A DATA FLOW LANGUAGE FOR OPERATING SYSTEMS PROGRAMMING

Paul R. Kosinski
IBM Research Center
Yorktown Heights,
New York 10598

Abstract

This paper describes a graphical programming language based on the concept of pure data flow sequencing of computations. Programs in this language are constructed through function definition and composition, and are based on the primitive notions of iteration, recursion, conditional expression, data replication, aggregation and selection, and the usual arithmetic and logical operations. Various useful programming devices such as the DO loop and, surprisingly, the memory cell are defined in terms of these primitives. Programs in this language are determinate in operation unless indeterminism is explicitly introduced. The utility of this language for designing and implementing operating systems is discussed.

Problems of Operating Systems Programming

There are eight problems facing the programmer of operating systems which need solution in order to raise the current level of software engineering. First, one must be able to construct systems which are quite parallel in their operation, and yet determinate, or free of undesired timing dependencies. Second, the operating system must be partitionable into relatively independent modules such that each module can only affect its own locality, except for rigorously defined intermodule communication, and thus bugs cannot cause unbounded damage to the whole. Third, the system must be understandable in the large, in detail, and at all other levels of abstraction. Fourth, it must be possible to make use of experience gained while implementing the system in order to improve its design in a clean and consistent manner, and not have the implementation depart from the design in some unknown manner, as often happens today. Fifth, the system must be modifiable without excessive difficulty. Sixth, the system code must be reasonably efficient, which is not trivial to obtain when using higher level languages. Seventh, it must be possible to measure the performance of the operating system in order to determine what improvements are necessary. Eighth and last, it must be possible to predict the performance of a system without implementing it entirely. This could save much embarrassment in the future.

A Programming Language Attack

This paper proposes a programming language DFPL [1], which attacks the first five problems mentioned above, without making the other three problems any worse. We might mention that solutions to these problems would benefit all programmers.

First of all, DFPL is based on data flow instead of control flow. That is, the sequencing of operations is determined by the availability of data for them, rather than by a separate and explicit locus of control. This approach is used to guarantee determinate operation in the presence of parallelism. Such languages and schemas have been investigated by a number of people, particularly by the Computation Structures group at MIT under Dennis [2]. DFPL differs from these languages in several ways. Most significantly, DFPL provides a single indeterminate primitive to deal with the timing dependencies encountered in the physical world.

The GOTO statement has been much criticized as a source of difficulty in conventional programming languages, and other control structures have been suggested to replace it [3]. Recently, the global variable has also come under attack for making programs opaque [4].

DFPL, since it is not based on control flow, does away with the GOTO statement. More surprisingly, it also does away with the assignment statement, and even does away with variables and memory as primitive notions. The purpose of this radical excision is to make modularization more natural, enforce locality of action, and remove the most tempting source of indeterminate communication between processes, the shared variable or memory cell. We will now discuss our rejection of

memory as a primitive notion.

Memory, or explicit storage of variables, seems to be needed in the following two situations. First, a program may be intended to be a filing or inquiry system and communicate with the outside world in a continual and unpredictable manner. In this case, memory is needed in order for the program to perform its function, which is to store and retrieve data for use on the outside at random times. Second, the program may be so complex that the programmer uses memory to simplify his programming job, that is, so he doesn't need to keep in mind all the data at once. A compiler is an example of such a program: a PL/I compiler may be a computable function, but a purely recursive expression of it would likely be quite incomprehensible. Unfortunately, current languages force the programmer to use memory when neither of the above two reasons apply. The PL/I programmer, for example, often must define variables such as "I" and "TEMP" which are used solely within a DO loop to compute the sum of a set of numbers. DFPL obviates the need to do this, since, as we shall see, such variables are replaced by nameless data paths. This is a natural continuation of the trend wherein the programmer has had to worry less and less about the details of his program.

DFPL allows the definition of functions with memory for those programs which need memory to provide their intended services. Similarly, memoried functions may be defined in those cases when memory is necessary for understandability. In any case, modularization applies to both functions and memory at the same time since memory may not exist outside a function.

Since DFPL is based on the mathematical notions of function definition and function composition, it is natural to make use of the Structured Programming style [5]. In fact, one might say that Structured Programming, in the general sense of the term, is no more than building bigger programs out of smaller programs, just like one defines more complex functions by composing simpler ones. Both "top down" and "bottom up" programming techniques may be practiced in DFPL.

