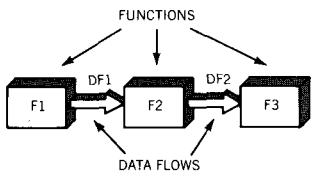
This paper presents the technique of data flow and how it can substantially improve application development productivity. Flows of data are the only connections needed between functional components of a computer program. Components which pass only data are so independent that they can easily be shared and reused. Such components can be developed independently, which substantially reduces the complexity of development and makes them much easier and faster to design, implement, test, and change. Building programs in this way can yield substantial increases in productivity over developing monolithic programs or even structures of called modules. The compatibility of data flow to natural human views of applications and other parts of data processing, such as distributed processing and high-performance architectures, is also presented. Recommendations are included.

How data flow can improve application development productivity

by W. P. Stevens

The time and effort needed to develop computer applications continues to be a major bottleneck. As hardware price/performance improves and more applications are justified, there is growing pressure to increase the productivity of application developers. A major improvement in productivity is necessary in order to have a significant effect on the problem (see Reference 1 for an example). Many companies are trying to find ways to substantially improve productivity, and IBM has expended a considerable amount of effort toward finding solutions to this problem. This paper describes why the technique of data flow can provide major improvements in application development productivity.

Figure 1 A data flow diagram (linear)



Data flow is a technique that allows programs to be developed as combinations of independent functions which are connected solely by flows of data. A data flow diagram is a convenient way to depict functions that are connected by flows of data. The connections between functions can be "linear" as in the data flow diagram in

Figure 2 A data flow diagram (network)

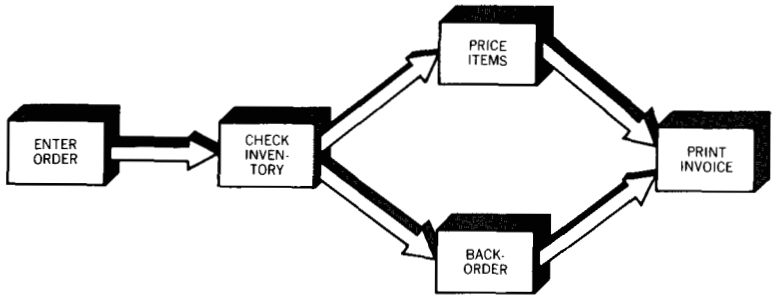


Figure 1 or can be a network as in Figure 2. Each function processes input data and puts out the resulting data. The network defines the flows of data between the various functions.

Benefits of using data flow come from

- Reusable functions
- Reduced complexity
- Easy use of subassemblies
- Natural, consistent application view
- Easier application development

The most important of these is the ease with which previously developed functions and/or combinations of functions can be used and reused. The productivity improvements from reusing functions can be so significant that this benefit alone can justify basing application development on the data flow approach. Functions can output data *without naming* the target functions. The destination of each flow of data is specified externally to the function in the network definition. The network definition is used at execution time by a system service which does the physical passing of data between functions. This makes it possible to reuse functions in other programs without having to modify them. The ability to reuse a function in another program is then determined by the need for that function and the format of the data which the function uses. Networks can be specified as easily as F1 & F2 & F3 for Figure 1—where the & means that the output of one function is to be fed into the input of the next. The UNIX™ Operating System can do this for a linear sequence of functions with the syntax F1 | F2 | F3.[2] The relationships between functions in a data flow network are similar to the relationships between programs in a traditional system "flowchart" which depicts jobs and/or job steps and their input and output data. The programs are independent of each other and can be run at different times (i.e., asynchronously) since they are only connected by input and output data.

DeMarco[3] and others have shown that users view manual applications as functions connected by flows of data. The data flow technique directly implements this application view. Thus, the application user is able to understand the design of the application since it can be made identical to his application view.

It has been known for some time that modularity—building programs from small independent pieces—plays a crucial role in reducing the complexity of application development and in improving the ability to reuse developed functions.[4] The more independent the pieces, the greater the reduction in complexity. Data flow provides greater independence between pieces than other modularization techniques (see the section, *Reduced complexity*).

Data flow implementations

Implementing a program as a combination of functions connected by data flows consists of defining the network (e.g., by specifying F1 & F2 & F3 for the example in Figure 1) and creating or obtaining the necessary functions. The functions are implemented as independently compiled subroutines. Each function can be developed almost totally independently of the other functions. They can be developed by different techniques, at different times, in different languages, by different people, in different locations, and even by different companies—as is most appropriate for that particular function. Most important, if functions exist that do what is needed, they do not have to be redone—they can simply be reused. The constraints are that they input and output the desired data and execute in the target environment.

