programming techniques.

2. STRUCTURED PROGRAMMING

Structured programming is the implementation of proper programs. A proper program is defined as a program which contains only one entrance and one exit. Structured programming maintains that the large programming systems in which we are interested can be broken into subsegments, each of which is itself a proper program and is comprised only of proper programs.

The concepts of structured programming are not new. (E. W. Dijkstra, the "father of structured programming," admits that he has primarily made explicit what competent programmers have already done for years.)[2] Nevertheless, recently, as structured programming techniques have become more refined, as appropriate tools have become more available and the benefits of the techniques have become more visible, great emphasis has been placed on these techniques. It is important, however, to recognize the difference between the techniques by which structured programs can be developed and the concept of the structured program itself. Also, we should recognize that structured programming of itself is not the final panacea for all program development problems, but only an additional tool which can be used to solve some of these problems.

The primary technique used today to ensure structured programming is "GOTO-less" programming. This concept developed after it was proven that any proper program could be written using only three forms of logic structure:[3]

- Sequences of two or more operations.
- Conditional performance of one of two operations (IF a THEN b ELSE c)
- Repetition of an operation while a condition exists.

Constraining programmers to the use of these three constructs forces programs into a structured mold by eliminating branches. It should be noted, however, that structured (proper) programs can be written without eliminating GOTO from the programmer's vocabulary. As an example, consider the following structured code, written for the AN/UYK-7 computer using the CMS-2Y compiler:

```
IF ENGINE (X) EQUAL 'HOT' THEN
    BEGIN $
        ENGHOT INPUT X "SET ENGINE
        PARAMS FOR HOT" $
        MALDIS          "SET UP MAL-
        FUNCTION DISPLAY" $
    END $
ELSE
    BEGIN $
        ENGNRM INPUT X "SET ENGINE
        PARAMS FOR NORMAL" $
        NRMDIS          "SET UP
        NORMAL DISPLAY" $
    END $
```

The brackets describe the control structure of this program, one major block and two nested blocks, each having only one entry and one exit, each procedure call being considered as a single entity.

The control code generated by the CMS-2Y compiler for this sequence is:

```
            LA      A1, ENGINE, K3, B7, S0
            C       A1, +1, K0, B0, S0
            JNE     GL001132, B0, S0
            SB      B7, ENGCON, K4, B0, S0
            LBJ     B6, ENGHOT, B0, S0
            LBJ     B6, MALDIS, B0, S0
            J       A0, GL001133, B0, S0
GL001132    SB      B7, ENGCON, K4, B0, S0
            LBJ     B6, ENGNRM, B0, S0
            LBJ     NRMDIS, S0, S0
GL001133    —
            —
            —
```

The effect of the generated code is to maintain the three blocks, but to implement the control mechanisms using jumps. Obviously, the same effect could be provided with GOTO statements if the IF-THEN-ELSE structure were not available; i.e.,

```
            IF ENGINE (X) NOT 'HOT' GOTO
            NORM $
            ENGHOT INPUT X $
            MALDIS $
            GOTO NEXT $
NORM        ENGNRM INPUT (X) $
            NRMDIS $
NEXT        —
            —
            —
```

The structure of the program has been maintained, although some of the readability has been lost because of the use of GOTOs and the lack of indentation. The increased readability is an advantage of GOTO-less programming, in addition to the forced structure. In passing ing, it should be noted that while there is widespread acceptance of GOTO-less programming techniques, this acceptance is not universal.[4]

A second development technique frequently associated with structured programming is top-down programming. This idea develops from the concept of the proper subsegments which compose the structured program. Because of the relationship among these subsegments, the proper structure normally portrayed as shown in Figure 1 can also be portrayed as shown in Figure 2,

