SLAC-PUB-573
March 1969
(MISC)

HARDWARE AND SOFTWARE FOR EFFECTIVE MAN-MACHINE INTERACTION IN THE LABORATORY*

W. F. Miller

Stanford University, Stanford, California

I. INTRODUCTION

In the last few years we have transposed a certain amount of our instrumentation problems from one of hardware engineering to one of software engineering. The advantages of software instrumentation have been discussed in a number of conferences, but the realization of these benefits has been slower and more costly than anticipated. We commonly hear the complaint that we underestimate the software effort or that software failures have hampered the full utilization of a system. I should like to call attention to a number of research activities under way which are directed toward helping with the software problem. In particular, I refer to the work on META-SYSTEMS. Meta-systems are systems for describing and implementing other systems, for example, for describing and compiling compilers, operating systems, console languages, etc.

Meta-systems are in contrast to generalized and/or extensible systems. In the latter the idea is to be able to add new features at run time. Both meta-systems and extensible systems require higher level description languages that are sufficient to include the features to be added. The difference resides in the method of implementation. In one case you re-compile to get the new features; in the other case you make run time extensions that have to be interpreted by the parent system. An excellent survey and critical evaluation is contained in the paper by Feldman and Gries.[1]

Another area of research that should make contribution to the design and implementation of man-machine systems is the work on models of control. There is no good reference that I can give you other than the bibliography of references for a course I give on Control of Computing Systems.[2] In a number of papers at this conference we hear reference to the need for systems analysis. The systems analysis and model building in the area of control go hand-in-hand.

I believe that the work on meta-systems and models of control will be important on the hypothesis that the next few years will see refinement of technique and increased sophistication in applications, some of which will introduce changes of scale and some will require greater generalization. In all cases there will be a heavy dependence on software, so that software engineering will be a foremost concern. I believe that we can expect a greater trend toward random access secondary storage (disks, drums, etc.) as opposed to sequential secondary storage (tapes). This transition will be essential for those experiments which expect to interact with large volumes of data on-line. I include both cases where (1) an experimenter interacts by means of a console with a large data base or (2) a control program must interact with a large data base to carry out its control function. I believe very much in the experimental approach to the development of the man-machine systems. We can design, then redesign and re-compile systems as our experience dictates by use of a meta-compiler. The meta-compiler has the same advantage for the system designer and implementer as FORTRAN has for the applications programmer.

*Work supported by the U. S. Atomic Energy Commission

(Presented at the Conference on Computer Systems in Experimental Nuclear Physics, Skytop, Penn., March 3-6, 1969)

II. META-SYSTEMS

The goals of the meta-systems research are: (1) to find description languages that represent the processes being studied, (2) to find machine representations of these descriptions, and (3) to find useful implementations of the meta-system for the analysis of the system and/or the control of its construction (compilation).

The order of increasing complexity of tasks for a meta-system is, in my experience, shown in Fig. 1.

I can point to highly successful utilization of meta-systems for area (1), moderately successful utilization in areas (2) and (3), and rather less use in areas (4) and (5). Use in areas (4) and (5) is increasing as we gain better understanding of these processes. My own involvement is in (4) and (5) and it is in these areas that I shall illustrate the functioning of a meta-system.

I think it is clear why complexity increases from (1) to (5). As we go from (1) to (5) we become increasingly more involved in the dynamics of the computation. Also the data structures which form the low level representation of the processes become more complex from (1) to (5). In area (1) logical correctness is the main concern. Efficiency considerations are mainly related to code density. Areas (2) and (3) are concerned with storage allocation, either static or dynamic, whereas areas (4) and (5) are concerned with allocation of many resources and with timing considerations, i.e., scheduling. The need for experimentation in these latter areas is greater but the process description and modeling has been more elusive.

Intellectually, meta-systems contribute to better understanding of the programming processes. Practically they help one get systems "on the road" faster. The key idea is to use the meta-system to get a quick and dirty system "on the road" and then tune it in the areas where it needs improvement. A number of the manufacturer supplied compilers that you are using on your systems have been built that way. You get the quick version going and make certain that it is logically correct and then concern yourself about efficiency. The success of this technique in area (1) of Fig. 1 has been so enormous that I shall not dwell on it here.

