LOGOS and the software engineer*

by C. W. ROSE

Case Western Reserve University
Cleveland, Ohio

INTRODUCTION

Most of us consider a well-engineered product to be one which is structurally sound; which communicates with its environment in a predictable, well-disciplined manner; which has been thoroughly tested; and which is reliable and easily maintained. In any engineering field, the structural philosophy, design disciplines, and checkout methods which yield such a product are called "good engineering practices." Software engineering is the application of good engineering practice to the design, implementation and final checkout of large programs. The result of effective software engineering should be:

(1) The production of a correct program (certifiable)
(2) The availability of means of efficiently determining the correctness of a program (certification)
(3) The ability to modify a program so that recertification is possible.[1]

The goal is to organize complexities, master multitude, and avoid its bastard chaos as effectively as possible.[2]

However, unlike many types of engineers, the software engineer has had few tools, either for implementation or analysis, with which to accomplish his task.

Many of the problems in operating systems which occurred during the mid-sixties can be traced to an inability to enforce the design disciplines indicated by good engineering practices, or to determine after the fact that they had been applied. In some cases the faults appeared years after the system was in the field. Higher level implementation languages for software remove many trivial coding errors and deal effectively with the problem of storage allocation; however, they do little in reducing the major problem of complexity—inter-module communication, software/hardware interface conflicts, and mishandling of real and apparent concurrency within the hardware/software system. This is more obvious when one remembers that most large design efforts are multiperson, and that software modules and hardware designed by many people must communicate properly at the many interfaces.

The hardware designer is somewhat better off since he can call upon switching algebras, flow table analysis[3] and register transfer languages[4,5] to aid him in the design. Unfortunately, these tools are not amenable to the design and analysis of very large systems, and the designer soon learns to modularize his system and to apply his techniques to several modules of manageable size. It is at the interfaces of these modules though, that problems equivalent to those in software arise.

The net result of this inability to systematically deal large scale complexities has been the late delivery of expensive and buggy computer systems. This is not to suggest that the several successful structural approaches to systematic operating system design[6,7] are insignificant, but rather that the difficulty of enforcing their requisite structural and communication disciplines becomes very great as the size of the target system increases.

Hardware engineers encountered this problem of complexity very early in terms of implementation, and responded by developing computer-aided design (CAD) systems for logic diagram production, package placement, wire routing and mask generation, and simulation and test generation.[8] Many other engineering disciplines have also turned to the computer to help deal with the complexities of large systems.[9] It is ironic, however, that the computer, which has great analytic capability, doesn't often forget details, and can enforce structural and communications disciplines by syntactical analysis, has not, to date, been applied to the conceptual and detailed design of computer systems.

* This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Department of the Army under Contract No. AMSEL-PP-Cn122(CAE).

Project LOGOS was begun in 1968 at Case Western Reserve University to exploit these capabilities by creating a computer-aided design environment for both the hardware and software of large-scale computer systems. An integrated hardware/software design system was chosen because mismatches at that particular interface in a computer system are the most costly and time-consuming to correct. The goals of LOGOS can be simply stated: the creation of a multi-designer environment in which computer system designers can define a system in which a high degree of parallelism or concurrency exists, verify its logical and functional consistency, evaluate its expected performance before implementation, and finally implement the hardware and generate the code for the software. Inherent in any CAD system is a philosophy of target systems structure and an associated representation system which both embodies that philosophy and has a well-defined syntax and semantics. It would be helpful here to briefly describe both to set the stage for a discussion of LOGOS' contributions to the software engineer.

A LAYERED VIEW OF SYSTEM STRUCTURE

From a user's viewpoint, a computer system presents an environment to each user which is characterized by a collection of service facilities.[10] Each facility may be activated and directed according to a well-defined communications discipline. Since users do not, in general, act in coordination, the system facilities must cope with multiple and asynchronous requests for services.

Response to a request activates the facility, an instance of which we shall call a task, and the method of handling multiple requests depends upon the nature of the facility. A single-user facility would queue all requests in excess of one, while a limited resource facility such as a magnetic tape controller with six tape drives would allow six concurrent activations before queueing requests. On the other hand, a fully reentrant software procedure would have no limit to its activations although exhaustion of some other resource such as core memory would impose a limit externally.

Users of facilities very often do not care about how a facility is implemented internally, but rather how it interacts with its environment. This concern with the input/output function of a facility is the "external" or "primitive" view. Conversely, a user and, in particular, a designer may need to know the details of implementation as well as the I/O function of a facility. This is the "internal" view.

A facilities approach to viewing systems immediately gives rise to a hierarchical structure. Many facilities in a system provide essentially identical services, or equivalently, have identical subtasks. These subtasks could be viewed as instances of activation of separate facilities shared among those requiring the particular services. The most primitive shared subtasks in a software system are the machine instructions. By the same token, however, the reading of a text file appears primitive to a compiler using the file system facility, although an internal view of the file system shows that the read file operation is quite complex and uses other shared facilities such as the disk channel.

It is natural, therefore, to structure a system as a partially ordered hierarchy of layers, the highest layer being the interface with the system users, and the lowest, the system primitives. A system primitive for a software system might be a machine language instruction or a library subroutine, while a hardware primitive might be a NAND gate or a four-bit MSI adder. A facility on a lower layer may be activated by a task of a facility existing on a higher layer. Its tasks may, in turn, activate facilities on still lower layers.

Between any two layers, there is an interface partitioning the system into facilities below the interface and users of those facilities above it. Users above present an environment of service requests and arrival rates, while facilities below present an environment of service available and service times.

The ability to "collapse" or look at a facility as a primitive suggests that consistency analysis of a facility could be done by exposing the internal structures, analyzing it, collapsing it, and then analyzing its interactions with its environment as a primitive. This is the only practical way of analyzing large systems, and the representation which accompanies this philosophy allows just that.

Implied in all of this is the existence of an interfacility communications discipline for both data and control. Several might be defined such as Dijkstra's P-V discipline[11] or Multics' mailbox scheme.[12] What is important is that whichever one is chosen, it must be enforced, or the layered model will break down, and the analytic capability afforded by this scheme will be lost.

