Using a programming discipline called the Data Stream Linkage Mechanism (DSLM), a program can be built by linking program modules to form a network through which data passes. The network is specified by the program designer using a mixture of precoded and custom coded modules. This linkage technique and the capabilities that result from it constitute an approach to programming that is radically different from conventional techniques. It can increase the productivity of programmers and can result in programs that are easier to understand and to maintain.

This paper gives examples based on a specific implementation of DSLM and describes some of the experience gained from the implementation over the last six years.

Data Stream Linkage Mechanism

by J. P. Morrison

Symptoms of a problem in conventional application development have been evident for a long time. Almost every programmer has experienced cost and schedule overruns, long debugging times, and difficult and costly maintenance of programs. Solutions that have been tried include various clerical and management disciplines and a number of novel programming languages and programming techniques, with varying degrees of success.

It has been proposed that the problem could be addressed by applying engineering disciplines to application development.[1, 2] A key concept referred to as *design modularity* was identified by R. B. Miller of IBM as early as 1966.[3] It provided principles on which systems could be designed in terms of modules with well-defined functions. The intent was to allow easier construction of new systems and modification of existing ones.

While there are doubtless many reasons for our inability to achieve this modularity in programming, a major factor identified by a number of writers is the control-flow orientation of conventional programming.[4, 5] This orientation is related to the fact that

conventional computers are of the von Neumann type, with a single instruction counter and a uniform array of memory cells. This structure is reproduced in the languages that have been developed to simplify the programming of these computers. While this structure is well adapted for mathematical computation, it is less suitable for the non-numeric applications that are the concern of the majority of programmers.

The major concern of the programmer in this environment is determining the exact sequence of the various atomic operations that make up the application (mostly arithmetic and data-moving operations and decisions). What the programmer *should* be concentrating on is the flow of data through functions that correspond more closely to familiar "real world" functions. He would like to be able to easily convert those functions into working programs. As long as data flow is only a design tool, there inevitably will be a chasm between the program design and the actual program. The chasm, which must be bridged at great cost by the programmer, gets progressively wider as the program is modified to meet changing requirements.

In 1971, the author described the essential features of DSLM in an IBM Technical Disclosure Bulletin.[6] The concepts were at that time embodied in a prototype known as the Advanced Modular Processing System, or AMPS, on which the experience described in this paper is based.

Perhaps because AMPS was developed independently from other work in the field, its terminology is often different from terms used elsewhere in the literature. A glossary appended to this paper gives informal definitions of key DSLM (AMPS) terms.

During the last few years, the author has been attempting to relate DSLM to other work described in the literature, and to determine its similarities to, and differences from, this other work. Clearly one of the seminal papers in this area is the discussion by Conway[7] of programs that run in an interleaved mode, the relationship between the programs being cooperative rather than hierarchic. Conway called such programs *coroutines* (*co-* meaning *with*) as opposed to *subroutines* (*sub-* meaning *under*).

In 1967, Morenoff and McLean[8] described programs (essentially particular types of coroutines) that communicate by means of a one-way flow of data through buffers referred to as *buffer files*.

Somewhat later, Balzer[9] described a system called PORTS, in which an attachment point, or *port*, on one program could be linked to a port on another in such a way that when one program sends data to its output port, the data becomes accessible to the other program.

Weinberg[10] describes a port as ". . . a special place on the boundary through which input and output flow. . . . Only within the location of the port can the dangerous processes of input and output take place, and by so localizing these processes, special mechanisms may be brought to bear on the special problems of input and output."

In DSLM, two or more communicating ports are connected by a buffer called a *queue*. These connections are defined in a *network definition*, a diagram that uses a fairly standard notation which can easily be converted into a series of macroinstruction statements, one statement per diagram block.

In none of these systems does a process need information about the identity of its successors or predecessors in the network. Processes, or *modules*, as they are called in DSLM, are thus completely portable. They can be connected into a network anywhere, provided that modules sending data to them can provide the right kind of input, and that modules to which they send data can handle their output. A module can be used for many different applications, with no need for internal code changes.

configurable modularity

DSLM, PORTS, and the system described by Morenoff and McLean all achieve portability in different ways, but common to all of them is an underlying principle which Edwards[1,11] refers to as *configurable modularity*. Edwards describes the characteristics of systems with this property and shows how it allows engineering disciplines to be applied to program development.

Interest in this kind of system has grown in recent years to the point where a number of papers[12,13,14] presented at IFIP Congress '77 in Toronto addressed this general area. Most of these papers dealt with communicating coroutines as an architecture for improving the reliability of system software. DSLM is unusual in that its main orientation is toward improving programmer productivity. This orientation is shared by Boukens and Deckers' CHIEF,[15] which is remarkably similar to DSLM in its architecture, and MORAL, which is described by Jackson.[13]

An essential difference between DSLM's data concept and that used in most other studies is that DSLM uses objects known as *data entities*, which correspond to *messages* in some other systems. The data they carry is formatted, however, so they more closely resemble the file records in conventional systems, except that they are not simply areas into which data is read, as in conventional programs, but actively travel through the network, initiating processing. They are discussed in more detail under *Basic concepts*, below.

This concept, while unusual in programming methodologies, is common in discrete simulation systems such as GPSS,[16] in which a data entity is referred to as a *transaction*. DSLM may therefore be thought of as a fusion of discrete simulation concepts with a program development methodology. It is not surprising that DSLM has proved to be an effective simulation tool.

The data entities passing across a particular queue constitute what is often referred to as a *stream*, a sequential file of data continuously produced by one coroutine and consumed by another. Burge[17] discusses streams in the context of a functional notation related to LISP,[18] and shows how the stream concept allows a program to be designed as if it were a multipass program, with the simplification of logic that this design provides. The passes are interleaved, however, since they are coroutines.

This paper describes two examples (implementation of which is discussed under *Implementing an application program*, below), which are drawn from a paper by Petersen on data state design (DSD)[19] and a paper by Leavenworth on the Business Definition Language (BDL).[20] BDL originally was defined by Hammer et al.[21]

DSD is a data-oriented system design tool, in which is developed a graphic representation of a system that shows close affinities with a DSLM network. DSD concentrates on the transformations applied to the data in a system, viewing the system as a multipass operation. Each intermediate data file is seen as existing at a single moment. This is indeed a natural way of describing a system. The problems arise in converting a DSD design to a conventional programming language. Without software for handling streams, either an extremely inefficient design will result, or the program will have no structural relationship to the DSD design.

The DSLM approach seems to offer a way out of this dilemma, since the stream concept allows a programmer to develop a running system from a DSD design or network without a drastic change of viewpoint. Although the stream concept in DSLM can be regarded as just one of a number of synchronization techniques that can be applied to the DSD data state dependency network, in practice the program designer starts thinking in terms of the stream concept very early on, and for many simple applications he probably will bypass the DSD design phase entirely.