Data flow architectures already exist for programs. The job control languages of Multiple Virtual Storage (MVS) and Disk Operating System/Virtual Storage Extended (DOS/VSE) define functions (job steps) and the flows of data (files) among them. On-line systems contain independent transactions which are initiated based on data that is input by the terminal user. Distributed systems can execute programs independently and asynchronously on different processors which send data to each other.

There are also implementations of data flow mechanisms within a program. One example is described by Morrison.[5] He shows how a program can be specified as a network of functions connected by flows of data. Functions call a data flow service to get or put their data. The data flow service transfers data to subsequent functions based on data flow connections specified separately in an application network (such as Figure 2) rather than within the individual functions. The data flow service also schedules the functions, which can be run asynchronously—any function can run if it has input data to

process. The distinction between multiple data flows in or out of a function is made by indicating a numeric "port" number for the flow.

Data flow within a program is also used in the UNIX Operating System.[2] Here, the "shell" has the capability to dynamically specify programs as combinations of reusable functions. The data from each function is routed to the next function based on the specified function sequence.

In both the UNIX system and the Morrison approach, a function can pass data to other functions as if simply doing I/O operations to a sequential file. For example, simple functions could have the following form:

```
INPUT ──→ ┌──────────────────────┐
          │ START                │
          │ GET INPUT DATA       │
          │ PROCESS THE DATA     │
          │ PUT RESULTING DATA   │──→ OUTPUT
          │ RETURN               │
          └──────────────────────┘
```

Functions with more sophisticated logic, such as initial or end of file processing, and multiple input or output streams, could take the following form:

```
                ┌──────────────────────┐
                │ START                │
                │ INITIAL PROCESSING   │
                │ DO UNTIL EOF         │
                │    LOGIC             │
INPUT 1 ──→     │    GET INPUT DATA 1  │
                │    LOGIC             │
INPUT 2 ──→     │    GET INPUT DATA 2  │
                │    PROCESS THE DATA  │
                │    PUT RESULTING DATA│──→ OUTPUT
                │    MORE LOGIC        │
                │ END DO               │
                │ EOF PROCESSING       │
                │ RETURN               │
                └──────────────────────┘
```

The ability to handle input and output data as though they were sequential files keeps each function independent of the source(s) and destination(s) of its data. A function can thus be used in any program where its transformation of input data to output data (e.g., a rate calculation) is needed. Data flow allows *all* of the functions in the program to be written in the same reusable form.

The first form shown above is similar to a callable subroutine that is passed data via call parameters. However, in order to reuse callable subroutines, it is necessary to write a calling routine. Writing this routine can require significantly more work than only having to specify F1 & F2 & F3. Also, callable subroutines can become difficult to program when two or more input and/or output data streams are needed, or even when the data varies in content or occurrence. It is much simpler to program such functions if *they* can call a service to get their data rather than be called and have their data passed to them. For example, consider the network in Figure 3. Here, functions

Figure 3 A network example

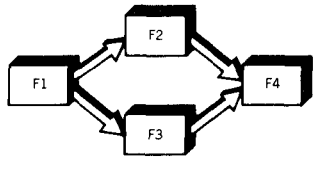
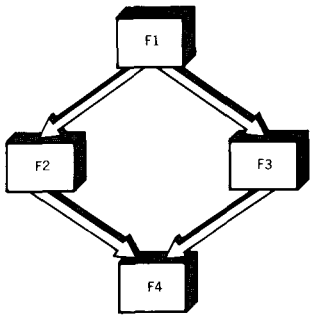


Figure 4 Sample coding for all functions

```
F1: START          F2: START          F3: START          F4: START
    LOGIC              GET INPUT 1        GET INPUT 1        LOGIC
    GET INPUT 1        PROCESS IT         PROCESS IT         GET INPUT 1
    LOGIC              PUT OUTPUT 1       PUT OUTPUT 1       LOGIC
    GET INPUT 2        RETURN             RETURN             GET INPUT 2
    MORE LOGIC                                               MORE LOGIC
    RETURN                                                   RETURN
```

Figure 5 One call structure

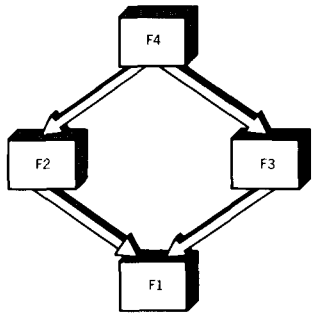


F1 and F4 each deal with two data streams. It is easier and more flexible if *both* F1 and F4 can be developed as shown in Figure 4.