To date more effort has gone into meta-systems for compilers than for other areas. Some of the more successful ones known to me are listed in Fig. 2. For more complete comment readers are referred to the first paper in the bibliography. [1]

In addition to the special systems mentioned above, a number of compilers have been written in ordinary FORTRAN, ALGOL, or PL/I. Livermore has been writing their FORTRAN compilers using an extended version of FORTRAN for several years. The ACME PL time-sharing system at STANFORD was written in the IBM H-LEVEL FORTRAN. IBM is now putting a substantial effort into the development of PL/I as a system writing system.

Let me point out that there is no particular magic in the meta-compiler operating on the machine for which it is compiling a system. The same statement is true for ordinary compilers and assemblers. There are a number of compilers (and assemblers) that compile on one machine for running on another. It is particularly convenient to compile and get a program to be syntactically correct using an interactive time-sharing machine before running on another machine. This may suggest to many a way out of the problem of having to buy a large amount of resource just to run a FORTRAN compiler.

Let us turn our attention to areas (3) and (4). The system problems here include the principal system problems for on-line data acquisition and control systems. The investment of effort to make a meta-system for either graphics or operating systems becomes rather large and is not likely to be done by a small or even intermediate size applications group. I shall say more about the size of the effort later.

Figure 3 lists several meta-systems that have been used for compiling graphic control and/or operating systems. All have been used in some form except item (4), GLAF, which is still under development. Perhaps the most successful in the sense of widest use of its products is the extended ALGOL of BURROUGHS. This was used to write the Master Control Program (MCP) for the disk-file monitor and succeeding monitors on the B-5500.

A much more ambitious project is the MIT PROJECT MAC Multics System. A special version of PL/I was written (EPL) which was in turn used as the meta-compiler for the Multics operating system. The Multics system is designed for great generality, and it is currently struggling into operation and use. The effort has been much berated by some, but few people appreciate the enormity of the task. I believe they made the correct decision and seriously doubt if they could have been operational at all had they gone the path of machine language code.

III. GLAF, A GRAPHICS META-SYSTEM

My students and I are developing a Picture Calculus[10,11] that forms a model for picture processing, including both graphics and pattern recognition. In order to experiment with varying forms we need a meta-system for compiling systems based on our model.

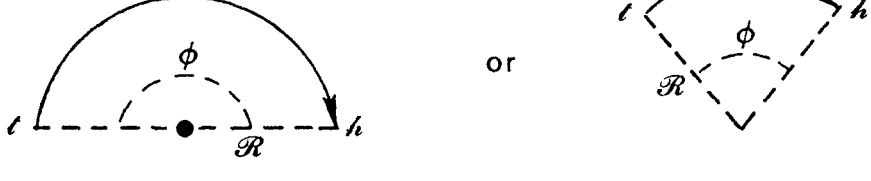
Let me describe very quickly a particular instance of our general model of picture processing. Our model is a linguistic or grammatical model. We have primitive elements which are the basic entities from which pictures can be formed and a set of rules or operations for associating these primitives together. The rules for allowable associations are given in terms of a grammar.

The grammar can generate all allowable pictures, that is all pictures that can be recognized or generated by this particular instance of our general model.

The picture description language, PDL permits one to describe the concatenation of the primitive elements of a picture. The choice of primitives will depend on the particular application. A primitive may be defined as any two-(n-) dimensional object with two distinguished points, a tail and a head. In general, a primitive will be a picture that can be handled more conveniently as a unit than in terms of its subparts. Primitives are concatenated together only at their heads and/or tails. Abstractly, a picture described in PDL can be represented as a labled, directed graph with the primitives as directed edges pointing from tail to head. We often refer to the graph of a picture.

Picture primitives are specified in terms of primitive classes which in turn are specified by an attribute list. The attribute list contains the class name, a tail and head specification, and an arbitrary list of additional attributes. As an example we may define the primitive class of arcs of circles of any radius, negative curvature, and arc less than 180^{o}. The attribute of the primitive class ARC has the form

ARC ≡ (ARC, Counterclockwise limit, Clockwise limit, Curvature < 0, $\phi < 180^{o}$)



This is a specific instance of the general form

PRIMITIVE CLASS ≡ (< NAME>, < tail specification>, <head specification>, <1st attribute>, <2nd attribute>, ... <nth attribute>)

A superscript label identifies a particular member of a primitive class. An element $ARC^{\dagger}$ of the class ARC, for example, has a value list containing specific values for each attribute on the attribute list of the class, e.g.,

$$\text{VALUE}(ARC^{\dagger}) = \left(ARC^{\dagger}, \vec{X}_{TAIL}, \vec{X}_{HEAD}, (\text{CURVATURE} =) -2, (\phi =) 60^{o}\right)$$

There may be redundant information on the attribute list. Some of the attributes may be irrelevant for some uses. However, for generality of form and application all information is retained.