Leavenworth[20] describes a high-level nonprocedural language (BDL) which is suitable for describing business applications. A goal of this approach is to eliminate *arbitrary sequencing*, defined as "any sequencing not dictated by the data dependencies of the application." It is eliminated by ". . . representing an application by a data flow network. By decomposing the application into a set of steps which communicate with one another only across

linking paths, the sequencing is governed strictly by data dependencies, i.e., one step cannot consume data until it has been produced by its predecessor steps." The data flow concept described by Leavenworth is similar to that of DSLM, and preliminary work (not described here) suggests that, in a DSLM environment, BDL with minor modifications can be a good notation for describing modules, and perhaps even for automatic module generation.

Kay[4] describes three stages in the evolution of programming languages:

- Conventional languages with "passive" building blocks (data structures and procedures).
- "Message-activity" systems in which many parallel activities communicate via messages (examples are DSLM and SMALLTALK, the system described by Kay, as well as most of the systems cited in the references in this paper).
- "Observer" languages, just being developed, which constitute a more powerful programming approach than even the message-activity systems.

Kay feels that a programmer's concepts of programming are strongly influenced by the first programming language he encounters. Thus SMALLTALK was developed to introduce message-activity systems to children before they have much exposure to conventional programming languages. According to Kay, children find the system natural and easy to learn because the modular structure of SMALLTALK is analogous to the highly parallel environment of the real world. It is the rigorous sequentiality of conventional programming, he maintains, that is unnatural in the real world.

This observation has been borne out by our experience in the use of AMPS (the DSLM prototype). While almost all users experienced some productivity gains, new programmers tended to adapt to it more readily than more experienced programmers, and they showed more pronounced improvements in productivity. More important, they learned to think in terms of the concept, instead of treating it as just another programming language. The individual who became most proficient at AMPS had been a machine operator and had had only two weeks of formal programming training—one week using assembler language and one week using AMPS. A typical programming job comprised a network that contained 16 precoded modules and two programmer-coded modules. The job took four hours to design, four hours to code and keypunch, and two hours to test, in a conventional key-punch, batch environment. The ease with which programmers with little experience were able to develop nontrivial programs in this environment is an indication of the potential of AMPS for productivity enhancement in a suitable interactive environment.

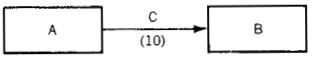
Our success with new programmers points up the fact that DSLM is not a *complex* concept or system. The amount of software required to support it is small, and it is very easy to install (programmers at one location took four days from the time they started working with the system to the time they wrote their first working program, with minimal involvement on the part of the author). DSLM does require a change in the way we look at the programming process, however, and it is perhaps an advantage not to have too many preconceptions in this area.

Basic concepts

modules and queues

Consider two processes, or modules, that communicate by means of a buffer, or queue, over which passes a one-way stream of data entities carrying formatted data. (Data entities are referred to simply as *entities* in what follows.) The two modules run concurrently, one sending and one receiving entities.

Figure 1 Module linkage notation



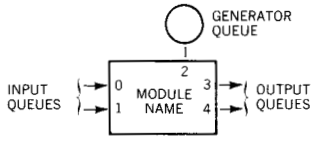
A: sending module
B: receiving module
C: connecting queue capable of holding up to ten data entities of any size

It is necessary that the queue have a finite capacity so that entities will not accumulate in it indefinitely if the receiving module is running slower than the sending module. Thus if the queue fills up, the sending module stops temporarily and becomes *suspended*. The receiving module may also be suspended if the queue is temporarily empty. Data passing through the queue is handled one entity at a time by the sending and receiving modules respectively, forming a stream of data entities. This linkage is represented by the notation shown in Figure 1.

The last entity in a stream is a special *end-of-stream* entity which indicates to the receiving module that no more data follows. From time to time the queue may become empty, but this in itself does not indicate the end of the stream, as the situation may well be temporary.

The capacity of the queue is usually of interest only in tuning an operating program for optimal performance. Therefore it is not usually shown in the diagrams that accompany this paper.

Figure 2 General module notation



The numbers are *port numbers* used by the module to refer to the queues internally

A number of queues can be attached to a module for input or output, or else as *generator queues* (sources of "empty" data entities), which are required for introducing new entities into the system. See Figure 2.

A number of modules can be connected by queues to form a network for the program as a whole. The primary representation of this network is the *network diagram*, or *flow specification*, which can be converted easily to a running program by coding one macroinstruction statement for each diagram block, plus a few additional statements for related information.[22]

Figure 3 Independent subnets within a network

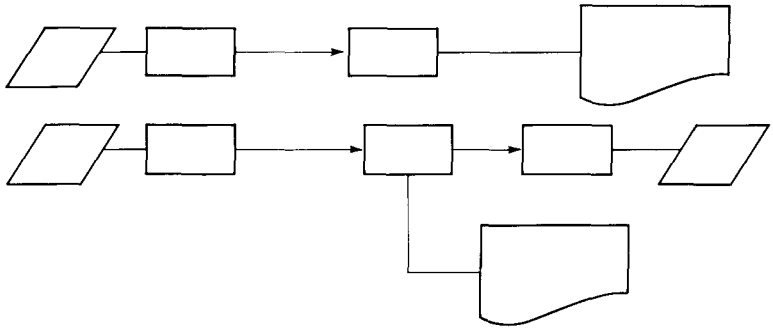
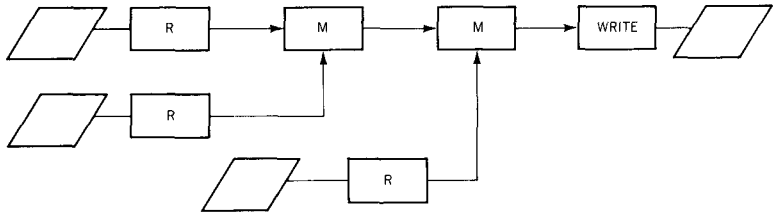


Figure 4 Multiple occurrences of a module



R: SEQUENTIAL READ module
M: MERGE module — combines two streams

The network shown in the flow specification need not be completely connected. Several discrete sections may run asynchronously with each other. For an example, see Figure 3.

Portability of modules is achieved by having modules refer to internal port numbers (for example, *send entity X to port number 2*). It is the network definition that associates the port number with an actual queue. Port numbers are represented in the diagrams by numbers inside the process block, or, for modules with few input or output queues, the position of attachment of the directed lines to the module block.

A module can be attached at several places in the same network, if desired, and will then multithread with itself, provided it is coded in a re-entrant manner. Thus separate processes can use the same piece of code. For example, if several streams must be merged, the program designer may choose to use a "two-into-one" merge module to progressively merge streams until a single stream results. Input and output modules are frequently multithreaded with themselves, but usually will be working with different files. Figure 4 shows the output of a merge module, M, being used as input to another instance of the same module. A generalized READ module occurs three times in the diagram.