A facility in general consists of four elements:

(i) Resources of one or more types which may be required by the facility subtasks.

(ii) An enclosing control which determines, based upon resource availability, if a subtask should be activated or if the request should be queued or dismissed.

(iii) A set of algorithms defining the subtasks. Algorithms are called activities; instances of their activation are called processes.

(iv) An interpreter which accepts user directives and determines appropriate action.

A facility need not have all of these elements. A wholly software facility would not have local resources, while a storage allocator would control a resource but have no set of algorithms to be selected by a user.

This philosophy of system structure can be applied to both hardware and software. It is consistent with the structural approach to proving program correctness[13,2,14,15] which is to force the structure of the program text (or representation) to correlate strongly with the structure of the actual computation, thus allowing analysis of the computation by analyzing the structure of the representation by stepwise decomposition.

THE LOGOS REPRESENTATION SYSTEM

The central part of any CAD system is its *representation system* which consists of the design data base in which the description of the target system is accrued, the external representation of this design information, and the translators between the external and internal form. The representation system must satisfy several global constraints.

First, the representation must be sufficiently general to describe all interesting and desirable objects in the set of design objects, while at the same time, it must be sufficiently specific to allow algorithmic consistency and performance analysis. Second, the internal representation must be decomposable into elements which may be implemented directly. Finally the designer must be comfortable with the external representation and the constraints it places upon his freedom of expression.

In the case of LOGOS, the target objects are facilities, which can be described by a number of algorithms implemented in either hardware, software, or a combination of both. Therefore, the representation must be suitable for describing both and must yield the target system implementation directly.

The representation must be consistent with the hierarchical, layered view of system structure. It must, therefore, be declarative in that it must express both the structure and function of the target facility to allow algorithmic consistency and performance analysis. It must allow the design to be described in multiple levels of abstraction to accommodate the primitive and internal views of facilities required for stepwise analysis. This feature is especially important to designers since they tend to work "around" in a design rather than in a strictly top-down or bottom-up manner.

Finally, since many of today's computer systems and those proposed for the next generation contain parallel processing capabilities, the representation must allow the specification of parallelism or concurrency in a natural way and be capable of analyzing its effect on consistency and performance.

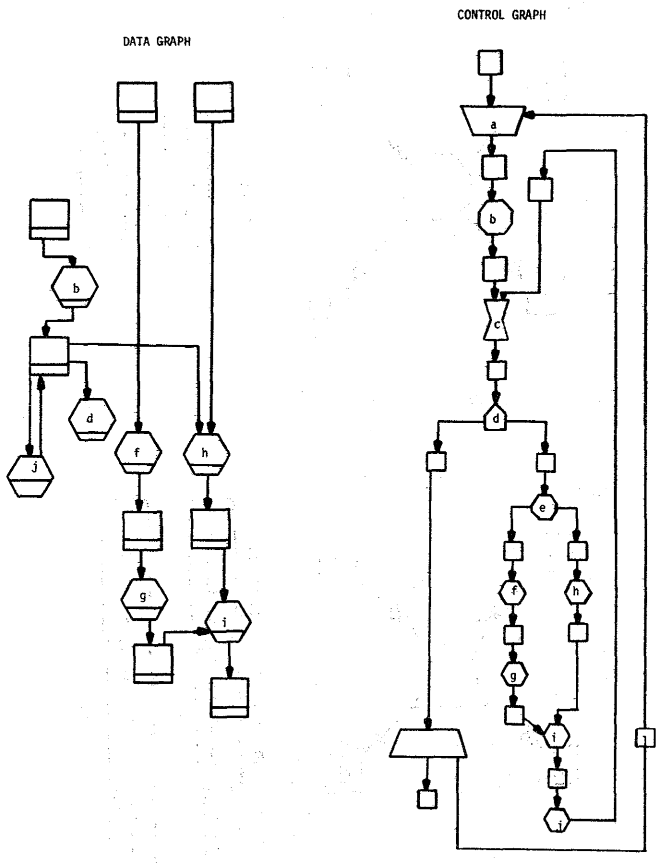
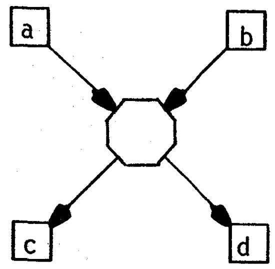


Figure 1—Example of an activity

LOGOS chose a graph-theoretic system of representation which satisfies the above constraints. The system is a synthesis and extension of valuable work done by Petri,[16] Karp and Miller,[17] Holt,[18] Luconi,[19] and others. The extensions were required because (1) LOGOS deals with very large systems and must localize analyses, (2) little work had been done in the representation and analysis of data structures in graph models, and (3) because LOGOS must actually implement rather than merely analyze the target algorithms or systems.

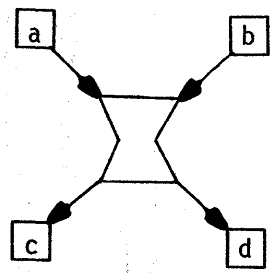
A complete treatment of the representation may be found in References 20 and 21. Briefly, the representation of an algorithm consists of a pair of directed graphs. The data graph (DG) defines the algorithm data structures and the transformations upon them, while the associated control graph (CG) sequences the transformations and defines the control flow. The schema formed by a CG-DG pair is called an activity and will be seen

AND:



$$\begin{array}{c} a\\ b\\ c\\ d \end{array}\begin{bmatrix}1\\1\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$$

OR:

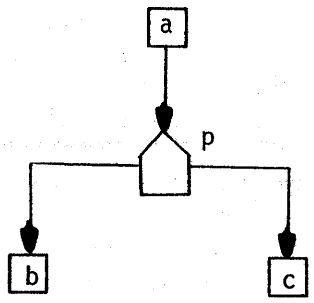


$$\begin{array}{c} a\\ b\\ c\\ d \end{array}\begin{bmatrix}1\\0\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$$

$$\begin{array}{c} a\\ b\\ c\\ d \end{array}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$$