Implementing the functions in Figure 3 as a hierarchy of called modules results in the limitation that there can be only one module at the top of the hierarchy. The basic alternatives are shown in Figures 5, 6, and 7. Either F1 or F4 or both of them end up having to be called by other modules which pass the data. In Figure 5, for example, the implementation of F4 is complicated because it must save input 1 until it gets called with input 2. If its logic can result in F4 needing a second input 1 before being able to accept an input 2 (e.g., the first input was not valid), it gets even more difficult. More logic will have to be implemented in F4, and maybe also in F2, to give F4 the ability to access multiple inputs before F2 returns to F1. This added complexity results from the imposed limitation that a call hierarchy has only one module at the top.

What is needed is the ability for all functions to be able to invoke a service to get or put their data as in Figure 8. Then *all* functions can be coded as in Figure 4.

Figure 6 The inverse call structure



Programming the merge function in Figure 9 demonstrates the above problem. It is difficult to program a merge as a *called* subroutine since the *calling* routine would not know which record to pass next. The called merge routine is the one that has the information about which input is needed next. However, it is straightforward to program the merge as a data flow function, since it can then decide which input it needs next and get that input from the program data streams.

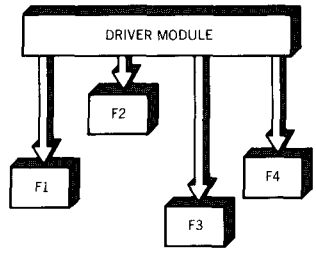
The inherent limitations of implementing programs as hierarchies of functional modules which call each other are as follows:

1. The calling module names the called module and thus cannot easily be used in other programs that need other functions as targets for that output data.
2. The calling module passes control along with the data and thus becomes responsible for determining when the called module should execute.
3. Hierarchies result in less flexibility and independence between modules than networks do. Modules can only share data with modules they call rather than any other module in the program.

4. Applications being automated often have network flows of data which cannot be directly mapped into a call hierarchy. Thus the developer must manually transform the desired network flow into the hierarchial form—a task which can become very difficult.

The belief that modularization always makes the performance of a program worse is a myth. Instead, it is likely that a modular program will run *faster* than if it were developed as a monolithic program (see Chapter 11 of Reference 6). Morrison[5] further shows that the data flow approach also allows the option of exploiting parallelism to improve the performance of data-flow-connected programs—a technique not possible for monolithic programs or even for ones modularized with structured design. Thus, data flow connections need not be detrimental to performance and can even be a valuable way to *improve* the performance of programs.

Figure 7 All functions called

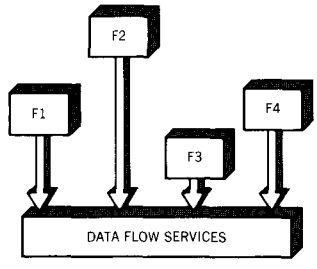


The ability to reuse functions

The ability with which functions of a program can be reused varies from extremely difficult to trivial based on how highly related the functions are to each other. The lower the relationship between functions, the easier it is to reuse them. The most independent, reusable functions are small modules that are separately compiled, do only a single transformation of data, and share minimum data with other functions (see Reference 6). Functions can be related in one or more of the following ways:

1. Sharing data with other functions
2. Transferring control to another named function (e.g., via a call statement)
3. Being compiled into the same physical module
4. Sharing the same local variable and line label definitions
5. Branching into and out of the code of other functions
6. Being physically spread throughout other functions (i.e., the code of the function is not contiguous)

Figure 8 All functions call for service



Sharing data is a necessary requirement in order for functions to be part of the same program. With the data flow approach, Item 1 is the *only* relationship between functions in a program.