Table 1 Comparison of DSLM and unit-record concepts

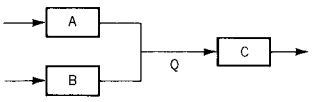
DSLM	*Unit record*
Modules (processes)	Machines
Data entities	Cards, card decks
Network definition	Operator instructions
Scheduler (controlling software)	Operator

In DSLM, one queue is allowed to feed data to only one module, but data can enter the queue from a number of modules, as shown in Figure 5. The entities from modules A and B will arrive at C on a first-come-first-served basis. Both A and B will send end-of-stream entities to Q, however, so in order to prevent C from being terminated prematurely, only the last end-of-stream entity is presented to C by the system software (the *scheduler*).

data entities and streams

A data entity is created (has space allocated for it) by one module, is passed from module to module until it is no longer needed, and is then destroyed (its space is returned to a pool of available space). As stated above, the entity is thus analogous to a transaction in GPSS[16] and to a message in a message-oriented system. The entities can be thought of as items to be worked on, and the modules as work stations, in a data processing "factory." Punched card accounting systems exemplify this environment. The work stations are the accounting machines—sorters, tabulators, calculators—between which flow punched cards. In fact there is a remarkably close parallel between an accounting machine application and a DSLM program, as can be seen in Table 1.

Figure 5 Multiple modules that feed one queue



A, B, C: arbitrary modules
Q: queue that connects modules

Just as most factories process different sizes or kinds of items, DSLM entities may be of different types, or classes. Classes can be mixed freely in any stream, and in general there is no direct correlation between a given queue and the classes of the entities that pass through it.

The streams of entities that pass through a network are themselves objects of interest to the programmer. A DSLM network can thus be thought of as a system of streams which are constantly being expanded, contracted, merged, sorted, split, or transformed.

During the design process, the programmer will switch among the following viewpoints:

- The overall data flow through the system.
- The viewpoint of one module as it handles a series of data entities, one at a time.

- The viewpoint of an entity as it passes from one module to another, starting with its entry into the system and ending with its exit from the system.

These viewpoints seem natural to the program designer, paralleling habits of thinking that are taken for granted in the world of material objects, as in factories, cafeterias, and supermarkets.

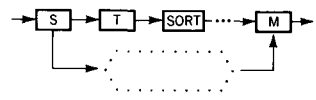
At any given moment, an entity is either owned by a module or queued between modules. It can be owned by only one module at a time, and all entities owned by a module must be positively disposed of before the module returns to its caller (the scheduler). A module can dispose of the entity in various ways, just as a person disposes of a letter—he may destroy it, forward it to someone else, clip it to another letter, or file it. Corresponding DSLM actions are *destroying*, *putting*, and *chaining*. In chaining, an entity is attached to another entity, the resulting structure traveling through the system as one entity. Filing corresponds to sending an entity to an input/output module.

DSLM functions

Many DSLM functions parallel unit-record functions such as COLLATE, SORT, SELECT, and MERGE. These functions are as natural to data processing as multiplication and division are to arithmetic. However, they seldom appear as primitives in high-level languages (with the possible exception of SORT), and their function is usually distributed across the entire program in conventional programming languages.

SORT, for example, is a natural stream operation. The ability to select and transform entities that pass into or out of SORT modules allows the program designer much greater freedom in his design, and also in his record layouts, since SORT tags can be created dynamically and then thrown away (not stored). A section of such a network might look like Figure 6.

Figure 6 A SORT module in a network



S: SELECT *module — determines which entities* are to bypass SORT

T: TRANSFORM module

M: MERGE module — merges sorted and unsorted entities

SORT differs from other stream functions in that no entities can be produced as output until *all* entities have been processed. The entities have to be stored on an external storage device, unless their number is small.

In this implementation, we discovered that since the SORT "control cards" are held in memory, rather than on a file, they can be generated by macroinstructions, using the symbolic names of the fields involved. If the format of the record being processed changes, the SORT control cards can be changed automatically by recompiling.

When more than one module depends on a particular record layout, program modification can be reduced by describing record layouts by means of macroinstructions. Thus only one code com-

ponent (the macroinstruction) has to be changed manually. A *cross-reference* program can then be run periodically to determine which programs use which macroinstructions, and the output of this run can be used to determine which routines should be recompiled when a given macroinstruction is changed.

DSLM provides a convenient tool for working with files of various structures. A READ module can convert data from a format appropriate for tape or disk to a format that is appropriate for internal processing. A matching WRITE module can then recreate the data on tape or disk after any desired changes have been made. Thus each matching READ/WRITE pair can be considered an implementation of a different data organization. DSLM allows the programmer to concentrate on the data structure he wants to work with. It is well adapted for building interfaces between systems and for many programming tasks in which the prime concern is the management of data.

Control flow and data flow

Conventional programming concentrates on the flow of control, rather than the flow of data. A conventional program specifies the exact sequence of actions and decisions to be followed while processing one or more pieces of data. A data-flow approach concentrates on the flow of data through a system and the transformations that apply to them.

The module of the control-flow approach is the subroutine, which is an excellent structure for generalized computation and logic functions, but it does not yield useful generalized functions for data handling. The module of the data-flow approach is the data-linked coroutine, which yields many useful generalized functions for data handling and non-numeric uses. These two module types are complementary, and judicious combining of the two enables the programmer to create highly modular systems.

One way of visualizing the problem with control-flow, subroutine-oriented programming is to realize that the programmer is required to specify the precise timing relationship of every program event to every other. The subroutine technique, while allowing specification of program logic at a higher level, still requires a rigid *do this then do that* structure.

Table 2 Data dependence in code

Program 1	*Program 2*
MOVE A TO B	MOVE A TO B
MOVE B TO C	MOVE C TO D

The timing of events in programming depends almost entirely on the use and availability of data. Consider the two pairs of statements in Table 2. Clearly, the sequence of the two statements in Program 1 is significant for the functioning of the program, but if the second statement is changed to MOVE C TO D, as in Program 2, the sequence of the statements becomes irrelevant because they

share no data. Compiler optimizers devote a lot of logic to determining which statements share data (have timing constraints relative to each other) and which do not. When loops and branches are introduced into the logic of a program, the programmer is faced with the task of finding a sequence of instructions that fits a large number of timing constraints.