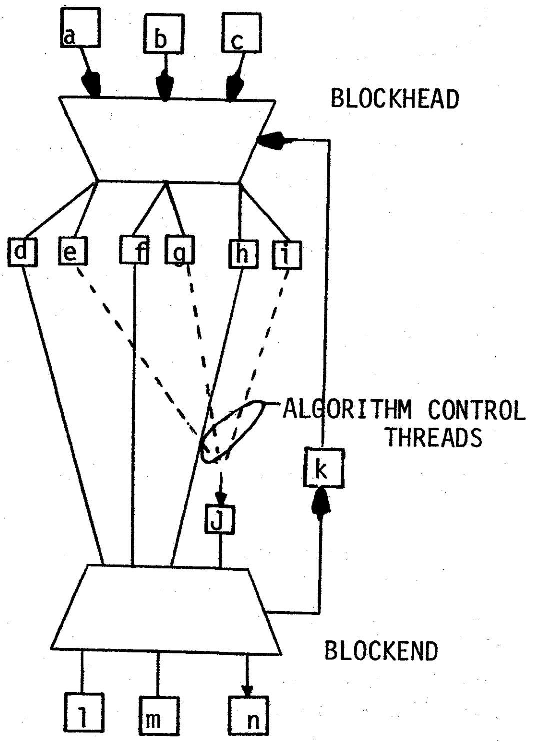
$$\begin{array}{c} a\\ b\\ c\\ d \end{array}\begin{bmatrix}1\\1\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\1\\1\\1\end{bmatrix}$$

PREDICATE:



$$\begin{array}{c} \\ a\\ b\\ c \end{array}\begin{bmatrix}p\\1\\0\\0\end{bmatrix}\quad\begin{bmatrix}-\\0\\1\\0\end{bmatrix}$$

$$\begin{array}{c} \\ a\\ b\\ c \end{array}\begin{bmatrix}\bar{p}\\1\\0\\0\end{bmatrix}\quad\begin{bmatrix}-\\0\\0\\1\end{bmatrix}$$



BLOCKHEAD

$$\begin{array}{c} a\\ b\\ c\\ d\\ e\\ f\\ g\\ h\\ i\\ k \end{array}\begin{bmatrix}1\\X\\X\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}\rightarrow\begin{bmatrix}0\\X\\X\\1\\1\\0\\0\\0\\0\\0\end{bmatrix},\begin{bmatrix}0\\1\\X\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}\rightarrow\begin{bmatrix}0\\0\\X\\0\\1\\1\\0\\0\\0\\0\end{bmatrix},\begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}\rightarrow\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\\1\\0\end{bmatrix}$$

BLOCKEND

$$\begin{array}{c} d\\ f\\ h\\ j\\ k\\ l\\ m\\ n \end{array}\begin{bmatrix}1\\0\\0\\1\\0\\0\\0\\0\end{bmatrix}\rightarrow\begin{bmatrix}0\\0\\0\\0\\1\\1\\0\\0\end{bmatrix},\begin{bmatrix}0\\1\\0\\1\\0\\0\\0\\0\end{bmatrix}\rightarrow\begin{bmatrix}0\\0\\0\\0\\1\\0\\1\\0\end{bmatrix},\begin{bmatrix}0\\0\\1\\1\\0\\0\\0\\0\end{bmatrix}\rightarrow\begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\1\end{bmatrix}$$

Figure 2—LOGOS atomic control operators

to be the static template of a task. Figure 1 is an example of an activity.

The CG consists of two node types: the squares are control variables or c-cells, and the remaining nodes are control operators. Cells must be connected only to operators by directed arcs and vice versa. There are several types of control operators as denoted by the different shapes in Figure 1. Each type of control operator has an associated enabling or transfer function defined on its input and output c-cells.

The DG consists of cells (squares) which represent the information structures of the activity, and data operators which perform the transformation upon them (e.g., Add, Move, Integrate, etc.). Each data operator is associated with a unique control operator which determines when the data transformation may take place. The initiation of a control operator initiates the associated data operator which then reads its input cells (data structures), performs the data function, and writes the results into its output cells. Upon writing, the data operator communicates to the control operator that it has terminated, and the control operator terminates by alterings its c-cells appropriately.

The flow of control in the CG is determined by the values in the c-cells and the nature of the c-operators to which they are connected. The c-operators are defined so that asynchronous or synchronous control and data flow can be represented. The atomic or first level control operators are shown in Figure 2 together with their transfer functions in vector form.

The AND operator of Figure 2 is used to resynchronize parallel control paths and functions analogously to an AND gate in hardware. The OR operator is asymmetric in that if both of its input c-cells contain 1's, the initiation of the operator preserves the 1 in the second c-cell. It will then reinitiate as soon as possible. This "conservation" of 1's is required to insure determinacy, a property of consistent systems with concurrency which will be discussed later. The OR operator is analogous to an OR gate in all other ways.

The PRED operator is the interface between data values and the control flow in the CG. It is a data dependent control branch whose associated data operator performs a test on its input d-cells. The result of this test conditions the branch in the control.

The blockhead (BH) and Blockend (BE) operators are paired to delineate an activity and form the enclosing control for the facility task being represented. The control algorithms must perform the following functions:

(i) arbitrate access to the facility
(ii) provide a communication discipline between the facility and its users
(iii) define the number of concurrent users which may be served by the facility.

The BH/BE pair described in Figure 2 act as a P-V pair. The arbitration algorithm shown is a fixed left-to-right priority, but round-robin and other disciplines have been implemented also. The BH and BE communicate primarily via the feedback c-cell, which initially contains, if it is present, the number of concurrent activations possible. All control flow is restricted to enter and leave the activity via the BH/BE pair with the exception of nested subroutines or procedure calls (i.e., calls upon activities on the same layer of the system) which are controlled by Call/Return operator pairs constructed from a common control primitive.

These control operators may all be constructed from a common primitive control operator whose definition is logically complete. This primitive operator may be realized directly in hardware, but for the purposes of the software engineer, it is sufficient to state that other, higher-level control operators may be constructed from the primitives and placed in a macrolibrary.