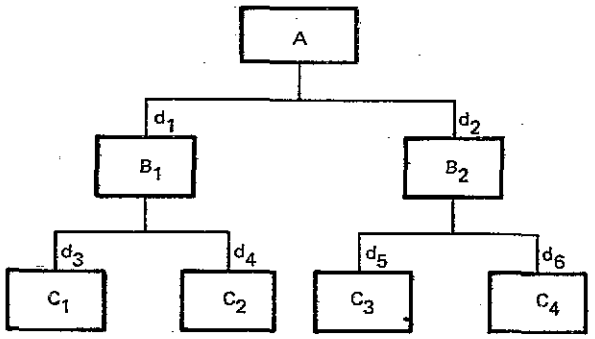


Figure 1. Standard Structural Portrayal

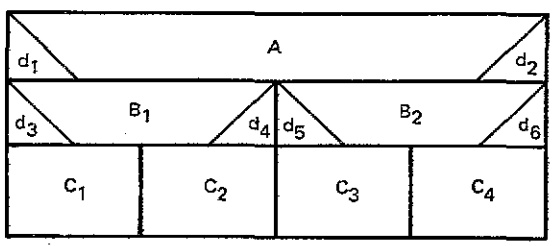


Figure 2. Chapin Chart Portrayal

called a Chapin chart. This chart shows the top-down structure or inclusiveness of the program and the decisions (represented by the letter "d") which control the various paths.

It has been advanced that the appropriate way to approach the development of this structure is to begin with the highest level segments and continue to develop the subsegments down to the lowest level, always completing an entire hierarchical level before proceeding to the next.[5] This progression is advocated during design and also during implementation. However, it is possible to develop a structured program without applying top-down techniques, and top-down techniques applied during design need not necessarily also be applied during implementation.

Investigation of the development of structured programs using the top-down approach reveals that, although advantages are derived from these techniques, certain attendant disadvantages exist.

The first disadvantage relates to the idea of a strict top-down program development. Although design can proceed from a top-down direction, implementation from the point of view of schedule and cost is frequently more efficient when lower level routines are implemented first. A simple example of this involves the many library routines from which a large system is built. The design of the overall system may possibly require a

considerable amount of time; but it is frequently known quite early which library routines will be required. In addition, the design and implementation of mathematical routines such as sine, cosine, or randomization routines may be sufficiently well known to allow beginning their implementation and test even before the total system design is finished, thus permitting a paralleling of efforts and a reduced schedule.

An additional argument regarding top-down implementation is that, although the system is theoretically integrated at each step of the development process, in practice it turns out that actual functional operation is not verified until the lower level routines are implemented. Thus, it is possible to find oneself toward the end of a large project with the surprise that the theoretically integrated system has fallen apart functionally.

3. THE USE OF THREADS

Obviously, software which is to be produced for the tight schedule, fixed-price environment typical of trainer procurements cannot allow for last-minute surprises such as the one described in the previous section. It has been found that these surprises are principally based on two factors, both functionally oriented.

The first factor is that one cannot truly test a functional capability until all contributions to the functional capability have been implemented. Frequently, the components of a function which provide details of great concern to a user are implemented on a low level. As a result, these details are not verified until very late in the implementation process, if the implementation is performed totally top-down.

The second factor is a lack of correspondence between the top-down control flow (which is an advantage in the program design) and the overall data flow, caused by program operation. This data flow does not necessarily correspond to the flow of control. The program design is specified in terms of top-down structured flowcharts or pseudo-code which describes the control relationship among the software subsegments. However, these charts do not describe the input/output transfer function which the software is actually to provide. This problem is depicted in Figure 3, which superimposes the flow of data corresponding to an input/output requirement on the control structure previously shown. The input stimulus is fielded by an executive level routine and uses a top-down path to the data base. Sometime later, the executive activates a chain of routines which channel data back to an executive level I/O routine that provides the actual response. This data path does not correspond to the control path.

The discrepancies between the portrayal of control and data flow can be resolved by associating with the structured approach an additional tool called Threads.© The Threads technique provides a design and implementation tool, as well as a management tool for controlling the overall program development by automating the reporting processes required.