The DSLM module structure, on the other hand, is simple in terms of its data-use patterns. It splits a conventional monolithic program into a number of pieces related by two simple and natural data-use constraints:

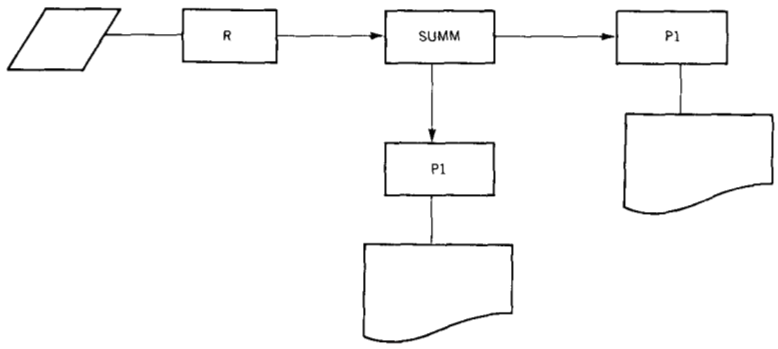
- If a data entity passes from module A to module B, B processes it after A does (the flow constraint).
- If a stream of data passes from A to B, B processes the entities in the stream in the same order in which A sent them (the order-preserving constraint).

There are no other constraints between modules. It is the DSLM scheduler software that determines the sequence of operations that conforms to the above constraints. In a conventional program, on the other hand, the programmer has to find a sequence of operations that conforms to his given constraints and performs the desired function.

A common problem in program design is the difficulty of deciding which subroutine is to call which. Often it is an arbitrary yet highly constraining decision as to which program becomes the driver. Since a subroutine cannot preserve information from one invocation to the next, higher-level subroutines have to set up storage for use by lower-level subroutines, and it becomes impossible to change this relationship later. In DSLM, a module maintains its own internal environment; it controls what it will accept as input and what its output will be, so that the form of the routine is far more independent of its external environment. Of course, a module can call subroutines as in conventional programming, so that the total system consists of many subroutine tree structures communicating via data streams. A conventional program is therefore a special case of a DSLM network—one with only one module.

Programming productivity is enhanced with DSLM by the ease with which modules can be linked. DSLM enables programming to become a process of assembly in which the programmer assembles a program mostly out of precoded modules, using some new modules when required. Trial modules can be constructed and evaluated for ease of use and performance. New modules can be more or less general, depending on economic factors such as potential use compared to development cost. The more general modules become part of a "mental tool kit" which programmers

Figure 7 Testing the module SUMM



R: READ module — generates test data
SUMM: the module to be tested
P1: simple PRINT module

and program designers can use to speed design and development. It should be stressed that there is no "perfect" module—only modules that have been built by programmers with an ability to generalize, and that have then proved useful. Other modules may not have gained wide acceptance and will be used infrequently.

With DSLM, a programmer's knowledge and experience can be preserved and disseminated more widely because it can be embodied in a self-contained module which others can use in complete ignorance of its internal structure. Examples might be modules designed to handle special hardware, interpreters of specialized languages, and modules that use special system facilities. The programmer learns to think in terms of the available modules. He has a reference manual to help him with details of parameters, queue numbers, etc., but it helps if a module's function can be expressed in a few sentences. The more complex a module is, the less portable it is. Our experience has been that some of the most useful modules are also the simplest: one heavily used module consisted of only a dozen statements (assembler and macroinstructions).

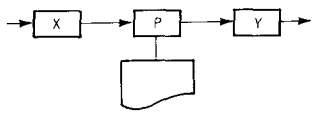
Testing

Testing is facilitated by the fact that the modules are pretested, and also by the ease with which they can be assembled into working programs. A given module is designed to receive a stream of entities of a certain form, regardless of how the entities were generated. It can therefore be tested with manufactured information read by a simple reader or test-data generator. Output can be handled by a simple PRINT module, or even by a module that simply dumps each entity.

As an illustration of this approach, suppose a programmer has coded a summation module with one input queue and two output queues. The simplest way to test it probably would be to set up a "scaffolding" network as shown in Figure 7. The input files can be on cards or, in an interactive test environment, they can be edited data sets. P1 is a deliberately simple PRINT module which displays each entity sent to it, without any editing. P1 is used when it is desirable to see entities unchanged. Alternatively, a DUMP module could be used to dump each entity in hexadecimal and character formats.

Conversely, if a programmer wants to display the entities that pass between any two modules, all he has to do is insert a PRINT or DUMP module between them, as shown in Figure 8.

Figure 8 Displaying the flow between two modules



X, Y: arbitrary modules
P: the inserted PRINT or DUMP module

Multithreading

A DSLM network naturally multithreads with itself, each module constituting a thread. Along with DSLM's implications for improving programmer productivity, multithreading can improve performance when peripheral devices are involved, since such devices usually run more slowly than the central processing unit (CPU). Note that, since modules are re-entrant, any number of modules in a network can use the same code.

In a paper on AMPS, Ballow[23] describes a heavily input/output-bound job whose elapsed time was reduced significantly (from two hours to 18 minutes) by changing from a serial network to a parallel one, replacing a READ module by a faster (but still fairly general) one, and adding a module to balance the loading.

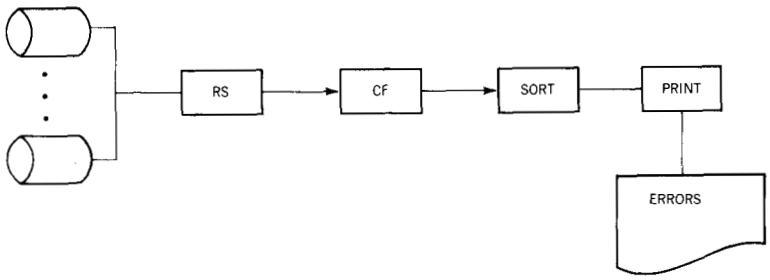
The solution chosen in this example illustrates two principles for reducing elapsed time by multithreading:

- Assign a module to control an independent device.
- Replicate modules where unduly long sequential processes occur.

For the first principle, suppose that a program needs to read a number of disk data sets residing on separate disk packs, and that the records are not required in any particular order by the next module in the network. There are two ways of handling this: by defining one READ module to read concatenated data sets, or by specifying several READ modules, each reading one data set and feeding entities into the same queue asynchronously. The output of the READ modules will be received in random sequence, but in this case this is quite acceptable.