DFPL is a single language which may be used to express both the initial design and the final implementation of an operating system. The flowchart-like notation of DFPL makes it appropriate for expressing the gross design of a system in terms of large functional units and their interactions in terms of passing data back and forth. It might be mentioned that this form is often used to sketch out the hardware configuration of a system. Since DFPL is a programming language, however, there need be none of the traditional vagueness and ambiguity that usually characterize such gross designs and sketches. For the same reason, DFPL is perfectly appropriate to program the detailed internals of the system, especially when parallel operation is involved. The established techniques of "top down" programming can be applied to ensure that the resulting implementation is consistent with the design, since the design itself is just a DFPL program at a higher level of abstraction. As the implementation is carried out, any problems which are discovered may have their solution fed back to the design level fairly easily, since the design and implementation are one structured program, and changing the design is just changing one part of the program to be consistent with another part, the implementation.

Since DFPL is based on data flow, and since inter-module communication is strictly via explicit paths, it is relatively easy to connect modules together. Moreover, a given module or subgraph may be replaced by another without worry about whether more or less parallelism results, or whether global references remain consistent. In general, it is easier to follow Parnas' criteria for modularization [6] when programming in DFPL rather than in control flow languages.

As far as optimization, measurability and predictability go, DFPL is competitive with existing higher level languages. Most existing optimization techniques are applicable to DFPL programs [7]. The only area yet unexplored is the efficient compilation of functions whose intercommunication is like coroutines or tasks.

Finally, DFPL makes the structure of an operating system more comprehensible. This should not make it harder to devise instrumentation or make performance predictions.

An Outline of DFPL

A program in DFPL is a directed graph (see Figure 1) where the nodes are functions and the arcs are data paths (or just "paths"). Each function node has the name of the function written in it. Functions may either be primitive or defined as explained below. Data paths carry data from one function to another, hence the term data flow.

The data carried by a data path may be any type desired, from a Boolean to a complex structure, just as in languages such as PL/I or ALGOL 68. Unlike these languages, a data path is not a variable, in that it does not correspond to a memory

cell, but is more like a hardware data bus. Besides carrying data, paths carry two synchronization signals called PRESENCE and DONE. PRESENCE travels in the same direction as the data, and tells when meaningful data is available on the path. DONE travels in the opposite direction to the data, and is used by the receiving function to tell the sender that it no longer needs the data. A path always must go through the sequence: idle, PRESENT, PRESENT and DONE, DONE only, and idle again. That is, data remains PRESENT on a path until DONE appears.

Functions execute asynchronously of one another. A function will execute when all of its appropriate inputs are PRESENT, and after some variable amount of time, it will produce some outputs, making them PRESENT. Simple functions require all their inputs to be PRESENT before they execute, and then present all their outputs at once. More complicated functions may execute when only some inputs are PRESENT, and present not all their outputs. It is even possible that a function require a sequence of several presentations of one input and then make a sequence of presentations of some output. This means that a DFPL function is quite unlike the usual notion of subroutine, but is more like the notion of coroutine. DONE signals behave in a manner complementary to PRESENCE signals. Simple functions send DONE on all their inputs when they when they receive DONE on all their outputs. The sequencing function mentioned above must send DONE after each presentation of its input before it can receive the next. Similarly, it must receive DONE after each presentation of its output before it can send the next. This synchronization rule guarantees determinacy.

Any subgraph of a DFPL program graph may be defined as a function. This is done simply by drawing a boundary around the subgraph and attaching the name of the defined function to the boundary. The data paths which are cut by the boundary are then the parameters of the newly defined function. The subgraph is then replaced by a single node which contains the name of the new function, and the data paths which were cut are attached to the new node to correspond in position to the parameter paths. This new function executes in the same way as the subgraph used to execute, that is, according to the composition of execution rules of the constituent functions of the subgraph. Functions may be defined recursively in the usual way. Recursive functions execute according to a complicated variation of the ALGOL "copy rule": a copy must be made as soon as any input becomes PRESENT.

The fact that any subgraph of a DFPL program may be defined as a function, means that modularization can be done according to any criterion. In particular, a system may be modularized in order to make it most understandable. This is not always possible when one is constrained to subroutines and tasks.