The activity of Figure 1 is shown in Figure 3 with interpretations placed upon the data structures and data operators of the DG (these are informal interpretations; a formal syntax will be introduced later). The activity is the representation of an ALGOL 60 FOR statement with a parallel DO ⟨statement⟩ part. When the task is activated, the stepping variable is initialized to 5, and the loop head is passed. Note that there is no data operator associated with the loophead OR. If the predicate is false, the parallel DO ⟨statement⟩ is executed which allows the sequence of data functions ϵf to be time independent of h. The threads are resynchronized at i whose data operator uses common results. n is decremented and the loop is re-entered. Thus we have represented:

FOR N=5 Step−1 until ∅ DO ⟨statement⟩

The 1 in the feedback c-cell indicates that the activity may be initiated only once before terminating.

The nesting of activities on a layer allows the imposition of an ALGOL-like block structure upon the representation. If the activity in Figure 3 were in a block structured environment, data cells A, B, and C might be global to the block while n, m, p, 5, and r are local.

Thus, a collapsed or primitive view of the activity is that of a single control-data operator pair as shown in Figure 4. For most types of analysis performed by LOGOS, the local structure of an activity is analyzed in its internal or expanded form, and the activity is then collapsed. All further interactions with its global environment are analyzed in the collapsed form. In this

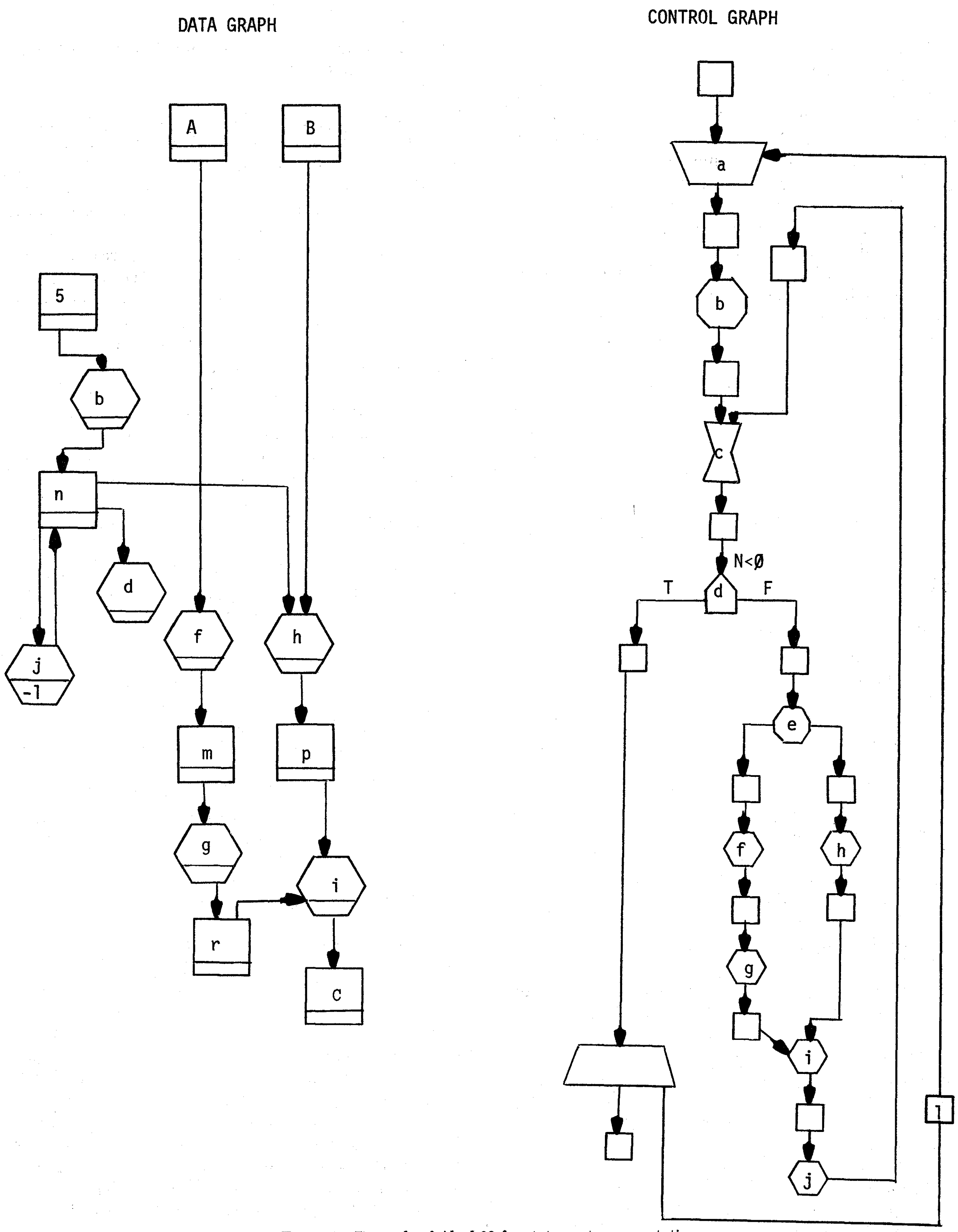


Figure 3—Example of Algol 60 for statement representation

way the analysis of a software (or hardware) system can be carried out in a stepwise, computationally efficient manner.

The imposition of an ALGOL environment is optional, of course, and does not affect the representation itself. To do so does imply the existence of an ALGOL-like run time environment layer which implements the necessary storage allocation and other semantics. A cactus stack is required to keep track of the concurrently active and executing tasks.

The representation may be generalized to allow control variable contents to be non-negative integers with the control operator definitions changed to allow decrementing of input c-cells and incrementing of output c-cells by greater than one. The constraint that output cells be Ø before initiation of the control operator is removed, and the initiation and termination of control and data operators are made distinct to allow multiple initiations of data operators before termination of preceding activations as resources allow. This generalization is useful in describing higher level software processes and hardware such as pipeline systems.

Thus far, only an ALGOL-type level structure has been suggested. Where does layering enter the picture? The concept of layers enters the representation at the data operator. The function performed by a data operator may be truly a system primitive or it may be a "call" on a lower layer facility. That is, its data cells may be parameters to a task existing on a lower layer which is activated by the initiation of the data operator. This may in turn activate other tasks on still lower layers, but the entire data function appears primitive at the layer on which it is initiated. This is an explicit call upon a lower layer. An implicit call would be the activation of the storage allocator upon activation of an activity in a block structured environment.