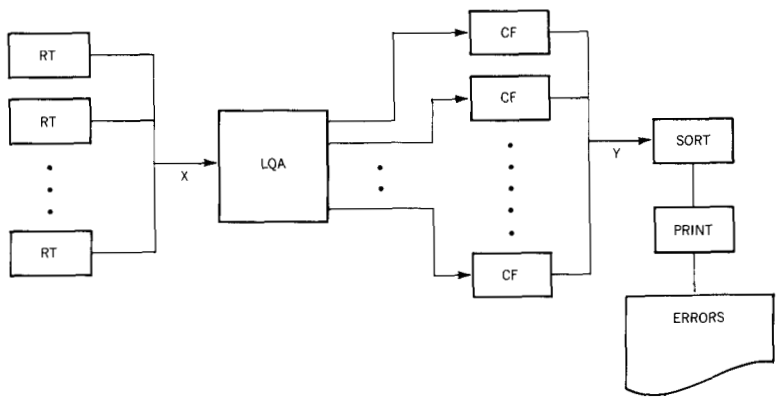
In general, it is often convenient to have one module control a single serially reusable resource, and local optimization can often

Figure 9 Single-threading version of scanning program



RS: SEQUENTIAL READ module — reads concatenated disk data sets
CF: chain-following module — follows chains of records across several disk packs, verifying validity of the chains
ERRORS: report of discrepancies sorted by record identification

Figure 10 Multithreading version of scanning program



RT: full-track READ module (five occurrences, one per disk pack)
LQA: load-balancing module
CF: chain-following module — follows chains of records across several disk packs, verifying validity of chains
X, Y: queues fed by RT and CF, respectively
ERRORS: report of discrepancies sorted by record identification

be applied to get further improvements because the module is in complete control of the state of the resource.

The second principle can be applied where a module performs a sequential process that takes a long elapsed time relative to the amount of CPU time used. The elapsed time required for the process often cannot be reduced, but system throughput may be increased by replicating the process. Of course the above requirement for logical independence of the threads still holds.

In the application described, the requirement for logical independence of processes was satisfied because the purpose of the program was to scan a data base and prepare a report on any discrepancies found in the record chaining, which was needed in a sequence different from that in which the data was stored. This freed the scan from the requirement of keeping discrepancy information in master record sequence.

Figure 9 shows the program as it was first coded. Figure 10 shows it after the improved READ module (full-track) had been written and the network converted to a highly parallel structure.[24] The chain-following module (CF) was characterized by long elapsed time and low CPU time, so in the redesigned network it was multithreaded with itself 18 times (large enough to be effective, but not so large that contention would start to be significant). To balance the load, one additional module (LQA) was written which allocated work to the downstream module with the smallest backlog of work.

As the demand grows for real-time response on the part of data processing systems, it becomes more and more important in many environments to minimize total elapsed time, as compared with CPU time, and this is much easier to control using DSLM. An alternative approach to reducing elapsed time is to distribute function among different machines. This approach leads naturally to hardware architectures that parallel DSLM's software architecture.

Practical details

It is outside the scope of this paper to describe in detail the workings of the AMPS prototype, but some implementation information is given here to aid in visualizing the working of a program constructed using AMPS.

All modules and routines in an AMPS environment are made reentrant, as is the scheduler software, by avoiding the use of self-modifying code and by ensuring that all storage that can be modified by a module is unique to that process (occurrence of the module in the network).

All routines have a single entry point and a single exit (although a module can branch to the exit from any point in the code), for which standard *entry* and *exit* macroinstructions are used. The *entry* macroinstruction causes storage to be allocated, for that invocation of the routine, for a *register save area* and for a "scratchpad" (used for temporary results of calculations and the like). The macroinstruction determines how much storage space is required and allocates it from a *module control block* that is unique to that process. When the routine terminates (returns to its caller), it uses the *exit* macroinstruction to make that storage space available for use by other routines. Since no routine can terminate after its invoking routine terminates (within a given process), the space in the module control block can be used as a *stack* ("pushed down" when a routine is invoked, and "popped up" when it terminates).

Other AMPS services, such as GET, PUT, and CREATE, are also requested by means of macroinstructions.

Although a module may have multiple input queues, only one is allowed to trigger module execution. This is the queue attached to port number 0 of the module and is called the *triggering queue*. Data entities that arrive along other input queues (if any) are obtained by means of a GET request.

At the beginning of a job step, an AMPS module is in what is referred to as the *dormant* state. When an entity arrives along the triggering queue, the module is invoked by the scheduler and is passed the addresses of the incoming entity and of a *parameter block*—a storage area included in the network description where the parameters for that particular use of the module are specified.

When the module has finished processing the current entity, it may terminate, returning control to the scheduler, in which case it again becomes dormant. Or it may not terminate, but rather issue a GET request for another entity from the triggering queue (processing may be suspended if no entity has arrived yet).

The last entity in any stream is always an end-of-stream entity, which signals the receiving module that no more data should be expected. The scheduler will not re-invoke a module when it terminates after *end of stream* is presented at its triggering queue (port number 0), so the module must send the end-of-stream entity on to all downstream modules prior to terminating. The module is then effectively removed from the network and no longer takes part in the scheduling process.

When all modules have closed down in this way, the scheduler determines whether all queues are empty, and, if so, terminates the job step. If not, an abnormal termination occurs.

Absence of a triggering queue indicates that the module is to be started at the beginning of the job step. This is the way READ modules are normally started in batch jobs, but it also provides a way to defer a READ module for a time: a module with a triggering queue specified cannot start until the first entity arrives along this queue, so a triggering queue is specified for the READ module, and a *signal* entity is sent along this queue to indicate that the module is to start execution.

In the DSLM prototype (AMPS), only the entity's address is moved as the entity passes through the network, so it is quite reasonable to conceive of large tables passing through the network. Only the owning module normally will be able to address the entity, so no other module can modify that entity at that time. In fact, the total amount of storage that a module or program can affect is quite

limited. Since all programs are re-entrant, they can never modify themselves. The only storage a program is allowed to modify is its scratchpad and those entities that it currently owns (has responsibility for).

There are two kinds of entity in an AMPS system: *dynamic* and *static*. Dynamic entities are the normal entities described above. They are used for transporting dynamic data through the system. Since dynamic entities are not necessarily freed in the sequence in which they were created, a facility is required for storage allocation and de-allocation. In AMPS, all entities of a given type, or class, are the same length, so the first time a routine requests the scheduler to allocate an entity of a given class, a subpool of some (user-specified) number of entities of that class is allocated with the available entities chained together. Thereafter, when a routine requires an entity, it is taken from the head of the chain. When an entity is freed (destroyed), it is added to the head of the chain. If all entities in the subpool are in use, additional CREATE requests are satisfied by using the GETMAIN macroinstruction.

Static entities, on the other hand, are unmodifiable, so it does not matter what program owns them. In fact, since only addresses physically move through the network, a static entity can be treated as though it were in many places at the same time. For the same reason, static entities do not have to be positively disposed of, as do dynamic entities. Examples of static entities are the end-of-stream and signal entities referred to above.