We allow blank (invisible) and don't care primitives. They may be used for connecting disjoint parts of a picture or to specify the geometrical relationship between parts of a picture. One special primitive, the null point primitive, λ, plays a special role. It consists only of a tail and head with identical position and it is represented as a labeled node in a graph.

The syntax of PDL is given as follows:

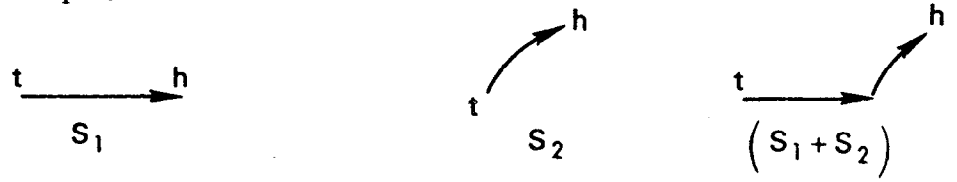
A sentence, S, in the language is defined by

1. $S \rightarrow p \mid (S \theta S) \mid (\sim S) \mid (\overline{S}) \mid T(\omega)S \mid S^{\ell}$
2. $\theta \rightarrow + \mid \times \mid - \mid * \mid \sim$
3. p is a primitive class
4. $\{+, \times, -, *\}$ are concatenation operators described below
5. $\{\sim, \neg, T(\omega)\}$ are unary operators
6. ℓ is a label designator

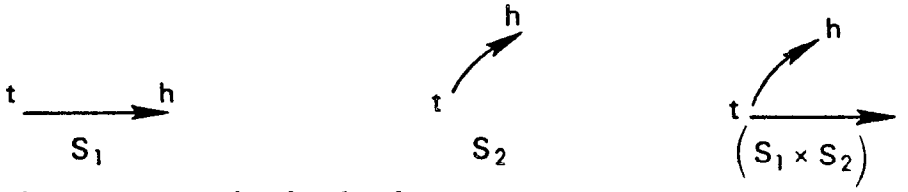
The concatenation operators +, ×, -, *, and ~ are binary operators defined below. In all cases

$$\begin{cases} \text{Tail}\ ((S_1 \theta S_2)) = \text{Tail}\ (S_1) \\ \text{Head}\ ((S_1 \theta S_2)) = \text{Head}\ (S_2), \quad \theta \in \{+, \times, -, *, \sim\}. \end{cases}$$

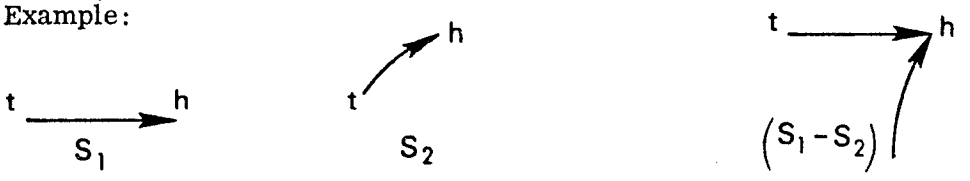
7. The + operator: head to tail:
   $(S_1 + S_2)$ concatenates the head of S_1 to the tail of S_2.
   Example:



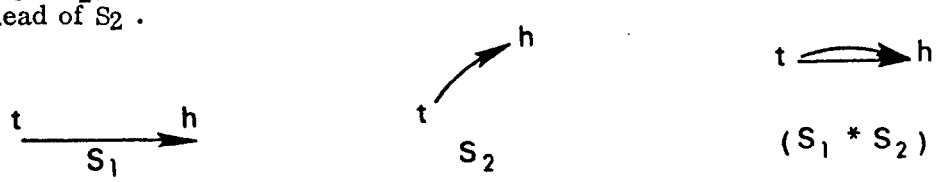
8. The × operator: tail to tail:
   $(S_1 \times S_2)$ concatenates the tail of S_1 to the tail of S_2.
   Example:



9. The - operator: head to head:
   $(S_1 - S_2)$ concatenates the head of S_1 to the head of S_2.
   Example:



10. The * operator: head to head and tail to tail:
$(S_1 * S_2)$ concatenates the tail of S_1 to the tail of S_2 and the head of S_1 to the head of S_2 .