Primitive Functions

Figure 2 shows the seven DFPL primitive functions. A Primitive Computational Function (PCF) demands all its inputs be PRESENT before it presents all its outputs. Conversely, a PCF sends DONE on all its inputs only when its receives DONE on all outputs. Two PCFs are shown here as examples, but others would be assumed depending on the type of programming to be done. The first PCF presents as output the quotient and remainder obtained by dividing one input by the other. The second PCF modifies an array by replacing an element, and presents the new array as output. This notion replaces the notion of assignment.

The CONSTANT provides a single constant value forever. The output is always PRESENT and DONE will cause a hang-up.

The FORK, which is the primary source of parallelism, merely passes its input on to all its outputs. DONE is sent back on the input when DONE arrives on all outputs.

The Inbound and Outbound SWITCHes provide conditional execution capability. The Control input, which must always be PRESENT for the SWITCH to execute, selects which way the Data is to be routed. Only one of the alternate Data inputs (outputs) should be (becomes) PRESENT. DONE is sent back on the Control and the (selected) Data input when it arrives on the selected (only) output.

The LOOP is started by presentation of a datum on its Initial path. This datum is presented on the Output path, but DONE is not yet sent on the Initial path. When a Boolean value arrives on the Control path, DONE is immediately returned on that path, and the Boolean value is retained by the LOOP. When a datum arrives on the Feedback path, DONE is also returned on that path, and the datum is retained by the LOOP. Now, when DONE arrives on the Output path, the LOOP either stops, if the Boolean was FALSE, or repeats, if the Boolean was TRUE. The LOOP repeats by presenting on the Output path the datum retained from the Feedback path. The LOOP stops by returning DONE on the Initial path.

The Gated Presence (GPR) provides the ability to execute indeterminately. It does this by allowing one to sense the PRESENCE or absence of data on a path. The Data input of GPR is just sent straight through to the Data output as if GPR were not here. However, when any datum is

presented on the Gate input, the Sense output will be TRUE or FALSE, depending upon the PRESENCE of the Data input at the instant the Gate became PRESENT. DONE is returned on the Gate path when received on the Sense path.

These seem to be a minimal set of primitives, in that the removal of any one would make it impossible to write certain kinds of programs. It is interesting that the DONE signal is only needed for LOOPs; purely recursive programs could operate with PRESENCE alone. Also, it is interesting that no determinate function can be defined which does not need at least one input guaranteed PRESENT before it can execute. This input is needed to tell the function which others are absent.

Built-in Defined Functions

The six functions shown in Figure 2 are examples of the sort of generally useful functions which can be defined in terms of the primitives described above. It is noteworthy that they all require the LOOP in their definition.

The SOURCE function presents a constant output perpetually. It differs from the CONSTANT in that SOURCE accepts DONE, and just presents the same output again.

The SINK function merely accepts an input and returns DONE. The input value is disposed of.

The LOOPTLF is an example of a set of extended loop functions. It is analogous to the DO WHILE loop in PL/I. When started by presentation of a datum on its Initial input, LOOPTLF presents that datum on the Test output, and upon receipt of a TRUE on the Control input, presents the same datum on the Local output. Then when a new datum is presented on Feedback (and DONE received on Test and Local), LOOPTLF repeats. If FALSE was presented on Control (and DONE received on Test), LOOPTLF stops and presents the current datum on the Final output. When DONE is received on the Final output, DONE is sent back on the Initial input. Similar extensions may be had by deleting some of the Test, Local and Final outputs.

The CHOPTL function is representative of the class of choppers which serve to introduce into a loop a datum which must remain constant. When started by presentation of a datum on the Input path, CHOPTL immediately presents it on the Test output. If TRUE is received on Control, the datum is presented on the Local output. When DONE is returned to both Test and Local, CHOPTL repeats. If FALSE was presented on Control (and DONE returned on Test), then CHOPTL stops and returns DONE on Input. The Test and Local outputs provide data suitable to be combined with other Test and Local outputs respectively. Similar choppers may be defined by deleting either Test or Local outputs.

The simple MEM function behaves much like a single storage cell. When a FETCH is presented on the Control path, the current datum contained in MEM is presented on the Fetch output, and DONE is sent back on Control when received on Fetch. If a STORE is presented on Control, then the datum presented on the Store input replaces the datum contained in MEM, and DONE is automatically sent back on Control and Store. Note that the Control input must be presented in either case. It is straightforward to define more complicated memory functions such as queues or random access memories. In the latter case, an address would be presented on the Control path too.