A formal syntax and semantics for data structures

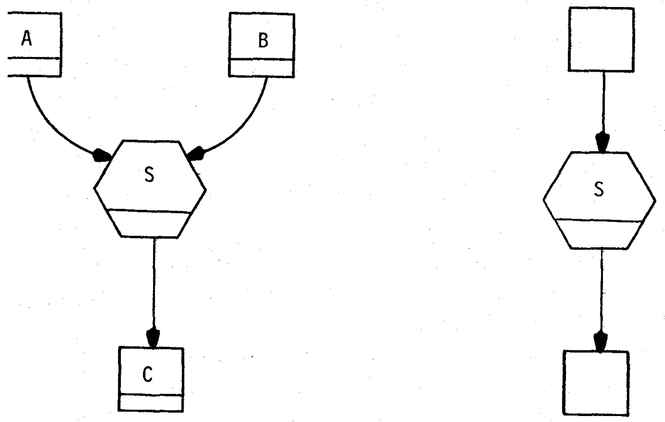


Figure 4—Collapsed activity

```
CDF WORD = (36);
(a)

CDF DOUBL_WD = <CONTIGUOUS> WORD (2);
(b)

STRUCT INTEGER = <WORD>;
(c)

STRUCT LISTEL = <DOUBL_WD> (INTEGER:
  DATA, <WORD> REF LISTEL: PTR);
(d)
```

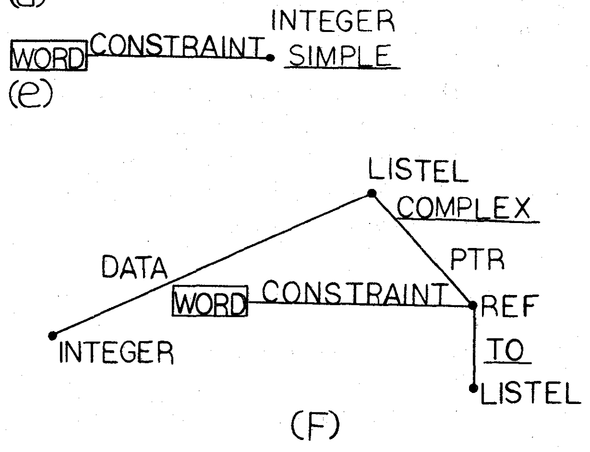


Figure 5—Example of data structure declaration

and a syntax for data operator declarations is being developed. The declarative language is similar to ALGOL 68 and the resulting graphic representation of the data structure descriptors resembles those of Early's VERS.[22] The language consists of six basic building block structures—SIMPLE, MULTIPLE BIT STRUCTURE (MBS), ARRAY, REFERENCE, UNION, and COMPLEX. Examples of SIMPLE structures are integers, reals, etc. MBS's are used to define fields in words or tables. REFERENCE structures denote address variables. UNION is meant in the Set sense, and thus UNION is a place holder for one of a set of structure types. A COMPLEX structure is heterogeneous, consisting of more than one type of basic building block.

Another fundamental concept is that of a constraint. The data structure declarations define logical relationships only. Constraints are used to relate these to physical realities such as words, right half words, etc. These two primitive constraints are: *WORD* and *CONTIGUOUS*.

As an example, consider Figure 5. The length of a WORD is defined in Figure 5a. The constraint

DOUBL_WD is defined in Figure 5b, and an integer is Figure 5c. A complex structure LISTEL (list element) is defined in Figure 5d. It is constrained to occupy a double word, one being an integer, and the other a reference to a LISTEL. The terms DATA and PTR are accessing function names. Figures 5e, f, and g show the graphic representation of the resulting templates. Instances of these data structures may be declared which create descriptors based upon the template but with nodes for allocation information added.

Data operators are defined in terms of the types of their input and output data structures. LOGOS has no formal semantics for data operators, so functional definition is not possible at present. However, an informal semantics is being developed to enhance inter-designer communication and to allow simulation of activities if desired.

The intent of this brief and incomplete description of the structural philosophy and representation system of LOGOS has been to set the stage for a discussion of the use of LOGOS and the analysis tools it provides the software engineer and systems designer.

THE DESIGNER'S ENVIRONMENT

Before discussing the types of analysis tools LOGOS provides the software engineer and system designer it would be helpful to examine the LOGOS environment by describing a typical design scenario.

The systems architects, two or three highly skilled analysts, will either be given or will create a specification for the target system in terms of capabilities, number of users, service times, arrival rates, etc. They will pick the design parameters, block the system into facilities, and identify any obviously common facilities such as memory. In the case of a software system built on an existing computer, the system primitives—the machine instructions—will be specified in advance. The facilities will probably be specified in terms of their external characteristics and will have required performance parameters associated with them.

The information will be given to a group of designers (perhaps the architects themselves) who will define these facilities in the LOGOS design data base from their graphics consoles. The individual tasks performed by facilities will be roughed in, and performance parameters defined for them from those on the facility itself.

The designers may define canonical schemata and store them in a macrolibrary to be inserted and expanded during the design process. These may include structures such as IFTHENELSE and DOWHILE, the primary control elements of structured programming.[23] In terms of hardware, these macros will include the set of MSI functions available to the designers.

As the description of a task becomes complete on a layer, the resulting activities can be analyzed, and the activity collapsed. Common tasks may be grouped into facilities on lower layers and defined accordingly, each having a performance specification derived from above. The designers will make their work available to each other by placing it in a common global data base. Here, lower layer facilities common to several designers may be identified. Duplicate and similar facilities and tasks will be replaced by commonly shared facilities.

Modifications may be evaluated along the way by substituting modified tasks into the data base and re-analyzing the affected portion of the design. This process is continued across descending layers until the data operator functions are in terms of the software primitives of the target system, i.e., machine instructions, implementation language statements, or a trial set of instructions if the entire hardware/software system is being created. Code optimization can then be attempted using one of the newer graph-oriented optimization techniques. Code generation will be discussed briefly in the next section.

If the hardware and/or implementation language exists, actual times will be available for the software primitive operations. These can be reflected up and across layers to determine if the performance requirements were met. If not, redefinition of tasks and/or layer boundaries will be required to attempt to meet the specifications.