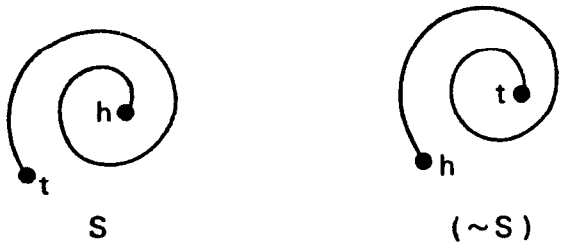
The * operation may be undefined for some combinations of structures, S, just as the arithmetic operation a/b is undefined for certain values of a or b.
Example:

VECT1 ≡ (VECT1, tail at origin, head at upper right, unit vector in first quandrant)
VECT2 ≡ (VECT2, tail at origin, head at lower left, unit vector in third quadrant)
(VECT1 * VECT2) is undefined.

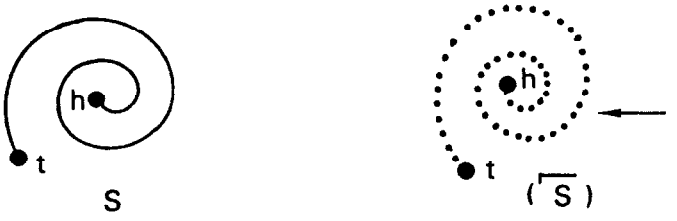
11. The binary ~ operator:
$(S_1 \sim S_2) \equiv (S_1 + (\sim S_2))$
That is, the binary ~ means simply + the unary ~ of S_2

The operators ~ and ⌐ are unary operators defined as follows:

12. The unary ~ operator: switches head and tail.
Tail ((~S) = Head (S)
Head ((~S) = Tail (S)
The structure remains the same.
Example:



13. The ⌐ operator: blanks out all points.
Head ($\overline{S}$) = Head (S)
Tail ($\overline{S}$) = Tail (S)
All points in the structure are turned to null points.
Example:



Let us consider a class of particle physics interactions starting with a negative particle which may pass through the picture or may scatter from a positive particle or may decay into a neutral and another negative particle. Then

14. $T_- \rightarrow t_- | (t_- + (T_- \times T_+) | t_- + (T_- \times T_n))$

where T_- is a negative track with all subsequent events, t_- is a primitive negative track, T_+ is a positive track with all subsequent interactions, and T_n is a neutral track (not seen) with all subsequent interactions.

Let us consider only positive particles that continue through the chamber, that is,

15. $T_+ \rightarrow t_+$

where t_+ is a primitive positive track.

Let us consider neutral particles that may decay into pairs, that is,

16. $T_n \rightarrow t_n \,|\, (t_n + (T_+ \times T_-))$.

A picture that must start with a negative track would be represented by:

17. $P \rightarrow T_-$
$T_- \rightarrow t_- \,|\, (t_- + (T_- \times T_+)) \,|\, (t_- + (T_- \times T_n))$
$T_+ \rightarrow t_+$
$T_n \rightarrow t_n \,|\, (t_n + (T_+ \times T_-))$

Figure 4 shows a picture that would be generated by sentences in the grammer (17).

The object of this discussion is not to tell you how linguistic models of picture processing work but to show you the kind of objects we wish to manipulate with the meta-system.

We have two goals in mind. First we are seeking device independence and secondly the ability to redefine such quantities as the primitive definitions, the syntax or semantics of the operations, the features in the control language, or the basic low level data structure which defines the pictures. We expect to be able to redefine our system, re-compile it and thus have a new system.

GLAF provides the facilities to specify and implement a system. These facilities can be divided into four parts: (1) a graphic description language, (2) a definition language, (3) a control language, and (4) a basic display file.

The graphic description language is used to specify the linear string graphic language of interest, e.g., the PDL just shown. The DL is used to define the naming and specifying of the construction of the primitives defined in GDL; the CL specifies the operations of the system such as interactive or noninteractive, function calls for such control operators as initialize, display, and construction of valid strings in GDL; the BDF is the target of the translation from a given input language. The BDF can be viewed as low level graphic language which is device independent. The formal structure is specified in the semantics of GDL and in the DL specification of the primitives; the user then supplies a driver to translate the BDF to the particular display device to be used.

Figures 5 through 8 show the general structure of GLAF. For greater detail one is referred to Reference 8.

IV. MODELS OF CONTROL

GLAF is a meta-system oriented particularly toward graphics and patterns recognition. Meta-systems for more general purpose control systems are being developed as an outgrowth of work on models of control.