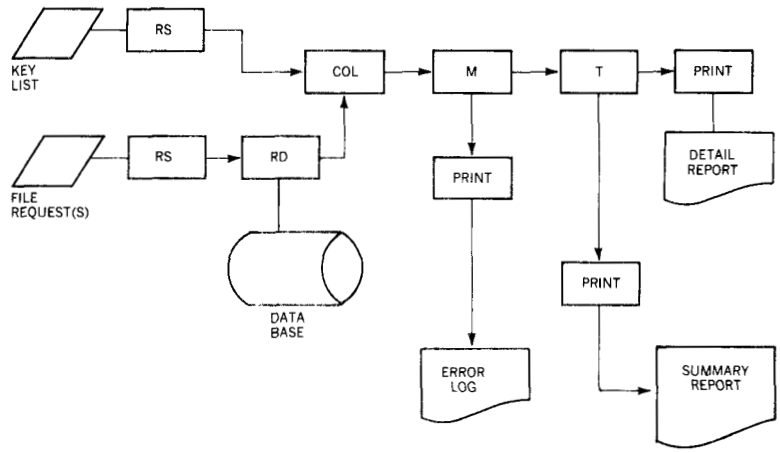
AMPS provides two basic language levels—the network definition language and the language used to construct new modules. Assembler language with many macroinstructions is the basis for both of these languages, but the macroinstruction families involved can be thought of as forming special-purpose languages. For example, the network definition is normally all macroinstructions, plus DEFINE CONSTANT (DC) statements to define parameters, but assembler language can, if necessary, be intermixed with the macroinstructions.

An experimental version of AMPS has been developed that allows modules to be coded in PL/I and that uses OS/VS2 Release 2 multitasking to provide the multithreading capability, but it is still too new for any report on experience in its use. The following section, therefore, is based only on the existing version of AMPS, the DSLM prototype.

Implementing an application program

Discussed below are two example application programs as they would be implemented using AMPS, including the module library that is presently available. The examples are key matching and

Figure 11 Network for key-matching example



RS: SEQUENTIAL READ module

RD: DATA-BASE READ module — accepts a file request and puts out a stream of data-base records

COL: generalized COLLATE module — merges two or more streams on basis of specified control fields; inserts *break* entities between entities with different control-field values

M: module that sends key-matching *indications* out at one port and nonmatching *entities* out at the other (matching entities can simply be destroyed at this point)

T: module that accepts key-matching indications and sends them out at one port, and sends summary lines out at the other

sales statistics, both based on applications discussed in recent literature.[19, 20] Note that both are batch programs, reflecting the fact that all of our experience so far with AMPS has been in batch processing; hardly any work has been done on the implications of the DSLM concept for real-time application development. However, the internal architectures of many real-time systems bear a strong resemblance to the DSLM architecture, suggesting that DSLM will prove to be applicable in a real-time environment.

key matching

Petersen[19] illustrates a DSD data state dependency network which performs the following function: On receipt of a *file request* (the file name) it obtains the specified file from the data base, passes it against a list of keys, and produces detail and summary reports for matching keys, as well as an error log for mismatches. The DSLM notation for this network is shown in Figure 11. A KEY+FILE module in the DSD data state dependency network is replaced in Figure 11 by two modules, COL and M, because in the AMPS library there is a generalized COLLATE module (COL) which has proved useful for applications that require one file to be passed against another. It merges two streams into one, and also inserts *break* entities at control breaks, simplifying the logic of downstream modules. An AMPS user would be aware of this function and use it for most such applications. The function of the KEY+FILE node in Petersen's example, therefore, is performed in DSLM by having COL send data to another more specialized module.

The network definition is first laid out graphically on paper, with comments added freely to show suggested SORT parameters, DD (data definition statement) names, stream descriptions, etc. These stream descriptions correspond to the FILE nodes in a DSD data state dependency network. Such an annotated network is the main working document and is a good communication vehicle, while containing sufficient detail so that it can easily be converted to a running program.

The next step is to consider which data structures to use and to determine which standard modules can be used from the AMPS library. For each module in the library there is a brief functional description, together with parameters and any other external interface information. The program design logic might be as follows:

- There is a standard READ module (R1) which takes a sequential blocked or unblocked file from tape, disk, or cards and puts out fixed-length record entities. The designer decides this module is appropriate for both the *key list* and *file request* files.
- The key list, therefore, is a sequential, fixed-length record file, which R1 will convert into a stream of entities followed by *end of stream*.
- If the same module is used for the *file request* file, the R1 module for file requests will read a single record and generate one entity followed by *end of stream*.

The designer can proceed in this way across the network, deciding which standard modules and corresponding data structures to use. At this point he will start to annotate the diagram, assigning names to modules and queues (any mnemonic will do), DD names to input and output modules, and report titles to PRINT modules.

Assuming that no suitable standard modules exist for RD, M, and T, the next stage is to design, code, and test these modules. RD can use any OS access method down to the EXCP level, but will most likely use a *basic* access method (BSAM, BDAM, BPAM), since using one of these (or EXCP) allows other modules to continue execution while RD is waiting for completion of an input request. AMPS provides a *module wait* facility which will suspend only the module requesting the wait.

Module M is essentially a pattern-matching function which repeatedly looks for this sequence of entities:

key (from the key list); *record* (from RD); *break*.

Any other sequence causes an error.

While the programmer charged with designing and coding M may make it specific to this application, he may be able to generalize

the function so that its applicability is broader than the specific need. Thus he may be able to reduce the cost of the next application that requires a similar function. Alternatively, its function can be broadened later, provided that the parameters of the original function were designed for possible future expansion. The same is true for the module T.

The last stage is coding and testing the network. All new modules can be tested initially in parallel, and, as a new module is debugged, it can be used in testing other modules. Thus several programmers may be working on different parts of the same system in parallel, creating special networks of "scaffolding" (as described above). Gradually they will start to need each other's modules, and the dependencies between them will start to increase, but at the same time the reliability of the modules will be increasing. Modules can easily be integrated into larger and larger networks until eventually the network is in its final running form.

It is recommended that during testing only one unknown be introduced at a time. For this reason, during much of the testing, printing will be done by a simple PRINT module (P1), which puts out one entity per line with minimal modification. This allows the programmers to inspect the entities put out by a module in as close to their original form as possible.