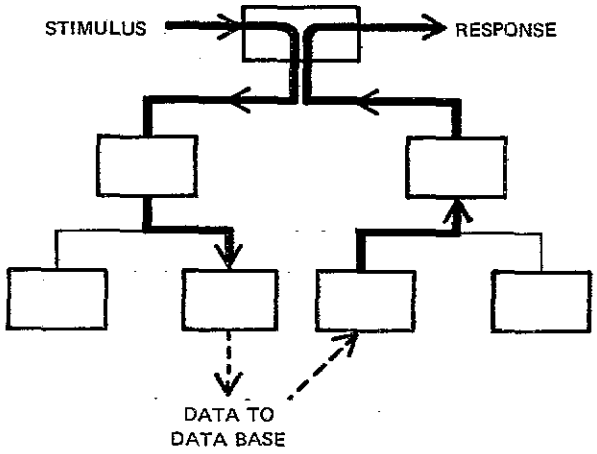


Figure 3. Relationship of Control Structure and Data Flow

A thread is the path (sequence of events) that a functional requirement traces through the components of a system. (In fact, the heavy line portion of Figure 3 represents a single thread.) Consequently, a thread comprises all the specific tasks necessary to fulfill the functional requirements. These tasks can be logically categorized into several levels, as shown in Figure 4. Figure 5 shows the paths of threads through a system and the graphical portrayal of the various thread levels.

At the top level, threads relate to system elements and functions; at the intermediate level, they reflect subsystem elements and subfunctions. Subsystem elements are the major components of each system element. In the example presented in Figure 5(c), the system element entitled "Command and Control" is divided into subsystem elements. Each of the remaining system elements has its respective subsystem elements.

Subsystem functions at the intermediate level correspond to functions at the top level. Subsystem functions consist of the detailed requirements for each function. In the example, the command and control requirement might be to "notify the command and control officer." The corresponding subsystem function might be to "receive the message from the tracking system element (Tracking Interface program), perform tracking processing (Tracking program), execute alert procedures (Alert Maintenance program), transfer and display messages (Data Transfer program), and transmit data to weapons control system elements (Weapons Control Interface program)." When these subfunctions are traced through the subsystem elements, subsystem level threads are

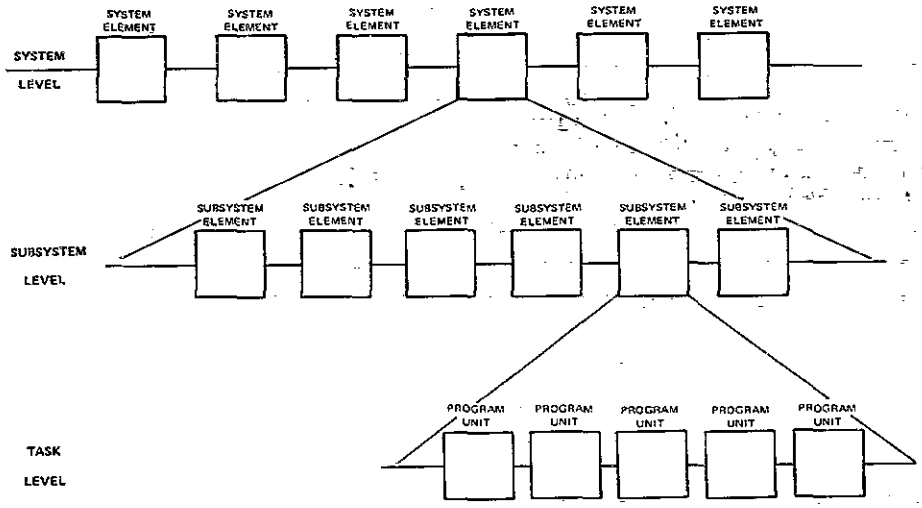


Figure 4. Relationship of Thread Levels

produced. As in the case of a system level thread, a subsystem level thread may be depicted in a series of graphics, with each unit representing a subsystem element. Similarly, subsystem level threads can be drawn for each of the other system elements involved in the system level thread.

At the lowest level of threads, program units and tasks are related to the subsystem elements and subfunctions of the intermediate level. Program units are the major divisions of a subsystem element. In computer programs they are software subroutines; in manual operations, they are assignable work units. The Data Display program subsystem elements were selected as the subfunctions for the example given in Figure 5(e).