This paper is not intended as a comprehensive review of control models but is intended only to illustrate the type and nature of this research. The four models discussed below are sufficiently illustrative to indicate the various directions of the work.

A. Dijkstra[13]

Dijkstra has been concerned with providing the primitive operations for control of shared processes. He has provided primitive operators with which one can guarantee the proper synchronization of two cooperating processes. Two particular dangers are (1) inadequate exclusion and (2) mutual exclusion. There are times when it is necessary to guarantee that all processes except one distinguished one are excluded from interacting with certain data or devices while the distinguished process is interacting (critical section). Dijkstra "semaphores" provide the mechanisms for providing this exclusion from critical sections. The contrary danger, however, is that the synchronization mechansim will permit mutual exclusion. Mutual exclusion arises when each of two (or more) processes lays claim to a critical section only to find when it checks its claim that the other process has a claim thus excluding both processes. Such mutual exclusion is also prevented by the Dijkstra model of synchronization.

Dijkstra used his model as the basis of the design for "THE"-Multiprogramming System.[14] The model contributes to greater reliability and simplified debugging by prior knowledge of the logical soundness of the programs written.

B. Dennis and Van Horn[15]

Dennis and Van Horn develop a set of primitive operations which not only provide for protection of critical sections but also permit the passage of control from one computation to another. This type of passage of control is essential to the development of a "utility" system which expects to provide for common use of data and programs.

Many of the concepts advanced by Dennis and Van Horn have been utilized in the design of MULTICS.[16]

Both Dijkstra and Dennis and Van Horn have been concerned with the development of primitives and schemes for their use that provide for synchronization, various types of access control, and the passage of control from one process to another. Neither of these models provides global rules or a syntax for use of these primitives. A higher level language which incorporates these concepts might be developed and used for system writing.

C. Karp and Miller[17]

Karp and Miller present a flow graph model of computation within which the sequencing of the processes of a computation is determined by the flow of data through the processes. Similar work is being done at MIT and at UCLA by a number of participants and at Stanford by the author and his students. A brief review of the work is contained in Reference 18.

In the flow graph models, the nodes of the graph represent processes (computation, data movement, etc.) and the edges represent storage queues for transfer from one process to another. An example of a computation graph for the original Karp and Miller model[17] is given in Fig. 9.

The careful definition of the conditions for initiation and termination of a process permit one to prove a priori that a program written in the model is determinate. By determinate we mean that no matter what the speed of the various processes nor what is the order of initiation of parallel processes, the sequence of outputs is always the same. This model has rather far reaching implications and a great deal of work is being directed toward exploiting the concepts.

D. Dahm, Gerbstadt, and Pacelli[19]

These authors developed a set of primitive operations for management of resources in an event driven operating system. Similar ideas are contained in the work of Brown[20] et al., and Russell.[21] Dahm et al., do not present scheduling and resource allocation

algorithms but propose a framework within which resource management problems can be expressed.

The main concepts are concerned with linking events to processes and requesting and releasing of resources.

V. HARDWARE AIDS

Two important hardware aids to system development are the read only store (ROS) for microprogramming and the cathode ray tube display for system debugging and analysis.

The read only store (ROS) has an impact on software design. The read only store can be used to program control of the machine in the main processor or in peripheral processors. That is, the hard wired functions of decoding instructions, gating data, and instruction sequencing are programmed in a read only store with a very simple execution code of its own.

The advantage to the software designer is the freedom of choice in instruction repertoire and data accessing available to him. It is a clear matter of convenience to be able to introduce a new machine (macro) instruction but there are advantages beyond that. It may be important for synchronization purposes to be able to make certain operations indivisible.

Also the software designer may wish to choose his own default conditions for particular operations such as underflow or overflow. Perhaps the greatest advantage of all is the ability to choose a data accessing scheme to suit the need. By choice of micro code one can consider the computer as a stack machine or a multiple address machine or a variable word length machine. That is, the designer can choose the basic data structure of the machine through his choice of access control. This offers great advantage to the designer of systems for non-numeric processing.

The cathode ray tube display is a great advantage for the analysis and debugging of systems. One can in essence simulate any kind of control panel he so desires with the CRT display. This makes it possible to watch the flow of control through a system at whatever level is necessary to analyze performance or to examine for faults. For example, one can program the CRT to show how data or jobs queues are building up and watch the effect on them by changes in allocation algorithms. This offers the systems programmer a powerful tool for tuning his system. There are a number of debugging packages that have used CRT's to trace and control breakpoints in programs.

VI. COSTS

It would not be fitting to engage in a discussion of systems without at least a brief note on costs.

