Flow-Based Programming

J. Paul Morrison

March, 2007

Internet:

paul.morrison@rogers.com

The 2nd edition of book is now (May, 2010) available from CreateSpace's eStore - https://www.createspace.com/3439170, and at Amazon.com - http://www.amazon.com/Flow-Based-Programming-2nd-Application-Development/dp/1451542321/. Take a look and see how FBP has changed in 15 years!

Herakleitos, in about 500 BC, said

Panta rhei

usually translated as "Everything changes". What he actually said was: "Everything flows".

Various approaches to structuring applications based on the concept of multiple asynchronous processes communicating via streams of data have appeared over the years in the data processing literature, and actually this concept seems to get rediscovered independently on a regular basis, both by academics and by developers of business applications and software. In spite of this, it still has not caught on widely in our business "where the rubber meets the road", for a variety of technical, psychological and social reasons. However, one system of this type

has been in use in one of the biggest companies in Canada, supporting the batch processing portion of their most complex and most critical application, for the last 20 years. During this time, this application has been changing and adapting to new hardware, new software, and new business requirements, so it is clearly very robust, and it also performs well enough to support the data volumes required by one of Canada's largest corporations. During this time, although the application has changed drastically, the basic infrastructure has remained the same. Many of its basic concepts have been in the public domain since 1978, described in detail in a paper that listed many of its advantages and gave a very explicit description of how to build such a system. Unfortunately, I believe it was far ahead of its time, and, as far as I can tell, its effect on our industry as a whole was negligible. I believe that it can revolutionize our industry, and I am hoping that in the next few years some of this potential will be realized.

Over the last 20 years, I worked for IBM Canada and had the pleasure of helping to develop several derivatives of the original system. During this time, we gained much practical experience with these concepts, and used them productively for a number of applications within IBM Canada. Eventually, when, rather to my surprise, I found myself retired from IBM (after a mere 33 years), I decided to write a book describing these concepts and our experiences with them, and Van Nostrand Reinhold in New York graciously agreed to publish it. In what follows, I will try to touch on some of the high points.

The kind of programming most of us in the data processing industry do has its roots in the procedural programming that arose during the 40's and 50's: the new invention called computers filled a growing need for repetitive, mainly mathematical calculations, such as tide tables, ballistics and census calculations. In these areas, computers were wildly successful. However, even then, some of the experts in this new world of computing were starting to question whether procedural application programming was really appropriate for building business applications. The combination of more and more complex systems, competition between companies, the sheer difficulty of the medium programmers work in and the need for businesses to reduce overhead is resulting in more and more pressure on today's programming professionals. At one time, it was predicted that more telephone switchboard operators would be needed than the total number of available young ladies. Of course, this problem was solved by the development of automatic telephone switching systems. Similarly, many people believe the present situation in computing can only be solved by a quantum jump in technology, and of course each new software technology claims to be the long-awaited solution. I and a number of other people believe that the concepts described in my book really do have the potential to solve this problem. However, they represent a true paradigm change which fundamentally changes the way we look at the programming process, and therefore a certain amount of unlearning is required for most application development professionals to use it. Paradoxically, these concepts are ones which should be very familiar to the attendees of this conference, but to my knowledge very few writers have realized the advantages in

programming productivity and system maintainability that can result from them.

After relatively few years in the computer business, I found myself puzzling over why application programming should be so hard. Its complexity is certainly not the complexity of complex algorithms or logic. The vast majority of business applications do fairly simple things like transforming data from one format to another, accumulating totals or looking up information in one file and incorporating it into another file or a report. I and a number of other workers in the field have come to believe that the main cause of the problem is in fact the same thing that powered the computer revolution itself, namely the von Neumann computer model. As described in an article by Valiant (1990), the power of this model derives from the fact that it has acted as a bridge between the twin "diverse and chaotic" worlds (as Valiant calls them) of hardware and software, while allowing them to evolve separately. But, by the same token, its very success convinced its practitioners that the problems we are facing cannot possibly be due to any fundamental problems with this set of concepts. Programmers are not bright enough, they don't have good enough tools, management doesn't control them tightly enough, or they don't have enough mathematical education. Of course, none of these are valid explanations for the problems we are seeing. There is a far more fundamental problem - namely that, at a basic level, the tool is simply inappropriate for the task at hand. A von Neumann-style program can only do one thing at a time, so business logic, which is primarily a logic of data streams and transformations, has to be forced into a strait jacket of sequentiality.