If the hardware has not yet been designed, it can be begun at this point with the trial instructions and their required times as the target. The process is identical to the one above, but the lowest layer hardware primitives will be hardware elements such as NAND gates, MSI chips, etc. Once again, actual performance information becomes available and is backed out to the software layers.

If the resulting performance is inadequate, the interpreter (hardware) can be speeded up by increasing the degree of parallelism or upgrading the technology. On the other hand, the hardware/software interface can be adjusted by redefining as primitives certain key data operators which were originally implemented as calls upon lower layer facilities. These procedures may be applied interactively in combination to reach the desired performance, if indeed, it is attainable at all. Once the instruction set is frozen, code may be generated, and the necessary steps taken for implementation of the hardware.

Note that this series of events is a departure from the usual practice of defining the target instruction set as almost the first step in system design and then sending the hardware designers away to build a computer and the programmers to build an operating system. The integrated design approach advocated here should (1) reduce the hardware/software interface mismatches, and (2) allow cost/performance tradeoffs to be intelligently evaluated at the proper time—before commitment to hardware and code.

The data structures, data operators and resulting code of the operating system are simply data in one of the data structures—memory—of the interpreter (hardware processor). This is true of all program/interpreter systems. If the interpreter were not to be implemented in hardware but on, say, an 1108, then the data operator primitives would be 1108 machine instructions, and code rather than hardware would be generated.

In addition to a framework for representing layered systems, LOGOS will provide the designer with several types of consistency and performance analyses. Further, code generation of target system software, and ultimate implementation of target system hardware are goals which appear attainable.

The analyses can be separated into two classes—uninterpreted and interpreted. Uninterpreted analysis implies that no interpretation is placed upon the function performed by the data operators for purposes of the analysis. Thus, uninterpreted analyses deal primarily with the control graphs and are topological in nature.

The addition of parallelism or concurrency to an activity raises several analysis questions. Of primary interest is whether multiple activations of a parallel activity (schema) with a given initial control state (contents of its c-cells) and data values will result in the same final values in a set of "result" locations. This condition is called determinacy and was formulated originally by Karp and Miller.[17] This condition, even after formalization, is mathematically difficult to prove. Another condition, more stringent but easier to verify, has been formulated by Karp and Miller.

1. A schema is determinate if, given an initial state, q_o and an initial set of values, each data location has a fixed sequence of values.

With this condition satisfied, then a schema will surely produce consistent values in the "result" locations provided that the algorithm terminates. Karp and Miller further showed that the above condition is equivalent to the following two conditions.

2. (i) No two data operations can be concurrently enabled to "write" into a common data location. (ii) No data operation can be enabled to "write" into a data location while another data operation is simultaneously enabled to "read" from the same location.

From conditions (i) and (ii), a schema is determinate provided that it is free of "race" conditions of two types. This situation should not startle hardware designers who have always faced this problem.

Karp and Miller gave conditions on a parallel schema which allow determinacy analysis to be conducted on the control graph. The analysis tool is a mathematical construct called a "vector addition system" (VAS); for a given schema, the vectors used have one component corresponding to each c-cell in the control graph. A vector q_o gives the initial control state, and, for each control operator, e, a vector δ_e gives control state changes when control operator e occurs. These change vectors may be derived from those shown in Figure 2, but may be generalized to integers greater than ± 1 for higher level representation. A "tree of nodes" is generated from the root node q_o which corresponds to the tree of attainable control states of the schema. The algorithm identifies loops in the control and may be used for finite and infinite attainable state schemata. A complete treatment may be found in Reference 20.

The resulting tree can be used to determine those control operators which can be simultaneously enabled, and, hence, those data operators which are concurrent.

By examining the input and output data cells of those data operators, conditions (i) and (ii) above can be verified. The blockhead/blockend of the activity in LOGOS limit the scope of the analysis, and thus can limit the size of the tree to manageable size. The activity can be analyzed for determinacy and collapsed. It will then appear as a single operator pair in more global analyses.

The vector addition system can be used to check for proper termination of an activity, i.e., can a control/data operator pair remain enabled after the blockend of an activity is enabled? Further, is the topology of the control graph such that the activity will not terminate? Remember that this is uninterpreted analysis, and, consequently, the results of predicate operations are not known. Therefore, in some cases, all that can be said is that there exists a path which if taken, will result in no termination.

Similarly, by viewing all activities as primitives, a potential recursion analysis can be carried out using the vector addition system. These types of analyses fall into the category of general control path analysis, and additional algorithms in this family can be identified and easily implemented using the VAS.

A major weakness in the integrity of computer systems has been the management of system resources and the prevention of system deadlocks. This is particularly true in systems with a high degree of real or apparent concurrency. This problem has been extensively studied, and much insight has been gained.[6,11,24,25] Holt[25] has developed graph models for deadlock and resource allocation which are directly applicable to the LOGOS environment. Resources are represented as control cells, and a topological analysis using adaptations of Holt's results can be performed. Once again, a layered structure tends to limit the scope of analysis.

System performance analysis depends upon knowledge of arrival rate and service request distributions, and, thus is only as good as the model load. However, actual path transit times can be computed in the LOGOS environment, and if model service request distributions and arrival rates are available, performance statistics can be gathered before implementation using a combination of path analysis and simulation, if necessary.

Interpreted analysis deals with the correctness of the algorithms used in implementing the activities. At present, LOGOS has no automated solution to the program correctness problem. The layered structure of target systems, together with the communications disciplines enforced by the syntax of the representation and the various other analysis algorithms tend to assure logical and structural consistency. However, a logically consistent, *but* incorrect algorithm is undetectable. Current work by Scott and Strachey,[26] leading toward a formal mathematical theory of hierarchical systems and semantics, may well be the answer. Results of this work could be adapted to replace LOGOS current data graph syntax and semantics and provide a certifiable representation. In the interim, interpreted data analysis algorithms based upon the functional attributes of the data operators are being considered. For example, a data operator must access data structures of the appropriate type and compute results which correspond to the types of output data structures to which it is connected. This is useful in analyzing data functions which are implemented by interlayer facility activations. In critical areas, actual simulation of the algorithms in question may be performed directly.