Reliability is still a major complaint with software. Meta-systems and microprogramming are clear ways toward greater reliability. The advantage for small systems may be slight. The advantage for large systems has already been demonstrated to be substantial.

On even the smallest systems we usally find one or two man years investment in general programs, library, etc. On very large systems the software effort per installation is ten or fifteen man years per year. These costs are still less than the hardware investment, but it is not uncommon for the software cost to be from 20% to 50% of hardware costs. Table I shows the hardware costs and the man months of programming for a number of small and intermediate size systems at SLAC. If one takes $30,000.00 as the man year cost (including overhead) for programming we quickly see that software is a significant cost of the system.

REFERENCES

1. J. Feldman and D. Gries, "Translator Writing Systems," C. A. C. M. 11, 77 (Feb. 1968).

2. W. F. Miller, Bibliography of Papers for CS246, Control of Computing Systems, Computer Science Department, Stanford University, Stanford, California (1968).

3. W. M. McKeeman, J. J. Horning, and D. B. Wortman, XPL Users' Manual, Stanford University Computation Center, Stanford University, Stanford, California (1968).

4. D. T. Ross et al., AED-O Programmers Guide and User Kit, Electronic System Laboratory, MIT, Cambridge, Massachusetts; and
D. T. Ross, "The AED Approach to Generalized Computer-Aided Design," Proc. 22nd Nat'l. Conf. of ACM, (1967) 367.

5. L. Mondschein, "Vital Compiler Reference Manual," TN-1967-1, Lincoln Lab. MIT, Lexington, Massachusetts (1967).

6. R. J. Pankhurst, "GULP—A Compiler-Compiler for Verbal and Graphic Languages," Proc. ACM Nat'l. Conf. (1968) 405–421.

7. James E. George, "Calgen—An Interactive Picture Calculus Generation System," Report No. CS-114, Computer Science Department, Stanford University, Stanford, California (1968).

8. James E. George, "The System Specification of GLAF," Report No. GSG-61, Stanford Linear Accelerator Center, Stanford University, Stanford, California (1969).

9. Reference Manual on the Disk I/O Language of Extended ALGOL for the Burroughs B5500, Systems and Program Development, Burroughs Corporation, Pasadena, California (1964).

10. F. J. Corbató and V. A. Vyssotsky, "Introduction and Overview of the Multics System," Proc. FJCC, AFIPS 27, 185-196 (1965).

11. W. F. Miller and A. C. Shaw, "A Picture Calculus," Proc. Conf. on Emerging Concepts in Computer Graphics, (Benjamin Press, New York, N. Y. 1968)(SLAC-PUB-358).

12. W. F. Miller and A. C. Shaw, "Linguistic Methods in Picture Processing—A Survey," Fall Joint Computer Conference (1968) 279. (SLAC-PUB-429.)

13. E. W. Dijkstra, "Cooperating Sequential Process," Lecture Notes, Mathematics Department Technological University, Eindhaven, Holland, (Sept. 1965).

14. Edsger W. Dijkstra, "The Structure of the Multiprogramming System," C. A. C. M. 11, (May 1968) 341.

15. J. Dennis and E. Van Horn, "Programming Semantics for Multiprogrammed Computations," C. A. C. M. 9, (March 1966) 143.

16. R. C. Daley and J. B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS," C. A. C. M. 11, (May 1968) 306.

17. R. Karp and A. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queucing," J. Appl. Math SIAM 14 1390 (Nov. 1966).

18. Duane A. Adams, "A Computation Model with Data Flow Sequencing," Report No. TN-CS-117, Computer Science Department, Stanford University, Stanford, California (Dec. 1968).

19. D. M. Dahm, F. H. Gerbstadt, and M. M. Pacelli, "A System Organization for Resource Allocation," C. A. C. M. 10, 772 (Dec. 1967).

20. R. M. Brown, M. A. Fisherkeller, A. E. Gromme, and J. V. Levy, "The SLAC High-Energy Spectrometer Data Acquisition and Analysis System," Proc. IEEE 54, 1730 (Dec. 1966) (SLAC-PUB-205).

21. R. D. Russell, "MIDAS: A Multilevel Interactive Data Acquisition System," IEEE 15th Nuclear Science Symposium, Montreal, Canada, (Oct. 23-25, 1968) (SLAC-PUB-511).