Functions within large monolithic programs are often related in all of the above ways. As a result, it is so hard to reuse (or even find) functions that it is easier to develop them again. Structured programming[7] done using INCLUDE to reference source language segments eliminates Items 5 and 6 because of its one-in one-out segments of code. It makes the functions easier to reuse, but not easy enough to make reuse practical. The key difficulties are that: (1) the programmer must understand the old program in order to extract a function; (2) variable names and labels have to be cross checked and changed

Figure 9 An example with two inputs

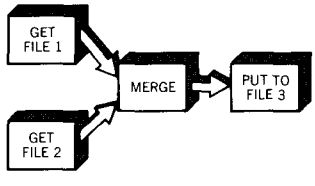
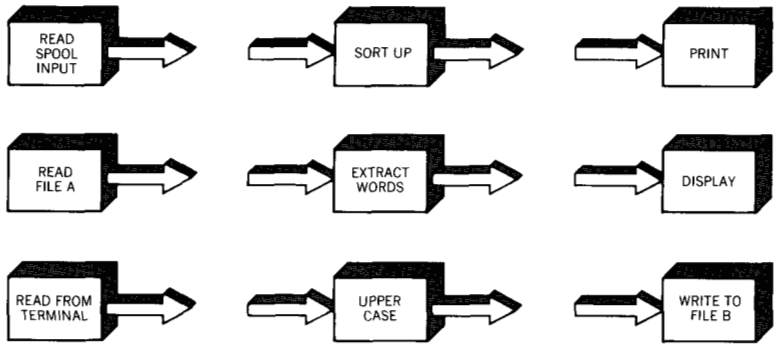


Figure 10 Data flow modules



in cases of duplication with names and labels in the new code or in other functions to be reused; and (3) the functions all have to be in the same language.

Structured design[4] eliminates Items 3 to 6 by dividing programs into hierarchies of separately compiled modules which call each other passing data as CALL parameters. When a function is separately compiled, reuse is practical because no extra work is required to understand, extract, and reuse it. However, passing data via a CALL statement requires specifying the *name* of the target function and passing *control* to it with the data. Thus, it is difficult to reuse the calling routine in another program where the target function is different. It is also hard to pass multiple data streams to a function from different sources—especially when the target function should not execute until all of the data inputs are available (see discussion of "merge" above). Thus, it is hard to implement general networks of functions as hierarchies of called subroutines. Also, when functions pass control to each other, they cannot be run asynchronously (see the section, *Natural, consistent view*).

A function is much easier to use when it does not pass control to other functions. Thus, even for hierarchies of called modules, reuse is more likely for those modules that call no others—i.e., those where only Item 1 happens to exist for the module. Data flow provides more independence between functions within a program. With data flow connections, a program can be created out of functions that are only related by the data they pass. It is not necessary for *any* of the functions to call or include any others. Thus, it is possible to reuse any function in other programs without having to change it.

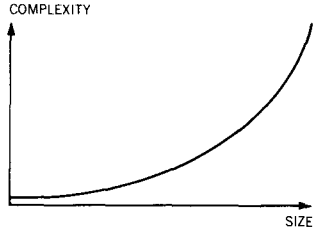
Once functions are easily reusable, it becomes possible to develop programs considerably faster than is usually done. For example, the functions in Figure 10 can be used to create several programs very rapidly. The data into and out of the functions in this example are 80-character text-records.

In order to read File A, to sort the records in an ascending sequence, and to display them, combine READ FILE A, SORT UP, and DISPLAY (Figure 11). In order to read terminal lines, to make all letters upper case, and to print them, combine READ FROM TERMINAL, UPPER CASE, and PRINT. To put the words from input text onto File B in ascending sequence, combine READ SPOOL INPUT, EXTRACT WORDS, SORT UP, and WRITE TO FILE B. Other programs can be built in the same fashion. To display words from File A without seeing duplicates requires another function to be obtained or created (ELIMINATE DUPLICATE WORDS)—but does not require the whole program to be developed from scratch. To the extent that needed or usable functions are available, the implementation phase can be orders of magnitude faster than with any implementation technique that requires the functions to be written (i.e., compare the time to enter F1 & F2 & F3 with the time to develop and test three functions by any technique).

Figure 11 One program

The ability to reuse functions is critical to dramatic improvements in programming productivity. It is the only way to eliminate major parts of the application development task. The ability to reuse functions is enhanced by making them more independent of one another. The data flow technique provides a higher level of independence between program modules than ways which pass data *and* control, such as the CALL statement.