Finally, if a global semantic such as ALGOL 60 or FORTRAN is imposed, environmental consistency algorithms such as scope of reference can be included modularly.

CURRENT STATE OF LOGOS

The LOGOS system is being implemented on a Digital Equipment PDP-10 with a Bolt, Beranek and Newman paging box and TENEX executive system. The primary graphics terminals are two IMLAC PDS-1 display systems which communicate with the PDP-10 at 9600 baud. The implementation language is SAIL (Stanford Artificial Intelligence Language). A multi-designer data base management system is being implemented using the LEAP associative data structures of SAIL and the TENEX virtual memory facilities. The system provides for local (single user) and global (shared) data bases with linking between local and global information in a controlled manner. The data base management system is based upon earlier work by M. Pliner.[27]

The graphical representation system is implemented together with the following analysis algorithms: graphical syntax checking, determinacy, halting and termination, and repetition freeness. Implementation of generalized control path analysis is also under way.

The remainder of the control analyses, deadlock and resource allocation, are scheduled to be implemented and integrated by September 1973. It should be noted here that all of the analysis packages are modular and act upon the standard internal representation, thus allowing new packages to be added when necessary.

The implementation specifications for the data structures and data operators are scheduled for completion in December 1972, and implementations should be complete by September 1973 along with the associated analysis routines. These analysis routines assume a FORTRAN environment with a static block structure but may be replaced if another semantic is chosen.

Performance analysis algorithms should be implemented and integrated by September 1973.

Thus, with the very major exception of a formal semantics and corresponding attack on program correctness, LOGOS is scheduled to have a running representation and analysis system by September 1973.

The implementation of target systems requires the production of a code generator for the software and a "hardware compiler" for the hardware portions of the representation. Here again, Scott's work may provide a general solution to the semantics problem for the code generators, but even without such results, if the software primitives in the data graphs are machine language instructions of the target machine, code generation becomes rather straightforward. In addition, the graphic form of the program tasks will allow application of the newer optimization techniques to the target software. A first cut code generation scheme for sequential (rather than parallel) systems should be implemented in early 1974.

Rather than re-create a "hardware compiler" which would require 30-50 man years, LOGOS has chosen to interface with existing hardware CAD systems at the logic equation/logic diagram level. Although much of the information which could help in optimization of the hardware will be lost in going to the equations, the time scale and scope of the project preclude attacking the hardware CAD problem directly. It is felt, however, that the graphic representation may provide helpful insight in the partitioning and placement operations of hardware CAD, and those problems will continue to be studied. The hardware equation/diagram outputs are scheduled for September 1974.

In parallel with these efforts, an attempt is being made to define one or more programming languages to serve as alternate external representations of the target system rather than the current graphical representation. This is being done because some programmers may feel uncomfortable with the graph form, and because the human engineering and scope management problems become significant as the complexity of the target graphs increases.

The LOGOS representation has been used off-line to describe various types of small systems and subsystems including a PDP-8. The resulting descriptions are concise, and being able to see both the structure and function of the systems in one "picture" aids in understanding the target system.

With regard to implementation, the resident executive in the IMLAC display processors was designed according to the LOGOS structural philosophy.

The IMLAC system provides the designer with facilities of (1) creating a picture and designating it a subroutine for transmission to the PDP-10, (2) editing a subroutine, and (3) deleting a subroutine. The system exists on six layers as shown in Figure 6. The lowest is the PDS-1 hardware used by all higher layers. The next

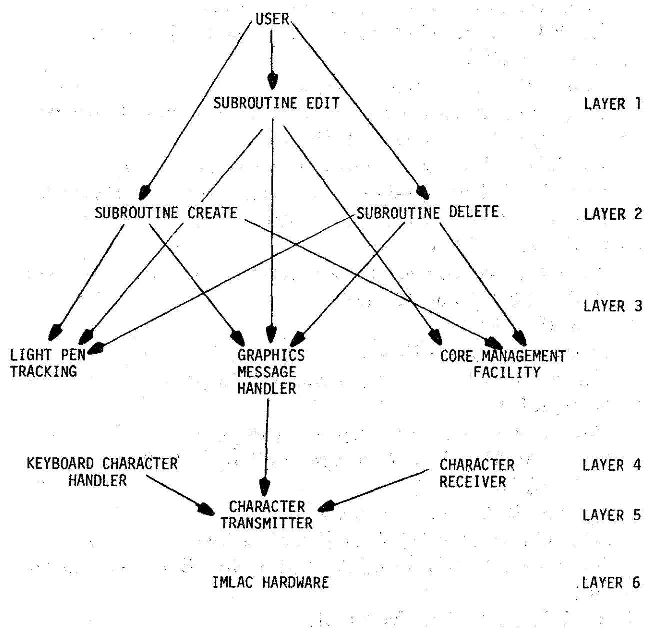


Figure 6—Layer structure of Imlac executive

layer facility is the character transmitter (all messages, text and graphics are sent to the PDP-10 as multiple character strings). Layer 4 contains the keyboard character handler and the character receiver both of which are users of the character transmitting facility. The character receiver uses the character transmitter facility to control the transmission of characters from the PDP-10 to the IMLAC. The next layer has three independent facilities—the light pen tracking facilities, the graphics message handler, and the core management facility. All of these facilities are used by the facilities on layers 1 and 2, the subroutine edit and subroutine create and delete facilities. The communications discipline between the facilities are well-disciplined according to LOGOS design principles.

The design and implementation of the executive required about six man months of effort. It occupies approximately 3000 words in IMLAC core and was coded in assembly language. As with the 'THE'[6] and 'VENUS'[7] systems, coding errors were discovered, but few logical errors were committed in the design. These proved easy to identify and correct.