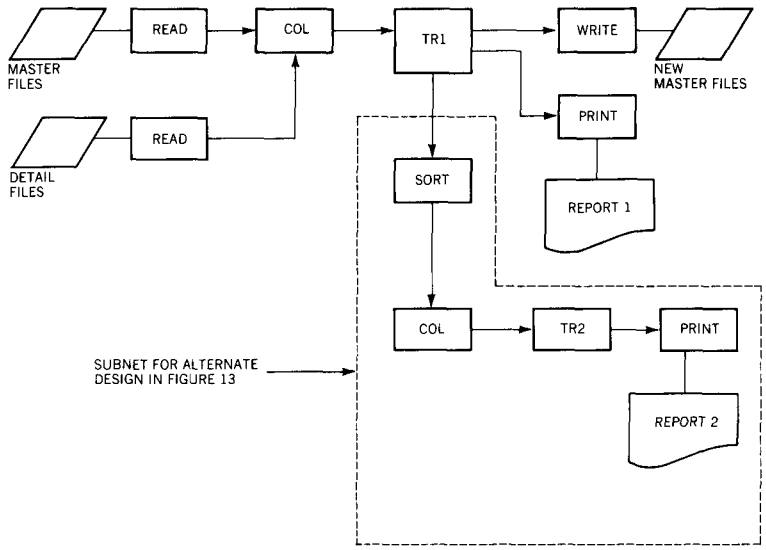
In the final form of the program, more sophisticated PRINT modules will normally be used. One such module (P2) performs a number of formatting and output control functions, but, since it displays the data being sent to it in a different form from the way it received it, the programmer will want to add these modules to the network after much of the other testing is complete. However, this is simply an incremental effort, involving no change to the rest of the network, or to upstream modules.

sales statistics

The second example is based on the example used by Leavenworth[20] in his paper on BDL (the Business Definition Language), in which he describes a set of programs that generate sales statistics reports. This application is used to update a master file of products on a regular basis given a sorted detail file of product sales, and to produce two reports: a summary by product and a summary by district and salesman.

While the first report is in product sequence, as is the master file, the second is ordered by salesman within district, independently of the product. Explicit sequencing is expressed in BDL by the WITH COMMON operator, and although this is more general than specifying a SORT, the AMPS program designer will already be deciding if he wishes to use a SORT or some other mechanism, and SORT is in fact a natural data stream operator (see under *Application independent modules* in Reference 1).

Figure 12 Network for sales-statistics example



COL: generalized COLLATE module — merges two or more streams on basis of specified control fields; inserts *break* entities between entities with different control-field values (if used with only one stream, this module simply inserts *break* entities, as in the second occurrence of COL)

TR1: module that corresponds to Leavenworth's Tran-1;[20] it accepts a merged stream of *product masters* and *details* and generates a stream of *new masters*, a stream of *product summary* lines, and a stream of *extended details* (*details* with *quantity* x *unit price* calculated and inserted into the entity)

SORT: *generalized module that sorts extended details from TR1 by salesman within district*

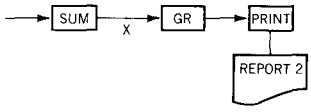
TR2: module that corresponds to Leavenworth's Tran-2; it accepts a sorted stream of *extended details* and puts out report lines for REPORT-2

Assume that SORT is chosen to do the resequencing desired. Remembering that AMPS has a COLLATE module which merges two sorted streams and inserts *break* entities whenever the control field changes, the example in Reference 20 can be represented as shown in Figure 12. The second occurrence of COL is used to insert breaks into the sorted *extended details* stream coming out of SORT, in order to simplify the logic of TR2 in Figure 12.

Figure 12 assumes that the incoming details are sorted by product number. Although this SORT could be included as part of the network, it was not, for purely pragmatic reasons which are outside the scope of this paper.

A valid objection can be raised that sorting is just one way of arranging information into a desired sequence, and that the decision as to the exact technique should not be made too early. The point is that DSLM allows the designer to concentrate on the flow of data and in fact makes the available options more visible and more controllable. For instance, in the above example the designer may indeed decide that, for various reasons, he prefers to construct a table of district and salesman codes and totals, which will be updated randomly as the extended details come out of TR1.

Figure 13 Alternate design for part of Figure 12



SUM: module that updates district and salesman totals in random sequence

GR: GENERATE REPORT module — steps through the totals, formatting report lines

In that case, a separate module will be needed to go through the table afterwards, preparing report lines. This module can be designated GR (GENERATE REPORT). The part of the network within the broken lines can then be replaced by the subnet illustrated in Figure 13. The queue marked X in the diagram could simply pass a signal indicating SUM's final termination to GR, which would then start working on the totals table, to which it has also been given addressability. A cleaner solution, however, is for SUM to send the entire table in an entity to GR immediately prior to termination. This will be GR's only input data entity and will start the GR function going. (Since only the address of the table is actually moved, in DSLM it is possible to conceive of sending large tables from module to module.)

Conclusions

DSLM, as exemplified by the AMPS prototype, has measurably enhanced programmer productivity and program maintainability in everyday application programming by replacing the conventional programming technique with the more natural process of plugging together data-driven functional modules.

Although the original motivation behind the search for something like DSLM was to improve programming productivity, many other advantages, such as improved control of performance, were discovered later as valuable side-effects.

Ballow's summation[23] covers many of the basic concepts and advantages of DSLM. The main points given there can be restated as follows:

- Separately compiled and debugged portable modules can easily be assembled to do a specific job and can be rearranged and replaced as testing proceeds or as maintenance requirements change.
- Subnets in the network can be consolidated by replacing them with fewer (but less generalized) modules. Time-space tradeoffs can be made simply by selecting modules or changing networks.
- Functional modules can be stored in program libraries, to be used whenever required.
- While the programmer can derive many benefits from using precoded and pretested modules, he still needs the capability of coding his own modules if he decides it is best to do so. This is straightforward in DSLM (AMPS) using the macroinstructions provided. A new module is easy to build and test, and once it is finished and working, it can be added to one of the module libraries, where it will be available to anyone who needs it.

d'Agapeyeff[25] eloquently describes the programmer as a "pavement artist," constantly investing effort and creativity in essentially transient constructions. DSLM points to a future programming environment in which programmers, in creating more lasting constructions, may experience the satisfactions of authorship and greater recognition. They can become more productive, and applications can be designed and brought on line faster and more reliably, providing improved service and responsiveness to users.

ACKNOWLEDGMENT

The author wishes to thank the many individuals who have supported this effort with their encouragement, ideas, and hard work, and the editors of the *Systems Journal* and the referees, for their time and many useful comments.

Appendix: Glossary

AMPS—prototype system that embodies the DSLM concepts.

Break—static *entity* (q.v.) used to indicate a control break in the output stream of a COLLATE module.

Capacity—the maximum number of entities a given queue can hold; specified in the *flow specification* (q.v.) if different from the default value.

Chaining—attaching one entity to another so that the resulting structure can be moved through the network as a single entity. Complex structures can be built up in this way.

Class—type of entity.

Coroutine—a routine or program that runs interleaved with, but in constant communication with, one or more other routines, in a cooperative rather than a hierarchic relationship.

Create—allocate space for an entity and initialize a control block that defines it.

Destroy—return an entity control block to the pool of available space.

Dormant—the state of a module that either has not been invoked by the scheduler, or has been invoked but has returned control to the scheduler after having processed one or more entities.

Entity—a carrier of formatted data. There are two main types: dynamic entities, which are modifiable and can be owned by only one module at a time; and static entities, which are read-only entities that may appear to be in several places at the same time (examples are breaks, signal entities, and *end of stream*).