Figure 12 Complexity grows exponentially with size

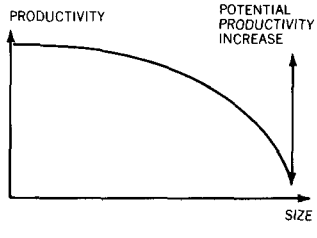


Reduced complexity

The data processing industry is constantly pushing against the limits of what can be produced. Complexity is one of the biggest limitations. As the size of a program to be developed increases, the time and complexity to implement it increase exponentially—as in Figure 12 (see Reference 6).

The relationship is exponential primarily because the number of possible interrelations between the parts of the programs grows exponentially. The developer must understand and cope with all of these possible connections. Figure 13 shows the resulting productivity as a function of size. For large programs implementation complexity can exceed the capability of even a team of developers. Increasing the size of a team is often counterproductive unless the job can be broken into independent subtasks.[8]

Figure 13 Resulting productivity loss



Modularizing a program reduces the number of possible interrelations, therefore reducing the implementation complexity. The productivity approaches that of developing small programs because the program is created out of what are effectively a number of small programs. The more independent the modules are, the closer the productivity is to that of developing small programs. Since data flow allows more independence between the modules than hierarchies of

called routines, its potential for productivity is higher than structured design—which, in turn, is higher than structured programming with its more highly connected segments. Since the complexity, and thus the productivity, varies exponentially with the size, the improvement in productivity resulting from reducing the effective size of large applications can be a factor of two or three or more. Also, the more independent the functions are, the easier it is to shorten the elapsed implementation time by spreading it among many developers. This is true because independent functions require less communication among the developers of those functions.

In order to make substantial strides in the level of complexity that can be handled, it is also imperative to provide good support for independent program subassemblies. A subassembly is a combination of pieces which, itself, can be used in a larger assembly as if it were a single piece. Application developers can build highly complex systems as long as those systems can be constructed from smaller, understandable pieces, where each piece may itself be complex and be built of smaller and simpler pieces. The concept of subassemblies is widely used in manufacturing where subassemblies reduce manufacturing costs and raise the level of complexity that can be handled.

In program development, the use of the subassembly concept at the program level has been limited by lack of adequate characteristics of program modules and by lack of adequate support facilities in the operating and subsystem environments. Support of subassemblies requires maximum reusability and independence between the pieces of a program and the ability to connect pieces into larger combinations of pieces (subassemblies). The data flow approach provides the needed reusability and independence. Data flow also provides a simple way to combine pieces by specifying only the data flow connections.

subassembly advantages

The use of subassemblies provides multiple advantages, including the following:

- Reduced complexity for the developer
- Reusable standard assemblies
- Parallel design and/or development

By far, the biggest advantage of the use of subassemblies is the reduction in complexity it provides. The key to productive use of a subassembly is the ability to use it as a single function without having to know that it is internally made up of multiple functions. For example, the SORT UP function in Figure 10 may be a subassembly of 100 functions. Yet it is possible to build programs with it *without* having to know whether it is a subassembly or not. That is the point of subassemblies. If the components of the subassembly have to be changed *however slightly* in order to use it, then the major advantage of subassemblies is lost. Thus, a crucial requirement in order to use

the subassembly concept is the ease with which functions can be reused—it must be possible to reuse functions without having to change them.

Manufacturing industries already rely heavily on the use of subassemblies. Airplanes, cars, boats, and all kinds of vehicles, instruments, tools, buildings, clothing, telephones, computers, etc. are made from subassemblies. In fact, most manufacturing industries could not even produce these complex products if it were not for the subassembly concept. Once the system interconnections between the major functional units have been determined, the requirements for the subassemblies can be specified. Then the subassemblies can be built in parallel or simply provided off the shelf by their suppliers, which substantially reduces the time necessary to develop products.

An airplane manufacturer, for example, could never build one of today's complicated planes if it were necessary to design and build every component of the plane, including the glass, metal, copper wires, paneling, rugs, etc. Similarly, architects could never handle the complexity of designing a building if they also had to design the windows, cabinets, chairs, tables, desks, phone equipment, or even just the I-beams, wall board, and heating systems used in buildings.

The sheer complexity of having to design and build all of the subassemblies and the reusable parts of most products is beyond any single manufacturer's capability. No single manufacturer could have the skills, let alone the necessary techniques, tools, or time to ever do the job.

subassemblies in programming

The subassembly concept is already in use within data processing, too. The programmed intelligence of an application usually includes the following levels:

- Job stream
- Program (job step) or on-line transaction
- Subroutine/macro
- High-level language statement
- Machine instruction
- Microcode instruction

Each level is programmed by assembling combinations of components from lower levels. Those components can be used as though they were single functions, even if they are subassemblies of components themselves. For example, a subroutine is called as a single function but can contain many language statements. A high-level language statement may invoke many machine instructions. Machine instructions may be implemented from microcode instructions. Whether a function is a subassembly or not should be irrelevant to the user at the next higher level. The reduction in complexity occurs

when components from lower levels can be used without having to understand their internals.