22. M. J. Flynn and M. D. MacLaren, "Microprogramming Revisited," Proc. ACM Nat'l. Meeting, (1967) 457.

TABLE I

SLAC SMALL COMPUTER INVESTMENTS

Computer	SDS 925	SDS 9300	SDS 930	ASI 6020	PDP-8	PDP-9	IBM 1800	PDP-9
Installation Date	11/65	7/65	11/67	10/66	2/67	9/67	10/66	11/68
First Use Date	4/66	9/66	---	3/67	4/69 est.	1/69	4/67	4/69 est.
Application	Beam Switch-yard	Spectrom-eter	Spectrom-eter	Measuring Machines	Counter Exp.	Spiral Reader	Wire, Spark Chamber	Accelerator Control
Hardware Cost*	$100K	$500K	$150K	$130K	$30K	$50K	$300K	$90K
Software Effort (Man-Months)								
General**	5	100	12	19	5	23	21	3
Applications†	16	40	0	12	6	--	26	3
Continuing Rate (permonth)	1	2	1	1	1	1	1	1.5
Dependence on Other Machines	9300 for assem-blies	none	9300	1401 for print/ punch	none	360 for assem-blies	none	none

* To nearest $10K

** This includes testing and learning the supplier's software, modifications and additions to the supplier's software, local developments on the control programs, general graphics packages, etc. -- these are programs that will be used by many or all the experiments.

† This includes the analysis and data manipulation routines related to specific experiments.

TASKS FOR META-SYSTEMS

(1) Text Editors and Console Languages
(2) Numerical and Algebraic Languages
(3) List Processing and Symbol Manipulation Languages
(4) Graphic Systems
(5) Operating Systems

FIG. 1--Order of increasing complexity of tasks for meta-systems

META-COMPILERS

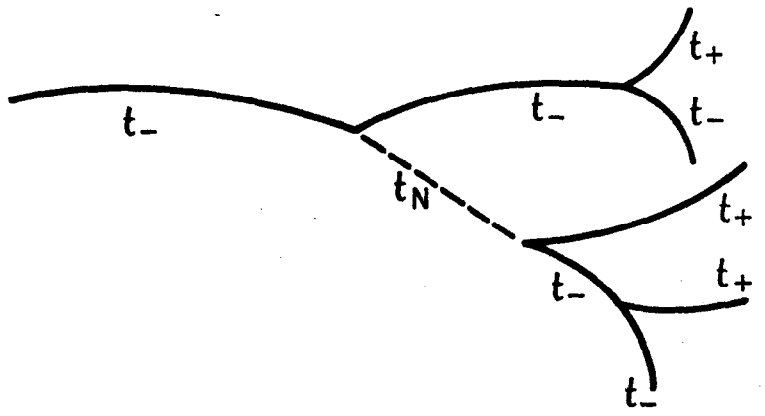
NAME	YEAR	AUTHOR	AFFILIATION
(1) TMG	1965	McClure	Texas Instruments
(2) META Series	1964	Shorre et al.	UCLA
(3) COGENT	1965	Reynolds	ANL
(4) FSL	1966	Feldman	Carnegie-Mellon University
(5) TGS	1965	Cheatham et al.	Computer Associates
(6) EXTENDED ALGOL	1966	---	Burroughs
(7) EXTENDED FORTRAN	1965	---	LRL
(8) XPL	1968	McKeeman[3] et al.	Stanford
(9) PL/I Many Versions	1966-present	---	MIT IBM Others

FIG 2--Compiler writing systems

META'S FOR GRAPHICS AND CONTROL

(1)	AED	1964	Ross[4]	Graphics	MIT
	VITAL	1967	Mondschein[5]	Graphics	Lincoln Lab. MIT
(2)	GULP	1968	Pankhurst[6]	Graphics	Cambridge Univ.
(3)	PL/I	1968	George[7]	Graphics	Stanford
(4)	GLAF	Current	George[8]	Graphics	Stanford
(5)	EXTENDED ALGOL	1966	---[9]	B5500 Operating System	Burroughs
(6)	MIT PL/I	1968	Project MAC[10]	Multics	MIT

FIG. 3--Meta-systems for graphics and operating systems



$$P \longrightarrow \left(t_- + \left(\left(t_- + \left(t_+ \times t_-\right)\right) \times \left(t_N + \left(t_+ \times \left(t_- + \left(t_+ \times t_-\right)\right)\right)\right)\right)\right)$$

867A4

FIG. 4-- PARTICLE PHYSICS PICTURE

Control Language
   a. Function Part
   b. Construction Part

Definition Language

Graphic Description Language
   a. Primitive Representation
   b. Syntax
   c. Semantics