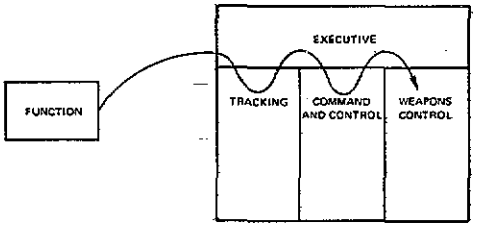
Each of the other subsystem elements would have its own set of program units. Tasks at the lower level correspond to the subfunctions at the intermediate level; they are the detailed requirements for each subfunction. For example, "data display" would be a task. When these tasks are traced through the program units, task level threads are produced.

The thread methodology represents hierarchical structure in a manner enabling the efficient control of program developments and implementation. Furthermore, the threads provide a functional review which must be superimposed upon the software design in order to ensure that all system requirements have been matched. The superposition of the thread on the top-down software design ensures that the functional requirements have been met by associating each task in the functional requirement with a software block in the design structure.

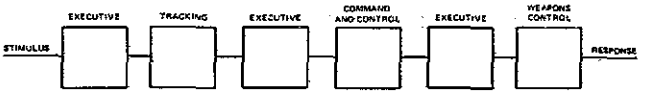
4. IMPLEMENTATION

Having completed the design, with control specified by the top-down software structure and with functional requirement verified via the specification of functional threads, the next question to be dealt with is the technique for implementation. In order to ensure software reliability, the implementation technique must be based on enhancing the capability for software verification. Such verification should be performed on a functional level. Although it is important to ensure that a software system as developed provides the structure and control provided by the structured design, the user (and therefore, the programmer) must be primarily interested in whether the system performs the functions required of it (i.e., is the input/output transfer function specified by the totality of threads satisfied by the implemented software). Since the goals are functional and the design has been verified functionally (using Threads) the implementation should proceed along the same functional lines. As the implementation proceeds, functions can be tested as small, manageable subsets of the overall requirement. This is the crux of the "build-a-little, test-a-little" implementation philosophy, which has proven successful on many system development projects.

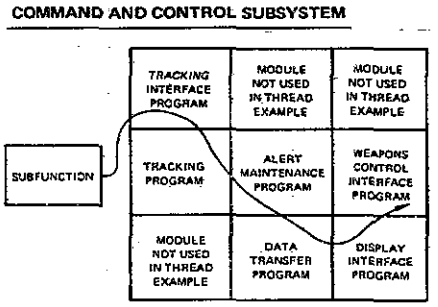
The entire development process can be described as follows: First, the overall design is developed in terms of a top-down structured software system, but one which functionally satisfies the requirements expressed by the set of functional threads. Implementation is then performed using the structured constructs if they are available in the language being used or using a structured discipline associated with the



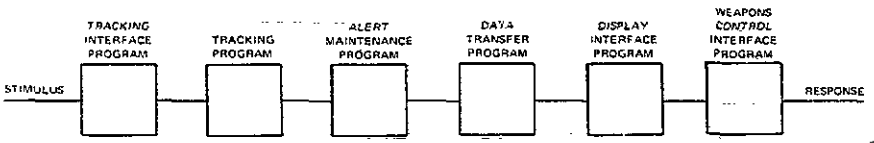
5(a). Paths of Functions Through System Elements



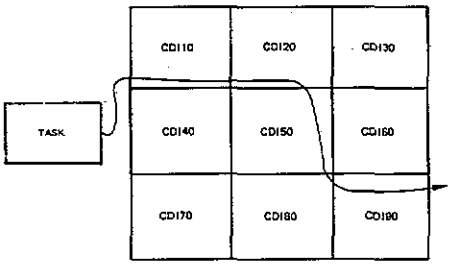
5(b). Graphic Representation of System Level Threads



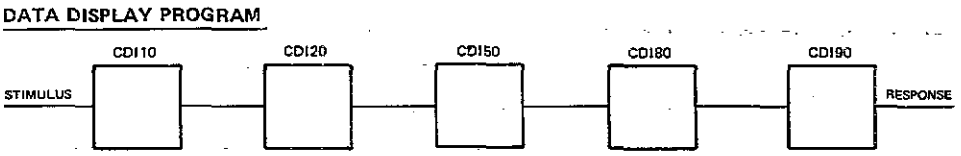
5(c). Paths of Subfunctions Through Subsystem Elements



5(d). Graphic Representation of Subsystem Level Threads



5(e). Paths of Tasks Through Program Units



5(f). Graphic Representation of Task Level Threads

Figure 5. Levels of Threads Analysis

language constructs available. The implementation is generally a combination of top-down and bottom-up development, in that the highest level of the system, the executive level, is developed as a communication and test tool, but the problem-oriented portions of the system are developed from the lower levels using the "builds" technique.