The Priority Arbiter (PARB) function presents on its Data output the datum received on the highest numbered PRESENT Data input. it also presents the number of that Data input path on the Index output path. When DONE arrives on both Data output and Index output, DONE is returned on the chosen Data input, and PARB is ready to choose again. PARB will not present any output until at least one input is PRESENT. PARB thus provides a facility analogous to asynchronous interrupts looked at from a hardware point of view. Its implementation uses one GPR per Data input path.

A Sample DFPL Program

A sample DFPL program is shown in Figure 1. The function defined for general use is the memorylike Priority Queue (PQ). It contains the definition for the PDQ function which is local to PQ. PQ has 5 inputs which accept queue elements to be put on the queue, and 3 inputs for get requests. A queue element consists of a priority field and a data field. The queue is an array of such elements and may grow and shrink. Whenever a Put input arrives, that element is enqueued. Whenever a Get input, which is a priority number, arrives, a queue element is returned on the associated Get output. That queue element is selected which is the oldest one whose priority number is greater than or equal to the priority number presented on the Get input. If no such element is found, as special "Fail" element is outputted. The PDQ function does the queue search to provide the element and also outputs the queue with that element deleted.

Note the following: the end-to-end touching of SWITCHes or loops means that they are driven by the same Control input; PQ operates indeterminately, much like a task dispatcher, in fact; the 5 Put inputs and the 3 Get inputs are separately arbited since only one need be PRESENT at

a time and Puts must be distinguished from Gets; the third PARB ensures that Get takes priority over Put; the queue is "stored" in the LOOP which repeats indefinitely; the PCFs used here are no more complicated than APL primitives.

References

[1] Kosinski, P.R. "A Data Flow Programming Language". IBM Research Report RC4264 (March 1973).

[2] Dennis, J.B. et al. "Computation Structures". Project MAC Progress Report VIII, MIT, Cambridge, Mass. (July 1971) pp 32-44.

[3] Dijkstra, E.W. "GOTO Statement Considered Harmful". Letter to the Editor, CACM 11 (March 1968) pp 147-148.

[4] Shaw, M. and Wulf, W. "Global Variable Considered Harmful". Carnegie-Mellon University (August 1972).

[5] Mills, H. "Top Down Programming in Large Systems". Debugging Techniques in Large Systems (Editor: Rustin, R.), Prentice-Hall, Englewood Cliffs, N.J. (1971) pp 41-55.

[6] Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules". CACM 15 (December 1972) pp 1053-1058.

[7] Allen, F.E. and Cocke, J. "A Catalogue of Optimizing Transformations". IBM Research Report RC3548 (September 1971).