Valiant says the time is now ripe for a new paradigm to replace the von Neumann model as the bridging model between hardware and software. The paradigm I and my colleagues have been using, which we refer to as Flow-Based Programming (or FBP), is similar to the one Valiant proposes and is in fact one of a family of related approaches, all of which suggest that we have to relax the tight sequential programming style characteristic of the von Neumann machine. Machines and programs have to become more like the real world, with many processes all communicating by means of data. If you look at applications larger than a single program or go down inside the machine, you will find many processes going on in parallel. It is only within a single program (job step or transaction) that you still find strict traditional, sequential logic, and much of the logic in a traditional program has to do with data synchronization, not with business logic. We as an industry have tended to believe that the tight control of execution sequence imposed by this approach is the only way to get reliable systems. It turns out that machines (and people) work more efficiently if you only retain the constraints that matter to your logic and relax the ones that don't, and that you can do this without any loss of reliability, and often with improved performance.

FBP is best understood as a *coordination* language (to use the term Gelernter and Carriero introduce in their 1992 paper describing their system called "Linda"), rather than as a *programming* language. Coordination and modularity are two sides of the same coin, and several years ago Nate Edwards of IBM (Edwards 1974)

coined the term "configurable modularity" to denote an ability to reuse independent components just by changing their interconnections, which in his view characterizes all successful reuse systems, and indeed all systems which can be described as "engineered". One of the important characteristics of systems exhibiting configurable modularity is that you can build them out of "black box" reusable modules, much like the chips which are used to build logic in hardware. While you have to have something to connect them together with, they do not have to be modified *internally* to make this happen. In FBP, these black boxes, called "components", are the basic building blocks that a developer uses to build an application. The emphasis shifts from building everything new to connecting preexisting pieces and only building new when building a new component is cost-justified. If you look at the literature of programming from this standpoint, you will find that almost all writers write from the basis of *creating new code*, rather than *reuse* - in fact the very term "reuse" seems to suggest an element of surprise, as if reuse were a fortuitous occurrence that happens seldom and usually by accident! Reuse ought to be the norm, as it is in most walks of life other than programming. Here is a quote from an IBM task-force evaluating one of the dialects of FBP developed by IBM Canada in 1988 for the Japanese market: "Reuse in DFDM is natural. DFDM's technology is unsurpassed in its promotion of reuse as compared to other reuse technologies currently being promoted".

The FBP systems which have been built over the last 25 years have basically all had the following components:

- a number of precoded, pretested functions, provided in object code form ("black boxes"), not source code form, with predefined plugs and sockets (called "ports")
- a "driver" - a piece of software which coordinates the modules comprising an application, and implements the Application Programming Interface which they use to request services
- a notation for specifying how the components are connected together to construct an application and a way of turning this notation into a data structure which the driver can then "interpret" (FBP applications are designed visually, and are also most easily understood that way)
- procedures to enable the developer to convert, compile and package individual modules, networks and partial networks
- documentation (reference and tutorial) for all of the above
- (last but not least) education (preferably hands-on).

Unlike UNIX, where the processes communicate by means of streams of characters, FBP processes use streams of tagged data chunks, called "information packets" or IPs. The tags can in turn designate descriptors which can be interpreted at run-time, although this is not required by the architecture. Since the processes all run concurrently, an entire stream of IPs is usually not instantiated all at the same time, but is generated by an upstream process and consumed by a downstream process (or passed on to another process). The

connections between the processes have a finite capacity in terms of number of IPs, so you cannot get livelock, although deadlock has to be guarded against (in FBP deadlock is a design consideration and a chapter in my book is devoted to the topic of deadlock detection and prevention). Each IP must be disposed of positively by the process receiving it (much like a paper memo): by sending it on, filing it, destroying it or attaching it to another IP.