There has been too little use of this subassembly concept at the program/transaction level. This level can include thousands or even hundreds of thousands of instructions without any intermediate subassembly levels. The resulting complexity (see Figure 12) can drastically reduce productivity.

Data flow allows networks of functions to be used as subassemblies in other networks. DeMarco[3] recommends depicting applications as hierarchies of data flow diagrams in order to make them easier to understand. The use of such hierarchies can even be crucial to the ability to *understand* a large application. For the same reasons, the ability to develop a large program can be greatly simplified by being able to implement it as a hierarchy of subassemblies.

Natural, consistent view

There is a growing, worldwide interest in data flow methodologies and their broad applicability to the information processing business. Data flow networks are a natural way for people to depict applications. The data flow view of an application can be used throughout the development process. Data flow architectures are also affecting hardware design, since they offer the prospect of parallel machines vastly more powerful than the Von Neumann machines of today.

People can view interrelated functions naturally as data flow networks because data flow networks reflect how *people* interrelate. Typically, people perform functions and share data with other people who have other functions that they perform (asynchronously). Data flow diagrams picture these interrelations as they exist naturally. The information processing needs of an organization can be modeled with data flow diagrams which show the flows of information among functions. For example, Business Systems Planning (BSP)[9,10] is a technique for analyzing and understanding the information processing needs of a business by analyzing the flow of data among the functional units of the business. Similarly, applications can be described using data flow diagrams as is done in structured analysis[3] and Structured Analysis Documentation Technique (SADT™).[11]

The view of an application that the development system implements is crucial to productivity. Developers can be more productive if the system they are building is easier to understand (see pages 446–449 of Reference 12). Manual applications consist of information flowing among people who do functions asynchronously with one another. Thus, if the application can be implemented as a data flow network of functions, the developer can work with a natural form he or she already understands.

In most current development systems the developer has to make a transition from the data flow of the manual application to a procedural view somewhere during the development process. This transition is necessary because today's Von Neumann computer hardware is procedurally oriented. Analysis and design of programs has long been done with flowcharts because they model how the *computer* executes an application. However, since flowcharts do not model the user's application directly, the user cannot verify that the flowchart correctly represents his application needs.

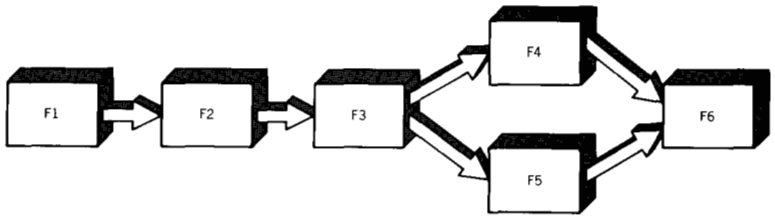
Drawing the application as a data flow diagram during analysis and design avoids having the developer translate it to some procedural form and also depicts the application in a form that the end user can understand, verify, and change effectively. Being able to then implement the data flow diagram directly can delay the transition to a procedural view until individual functions are to be created. The transition to the machine-oriented procedural view can be avoided entirely for those functions that already exist or which can be generated with nonprocedural generators (see the section, *Easier application development*). In any case, the procedurally oriented view can be isolated to the implementation of small individual functions.

Underscoring the value of using data flow as a consistent view of the application are the characteristics of distributed and high-performance architectures. An application developed to run in a distributed environment—where parts of the same program/application are run asynchronously on distributed processors and send data to each other—requires a data flow view of the processing. Thus, if the application, which starts out with a data flow view, is changed to a procedural view, it will have to be *changed back* again in order to run it in a distributed form.

Current high-performance hardware architectures and multiprocessors already exploit parallelism. Programs that are developed as monoliths, or out of callable subroutines, cannot take advantage of multiprocessors *within* a given program. (The parallelism of multiple processors is exploited either through multitasking or by running multiple programs.) Functions that are connected by data flows are so independent that they can be run asynchronously.