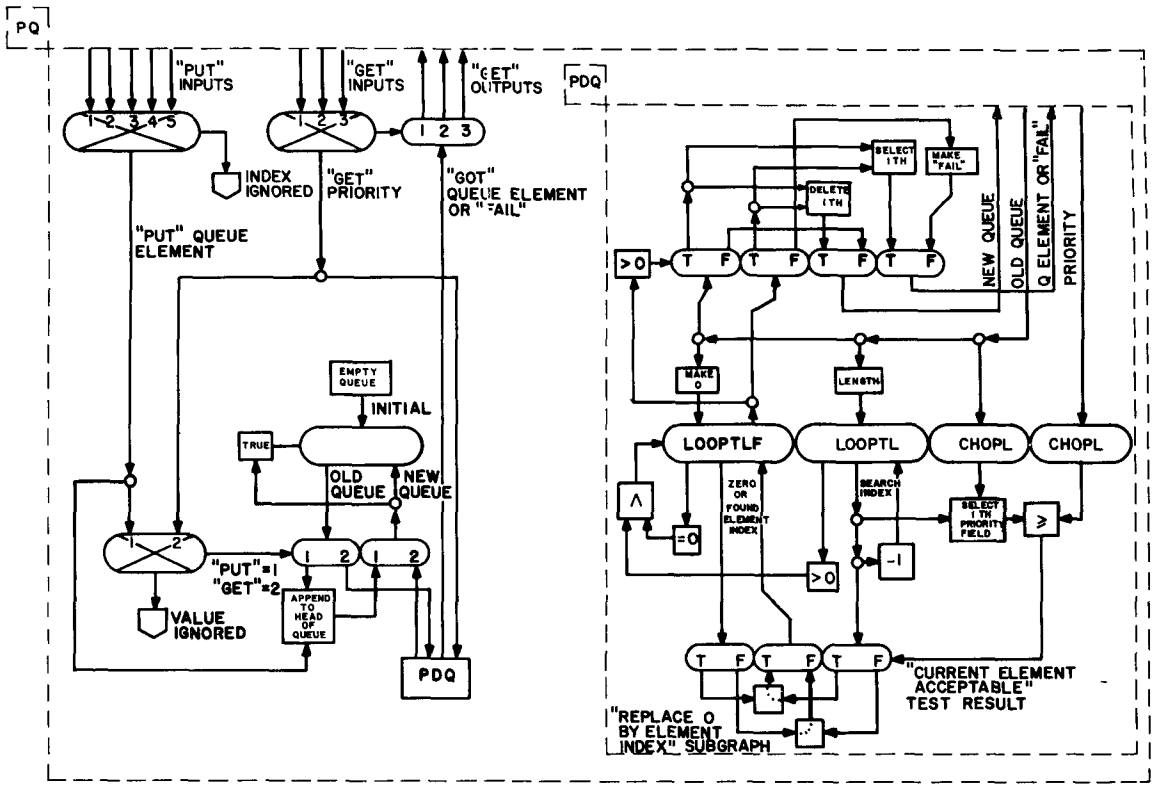

FIGURE 1

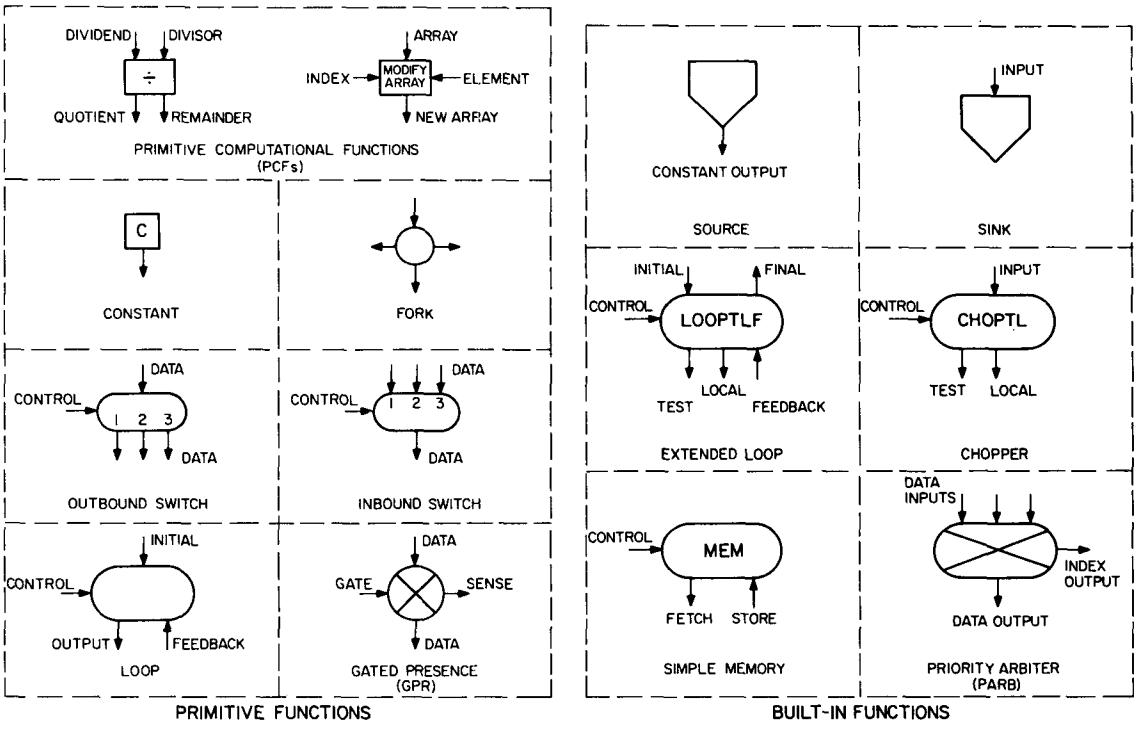

FIGURE 2