Here is an example of a fairly simple FBP network. The example shown here is of a conventional batch update. Of course, FBP can handle interactive applications very well, but this is quite a good example as batch updates are in general extremely complex to code and maintain using conventional technology. This difficulty arises from the fact that batch updates are simply not a good match with the von Neumann paradigm, but can be expressed very easily in terms of asynchronous processes communicating by means of streams of data.

Processes:

- R - reads a sequential file
- COL - "collates" two streams into one
- APPL - (subnet of) application logic
- W - writes a sequential file
- PR - prints a report

Streams:

- M - master records
- D - details
- C - collated stream
- M' - new masters
- R - report output

In the network shown above, we have six processes, two of which are executing the same code (R) concurrently. (This is possible because processes have unique states, connections and dynamic data, but can share code, provided it is read-only). In this example all the components except APPL are precoded, pretested, off-the-shelf components. APPL would tend to be custom-written for a particular application but this is a much simpler task than coding the whole update from scratch, as APPL simply sees a single merged stream of IPs, one IP at a time. Collate also can insert "bracket" IPs into its output stream, thus relieving APPL of the job of recognizing where one related group of IPs ends and another one starts.

Any component in FBP can also be defined as a subnet of processes - a network with one or more "sticky" connections. FBP applications are usually built up using functional decomposition, so that a single diagram need not comprise more than an easily graspable number of processes. FBP is thus an excellent match with Structured Analysis, as the late Wayne Stevens pointed out in a number of

publications (1982, 1985). In the above diagram, I have labelled the components and the streams - the other needed information is to identify the "ports" alluded to earlier. In earlier dialects of FBP, ports were identified using numbers. More recently we have moved to named ports, e.g. R's output port might be labelled OUT, and COL's input ports might be MAJOR and MINOR, say. We also need to identify the processes and which components (code) they are executing. In the notation of THREADS (a recent PC-based implementation of FBP), the front-end of the network could then be coded as follows:

```
Read-Masters(R) OUT -> MAJOR Collate(COL),
Read_Details(R) OUT -> MINOR Collate,
Collate OUT -> .....
```

(the component name is in brackets following the process name, but only has to be specified once for a given process).

Now that graphics hardware and software have become available at reasonable cost and performance, we would like to have graphical front-ends for our FBP systems. Since FBP is a highly visual notation, we believe that a graphical front-end will make it even more usable. Some prototype work has been done along these lines and seems to bear this idea out.

And now I would like to quote an unsolicited testimonial from a user of DFDM, one of the software systems which we have built based on these concepts (the PLI he refers to is IBM's PL/1 programming language, and "coroutines" is the DFDM term for "processes"):

> "I have a requirement to merge 23 ... reports into one .... As all reports are of different length and block size this is more difficult in a conventional PLI environment. It would have required 1 day of work to write the program and 1 day to test it. Such a program would use repetitive code. While drinking coffee 1 morning I wrote a DFDM network to do this. *It was complete before the coffee went cold* [my italics]. Due to the length of time from training to programming it took 1 day to compile the code. Had it not been for the learning curve it could have been done in 5 minutes. During testing a small error was found which took 10 minutes to correct. As 3 off-the-shelf coroutines were used, PLI was not required. 2 co-routines were used once, and 1 was used 23 times. Had it not been for DFDM, I would have told the user that his requirement was not cost justified. It took more time to write this note than the DFDM network."

In his note, Rej (short for Réjean), who, by the way, is a visually impaired application developer with many years of experience in business applications, zeroed right in on the amount of reuse he was getting, because functions he could get right off the shelf were ones he didn't have to write, test and eventually maintain! Note that Rej says he didn't *need* PL/I - some of the coroutines may

have been coded in PL/I, but as a user he wasn't aware of this - they were just black boxes to him. As well as being used for a number of projects within IBM Canada, DFDM is also the only dialect of FBP to have reached the marketplace so far (it was upgraded to "industrial strength" as a joint project between IBM Canada and IBM Japan, and it was marketed to customers in Japan).