Current high-performance single processors "pipeline" executing instructions by doing a dynamic data flow analysis of the instruction stream. Optimizing compilers go through a similar approach in order to streamline execution of high-level procedural language coding. Future high-performance architectures currently being researched also plan to take advantage of parallelism to improve performance.[13] With these architectures, it would be possible to avoid translating to a procedural view for all functions rather than only when they already exist or can be generated.

Figure 14 The final program



Data flow can also *reduce* the total work to be done. Consider the following instructions (from Morrison[5]):

MOVE A TO B
MOVE C TO D

The execution of these instructions can be done in either order, or even simultaneously. Procedural approaches (e.g., current procedural languages) require that the developer decide which instruction to place first. This decision is then frozen and is hard to change. It is impossible to distinguish later which precedences were necessary and which were arbitrary. The procedural approach, therefore, results in unnecessary work and complexity and an increased potential for errors (e.g., if the maintenance programmer decides a particular precedence was arbitrary when it was not). With data flow architectures, only the necessary data flow precedences are specified to the computer. The developer never needs to specify unnecessary precedences forced simply by the procedural nature of the machine.

Easier application development

Figure 15 A first prototype

Data flow provides advantages for all phases of the application development cycle. The application identification, analysis, and design phases were discussed previously. Data flow also makes it much easier to prototype, create, test, change, and thus, to maintain and optimize programs. Each of these activities is easier when functions can be extracted and dealt with independently of the program—and when programs can be dealt with independently of any particular function. And, because the activities are easier, they can be done more completely and accurately—resulting in increased quality.

Prototyping is widely recommended but seldom done. One difficulty is that prototyping is often accomplished by building something by a different, though quicker, method. The prototype is evaluated and changed before construction by the preferred, but slower and harder, method. Data flow connections can allow prototyping to be done *without* duplicating the implementation. Skeleton or similar existing functions can be connected to form a prototype of the final program.

Figure 16 A later prototype

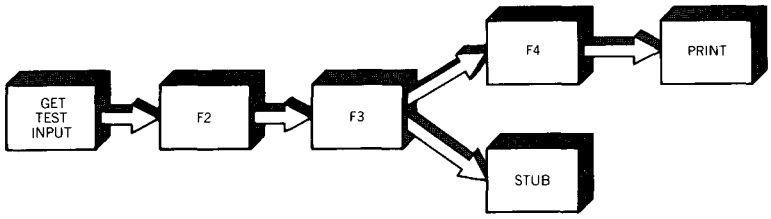
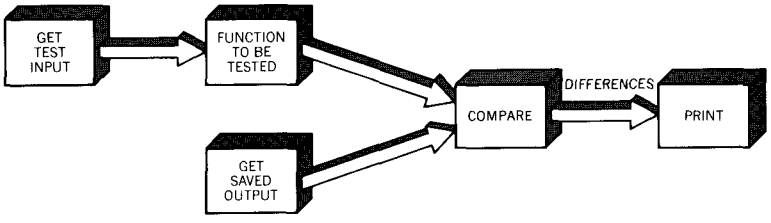


Figure 17 A testing network



As each function is implemented, it is added to the prototype in what becomes the final program as illustrated by the example in Figures 14–16.

Modules constructed for a prototype can later be replaced by more specialized functions without disrupting the system. But the need to construct special modules for the prototype should decrease as the library of existing functions increases. This way, the prototyping effort is not extra, but rather is simply the first step of building the final program.

Testing is much easier when a function can be easily inserted into a network that can automatically test the function (as in Figure 17). Input and output data for the test can be generated, entered manually, or saved from a test that is run with manual input. The test network could then feed the same data into the function or program to be tested, compare the output against previously saved output, and report any differences. It may even be possible to automate the testing of some functions completely by inserting them—without change—into testing programs which use pre- and post-condition specifications to validate test results automatically.

Regression testing is another practice that is highly recommended but seldom implemented. The difficulty seems to be the human involvement necessary to do the testing. The testing approach discussed above could allow regression testing to be automated.

In a data flow environment, strictly separating I/O functions from logic functions can also facilitate testing. When the I/O code and the application logic are bound into the same unit, they must be tested together. By separating the I/O and logic functions, the I/O functions simply get the data from the I/O device and pass it (unchanged) as a data stream to the logic function(s). With data flow connections it is then trivial to replace an I/O function with a (test) I/O function that passes the same stream of data from a test file (as in Figure 17). Thus, the logic can be tested independently of the I/O functions. The logic can even be tested on a physically separate development system that does not support the same I/O operations as the target system. I/O functions can be tested by connecting their output data stream to a test function that displays, prints, or automatically checks that data.