A build is a combination of various threads which together perform a useful, demonstrable system function. The development of a build is under the control of a build leader similar to the chief programmer concept which has been described in other literature.[6] The build leader is responsible for the coding and unit testing of the threads associated with his build, and for integration of these threads into the build. When the build itself has been functionally tested, it is then integrated with previously developed builds to provide a new software package which can perform the function specified by the summation of the build it includes. Thus, at each step of the way, as builds are added to develop the entire package, the program can be tested functionally on the basis of its input/output transform requirements.. This technique of development by builds is shown in Figure 6. Five builds, some developed in parallel, are depicted in a development process, with functional tests and demonstrations shown at each level. By demonstrating actual functional capability at each development phase, the integration surprises which sometimes accompany a total top-down development effort are eliminated, and a more easily tested product can be delivered.

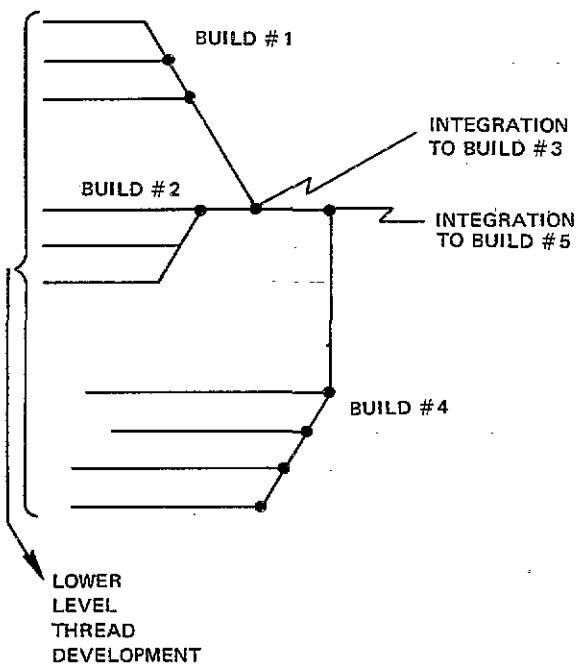


Figure 6. Build Development

REFERENCES

1. Nashman, A.E., Software Development Management - The Key To Quality Software Products, IEEE Electronic and Aerospace Systems Conference, 1974.

2. E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctives," BIT 8, 1968.

3. C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines and Machines and Languages with only Two Formation Rules," Comm. ACM 9 (1966).

4. W. A. Wulf, "Programming Without the GOTO," Information Processing 71, North-Holland Publishing Company, 1972.

5. H. Mills, "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, Ed. Randall Rustin, Prentiss-Hall, Englewood Cliffs, N.J., 1971.

6. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Vol. II, Number 1 (1972).

ABOUT THE AUTHOR

DR. M. LEONARD BIRNS is a Senior Computer Scientist at Computer Sciences Corporation (CSC), Defense Systems Center, Moorestown, New Jersey. Dr. Birns received a B.E.E. degree from the City College, New York; an M.S. degree in Physics, Magna Cum Laude, from Fairleigh Dickinson University and a Ph.D. in Operations Research from New York University. He joined CSC in 1973, specializing in advanced techniques in real-time programming. Previously, he was with RCA for 2 years where, as a principal member of the engineering staff, he participated in several projects, including those involving the design of real-time operating systems for radar control and the development of special purpose display software. At Decision Systems, Inc., where he was Deputy Director of Programming, he was responsible for mathematical modeling and systems analysis for the LEM simulator, and was responsible for the development of COMPOSE, a simulation display preparation language and control processor for a Coast Guard helicopter simulator.