Flow specification—diagram showing the modules that constitute an application, the queues relating the modules to each other, and

other explanatory information; also, the macroinstruction statements that implement the diagram.

Generator queue—a *queue* (q.v.) specified in the flow specification, not as a communication link between modules, but as a source of unused entities of a particular class.

Module—a routine that runs asynchronously with other modules in the flow specification (that is, a coroutine); it can be the root of a tree of subroutines.

Module control block—a control block used by the AMPS scheduler to control the operation of a single process. It is used also for allocating *register save areas* and scratchpads for routines as they are invoked during process execution.

Multithreading—interleaved running of sections of code in a single CPU. The sections of code compete for control of the CPU, and control is switched among them by a piece of software called a *scheduler* (q.v.).

Network description—same as *flow specification*.

Ownership—a module owns a dynamic entity if the entity has arrived on a queue and caused *triggering* (q.v.) or has been obtained with a GET, if it has been unchained from another entity, or if it has been created or obtained from a generator queue. The module owns the entity until it positively disposes of it (puts it, chains it, or destroys it).

Parameter block—a block of read-only data coded with the flow specification, to specify application-dependent parameters for generalized modules.

Port—point of attachment of queue to module, specific to the function of that queue for that module (for example, port number 1 is the output port for the majority of READ modules).

Process—same as *module*.

Queue—a buffer that acts as the communication path between two or more modules; it has a capacity of some number of entities.

Scheduler—software that controls the flow of control between modules and the flow of entities from one module to the next.

Scratchpad—temporary storage allocated to any program or subroutine in the AMPS environment; it can hold temporary results that are not required across more than one invocation.

Stream—a set of entities that pass across a given queue.

Subnet—a section of the flow specification.

Subroutine—a routine invoked by, and subordinate to, another program or subroutine; the invoker is suspended until the subroutine terminates.

Triggering queue—a queue that contains the only entities that can trigger invocation of the module; the queue is attached to port number 0 of the module.

CITED REFERENCES AND NOTES

1. N. P. Edwards, *On the architectural requirements of an engineered system*, Research Report RC 6688, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598 (August 1977). (ITIRC AAA 77A004397.)
2. *Software Engineering*, Infotech State of the Art Report 11, Infotech Information Limited, Maidenhead, England (1972).
3. R. B. Miller, personal communication with the author, 1966.
4. A. C. Kay, "Microelectronics and the personal computer," *Scientific American* **237,** No. 3, 230–244 (September 1977).
5. Arvind and K. P. Gostelow, "A computer capable of exchanging processors for time," *Information Processing 77* (Proceedings of IFIP Congress 77, Toronto, B. Gilchrist, editor), 849–853, North-Holland Publishing Company, New York (August 1977).
6. J. P. Morrison, "Data responsive modular, interleaved task programming system," *IBM Technical Disclosure Bulletin* **13,** No. 8, 2425–2426 (January 1971).
7. M. E. Conway, "Design of a separable transition-diagram compiler," *Communications of the ACM* **6,** No. 7, 396–408 (July 1973).
8. E. Morenoff and J. B. McLean, "Inter-program communications, program string structures and buffer files," *AFIPS Conference Proceedings* **30** (1967 Spring Joint Computer Conference, Atlantic City), 175–183 (April 1967).
9. R. M. Balzer, "PORTS—a method for dynamic interprogram communication and job control," *AFIPS Conference Proceedings* **38** (1971 Spring Joint Computer Conference, Atlantic City), 485–489 (May 1971).
10. G. M. Weinberg, *An Introduction to General Systems Thinking*, John Wiley & Sons, Inc., New York (1975).
11. N. P. Edwards, "The effect of certain modular design principles on testability," *Proceedings, International Conference on Reliable Software* (Los Angeles), 401–410 (April 1975).
12. C. Hewitt and H. Baker, "Laws for communicating parallel processes," *Information Processing 77* (Proceedings of IFIP Congress 77, Toronto, B. Gilchrist, editor), 987–992, North-Holland Publishing Company, New York (August 1977).
13. K. Jackson, "Language design for modular software construction," *Information Processing 77* (Proceedings of IFIP Congress 77, Toronto, B. Gilchrist, editor), 577–581, North-Holland Publishing Company, New York (August 1977).
14. G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," *Information Processing 77* (Proceedings of IFIP Congress 77, Toronto, B. Gilchrist, editor), 993–998, North-Holland Publishing Company, New York (August 1977).
15. J. Boukens and F. Deckers, "CHIEF, an extensible programming system," in *Machine Oriented Higher Level Languages* (Proceedings of the IFIP Working Conference on Machine Oriented Higher Level Languages, Trondheim, 1973, W. L. van der Poel and L. A. Maarssen, editors), North-Holland Publishing Company, Amsterdam (1974).
16. *General Purpose Simulation System V Introductory User's Manual*, IBM Systems Library, order number SH20-0866, IBM Corporation, Technical Publications Department, 1133 Westchester Avenue, White Plains, New York 10604 (August 1971).
17. W. H. Burge, "Stream processing functions," *IBM Journal of Research and Development* **19,** No. 1, 12–25 (January 1975).
18. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP 1.5 Programmer's Manual*, Second Edition, The MIT Press, Cambridge, Massachusetts (1965).

19. J. E. Petersen, "Data state design," *Digest of Papers, Fall Compcon 76* (13th IEEE Computer Society International Conference, Washington), 102–104 (September 1976).
20. B. M. Leavenworth, "Non-procedural data processing," *The Computer Journal* **20,** No. 1, 6–9 (February 1977).
21. M. Hammer, W. G. Howe, V. J. Kruskal, and I. Wladawsky, "A very high level programming language for data processing applications," *Communications of the ACM* **20**, No. 11, 832–840 (November 1977).
22. In this paper, modules are represented by rectangles, and queues by directed lines. As far as possible, a left-to-right and top-to-bottom flow is used. When modules are concerned with input or output, an appropriate symbol usually is shown attached to the module in question by an undirected line to distinguish the connection from a queue. The symbols used for input and output are the standard ones used in flow charts for card decks, reports, tape, and disk, plus the lozenge, which is used in this paper for a sequential file when the medium is not important. A partial diagram, or subnet, is sometimes indicated by directed lines with a module block attached only at one end.
23. R. E. Ballow, "A modular processing system," *Proceedings, GUIDE 41* (Denver), 4–16 (November 1975).
24. The full-track READ originally was coded to start reading at record number 1. It was realized, however, that READ actually could start at any record, eliminating much of the rotational delay, provided that the READ module was enhanced to send disk records to its output queue in the correct order. Once such modules have been created, the program designer can make this kind of decision merely by selecting the appropriate module.
25. A. d'Agapeyeff, "Programming: the unwanted, unloved profession," *Computers and People* **26,** No. 1, 10–11 (January 1977).

Reprint Order No. G321-5081.