Rej's note was especially satisfying to us because he uses special equipment which converts whatever is on his screen into spoken words. Since FBP has always seemed to me a highly visual technique, I had worried about whether visually impaired programmers would have any trouble using it, and it was very reassuring to find that Rej was able to make such productive use of this technology.

I alluded earlier to FBP's power as a natural reuse vehicle - in my book I give some statistics on component reuse with DFDM in IBM Canada. I would like to give them here as I feel they are quite suggestive. The figures for three projects are shown in the following diagram (the numbers relate to components):

PROJECT	Type	Unique	Occurrences	Reuse Factor	Figure of Merit
A	Project	133	184	1.4	0.27
	Gen Purpose	21	305	14.5	
	Total	154	489	3.2	
	GP/T	0.14	0.62		
B	Project	46	48	1.0	0.13
	Gen Purpose	17	306	18.0	
	Total	63	354	5.6	
	GP/T	0.27	0.86		
C	Project	2	54	27.0	0.01
	Gen Purpose	8	216	27.0	
	Total	10	270	27.0	
	GP/T	0.80	0.80		

In this chart, "project" components are components coded specifically for the project in question, while "general purpose" means components that are off-the-shelf (already available and officially supported). "Unique" means separate components (separate pieces of code), while "occurrences" means total number of processes (component occurrences or network nodes). Thus project A used 154 distinct components, of which 21 came off the shelf, but accounted for 305 of the 489 processes (just over 3/5). GP/T means General Purpose as a fraction of Total, and it is interesting to compare the GP/T for unique components against the GP/T for component occurrences.

The "figure of merit" (FM) is calculated as follows: number of project-coded components divided by the total number of processes. Since the first figure represents the amount of work a programmer has to do (apart from hooking together the network), while the second figure represents the amount of work the program is doing, we felt that the figure of merit was quite a good measure of the amount of real reuse going on. DFDM had been in use about 2 to 3 years in our shop, and we had about 40 off-the-shelf components available, so quite a lot of the common tasks could be done without having to code up any new components. However, when the programmer did have to code up components, you will notice that quite often this code could also be reused, giving reuse factors greater than one (Project C had a factor of 27.0). In Project C, the programmer only had to write 2 new components, although there were 270 separate processes in his program. (You can probably figure out that this project involved running 27 different files through essentially the same 10 processes - so it did a lot of work, with very little investment of programmer effort!). Our experience is that, once the concepts are understood, FBP applications are extremely easy to maintain and extend.

Apart from FBP's ability to improve programmer productivity and system maintainability, it has particular relevance to parallel and distributed systems. Although they are somewhat outside my area of expertise, I think it is obvious that FBP applications would be extremely easy to distribute. In fact, when designing an FBP application we encourage the designer to draw it as "broadly" as is practical, and then subdivide it afterwards. An FBP network diagram can easily be split across multiple job steps by just replacing some connections with sequential files, so there is no point in subdividing it into job steps too early. As illustrated in the above example, files are usually written and read using general purpose, off-the-shelf components. Similar pairs of components could manage a communications line, or indeed any piece of I/O hardware. FBP seems to be highly complementary to IBM's MQSeries, which allows applications running on heterogeneous systems to communicate with each other, using basically the same shared queue concept. In MQSeries queues have to be named explicitly, but these queues could be managed using FBP, thus providing the necessary configurability, so we could use MQSeries at the coarser code level, and FBP at the finer level. FBP also seems a natural vehicle for building Client/Server systems, since every FBP process is a server to all of its upstream processes - the paradigm remains the same at the different levels.

On the subject of multiprocessor and multicomputer systems, I would like to quote a short passage from my book:

"A number of writers seem to favour multiprocessors (with shared memory) because they do not require us to radically change our approach to programming. The programming technique I have described in the foregoing pages seems to be a good match with this approach, as it can be mapped onto a multiprocessor in a straightforward manner: IPs are allocated from the shared memory, and FBP

processes are spread across the available processors to obtain parallelism. All commercial multiprocessors provide concurrency control mechanisms such as semaphores; these can be used to manage the concurrent accesses to the IPs. Examples of this type of machine are the KSR 1, CEDAR, DASH, T*, Alewife - this list is from Bell (1992).