It would also be easier to create generators for new I/O functions if the functions to be generated only had to move the data between the external device and memory. The generator would need only the nonprocedural definitions of the data on the external device and the data in memory. Generators for new logic functions would also be easier to create and use if they did not also have to generate all of the various types of I/O capabilities. Such application *component* generators could greatly simplify the complexity and effort of application development by handling the complexity of doing I/O operations and a portion of the logic automatically. Data flow provides an easy way to connect independent I/O and logic functions, thus facilitating the development and use of such application *component* generators.

Functions are much easier to change if they are independent and replaceable. The developer can work with only the function to be modified rather than the entire program. Modifications to a function can be tested independently of the program before reinserting the modified function. The original function can be left in place until the modifications are completely tested and an archived copy of it used again later if errors show up in the modified version.

Conclusion and recommendations

Use of the data flow technique is a key to increasing the productivity of the application developer. It allows greater independence between functions of an application, which substantially improves the ease with which functions can be *reused* and can greatly *reduce complexity*. These two advantages alone can provide a substantial increase in productivity, even with today's tools. The independence also makes it easier to understand and deal with the functions, thus making it easier to prototype, develop, test, and change them. Each of these tasks can be much easier when each function can be dealt with independently of the other functions in any particular program.

Most manufacturing industries build products out of reusable subassemblies. This approach is fundamental to the ability and speed with which complex products are produced. It is also used to substantial advantage in traditional data processing above and below the program level. The ease of reuse provided by data flow allows making productive use of the subassembly concept *within* a program.

Data flow is a natural view of a manual application that can be used consistently throughout the development process. The data flow view is also consistent with the relationships among steps in a job and between nodes in a distributed processing network.

To capitalize on the advantages provided by data flow, begin integrating tools and techniques which support data flow into your application development process. A valuable first step is to use data flow diagrams in the analysis and design phases, which will allow the user to view his application in a one-to-one relationship with manual procedures. Documenting analysis and design with data flow diagrams can greatly improve the communication between the data processing developers and the end users and also make it easier for the data processing developer to do analysis and design. Business Systems Planning[9,10] is a valuable way to do application identification. It provides information about data flows within the company, which can be used in the analysis and design of applications.

The use of data flow diagrams to document the output of application identification, analysis, and design phases can avoid the transition to procedural representation until the implementation phase of the development cycle. This provides substantial benefits in the early phases of development and results in designs which can exploit data flow capabilities provided in the execution environment.

CITED REFERENCES

1. J. Martin, "With current programming methods applications increase unattainable, Martin says," *Computerworld* **XIV** (December 8, 1980), p. 21.
2. S. R. Bourne, "UNIX time-sharing system: The UNIX shell," *The Bell System Technical Journal* **57,** No. 6, Part 2, 1970–1973 (1978).
3. T. DeMarco, *Structured Analysis and System Specification,* Yourdon, Inc., New York (1978), pp. 1–125.
4. E. Yourdon and L. Constantine, *Structured Design,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1979).
5. J. P. Morrison, "Data Stream Linkage Mechanism," *IBM Systems Journal* **17,** No. 4, 383–408 (1978).
6. W. P. Stevens, *Using Structured Design,* John Wiley & Sons, Inc., New York (1981).
7. J. K. Hughes and J. I. Michtom, *A Structured Approach to Programming,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1977).
8. F. P. Brooks, *The Mythical Man-Month,* Addison-Wesley Publishing Company, Reading, MA (1978).

9. "Business Systems Planning—Information Systems Planning Guide," GE20-0527, IBM Corporation (July 1981); available through IBM branch offices.
10. J. A. Zachman, "Business Systems Planning and Business Information Control Study: A comparison," *IBM Systems Journal* **21,** No. 1, 31–53 (1982).
11. *An Introduction to SADT—Structured Analysis and Design Technique*, 9022-78R, Softech, Inc., Boston (November 1976).
12. W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics*, Second Edition, Chapter 28, McGraw-Hill Book Company, Inc., New York (1979), pp. 443–467.
13. J. B. Dennis, "Data flow supercomputers," *Computer* **13,** No. 11, 48–56 (November 1980).

The author is in the IBM Information Systems Group staff, 1000 Westchester Avenue, White Plains, NY 10604.

Reprint Order No. G321-5165.