CONCLUSION

The aim of Project LOGOS is to provide the computer system designer with a computer-assisted design environment in which good engineering practice can be applied to large-scale target systems and verified after the fact. The basis of this good engineering practice is a structural view of computer systems which is a generalization of Dijkstra's[2] and Mills'[23] structured programming for sequential software. Dijkstra's 'THE' system[6] is a result of this philosophy as is the VENUS system.[7] Both these and the IMLAC executive have demonstrated the payoffs of a well-disciplined approach to structure. They were implemented in a fairly short time by small design groups (VENUS required about 6 man years for the design and implementation of the operating system and the support software). They were easily checked out and modified, and have proven to be stable, reliable systems. The primary contribution of LOGOS in this area is that it provides a uniform, analyzable representation in which to express these otherwise abstract notions of system structure, one which leads directly to the implementation of the target software or hardware. It also allows the designer to express the maximum degree of real and apparent concurrency in his target system and provides the analyses required to evaluate its effect.

Both 'THE' and VENUS are small operating systems implemented on small to medium scale machines, yet even they were found to contain a few errors resulting from breaches of discipline. True, these errors were easily corrected, but as the size and complexity of the operating system and hardware increases, the difficulties of enforcing the disciplines, detecting errors, and correcting them without introducing more will increase nonlinearly. It is because of this complexity explosion that a CAD environment such as LOGOS is required for large scale systems.

A LOGOS-type system can provide several other advantages to the software engineer and system designer. First, because performance measurements can be made before rather than after implementation, modifications to the system can be proposed and their effects evaluated economically. In particular, the final positioning of the hardware software interface can be postponed until quite late in the design cycle and can be made a true function of performance vs. cost.

Second, the design team will tend to be smaller. The computer will act as the "bookkeeper" and will perform many of the analyses which have traditionally been attempted manually or not at all.

Third, the increased degree to which a target system can be certified before implementation (even without formal semantics) should reduce the integration and checkout cycle significantly. It may also be possible to produce more complete diagnostics in a LOGOS environment since the entire system description as well as its implementation is stored within the design data base. This is an area for continued research.

Finally, although this hints of "big brother," valuable management and scheduling information can be extracted from such a system. The effectiveness of designers, the times required to complete various portions of the system, etc., could be used in estimating, staffing, and scheduling future systems.

LOGOS is an open-ended system. Although a first producing system will be complete in 1974, it is expected that the users themselves will enhance, modify and tailor the design environment to their needs as new technology becomes available.

ACKNOWLEDGMENTS

The LOGOS design environment described in this paper is the result of work done over the past three years by Professor E. L. Glaser, principal investigator; Dr. F. T. Bradshaw; S. Katzke; the author and several others. In particular, much of the philosophy of system structure which underlies the LOGOS system was articulated by Dr. Bradshaw, and the syntax and semantics of target system data structures and data operators were developed by S. Katzke during his doctoral research.

REFERENCES

1 F G HEATH C W ROSE
The case for integrated hardware/software design with CAD implications
IEEE Computer Conference Digest September 1972

2 E W DIJKSTRA
EWD249—notes on structured programming
T. H. Report 70—Wsk—03
Technological University Eindhoven Netherlands April 1970

3 T BREDT
A model for parallel computer systems
Technical Report No 5 STAN-CS-70-160 Stanford University April 1970

4 C G BELL A NEWELL
Computer-structures: reading and examples
McGraw-Hill Book Company New York New York 1971

5 M BARAY Y H SU
A digital system modelling philosophy and design language
Proceedings Eighth Annual Design Automation Workshop 1971

6 E W DIJKSTRA
The structure of the T.H.E.—multiprogramming system
Comm ACM Vol 11 No 5 May 1968 pp 341-346

7 B LISKOV
The design of the VENUS operating system
Comm ACM Vol 15 No 3 March 1972 pp 144-149

8 C D MARSH
Automation of the design and manufacturing of a large digital computer
IEE Electronics & Power October 1970 pp 375-379

9 M R CORLEY
The graphically accessed interactive design of thermally stressed pipe systems
Proceedings Ninth Annual Design Automation Workshop 1972

10 F T BRADSHAW
Some structural ideas for computer systems
IEEE Computer Conference Digest September 1972

11 E W DIJKSTRA
Co-operating sequential processes
Programming Languages ed F Genuys Academic Press 1968

12 M J SPIER E I ORGANICK
The MULTICS interprocess communication facility
Second ACM Symposium on Operating Systems Principles Princeton University October 1969

13 E W DIJKSTRA
A constructive approach to the problem of program correctness
BIT Vol 8 1968 pp 174-186

14 C A R HOARE
Proof of a program FIND
Comm ACM Vol 14 No 1 January 1971 pp 39-45

15 N WIRTH
Program development by stepwise refinement
Comm ACM Vol 14 No 4 April 1971 pp 221-227

16 C A PETRI
Kommunikation mit automaten
Schriften des Reinsch-West Falischen Inst
Instrumentelle Math und der Universitat Bonn Nr 2 Bonn 1962

17 R M KARP R E MILLER
Parallel program schemata
Journal of Computer and System Sci 3 1969 pp 147-195

18 A W HOLT F COMMONER
Events and conditions an approach to the description and analysis of dynamic systems
Third Semi-annual Technical Report Part II For the Project Research in Machine-Independent Software Programming Applied Data Research Inc April 1970

19 F L LUCONI
Asynchronous computational structures
Doctoral Thesis MIT Cambridge Mass January 1968

20 F T BRADSHAW
Structure and representation of digital computer systems
Jenning Computing Center Report No 1114 Case Western Reserve University Cleveland Ohio January 1971

21 C W ROSE
A system of representation for general purpose digital computer systems
Jennings Computing Center Report No 1113 Case Western Reserve University Cleveland Ohio August 1970

22 J EARLY
Toward an understanding of data structures
Comm ACM Vol 14 No 10 pp 617-627

23 H D MILLS
Mathematical foundations for structured programming
FSC72-6012 Federal Systems Division International Business Machines Corporation Gaithersburg Maryland February 1972

24 A N HABERMANN
Prevention of system deadlocks
Comm ACM Vol 12 No 7 July 1969 pp 373-385

25 R C HOLT
On deadlock in computer systems
Doctoral Dissertation Cornell University Ithaca New York January 1971

26 D SCOTT C STRACHEY
Toward a mathematical semantics for computer languages
Tech Monograph PRG-6 Oxford University Computing Laboratory August 1971

27 M S PLINER
PDMS—a primitive data base management system for representing structured data in an information sharing environment
Doctoral Dissertation Case Western Reserve University Cleveland Ohio September 1971