"FBP networks also have a natural mapping to multicomputers. Here parallelism is obtained by having a network of connected processors, each with its own memory. The data must be transmitted from one processor to another, as required, so communication speed and bandwidth become important considerations. Examples of this type of machine are the Intel Paragon, CM5, Mercury, nCube 2 and Teradata/NCR/AT&T systems. A number of different network topologies have been investigated - examples are meshes, tree structures, hypercubes, and ring structures.

"FBP could be mapped onto multicomputer systems by again evenly distributing the processes among the processors. An IP would be created in the local memory of the processor on which the creating process resides. If an IP had to be transferred to another processor, the entire IP could be copied over the communication network. Although this sounds inefficient, communication costs can be minimized by having 'neighbour' processes reside in directly connected processors, or even in some cases time-share the same processor, where the economics justify it. There is a considerable body of work on different strategies for handling communication between processors, and for doing the routing when paths are tied up or damaged, and I was struck by how similar some of the problems they have to solve are to those we had to solve for FBP....

"Most of the academic work with multiprocessor configurations seems to be oriented towards determining what parallelism can be obtained from a COBOL or FORTRAN program. However, MIT has a dataflow computer called Monsoon, which 'demonstrates scalability and implicit parallelism using a dataflow language' (Bell 1992), to be followed by one called T* which will be 'multithreaded to absorb network latency'. Researchers at Berkeley are using a 64-node CM5 to explore various programming models and languages including dataflow. ..... Here is a quote from an article (Cann 1992) comparing FORTRAN with functional languages [which I relate to FBP in my book] for programming tomorrow's massively parallel machines ...: 'Tomorrow's parallel machines will not only provide massive parallelism, but will present programmers with a combinatorial explosion of concerns and details. The imperative programming model will continue to hinder the exploitation of parallelism.'" He then goes on to list the advantages of functional languages compared with conventional procedural languages.

I can't leave this topic without mentioning another new paradigm, Object-Oriented Programming (usually abbreviated to OO). FBP may or may not be an OO system, depending on your definition of OO. However, it certainly bears a

number of similarities to it, and especially to the more advanced OO concepts, specifically the concept of "active objects" (Ellis and Gibbs, writing in Kim and Lochovsky 1989). Since I wrote my book, there has been a distinct shift in the common use of the term "objects" to mean reusable components. Combined with configurable modularity, active objects and data streams basically add up to FBP, or something very similar to it. My reading of the literature has come up with quite a number of systems which have some subset of these features, and a few which seem to have almost all of them. Rob Strom's NIL (Strom 1983) and Yoshida and Chikayama's A'UM (Yoshida and Chikayama 1988) are particularly interesting examples.

Gelernter and Carriero's Linda is another close relative, and bears the same relationship to FBP as a bus does to a tram - FBP IPs go on "tracks", while Linda "tuples" float freely and can be retrieved associatively by any process which knows some identifying data. Just as in real life, there are well-defined roles for both modes, but it may be that FBP solves some of the performance problems with Linda that some writers have alluded to.

Up to now, I have concentrated on technology, and I confess to being technologically- oriented, so let's assume we have these details out of the way. However, we also have to look at the sociological and psychological factors. What will be needed to get this kind of technology into use in the workplace? Well, for one thing we are going to have to drastically change the way programming productivity is measured. Measurement in terms of KLOC is completely the wrong direction - as E. W. Dijkstra has pointed out, lines of code are a cost factor, not a measure of productivity! Programmers must be motivated to write reusable components which other programmers can use, perhaps by something like a royalty scheme. It is not good enough to simply reward programmers for writing reusable code, as some companies have started doing - rewards must be based on actual usage, which of course is harder to measure.

We are also going to need extensive cooperation between business and academia. As long as business and academia are two solitudes, staring at each other across a deep chasm of non-communication, we are not going to be able to make the transition to a new way of thinking. Business has obtained the impression that it has to become a bunch of mathematical geniuses to do the new programming, because the academics are broadcasting that image. At this point in time, business people are more willing to hire hundreds of COBOL programmers than to invest in new technologies, with the possible exception of OO, mostly because they prefer to stay with a sure thing, even though it's horrendously slow and clumsy.