Basic Display File

GLAF Components

Fig. 5

<GLAF-language> ::= *GLAF* <data> ; <syntax> <semantic-list>
<definition language> <control language> *END-GLAF*

<semantic list> ::= <semantics> | <semantic list> <semantics>

GLAF Syntax

Fig. 6

```
<primitive> ::= <name> <sep> <GDL-exp> <sep>
                <attr-list> <sep> <BDF-file>

<attr-list> ::= <empty> | <name> <connector> <value> |
                <attr-list> <name> <connector> <value>
```

Primitive Representation

Fig. 7

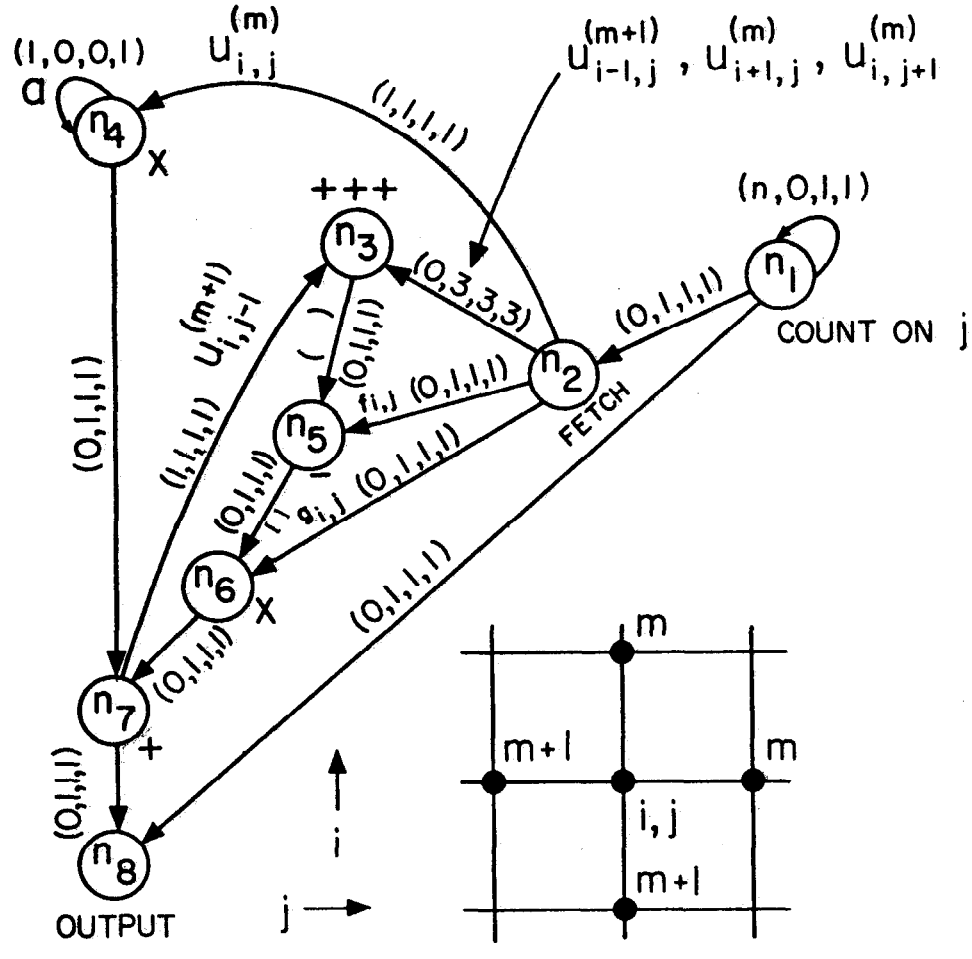
```
<BDF-file>  ::= <empty> | <name> <parm-list> |
                <BDF-file> <csep> <name> <parm-list>

<parm-list> ::= <empty> | <value> | <parm-list> <value>
```

Basic Display File

Fig. 8

$$\nabla^2 u(x,y) = G^2(x,y)\, u(x,y) = f(x,y)$$



$$u^{(m+1)}_{i,j} = a\,u^{(m)}_{i,j} + g_{i,j}\left[f_{i,j} - \left(u^{(m+1)}_{i-1,j} + u^{(m)}_{i+1,j} + u^{(m)}_{i,j+1} + u^{(m+1)}_{i,j-1}\right)\right]$$

126081

FIG. 9--Computation graph for partial differential equation.