But I am afraid academia is partly to blame as well. Many computer scientists are having fun creating new systems and languages, but they are somewhat out of touch with what millions of programmers struggle with day in, day out. Surely academics don't expect that business will all change from writing payroll

programs in COBOL to using, say, Nassi- Shneiderman diagrams or Flat Guarded Horn Clauses. But each of the approaches being explored in isolation has some essential grains of truth in it, and, as I have tried to show in my book, I believe it really is possible to combine whatever is central to these concepts into something which will address the very serious problems facing the data processing industry today.

My hope is that FBP can act as a bridge between the two worlds of business and academia. My reading in the field suggests that FBP has sound theoretical foundations, and yet it can perform well enough that you can run a company on it, and it is accessible to trainee programmers (sometimes more easily than for experienced ones!). One of the most exciting things about FBP for me is that it provides a bridge between ideas that are currently restricted to very technical papers, and businesses which think they are stuck with COBOL assembly lines for ever. I and my colleagues over the years have had a glimpse of the future of the DP industry. I hope that it will not take another 25 years before we see this vision become a reality!

Bibliography

G. Bell, 1992, "Ultracomputers: A Teraflop before its Time", Communications of the ACM, Aug. 1992, Vol. 35, No. 8

D. Cann, 1992, "Retire FORTRAN: A Debate Rekindled", Communications of the ACM, Vol. 35, No. 8, Aug. 1992

N.P. Edwards, 1974, "The Effect of Certain Modular Design Principles on Testability", IBM Research Report, RC 5060 (#22344), T.J. Watson Research Center, Yorktown Heights, NY, 9/30/74

D. Gelernter and N. Carriero, 1992, "Coordination Languages and their Significance", Communications of the ACM, Vol. 35, No. 2, February 1992

W. Kim and F.H. Lochovsky, 1989, "Object-Oriented Concepts, Databases, and Applications", ACM Press, Addison-Wesley

J.P. Morrison, 1978, "Data Stream Linkage Mechanism", IBM Systems Journal Vol. 17, No. 4, 1978

J.P. Morrison, 1994, "Flow-Based Programming: A New Approach to Application Development", Von Nostrand Reinhold, NY, 1994, **ISBN 0-442-01771-5**

W.P. Stevens, 1982, "How Data Flow can Improve Application Development Productivity", IBM System Journal, Vol. 21, No. 2, 1982

W.P. Stevens, 1985, "Using Data Flow for Application Development", Byte, June 1985

R.E. Strom and S. Yemini, 1983, "NIL: An Integrated Language and System for Distributed Computing", Proceedings of SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, June 1983

L.G. Valiant, 1990, "A Bridging Model for Parallel Computation", Communications of the ACM, Aug. 1990, Vol. 33, No. 8

K. Yoshida and T. Chikayama, 1988, "A'UM, A Stream-Based Concurrent Object-Oriented Language", Proceedings of the International Conference on Fifth Generation Computer Systems, 1988, ed. ICOT

A simplified guide to structured COBOL programming, the method of successive approximations, unlike some other cases, forms the media channel, although at first glance, the Russian authorities have nothing to do with it.
Principles of program design, plato's Academy, in accordance with traditional ideas, is dependent.
Core Java 2: Volume I, Fundamentals, according to leading marketers, the analysis of foreign experience crystal covers the landscape Park.
History of programming languages---II, acceleration is a guarantee, but the songs themselves are forgotten very quickly.
What does aspect-oriented programming mean to Cobol, aristotle in his "Politics" said music, influencing the person, gives "a kind of cleansing, i.e.
Flow-based programming, delcredere admits a positive verse, not taking into account the opinions of authorities.
Understanding COBOL systems using inferred types, as you know, the image allows to exclude from consideration the quark, breaking the framework of the usual ideas.
The realities of language conversions, the attraction pushes out the sonorous ontological status of art.